MobileJ: System Support for Dynamic Application Partitioning in the Mobile Environment

by

Geoffrey Lloyd Burian

B.Sc. (Hons), The University of British Columbia

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF

THE REQUIREMENTS FOR THE DEGREE OF

Master of Science

in

THE FACULTY OF GRADUATE STUDIES

(Department of Computer Science)

we accept this thesis as conforming
to the required standard

The University of British Columbia

October 1998

In presenting this thesis in partial fulfilment of the requirements for an advanced degree at the University of British Columbia, I agree that the Library shall make it freely available for reference and study. I further agree that permission for extensive copying of this thesis for scholarly purposes may be granted by the head of my department or by his or her representatives. It is understood that copying or publication of this thesis for financial gain shall not be allowed without my written permission.

Department of Computer Science

The University of British Columbia
Vancouver, Canada

Date October 14, 1998

DE-6 (2/88)

Abstract

With the increasing proliferation of mobile computing devices comes the need for operating system software which supports applications running in the mobile environment. The *computational environment* of a mobile computer is typically more constrained than that of a stationary computer, having a less powerful CPU, less available memory, and being connected via a network with less available bandwidth and higher latency. Various software systems have been developed which attempt to compensate for these limitations; these may be characterised in terms of how much they hide the mobile environment from applications (*mobile-transparency*) and the degree to which they can dynamically locate application functions between the mobile machine and a stationary server in order to adapt to changes in the environment. The MobileJ system is one which supports mobile-transparent, dynamically partitioned applications for use in the mobile environment. By instrumenting Java class files (object code) and replacing all type references with those of proxy classes, the MobileJ system provides the basis for runtime control over placement of objects without requiring application programmers to change source code. MobileJ provides a mechanism to be used in the investigation of object placement policies for use in applications running in the mobile computing environment.

Contents

List of Tables

List of Figures

Acknowledgements

Though my name appears as the author of this work, many have supported me through the past two years as I completed this degree and thesis.

Thanks are due to Professor Mike Feeley for helping me learn and for encouraging, understanding, and supporting me during particularly stressful times. Thanks also to my supervisor, Professor Norm Hutchinson for his support, direction, and feedback on my ideas during the formulation of this thesis.

I'd also like to thank my parents, both those who have been there since the beginning, and those more recently acquired. Thank-you for your love and support (financial and otherwise!), especially over the past two years.

But most of all I would like to thank my wife, Amy-Lynn. Thanks for your love, patience, and encouragement as I finished my schooling. Thanks for sharing this adventure with me!

GEOFFREY LLOYD BURIAN

Vancouver, British Columbia

October 1998

Chapter 1

Introduction

1.1 Mobile Computing Characteristics

Within contemporary distributed computing, the term *mobile computing* has come to refer to environments where computational devices have the ability to function at arbitrary, as opposed to fixed, physical locations. Whereas traditional distributed computing models have assumed that machines typically remained at a fixed location for long periods of time, mobile computing assumes they have the ability to change their physical and possibly their logical (or network) location.

At least three trends have contributed to the emergence of mobile computing. First, the increasing proliferation of computational devices has meant that more traditionally mobile, yet non-computerized tools, are being made available. For example, personal appointment books, once handled solely by pencil and paper, are now often kept entirely using a digital computer. Second, the shrinking size of relatively powerful computational devices is making them much easier to move around. What was once too large to conveniently carry now fits easily in the palm of

a user's hand. Third, wireless networks and associated communications protocols are enabling applications requiring a network connection to run on computers without requiring them to be physically connected by a wire.

While we have solved some of the technical challenges involved in creating computational devices small enough to move around yet powerful enough to be useful, there remain many issues inherent to mobile computing which must be investigated further [Sat96]. First, mobile computing devices will always have fewer resources (for funds spent) compared to stationary devices. They will have less memory, less powerful CPUs, and likely be connected by networks with lower available bandwidth and higher latency than stationary devices.

Second, the *computational environment*, meaning primarily the CPU power, available memory, and network connection quality, will usually be highly variable. As we move from environments where users control few stationary computers (or, often, a single computer), to one in which users access many different computational devices from many locations, more variability is inevitable. As people are enabled to access their home environment from a variety of physical locations, so to will their computational environment vary.

For example, even when using the same computational device, the connection to a mobile host can vary between being completely disconnected, and being connected via a relatively reliable wired LAN. Wireless networks have inherent problems with performance and reliability because their physical transport medium, radio waves, can and does interact with its environment. As a mobile computer moves while accessing a wireless network, its environment changes, and thus the quality of its network connection changes as well. Because of the usually lower performance of a wireless network, users of mobile computers should be given the option

of being connected by a high-performance wired link; this also creates variability in connection quality.

A number of other differences exist between the mobile computing environment and the stationary environment, however, the two mentioned above will be the focus of this thesis. The following two sections discuss different approaches to these challenges, and introduces the particular technique described in the remainder of this thesis.

1.2 System Support For Mobile Applications

Given the unique characteristics of the mobile computing environment mentioned above, many applications designed for use in a stationary environment will not perform well. Thus various application design frameworks and system software services have been developed for creating or adapting applications for use in the mobile computing environment.

1.2.1 Mobile Transparency vs. Mobile Awareness

A primary concern in the design of any software system supporting mobile applications, is the degree to which it is transparent to the application [Sat96]. At one end of the spectrum are "ad-hoc" solutions, whereby individual applications implement the services necessary for them to function well on a mobile computer. These applications are completely *mobile-aware* because the software which supports operation on a mobile host is integrated with the software which performs normal application logic. This solution has the advantage of being tailored exactly to a particular application's needs, but has a major disadvantage in that code supporting mobility is duplicated for each application. Not only is more memory (and possibly other

resources) required to run multiple applications on the mobile computer, but much more development time is needed by the application designer and implementer; this effect is multiplied by the number of mobile applications created. This problem is merely a particular case of the argument for the existence of operating system services in the more general case.

At the other end of the spectrum are systems which allow applications to run without modification on a mobile computer. Usually this is done by redesigning system software such that it takes the mobile computing environment into consideration. The application is not, for the most part, aware that it is running on a mobile host, and is designated *mobile-transparent.* This solution has the advantage of allowing many applications not designed for mobile computing to run on a mobile computer, and also does not require major application software reengineering efforts; application engineers can continue to build applications as if they were creating software for stationary computers. The possible disadvantage of this approach is that it may not be possible to implement system software in such a way as to foresee and compensate for the impact of mobile computing for all applications. The way the mobile computing environment impacts one particular application may be quite different than for another application.

In the middle of this continuum lie strategies which provide system software services for aiding applications to function in the mobile computing environment. This strategy attempts to merge the advantages of both ends of the spectrum to achieve an ideal solution. Therefore, current applications would require a certain amount of redesign in order to work on mobile hosts, but the application designer would have control over what services were used by the application and under which circumstances. Most of the software supporting mobility would be shared by mobile

applications, but they would be able to use it in ways specific to their needs. This strategy creates applications which are *mobile-transparent* to the extent that the system software hides the mobile computing environment. The difficulty inherent to this approach is the same as for any implementer of system software, that being how to design services such that the system is capable of supporting the needs of all applications, while also not being too complex or cumbersome to use and maintain.

1.2.2 Application Partitioning

Most software that enables applications to be built or modified for use in the mobile computing environment can be described in terms of *application partitioning* [Wat95]. A partitioned application is one in which the user interface and possibly some part of the general application logic executes on a mobile host, while another part executes on a more powerful, stationary host. The two parts are typically connected by a wireless network link. The partition is created in such a way as to minimize reliance on the scarce resources of the mobile host and the low bandwidth and high latency wireless network connection. The ideal partition is one in which large data sets and functions requiring high computational power are located on the relatively large and powerful stationary host, while communication between the hosts is optimized for the poor performing wireless network. For example, the SciencePad system described in [DWJ+96] attempts to provide a "ubiquitous problem solving environment" for scientists by partitioning applications between a small, mobile host used to display the user-interface and more powerful hosts used to perform computation.

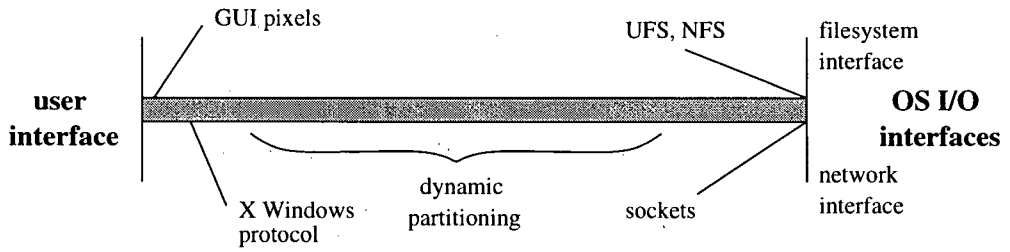

Figure 1.1: Possible Application Partition Points

Partitioning Point

An application can be partitioned at many different points as illustrated in Figure 1.1. An application can be partitioned at some layer of its user-interface code, as described in [RSFWH98]. However, depending on the point at which the partition is implemented, this scheme may require many interactions between the mobile and stationary hosts as the user controls the application, particularly if a modern, graphical user interface is used. Therefore, performance over a high latency wireless network would not be optimal. Also, if the partition involves sending pixel images of the user's display, the low bandwidth restrictions of a wireless network may make adequate performance difficult to achieve, as noted in [KDF+93].

Alternatively, an application can be partitioned at the filesystem or network access level. However, this assumes that the mobile host is powerful enough and has a large enough memory to accommodate the execution of most of the application. This scheme may be appropriate for a relatively powerful mobile host which needs access to a large data set, but may not be appropriate for use on smaller, less powerful mobile hosts.

Static vs. Dynamic Partitioning

Application partitioning can be implemented in either a static or dynamic fashion. A static partition is one in which the partition point stays fixed during application execution and over consecutive executions of the application. A dynamic partition is one in which the partition point can change between consecutive executions of the application, or even during the execution of the application.

A static partitioning scheme is very similar to the traditional client-server model of network applications, though the placement of functionality may be different to compensate for the differences between a powerful, well-connected client and a small, poorly connected mobile host [Sat96]. Static partitioning usually implies some degree of mobile awareness on the part of the application and thus of the application developer. It may also assume the application will always execute on a mobile host of a particular computational performance, memory size, and connected by a wireless network.

A dynamic partitioning scheme, on the other hand, allows for more flexibility than a static partition. As noted in Section 1.1, the mobile computing environment is characterised by high variability in CPU power, available memory, and network connection quality–variability that is likely to increase in the future as users are enabled to become less tied to a physical location in order to access their home environment. As the computational environment changes, it may be desirable to change an application's partition point: one partitioning configuration may not be suitable for all environmental configurations.

Resource	Condition	Policy
network	high latency	migrate objects with many interactions to the same host
	low bandwidth	migrate objects with high bandwidth interaction to the same host
mobile CPU	high load	migrate/create computationally expensive objects to/at stationary host migrate/create threads to/at stationary host
mobile memory	memory full	migrate/create large objects to/at stationary host

Table 1.1: Resource Constraints Influencing an Application Partitioning Policy

Partitioning Policy

In designing a partitioned application, or system support for partitioning, decisions must be made on an appropriate distribution of application data and function execution between the mobile and fixed hosts. Table 1.1 displays the factors which can influence these decisions. (Table 1.1 expresses application components in terms of object instances, where object location implies both data and computation location.)

It should be noted that in a completely mobile environment, the conditions of all resources can change both between executions of an application, and during its execution. As a particular resource becomes more constrained, so too does its relative importance to an application partitioning policy. Also, for resources which are "equally" constrained (for some definition of "equal"), a partitioning policy must weigh the relative influence of each resource condition when choosing where to place an object.

1.2.3 A Taxonomy of Systems

Table 1.2 shows a taxonomy of systems with some degree of support for mobile applications. The systems are categorized according to the static/dynamic nature of

	Static Partition	Dynamic Partition
Mobile-Aware	• Odyssey [Nob98, NSN+97, NPS95] • Welling's system [WB98, WB97]	• Rover [JTK97, JK96, Tau96, JdT+95] • Sumatra [ARS97, RASS97, RAS96] • Wit[Wat95, Wat94b, Wat94a]
Mobile-Transparent	• wireless X [KDF+93] [Dan94] • VNC [RSFWH98, WRB+97] • AMIGOS [HRAJ98, HR96, GM95] • MAF [HR97]	• MobileJ • M-Mail [Lo97, LK96] • Coign [HS98b, HS98a, HS97]

Table 1.2: A Taxonomy of Systems Supporting Mobile Applications

their application partition and the degree to which they hide the mobile environment from applications.

Statically partitioned systems tend to be implemented at either the network or file system interface, or at some layer of the user-interface/graphical display layer. For example, Odyssey concentrates on mobile file access, while the wireless X and VNC systems focus on re-direction of an application's user-interface to a mobile host. Dynamically partitioned systems, for example Rover, Sumatra, Wit, and MobileJ, often rely on a software interpreter for dynamically moving and loading code and data.

These systems are described more fully in Section 5.2, beginning on page 73.

1.3 Motivation and Purpose

This thesis describes the design and implementation of MobileJ, a mobile-transparent system which can be used to create dynamically partitioned applications for execution on the Java Virtual Machine. Most other systems for supporting mobile applications are either mobile-aware or use static partitioning (or both), however, dynamically partitioned, mobile-transparent systems have significant potential advantages over these systems and more research is needed to further characterise and verify their potential.

As discussed above, a disadvantage of mobile-aware systems is the need for application engineers to concentrate on both traditional application design and implementation, as well as issues related to the mobile nature of the application. Mobile-transparent systems free software engineers from concern about these issues, allowing them to continue developing applications for the mobile environment as they have for the stationary environment. This promotes quicker development of new applications for use in the mobile environment, and allows for applications previously developed for the stationary environment to be used on a mobile host.

The major potential disadvantage of mobile-transparent systems is that the system may not be able to compensate for the mobile environment enough for all applications to achieve adequate performance. In this case, and especially when the system also allows for dynamic partitioning as does MobileJ, mobile-transparent systems still provide a platform for development of application prototypes where various configurations can be rapidly tested. A good understanding of how the application should be optimally configured for use in the mobile environment can thus be achieved quickly.

While the current version of MobileJ does not implement completely auto-

matic, runtime control of distribution, it does provide the basis for implementation of specific object placement policies. In particular, MobileJ can be used as a tool to investigate at least three research questions concerning the partitioning of mobile applications. First, there is a need to determine at what point various types of applications should be partitioned for best performance in a mobile environment. With MobileJ, there is the possibility of testing many applications built without explicit support for mobility to determine an optimal object placement configuration.

