Babylon: A Java-based Distributed Object Environment

Matthew Izatt

A thesis submitted to the Faculty of Graduate Studies
in partial fulfillment of the requirements
for the degree of

Master of Science

Thesis Supervisor: **Dr. Tim Brecht**

Graduate Programme in Computer Science
York University
Toronto, Ontario

July 10, 2000

Babylon: A Java-based Distributed Object Environment

Matthew Izatt

A thesis submitted in conformity with the requirements

for the Degree of Master of Science

Graduate Programme in Computer Science

York University

Abstract

This thesis reports on the building of a system in Java, designed to handle distributed objects. A fully functional prototype system, named Babylon, has been built which implements the infrastructure to support distributed Java objects. By leveraging the security and portability features of Java, Babylon allows interested parties to contribute heterogeneous hardware resources to a shared computing system, without concern for security. While other Java-based distributed systems exist, we believe Babylon represents an improvement over other systems because of the combination of features supported by Babylon.

Babylon is composed of class libraries which are written entirely in Java and that run on any standard compliant Java virtual machine. These class libraries implement and combine several key features that are essential to supporting distributed and parallel computing using Java. Such features include the ability to:

- easily create objects, which require no special programming, on remote hosts and interact with those objects through either synchronous or asynchronous remote method invocations,
- freely migrate objects to heterogeneous hosts at any point in time,
- request console, file or socket input and/or output on the originating host, regardless of the location of the object,
- seamlessly handle the arrivals and departures of compute servers to and from the system.

We report on a system comprised of class libraries, which enables programmers to easily interface both new and pre-existing objects with independent Babylon server objects, which manage the physical resources available to complete the computation. Experimental results are included which demonstrate the overheads involved in each feature. Drawing from these results, and from experience using Babylon, we conclude with a discussion of Babylon's strengths and weaknesses.

Acknowledgments

The Babylon Project was a dream given form. I'd like to thank Dr. Tim Brecht for his supervision. His suggestions, comments and famous red pen made this a better-written and more comprehensive thesis. Also thanks to Dr. Richard Paige, Dr. Franck van Breugel and Dr. Rene Fournier for serving on my examining committee.

This research (and other personal items) was funded in part by a scholarship from the Natural Sciences and Engineering Research Council of Canada.

This thesis would not have been possible without the love, support and provisioning of food from my parents. In addition, I'd like to thank my colleagues in the grad program (Rick, Behrad, Leyla and Andrew) for their inspiration, help and friendship.

> "G'Quan wrote: 'There is a greater darkness than the one we fight. It is the darkness of the soul that has lost its way. The war we fight is not against powers and principalities, it is against chaos and despair. Greater than the death of flesh is the death of hope, the death of dreams. Against this peril we can never surrender.'"

– G'Kar in Babylon 5:*Z'ha'dum*

> "The past tempts us, the present confuses us, and the future frightens us. And our lives slip away, moment by moment, lost in that vast terrible in-between."

– Centauri Emperor Turhan in Babylon 5:*The Coming of Shadows*

> "No boom?"
> "No boom."
> "No boom *today*. Boom tomorrow. There's *always* a boom tomorrow. Boom, sooner or later. *BOOM!*"

– Babylon 5:*Grail*. Also quite applicable to software development.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Babylon is a system comprised of class libraries, which enables programmers to implement distributed objects in Java. We believe that Babylon's range of features, combined with its ease of use, make Babylon a powerful system for distributed programming and an improvement on other Java-based distributed systems. In addition, its implementation in standard Java makes it available to the widest possible audience.

In this chapter, we discuss the motivation behind implementing distributed systems in Java; we list the objectives of the Babylon project; and we summarize the contributions this thesis makes to the area of Java-based distributed systems. Finally, we outline the organization of the remainder of the thesis.

1.1 Motivation

The rapid rise of the public Internet, as well as smaller organizational intranets, has been due to the seamless access they provide to information that is distributed in remote locations across the network. The sharing of information is currently the major factor in the growth of networks due to improved accessibility.

This thesis describes the design, implementation and experimental evaluation of a system, called Babylon, which aims to make distributed computing resources as easily available as distributed information. A basic requirement is that the implementation of a system for distributed computing should benefit from the extra power and resources of the many computing devices located on large networks. In order to accomplish this, the varied computational resources available on wide area networks must be able to communicate with each other. Just as the invention of HTML created a language to facilitate the sharing of information, the invention of Java [17] created a language which allows the sharing of computational resources among heterogeneous computing systems.

The most compelling reason for implementing distributed and parallel applications in Java is that compiled Java programs produce byte-code[1] which can be executed or interpreted on any machine that implements the Java Virtual Machine (JVM) [26]. As well, since each Java virtual machine, which may be executing on different architec-

[1]Other programming languages can also be compiled into Java compatible byte-code.

tures, implements an identical machine, the programmer need not be concerned with problems due to differences in architecture that traditionally plague heterogeneous distributed and parallel applications. Such problems include differences in: sizes of data types, byte orderings and structure alignment. By providing a standard virtual machine, the difficulty of programming for heterogeneous systems is not only greatly reduced but it also provides for new opportunities to migrate executing objects to different hosts.

Another standard Java feature is dynamic class-loading. This provides the foundation upon which objects, which may not be known to the JVM at start-up, may be loaded and executed. This is important in order for distributed objects to be instantiated on any host, at any time. Additionally, Java provides a security manager which can be used to control access to resources in the system. This is especially important in a distributed system where untrusted user programs are granted permission to execute on unrelated hosts. Finally, Java provides a useful implementation of remote procedure calls. Java's Remote Method Invocation (RMI) [42] provides a useful tool for simplifying distributed programming. By taking advantage of built-in features designed to support RMI, such as serialization, it is possible to create projects of greater complexity than is possible in languages without this support.

In summary, the Java language is important due to its popularity and platform independence. This single advantage ensures that Babylon is available to a potentially large audience. In addition, Java provides standardized tools such as remote method invocation, object serialization and reflection, that greatly simplifies distributed programming. Thus, the advantages of Java motivate creating a Java-based system to support distributed objects.

1.2 Objectives

While Java provides an excellent base for creating applications, the features available for distributed object programming are limited. These features, and their limitations will be discussed in greater detail in Chapter 2. The goal of this project is to improve and extend an existing set of Java class libraries for distributed object programming, named Ajents [7], which is further discussed in Chapter 2. Babylon seeks to provide more powerful support for distributed objects, while simplifying the programmer's tasks and maintaining 100% Java compatibility.

In order to meet these goals, certain basic features are required in order for the resulting system to be effective.

- The system must allow users to seamlessly create objects on various machines. Users must be able to access these objects in a fashion similar to local objects.
- Remote objects must be mobile. Babylon is envisioned as a system which allows for continual changes in the availability of computing resources. Mobile objects are required in order for objects to be migrated between servers, should computing resources depart, or become overloaded. Mobility is also required

for mobile agent applications.

- Remote objects should be able to communicate with external objects, devices and networks through means other than method invocations. This includes console and file input/output, and network (socket) communication. Additionally, Babylon should be able to support communication interleaved with migration.

- The system should have techniques for handling the arrival and departure of servers and be able to adequately provide a link between client objects and servers available in the system.

- The system should not pose a security threat to those who contribute resources, in the form of servers.

- The system must be easy to use for experienced Java programmers.

Finally, we seek to demonstrate through performance benchmarks and qualitative analysis that the system features are functioning, useful and efficient.

1.3 Contributions

Babylon contributes new features and approaches to the area of distributed object computing in Java. However, the key contribution of Babylon is the integration of both new, and previously researched features, into a single comprehensive system.

While other systems implement some of the same features in different ways, none combine the breadth of features with the ease of use of Babylon.

Other existing systems for building distributed programs in Java almost universally include special non-standard keywords or other changes to the Java language specification in order for distributed objects to work properly while making programming reasonably easy. In Babylon, no modifications are made to the Java language and no preprocessors, special compilers, or special stub compilers are required. Therefore, Babylon and any programs written for it, can be executed on any standard Java virtual machine.

Another main contribution of Babylon is that it allows almost any Java object to be used in a distributed context. To our knowledge, all other distributed object systems require that participating objects use special keywords, extend pre-defined classes or otherwise be programmed in a special fashion. In Babylon, any object may be created remotely and its methods invoked remotely. To take advantage of migration, objects must be serializable, while to take advantage of I/O requires inheritance from a specific object. While the Babylon class libraries rely upon RMI to execute remote methods, objects need have no knowledge of this. This also means that existing objects for which only byte-code (i.e., no source) is available, can be remotely created, and methods can be remotely invoked. If the object is serializable, it can even be migrated.

