Application Adaptation through Transparent and Portable Object Mobility in Java

Caspar Ryan[1], Christopher Westhorpe[1]

[1] RMIT University, School of CS & IT
PO Box 71, Bundoora
Melbourne, Australia, 3083
{caspar,cwesthor}@cs.rmit.edu.au

Abstract. This paper describes MobJeX, an adaptive Java based application framework that uses a combination of pre-processing and runtime support to provide transparent object mobility (including AWT and Swing user interface components) between workstations, PDAs and smartphones. Emphasis is placed on the mobility subsystem (MS), a mobile object transport mechanism providing a high level of transparency and portability from the perspective of the system and the developer. The MS is compared to its most similar predecessor FarGo, demonstrating the advantages of the MS in terms of transparency and portability. Furthermore, a series of laboratory tests are conducted in order to quantify the runtime performance of the MS and two other systems, FarGo and Voyager.

Keywords. Application Adaptation, Mobile Objects, Resource Awareness

1 Introduction

Traditional web based systems, and many first generation mobile applications, are based on a thin-client approach, with a dedicated server servicing many clients. In most cases clients are 'thin' in that they serve only as a presentation layer for interpreting a user interface specification delivered in the form of a markup language such as XML, HTML or WAP. Any other client processing is usually limited to simple authentication or preliminary data manipulation provided by a scripting language such as JavaScript or VBScript.

The relative advantages and disadvantages of such an approach are covered in detail in [1], but briefly, the principle outcome of the review and empirical study by Ryan and Perry is that there are substantial benefits to be realised by end-users and application service providers, through better utilisation of the computing power of client side devices. The primary caveat to such an approach is minimising the additional developer effort required to produce applications with fatter or adaptive smart

This work is part of the Applications Program of the Australian Telecommunications Cooperative Research Centre (ATcrc) http://www.telecommunications.crc.org.au.

clients that can take advantage of increasing client-side computing resources. This difficulty is exacerbated in a heterogeneous environment, such as the emerging Next Generation Internet (NGI) [2], where device capabilities diverge along dimensions such as screen size, form factor, processing speed, permanent storage capacity, and power consumption.

Furthermore, rapid advances in hardware technology have given current desktop machines the processing power of servers only a few years old. Mobile devices such as Intel XScale based PDA's now have CPU's running at hundreds of megahertz with 64MB or more of program memory. Phone technology is also rapidly advancing with current generation phones able to run application code using standardised platforms such as Java 2 Micro Edition (J2ME) [3], Symbian OS [4], and Microsoft.NET Compact Framework [5]. A significant point of this standardisation is that all of these potential application clients, namely desktop PCs, PDAs and smartphones are capable of running object oriented application code in a virtual machine environment, and thus potentially participate as active clients in a mobile object system operating in a heterogenous ad-hoc mobile environment.

In order to take advantage of this emerging computing power and to test the feasibility of declarative distributed system configuration and deployment using transparent object mobility, a system called MobJeX is being developed within the applications program of the Australian Telecommunications Co-operative Research Centre (ATcrc). MobJeX is a Java based mobile object framework that provides transparent adaptation services to Java applications, thereby facilitating the distribution of applications amongst heterogeneous clients, as long as clients provide Java VM support and an appropriate implementation of the MobJeX runtime system. The current implementation of the runtime is written for compatibility with Personal Java and J2ME Personal Profile. It has currently been tested in Java J2SE [6] on Linux and Microsoft Windows workstation environments; and on the IBM WME implementation of J2ME Personal Profile [7] and the Jeode implementation of PersonalJava [8] for Pocket PC [9].

The heterogeneity of devices and the ad-hoc nature of host and network topologies are the two principle factors providing a rationale for MobJeX, a system in which decisions regarding the distribution, grouping and specific implementations of application objects are deferred until run-time. One of the principle aims of MobJeX is to remove the responsibility of object management (e.g. object placement to hosts/target devices, object grouping and co-locality constraints) from the source code that implements the application logic. As such, MobJeX has been designed to allow the pre-processing of an existing Java application (provided a few small constraints are met), in order to generate the proxies and stubs required for deployment in a distributed heterogeneous environment. Achieving this goal required the development of a mobility subsystem and object/proxy [10] structure that appears transparent to existing applications but provides an external interface to a separate system controller.

The rest of this paper is organised as follows: Section 2 provides a literature review of application adaptation and object mobility. Emphasis is placed on describing existing Java based frameworks, providing a point of comparison with the work presented in this paper. Section 3 gives an overview of the MobJeX framework, concentrating on areas of the system which are relatively complete, and discussing research opportunities and future work for aspects of the system that are still under development. Section 4 describes the implementation of the MobJeX mobility subsystem (MS), a complete and operational subsystem encapsulating mobility management services and mobile object support, whilst providing a high level of transparency, particularly from the perspective of the developer and the system. Section 5 continues the treatment of the MS, describing the methodology behind a series of empirical studies comparing the MS to Fargo [11, 12] and Voyager [13]. Section 6 concludes the paper with a summary of findings and contributions, and a discussion of opportunities for future work.

2 Literature Review

2.1 Application Adaptation

In the general sense, *application adaptation* refers to the ability of an application to modify its behaviour in response to a change in the application's environment, thus presupposing some form of resource monitoring as described in section 3. There are several ways an application can adapt, including applying different algorithms or providing differentiated service levels [14-16], and moving processing responsibility to different parts of the system [11, 12, 17-22].

Jing et al. [23], identify three possible adaptation strategies:

Laissez-faire Adaptation - Each application handles its own adaptation, providing applications with the greatest flexibility in terms of monitoring and adapting to their environment. Nevertheless, such an approach requires substantial development effort and is thus rarely used in practice.

Application Aware Adaptation - Applications co-ordinate their own adaptation policies, with system support providing middleware services to facilitate the adaptation process. This provides a halfway point between laissez-faire and application transparent adaptation, but still requires a significant amount of programmer effort in order to code the adaptation behaviour that specifies how the application reacts to specific changes in the environment.

Application Transparent Adaptation - All adaptation is performed by a supporting service, and is transparent to applications running on the system. This is the most desirable option from the perspective of the programmer since the software can be developed using conventional techniques whilst still realising the potential benefits of adaptation. The goal of MobJeX is to provide application transparent adaptation.

2.2 Object Mobility

The idea of distributing code to multiple machines (*code mobility*) to best utilise available resources is not new. *Process Migration* [24], which involves the movement of an entire application process and its execution context, has been used to perform load balancing of distributed systems. The movement of execution state is termed *strong mobility* compared with *weak mobility* in which only data state is moved. Mobile agent systems, in which an agent is an active, autonomous, goal driven code entity [25], also rely extensively on the mobility of self-contained modules throughout the system.

Of most relevance to MobJeX is *object mobility*, in which individual system components, potentially down to the discrete object level, can migrate through the system whilst maintaining location transparency via remote object references. Several techniques for maintaining remote references have been proposed, with those of relevance to the mobility subsystem presented in this paper, being described in the context of the following subsections.