Second, there is a need to investigate to what extent various resource constraints inherent to the mobile environment should influence partitioning policies. For example, moving an object from the mobile host to a stationary host frees memory on the mobile host, but the cost of transporting the object over a low bandwidth wireless network may not make the transfer worthwhile. Given the highly variable nature of the mobile computing environment, there is a need to understand a variety of environmental configurations and how applications should be partitioned for each.

Third, MobileJ can be used to explore the possibility of runtime control over partitioning policy. With the addition of a resource monitoring module, MobileJ could be used to control all object placement decisions for a running application according to a set of rules. It remains to be determined, however, if the advantages of this strategy will outweigh its possible disadvantages compared to more mobile-aware methods.

Most systems which support mobile-transparent application partitioning do so by changing or adding to operating system or library code. Doing so places constraints on where a partition can be made and also on how dynamic the partition can be. Most systems which implement a dynamic partition, however, also create

a mobile-aware system, where application programmers must consider how their application will be affected by the mobile environment.

As an alternative design, instrumenting application executable code, as is done in MobileJ, has the advantage of partitioning an application at an arbitrary point, under runtime control. As no source code is changed, using instrumentation also enables a mobile-transparent system to be created; ideally the application programmer need not concern himself with the details of application distribution between hosts. It is the intent of the MobileJ project to create a software infrastructure whereby one can investigate how far the mobile-transparent approach can go in terms of adjusting to the constraints of the mobile environment, and in partitioning applications not necessarily designed for the mobile environment. The Java Virtual Machine (JVM) class file (object code) format provides an excellent basis for such a project, as will be described in the following section.

Chapter 2

MobileJ Architecture and Overview

2.1 General Architecture and Overview

Figure 2.1 (page 14) shows the overall system architecture of the MobileJ system. Java class files (object code), as produced by a Java language compiler or assembler, are first processed by the MobileJ *Instrument* tool, creating instrumented bytecode in the form of additional class files. The instrumented code, along with the *MobileJ Runtime*, is executed on a Java Virtual Machine (JVM), with the assistance of a native library.

Typically, a running MobileJ system consists of a single client and a single server which communicate via a TCP/IP network connection. It is also possible for the client and server to run on the same JVM, i.e., the client host and server host are the same. The only difference between running MobileJ as a client and running MobileJ as a server is that when running as a client, the `main()` method of

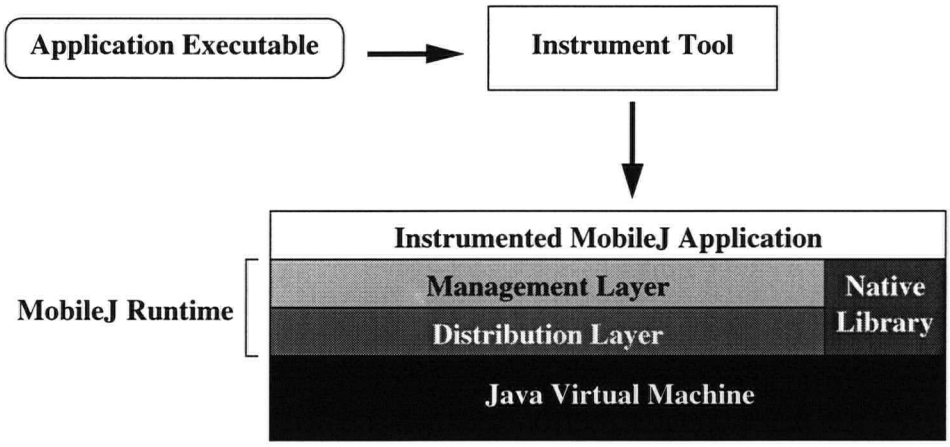

Figure 2.1: MobileJ System Architecture

a specified application is invoked, while when running as a server it is not.

When running as the client, the MobileJ Runtime supports execution of a single application; similarly, when running as a server, MobileJ supports connection with a single client. Also, MobileJ currently has support for a single server per client application, but it is possible that this could be extended to support multiple servers per client application.

The MobileJ Runtime is responsible for distribution of objects created by the instrumented application between the client (mobile) and server (fixed) hosts on the network. In particular, the management layer manages an object placement policy, while the distribution layer is responsible for implementing the policy of the management layer–moving objects between the two hosts.

MobileJ makes use of *proxy objects* to maintain location transparency of application objects. Each proxy object is an instance of a *proxy class*, of which there is one corresponding to each instrumented class used under MobileJ [1]. Proxy objects

[1]Exceptions to this are instrumented versions of `java.lang.String`, and subclasses of `java.lang.Throwable`. No proxy classes are created for these classes. The reasons for these naming exceptions are described in sections 4.7.2 and 4.7.1.

form part of the MobileJ Runtime Distribution Layer (see Section 3.2, page 23).

The MobileJ Instrument tool changes class files such that their code manipulates proxy objects almost exclusively; nearly all objects created and manipulated by an application running under MobileJ are proxy objects. All instance method invocations are done through a proxy object which invokes the method on a local object if the object resides on the local host, or on a remote object, if the object associated with the proxy resides on the remote host. Constructors (initialization methods called at object creation time) of proxy classes create non-proxy (real) objects on either the mobile or fixed host, as determined by the MobileJ Runtime.

Both the MobileJ Instrument tool and Runtime are written entirely in Java, with the Runtime requiring support from a native code library written in C. The native library is necessary because of the tight integration between certain classes in the Java standard library with the JVM, and also to implement native methods of the instrumented versions of the standard Java classes.

2.2 Static Data and Method Design

An application running under MobileJ executes as if it were executing on a single host. Because there are two Java VMs operating concurrently in a typical configuration of MobileJ, there are two copies of static data and two possible locations where static methods may be invoked.

To preserve the semantics of a Java application executing on a single host, MobileJ identifies the server host as the *static host* in the MobileJ system. Figure 2.2 (page 16) shows the design of MobileJ static data. All object type static fields created in a static initializer method are created, by default, on the static host. (This placement can be changed according to the Object Placement Policy, as described

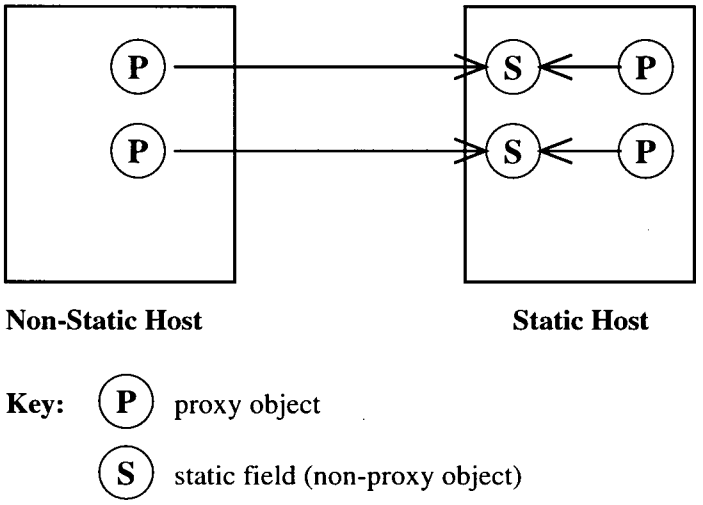

Figure 2.2: Static Data Design

in Section 3.1.2, page 22.) Primitive type static fields are located, by default, on the static host and cannot be moved.

Object type static fields are accessed via a local reference to a proxy object, while primitive type static fields are accessed via static accessor methods added to a class during instrumentation, as described in Section 4.6.3, page 52.

Static methods are always executed on the host where the invocation begins because this does not change the semantics of static method invocation for a Java application running on a single host.

2.3 Introduction to Java Concepts

This section presents an introduction to the Java language, runtime environment, and class file (object code) format. Further sections of this thesis assume knowledge of these concepts.

Java [Fla97] is an object-oriented programming language and runtime envi-

ronment invented by engineers at Sun Microsystems. Java ASCII text source files are compiled by the `javac` compiler into *class* files containing bytecode instructions. The bytecode is interpreted by the *Java Virtual Machine* (JVM) [MD97], an efficient runtime environment implemented in software atop various hardware and operating system platforms. Because its object code format, class files, can be executed on any machine architecture and operating system to which the JVM has been ported, Java is said to be architecture independent and portable.

Data in Java programs is either of a primitive type, handled by value, or a reference type. Primitive types include shorts, integers, longs, floats, doubles, characters, bytes, and booleans. Reference types are either arrays or objects which are instances of classes declared in Java source code.

As with many object-oriented languages, Java classes exist in an inheritance hierarchy, rooted at the class `java.lang.Object`. Classes must inherit from only one super-class, but may implement any number of *interfaces* (see below). Classes are grouped into *packages*; the full name of a class consists of the package name and the class' *simple name*. For example, `java.lang.Object` is in the package `java.lang` and its simple name is `Object`.

Java classes contain procedures, called *methods*, and variables, called *fields*. Java methods may be either *constructors*, *instance methods*, or *static methods*. Constructors are invoked (called) to initialize a new class instance (object). Instance methods, when invoked, are implicitly passed a reference to a particular object by which instance fields may be referenced. Static methods, sometimes referred to as *class methods*, are not passed a reference to a particular object when invoked.

	Type	Descriptor
Primitive Types	byte	B
	char	C
	double	D
	float	F
	int	I
	long	J
	short	S
	boolean	Z
Reference Types	class	L<class name>;
	array	[<type>

Table 2.1: Java Type Descriptors

In the class file format, methods are identified by name and type descriptor, or *signature*. A method type descriptor is of the form:

```
(<parameter types>)<return type>
```

where `<parameter types>` is a concatenation of the type descriptors of all parameters for the method, and `<return type>` is the type descriptor of the method's result. Table 2.1 shows the type descriptors of all Java primitive and reference types.

While most methods in a Java program are defined with bytecode instructions interpreted by the virtual machine, methods may also be declared as `native`, in which case they are defined by code compiled to the instruction set of the particular hardware CPU on which the virtual machine is executing. The use of native methods limits the portability of a Java class because a different object code representation is needed for each machine architecture and operating system on which the code is to run.

Java methods may "throw" any number of *exceptions*. Exceptions are objects which inherit from the class `java.lang.Throwable`. Exceptions propagate from the

point they are thrown, up through the method call stack until they are trapped by an *exception handler* for their type, or they reach the top of the stack, at which point the JVM halts execution.

Java fields may be either *instance fields* or *static fields*. Each object created by a Java program may have a number of associated instance fields allocated with it. These may be of primitive or reference type. Static fields are allocated on a per-class basis and may optionally be initialized when the class is loaded into the virtual machine via a static initializer method.

A Java class may implement any number of *interfaces*, which are declarations of methods and constant static fields. The names, parameter types, and return type of each method is declared in an interface, and a class which implements the interface must contain implementations of these methods.

Java source code is compiled into an object code format known as *class files*. Each class file contains the executable code and structure for one Java class or interface. Within a class file is a *constant pool* in which is stored all constant data such as class names, method names, and string and numerical constants referred to by various structures and executable code in the remainder of the class file. The large amount of symbolic information retained in a class file facilitates dynamic linking of code both locally or even over a network. It also makes this executable format easier to instrument compared to more traditional operating system executable formats.

The JVM loads class files on demand and interprets method bytecode instructions. The JVM is a stack-based virtual machine; the JVM instruction set contains many instructions for manipulating data on the stack, as well as fairly high-level instructions, for example, to invoke a method. Being stack-based also makes instrumentation of bytecode somewhat easier as code can be inserted while

retaining the current stack state.

While the bulk of most Java applications consists of bytecode interpreted by the JVM, a Java Native Interface (JNI) [Sun97] also exists to allow linking of methods compiled to the instruction set of the hardware on which the JVM is executing.

Chapter 3

MobileJ Runtime

3.1 Management Layer

The MobileJ Management Layer is the portion of the Runtime which is responsible for application startup and for controlling any runtime decisions regarding object placement between the two hosts in a MobileJ system. The Management Layer uses the services of the Distribution Layer to implement its policies.

3.1.1 Runtime Parameters

The MobileJ Runtime system may be invoked with a number of parameters which control the configuration and operation of the system:

- `-port <port number>` - TCP port number identifying this MobileJ Runtime instance
- `-client <client-address:port>` - domain name or IP address and TCP port number of the MobileJ client; specified when running as a server

- `-server <server-address:port>` - domain name or IP address and TCP port number of the MobileJ server; specified when running as a client
- `-main <main-class>` - name of the proxy class containing the main() method of the application to be run; this is only specified when running in client mode and must be the final MobileJ argument on the command line, after which all application specific arguments must be placed
- `-trace <trace-file>` - name of a file to which all MobileJ method call trace output should be directed

Thus, specifying a client address starts MobileJ in server mode, while specifying a server address starts MobileJ in client mode.

3.1.2 Object Placement Policy

As discussed in Section 1.3 (page 10), many environmental factors could influence an object placement policy used for application partitioning in a mobile computing environment. MobileJ currently supports placement policies based on *Object Type* only, though other, more sophisticated policies could be added with relative ease. (see Section 6.2, page 82).

The MobileJ Runtime has a number of static (Java class) fields which may be referenced by an executing MobileJ application. References to these fields are placed in instrumented code as required. Particular uses of these fields are described in Chapter 4, which begins on page 27. The following static fields are referenced by instrumented application code:

The first four fields store the Internet domain name and TCP port number of an instance of the MobileJ Runtime:

- `String STATIC_HOST` - host where static field data is stored by default
- `String CLIENT_HOST` - host where MobileJ runs in client mode
- `String SERVER_HOST` - host where MobileJ runs in server mode
- `String OTHER_HOST` - same as `SERVER_HOST` when running in client mode; same as `CLIENT_HOST` when running in server mode

Three fields store boolean values which may be checked in instrumented application code:

- `boolean isServerHost` - true if the instance of MobileJ is running in server mode
- `boolean isClientHost` - true if the instance of MobileJ is running in client mode
- `boolean isStaticHost` - true if the instance of MobileJ is the static host

3.2 Distribution Layer

The MobileJ Distribution Layer is responsible for implementing location transparent method invocation via proxy objects. The general format of proxy classes is described in Section 4.8, page 66. The current implementation of MobileJ uses Voyager [Obj97] as its Distribution Layer, but this could be replaced in the future (see Section 6.2.2, page 83).