An additional contribution of Babylon, and this thesis, is in the area of object migration in Java. The most notable contribution is support for the immediate migration of objects, including objects currently executing a method invocation. The methods used to implement this type of migration in Babylon, notably the use of checkpointing and rollback, are, to our our knowledge, new to Java-based distributed systems.

The use of remote method invocations often limits interaction with a remote object to method invocations, parameters and return values. Traditionally, remote objects (and other existing Java based systems that we are aware of) are only able to provide output to the host upon which they are executing. Babylon includes built-in support for remote input and output to and from consoles, files and sockets. This is done in a manner which is relatively simple to use, fully satisfies security requirements, and operates seamlessly even when objects are migrated.

In addition to the combined features listed above, this thesis also contributes towards the eventual creation of a hierarchical scheduling system which can combine the power of distributed objects with the approaches of workload management to effectively utilize available resources. This is done by providing simple mechanisms upon which scheduling policies can be built.

Babylon contributes several features which are unavailable in existing Java based systems, in ways that ensure that it is effective yet easy to use. In addition, this thesis includes experimental results which show that: good speedup is achieved in an example parallel application; the overheads introduced by Babylon's implementation do not adversely affect remote method invocation times; and perhaps surprisingly, the execution time of objects used in our experiments is not greatly impacted by the cost of migration.

1.4 Outline of the Thesis

The remainder of the thesis is organized as follows.

Chapter 2 discusses work related to Babylon. The relative strengths and weaknesses of other Java-based systems for distributed computing are discussed. This includes Ajents, the system from which portions of Babylon's infrastructure evolved.

Chapter 3 describes the architecture of Babylon and discusses the mechanisms implemented. This chapter covers the overall design strategy, and the resulting object hierarchy. It then continues to describe the important features of the infrastructure, including remote class loading, and the various forms of object migration supported by Babylon.

Chapter 4 reports on the methods used to implement remote input and output in Babylon. Effectiveness, ease of use considerations, and pitfalls of the approach used are also reviewed.

Chapter 5 discusses the requirements for scheduling a distributed system supporting migration and details the scheduling mechanisms that have been implemented in

Babylon.

Chapter 6 evaluates the performance of several of Babylon's key components as well as the performance of some example applications that have been implemented using Babylon. Following these results, we discuss the strengths and weaknesses of Babylon.

Finally, Chapter 7 draws conclusions about the success and failures of Babylon. In addition, the opportunities for future work are discussed.

Chapter 2

Background and Related Work

Considerable work has been completed in the field of Java-based parallel and distributed programming. In this chapter, we consider research related to Babylon and discuss respective strengths and limitations.

Section 2.1 describes the features provided by the Java language which are vital to the implementation of Babylon components. Section 2.2 describes research done on mechanisms and complete systems which are related to Babylon. Section 2.3 describes Ajents [7], a project which predates Babylon, and from which portions of Babylon have evolved. Finally, Section 2.4 concludes the chapter with a discussion of the related work.

2.1 The Java Language

The Java platform [43] supports a great number of features on top of the language specification [16]. As part of Java's standard API [43], Java includes features specifically designed to aid with distributed systems programming. Babylon takes advantage of a number of built-in high-level Java features such as remote method invocation (RMI), object serialization, reflection, the Java security manager, and classloader to produce a system which would be far more difficult to create with other languages, lacking these features. To our knowledge, no other programming language supports all of these features as part of the language standard. In this section, we describe the Java features which serve as important building blocks for implementing Babylon.

2.1.1 Remote Method Invocation

Java's version of Remote Method Invocation [42] provides a level of abstraction which allows a user to utilize a remote reference in the same manner as a local reference. This is an object-oriented version of a Remote Procedure Call (RPC). In order to accomplish this level of transparency, Java adds a layer of complexity to remote object programming. RMI programming requirements and limitations include:

- Instance variables of remote objects are inaccessible, even if they are labeled `public`.
- Remote invocations can not be invoked on static methods, nor can remote methods access static variables.
- An interface must be declared for the remote object, which must include every method available for remote invocation.

- All remotely accessible methods must be declared public, as well as throw a `RemoteException`.

- Remote objects generally extend one of RMI's classes, e.g., `UnicastRemote-Object`. Java only allows single inheritance, thus prevents remote objects from being sub-classes of other related objects.

An additional requirement is that the remote object must be compiled using a special stub compiler (`rmic`) which produces two extra layers of code. These are: a skeleton for the remote object which runs on the server side, and is responsible for dispatching calls to the remote object, and a stub for the remote object which runs on the client side and forwards method invocations to the server side. The remote reference held by the client is a reference to that stub, rather than a direct reference to the remote object. In addition to handling the method invocations, stubs and skeletons also serialize/deserialize parameters and return values.

RMI in Java is limited to synchronous method invocations. However, this limitation is overcome in Ajents [7] and ARMI [35]. Java RMI is also limited by the requirement that remote objects must be created by a server program which runs all the time. However, no such server program is provided with the JDK. Thus the Java standard provides no methods for the dynamic creation of remote objects. Ajents implements an interactive server program in order to provide remotely accessible dynamic object creation.

Activation

Java 1.2 introduces `Activatable` objects [44]. An activatable object describes an object which is statically registered with a daemon, but is not actually created until a remote method is invoked upon the object. Registering with the daemon is completed by using a "setup" class, which must be executed on the remote host. Later, upon receiving the first RMI request to the registered object, the daemon instantiates the object. While this reduces the need to have remote objects permanently active (whether they are currently in use or not), it still does not provide the flexibility of dynamic on-demand object creation. Activation is mainly intended as a means of improving resource control rather than for supporting remote object creation. In Babylon, only the Babylon server must be started on a remote host and once it is running it provides the facilities necessary to create any user defined remote object.

2.1.2 Object Serialization

Java provides a standard method [41] for objects to be converted into a format where they may be stored as a sequence of bytes. This process is called serialization, and in the context of other languages is sometimes referred to as marshaling. Serialization is vital for the transfer of objects between hosts, or in order to implement persistence (via saving objects to disk). Object serialization occurs transparently when an object is passed as a parameter in a remote method invocation or when an object is passed as a parameter through an I/O stream. In order to ensure that programmers are aware of this feature, objects which may be serialized must implement the serializable interface. Since Babylon's implementation of migration relies upon object serialization to move objects between hosts, it is therefore impossible to migrate any objects that are not serializable using a standard Java virtual machine. It should be noted that some core Java API objects are not serializable. This includes most of the graphical user interface as well as the Java thread package. Thus, an executing Java thread is not migratable, because it is not serializable.

2.1.3 Classloading

Most programming languages and environments require that all code be statically compiled and linked as a single step in creating an application. Java, however, supports dynamic class loading [17]. This means that code that is used to implement an object is only loaded when required (i.e., when that class is first encountered during execution). Dynamic class loading implies the ability to reference and instantiate objects which are unknown to the application at compile-time. This makes it possible for a Babylon server to be executing, and users to request remote object creation, without the Babylon server having any prior knowledge about the objects which are to be created.

2.1.4 Reflection

Java provides a feature, called reflection [43], which allows the object to determine type information from an object instance, and to create an object instance from type information. This feature is vital to portions of Babylon. Reflection allows instances of objects to be created even if the class name is not known until run-time. For example, a user-provided class name (e.g. `‘‘ca.yorku.cs.myClass’’`) can be used to create an instance of that class. Reflection can also used to determine the methods and fields of objects that are dynamically created. For example, given an object named `myObject`, of class `myClass` and the method (as a string) `‘‘getName’’`, reflection can be used to invoke `myObject.getName()`.

2.1.5 Security Manager

The Java security manager [43] is a class which allows applications to implement a security policy, which defines which Java features, objects are allowed to access. To enforce this, as the first step in performing many Java API methods (e.g., file open, process creation), a security manager method is called to check whether the current application is allowed to access that method. The security manager allows Babylon to ensure security for those who provide computing resources by restricting object access to those API features which do not affect the host. This also provides flexibility, as each server application may alter the security manager to their own specifications.

For example, on a trusted internal network, users may be allowed access to the file system, while on the public Internet, objects should not have any access to the file system. Babylon uses a default RMI Security Manager which prevents remote objects from accessing all potentially dangerous functions[1]. It is possible to change these restrictions, by extending the RMI Security Manager class.