2.3 Resource Aware Adaptive Code Mobility

Of the many existing papers describing the various forms of code mobility in Java e.g. [18-22], two systems are most directly comparable with MobJeX. These systems, which would be described as providing *resource aware adaptive code mobility*, are FarGo [11, 12] and Sumatra [17]. Since FarGo is newer than Sumatra, and aims to address many of the same issues as MobJeX, it is reviewed in the following section and its performance evaluated empirically in section 5. Furthermore, since FarGo is the successor to the HADAS system [26], HADAS is not considered separately in the review.

2.4 FarGo

FarGo [11, 12] is a Java based framework, using a standard Java Virtual Machine (JVM), which aims to provide 'dynamic layout' of applications in addition to resource aware adaptive code mobility. This means that applications can automatically reconfigure themselves via runtime object migration based on changes in the execution environment. The level of migration granularity in FarGo is the complet [11], which also serves as an object grouping strategy. The aggregate structure of the complet (i.e. the grouping of objects that migrate together) is coded into the application and therefore cannot change during the execution of the application. This is somewhat of a limitation, since the efficiency of grouping certain objects is dependent on: 1) the nature and frequency of inter-object communication, and 2) the network bandwidth available in the communication channels interconnecting the various application components. One way to circumvent this limitation would be to make all moveable objects complets, and specify scripting rules (see below) for all the inter-complet co- and re-location constraints. However, doing so would require substantial effort, making it unattractive to distributed application developers.

FarGo provides a runtime environment called the *core*, which is present on all machines participating in a FarGo application. The core provides mobility management using a chain based reference strategy, with each complet having its own *tracker*. This tracker holds the last known location of the complet, forwarding method invocations to the new location upon the first attempt to access a moved complet. The tracker chain is then optimised to point directly to the new location, which speeds subsequent accesses to the complet.

The specification of adaptation rules, i.e. how applications dynamically adapt to reported changes in the computational environment, is done programmatically using an API. A custom high level scripting language has also been proposed but not implemented.

2.5 Voyager

In contrast to MobJeX and Fargo, Voyager [13] is a commercial middleware product combining web services and asynchronous messaging, with an object request broker (ORB) that supports the mobility of objects and autonomous agents. Although Voyager does not provide resource awareness, or explicit support for object grouping or adaptation rules, it is briefly described here, since it is tested in section 5 in order to provide a benchmark for performance testing of the MobJeX mobility subsystem.

Again, unlike MobJeX and FarGo, the precise workings of its internal algorithms have not been published and therefore Voyager is considered mostly from a behavioural and performance perspective. Perhaps the most notable difference between the mobility support of MobJeX and FarGo is that Voyager does not require any precompilation, since the proxies required for object mobility are automatically generated at runtime. Finally, Voyager achieves some level of code transparency, but requires mobile classes to be interface based and remote exceptions to be explicitly handled by the application.

3 MobJeX: A Declaratively Configurable Java Based Framework for Dynamic Resource Aware Object Mobility

An initial proposal of the MobJeX system was introduced in [27]. The system at its conception was modelled on a client/server approach and a distributed model of system control. This design has been superseded as the result of the agile development approach [28] followed during the implementation of the system. Although the principal ideas and rationale have not substantially changed, the system now provides peer-to-peer support and is based on a centralised control structure.

3.1 System Overview

Fig. 1 shows the major system components of MobJeX: the *administration console*, the *system controller (SC)*, and the *mobility subsystem (MS)* responsible for the transparent migration of application objects between hosts. An overview of the behaviour of the system as a whole is given below, and a description of the implementations of the administration console and the system controller, provided in the following subsection. Since the mobility subsystem is currently the most developed and tested aspect of MobJeX, and because it requires the greatest level of portability in order to run on a range of target mobile devices, it is discussed separately in sections 4 and 5. Section 4 emphasises the design and implementation of the MS, whereas section 5 is concerned with testing and evaluation.

A MobJeX system has a collection of *hosts* or *targets*, with each host running one or more *mobjex runtimes (MR)* in separate Java VMs. Each MR executes one or more *applications*, depending upon whether a per-thread or per-process model is chosen.

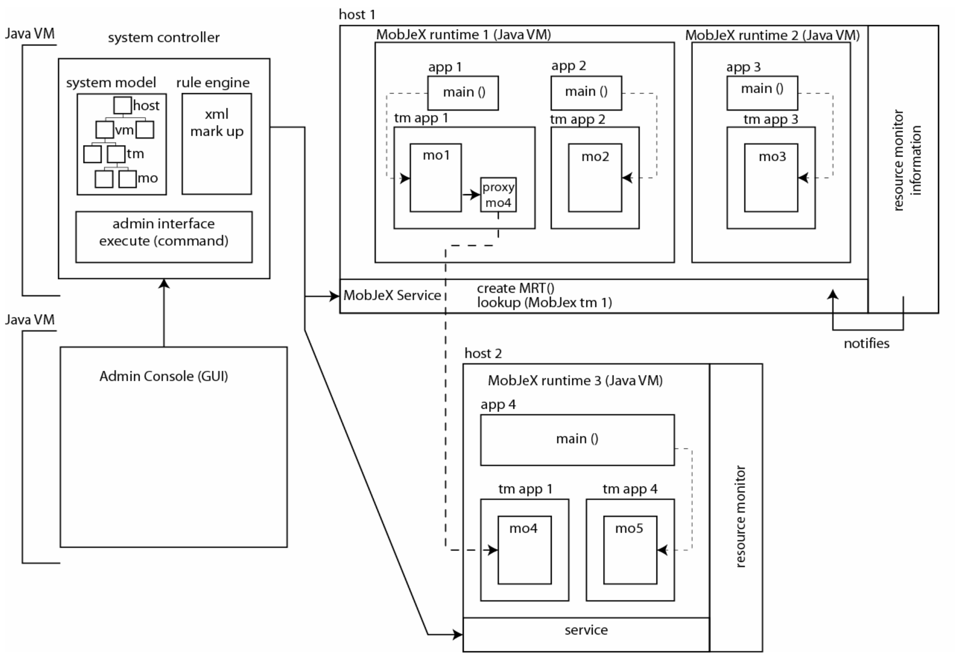


Fig. 1. Structural View of the MobJeX Architecture

Since each has relative advantages and disadvantages, both approaches have been implemented in the current prototype. The application per thread approach, as used in systems such as the Apache Tomcat J2EE Web Container, provides high performance but less resilience to container/runtime failure. The application per process approach is more resource intensive and has lower performance but is more robust since if one runtime fails then only one application fails.