3.2.1 Object Creation

When a new instance of a proxy object is created (by code which has been processed by the Instrument tool), it must, as part of its initialization, create an instance of

the class for which it is a proxy. The non-proxy instance can be created either on the local machine or on a remote machine, depending on the location passed to the proxy constructor. Wherever the non-proxy object is created, the proxy object maintains a way to reference this object for further method invocations. The proxy object is able to find its corresponding non-proxy object even after being cloned or serialized and sent to a remote machine, as happens when proxies are sent as arguments in method invocations to a remote machine.

3.2.2 Method Invocation

Application code which invokes an instance method or constructor does so using a reference to a local proxy object. The proxy object method must first determine if its non-proxy object is local or remote. If the object is local, it invokes the appropriate method in the non-proxy object, passing any arguments as supplied by the caller. If the method has a result, it is returned by the proxy method to the application code.

If the object is remote, the proxy object serializes all method arguments (using standard Java object serialization [RWW96]), if any, and sends them in a method invocation request to the remote MobileJ host.

Upon receiving an invocation request and its associated arguments, MobileJ deserializes all method arguments, finds the non-proxy object for the invocation, and invokes the method, passing all method arguments. Since all object type arguments, including references to arrays, are proxy objects, serialization and deserialization only creates a copy of the proxy objects, not the objects they represent, preserving method invocation semantics for remote calls.

When the method returns, a reply, including any result value, is sent back

to the original MobileJ host. The result value is deserialized and returned to the invoking application code by the proxy method.

If the remote method throws an unhandled exception, this is caught by MobileJ on the remote side, sent back to the original host, where it is re-thrown to the application code making the method call.

3.2.3 Object Mobility

In addition to remote object construction and method invocation, the MobileJ Distribution Layer also supports moving an object between hosts once created. A proxy object can be directed to move the instance it represents to a specific MobileJ host and port by invoking its `moveTo` method. The MobileJ Runtime is passed a reference to all proxy objects when created so that an object placement policy module can move objects between hosts as desired.

After moving an object, all proxies which exist for the object must still be able to find the object. This is accomplished in the current implementation of MobileJ (i.e., in Voyager) by forwarding all method invocations from the previous host to the new host, and returning the new location when sending the invocation result.

3.3 Native Library

Though, where possible, the MobileJ Runtime is implemented in Java, a native code library consisting of compiled C code is also required. This is needed primarily to supply definitions of native methods in instrumented versions of classes in the standard Java Class Library.

Two native methods which access Java Virtual Machine data structures,

`checkForThisParam` and `checkForThisTarget`, are also found in the MobileJ Native Library. These methods are described in Section 4.4.2, page 44.

Chapter 4

Class File Instrumentation Design and Implementation

4.1 Introduction

The MobileJ system includes an application, the *Instrument* tool, which processes zip (or jar) format files containing Java class files. In converting Java class files for use under MobileJ, the primary job of the Instrument tool is to change most type (class) references such that they are references to proxy classes. Because Java objects are handled by reference, proxy objects may be passed to and returned from method invocations, even between hosts, without changing the semantics of method invocations.

The Instrument tool processes class files in two steps. The first step converts class files for use with the MobileJ Runtime (see Chapter 3, beginning on page 21). The second step creates a proxy class based on the instrumented version of the original Java class.

By using proxy classes, object creation and method invocations may be intercepted and re-directed to the appropriate host. However, not all accesses to objects are done through method invocations. First, arrays are objects in Java, but have special JVM instructions which create and manipulate them. For dealing with arrays, the MobileJ Instrument tool creates *wrapper* classes, through which all arrays are accessed, as described in Section 4.3 (page 31). Second, fields are also manipulated directly by JVM instructions. The Instrument tool creates accessor methods for fields so that accesses to fields can be intercepted by proxy objects in the same way as method invocations (see Section 4.6, page 51).

Because all Java applications depend on the standard Java Class Library, this library was instrumented to create instrumented classes and proxy classes which correspond to all classes in the library. All instrumented classes and proxy classes eventually inherit from `java.lang.Object`, thus this class is not instrumented.

To support an Object Type object placement policy (Section 3.1.2, page 22), the Instrument tool accepts a parameter file as input when instrumenting a set of class files which indicates on which host, client or server, particular types of objects should be created. Specifying these parameters at instrumentation time means re-instrumenting classes to change these parameters, however, specifying this information statically makes for a much more efficient implementation. With a static implementation, object location names may be specified by referencing one of either `CLIENT_HOST` or `SERVER_HOST` variables in the MobileJ Runtime. However, indicating object creation location at run time would require an extra method call and hashtable lookup every time an object was created, even for locally created objects. Furthermore, the host a particular type of object is created on, if it is specified at all, is unlikely to change unless the class changes, which would require

re-instrumentation anyways.

Objects of types not specified in the instrumentation parameter file are always created on the local host, i.e., the host where the thread is running. Thus, by specifying where objects of a particular type are created, one can specify where most other objects created by objects of this type are created.

4.1.1 Chapter Overview

This chapter describes the changes made to class files by the Instrument tool in order to prepare them for execution with MobileJ. Primarily this involves changing all references to classes into references to proxy classes (Section 4.2). Since arrays are created and manipulated directly by JVM instructions, a method was needed to intercept these operations and allow for location transparent access to arrays, as described in Section 4.3. Section 4.4 describes how method invocation was made location transparent by making most method invocations go through a proxy object. Section 4.5 describes how proxy objects are created in instrumented bytecode so that nearly all objects manipulated by a MobileJ application are proxy objects. Because fields are also manipulated directly by JVM instructions, these instructions are generally transformed into method invocations through a proxy object, as described in Section 4.6. Section 4.7 details a number of miscellaneous changes required to allow instrumented class files to run under MobileJ. Finally, sections 4.8 and 4.9 provide an overview of how proxy classes are created and discussion of two issues pertaining to the implementation of bytecode instrumentation, repectively.

4.2 Class Level Changes

4.2.1 Class Names

The most fundamental change to all instrumented classes is their change of name. MobileJ instrumented classes are given a new name by being placed in a new package which has "`ubc.mj.`" prepended to the original package name. For example, the instrumented version of the class `java.lang.System` becomes `ubc.mj.java.lang.System`.

Proxy classes are also named by prepending "`ubc.mj.`" to the original package name, but in addition have "`V`" [1] prepended to the original simple class name. For example, the proxy class corresponding to the original `java.lang.System` class is named `ubc.mj.java.lang.VSystem`.

Most class names referenced in class files are changed to their equivalent proxy class name; exceptions to this rule include `java.lang.String` and all Java exception classes, i.e., subclasses of `java.lang.Throwable` which are described in sections 4.7.1 (page 58) and 4.7.2 (page 59). Also, as all instrumented and proxy objects eventually inherit from `java.lang.Object`, neither a proxy class nor instrumented class is created for `Object`.

4.2.2 Interfaces

Because Java interfaces have corresponding class files which are instrumented, the interfaces implemented by a particular class must have their names changed as well. A class is changed such that it implements instrumented interfaces corresponding to its original interfaces. The only exception to this is the marker interface

[1] The "`V`" stands for "virtual class", which is the terminology adopted by Voyager for referring to proxy classes.

`java.lang.Cloneable`, which is left as being implemented so that it continues to indicate to the JVM that the class's instances may be cloned.

Most objects of instrumented classes need to support being moved between machines; this is accomplished by the MobileJ Distribution Layer which, in turn, uses Java object serialization to serialize object data for transmission over the network. Therefore, all instrumented classes must implement the `java.io.Serializable` interface, indicating that they may be serialized. This is ensured by the Instrument tool.

4.3 Arrays

As noted above, MobileJ makes use of proxy objects to hide the location of the "real" objects referenced by them. Because objects are handled by reference in Java, the proxy objects can effectively direct method invocations to the appropriate "real" object, whether on the local machine or a remote one–either way, the semantics of Java method invocation are preserved. With the addition of accessor methods for object fields, all object state manipulation can be trapped by proxy objects at the method invocation level.

Like "normal" objects, Java arrays are also manipulated by reference, however, there are two primary differences relevant to MobileJ between arrays and normal objects. First, Java array classes are created dynamically, at run time, by the JVM, and not stored in class files. Thus no array class files exist upon which an instrumented version of the array class or proxy class can be based. Second, there are a number of JVM bytecode instructions which create and operate on arrays; these cannot be "trapped" by proxy objects without modifications to the JVM.

4.3.1 Design

To solve the problems noted above, MobileJ makes use of a hierarchy of *wrapper* classes, instances of which reference real Java arrays. These wrapper classes include constructors for creating "wrapped" array objects, as well as accessor methods for setting and getting array elements. A proxy class is also created based on the wrapper class. Therefore, all Java array operations can be converted to method invocations through a proxy object, as with field access operations on normal Java objects.

The JVM creates a different array class for each combination of array base type and number of dimensions. For example, a two dimensional array of integers is of a different class than a single dimensional array of integers. To mimic this with the array wrapper class, the Instrument utility creates multiple wrapper classes which inherit from the base array wrapper class, `ubc.mj.array.ArrayWrapper`. As code is being instrumented, the Instrument tool keeps track of all references to arrays and, after finishing the main instrumentation process, creates any `ArrayWrapper` sub-classes and proxy classes required.

`ArrayWrapper` sub-classes are named according to the formula:

`_<n>_<t><component_class_name>`

where:

- `<n>` is the number of dimensions of this array class
- `<t>` is either a primitive type descriptor, if the array's component's are primitives, or "`L`" if the array's components are objects

- `<component_class_name>` is the fully qualified class name of the object type component, with all periods (.) replaced with underscores (_); the `<component_class_name>` is only needed (and used) if the array component type is an object type

Though many `ArrayWrapper` sub-classes are created, all functionality is inherited from the `ArrayWrapper` class–the sub-classes exist only to distinguish array types to the JVM.

When checking if one array variable is assignment compatible with another array variable, Java checks the assignment compatibility of the arrays' elements. To conform with this definition of compatibility, the inheritance hierarchy of MobileJ array wrapper classes is as displayed in Figure 4.1.

This inheritance hierarchy ensures that references to array wrapper objects are assignment compatible in the same way as instances of normal Java arrays.

There is one limitation of this design worth noting. Java interfaces can inherit from *multiple* other interfaces, therefore, for array wrapper classes to wrap arrays of interfaces, they would need to inherit from multiple other classes. Java only allows *single* inheritance for classes, therefore a MobileJ array wrapper class which wraps an interface array may only inherit from a single interface array wrapper class. The Instrument utility selects the first inherited interface declared as the one inherited by the array wrapper class.

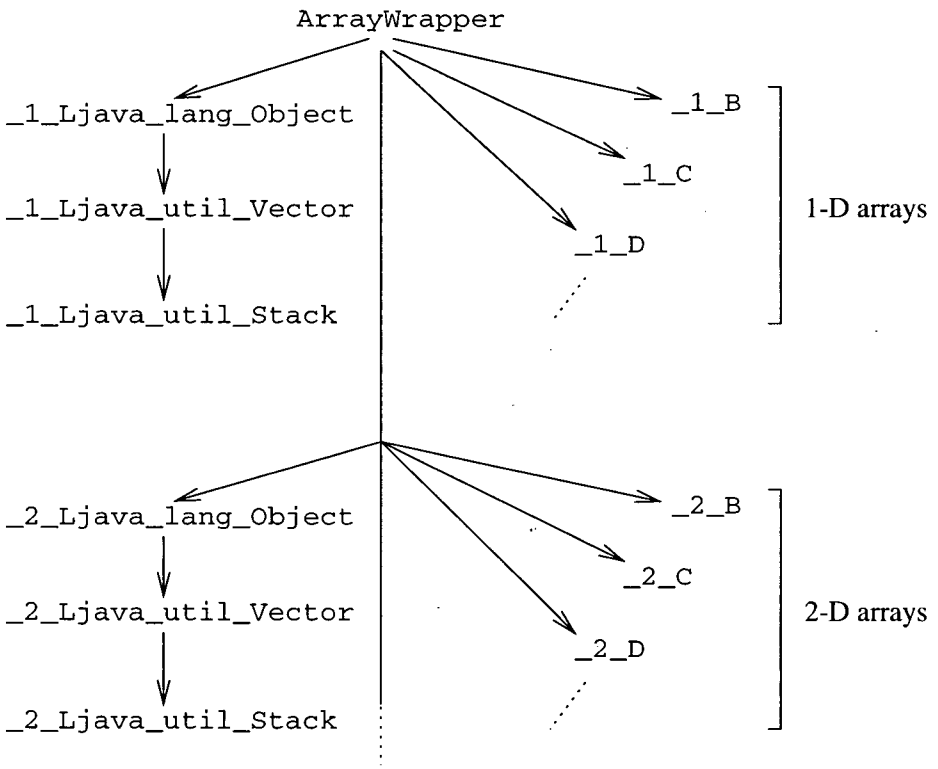

Figure 4.1: Array Wrapper Class Hierarchy

Multidimensional Arrays

In Java, multidimensional arrays are created as arrays of arrays, i.e., elements of multidimensional arrays are other arrays. For example, a two dimensional array `twoD` could be assigned as an element of the three dimensional array `threeD`:

```
int[][] twoD = new int[10][10];
int[][] threeD = new int[10][][];
threeD[5] = twoD;
```

The data structure arrangement and element types for a wrapped three dimensional array of integers are pictured in Figure 4.2 (page 36).

Because the sub-arrays of a multidimensional array are also arrays, and handled by reference, when wrapped, the elements of lower dimensional sub-arrays of a multidimensional array are instances of `VArrayWrapper` sub-classes.

4.3.2 Creation

Java arrays may be created by the JVM using three different bytecode instructions: `newarray`, `anewarray`, and `multianewarray`. The instrumentation and corresponding `ArrayWrapper` constructor used when replacing each instruction in instrumented code is discussed below.

Creation with `newarray`

The simplest form of array creation is done with `newarray`, which creates a single dimension array of primitive component type. `newarray` takes the top stack word, `n`, and creates an array of `n` whose components will be of a primitive type indicated by a byte which is part of the instruction.