2.1.6 Distributed Garbage Collection

In addition to garbage collection done on a single host, Java also supports distributed garbage collection [42]. It does this through a reference-counting algorithm, which keeps track of all live references, local and remote. When references are created to objects, that object's reference count is increased. Similarly, when remote references are deleted, the reference count is decremented. When the object has no remaining references, it may then be garbage collected. This method implies that creation and deletion of remote references involves some number of network messages. Thus, the creation and transfer of remote references should be done as sparingly as possible, to avoid performance degradation. Java support for local and distributed garbage collection simplifies the Babylon implementation by minimizing the amount of attention that needs to be paid to garbage collection.

[1] See the Java API [43] description of the RMI Security Manager for a complete list.

2.2 Related Work

A number of systems for distributed computing in Java have been developed either concurrently with Babylon or prior to it. These systems also leverage the Java virtual machine in order to provide distributed or parallel computing platforms in heterogeneous computing environments. This section concentrates on research most directly related to Babylon, primarily those which support Java-based distributed computing. Specialized work has been completed in many of the techniques and mechanisms that are used in Babylon (such as migration and checkpointing). However, this section restricts itself to the most directly related work, mainly dealing with complete systems for Java-based distributed computing. Research into general scheduling and I/O issues is covered in Chapters 4 and 5. Readers interested in Java parallel and distributed computing research are also refered to the Java Grande Forum [21] web site, which has links to all the major Java related conferences.

2.2.1 JavaParty

JavaParty [34] is an extension of Java designed to support transparent remote objects. It introduces the `remote` class modifier to the Java language. Adding this new keyword to the class definition denotes that the class should be used in a distributed fashion. To accomplish this, JavaParty uses a preprocessor which converts `remote` classes into pure Java code utilizing RMI. The change to the language is designed to simplify RMI programming, placing the burden of creating and handling remote proxies upon the preprocessor. The main drawback is that it introduces modifications to the Java language and therefore necessitates the use of the JavaParty preprocessor. This greatly simplifies the programming task, but results in increased complexity in the actual Java code produced by the preprocessor. JavaParty creates ten Java byte-code files for every single JavaParty class. Overall, the focus of JavaParty is on simplifying distributed RMI Java programming, at the expense of source code portability. Babylon provides mechanisms to allow objects to be created remotely and their methods invoked, without modification to either object source or byte-code.

2.2.2 Javelin

Javelin [8] (previously titled SuperWeb [1]) implements a "global computing infrastructure" which brings together three types of entities: clients, brokers and hosts. *Clients*, who seek computing resources, register with a *broker* and submit their work in the form of an applet. *Hosts*, who are willing to donate resources, contact the broker and run the applets. This work emphasizes methods for bringing together clients and hosts, and focuses on ways of bartering for CPU time. Support is concentrated on programs which can be divided into and run in Java applet form. Due to the reliance upon applets, Javelin does not support object interactivity (in the form of remote method invocations), or advanced features like migration.

Javelin++ [29], is the most recent extension of Javelin. This research concentrates on issues related to the broker, the entity that connects the clients and hosts. The two key issues addressed are how to ensure the scalability of the broker network, and how to schedule work. Javelin++ introduces a distributed broker network, designed to overcome single broker bottlenecks, and to scale on a global basis. Javelin++ also compares two different schemes of work distribution, a deterministic scheduler, and a probabilistic work-stealing scheduler. The results provided are promising and suggest that their hierarchical scheduling scheme performs well on a scale larger than existing projects, including Babylon. Babylon does not implement advanced scheduling algorithms or mechanisms, and the Javelin++ research may be quite applicable to future Babylon scheduling infrastructure and algorithm work. Javelin++, as with Javelin, is still designed around dividing computational jobs into defined pieces rather than distributed interactive objects, and does not support object migration.

2.2.3 Charlotte

The goal of Charlotte [4, 22] is to support distributed shared memory on top of the Java virtual machine. It provides classes which encapsulate the behavior of a distributed shared memory system (DSM). All accesses to shared objects must be done through the provided interface (e.g., `myInt.get()`). Charlotte provides an abstraction of DSM, with fine-grained support for distributed work. Charlotte is an interesting approach to distributed object programming. However, the variable access methods adds extra programming complexity to accessing shared variables, in addition to the overhead required to keep shared memory consistent across nodes. Babylon avoids these complexities, providing distributed object access through remote method invocations rather than shared memory.

2.2.4 Java/DSM

Java/DSM [47] parallelizes the virtual machine itself, by making the virtual machine a TreadMarks [23] client. This work allows for distributed shared memory (DSM) applications in Java. In order for Java/DSM applications to work correctly, they must include annotations to assist the underlying system in keeping memory consistent across machines. TreadMarks is a DSM system which is heavily dependent on operating system and architectural support. Thus, the modified virtual machine is limited by the homogeneous machine requirements of TreadMarks.

2.2.5 ARMI

Previous work has considered mechanisms for implementing asynchronous remote method invocations [35]. In this work a modified Java RMI stub compiler (`armic`) is used to implement asynchronous remote method invocations. If the user chooses to use `armic` (instead of the usual `rmic`), all stub classes created will be modified so that (synchronous) RMI calls are made by a separate thread resulting in asynchronous behavior by the calling object. The user is provided with the option of allowing all

RMI calls within a class to be either synchronous or asynchronous, but not both. Babylon overcomes this problem by providing two distinct interfaces for synchronous and asynchronous remote method invocations. An additional benefit of Babylon's approach is that Babylon does not require users to compile their classes using any stub compilers (not even `rmic`).

2.2.6 Transparent Migration

Fünfrocken [13] presents a technique that allows Java programs to migrate both object state and program counter position transparently. This is done by preprocessing the Java source code. Code is added that saves the runtime state at certain points (dividing the code up into regions), making it possible to restore this state at the same point. Restoration to the proper point is done by creating an artificial program counter. Each region is surrounded by `if`-statements which increment the program counter prior to migration, and checks the artificial program counter after migration, to ensure code previously executed is not repeated.

Fünfrocken's approach requires access to all source code; thus program libraries can not be instrumented. Support is not provided to ensure that local references such as files handles work correctly after migration. Finally, there is significant overhead: file size space penalties for the increase in source code size (a factor of 4 in their results), a memory space penalty to store a copy of all local variables, and compile and runtime time penalties.

Babylon's implementation of transparent migration bears some similarities. Babylon stores object state and program counter position at the method invocation level, rather than at artificial points within the program. Due to this, Babylon does not require any source code pre-processing, and thus does not increase source code size. Fünfrocken's approach checkpoints and saves state more often than Babylon, thus losing less computation after any migration, and restart. However, while Babylon loses more computation after migration, rollback and restart, Babylon's fewer checkpoints approach means less overhead than Fünfrocken's approach.

2.2.7 Java-based Mobile Agent systems

Agent-oriented Java systems, some examples of which include Mole [40] and Aglets [25], are designed to support Java-based autonomous software agents. Remote objects (agents) may be created, and are provided with mechanisms for mobility and communication. This independence allows objects independence, but limits ties to the original client who created the object. Typically, these agent-oriented systems do not provide references to outside objects, thus limiting the user's ability to easily perform remote method invocations. Generally, agent-oriented systems concentrate upon the independent movement of objects, while providing less support for interaction and control by the object's creator. In such systems, in order for agents to behave intelligently, yet independently, their behavior must be programmed. For example, it is typical for such systems to require a stop method to be called, before migrating an object. Therefore, each agent must be programmed to handle a stop method in which it prepares to migrate, including ensuring that it reaches a steady state. The agent also must be programmed to handle a restart method after migration is complete. This type of intelligent behavior places the burden upon the programmer, rather than the system. Babylon, on the other hand, is targeted towards easily building distributed object applications and as a result emphasizes programming simplicity and object control. Unlike existing agent-oriented systems, Babylon objects are not expected to exert self-control (such as stop methods), or interact with the system.

2.2.8 Voyager

ObjectSpace's Voyager [32, 15] is an extensive commercial product line which includes an object request broker, application server and other products. Voyager's distributed systems toolkit supports remote object creation, asynchronous remote method invocation and migration as well as substantial features beyond what Babylon supports such as multi-casting, name spaces, servlets, timers and CORBA support. Similar to Babylon, Voyager allows any class to be remote-enabled without modification, including third party classes without source. As with Babylon, Voyager requires no extra stub creation utilities as all the "distributed glue" is generated at run-time. As a private commercial package, Voyager does not provide any details of how their "distributed glue" is constructed, thus not allowing any implementation comparison with Babylon. Unlike Babylon, Voyager's migration facilities do not support migration with checkpointing and rollback, a key feature of Babylon. In addition, Voyager provides no support for distributed I/O using files and sockets, instead supporting messages for distributed communication.