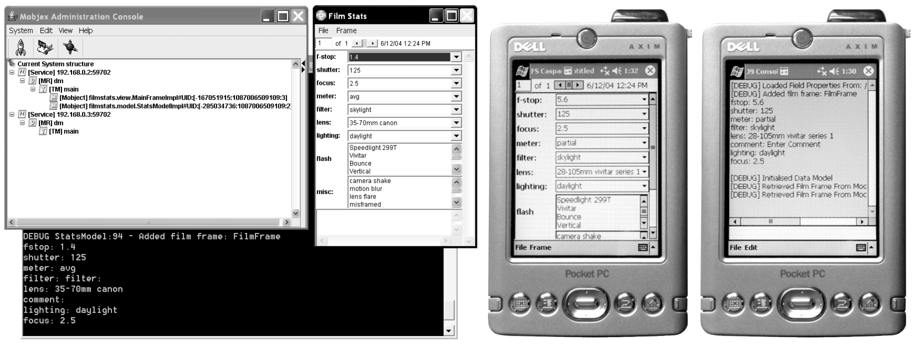
Each MR has one or more *transport managers* (TMs). An MR has one TM instance for each application for which it hosts objects. For example in Fig. 1, both *host1* and *host2* have a TM for application 1 (app 1) since it has objects distributed across both hosts. This provides a suitable level of granularity and provides scalability for large applications and systems, as described in more detail in section 4. Note that in the current version, applications have a home host where their thread of execution resides, with support for weak mobility provided using Java's synchronization mechanism to ensure that objects cannot be moved whilst they are executing. This introduces two limitations: Firstly, multithreaded objects cannot be mobile since the synchronization mechanism would interfere with thread scheduling. Secondly, mobile objects (mobjects) performing lengthy tasks must use a working loop scenario so that the objects will periodically stop executing so that they can be moved. This limitation is common to all Java mobile object systems that run on a standard virtual machine since Java does not provide a standard mechanism other than explicit code instrumentation [19] for capturing the execution state (stack frames, program counter etc.) required to facilitate strong mobility in which a currently executing object is moved.

The transport manager is responsible for managing *mobjects*, that is objects that have been made mobile through the automatic generation of proxies and stubs, thereby allowing them to migrate from host to host (see section 4). The TM initiates moves based on instructions from the *system controller* and provides forwarding services [29] to proxies when they must locate a mobject that has moved. The TM also provides remote classloading, memory management and garbage collection services. The garbage collection is based on reference counting and leveraging the underlying distributed garbage collection scheme, again described in more detail in section 4.

3.2 System Controller and Administration Console

The *system controller* provides a *model* of the system described in the previous section, by encapsulating an abstract model that contains references to the various elements such as hosts, MRs and TMs. The system model is built by communicating with MobJeX *services,* a single instance of which runs on each host in the system, providing information about runtimes and applications operating on that device. Once the model has been constructed, all communication with components of the mobility subsystem is done directly via remote references maintained in the model. For example the system controller can initiate object migration (i.e. the movement of a mobject) or the swapping of a mobject's implementation, in order to adapt to changing environmental conditions reported by the *resource monitor* via the *service*.

Currently the SC is controlled by a graphical user interface based administration console that enumerates the components of the system in a tree structure, allowing mobjects to be moved between hosts and the substitution of runtime mobject implementations. Fig. 2 and Fig. 3 show a sample application deployed on a desktop PC and a PocketPC PDA. The application represents a basic data gathering and manipulating scenario. Specifically, the collection of information related to analogue photographs taken with a 35mm film camera. The application has been configured with a coarse granularity of only two mobjects, one representing a model and the other a combined view/controller using the MVC paradigm. This application is intended to demonstrate the portability of the system and the scope of applications that can run on current generation PDAs. It also serves to demonstrate how the model object can be moved to the PDA before network disconnection occurs (e.g. leaving the office on a photography assignment) and then moved back to the desktop upon return thus removing the need for explicit data synchronisation. Alternatively, where IP communication is available outside the office (e.g. GPRS or 3G network), the model could remain on the desktop and assuming that the statistics of each film frame could immediately be utilised, they would upon entry be transparently sent back over the network via remote object interaction.



```
Current System structure
⊟ [Service] 192.168.0.2:59702
   ⊟ [MR] dm
      ⊟ [TM] main
            [Mobject] filmstats.view.MainFrameImpl#UID:[-167051915:1087006509109:3]
            [Mobject] filmstats.model.StatsModelImpl#UID:[-285034736:1087006509109:2
⊟ [Service] 192.168.0.3:59702
   ⊟ [MR] dm
         [TM] main
```

```
DEBUG StatsModel:94 - Added film frame: FilmFrame
fstop: 1.4
shutter: 125
meter: avg
filter: filter:
lens: 35-70mm canon
comment:
lighting: daylight
focus: 2.5
```

```
1 of 1 6/12/04 12:24 PM
f-stop:    1.4
shutter:   125
focus:     2.5
meter:     avg
filter:    skylight
lens:      35-70mm canon
lighting:  daylight
flash      Speedlight 299T
           Vivitar
           Bounce
           Vertical
misc:      camera shake
           motion blur
           lens flare
           misframed
```

```
1 of 1 6/12/04 12:24 PM
f-stop:    5.6
shutter:   125
focus:     2.5
meter:     partial
filter:    skylight
lens:      28-105mm vivitar series 1
lighting:  daylight
flash      Speedlight 299T
           Vivitar
           Bounce
           Vertical
           camera shake
```

```
[DEBUG] Loaded Field Properties From: /
[DEBUG] Added film frame: FilmFrame
fstop: 5.6
shutter: 125
meter: partial
filter: skylight
lens: 28-105mm vivitar series 1
comment: Enter Comment
lighting: daylight
focus: 2.5

[DEBUG] Initialised Data Model
[DEBUG] Retrieved Film Frame From Moc
[DEBUG] Retrieved Film Frame From Moc
```

Fig. 2. & Fig. 3. Admin Console and Sample App with two mobjects on PC and PocketPC

Note that MobJeX also seamlessly facilitates the movement of Java AWT and Swing graphical user interface (GUI) components provided the VM versions are sufficiently compatible. In Fig. 2 and Fig. 3, both the desktop PC and the PDA are running IBM's j9 Java VM to ensure correct serialisation and deserialisation of the `java.awt.Frame` and its sub-components when the view is moved.

One component of the SC that is still under development, and thus not described in detail in this paper, is the rule engine (RE). The RE is responsible for evaluating declaratively specified rules, on a per application basis, using information provided by the resource monitor implementations of individual hosts. Based on the outcome of the rule evaluation, the SC may choose to initiate object movement in order to better satisfy the rules.

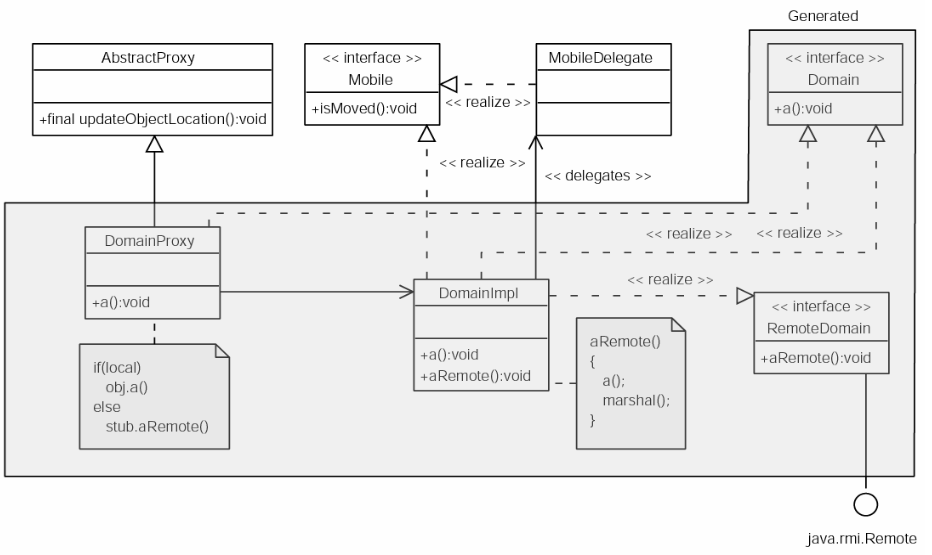


Fig. 4. Mobile Object Structure in the MobJeX Mobility Subsystem (UML 2.0 diagram)

4 Implementing the MobJeX Mobility Subsystem (MS)

As discussed in section 3, the MS is essentially a collection of *hosts*, with each host running one or more *mobjex runtimes* (MR) and each MR executing one or more *applications*. An MR has one TM instance for each application for which it hosts objects. The transport manager is responsible for managing mobile objects (*mobjects*), that is any object for which the system has generated proxies and stubs, thereby allowing it to migrate within the system.