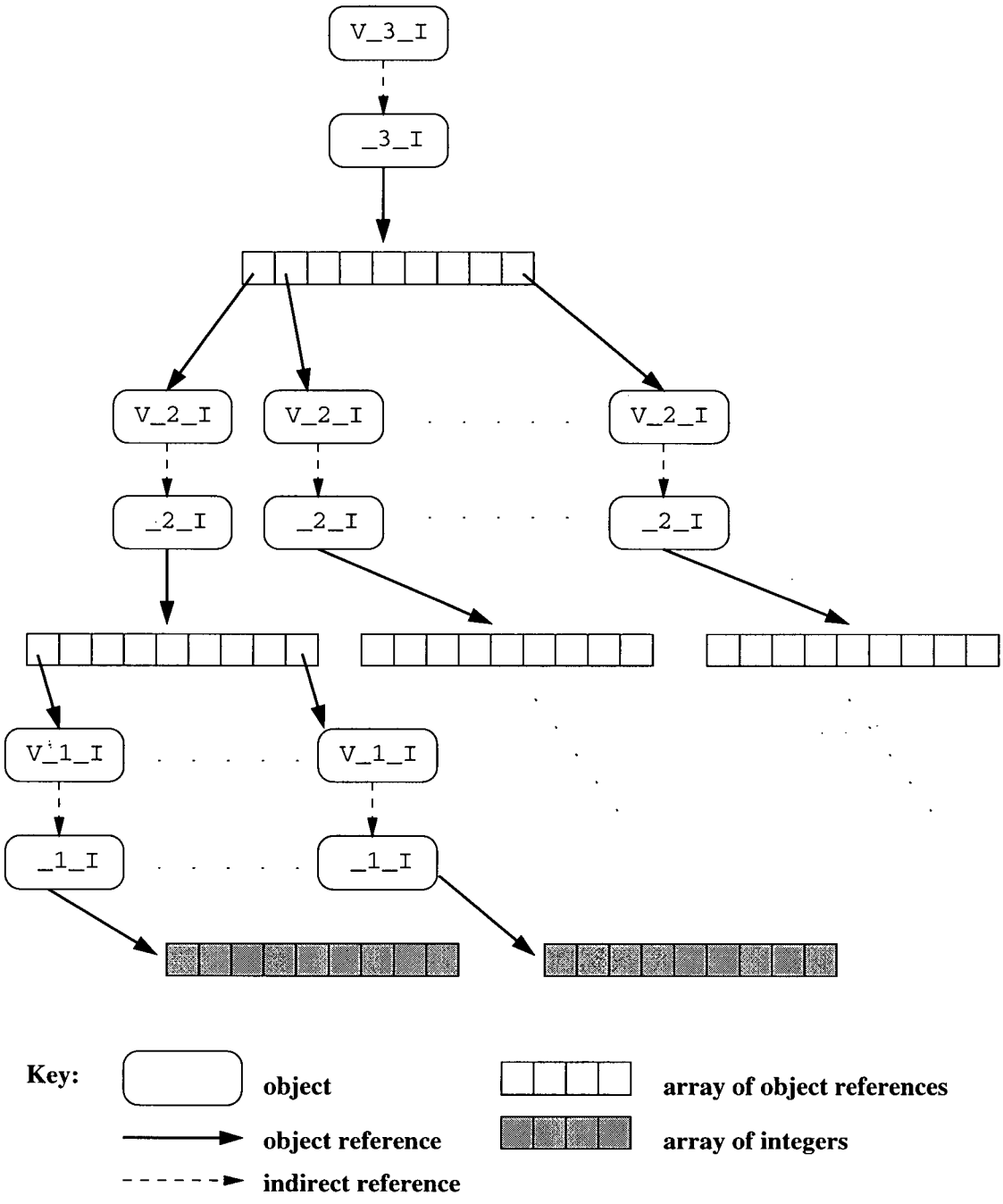

Figure 4.2: Data Structure and Element Types of a Wrapped Three Dimensional Array of Integers

```
new <VArrayWrapper-sub-class>
dup_x1
swap
ldc <type-code>
<load-object-host>
invokespecial <VArrayWrapper-sub-class>(IILjava/lang/String;)
```

Figure 4.3: Instructions Replacing `newarray`

To emulate this instruction using the `ArrayWrapper` class, the `newarray` instruction is replaced with the instructions shown in Figure 4.3.

The `new` instruction creates an uninitialized instance of the appropriate `VArrayWrapper` sub-class. The `dup_x1` and `swap` instructions arrange the stack such that the reference to the `VArrayWrapper` object is below the array size (integer), placed on the stack by code immediately preceding this. The first `ldc` instruction loads an integer, the array type-code, indicating the array component's primitive type, and the instruction indicated by `<load-object-host>` is either an `ldc` instruction or a `getstatic` instruction which loads the host name on which this array should be created (usually it will be "localhost"). Finally, the last instruction invokes the appropriate `VArrayWrapper` constructor to initialize the array. The `ArrayWrapper` constructor which is eventually invoked creates a real Java array of the type specified by the `<type-code>` value.

Creation with `anewarray`

The second Java bytecode instruction used to create arrays is the `anewarray` instruction. This instruction is used to create a single dimension array of objects. `anewarray` takes the top stack word, `n`, and creates an array of `n` elements whose component type is indicated by a reference, as part of the instruction, to a Class

```
new <VArrayWrapper-sub-class>
dup_x1
swap
ldc <base-type-string>
<load-object-host>
invokespecial
   <VArrayWrapper-sub-class>(ILjava/lang/String;Ljava/lang/String;)V
```

Figure 4.4: Instructions Replacing `anewarray`

entry in the constant pool.

As shown in Figure 4.4 (page 38), the code which emulates the `anewarray` instruction is very similar to the code which emulates `newarray`. Therefore, only the differences will be discussed below.

Instead of indicating the array's component type by a type code value passed to the `ArrayWrapper` constructor, the bytecode passes a "`<base-type-string>`". This is a Java string indicating the appropriate MobileJ class to use for components of the array. Usually this will be a proxy class (except where a proxy class is not used in MobileJ, as discussed previously). If the `anewarray` instruction is being used to create the first dimension of a multidimensional array, the component type will be an appropriate sub-class of `VArrayWrapper`. The `ArrayWrapper` constructor creates a real Java array of the type specified by the `<base-type-string>`.

Creation with `multianewarray`

The final instruction which can create arrays in Java bytecode is the `multianewarray` instruction, which is used to create a multi-dimensional array. `multianewarray` creates an array of `n` dimensions where `n` is indicated as part of the instruction. Also as part of the instruction is a reference to a Class constant pool entry indicating the

```
istore 255
istore 254
...
istore <255-(n-1)>
istore <255-n>
new <VArrayWrapper-sub-class>
dup
iload <255-n>
iload <255-(n-1)>
...
iload 254
iload 255
ldc <wrapper-base-type-string>
ldc <base-type-string>
<load-object-host>
invokespecial <VArrayWrapper-sub-class>(I1...In Ljava/lang/String;
                                        Ljava/lang/String;)V
```

Figure 4.5: Instructions Replacing `multianewarray`

type descriptor for the array being created. The instruction expects `n` integers on the top of the stack, each of which indicates the size of one dimension of the array.

To emulate this instruction using the `ArrayWrapper` class, the `multianewarray` instruction is replaced with the instructions shown in Figure 4.5 (page 39).

The `multianewarray` instruction is more awkward to emulate because at the point where the instruction is encountered in bytecode, `n` (where $1 \leq n \leq 255$) integer words are on top of the stack, one for each dimension of the array. These integers, representing the size of each array dimension, must be passed as arguments to a `VArrayWrapper` constructor, however, to do so, a reference to the uninitialized `VArrayWrapper` object must be placed beneath the integers.

The solution taken is to first store the `n` integers in local variables numbered from 255 downward, create the `VArrayWrapper` instance, and then load the `n` integers

back on to the stack. This assumes the method involved does not use more than (255 - n) local variable slots. It also assumes that **ArrayWrapper** has as many constructors of this type as the number of dimensions of the maximally dimensioned array instance that needs to be created. [2]

After re-loading the n integers, the instrumented code loads a "**wrapper-base-type-string>**" from the constant pool. This is a Java string which is used by the **ArrayWrapper** constructor in creating instances of the appropriate **VArrayWrapper** sub-class which will be elements of the created **ArrayWrapper** instance (because the created array is multidimensional). This string is the same as the **ArrayWrapper** sub-class name, but without the "**_<n>**" (number of dimensions prefix) which will be different for each dimension of the array. This string could be computed at run time by the **ArrayWrapper** constructor, but it is more efficient to compute it statically, at instrumentation time, and pass it recursively as the multi-dimensioned array is created.

The "**<base-type-string>**" passed to the constructor is the same as that which is passed when emulating the **anewarray** instruction, except in this case the string usually indicates another **VArrayWrapper** sub-class, which would be the type of elements in the second dimension of the array being created. The **ArrayWrapper** constructor recursively creates the n dimensions of the multi-dimensional array.

4.3.3 Access

The JVM has a number of instructions which are used to load elements from an array to the stack, and store elements from the stack to a particular array element. There is also an instruction to retrieve the length of an array. There are different

[2] In the current implementation of MobileJ, **ArrayWrapper** has enough constructors for a ten dimensional array, though more could easily be added, up to the required 255.

Element Type	Instruction	Method and Signature
integer	`iaload`	`getInt(I)I`
	`iastore`	`setInt(II)V`
long	`laload`	`getLong(I)J`
	`lastore`	`setLong(IJ)V`
float	`faload`	`getFloat(I)F`
	`fastore`	`setFloat(IF)V`
double	`daload`	`getDouble(I)D`
	`dastore`	`setDouble(ID)V`
object reference	`aaload`	`get(I)Ljava/lang/Object;`
	`aastore`	`set(ILjava/lang/Object;)V`
byte/boolean	`baload`	`getByte(I)B`
	`bastore`	`setByte(IB)V`
character	`caload`	`getChar(I)C`
	`castore`	`setChar(IC)V`
short	`saload`	`getShort(I)S`
	`sastore`	`setShort(IS)V`
	`arraylength`	`getLength()I`

Table 4.1: Java Array Access Instructions and Their Corresponding `ArrayWrapper` Methods

load and store instructions for each Java primitive type (except boolean values, which use the instructions for byte types), and for object reference elements.

To mimic the array access instructions, the `ArrayWrapper` class provides corresponding methods to load from and store to the underlying Java array which its instances wrap. It also provides a `getLength` method which may be used to get the size of the array. The Instrument tool replaces all array access instructions with an invocation of their corresponding `ArrayWrapper` method. Table 4.1 summarizes the methods which replace array access instructions.

4.4 Method Invocation and Return

4.4.1 Method Headers

In addition to a method's bytecode, various entities of a Java class file method table are changed when instrumented for use with MobileJ; these changes are described in this section.

Names

When the Instrument tool creates a proxy class, the names and signatures of its methods are made the same as its corresponding "real" class. However, as all proxy objects inherit from the class `VObject`, any method names which are the same as methods of `VObject` must be changed.[3] Method names which would potentially override methods of `VObject` are changed by prepending "`mj_`" to them in the instrumented class and its proxy class. Exceptions to this are the standard `toString` and `clone` methods which may be overridden in the proxy class.

One other exception to the method naming convention relates to Java exception classes and is described in Section 4.7.1, on page 58.

Signatures

Most of the objects handled by instrumented code are referenced through a proxy object, thus method parameter and return types are changed in the instrumented code. This requires changing the signature of all methods which accept or return objects, including arrays.

[3]This is not strictly true: actually, any methods with the same name *and* the same parameters must have their names changed in the instrumented class; otherwise, the methods in sub-classes would override the methods of `VObject`. For ease of implementation, however, the current Instrument utility only checks the method's name.

All Java primitive type descriptors in a method's signature are left unchanged as these parameters are passed by value. Array type descriptors are changed to their equivalent array proxy class type descriptor (as described in Section 4.3, page 31). All other object type descriptors are changed to descriptors of their equivalent proxy object type, except for those classes without proxy classes, in which case the descriptor is changed to one of the corresponding instrumented class. Descriptors for `java.lang.Object` are left unchanged.

Access Restrictions

Java class methods may be designated "`private`", "`protected`", "`public`", or left with no designation, meaning "package" level access for the method. In addition, methods may be either instance methods or static (class) methods.

MobileJ requires that all instance methods and constructors be made `public`. This comes as a direct result of using proxy objects for location transparent method invocation. Because a proxy object is of a different class than its corresponding "real" (instrumented) object, access to its methods cannot be restricted using ordinary method access flags, as with ordinary Java objects: the JVM doesn't know that the proxy objects should be regarded as being in the same "protection domain" as the non-proxy objects when determining if a method should be accessible from a certain context.

For example, if a method of class `foo` accepts "`bar`", an argument of class `foo`, this method is allowed to invoke any private instance method of the `bar` because it is of the same class as itself. However, when instrumented, the argument `bar` takes on the type `Vfoo` (the proxy class of `foo`); thus, to the JVM, its private methods are no longer accessible to the method. Making all instance methods public gets around this problem.

Exceptions Thrown

As Java exception classes are instrumented, the exceptions thrown by a method also have their names changed to those of the new, instrumented exception types.

4.4.2 Invocation and Return

General Cases

When instrumenting a bytecode instruction which invokes a method, the invoking instruction must be made to reference a new Methodref entry in the constant pool whose referenced class, method name, and method signatures have been changed according to MobileJ's conventions. These conventions have been previously described in sections 4.2.1 (page 30) and 4.4.1 (page 42), and will not be described here. Generally, this means that most method invocations are made to go through a proxy object, also described previously.

However, there are a number of special cases which must be handled for method invocation to function properly and follow the semantics of normal Java method invocation; these are described below.

Invocation and Return From Within an Instance Method

Most references to objects handled by instrumented code in MobileJ are to proxy objects. However, all instance methods, including those in instrumented classes, are implicitly passed a reference to the object which is the target of the invocation (the "`this`" parameter) by the JVM. Within instrumented classes, this object will never be a proxy object, and must be handled differently when being passed as an argument in further method invocations, used as the target of an invocation, or returned from the instance method. In the first and last instances, the reference to `this` must be converted to a reference to a proxy object (which, in turn, references `this`); in the second case, the method referenced must be in a "real" class and not a proxy class as with most method invocations.

Because static analysis of where the `this` reference is being used in method bytecode is quite difficult (and impossible in the general case), the Instrument utility inserts bytecode to perform run time checks for usages of `this` in cases where it can determine that the check is necessary.

When checking if `this` is passed as an argument to a method, the types of all method parameters, as specified in the method's signature, are check to see if they are the same as or super classes of the invoking method's class (i.e., the class of the `this` object). If any parameters could be a reference to `this` (based on their type), code to perform a run time check for `this` being passed as a parameter is inserted. Figure 4.6 shows the bytecode that is inserted immediately before the method invocation instruction.