2.2.9 ABC++

ABC++ [3], is a library designed for concurrent object-oriented parallel programming in C++. It has influenced the design of Babylon's (and previously Ajents') interfaces for remote object creation and asynchronous RMI. ABC++ does not support object migration, nor does it include special features for heterogeneous computing. ABC++ is limited by the homogeneous nature of compiled C++ code, as well as the lack of fully standardized high-level features in C++, as compared to Java, such as object serialization, dynamic class-loading and distributed garbage collection.

2.3 Ajents

Babylon retains parts of the underlying mechanisms used in Ajents [7], a previous project completed by Patrick Chan. Ajents provides portions of the codebase for this project, as well as some of the basic ideas which are extended in Babylon. The Ajents' subsystems which are reused and/or extended for Babylon are described here.

2.3.1 Remote Object Creation

While the definition of Java used for Ajents and Babylon (JDK 1.1) provides support for remote method invocation, the methods must be invoked upon existing objects that have been statically created. Ajents implements remote servers which handle requests for remote object creation. This allows an Ajents user to dynamically create objects on remote machines.

Remote object creation is supported by the Ajents class libraries by using a class, `AjentsObj`, which must be extended to create an object which executes on a remote host. `AjentsObj` and classes which extend `AjentsObj` are Java RMI classes and must satisfy Java's RMI programming constraints (detailed in Section 2.1). The key mechanisms used for remote object creation are implemented inside the Ajents server. Once a program calls `Ajents.remoteNew(object name, class name, server name)`, the method call is translated into an RMI call to a remote Ajents server. There, using Java reflection mechanisms [43], the requested class can be loaded into a `Class` object at the remote host and then instantiated. The Ajents server then adds the new object to its object table, and returns a remote reference to the calling object.

Because the `AjentsObj` is itself an extension of Java's RMI mechanisms [42], this method of creating a `Remote` object and returning a reference to it takes advantage of the distributed garbage collection built into the Java RMI implementation. That is, once the remote reference is no longer used, both the reference, and the remote object will be garbage collected. Babylon does not require objects to extend any Java RMI classes, thus freeing the programmer from knowing and following the details and limitations of Java's RMI specification. However, Babylon is still able to take advantage of Java's distributed garbage collection mechanisms.

2.3.2 Ajents Server

The role of the Ajents server is to provide a point of contact for creating objects on the host on which the Ajents server resides. When remote object creation is requested, the Ajents server instantiates the requested object based upon the class name provided. In Ajents, the byte-code for this class must reside on the server. Babylon introduces remote classloading so that users may create objects remotely, even if they do not have direct access to any remote hosts.

In addition, the Ajents server keeps track of objects currently residing on it. This permits the server to keeps track of a forwarding address in the form of a new reference which points to the new location of the object, if the object gets migrated.

2.3.3 Asynchronous Remote Method Invocation

Existing standard Java techniques for remote method invocation [42] cause the execution of the requesting object to be suspended until the remote machine is contacted, the method is invoked, and notification of method completion is received (usually in the form of a return value). Ajents provides a means for performing asynchronous remote method invocations thus enabling applications to overlap communication and computation.

In Ajents, when the user performs an asynchronous RMI, a separate thread is created which performs the RMI. As a result, the original object is only blocked during the creation of the thread, and thereafter continues execution. Meanwhile, the new thread performs a regular synchronous RMI.

To support return values, Ajents uses the concept of a future. In essence, the `Future` object is a temporary receptacle for the return value. The return value (which may be an object) is held inside the `Future` object until a `get()` request is made by the program. When a call to `get()` is made, the result of the asynchronous RMI is returned to the calling object. If the result is not yet available (i.e., if the `get()` request is made before the remote method invocation has completed), the object will block until the result is available, at which point the object is automatically unblocked and execution continues.

Ajents does not support the return of exceptions from remote methods to the calling thread. This feature is added in Babylon.

2.3.4 Remote Object Migration

Ajents leverages Java features to provide support for a simple means of migrating Java objects between different operating systems and even different architectures. Some of the mechanisms required in order to migrate Java objects are partially provided in current Java implementations in the form of object serialization [41], classloading [17], and the standardization of the Java virtual machine [26].

Ajents supports a basic method for migrating idle objects (i.e., objects with no remote methods currently executing). The migration of idle objects is fairly simple since we do not need to concern ourselves with the possibility that they will be modified during or after migration. This is accomplished by using the `java.io` package's `readObject()` and `writeObject()` serialization methods in combination with socket connections. In the case of an object currently executing a method, the migration request blocks until the execution is complete. Following completion of the method, the object is migrated, while further remote method invocations are blocked until the migration is complete.

2.3.5 Features not adapted for Babylon

In addition to the previously listed features, Ajents also includes two other main features. These are not applicable to the work done for Babylon, and thus are not adapted for or included in Babylon.

Ajents supports active objects by associating a thread with each remote object designated as active. Each time a remote active object is created, a new thread is also created. Active objects may override a `run()` method, which is invoked upon creation by the thread. This allows the object to complete its own tasks, independent of any method invocations. Objects not designated as active do not have their own thread of control, and can only be affected through method invocations.

Support for active objects is not adapted for Babylon because it is not possible to migrate threads in Java[2]. For this reason, active object support is not part of Babylon, in favour of complete support for migration.

Ajents also includes a Remote Object Monitor. This feature serves as a graphical tool to display the servers in the system, as well as the names of the objects residing on each server. While this feature could be adapted to Babylon, it is not considered a priority and therefore has not been updated to work properly with Babylon.

2.3.6 ParaWeb

Much of the impetus for the design of Ajents came from experience with ParaWeb [5], a Java-based environment for parallel computing. ParaWeb was designed and implemented before remote method invocation [46] and object serialization [37] were added to Java. Hence, objects communicate through the awkward sockets interface.

[2]With modifications to the virtual machine, or through careful alterations to the user's program, limited thread migration is technically possible [13].

In addition, ParaWeb does not support remote *object* creation and remote method invocation is accomplished using remote *thread* creation.

2.4 Discussion

As a language, Java provides an excellent base for implementing an object-oriented distributed system. Primary among Java's advantages are its platform independence, and its high-level support for distributed programming, as described in Section 2.1.

This advantage has lead to research into systems which, as described in Section 2.2, take many approaches to distributed computing using Java. While sharing many of the same goals as the systems discussed, we believe Babylon has fewer restrictions upon its use. Babylon is built entirely upon the existing distribution of Java tools in the JDK. No modifications to the virtual machine are necessary, the Java language has not been changed, and we do not require the use of modified compilers, stub compilers or preprocessors. We believe that Babylon is capable of supporting most of the features included in other, similar systems. While not completely transparent to the programmer, Babylon does provide these features more transparently in some cases and in other cases with fewer restrictions than existing systems.

We note that research into Java-based systems is a fairly young field, and some of the issues Babylon seeks to address have not been examined in a Java context before. For example, no other Java-based distributed system, that we are aware of, incorporates remote object I/O into their system. In fact, providing users with remote I/O operations for general parallel and distributed programming has received little attention [10]. In Chapter 4 we provide a brief survey of related work on distributed I/O operations, while Chapter 5 provides an overview of related scheduling research which could influence Babylon's further development.

Chapter 3

Babylon Architecture

In this chapter, the architectural design and implementation of Babylon is described. Detailed in this chapter are the goals of the project, the system architecture that is the result of these goals as well as a discussion of the compromises that were required in order to obtain these goals.

One over-riding goal which impacts every design decision is the desire to create a system which requires no special tools, altered virtual machines, or additions to the language specification. To achieve this goal, Babylon consists entirely of native Java applications and class libraries, written in pure Java, compiled using standard compilers. By restricting Babylon to native Java applications and class libraries, it may be used by anyone with access to a standard Java virtual machine. Thus Babylon is available for use by the vast majority of computer users.

3.1 Introduction

The design and implementation of Babylon involved making many decisions and trade-offs in order to achieve a useful distributed object system. In order to better understand the low-level details of Babylon, this section first presents a broader high-level view. First, some terminology is introduced, which is used throughout the thesis, then Babylon's features are described from a user's perspective.