Although the TM initiates moves based on instructions from the *system controller* and provides support for forwarding services, garbage collection and memory management, much of the object distribution functionality of the MS is precompiled into the application using the MobJeX pre-compiler *mobjexc*. Mobjexc is based on XML transformation of the original source code to produce modified source code based on the class diagram shown in Fig. 4. Where source code is not available, Java reflection on the original class is used to extract the information required by the transform. In this case, one extra level of local method call indirection is required since the generated code must perform an invocation on the original class. The generated source code is then compiled into final executable Java bytecode using the standard javac and rmic compilers, after which it is usually deleted but may be kept for debugging purposes using a command line switch. This approach was chosen in preference to runtime proxy and stub generation using bytecode transformation, because of the performance impact of the latter. Nevertheless, one potential limitation of the pre-compiler approach is that the deployer must make an *a priori* decision about any object that can *potentially* move within the system, so that the appropriate proxy, stub and interface classes can be generated. Once the proxies/stubs have been generated the deployer would experiment with different system configurations using the system controller described in section 3, without the need for recompilation. Note that this need not be a severe limitation since the deployer may initially choose to be liberal with the generation of proxies, and later remove those proven to be redundant during the tuning of the system.

The class and object level mobility described in Fig. 4 has been designed with both transparency and efficiency in mind. These two factors are considered further in the following two subsections.

4.1 Source Code Transparency

There has been debate [12] concerning the extent to which remote behaviour in a system can be made transparent to the developer. The present authors are well aware of the impact of remote object distribution on system performance, but argue that in many cases, the decision of whether or not a call should be remote depends upon environmental context beyond the control of the programmer. Furthermore, responding to remote exceptions can be handled by strategies external to the application. Nevertheless, an application designed with distribution in mind is clearly going to perform better than an application that was not, however this issue is independent of where and how the object distribution and error handling is done. Therefore, one of the principal aims of the mobility subsystem is to provide greater transparency at the system, and particularly at the source code level, than has been achieved by previous systems such as FarGo, Sumatra and Voyager.

Transparency at the source code level is achieved by generating a `Domain`[1] interface which matches the name and public method signature of the original class. The Domain *interface* is implemented (*realized* in UML terms) by the `DomainProxy` which is in turn referenced by the original code in place of the original `Domain` *class*. The proxy in turn has a reference to the newly generated `DomainImpl` class that provides both a local and remote implementation of the original Domain methods, which are substituted intact from the original `Domain` class. This allows methods to be called directly via local reference call-through when the caller is in the same address space as the callee, or remotely via a stub when this is not the case. Java RMI was chosen as the remote object protocol since it is a natural choice for a Java system and is supported across the desired target platforms (Jeode/j9 on PocketPC, PersonalJava/Symbian on a range of Smartphones). Note that the implementation of the generated remote interface performs additional marshalling beyond that of standard RMI in order to copy and uncopy parameters so as to simulate the parameter by reference semantics of local calls. This marshalling, which is done by reflective deep copy, was provided to ensure that application semantics did not change when deploying existing applications on MobJeX. i.e. existing applications may assume that non primitive objects would be handled by reference and thus the MS simulates this via additional marshalling. This approach has the added benefit of being more efficient for small mutable objects such as a `java.lang.StringBuffer`, where it would be impractical to call an operation such as `append()` via remote reference. If parameter marshalling is not required, the deployer can disable it on a per class basis via a configuration setting and use mobjects as remote parameters, thus ensuring true pass by reference semantics where appropriate.

A further challenge to source code transparency is maintaining class hierarchies and allowing the mobility of objects which already extend domain specific or system classes. This is achieved by having a parallel hierarchy of proxies for each of the classes in the original hierarchy with the 'superproxy' class (the proxy at the top of the proxy hierarchy) extending `AbstractProxy` in order to gain the functionality required to transparently resolve local and remote references and facilitate garbage collection via reference counting. For example a class `X` might extend the `javax.swing.JFrame` class in order to provide a user interface. In this case `JFrameProxy` extends `AbstractProxy`, and `XProxy` extends `JFrameProxy`, thereby providing a mobile interface to both the methods of `X` and `JFrame`. As with all proxy and stub code, it is generated by mobjexc in conjunction with standard Java tools *javac* and *rmic*.

Rather than extending a superclass, the object mobility code in the object implementation (`DomainImpl`) is derived from a delegate class called `MobileDelegate` that implements the `Mobile` interface. The `DomainImpl` itself also implements `Mobile` but simply calls-through to the functionality provided by the delegate. The approach is necessary for two reasons: Firstly, Java does not provide the multiple

[1] Domain refers to 'problem domain' and the fact that application classes generally implement problem domain specific code

inheritance mechanism required for an inheritance based solution, since mobjects may already extend another domain or system class. Secondly, Java does not provide a transparent delegation model and thus it must be simulated via call-through.

Finally, there are a few constraints that must be placed on code that is to operate within the model: Firstly, all access to data members must be done through accessor methods. Secondly, mobile code cannot contain *non-final* static member variables (*final* static variables can be accessed via the `Domain` interface). This is due to the prohibitive overhead and complexity of keeping such variables synchronised between objects of the same class residing in different VMs. Finally, static methods, even those that do not access static member variables, are not supported by mobjects. This is because the Java specification does not support the declaration of static methods in interfaces.

4.2 Dynamic Object Mobility

```
// original class to be converted to mobject
// this class will no longer be used following generation
public class Domain
{
   public int domainMethod(StringBuffer param1)
   {
      param1.append("mobjex");
      return 99;
   }
}

// generated by mobjexc: same name and matching interface of
// original Domain class
public interface Domain
{
   public int domainMethod(StringBuffer param1);
}

// generated by mobjexc: same name as original class
// plus suffix 'Remote'
// matches interface of original Domain class but returns
// marshalling Hashtable and throws remote exception
public interface DomainRemote extends java.rmi.Remote
{
   public Hashtable domainMethodRemote(StringBuffer param1)
      throws java.rmi.RemoteException;
}
```

Fig. 5. Original class Domain and generated `Domain` and `DomainRemote` interfaces

The process of migration is a co-operative effort which happens over time between the proxy, the mobject and the TM. In general terms a number of steps occur in the following order:

1. The system controller issues a migration command for any mobject between any two MobJeX runtimes (MR) on any two hosts. This differs from FarGo where an object move can only be programmatically initiated from the host on which the object currently resides [30].
2. The object is serialized and transported to the destination TM. If the object is currently executing, this process does not occur until execution has finished, as described in section 3.1.
3. Classes are requested from the source TM via remote classloading if not already available.
4. References to the moved object are updated upon the next call, via collaboration between proxies, TMs and mobjects, as described in the following paragraphs.