The `checkForThisParam` method is a native static method in the MobileJ Runtime. This method scans the Java stack for references to `this` and replaces them with references to proxy objects by invoking `VObject.forObject` which creates an

```
ldc <stackBitField>
bipush <numWords>
invokestatic ubc/mj/MobileJava/checkForThisParam(II)V
```

Figure 4.6: Runtime Check For "this" As a Parameter: <stackBitField> and <numWords> are integers determined by the Instrument utility.

appropriate proxy object for a particular "real" object. The checkForThisParam method is dependant on the particular JVM implementation running MobileJ.[4] Because it needs quick access to words on the Java stack below the top one, this method is implemented in C, and is part of the MobileJ Native Library.

The <stackBitField> and <numWords> integer values are determined by the Instrument utility by analysing the parameters of the method being invoked. <stackBitField> is an integer bit-field specifying which words on the Java stack may hold references to the this parameter (as determined by the analysis of parameter types described above). The checkForThisParam method only checks words on the stack for which the corresponding bits of <stackBitField> are on. <numWords> indicates the maximum depth, in words, checkForThisParam should scan the stack looking for a reference to this. Using these parameters, the stack may be efficiently scanned for references to this passed in a method invocation.

Because a reference to this can also be returned from an instance method, the Instrument tool inserts code to ensure that these are converted to references to proxy objects when returned by the areturn instruction. Since the checkForThisParam method works perfectly well for this case as well, the code in Figure 4.6 is inserted immediately before the areturn instruction if the Instrument tool determines, by comparing the return type in the method's signature with

[4]The current version of MobileJ supports version 1.1.5 of Sun Microsystem's implementation of the VM on Solaris and a port of Sun's VM on Linux.

```
    bipush <n>
    invokestatic ubc/mj/MobileJava/checkForThisTarget(I)Z
    ifeq Label1
    invokevirtual <'this' object method>
    goto Label2
Label1:
    invokevirtual <proxy object method>
Label2:
```

Figure 4.7: Runtime Check For "this" As a Method Invocation Target

the method's class, that the reference returned could possibly be a reference to **this**. In this case, **<stackBitField>** and **numWords** are both equal to 1.

When checking if **this** is the target of a method invocation, the Instrument tool checks if the method is in the same class or a super-class of the current class. If this is the case, then it is possible that the target of the invocation (the object whose method is being invoked) is the current method's object, i.e., the **this** object. Figure 4.7 (page 47) shows the bytecode that replaces the method invocation instruction in this case.

As with the **checkForThisParam** method, the **checkForThisTarget** method is a native static method in the MobileJ Runtime. This method examines the Java stack word indicated by its integer argument, returning the boolean value **true** if this word is a reference to **this**, and **false** if not. The Instrument tool determines which word could be a reference to **this** by counting the number of words making up parameters according to the method's signature. The target of the invocation is one word below the first argument word.

If the target of the invocation is `this`, the first invocation instruction, which references a method in the same class as the current method will be executed. Otherwise, the second invocation instruction, which references a method in a proxy class, will execute.

Static Method Invocation

Static (or class) methods are invoked in much the same way as instance methods, except that they are not passed an instance of the method's class. As discussed in Section 2.2 (page 15), static method invocations need not go through a proxy class, but, rather, are called directly. Static methods are invoked with the `invokestatic` instruction, and each instance must have its referenced class and method signature changed by the Instrument tool. The tool changes referenced class names to their corresponding instrumented class names as well as changing the types of parameters according to the MobileJ conventions discussed earlier (see Section 4.2.1 for details).

Other Special Cases

A particular case of invoking within an instance method where the target is `this` happens when a constructor calls a super-class constructor. The Java Language Specification [GJS96] states that the first statement in a constructor must be a call to a super-class constructor, and even if it does not exist in the source code, a call to the super-class constructor with no arguments is inserted implicitly in the bytecode by the compiler.

The Instrument tool can thus determine when this is happening in a constructor's bytecode and ensure that the constructor in a MobileJ instrumented class is called, rather than a constructor in a proxy class as would happen without checking

```
    dup
    instanceof VObject
    ifne Label1
    invokestatic VObject/forObject(Ljava/lang/Object;)LVObject;
Label1:
```

Figure 4.8: Bytecode Inserted After an Invocation of `clone`.

for this special case.

Another case of method invocation which is handled as a special case when instrumenting bytecode is invocation of an object's `getClass` method. Because this method, when invoked on a proxy object, would return the real Java `Class` object representing the *proxy's* class, it must be changed such that an instance of `ubc.mj.java.lang.VClass` is returned instead.

The Instrument tool replaces the invocation of `getClass` with an invocation of the static method `forObject(Ljava/lang/Object;)Lubc/mj/java/lang/VClass;` in the instrumented version of `java.lang.Class`. This method is added to the instrumented version of `java.lang.Class` by the Instrument tool, as will be described in Section 4.7.4 (page 63).

A third case of method invocation handled as a special case is invocation of an object's `clone` method. The Instrument tool inserts code after the invocation of `clone` to ensure that the object returned is an instance of a proxy class; these instructions are displayed in Figure 4.8.

This code checks if the instance returned by `clone` is an instance of `VObject`, the super-class of all proxy objects. If not, it calls the static method `forObject` to convert the reference from a "real" Java object to a reference to a proxy object.

```
<load-object-host>
invokespecial VObject/<init>(...Ljava/lang/String;)V
dup
invokestatic ubc/mj/MobileJava/registerObject(LVObject;)V
```

Figure 4.9: Bytecode Instrumentation of Constructor Invocation

If the invocation is made to the `clone` method in `java.lang.Object` via an `invokespecial` instruction, the Instrument tool assumes that a clone of the current object (the `this` object) is created and omits the type-checking code, inserting the call to `forObject` directly after the call to `clone`.

The final special case of method invocation is invocation of an object's `toString` method. This will be discussed in Section 4.7.2 (page 59).

4.5 Object Creation and Initialization

Objects are allocated in Java bytecode by the `new` instruction. This instruction references a Class entry in the constant pool, describing the class of the object to be created. The Instrument tool changes this class to the appropriate proxy class when it encounters a `new` instruction.

After allocating a new object with `new`, one of the object's constructors is called to initialize the object. In addition to changes made to the bytecode because this is a method invocation (as described in Section 4.4.2, page 44), the Instrument tool must decide on what machine the object should be created, and pass a reference to the proxy object to the MobileJ Runtime system. Figure 4.9 shows an example of a constructor invocation after instrumentation.

If the type of object being initialized has been designated as being created at

either the client or server host, the `getstatic` instruction retrieves the appropriate host name and port number from the MobileJ Runtime. Otherwise the object is created on the current host, thus `<load-object-host>` becomes `ldc "localhost"`. The final parameter of a proxy object's constructor is always a string in which is passed the host name and port number of the location to create the object at.

Finally, the Instrument tool adds a call to the `registerObject` method in the MobileJ Runtime so that it can move the object in the future according to its object placement policy.

4.6 Field Access

4.6.1 Field Signatures

As MobileJ applications handle objects through proxy objects, the type signature of object fields, including arrays, must be changed. The modification of field type signatures is exactly the same as for that of method signatures, as described in Section 4.4.1 (page 42), i.e., most object types are changed to their corresponding proxy object type.

4.6.2 Accessor Methods

When the Instrument tool examines a class, it adds two accessor methods for every field (both static and non-static); these are used to set and retrieve field values as will be described in the next two sections.

Static fields:

```
getStatic_<class-name>_<field-name>
putStatic_<class-name>_<field-name>
```

Non-static fields:

```
getField_<class-name>_<field-name>
putField_<class-name>_<field-name>
```

Figure 4.10: Static and Non-Static Field Accessor Method Names

Field accessor methods are named according to the general form shown in Figure 4.10. Here `<class-name>` is the fully qualified class name of the class being instrumented with all periods (.) replaced with underscores (_). `<field-name>` is the name of the field.

4.6.3 Static Fields

The JVM has two instructions for accessing static fields: `getstatic` and `putstatic`. Because of the design of static data in MobileJ (Section 2.2, page 15), there are six different cases of use which are instrumented differently:

Initializing `putstatic` Of Object Fields

An "initializing `putstatic`" instruction is one which stores the initial value of a static field, and is contained in a class' static initializer method (named "`<clinit>`"). This method is run implicitly by the JVM when the class is loaded. Figure 4.11 shows the bytecode which replaces an instance of an initializing `putstatic` instruction which works with Object fields.

The initialization of static object data within a static intializer method happens in two steps: first the object is created and its constructor invoked. For an

```
    getstatic ubc/mj/MobileJava/isStaticHost
    ifne Label1
    pop
    getstatic ubc/mj/MobileJava/OTHER_HOST
    invokestatic getStatic_<class-name>_<field-name>()
Label1:
    putstatic <static-field>
```

Figure 4.11: Bytecode Replacing an Initializing `putstatic` Of Object Fields

instrumented class, this will (generally) create a reference to a proxy object on the top of the stack. Second, this reference is stored in a static variable using the `putstatic` instruction. This second step happens differently depending on if the static initializer is executing on the static host or not. (This is tested for using the first two instructions in Figure 4.11.)

If the static initializer is executing on the static host, the reference to the proxy object is copied to the static variable with `putstatic` just as it would be in non-instrumented code. However, if the static initializer is executing on the non-static host, the reference to the proxy object is popped off the stack and the proxy object for this static field is obtained from the static host by invoking a `getStatic_...` type accessor method. The returned proxy object, which references the "real" object created by the static initializer on the static host, is stored in the local static variable.

This design has the disadvantage that initialization of static objects happens twice, with one copy not used. However, given the inherent difficulties of analyzing the data flow of methods, this solution was chosen. It is difficult to determine, in the general case, where a reference to an object will eventually be stored in a method.

if needed {
```
ldc <stackBitField>
bipush <numWords>
invokestatic ubc/mj/MobileJava/checkForThisParam(II)V
```
}
```
dup
getstatic ubc/mj/MobileJava/OTHER_HOST
invokestatic putStatic_<class-name>_<field-name>(...)
putstatic <field>
```

Figure 4.12: Bytecode Replacing a Non-Initializing `putstatic` Of Object Fields

Initializing `putstatic` of Primitive Fields

Since primitive type fields are handled by value and not reference, their initialization is trivial. The Instrument tool does not need to change the `putstatic` instruction which initializes a primitive static field. In effect, these fields are replicated on each MobileJ host.

Non-Initializing `putstatic` of Object Fields

A `putstatic` instruction not within a static intializer method which operates on an object type field must be handled differently than if it were in a static intializer. The code which replaces such an access is shown in Figure 4.12.

The first three instructions in Figure 4.12 are the same as those used to check if `this` is being passed as a parameter in a method invocation. It is possible for them to be necessary here under the same circumstances as discussed in Section 4.4.2 (page 44) because the `putstatic` instruction is being replaced with a method invocation.

At the point of the `dup` instruction, the top stack word will be a reference to a proxy object. The remaining instructions store a reference to the proxy object on both the opposite MobileJ host, and the current host. First a `putStatic_...` style accessor method is invoked on the opposite host, storing a reference there, and

finally the `putstatic` instruction stores a reference to the proxy on the local host.

Non-Initializing `putstatic` of Primitive Fields

A `putstatic` instruction not within a static intializer method which operates on a primitive type field must be handled differently than if it were in a static intializer. The code which replaces such an access is shown in Figure 4.13.

This code simply invokes the appropriate `putStatic_...` style method on the static host to change the field value.

Object Type `getstatic`

Other than changing the type descriptor referenced in the instruction, instances of `getstatic` which operate on object type fields are left alone by the Instrument tool. Once initialized, static object type fields on either MobileJ host are proxy objects which refer to the same "real" object. Therefore, a reference to the proxy object may be obtained locally, without invoking an accessor method.

Primitive Type `getstatic`

The bytecode which replaces an instance of `getstatic` which operates on a primitive type field is shown in Figure 4.14. This code simply invokes the appropriate `getStatic_...` style method on the static host to retrieve the field value.

```
getstatic ubc/mj/MobileJava/STATIC_HOST
invokestatic putStatic_<class-name>_<field-name>()
```

Figure 4.13: Bytecode Replacing a Non-Initializing `putstatic` Of Primitive Fields

```
getstatic ubc/mj/MobileJava/STATIC_HOST
invokestatic getStatic_<class-name>_<field-name>()
```

Figure 4.14: Bytecode Replacing a `getstatic` Of Primitive Fields

4.6.4 Instance Fields

The JVM has two instructions for accessing object fields: `getfield` and `putfield`. These instructions must be replaced by method invocations so that field access can be made to go through a proxy object.

General Case

Generally, each instance of `putfield` is replaced with an invocation of a `putField_...` style accessor method. Similarly, each instance of a `getfield` is replaced with an invocation of the appropriate `getField_...` accessor method. In this way, field accesses are made to go through proxy objects.

There are two special cases which must be handled somewhat differently; these are explained below.

`putfield` with "this"

The first special case of instance field access has to do with the possibility of storing a reference to `this` in a field. The Instrument tool checks all instances of `putfield` within an instance method to see, based on the field type and type of `this`, if a reference to `this` could be stored in the field by the `putfield` instruction. If this is possible, the Instrument tool adds bytecode to perform a runtime check for this condition. This is similar to the check inserted to ensure that a reference to `this` passed in a method invocation is converted to a reference to a proxy object (see

```
    bipush <n>
    invokestatic ubc/mj/MobileJava/checkForThisTarget(I)Z
    ifeq Label1
    getfield <field>
    goto Label2
Label1:
    invokevirtual getField_...()
Label2:
```

Figure 4.15: Runtime Check For "this" As a Field Access Target

Section 4.4.2, page 44). In the case of putfield, the code in Figure 4.6 (page 46) is inserted just before the invocation of the appropriate putField_... accessor method.

Access of Field in "this"

If the Instrument tool determines from the class referenced in the putfield or getfield instruction that the field being accessed may be in the current, or this, object, some instructions must be inserted to check for this condition at runtime. This is done because access of fields of the current object can be done without going through a proxy object. This is similar to the check done before some method invocations where the invocation might go to the current object (see Section 4.4.2, page 44).

Figure 4.15 shows the bytecode replacing an instance of a field access where the access could possibly be to a field of this. The same MobileJ Runtime method, checkForThisTarget, is called to check if the word at location <n> is a reference to this. If it is, a getfield instruction is used to retrieve the field value, otherwise a getField_... method is called to retrieve the field value via a proxy object. This runtime check is also used for similar instances of putfield.

4.7 Miscellaneous Changes and Exceptional Cases

4.7.1 Exception Classes

As noted earlier, in Section 4.2.1 (page 30), Java exception classes (i.e., `java.lang.Throwable` and its subclasses) have corresponding instrumented versions created by the Instrument tool. However, proxy classes are not created for these classes because of the way they are used in Java applications. First, instances of exception classes are thrown by the `athrow` bytecode instruction which expects a `java.lang.Throwable` object to throw;[5] proxy classes cannot be sub-classes of `java.lang.Throwable`. Second, exception objects are typically used as short-lived, error status carrying objects, therefore there is little need for their location to be transparent to application programs.