3.1.1 Terminology

In order to provide a more clear picture of how Babylon works, we describe a simple example of the object relationship between a main program, a *remote object*, created by the main program, and a scheduler when used in conjunction with Babylon.

The system upon which the user runs their main program is referred to as the *originating host*. This is the only host the user needs access to. All interaction between the user and Babylon takes place on the originating host. This includes the execution of the main program and all input to and output from remote and local objects. Additionally, the originating host can serve as the source for the byte code used to implement the remote objects. This host and the main program executing upon it are often referred to in this thesis as the *client*, in a client-server relationship.

A system upon which one or more remote objects are created and execute is referred to as a *remote host*. Remote object creation and migration is facilitated through a *Babylon server*. Remote objects can be created, and execute on any host which is running a Babylon server.

The system upon which a *scheduling server* runs is referred to as the *scheduling host*. The scheduling server may be contacted by both the originating host and the remote hosts, since it provides the user with a reference to an available Babylon server.

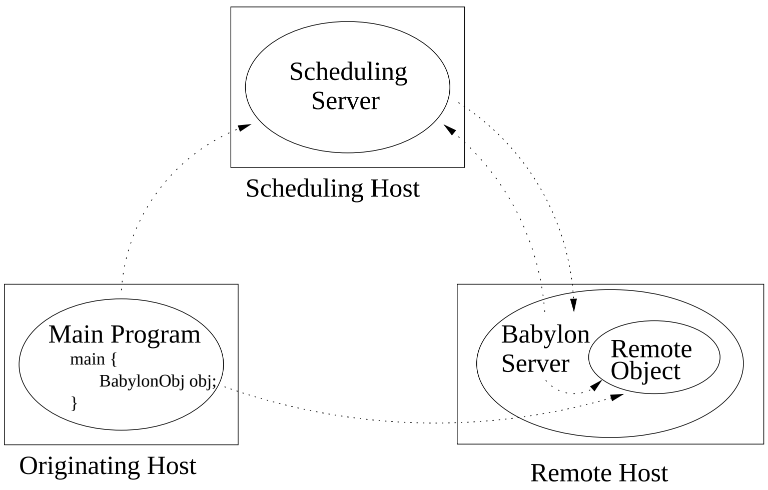

Figure 3.1: A Simple Example of Babylon's Referencing Structure.
Arrows represent the referencing structure, with the source being the remote reference, and the destination being the referenced object.

Figure 3.1 provides a high-level graphical representation of basic relationships between hosts and objects. The main program on the originating host contains references to the scheduling server, and to any remote objects it has created. The Babylon server contains references to all objects currently residing on it, as well as a reference to the scheduling server. The scheduling server contains references to all available

Babylon servers.

Note that it is possible for a single physical machine to act as the originating, remote and scheduling hosts.

3.1.2 A Short Programmer's Guide

This section presents short examples of how Babylon's main features are used by programmers. It demonstrates Babylon's features from a user's perspective. The mechanisms which implement these features are explained in greater detail in the following sections of this chapter.

Remote Objects

Figure 3.2 contains an example of an object which may be created as a remote object in Babylon. Remote objects in Babylon need no special code or compilation in order to be created or have their methods invoked. The object in the figure implements `serializable`, which means it may have its state serialized or deserialized. This also means that the object may be migrated by Babylon.

Remote Object Creation

In order for objects to take advantage of Babylon's features, objects must first be created by Babylon. Figure 3.3 shows an example of how a user creates the remote object defined in Figure 3.2. In the example, the user initially registers with the

```java
public class NewObj implements Serializable {

  private String name;

  NewObj() {
    name = new String("default");
  }

  boolean setName(String newName) {
    name = newName;
    return true;
  }

  String getName() {
    return name;
  }
 }
```

Figure 3.2: Code Example of a Remote Object
Note that this is a completely standard Java object. Nothing special is, or need be, added for it to be used as a Babylon remote object.

Babylon system in order to obtain a reference to a scheduler which is later consulted in order to access an available server. Schedulers are further explained in Chapter 5. Four parameters are passed to the `Babylon.new` method. These represent (1) the fully qualified class name, (2) a user-specified name for the object, (3) the location of the source code on the originating host[1], and (4) a Babylon server for the object to be created on. Note that the `BabylonObj` returned to the user is only a proxy object, which contains the object's real remote reference. Section 3.2 further describes the relationship between the `BabylonObj` and Babylon's internal representation of the remote object. The Babylon server and scheduling server, although accessed by the user, are started by the administrators of those hosts.

[1]One could implement mechanisms to obtain the source code from any available host. However, this is not currently implemented in Babylon.

```
   // Register to get a scheduler object
   // which knows about available servers
1  BabylonScheduler sched = Babylon.register();

   // Create a remote object on an available server
2  BabylonObj obj = Babylon.new("babylon.tests.NewObj", "NewObj1",
             "/home/babylon/tests/myobjs.jar", sched.AvailServer());
```

Figure 3.3: Code Example of Remote Object Creation

Remote Method Invocations

Communication with an object is completed primarily through remote method invocations (RMIs). Like Ajents, Babylon supports both synchronous and asynchronous remote method invocations (ARMIs). Babylon also handles exceptions for both RMI and ARMI (explained in Section 3.5.2).

Figure 3.4 demonstrates examples of `Babylon.rmi()` and `Babylon.armi()` invocations, and exception handling with ARMI. Line 1 demonstrates a synchronous RMI call made by invoking `Babylon.rmi()`. The parameters to this method represent (1) the object reference (originally obtained as shown in Figure 3.3), (2) the method name, and (3) optional method parameters, in this case a single `String` parameter. The result of the method is returned synchronously. That is, execution will block until the result is available. Line 3 demonstrates an asynchronous RMI call made by invoking `Babylon.armi()`. The parameters to `Babylon.armi()` are identical to those for `Babylon.rmi()`. However, a `Future` object is returned immediately, and execution continues. The result is not known until line 4, where the `get()` method of the `Future` object is called. This method blocks until the result of the ARMI call is available.

```
   try {
     // Make synchronous RMI call to the method setName
1    result = Babylon.rmi(obj, "setName", "NewObj2");
2  } catch (Exception ex) { ... }

   // Make asynchronous RMI call to the method getName
3  Future future = Babylon.armi(obj, "getName");

   // Use futures to get the results. armi exceptions are caught here.
   try {
4    String name = (String) future.get();
5  } catch (Exception ex) { ... }
```

Figure 3.4: Code Example of Synchronous and Asynchronous Remote Method Invocations

There is a notable difference in how exceptions are handled between RMI and ARMI calls. RMI exceptions are thrown and caught immediately (line 2), while ARMI exceptions are not thrown and caught until the `Future` object is accessed (line 5).

Migration

Migration, which will be discussed in detail in Section 3.4, is accessible to the user using the syntax shown in Figure 3.5. In this example, an object, previously created in Figure 3.3, is migrated (line 1). Following migration, checkpointing is enabled (line 2) in order to support migration during execution. An asynchronous method is then invoked (line 3) and the object is migrated a second time (line 4). Finally, a return value is obtained from the future (line 5). This example demonstrates the use of migration as well as the fact that migration occurs transparently to the programmer. They may still use the same `BabylonObj` after migration and are unaware of whether the second migration occurred before, or after, completion of the asynchronous remote method invocation. It is possible the ARMI call was interrupted, the object's checkpointed copy migrated, and the method restarted. However, the user still receives a correct result.

```
   // Migrate "obj" to a different host
   // Parameters:  object, new host
1  Babylon.migrate(obj, sched.AvailServer());

   // Turn checkpointing on
2  Babylon.setCheckPoint(obj, true);

   // obj is checkpointed and method is invoked
3  Future future = Babylon.armi(obj, "getName");

   // Migrate obj, possibly before armi completes
4  Babylon.migrate(obj, "tiger.cs.yorku.ca");

5  result = future.get();
```

Figure 3.5: Example of Code Performing Object Migration

3.2 Object Hierarchy Design

The design of the object hierarchy within Babylon is based upon a simple vision: to provide a system through which clients can easily and seamlessly create and interact with remote objects. This section presents Babylon's primary internal object hierarchy, and describes how the object hierarchy design aids in implementing Babylon's features.

Figure 3.6 demonstrates the object hierarchy, using the class names used in Babylon. The client's point of access to their remote object within Babylon is the `BabylonObj`. This object contains a reference to an `RObject` on the remote host. The `RObject` stores within it the user's remote object, as well as some state information for use by the Babylon Server. The remainder of this section describes the purpose of each object within the hierarchy.