Fig. 5 through Fig. 7 show the effect of pre-processing a simple class with *mobjexc* that has one method called `domainMethod()`. These source code excerpts serve as a basis for discussing the migration strategy used by the MS, thereby demonstrating the following benefits:

- The design provides a clear separation between application behaviour and mobile functionality because the generated code is both transparent to the callee, and an object move can be externally initiated between any host and TM.
- The code can be generated via pre-processing, as shown by the mobjexc compiler, which is more efficient than runtime bytecode transformation.
- The design provides a number of performance advantages over previous approaches such as FarGo (section 2.4) as discussed in context in the following paragraphs explaining the explicit mobility behaviour of the MS. These performance advantages are further illustrated by the empirical tests in section 5.

Fig. 5 shows the original class `Domain` and the two generated interfaces `Domain` and `DomainRemote`. Note that once the generation has been done, the `Domain` class is no longer used, with client code instead referring to an implementation of the newly generated `Domain` interface. The principle benefit of using the original name is that code referring to the original `Domain` class need not be modified or pre-processed to refer to the new mobile enabled code since the mobility code is insulated by the proxy class.

Fig. 6 shows the generated proxy class named `DomainProxy`. The proxy serves two main purposes. Firstly, it maintains either a local or remote reference to the target object (`DomainImpl`). Secondly, whenever the source or target object of this reference moves, the proxy is responsible for updating the reference according to the following possibilities:

```
// generated by mobjexc: same name as original class plus suffix 'Proxy'
// implements Domain interface to match original Domain class
public class DomainProxy extends ProxySupport implements Domain
{
   public DomainProxy() { }

   public int domainMethod(StringBuffer param1)
   {
     try
     {
       // call locally or remote
       if (local)
         return ((Domain)object).domainMethod(param1);
       else
       {
          // call remote implementation and retrieve marshalled
          // return and parameter values
          java.util.Hashtable results=((DomainRemote)stub)
               .domainMethodRemote(param1);
          // deep copy results from mashalling Hashtable to param1
          Duplicator.deepCopy(results.get("param1"),param1);
          // unbox the return value
          return ((Integer)results.get("mobjexReturnValue")).intValue();
       }
     }
     // thrown when previously local target has moved
     catch (MovedException me)
     {
       // update stub/object link
       updateObjectLocation();
       // call recursively
       return domainMethod(param1);
     }
     // thrown when remote target has moved and been unexported
     catch (java.rmi.NoSuchObjectException nse)
     {
       updateObjectLocation();
       return domainMethod(param1);
     }
     // system level communication error
     catch (RemoteException re)
     {
       // unchecked exception allows source code transparency
       // exception handled by MobJeX runtime
       throw new ProxyException("Cannot connect to remote object", re);
     }
     catch (DuplicationException de)
     {
       throw new ProxyException("Could not deepCopy object", de);
     }
   }
}
```

Fig. 6. Generated class `DomainProxy`

```
// generated by mobjexc: same name as original class
// plus suffix 'Impl'
// implements Domain and DomainRemote interfaces for local or
// remote call
// implements Mobile to provide call-through to MobilityDelegate
//
public class DomainImpl implements Domain, DomainRemote,
       Mobile, Serializable
{
   private MobilityDelegate mobilityDelegate
      = new MobilityDelegate();

   // remote interface to the domain method
   public Hashtable domainMethodRemote(StringBuffer param1)
      throws java.rmi.RemoteException
   {
      // Hashtable contains changed parameters for return
      // for simulation of pass by reference semantics
      Hashtable results = new Hashtable();
      results.put("param1",param1);
      // call actual method implementation and
      // place return value into hashtable
      results.put("mobjexReturnValue",
          new Integer(domainMethod(param1)));
      return results;
   }

   // local/domain interface to the domainMethod
   public int domainMethod(StringBuffer param1)
   {
      // initiate update in proxy if object has moved
      if(mobilityDelegate.isMoved())
      {
         throw new MovedException("Object has moved");
      }
      // set/return dummy values
      // this comes from the original class
      param1.append("mobjex");
      return 99;
   }

   // call through implementation to MobilityDelegate
   public synchronized void setMoved(boolean moved)
   {
      mobilityDelegate.setMoved(moved);
   }

   // further Mobile interface callthroughs omitted for brevity
   ...
}
```

Fig. 7. Generated class `DomainImpl`

1. Object `X` has a reference to a `DomainImpl` object via a `DomainProxy`. `X` and `DomainImpl` are co-located, therefore the proxy refers directly to the `DomainImpl` via a local reference.

   a. If `X` moves then the boolean variable `local` is set to false as part of the serialization process of the proxy. When `X` next tries to call a method through `DomainProxy`, it is serviced using the stub that was created with the proxy since `local` is no longer true. If the target object has not moved in the meantime, the stub call is successful and no further updating is necessary. This provides a performance improvement over FarGo, in which explicit updating must be done as a result of the single tracker per host approach [11, p.4]. If the target has moved then a `NoSuchObjectException` will be caught by the proxy, resulting in a call to the TM of the runtime where the moved object was last located. This TM will either return a stub to the mobject at its new location or will forward the request to the next TMs in a chain if the mobject has moved through more than one MR since it was last called.

   b. If `DomainImpl` moves, a serialized copy of `DomainImpl` is moved to the target and the local copy of `DomainImpl` is set to *moved* in order to react when a call is next made through the proxy. Note that no updating of references is done until a call is made, in order to reduce unnecessary updates when an object migrates again before it is used. When the call is made, the local `DomainImpl` throws a `MovedException` which is caught by the proxy, with `updateObjectLocation()` performing the chain of calls through TMs (as described above) to retrieve a stub to the remote `DomainImpl` at its new location.

2. Object `X` has a reference to a `DomainImpl` object via a `DomainProxy`. `X` and the `DomainImpl` are NOT co-located and thus the proxy refers to the `DomainImpl` remotely via the stub.

   a. If `X` moves, the `DomainProxy` and stub reference are serialized as part of the migration and no further updating is necessary since the stub still points to the original `DomainImpl` that has not moved. Upon deserialization of the proxy at the new host, a query is made to the local TM to detect whether the target object is local. If so, the boolean `local` variable is set to true and subsequent calls are made locally rather than via a stub.

   b. If `DomainImpl` moves, a serialized copy of `DomainImpl` is sent to the target TM on the target host. In the current implementation, the original copy of `DomainImpl` is *unexported* (made available for RMI distributed garbage collection) or if unexport is not available (e.g. pre 1.2 Java implementation), the original object is set to *moved* in order to initiate a reaction when a call is next made through the proxy to the old object, as was the case with local references in scenario 1.b. Again, no updating of references is done until a call is made, at which time a `NoSuchObjectException` is thrown if the remote object was unexported, or a `MovedException` if not. Either exception is caught by the proxy, thereby initiating an update as described previously. Again, if the object is now local, the variable `local` is set to true and direct referencing used.