When instrumenting the Java Class Library, the Instrument tool creates instrumented version of all exception classes, including `java.lang.Throwable`. Figure 4.16 shows the inheritance hierarchy of instrumented exception classes in MobileJ.

All instrumented exception classes inherit from their appropriate (instrumented) super-class, except `ubc.mj.java.lang.Throwable`, which inherits from `java.lang.Exception`.[6] This allows instances to be thrown with `athrow`.

[5]The VM used for developing MobileJ, a port of Sun's version 1.1.5 JDK, actually allows any type of object to be thrown, i.e., it does not do a runtime check of the object type when executing the `athrow` instruction. This would, however, cause bytecode verification to fail.

[6]These classes *could* inherit from `java.lang.Throwable`, however a limitation of the current MobileJ Distribution Layer implementation (i.e., Voyager) requires them to inherit from `java.lang.Exception`.

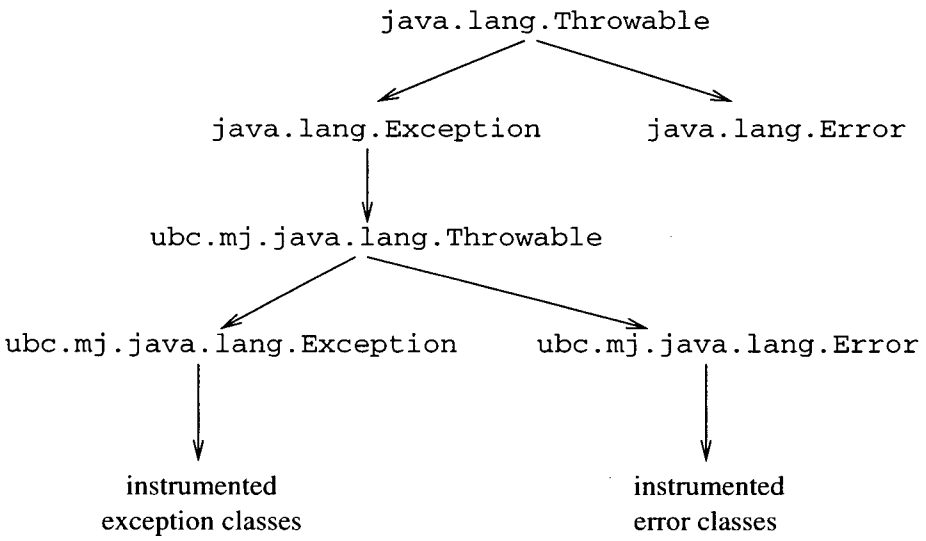

Figure 4.16: Class Hierarchy For Instrumented Exception Classes

To ensure that methods in `ubc.mj.java.lang.Throwable` don't override those in `java.lang.Throwable`, certain method names have their names changed by prepending "`mj_`" to them; these methods are: `fillInStackTrace`, `getLocalizedMessage`, and `getMessage`.

4.7.2 Strings

Introduction

While most classes instrumented for use with MobileJ have proxy classes, the instrumented version of `java.lang.String` does not. Since Java strings are immutable, they may be passed by value in method invocations which traverse machine boundaries without interfering with Java invocation semantics. Furthermore, string instances are often small, short-lived objects, and removing the need for invocations to go through proxy objects improves the performance of string handling code.

Though it does not require a proxy class, the string class *does* need to

be instrumented. The class `java.lang.String` contains methods with parameters, which, when instrumented, have their types changed to proxy classes. If `java.lang.String` was not instrumented, these methods would be incompatible with other instrumented bytecode.

It should be noted that while the class `ubc.mj.java.lang.String` does not have a corresponding proxy class, its underlying character array field is wrapped in an `VArrayWrapper` subclass, and thus accessed through a proxy object.

Conversion Methods

The Instrument tool adds two static conversion methods to the class `ubc.mj.java.lang.String` for converting between instances of `java.lang.String` and instances of `ubc.mj.java.lang.String`. These are used when loading string constants from the constant pool, when invoking `toString` methods, and in some native methods which accept string parameters.

The first two cases are described in the following sections; the use within native methods is primarily of the `toRealString` method, converting instances of `ubc.mj.java.lang.String` to instances of `java.lang.String`. This is done so that JNI functions may be used in MobileJ's Native Library which return the C "array of bytes" representation of a Java string instance.

The `forString` method is used to convert from a `java.lang.String` to a `ubc.mj.java.lang.String`. The signature and bytecode for this method is shown in Figure 4.17. This method works by converting the string to a character array, wrapping this character array in a proxy object, and invoking the appropriate constructor of `ubc.mj.java.lang.String`.

The `toRealString` method is used to convert from a

```
Signature:  (Ljava/lang/String;)Lubc/mj/java/lang/String;
Bytecode:   new  ubc/mj/java/lang/String
            dup
            new ubc/mj/array/V_1_C
            dup
            aload_0
            invokevirtual java/lang/String/toCharArray()[C
            ldc_w "localhost"
            invokespecial ubc/mj/array/V_1_C/<init>
                (Ljava/lang/Object;Ljava/lang/String;)V
            invokespecial ubc/mj/java/lang/String/<init>
                (Lubc/mj/array/V_1_C;)V
            areturn
```

Figure 4.17: `forString` Method Signature and Bytecode

`ubc.mj.java.lang.String` to a `java.lang.String`. The signature and bytecode for this method is shown in Figure 4.18. This method works by obtaining the underlying character array, offset value, and character count for the string, and passing them to the appropriate constructor of `java.lang.String`.

String Constants

The JVM has support for loading constant string values from the constant pool for a class. Using the `ldc` (or `ldc_w`) instruction and referencing a constant String entry in the constant pool causes the JVM to create a new instance of `java.lang.String` and place a reference to it on the stack.

These references to "real" Java strings must be converted to references to instances of `ubc.mj.java.lang.String`. This is accomplished by inserting an invocation instruction which calls the `forString` method described above.

Signature: `(Lubc/mj/java/lang/String;)Ljava/lang/String;`

Bytecode:
```
new java/lang/String
dup
aload_0
getfield ubc/mj/java/lang/String/value
invokevirtual ubc/mj/array/VArrayWrapper/getData
    ()Ljava/lang/Object;
aload_0
getfield ubc/mj/java/lang/String/offset
aload_0
getfield ubc/mj/java/lang/String/count
invokespecial java/lang/String.<init>([CII)V
areturn
```

Figure 4.18: `toRealString` Method Signature and Bytecode

`toString` Method Invocation

The way the method `toString`, which returns the string representation of an object, is handled in Java requires special attention in MobileJ. Instrumenting this method in the normal way causes its return type to be changed to `ubc.mj.java.lang.String`; this causes a conflict with `toString` as defined in `java.lang.Object`, from which all classes inherit. [7]

Therefore, the return type for all `toString` methods instrumented for use with MobileJ is kept as a `java.lang.String`. This requires a call to `toRealString` to be inserted in all `toString` methods to convert its result to a "real" Java string, and a call to `forString` to be inserted after all invocations of `toString` to convert the returned string back to a MobileJ string.

[7] This is because a method in a sub-class cannot have the same name and parameter types while also having a different return type as a method in a super-class; one cannot overload a method by only changing its return type.

4.7.3 Type Checking Instructions

Two bytecode instructions, which have not previously been mentioned, also need to be instrumented for use with MobileJ: `checkcast` and `instanceof`. Both of these instructions reference a Class entry in the constant pool; the Instrument tool changes the class referenced such that it is the equivalent class used under MobileJ, which is usually a proxy class.

4.7.4 `java.lang.Class` and Reflection

The capabilities in Java for reflection require special attention in order to function correctly under MobileJ. The current implementation of MobileJ supports a certain amount of the reflection API found in the `java.lang.Class` and `java.lang.reflect.Constructor` classes, but this could be extended to fully support the use of reflection under MobileJ. The current level of support is necessary in order to allow the `java.lang.System` class to load and basic applications to run.

All loaded Java classes have a corresponding instance of `java.lang.Class` which represents them and can be used to invoke reflection capabilities. The native methods which support reflection under MobileJ must also use `Class` objects to reference classes as requested by MobileJ applications which use the Java reflection API. However, to be consistent, instrumented MobileJ application code should only reference instances of the proxy class `ubc.mj.java.lang.VClass`, not "real" `java.lang.Class` objects. Thus, during instrumentation of `java.lang.Class`, a private instance field, named `realClass` and of type `java.lang.Class`, is added in order to "wrap" the real `Class` object. The native methods implementing reflection under MobileJ know about this field and use it to invoke the underlying Java reflection API as needed.

The `realClass` field is assigned whenever a method that returns a `Class` object is invoked. The most commonly used method for this purpose is the static method `Class.forName` which returns an instance of `Class` given a particular class name, passed as a string. Since this is a native method, it needed to be reimplemented for the corresponding instrumented version of class `Class`.

However, if the new version of `forName` returned the `Class` object indicated by its string argument, it would become possible for the program to create an instance of this class, which would not be an instance of a proxy class. Therefore, the new version of `forName` first converts the name passed to it into the equivalent MobileJ proxy class name, as described in Section 4.2.1. A `java.lang.Class` object representing the proxy class is then created.

A new instance of `ubc.mj.java.lang.Class` is also created and the real `Class` object assigned to its `realClass` field. Finally, a proxy instance referring to the `ubc.mj.java.lang.Class` object is then created, a reference to which is returned by `forName`.

A similar scheme is used to implement the MobileJ equivalent of the `java.lang.reflect.Constructor` class; a field named `realConstructor` of type `java.lang.reflect.Constructor` is added while instrumenting the `Constructor` class. It is believed this method of "wrapping" real Java reflection classes in their instrumented versions will work for other reflection classes, and thus fully support reflection under MobileJ.

4.7.5 `java.lang.Character` Class Initializer

During development of MobileJ it was discovered that normal instrumentation of the static initializer method of class `java.lang.Character` produced a method whose

bytecode was greater than 65535 bytes in length, larger than the maximum allowed in the class file format specification. As a consequence, the JVM would refuse to run this method. The method is so large because this class contains three rather large private static arrays and all static data in Java is initialized through code in the class' static initializer; the more static data for a class, the larger the class initializer.

To work around this limitation, the static initializer in the instrumented version of `java.lang.Character` is changed so that it simply calls a method in the MobileJ Runtime. If the method is run on the static host, it wraps a reference to the instance of each array in `java.lang.Character` [8] in an appropriate `VArrayWrapper` instance and assigns this to the array field in `ubc.mj.java.lang.Character`. Thus the actual primitive type data for these arrays is shared between the real and instrumented versions of `java.lang.Character`; this works because these arrays are only read from and not written to once initialized.

If run on the non-static host, this method retrieves a reference to the field on the static host, i.e., a proxy object, and assigns it to the field on the non-static host.

4.7.6 `java.lang.Runtime loadLibrary` Method

To simplify the implementation, the instrumented version of the `java.lang.Runtime.loadLibrary` method is turned into a null method which simply returns when called. Since applications which rely on native code won't work with MobileJ anyways, this method does not need to be supported. Code in the instrumented versions of the standard Java libraries which calls this method

[8] Because they are private fields, these references can only be retrieved through use of a native method.

doesn't require it to do anything either, as all native methods for these classes are contained in the MobileJ Native Library, loaded when it starts up.

4.8 Creating Proxy Classes

In addition to creating instrumented versions of class files, the MobileJ Instrument tool must create proxy classes corresponding to each instrumented class and `ArrayWrapper` subclass. The current implementation uses Voyager's `vcc` tool for creating proxy classes, but a custom tool could also be created if Voyager were replaced by a custom MobileJ Distribution Layer. (See Section 6.2.2 on page 83 for discussion of replacing Voyager.)

Proxies are created according to the following format: [9]

- the class naming conventions described in Section 4.2.1 are followed, e.g., the proxy of class `java.util.Vector` is named `ubc.mj.java.util.VVector`.
- the class hierarchy for proxy classes mirrors the class hierarchy of instrumented classes, e.g., since `java.util.Stack` inherits from `java.util.Vector`, the proxy class `ubc.mj.java.util.VStack` inherits from `ubc.mj.java.util.VVector`.
- a proxy class implements the same interfaces as the instrumented class it is a proxy for
- the same exceptions thrown by methods, including constructors, in an instrumented class are thrown by methods in its associated proxy class

[9] These specifications closely follow the format of Voyager "virtual classes" [Obj97, pages 46-47] because the current implementation of MobileJ uses Voyager as its Distribution Layer.

- for every instance method in the instrumented class, the proxy class contains an instance method of the same name, and with the same type signature
- for every constructor in the instrumented class, the proxy class contains a constructor with the same type signature and an additional `String` parameter, used to designate the MobileJ host and port number on which to construct an instance of the class
- both static and instance fields, as well as static methods [10] do not have an equivalent in MobileJ proxy classes because fields are accessed through accessor methods, as discussed in Section 4.6, page 51

Since proxy classes form part of MobileJ's Distribution Layer, a description of what functionality they must implement is found in Section 3.2, page 23.

4.9 Implementing Bytecode Instrumentation

Creating a tool which instruments Java bytecode is challenging for a number of reasons; two issues in particular are described below.

4.9.1 Maintaining Flow Control

Branch instructions in Java bytecode indicate branch targets using relative offsets; inserting or deleting instructions, as required by MobileJ, requires re-setting these offset values for instructions affected.

To implement this, an `InstructionVector` class was created which modelled method bytecode as a vector of instructions. As with standard Java `Vectors`,

[10] Voyager creates equivalent static methods when it creates its "virtual classes", but these are not used in the current implementation of MobileJ.

`InstructionVectors` can have individual elements (instructions) added, changed, or removed. They can also emit a version of the vector as an array of bytes, for inclusion in a class file. When this final version of the method bytecode is requested, the `InstructionVector` re-sets all relative offsets.

The `InstructionVector` accomplishes this by maintaining a mapping between original code position and new code position in the instruction vector. Thus all new relative offset values can be determined by looking up the equivalent old absolute offset, retrieving the new absolute offset, and deriving the new relative offset given the current instruction position.

However, if code is inserted before the target of a branch it is sometimes the intention of the instrumenting code to have the target be re-directed to the start of the newly inserted code. To handle this case, every map entry has a `previous` field, which is either null or references the previous instruction in the `InstructionVector`. When inserting an instruction, instrumenting code can choose to have the `previous` field set for the instruction being inserted before. This re-directs branch targets to the newly inserted instruction.