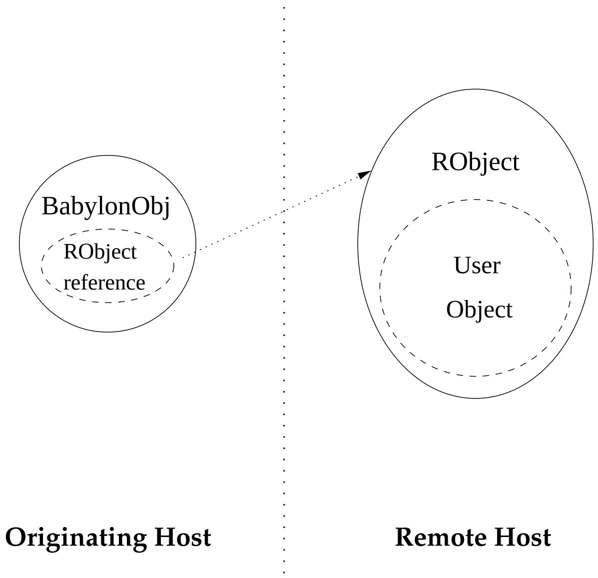

Figure 3.6: Object Hierarchy Design.
Arrows represent the referencing structure, with the source being the remote reference, and the destination being the referenced object.

BabylonObj

The `BabylonObj` represents the user's reference to their remote object. Rather than using a direct reference to the remote object stored on the remote host, the `BabylonObj` is a proxy object returned by `Babylon.new()` (as described in Section 3.1.2). All internal details of the `BabylonObj` are hidden from the user. The class contains a reference to an `RObject` on a remote host, as well as methods to update the `RObject` reference. Updating may be required because the `RObject` reference refers to the `RObject` object's last known residence. If the `RObject` object is migrated, the reference will require updating. Using a proxy object (`BabylonObj`), rather than allowing the user direct access to the reference, provides certain advantages for Babylon. By preventing the user from directly accessing the remote object, all interaction with that object must be completed through designated Babylon interfaces. This ensures that Babylon is aware of all methods invoked upon an object. As well, the use of a proxy allows the `RObject` reference to be altered by Babylon without involving the user. These features of the proxy object are vital to providing transparent third party migration, checkpointing, and rollback, as is described in Section 3.4.

This approach also has disadvantages. Because the `BabylonObj` is unaware of the remote object's interface, invoking remote methods in Babylon can not be done using the traditional `object.method(optional parameters)` interface. Instead the interface described in Section 3.1.2 is used (e.g., `Babylon.rmi(object, method,` `optional parameters)`. Thus the interface for creating and interacting with remote objects is different from local objects. This requires the application programmer to keep track of which objects are local and which are remote and to ensure that the proper interface is used for remote objects. This is not considered a large disadvantage since it forces programmers to remember which method invocations are remote when performance is an issue.

An additional disadvantage of the Babylon RMI interface is that the method and parameters passed as parameters to the RMI call can not be checked at compile time, nor can the types of the parameters that are to be passed to the specified method. As a result it is not possible to detect errors that might otherwise be detected at compile time, such as invoking a non-existent method, or invoking a method with incorrect arguments types, or an incorrect number of arguments. Babylon detects such problems at run-time and throws an exception appropriate for the error. However, we believe that these tradeoffs are outweighed by the advantages of using a proxy, and the desire to maintain 100% Java compatibility.

RObject

The `RObject` is a (Java RMI) `Remote` object. It exists as a container object for the user's remote object. In addition to storing the user's object, the `RObject` also keeps track of other information about objects required in order for Babylon to implement several features. The data members of the `RObject` class are presented here, although their uses are described in greater detail later in this thesis.

- The object's current status (idle, executing, migrating).
- An identification object (in the current version of Babylon, this is simply a user defined name).
- The server upon which the object resides.
- The thread currently executing a method of this object, if any.
- The object's byte code.
- A checkpointed copy of the object (if one exists).
- A boolean representing whether checkpointing is active or not.

All remote methods are invoked through the `RObject` class's remote method `rmi()`. This provides Babylon with an opportunity to intercept method invocations. This, in turn, allows Babylon to complete other tasks before (e.g., checkpointing) and after (e.g., redirecting exceptions) the actual method invocation.

User Object

This level of the object hierarchy represents the user's object. An example user object was provided in Figure 3.2, The user's object is not restricted by any required keywords, interfaces, exceptions or special compilation requirements. Any object may be created as a Babylon remote object, including objects for which only byte-code is available, and have its methods invoked remotely. The only limitation is that objects wishing to take advantage of Babylon's migration facilities must implement `Serializable`.

This lack of restrictions is possible because the user's object is a member variable of the `RObject` class. The `RObject` class maintains an `Object` reference[2], which references the user's object, once that object has been created using `new`. In Ajents, the user's object is inherited from a Java RMI class, and thus also inherited all the requirements and limitations of Java RMI. In Babylon, all remote methods are invoked upon the `RObject` object, which in turn, uses reflection to invoke methods of the user's object.

3.3 Remote Class Loading

One of the strengths of Java is dynamic class loading. This is the ability to load byte-code when required, at run-time, rather than the traditional requirement that class code be compiled directly into an application. This is especially important for Babylon, since Babylon servers create objects which are unknown at server start-up time. The classes for these objects must therefore be loaded dynamically and on demand.

[2]Since all objects in Java are, by definition, subclasses of the `Object` class, we can simply assign the user's object to this reference.

In order for the Java virtual machine to load the byte-code for a class, it accesses an environment variable (`CLASSPATH`) which provides a directory search path in order to find the appropriate file. By default, the system classloader is only able to dynamically load class files available through a locally accessible disk. The JDK also includes an `RMIClassLoader` class which supports network based classloading for RMI objects and parameters to methods of RMI objects. The dynamic class loading described in the Java RMI specification [42] requires that all class files be located either locally, in the `CLASSPATH`, or at a specified URL. This was deemed insufficient for Babylon, where the goal is to allow objects to be created on arbitrary participating machines. In order to utilize remote resources, Babylon users should not be required to place their class files on an accessible WWW server nor are they expected to have accounts on, or physical access to all participating hosts in order to preload a copy of the byte-code. Babylon requires the user to specify a Java Archive (JAR) file which must contain all the class files which may be required by the remote object. No special placement of this file in a specific directory, or upon a WWW server is needed. However, the Jar file must be available on the originating host[3]. This allows users without access to a WWW server to utilize Babylon.

In order to allow classloading to occur from sources unsupported by the default classloading options, Babylon implements its own remote classloader class (named

[3]This is an implementation detail, rather than a design limitation. Accessing the Jar file from a remote site is possible, but not implemented in the current Babylon implementation.

RemoteClassLoader), by extending java.lang.ClassLoader. In addition to allowing Babylon to directly load classes from the Jar archive, this allows Babylon to directly access the cache of Class[4] types contained within the classloader. These Class objects are used to instantiate remote objects upon Babylon servers, using reflection mechanisms. Since the Babylon classloader is just an extension of the default classloader, the same rules are applied for classloading[5].

The mechanics of remote classloading can be described by these basic steps. During remote object creation one of the parameters to the Babylon.new() method is the location of the Jar file containing the relevant byte-code. This file is read by a Codebase class object. The Codebase class stores the Jar file in a byte array, and creates a hashtable containing the names and sizes of the class files contained within the archive. The information in the hashtable is determined upon the initial reading of the Jar file, and is stored in the Codebase object for reuse. Babylon then transfers the Codebase object (through an RMI call) to the Babylon server upon which the new object will be created. On the Babylon server, the RemoteClassLoader associated with the server loads all the classes contained within the Jar file stored in the Codebase object. Once the classes are loaded, the remote classloader can instantiate a remote object of the requested type. The Codebase object is then stored as a member variable of the RObject object. This way, the Codebase object is available

[4] java.lang.Class is a class which encapsulates all typing information about an object.

[5] For example, classes must have distinct names. This is the primary reason for using the package statement, and applies to Babylon classloading the same as normal JVM classloading.

to be migrated along with the actual remote object, rather than requiring that the originating host be contacted for code retrieval prior to every migration.

A scenario similar to that outlined above occurs upon migration. The `Codebase` object is transferred from the current host to the new host prior to the actual object. The `Codebase`'s copy of the Jar file is then used to load the classes into the remote classloader. Following this, the user's remote object is transferred, since the remote classloader is now aware of the types of the objects that are being migrated. These steps occur in order for the receiving server to complete classloading prior to object arrival.