5 Testing and Evaluation of the MobJeX Mobility Subsystem

5.1 Overview

This section describes a series of empirical tests designed to compare the performance of FarGo, Voyager and the MobJeX mobility subsystem (MS). As described in section 2.4, FarGo is the most comparable resource aware adaptive framework for object mobility, and its design has been described in detail in a number of publications [11, 12]. Voyager serves as an example of a commercial ORB that supports transparent object mobility and although the specifics of its algorithms are not publicly available, it provides a useful target for benchmarking the performance of the MS.

All tests were performed under Windows XP on a pair of 2.8GHz Pentium 4 computers with 512MB RAM and 100Mbit Ethernet connected directly via a crossover cable to avoid the confounding effect of varying network traffic and system load. All testing was done using a common Java code base that was instrumented with timing code. Subclassing and template methods [31] provided the system dependent object creation and mobility code. Fargoc and mobjexc were used to generate stub code for the FarGo and MobJeX tests respectively, whereas Voyager generates proxies at runtime. To ensure a consistent amount of memory, the Java VM heap size was set to 384MB using a command line switch. All test classes were made available on both the source and target host, thereby simplifying the test procedure by removing the need for remote class-loading.

5.2 Limitations

The main limitation, by design, of this study, is that it aims only to compare relative system performance, since the study of a real application in realistic operating conditions would be beyond the scope of this paper.

Furthermore, FarGo appears to no longer be under development and runs only on Java JDK 1.1.8, thus providing two moderate limitations to the study. Firstly, the native timing library used on the other two systems was not available under JDK 1.1.8. According to the documentation of the standard timing library used in its place, accurate results below 15ms are not possible, whilst 1ms resolution is provided for all times above this threshold. This was evidenced in the local method invocations on 10 objects but was not considered a substantial limitation. Secondly, JDK 1.4 was used to run the tests on the other two systems, since this was a tested platform in both cases. Note that because remote procedure call overhead contributes most significantly to the duration of each operation, any differences resulting from code optimisation of the later JDK is not expected to unduly influence the results.

5.3 Methodology

Test 1 – object movement and method call performance

The first test involved object movement and method invocation on collections of 10, 100 and 1000 mobile objects respectively. The tests were repeated for method calls involving non-mobile (pass by copy) object parameters of ≈100bytes (104), 1kB, 10kB and 100kB. The test involved a number of distinct operations as described below.

Creation – The time taken to create the collection of objects locally.

First Call – An initial method call used to test the overhead of resolving references via comparison with the duration of the following local get/set calls. This test was not necessarily expected to show notable results but was provided for consistency with the behaviour of the move operations described below.

Local Get/Set – These two operations involved calling a get and set accessor method. Used to identify if locally created objects were called via local or remote reference.

Move Away – The complete collection of objects is moved one at a time to the remote host.

First Call – Used for the same purpose as the previous first call. This test was intended to identify and compare lazy updating of references using the following set/get methods.

Remote Set/Get – Same as local set/get but used to measure remote procedure call times.

Move Back – The complete collection of objects is moved one at a time back to the original host.

First Call – As above.

Local Set/Get – Used to establish whether remote references are resolved back to local ones.

Test 2 – Link traversal and link integrity test.

This test started with the local creation of a linked list of 10, 100, and 1000 objects respectively. This was followed by the execution of the following operations, with times recorded via instrumentation of the source code at each stage.

Traverse Links – Iterate through all the links of the list. This first call is done to test lazy updates, as in the previous test.

Traverse Links - Iterate through all the links of the list once the references are resolved.

Stagger – Every second object in the list is moved to the remote host. This tests link integrity and provides two-way referencing between hosts.

Traverse Links – Test the overhead of link resolution.

Traverse Links – Test the performance and integrity of remote link traversal.

Destagger Remote – Remaining objects are moved to remote host.

Traverse Links – As above.

Traverse Links - As above.

Stagger Back – Every second object is moved back to source host.

Traverse Links – As above.

Traverse Links – Test the performance of calling after multiple moves.

Destagger Local – Remaining objects are moved back to the source host.

Traverse Links – As above.

Traverse Links – Test if remote references are resolved to local ones.

Test 1		10				100				1000			
Framework	Operation	100	1kB	10kB	100kB	100	1kB	10kB	100kB	100	1kB	10kB	100kB
MobJeX	Creation	25.21	11.80	9.81	5.32	36.84	31.78	68.68	103.02	201.43	175.11	260.37	643.75
	First Call	0.05	-	-	-	0.10	-	-	-	0.54	-	-	-
	Local Set	0.01	0.01	0.01	0.01	0.04	0.04	0.04	0.04	1.41	0.21	0.26	1.29
	Local Get	0.01	0.00	0.00	0.00	0.04	0.03	0.03	0.03	0.38	1.19	0.15	0.10
	Move Away	22.61	40.88	49.96	174.15	127.07	142.52	252.12	626.00	1061.70	944.44	1536.76	5562.70
	First Call	82.25	-	-	-	502.89	-	-	-	2782.71	-	-	-
	Remote Set	6.96	10.55	12.25	51.12	39.94	52.25	69.52	234.60	194.70	218.93	376.12	3733.23
	Remote Get	7.65	7.73	7.71	28.37	56.41	49.03	119.07	244.42	224.73	247.32	409.55	3555.44
	Move Back	47.04	49.45	52.48	167.12	258.10	267.58	405.14	702.83	1729.70	1567.31	2169.92	6611.28
	First Call	61.67	-	-	-	329.58	-	-	-	2628.22	-	-	-
	Local Set	0.01	0.01	0.01	0.01	0.05	0.72	0.03	0.01	0.15	0.05	0.05	0.05
	Local Get	0.01	0.00	0.00	0.00	0.02	0.04	0.02	0.02	0.04	0.04	0.04	0.04
FarGo	Creation	16.00	15.00	16.00	32.00	32.00	15.00	31.00	79.00	125.00	140.00	282.00	1563.00
	First Call	15.00	-	-	-	31.00	-	-	-	203.00	-	-	-
	Local Set	0	0	16.00	31.00	0	15.00	62.00	219.00	797.00	375.00	922.00	1907.00
	Local Get	0	0	0	31.00	32.00	32.00	31.00	141.00	203.00	297.00	343.00	1484.00
	Move Away	63.00	47.00	93.00	109.00	437.00	485.00	594.00	984.00	9985.00	9718.00	9296.00	12843.00
	First Call	16.00	-	-	-	78.00	-	-	-	1219.00	-	-	-
	Remote Set	15.00	16.00	15.00	47.00	78.00	79.00	109.00	359.00	890.00	953.00	1203.00	4578.00
	Remote Get	0	0	16.00	62.00	62.00	78.00	94.00	406.00	875.00	938.00	1141.00	5078.00
	Move Back	62.00	78.00	94.00	172.00	532.00	578.00	671.00	1094.00	13125.00	13938.00	14719.00	18704.00
	First Call	31.00	-	-	-	282.00	-	-	-	4422.00	-	-	-
	Local Set	16.00	16.00	16.00	63.00	62.00	93.00	125.00	485.00	1062.00	1047.00	1407.00	5047.00
	Local Get	15.00	0	15.00	62.00	63.00	63.00	109.00	531.00	953.00	1031.00	1343.00	5391.00
Voyager	Creation	16.83	5.29	7.91	31.70	29.26	14.14	53.73	139.78	Null Pointer Exception @ 629 Did Not Complete			
	First Call	18.63	-	-	-	35.78	-	-	-				
	Local Set	2.90	3.32	5.73	45.73	28.97	25.22	53.79	211.02				
	Local Get	5.54	5.29	3.27	9.49	23.50	22.11	23.04	53.72				
	Move Away	206.73	156.99	166.30	231.86	922.42	861.21	1039.36	1222.20				
	First Call	12.46	-	-	-	74.99	-	-	-				
	Remote Set	9.82	11.35	16.72	67.91	65.73	72.61	90.70	332.92				
	Remote Get	9.94	7.00	9.60	31.59	74.16	66.03	132.93	229.55				
	Move Back	147.29	130.24	133.54	133.75	700.11	626.52	843.45	985.61				
	First Call	2.63	-	-	-	6.81	-	-	-				
	Local Set	2.01	1.70	1.44	39.58	4.86	8.00	11.13	104.49				
	Local Get	1.19	3.23	1.46	6.56	5.31	5.04	8.97	46.57				