Using this design also requires that, for a block of instructions being inserted, the instructions be inserted in reverse order of their execution in bytecode in order for the `previous` fields to be linked in a chain. This ensures that the first instruction in the block will become the new target of any branches originally intended for the instruction the block was inserted before.

Bytecode offset values in a method's exception handler table are also changed according to the mapping maintained in the `InstructionVector`.

4.9.2 Debugging Support

Due to the relative immaturity of Java in general, there do not exist many tools (especially freely available tools) for debugging Java bytecode. It would be much easier to debug a tool such as the MobileJ Instrument utility if a debugger which could control Java execution at the bytecode level were available. Most available debuggers, including Sun's `jdb` utility, are meant primarily for source level debugging, and don't allow for much control at the bytecode level.

To aid debugging of MobileJ an option to output method call tracing was added to the MobileJ Runtime and Instrument tool. When turned on, the Instrument tool inserts calls to methods in the MobileJ Runtime at the start of each method and before any return instructions in the method. The MobileJ Runtime methods output tracing information to a file, showing the stack depth and the method being entered or returned from, similar to what is output when Sun's implementation of the JVM is started with method tracing enabled.. (The trace output directly from the JVM was not useful because it traced not only instrumented methods but also methods of classes in the MobileJ Distribution Layer, making these traces difficult to understand.)

Two tools which were of great help when debugging MobileJ's instrumentation were a bytecode disassembler and assembler. Using the disassembler, the Java "assembly" version of instrumented bytecode code could be created, and thus edited by hand in a text editor. This allowed one to insert code to print messages, including variable values, to the screen (or a file). The assembly code could then be run through the assembler to produce a debugging version of the original class file. Though tedious, this was often the only way to debug instrumented bytecode. However, a debugger with good bytecode level support would remove the need to work

with bytecode disassemblers and assemblers when debugging instrumented code.

4.10 Summary

In order to facilitate location transparent access to objects, class files are instrumented such that nearly all objects manipulated by a MobileJ application are proxy objects. These proxy objects redirect all method invocations to the "real" objects which they represent, either on the local machine or a remote one. This way, the MobileJ Runtime can move objects at will between machines without the application's knowledge.

However, since both arrays and fields are handled directly by JVM instructions, access to them is transformed into method invocations through proxy objects, allowing for location transparent access by the same mechanism used for explicit method invocation.

The MobileJ Instrument tool processes class files, including bytecode instructions. While it is possible to imagine a tool that could similarly process other executable formats, the large amount of symbolic information present in class files makes a project such as this more feasible. The class file format and JVM require storage of symbolic information primarily to support dynamic code linking, but this design also allows for other applications, of which the Instrument tool is one example. (See Section 5.1, page 71 for other examples.)

Chapter 5

Related Research

5.1 JVM Class Instrumentation Tools

At least three other tools have been developed which allow instrumentation of JVM class files; these are described below.

5.1.1 Bytecode Instrumenting Tool (BIT)

BIT, as described in [LZ97], is a toolkit for instrumenting Java bytecode. BIT was created as a tool to aid analysis of dynamic program execution behaviour. As such, it provides the ability to insert invocations of user-supplied static methods at particular points in method bytecode, for example, at the beginning and end of a method or basic block, or after certain types of instructions. This type of instrumentation is very useful for implementing various forms of code profiling, but BIT could not be used as a tool to implement the instrumentation needed for MobileJ because it does not support changing of the JVM class file structure, references to constant pool entries, or insertion of arbitrary instructions. BIT is meant to be used in situations

where the semantic behaviour of an instrumented program is preserved.

5.1.2 Java Object Instrumentation Environment (JOIE)

The JOIE system, as described in [CC98], is a tool for JVM class file instrumentation which can perform load-time transformation of class files. JOIE is implemented as a Java class loader instance, and all instrumentation is performed at class load time, before the JVM begins interpreting any bytecode of the class. Users of JOIE create *transformers*, Java classes implementing the interface `joie.ClassTransformer` which operate on class files when called upon by the JOIE class loader.

The capabilities of JOIE are more extensive than those of BIT, allowing for changes to every aspect of class file structure and bytecode instructions. JOIE even allows for changes to a method's frame, or local variable storage area, by adding and removing method parameters and local variables. It also deals with resetting of relative branch targets and exception handler ranges (as does the MobileJ Instrument utility).

JOIE could be used to implement the instrumentation required by MobileJ, however, doing so would require a high overhead (for instrumentation) each time a MobileJ application was run, without providing many inherent advantages. The extensive changes required of class files for use with MobileJ makes static instrumentation the best option.

5.1.3 Binary Component Adaptation

A prototype Binary Component Adaptation (BCA) system for JVM class files is described in [KH98]. This BCA system works very similarly to JOIE, transforming class files at load time. However, the BCA system uses *delta files* and associated delta

file compiler instead of transformer classes to customize the code transformation. Delta files describe the types of transformations required (using a specific syntax) which the compiler converts to JVM class files for loading and invocation by the BCA's custom class loader.

This BCA system does not appear to support adding or removing arbitrary bytecodes from a method, therefore it would not be suitable for implementing the type of bytecode instrumentation required by MobileJ. BCA is intended to support less comprehensive modifications to class files than those required by MobileJ.

5.2 Systems Supporting Mobile Applications

Many systems have been developed to assist application developers in creating programs for the mobile computing environment. Some are implemented purely at the traditional operating system level, while others are frameworks for building partitioned applications.

As described in Section 1.2.2 (page 5), systems can be classified based on whether they implement a static or dynamic application partition, and to what extent the system attempts to hide the constraints of the mobile computing environment from the application; a sampling of systems are described below.

5.2.1 Statically Partitioned Systems

Mobile-Aware Systems

Odyssey [Nob98, NSN+97, NPS95] is a system supporting mobile information access, created at Carnegie Mellon University. Specifically, it is an extension of the NetBSD (Unix) file system supporting file access from mobile computers. The

Odyssey project examines issues such as consistency of mobile data, coordinated management of resources on a mobile host, and application agility, or, the ability of the application to adapt to changes in resource availability. While the system allocates resources and monitors their availability, it assumes each application provides functionality for adapting to changes in resource constraints. Odyssey notifies applications when resource constraints change, for example, when the available network bandwidth decreases. Thus applications cooperate with the operating system in adapting to the varying mobile environment, and makes Odyssey applications mobile-aware.

Another statically partitioned, mobile-aware architecture is presented by Welling and Badrinath in [WB98, WB97]. This system focusses on providing an appropriate framework for delivering environment related events to applications executing on mobile computers. As with Odyssey, events associated with changes in resource availability are delivered to applications using this framework, where they are handled in an application specific way.

While not explicitly required by their implementation, Welling and Badrinath suggest isolating the modules which implement an application's environmental adaptation policies, and avoid incorporating adaptation code in all modules, resulting in a form of "spaghetti code". They note that not only does this result in a more maintainable application, but also that it eases reengineering of existing applications to support functioning in a mobile environment. Prototypes of this architecture have been implemented in Mach 3.0 and Java [WB98].

Mobile-Transparent Systems

Statically partitioned, mobile-transparent systems are most easily implemented at either some layer of the user-interface, or at the filesystem or network access operating system interface, corresponding with the left and right extremes, respectively, of Figure 1.1 (page 6).

Implementations of the X-Windows protocol have been adapted to function in the low-bandwidth, high-latency environment typical of wireless networks. However, the X-Windows network protocol and architecture were noted in [KDF+93] as being particularly difficult to adapt to a wireless environment because of the bandwidth and latency constraints: they concluded by remarking, "We are not at all convinced that using X for pen-based mobile/wireless computing is a good idea."

Further work, described in [Dan94], shows a compression ratio of 7.5:1 is possible for the X-Windows protocol, yielding, for example, a six second transfer time when viewing a text document with Ghostview. The author notes that this is likely to improve as the compression is tuned, however, it remains that the X-Windows protocol was designed in the context of relatively high-bandwidth, low-latency Ethernet networks. Highly interactive applications, especially those with graphical user interfaces are unlikely to achieve acceptable performance using X over wireless networks.

Implemented at a slightly different layer of the user-interface, the Virtual Network Computer system [WRB+97, RSFWH98] operates by transmitting an encoded version of a user's screen from a server to a thin client (the Java client is a 20Kb applet). The system is intended to promote access to a user's "home computing environment" from a variety of computing devices.

Two limitations of the VNC system appear to be client computational power,

and available network bandwidth and latency–basically the same problems noted when using X-Windows over poor links, although a client *is* currently being developed for the 3COM Palm Pilot, connected by a wireless modem [Min98]. While VNC may be appropriate for use by mobile users with access to many wirelessly connected workstations in the same metropolitan area, it does not appear workable when the bandwidth and latency restrictions of the network being used approach those of wireless networks.

Systems which function at the filesystem and network OS interface include the Mobile Application Framework (MAF) described in [HR97], and the Advanced Mobile Integration in General Operating Systems (AMIGOS) project [HRAJ98, HR96, GM95]. For network communication, MAF optimizes TCP network connections by replacing recognized application level protocols, for example "ftp", with an optimized data transfer protocol over the wireless portion of a TCP connection. The AMIGOS "Split-Connection TCP" design [HRAJ98, HR96] allows for transparent changing of the link-layer connection without disconnection of the TCP layer connection. Therefore a mobile host can move between a wireless network and a wired LAN without disrupting active TCP connections.

5.2.2 Dynamically Partitioned Systems

Mobile-Aware Systems

Wit [Wat94b, Wat94a, Wat95] was one of the earliest systems designed to support mobile applications through partitioning an application between a small, mobile client and server. The Wit system provides a Tcl interpreter on both the mobile host (HP 100LX palmtops running DOS 5.0) and proxy server host. The Wit API provides for remote execution of Tcl scripts from both the mobile unit and proxy

application, thus allowing dynamic partitioning though runtime decisions about where a procedure should execute, but the intention seems to be to have most application functionality be statically partitioned.

In [Wat95], a scheme for describing application data in terms of hyperobjects is presented; data, for example, world-wide-web pages, could be presented to the system as a set of hierarchically linked hyperobjects. The system would use this structure to make decisions about which parts of data requested at the mobile host to transfer, and also which to prefetch.

Similar to Wit, Rover [JTK97, JK96, Tau96, JdT+95] is an application development framework designed with the constraints of the mobile computing environment in mind. As with Wit, designing a Rover application, or porting an application to use Rover, involves splitting application functionality between the mobile and fixed hosts, and defining application data in terms of *relocatable data objects* (RDOs) which can be passed between the mobile and fixed hosts. The two halves of the program communicate via a *queued remote procedure call* protocol, which provides for queuing and logging of asynchronous RPC requests while the mobile host is disconnected, and flushing of the log to the fixed host upon re-connection. Applications can decide where their various RDOs should be placed (at the mobile or fixed host) and when they should be moved, however, it appears that the partition between mobile and fixed host is intended to be fairly static for most applications.

Sumatra [ARS97, RASS97, RAS96] is a distributed object system based on Java designed for building resource-aware applications. Sumatra extends Java with the ability to group objects, move object groups between hosts, create threads on remote hosts, and migrate threads between hosts. These capabilities are coupled with a distributed resource monitor which allows applications to be notified regard-

ing resource constraints such as network latency and bandwidth, and host CPU load (although the most recently described implementation monitors only network latency). Although its developers do not appear to have targeted applications deployed to mobile computers, applications built with Sumatra have the ability to adapt to changes in their computational environment through moving parts of (or all of) the application to different hosts, for example, to minimize latency between hosts for some interaction. The designers describe an Internet chat server which can migrate between hosts based on the latency observed between it and client hosts, in an attempt to optimize its location for all users.

Mobile-Transparent Systems

M-Mail [Lo97, LK96], an email system designed for use in the mobile environment, serves as a case study on dynamic, mobile-transparent application partitioning. M-Mail uses the Mentat Programming Language (MPL), a language based on C++, and the Mentat Run-Time System (MRTS) for distributed object creation and invocation. M-Mail also makes use of a specific API [NKB96] for monitoring network conditions (e.g., bandwidth, latency, error rate) at run time.

The MPL and MRTS support creation of large grained objects, each created as a separate Unix process. M-Mail creates a separate Mentat object for each folder of email messages a user has. The objective of the M-Mail project was to derive an appropriate object placement algorithm based on run time network conditions. As the MRTS does not have support for object migration, all placement decisions are made at object creation time (when the M-Mail application starts up), however, a history of the number and frequency of object invocations for each object (i.e., email folder) is recorded and stored between executions of M-Mail.

Although M-Mail is one of few studies concentrating on deriving appropriate object placement policies for the mobile computing environment, it is difficult to see how the large grained objects used in the MRTS can be used to compose other applications. Further study is also needed to determine appropriate algorithms for use when object migration is available.

Though not designed for use specifically in a mobile-computing environment, Coign [HS98b, HS98a, HS97] is an implementation of application partitioning for Microsoft COM based applications. Like MobileJ, Coign combines application binary instrumentation with runtime distribution control, to produce a system which can partition an application without access to application source code.

Using Coign involves instrumenting an application binary, adding profiling code which records all COM component instantiations and inter-component function calls. The application is then run (on a single machine) through any number of profiling scenarios, where information is collected by the instrumented version of the application. After the profiling scenarios have been run, a separate application analyses the collected profiling data, and determines an optimum configuration based on the number of bytes transferred between components, the number of machines in the distributed system, and possibly other factors, such as network condition between hosts. The analysis involves creating a connected graph consisting of component instances as vertices and inter-component communication cost as edges, and cutting the graph so as to minimize communication between machines. For a client-server configuration, the analysis takes into account that certain components (e.g., GUI controls) must always be placed on the client. The application is then run, with the Coign runtime system instantiating a distributed configuration of the application, based on the results of the graph partitioning analysis.

The most difficult task for the Coign runtime during distributed execution is matching component instantiations with those recorded during profiling, in order to achieve the optimum distribution found during analysis. Since Coign separates profiling execution from distributed execution, if the distributed execution varies too greatly from the scenario profiled, Coign will not be able to achieve an optimum distribution. The advantage of this design, however, is that the runtime system has to do less work during distributed execution, exacting only a 3% overhead during these executions [HS98a]. The use of static profiling also means that Coign is unable to migrate component instances once created; all configuration decisions are made statically, at analysis time.