3.4 Migration

An important feature that is not readily available in many systems designed for implementing distributed applications is the ability to easily migrate computation, especially in heterogeneous environments. Because there is no easy means for compiling and executing a single application on multiple architectures, implementing applications such as mobile agents is extremely difficult (although heterogeneous migration is possible [38]).

The difficulty of migrating processes causes serious problems in a network of workstations environment where multiple individual workstations (usually located in people's offices) are shared among a number of users. The main problem occurs in such environments when the owner of the workstation returns to their office and wants to use their workstation, only to find that its resources are being utilized by someone else. Current approaches to this problem include suspending the intruding job until the user is no longer using the machine or in rare instances migrating the process to another machine of the same architecture. Babylon is able to take advantage of Java's platform independence to solve this problem through heterogeneous migration.

Babylon supports the immediate migration of idle objects, as well as two forms of migration for executing objects:

I. delayed migration (i.e., the migration is postponed until all executing methods have completed),

II. immediate migration, using checkpointing and rollback when necessary.

Inherited from Ajents [7] is the immediate migration of idle objects, and the delayed migration of executing objects. The use of checkpointing and rollback for the immediate migration of executing objects is unique to Babylon.

The basic core of migration is largely unaltered from the technique used by Ajents. A socket connection is opened between the current server and a specified new server, the object is written by the current server and read by the new server. One challenge, as previously described in Section 3.3, involves ensuring that the receiving server obtains and loads the relevant class files prior to the arrival of the object. This basic structure has been enhanced in Babylon to enable servers to migrate objects independently. This improves upon object migration in Ajents, which is only designed for a client-initiated migration. In Babylon, migration does not require any updating of user references, establishing a transparency that is lacking in Ajents. Finally, any remote objects in Babylon may be migrated, as long as they are `Serializable`. These are improvements over the alternate systems described in Chapter 2, including Ajents.

3.4.1 Overview of Object Migration

While Ajents supports two forms of migration, that of idle objects and delayed migration, there are limitations. Ajents only supports the basic case of allowing migration by the user who created the object. Babylon allows for third parties (servers, or other users) to request migration. This is especially important in allowing server-initiated migration to occur independently of any user, should the server no longer be available for use by Babylon. Another Ajents limitation was the requirement that the user's reference to their remote object be updated following migration.[6] As demonstrated in Section 3.2, Babylon uses a proxy object, which permits the same object to be used by programmers across migrations.

The new form of migration implemented in Babylon involves the ability to migrate an object at any time, regardless of the object's current state. This is important, because object migration will often occur at the request of the Babylon server, because

[6]The interface to migration in Ajents had the following syntax: `obj = Pcontrol.migrate(obj, machine);`. The main program is forced to update its `obj` reference with a new reference returned by the `migrate` function.

the remote host may be required for use by its owner, or because the host has simply become overloaded. This new form of migration adds to, but does not replace the forms supported in Ajents. Idle objects continue to be migrated immediately, while objects executing methods may now be migrated immediately, or wait until after the method is complete. Delayed migration continues to be supported in Babylon because it has efficiency advantages that can not be overcome by immediate migration.

Delayed migration and immediate migration with checkpointing and rollback are supported because Java does not provide for the ability to save and restore the state of an executing thread (e.g., there is no access to the program counter or stack). Therefore, it is not possible to support the migration of active threads without modifying the implementation of the virtual machine (i.e., the interpreter) [36]. As a result, immediate migration of objects which are currently executing methods is supported through checkpointing and rollback, rather than thread migration. Delayed migration is included in Babylon because it does not require the overhead of checkpointing or rollback. However, because it is unknown exactly how long the delay will be (i.e., until the method completes its execution), it is expected that Babylon's immediate migration features will be the primary form of migration used.

Babylon is able to accomplish the immediate migration of executing objects by taking advantage of Java features to: maintain a record of the objects state before executing each method invocation (checkpointing); interrupt execution when migration is requested; migrate the object in its pre-method invocation state; and re-execute the method invocation.

We now outline each of the steps involved in migration using checkpointing and rollback in more detail:

1. Checkpointing (making a copy of the state of an object), is done using Java serialization. This creates a deep copy of the object, meaning all member variables are themselves copied, rather than just their references. Babylon does this by intercepting each remote method invocation (at the server side) in order to checkpoint the object before allowing the method to execute. Babylon also stores a reference to the thread which will be executing the method (in order to interrupt the thread if necessary). The method name and parameters, however, do not need to be stored on the server side, because any method restart is accomplished through a re-invocation of the method on the client side. This is further demonstrated in Section 3.5.1.

2. Upon receiving a migration request the Babylon server interrupts the currently executing thread (using `java.lang.Thread.interrupt()`). This interrupt causes an exception, which is then caught by Babylon internally, instead of being returned to the user. Babylon is thus able to gain control of the currently executing thread and can then proceed with object migration.

3. Once execution of the thread has been halted, the object is in a static state,

however, there is no way to determine where the thread is executing or the state of the stack. As a result, we migrate the state of the object as it was at the time of the last checkpoint.

4. Following migration, we re-execute the interrupted method. This is done without user intervention, on the client side. The steps involved in re-invoking the method, without user intervention are demonstrated in Section 3.5.1.

The issues involved in designing migration using checkpoint and rollback are discussed in greater detail in the following subsections.

3.4.2 Checkpointing

Babylon allows for checkpointing prior to the start of any method invocation. This is done on the server side, and the serialized copy of the checkpointed remote object is kept within the same object (`RObject`) as the actual remote object. In order to create a deep copy of the object, Java serialization is used. The object is serialized (copied) into a byte array, and stored for later use, if necessary. The standard Java technique of using `clone()` is ineffective because the `clone()` method only creates a shallow copy. It only copies references within an object, not the actual objects pointed to by those references. Serialization is used by Babylon to create a copy of the entire object graph. At the same time as the object is checkpointed, a reference to the thread executing the remote method invocation is stored. This information is vital for stopping method execution when migration is requested, as will be seen in the following subsections.

Control over whether checkpointing occurs, lies in the hands of the user program which has remotely created the object and wishes to call its methods. The user may choose to enable (`Babylon.setCheckPoint(object,true)`) or disable (`Babylon.setCheckPoint(object,false)`) checkpointing for each object, at any time. Or they may wish to override the current setting prior to individual method invocations. Allowing the user this flexibility allows for the object to be checkpointed only at key method invocations, or for checkpointing to be avoided when consecutive read-only methods are being executed. Since the overhead involved in creating a duplicate copy of large objects can be significant, user control over checkpointing is vital to Babylon's efficiency. Furthermore, the overhead of checkpointing is only needed (and justified) for long-running jobs. For these cases, checkpointing occurs relatively infrequently, and substantial roll backs can be incurred without significantly impacting overall execution time. Benchmarks demonstrating the overhead incurred as a result of checkpointing are presented and discussed in Chapter 6.

Additional user control is provided by allowing the user to create a checkpoint at any time. As well, Babylon provides an interface for the user to request a copy of the checkpointed object. This feature is provided as a mechanism for supporting persistent remote objects and fault tolerance, which are projects for future work with

Babylon.

Babylon's checkpointing procedures contain some limitations. These limitations, minor as well as serious, are now discussed.

Because we rely on the object serialization primitives provided in Java, there are some limitations on the type of objects which may be checkpointed and/or migrated by Babylon. Any object which is not serializable falls into this category, and may not be migrated by Babylon. This includes core Java API features such as threads and the Abstract Window Toolkit (AWT). We feel this limitation is reasonable since the serialization (and migration) of threads is impossible in the current JDK, and we can find little reason to want to migrate any part of the AWT.

A potentially more serious problem arises in Babylon and other systems which implement checkpointing for distributed and parallel applications. This problem arises when a remote object `A` invokes a method, `M`, of object `B` which modifies that object's state. If object `A` is checkpointed, then it invokes method `M` of object `B` and it is later migrated and restarted on a different server, `A`'s execution will be rolled back to the checkpoint and continued from that point. Thus method `M` of object `B` will be called twice (rather than once). This potentially causes serious problems if method `M` altered object `B`. This is called the checkpointing consistency problem and is a well known problem [31, 45]. One intractable approach to ensuring checkpoint consistency would be to checkpoint all objects that object `A` could possibly interact with each time object

`A` is checkpointed. Unfortunately, it is not possible to keep track of and checkpoint all such objects because some objects may not be known by or controlled by Babylon (e.g., local objects, objects that are communicated with using standard Java RMI or using a socket). Thus, Babylon does not solve this problem, but exists with the limitation that objects which are to take advantage of checkpointing and rollback must be objects which do not alter, directly or indirectly, external (non-member variable) objects. Programmers must be aware of this limitation and program defensively to avoid it.