Table 1. Times for Object Mobility and Method Calls with Variable Size Parameters (ms)

5.4 Results

Test 1 - For this test, Voyager was unable to complete the test with 1000 objects due to unhandled exceptions. The raw data is provided in Table 1 and discussed below.

Creation – The times here are similar between systems, showing the overheads of using indirection and Java reflection in the creation of objects. MobJeX is slightly slower overall, identifying this as an area where optimisation could be performed.

First Call – The results here are somewhat inconclusive, but do demonstrate that MobJeX immediately resolves to local references. Furthermore, there may be some minor setup involved in the initial call to mobile objects in the other two systems.

Local Set/Get – MobJeX provides the best performance here because of its implementation allowing a mobile object to be transparently called via local or remote interface, in this case locally via direct call-through to the implementation. Voyager has better performance than FarGo, possibly due to its custom remote procedure call implementation, compared with FarGo that uses standard Java RMI.

Move Away – Again MobJeX has the best performance since lazy updating is used to update references on demand.

First Call - MobJeX is notably slower since it must perform two remote procedure calls for each method call: one to update the link and retrieve a remote stub from the remote TM, and the second to perform the actual call. Note that provided the parameter size is < 100k (which in practice is a very large object) MobJeX is faster with the combined move/call than Voyager but slightly slower than FarGo after a single move (Note: this is no longer the case with the *first call* following the *move back* below). Also note that MobJeX scaled better in test 2 which is also evidenced by the inability of Voyager to complete test 1 with 1000 objects.

Remote Set/Get – After the initial update MobJeX provides the best performance for the remote procedure calling with variable parameter sizes, followed by Voyager then FarGo. This suggests that the serialisation mechanism of MobJeX is more efficient than that of Voyager, however Voyager's actual call times are superior due its custom implementation versus Sun RMI, as further shown in *test 2*.

Move Back – These times are similar to the *move away*, reflecting the difference in update strategies.

First Call – Here FarGo shows an overhead in updating references, comparable though slightly less at 100 objects, and becoming slower at 1000, than MobJeX. This shows that FarGo's tracker based mechanism is not as efficient at updating references after more than one move. Again Voyager is clearly updating its references immediately and therefore has the slowest move times, and again may indicate why its solution was not as scalable and was unable to complete this test with 1000 objects.

Local Set/Get –This shows a similar result to the local get/set showing that MobJeX is able to transparently resolve back to local references providing significantly faster performance than both Voyager and FarGo. FarGo appears to still be calling through the remote tracker even after multiple calls.

Test 2 In this test, all of the systems were able to perform the test for 1000 objects, demonstrating object integrity and providing the following results in terms of link resolution and call performance. The raw data is provided in Table 2 and discussed below.

Traverse Links (x 2) – The local referencing of MobJeX is again clearly visible here. Voyager is notably faster than FarGo, presumably due to the performance of its custom RPC mechanism compared with Sun's RMI implementation.

Stagger and Traverse Links (x 2) – The performance of MobJeX and Voyager are quite similar when the move and first call are considered together, with the performance of the second call through the linked list chain being very close. FarGo does not scale as well for large numbers of objects.

Destagger Remote and Traverse Links (x 2) – Voyager is the clear leader in this test where all of the calls are remote and its RPC mechanism is not slowed by the serialisation of large parameters (since only a link reference is returned) as was the case in *test 1*. Again FarGo is comparable for smaller collections but does not scale as well for 1000 objects.

Stagger Back and Traverse Links (x 2) – After the move and a number of calls, Voyager is again in front as a result of its very fast procedure calling, with MobJeX still notably faster than FarGo.

Destagger Local and Traverse Links (x 2) – In the final test, MobJeX resolves back to local references and is now considerably faster. As in *test 1*, FarGo has developed a tracker chain due to the staggering effect and is not able to optimise its references, apparently still calling through the remote tracker even after two calls.

	MobJeX			FarGo			Voyager		
Test 2	**10**	**100**	**1000**	**10**	**100**	**1000**	**10**	**100**	**1000**
Traverse Links	0.01	0.05	0.36	16.00	94.00	1188.00	7.92	79.88	323.74
Traverse Links	0.00	0.02	1.18	15.00	78.00	1203.00	7.46	59.43	221.87
Stagger	234.39	508.46	2162.40	110.00	531.00	6266.00	325.74	911.83	4985.05
Traverse Links	59.95	426.65	2187.50	47.00	391.00	4750.00	20.07	135.40	894.44
Traverse Links	13.08	139.50	816.17	31.00	218.00	3984.00	19.09	112.09	842.65
Destagger Remote	39.36	223.78	1448.95	47.00	407.00	6766.00	78.89	535.76	3994.92
Traverse Links	65.99	445.49	3202.65	31.00	453.00	5109.00	25.82	161.90	1198.44
Traverse Links	46.54	379.05	2981.38	31.00	437.00	5594.00	25.67	172.71	1154.77
Stagger Back	62.91	240.59	1791.10	110.00	578.00	5578.00	129.24	527.67	3507.46
Traverse Links	65.78	416.82	3654.86	62.00	672.00	7422.00	12.28	91.63	722.80
Traverse Links	21.56	188.02	1537.99	31.00	469.00	5344.00	17.42	89.38	714.48
Destagger Local	25.95	134.83	1205.82	32.00	516.00	6593.00	74.47	478.05	3340.68
Traverse Links	33.97	304.51	2505.90	47.00	531.00	9735.00	7.67	29.44	222.12
Traverse Links	0.01	0.05	0.15	15.00	469.00	5656.00	5.55	28.04	216.67

Table 2. Operation Times for Linked List Stagger and Traversal (ms)

6 Summary and Future Work

This paper has described MobJeX, a framework under development within the ATcrc, which aims to provide dynamic application adaptation based on runtime resource monitoring, in order to support the transparent distribution of Java application objects (including AWT/Swing user interface objects) to desktop, PDA and smartphone devices in a heterogenous Internet environment. The MobeX Mobility Subsystem (MS) has been described in detail, with its performance compared to FarGo and Voyager in a series of laboratory tests. The results illustrate the scalability of the new proxy and interface model and the benefit of allowing mobile objects to be transparently resolved to local references, even after a chain of moves and method calls. However, the results also illustrate the advantage of the custom remote procedure call implementation of Voyager, a commercial ORB providing transparent object mobility, thus demonstrating that Sun's RMI implementation is not the most efficient way of supporting the mobility of objects.