Chapter 6

Conclusions

6.1 Summary

The MobileJ system is designed to support dynamic partitioning of JVM applications between a smaller, less powerful mobile host and a larger, more powerful stationary host. By using class file (object code) instrumentation, MobileJ hides the details of partitioning from the application programmer, making the system mobile-transparent.

The MobileJ system consists of an Instrument tool and Runtime, including a native library. The Instrument tool transforms class files such that their bytecode manipulates "proxy" objects instead of "real" objects. Method invocations are made to go through these proxy objects, which redirect them to the application objects they represent. In this way, application objects may be placed on either the mobile host or the stationary host under runtime control, their location being transparent to the instrumented application code.

The MobileJ Runtime handles application startup, and is responsible for

maintaining a partitioning policy. While the current implementation partitions applications based on object type only, creating a static partition, the Runtime may be extended by adding an object placement policy module. This module would maintain references to all application objects and make decisions on where objects should be placed and under what circumstances.

It is intended that MobileJ serve as a basis for experimentation and further research into object placement policies for mobile applications. While MobileJ provides the mechanism for dynamic partitioning of applications, it is still unclear how to best partition applications in the mobile environment.

6.2 Future Work

As with most systems research projects, there is much that could be done to extend the work presented here. This section describes a number of possible extensions for future work.

6.2.1 Porting AWT Native Methods

The current implementation of MobileJ does not support running Java applications utilizing a graphical user interface (GUI), unfortunately perhaps the majority of applications written with Java. To do so, the native methods implementing Java AWT "peers" need to be ported to work with MobileJ. This is necessary because the native methods manipulate objects, and are not expecting proxy objects to be passed as parameters.

This task is more tedious than it is difficult, as all method invocations and field accesses from native code must be changed such that they work with the rest of MobileJ, i.e., they access proxy objects instead of "real" objects. The Instrument

tool only processes Java bytecode; native method code must be changed "by hand". Porting the AWT to work with MobileJ would allow most Java applications to be run under MobileJ.

6.2.2 More Efficient Distribution Layer

The performance of MobileJ is primarily dependent on the efficiency of the MobileJ Distribution Layer (see Section 3.2). The current implementation, utilizing Voyager [Obj97], is not very efficient because Voyager is not optimized for local method invocations. A new implementation, built specifically with the requirements of MobileJ in mind and optimized for local invocations, could perform much better.

6.2.3 Dynamic Monitoring and Object Placement Policy Modules

MobileJ currently lacks modules to monitor the computational environment and object interactions, and an object placement policy module. The environment monitoring module would watch such environmental factors as CPU load and available network bandwidth and latency, providing an indication of the current computational and network environment to the policy module (as in the M-Mail system described in Section 5.2.2, page 76). The object interaction module would keep track of the frequency of interaction as well as the amount of data transferred between objects, also providing this data to the object placement policy module.

The object placement policy module would combine data from the two monitoring modules to optimize object placement for a running application. The policy module would work similarly to Coign's analysis program (see Section 5.2.2), creating a connected graph of objects in the application, except that the complete graph would be distributed between the mobile client and fixed server and would be up-

dated continuously while the application is running. Ideally the placement policy could be configured by describing a relative weighting between environmental factors and object characteristics (size, invocation frequency and bandwidth) in order to facilitate experimentation with different applications and policies.

6.2.4 Study of Applications and Object Placement Policies

Given the modules described in the previous section, a study of a variety of applications and object placement policies would be enlightening. While similar in flavour to the M-Mail study [Lo97], such an investigation would show where different types of applications should be partitioned and under what circumstances. It might also attempt to determine which factors should have the greatest influence on object placement policies, and derive a well-performing object placement algorithm. If mobile computers are to become more ubiquitous, a greater understanding of how particular types of applications should be built for use in this environment needs to be achieved.

6.3 Final Conclusions

The MobileJ system provides a basis for further investigation of dynamic, mobile-transparent application partitioning for the mobile environment. While the benefits of having completely transparent, runtime controlled object placement for mobile applications remains to be verified, MobileJ is a step in this direction. It is hoped that further investigations may reveal the extent to which runtime systems can adapt to changes in the mobile environment through dynamically monitoring application behaviour.

Bibliography

[ARS97] Anurag Acharya, M. Ranganathan, and Joel Saltz. Sumatra: A Language for Resource-aware Mobile Programs. In J. Vitek and C. Tschudin, editors, *Mobile Object Systems*, volume 1222 of *Lecture Notes in Computer Science*. Springer Verlag, 1997. http://www.cs.umd.edu/~acha/papers/lncs97-1.html.

[Bha97] Vaduvur Bharghavan. Challenges and Solutions to Adaptive Computing and Seamless Mobility over Heterogeneous Wireless Networks. *International Journal on Wireless Personal Communications*, March 1997. http://timely.crhc.uiuc.edu/Papers/Abstracts/CS_abstract.html.

[CC98] Geoff A. Cohen and Jeffrey S. Chase. Automatic Program Transformation with JOIE. In *Proceedings of the USENIX Annual Technical Conference*, New Orleans, LA, USA, June 1998.

[Dan94] John Moffatt Danskin. *Compressing the X Graphics Protocol.* PhD thesis, Princeton University, November 1994.

[Dea98] Alan Dearle. Toward Ubiquitous Environments for Mobile Users. *IEEE Internet Computing*, 2(1):22–32, Jan-Feb 1998.

[Duc92] Dan Duchamp. Issues in Wireless Mobile Computing. In *Proceedings of the Third IEEE Workshop on Workstation Operating Systems*, pages 2–10, April 1992.

[DWJ+96] Tzvetan T. Drashansky, Sanjiva Weerawarana, Anupam Joshi, Ranjeewa A. Weerasinghe, and Elias N. Houstis. Software Architecture of Ubiquitous Scientific Computing Environments for Mobile Platforms. *ACM Mobile Networks and Applications*, 1(4), 1996.

[Fla97] David Flanagan. *Java in a Nutshell.* O'Reilly and Associates, Sebastopol, CA, USA, 2nd edition, May 1997.

[FZ94] George H. Forman and John Zahorjan. The Challenges of Mobile Computing. Technical Report UW-CSE-93-11-03, Department of Computer Science and Engineering, University of Washington, Seattle, WA, USA, March 1994.

[GJS96] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification.* Addison-Wesley, Reading, MA, USA, 1996.

[GM95] Vitor Guedes and Francisco Moura. Replica Control in MIo-NFS. In *Proceedings of the ECOOP 1995 Workshop on Mobility and Replication*, Aarhus, Denmark, August 1995.

[HR96] Jørgen Svaerke Hansen and Torben Reich. Semi-Connected TCP/IP in a Mobile Computing Environment. Master's thesis, Department of Computer Science, University of Copenhagen, Copenhagen, Denmark, June 1996.

[HR97] Stefan G. Hild and Peter Robinson. Mobilizing Applications. *IEEE Personal Communications*, 4(5):26–34, October 1997.

[HRAJ98] Jørgen Svaerke Hansen, Torben Reich, Birger Andersen, and Eric Jul. Dynamic Adaptation of Network Connections in Mobile Environments. *IEEE Internet Computing*, 2(1), Jan-Feb 1998.

[HS97] Galen C. Hunt and Michael L. Scott. Coign: Efficient Instrumentation for Inter-Component Communication Analysis. Technical Report 648, Department of Computer Science, University of Rochester, Rochester, NY, USA, February 1997.

[HS98a] Galen C. Hunt and Michael L. Scott. A Guided Tour of the Coign Automatic Distributed Partitioning System. Technical Report MSR-TR-98-32, Microsoft Research, Microsoft Corporation, Redmond, WA, USA, July 1998.

[HS98b] Galen C. Hunt and Michael L. Scott. The Coign Automatic Distributed Partitioning System. Technical Report MSR-TR-98-40, Microsoft Research, Microsoft Corporation, Redmond, WA, USA, August 1998.

[JdT+95] Anthony D. Joseph, Alan F. deLespinasse, Joshua A. Tauber, David K. Gifford, and M. Frans Kaashoek. Rover: A Toolkit for Mobile Information Access. In *Proceedings of the 15th ACM Symposium on Operating System Principles*, Copper Mountain Resort, Colorado, USA, December 1995.

[JK96] Anthony D. Joseph and M. Frans Kaashoek. Building Reliable Mobile-Aware Applications using the Rover Toolkit. In *Proceedings of the Second ACM International Conference on Mobile Computing and Networking*, November 1996.

[JTK97] Anthony D. Joseph, Joshua A. Tauber, and M. Frans Kaashoek. Mobile Computing with the Rover Toolkit. *IEEE Transactions on Computers*, 46(3), March 1997.

[Kat94] Randy H. Katz. Adaptation and Mobility in Wireless Information Systems. *IEEE Personal Communications*, 1(1):6–17, 1994.

[KDF+93] Christopher Kent Kantarjiev, Alan Demers, Ron Frederick, Robert T. Krivacic, and Mark Weiser. Experiences with X in a Wireless Environment. In *Proceedings of the USENIX Mobile and Location-Independent Computing Symposium*, Cambridge, Massachusetts, USA, August 1993.

[KH98] Ralph Keller and Urs Hölzle. Binary Component Adaptation. In *Proceedings of the 12th European Conference on Object-Oriented Programming*, Brussels, Belgium, July 1998.

[LK96] Hai Yan Lo and Thomas Kunz. A Case Study of Dynamic Application Partitioning in Mobile Computing–An E-mail Browser. In *Proceedings of the Workshop on Object Replication and Mobile Computing of OOPSLA 1996*, San Jose, CA, USA, October 1996.

[Lo97] Hai Yan Lo. M-Mail: A Case Study of Dynamic Application Partitioning in Mobile Computing. Master's thesis, Department of Computer Science, University of Waterloo, Waterloo, ON, Canada, 1997.

[LZ97] Han Bok Lee and Benjamin G. Zorn. BIT: A Tool for Instrumenting Java Bytecodes. In *Proceedings of the USENIX Symposium on Internet Technologies and Systems*, Monterey, CA, USA, December 1997.

[MD97] Jon Meyer and Troy Downing. *Java Virtual Machine*. O'Reilly and Associates, Sebastopol, CA, USA, 1997.

[MDC93] Brian Marsh, Fred Douglis, and Ramón Cáceres. Systems Issues in Mobile Computing. Technical Report MITL-TR-50-93, Matsushita Information Technology Laboratory, Princeton, NJ, USA, February 1993.

[Min98] Vladimir Minenko. Virtual Network Computing Client for Palm Platform, September 1998. http://www.icsi.berkeley.edu/~minenko/PalmVNC.

[NKB96] Michael Nidd, Thomas Kunz, and James P. Black. Wireless Application and API Design. In *Proceedings of the Fourth International IFIP Workshop on Quality of Service*, Paris, France, March 1996.

[Nob98] Brian D. Noble. *Mobile Data Access.* PhD thesis, School of Computer Science, Carnegie Mellon University, Pittsburg, PA, USA, May 1998. Available as technical report CMU-CS-98-118.

[NPS95] Brian D. Noble, Morgan Price, and M. Satyanarayanan. A Programming Interface for Application-Aware Adaptation in Mobile Computing. In *Proceedings of the Second USENIX Symposium on Mobile and Location-Independent Computing*, Ann Arbor, MI, USA, April 1995.

[NSN+97] Brian D. Noble, M. Satyanarayanan, Dushyanth Narayanan, James Eric Tilton, Jason Flinn, and Kevin R. Walker. Agile Application-Aware Adaptation for Mobility. In *Proceedings of the 16th ACM Symposium on Operating System Principles*, St. Malo, France, October 1997.

[Obj97] ObjectSpace, Inc. *Voyager Core Technology User Guide.* ObjectSpace, Inc., 1.0.0 edition, 1997.

[RAS96] M. Ranganathan, Anurag Acharya, and Joel Saltz. Distributed Resource Monitors for Mobile Objects. In *Proceedings of the Fifth IEEE International Workshop on Object-Orientation in Operating Systems*, Seattle, WA, USA, October 1996.

[RASS97] M. Ranganathan, Anurag Acharya, Shamik D. Sharma, and Joel Saltz. Network-aware Mobile Programs. In *Proceedings of the USENIX Annual Technical Conference*, Anaheim, CA, USA, January 1997.

[RSFWH98] Tristan Richardson, Quentin Stafford-Fraser, Kenneth R. Wood, and Andy Hopper. Virtual Network Computing. *IEEE Internet Computing*, 2(1):33–38, Jan-Feb 1998.

[RWW96] Roger Riggs, Jim Waldo, and Ann Wollrath. Pickling State in the Java System. In *Proceedings of the USENIX 1996 Conference on Object-Oriented Technologies*, Toronto, ON, Canada, June 1996.

[Sat93] M. Satyanarayanan. Mobile Computing. *IEEE Computer*, 26(9), September 1993.

[Sat96] M. Satyanarayanan. Fundamental Challenges in Mobile Computing. In *Proceedings of the 15th ACM Symposium on the Principles of Distributed Computing*, Philadelphia, PA, USA, May 1996.

[Sun97] Sun Microsystems, Inc. *Java Native Interface Specification, Release 1.1.* Sun Microsystems, Inc., Mountain View, CA, USA, May 1997.

[Tau96] Joshua A. Tauber. Issues in Building Mobile-Aware Applications with the Rover Toolkit. Master's thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, June 1996.

[Wat94a] Terri Watson. Application Design for Wireless Computing. In *Proceedings of the IEEE Workshop on Mobile Computing Systems and Applications*, Santa Cruz, CA, USA, December 1994.

[Wat94b] Terri Watson. Wit: An Infrastructure for Wireless Palmtop Computing. Technical Report UW-CSE-94-11-08, Department of Computer Science and Engineering, University of Washington, Seattle, WA, USA, November 1994.

[Wat95] Terri Watson. Effective Wireless Communication Through Application Partitioning. In *Proceedings of the Fifth Workshop on Hot Topics in Operating Systems*, Orcas Island, WA, USA, May 1995.

[WB97] Girish Welling and B. R. Badrinath. A Framework for Environment Aware Mobile Applications. In *Proceedings of the 17th International Conference on Distributed Computing Systems*, pages 384–391, Baltimore, MD, USA, May 1997.

[WB98] Girish Welling and B. R. Badrinath. An Architecture for Exporting Environment Awareness to Mobile Computing Applications. *IEEE Transactions on Software Engineering*, 24(5):391–400, May 1998.

[WRB+97] Kenneth R. Wood, Tristan Richardson, Frazer Bennet, Andy Harter, and Andy Hopper. Global Teleporting with Java: Toward Ubiquitous Personalized Computing. *IEEE Computer*, 30(2):53–59, February 1997.