Future work on the MS will involve optimisation of the remote procedure call mechanism as well as the implementation and testing of a declarative authentication and security mechanism using functionality that is transparently embedded in the proxy and object implementation. Development of the not yet realised Rule Engine component of the MobJeX system is also underway. This will allow a high level of separation of application layout, object grouping and adaptation strategies. This in turn will complement current research being done in the ATcrc, involving an analysis of distributed mobile object systems and the correlation between object based software metrics such as method invocation overhead and relative execution time, and environmental parameters such as network and computational resources. This will lead to an analysis of the impact of such variables on the design, deployment and performance of mobile object systems.

Finally, although MobJeX aims to provide a model that is transparent to developers and simplifies the process of implementing distributed applications, there is interest in examining the effects of using asynchronous calls with future objects, as done in ObjectWeb's ProActive system [32, 33]. The research question of interest is to what extent the decision to make a call asynchronous can be done transparently and declaratively, and thus deferred until runtime. ProActive shows that asynchronous calls based on future objects can provide block if necessary behaviour to simulate synchronous call semantics, however more work is required to establish techniques for determining when a call in an existing application is a good candidate for asynchronous call scheduling.

References

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

[2] S. Moyer and A. Umar, "The Impact of Network Convergence on Telecommunications Software," *IEEE Communications*, vol. January, pp. 78-84, 2001.

[3] Sun Microsystems, "Java 2 Micro Edition URL: http://java.sun.com/j2me/," 2004.

[4] Symbian Ltd, "Symbian OS URL: http://www.symbian.com/," 2004.

[5] Microsoft Corporation, ".NET Compact Framework URL: http://msdn.microsoft.com/mobility/prodtechinfo/devtools/netcf/," 2004.

[6] Sun Microsystems, "Java 2 Standard Edition URL: http://java.sun.com/j2se/," 2004.

[7] IBM, "WebSphere Micro Environment URL: http://www-306.ibm.com/software/wireless/wme/features.html," 2004.

[8] Insignia Solutions Inc., "Jeode Runtime URL: http://www.insignia.com/jeoderuntime.shtml," 2004.

[9] Microsoft Corporation, "Pocket PC URL: http://www.microsoft.com/windowsmobile/products/pocketpc/," 2004.

[10] M. Shapiro, "Structure and Encapsulation in Distributed Systems: the Proxy Principle," presented at Proc.6th Intl. Conference on Distributed Computing Systems, Cambridge, Mass. (USA), 1986.

[11] O. Holder, I. Ben-Shaul, and H. Gazit, "System Support for Dynamic Layout of Distributed Applications," Techinon - Israel Institute of Technology 1191, October, 1998 1998.

[12] O. Holder, I. Ben-Shaul, and H. Gazit, "Dynamic Layout of Distributed Applications in FarGo," presented at 21st Int'l Conf. Software Engineering (ICSE'99), 1999.

[13] Recursion Software Inc., "ObjectSpace Voyager URL: http://www.objectspace.com," 2002.

[14] L. Capra, W. Emmerich, and C. Mascolo, "CARISMA: Context-Aware Reflective middleware System for Mobile Applications," *IEEE Transactions on Software Engineering*, vol. 29, pp. 929-945, 2003.

[15] P. Bellavista, A. Corradi, and R. Montanari, "Dynamic Binding in Mobile Applications : A Middleware Approach," *IEEE Internet Computing*, vol. 7, pp. 34 - 42, 2003.

[16] A. T. S. Chan, "MobiPADS: A Reflective Middleware for Context-Aware Mobile Computing," *IEEE Transactions on Software Engineering*, vol. 29, pp. 1072-1085, 2003.

[17] A. Acharya, M. Ranganathan, and J. Saltz, "Sumatra: A Language for Resource-aware Mobile Programs," in *Mobile Object Systems: Towards the Programmable Internet*, J. Vitek and C. Tschudin, Eds. Heidelberg, Germany: Springer-Verlag, 1997, pp. 111-130.

[18] M. Fuad and M. Oudshoorn, "AdJava - Automatic Distribution of Java Applications," presented at Twenty-Fifth Australian Computer Science Conference, Melbourne, Australia, 2001.

[19] S. Fünfrocken, "Transparent Migration of Java-based Mobile Agents," presented at Proceedings of the Second International Workshop on Mobile Agents (MA'98), Stuttgart, Germany, 1998.

[20] T. Sekiguchi, H. Masuhara, and A. Yonezawa, "A Simple Extension of Java Language for Controllable Transparent Migration and Its Portable Implementation," in *Coordination Models and Languages*, 1999, pp. 211-226.

[21] S. Bouchenak and D. Hagimont, "Zero Overhead Java Thread Migration," INRIA 0261, 2002.

[22] K. Shudo and Y. Muraoka, "Noncooperative Migration of Execution Context in Java Virtual Machines," presented at Proc. of the First Annual Workshop on Java for High-Performance Computing (in conjunction with ACM ICS99), Rhodes, Greece, 1999.

[23] J. Jing, A. Helal, and A. Elmagarmid, "Client-Server Computing in Mobile Environments," *ACM Computing Surveys*, vol. 31, pp. 118-157, 1999.

[24] D. Milojicic, F. Douglis, Y. Paindaveine, R. Wheeler, and S. Zhou, "Process Migration," *ACM Computing Surveys*, vol. 32, pp. 241 -- 299, 2000.

[25] D. Milojicic, F. Douglis, and R. Wheeler, "Mobility - Processes, Computers and Agents." Massachusetts, USA: The Association for Computing Machinery, Inc., 1999, pp. 682.

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

[27] C. Ryan and S. Perry, "MobJeX: A Declaratively Configurable Java Based Framework for Resource Aware Object Mobility," presented at On The Move Federated Conferences (OTM '03) Workshops, Catania, Sicily, 2003.

[28] M. Fowler and J. Highsmith, "The Agile Manifesto," *Software Development*, vol. August, pp. 28-32, 2001.

[29] R. J. Fowler, "The Complexity of Using Forwarding Addresses for Decentralized Object Finding," presented at Proc. 5th Annual ACM Symposium on the Principles of Distributed Computing, Calgary, Canada, 1986.

[30] O. Holder and H. Gazit, "FarGo Programming Guide," Technion - Israel Institute of Technology EE Pub 1194, January 31 1999.

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

[32] F. Baude, D. Caromel, F. Huet, and J. Vayssie, "Communicating Mobile Active Objects in Java," presented at Proceedings of HPCN Europe 2000, 2000.

[33] ObjectWeb Consortium, "ProActive URL: http://www-sop.inria.fr/oasis/ProActive/," 2004.