A final disadvantage of checkpointing lies in the significant memory usage required in order to create a copy of a remote object. With checkpointing enabled, objects effectively take up twice as much memory. While there is no obvious way to avoid this, it is an additional consideration in deciding when to activate checkpointing.

3.4.3 Migrating a Checkpointed Object

Once execution has been halted by interrupting the executing thread, the migration is ready to occur. At this step, a decision is made whether to migrate the serialized copy of the object or the original object. Babylon checks to see if a copy was made. If so, the copy is migrated. Otherwise the original object is migrated. This method works accurately because the copied object is deleted following the completion of the method invocation. Thus, no copy will exist except when one has been requested and a method is currently executing.

Babylon is also careful to minimize the information that gets transferred upon migration. Since the (`RObject`) object being migrated contains information important only to the current Babylon server, this information relevant only to a single Babylon server is declared `transient`[7] so that it will not be transferred. This helps Babylon minimize object transfer times as much as possible.

Another challenge immediately prior to migration is to be able to halt any threads executing a remote method of the object at the time immediate migration is requested. Since all method invocations are intercepted (as described in Section 3.5.1), Babylon uses this opportunity to store a reference to the thread executing the method. Upon receiving an immediate migration request, Babylon will interrupt the executing thread, causing it to throw an `InterruptedException`. This exception is not returned to the user, but rather is caught internally as an indication that the object is being migrated.

3.4.4 Reference Updating

Since objects may be migrated by Babylon servers, or other third parties, Babylon implements a scheme to ensure that references held by the user (through `BabylonObj` objects) still function after migration has been completed.

As in Ajents, each Babylon server keeps track of all objects that are residing

[7] The `transient` keyword indicates fields that should not be serialized, and will be null upon deserialization.

upon that server. In addition, objects which have been migrated still remain on the server while remote references to them remain. After migration, these objects only contain a reference to the new host of the object that has been migrated. All other internal references are changed to null. The reference to the new host of the object is maintained so that migration can be implemented in a way that is transparent to any object that has a reference to a remote object that has been migrated. This is in contrast to other approaches (e.g., Mole [40]) where the user program receives an exception and must update any references to the remote object that has been migrated.

Figure 3.7 shows the steps taken by Babylon to update object references and ensure transparency after an object has been migrated twice without the client's knowledge (e.g., by the server or another object). These steps are followed to restart an interrupted remote method invocation, as well as for the first remote method invoked after a third party migration. In this figure, (1) the `BabylonObj` uses its remote reference to `Obj1` on Host `A` to remotely invoke a method of `Obj1` that previously resided on Host `B`. `Obj1` had been previously migrated twice, first to Host `C` and then to Host `D`, where it now resides. (2) Since the reference is outdated, the Babylon server on Host `B` throws a `MovedException`, which returns to the originating host a new reference to what it believes to be the new object location (Host `C`). (3) The `BabylonObj` updates its internal remote reference and re-invokes the remote method, using the new reference to the object on Host `C` as the target. (4) Again, the reference is outdated, and a new reference is returned, this time to the object on Host `D`. Finally, the correct reference has been found. (5) The `BabylonObj` updates its internal remote reference, and re-invokes the remote method of `Obj1` using the new reference which correctly points to the new location, Host `D`. Babylon does all of this internally and it is therefore completely transparent to the user, whose contact with the remote object is entirely through the `BabylonObj` (i.e., references to remote objects that have been migrated continue to work).

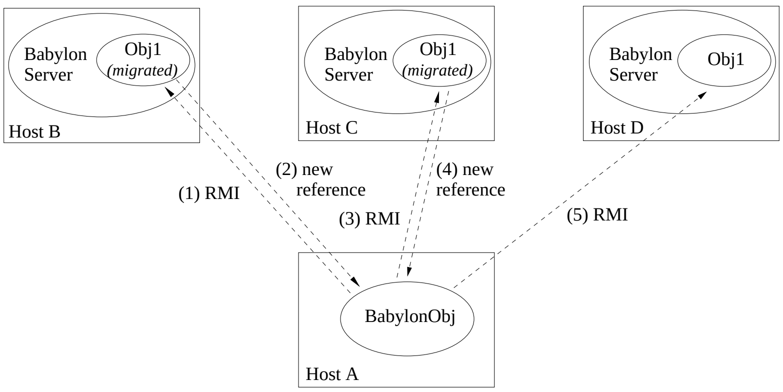

Figure 3.7: Remote Method Invocation After Third Party Migration

Updating remote references in this fashion is a form of lazy updating, since the references are not updated until they are used. An eager update approach would involve updating all references to a remote object immediately upon migration. This is not a reasonable choice for Babylon for two reasons. First, it would require remote objects to keep track of all objects containing references to themselves (the remote objects). This would be difficult to maintain, in addition to causing cycles which might degrade the effectiveness of the garbage collection system. Second, eager updating may update references that are no longer being used, thus wasting effort.

3.5 Additional Architectural Issues

3.5.1 Design of Babylon's Remote Method Invocation

This section is provided to demonstrate the flow of a remote method invocation request through Babylon. Figure 3.8 provides an overview of the path taken once a method invocation request is submitted through a `Babylon.rmi()` call.

The diagram begins with a user making a method call through the `Babylon.rmi()` interface. As previously described in Section 3.1.1, this method is passed two mandatory parameters[8]. These represent the object reference, and the method name. As the object reference (the `BabylonObj`) is only a proxy, the first step taken by `Babylon.rmi()` is to access the real `RObject` reference. This reference is then used to make a Java RMI call (`RObject.rmi()`), to the remote object. If the method executes successfully, `Babylon.rmi()` will simply return the result. However, if some other event occurs (e.g., migration, method returns an exception, method does not exist), then `Babylon.rmi()` must handle an exception.

[8]Additional optional parameters are not shown.

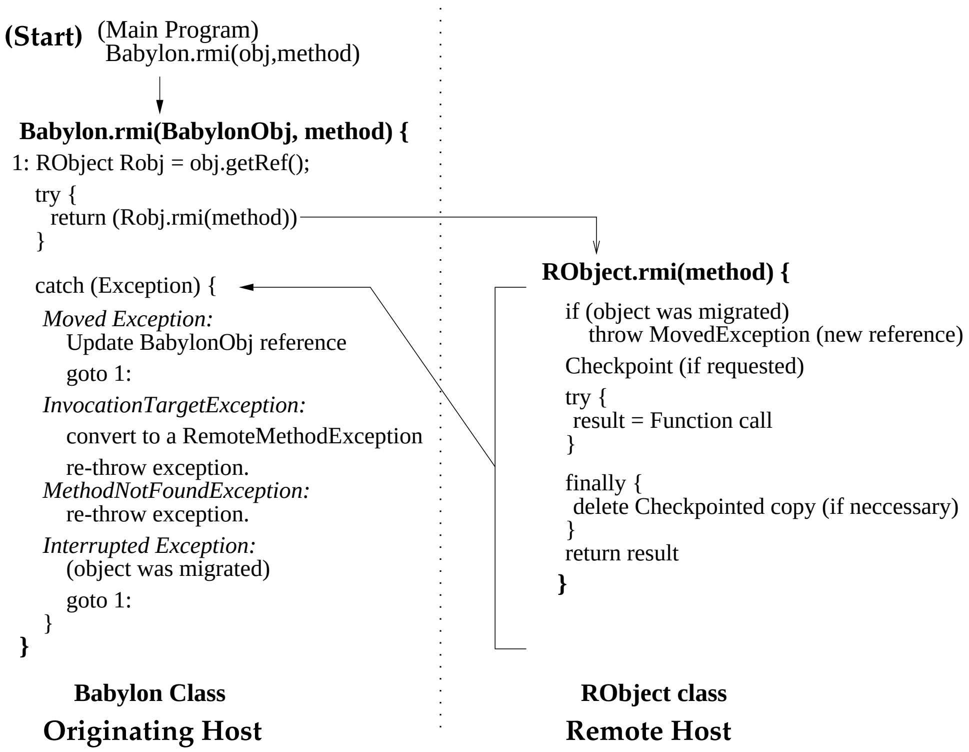

Figure 3.8: Internal Path for Remote Method Invocation.
While a `goto` is used to simplify the diagram, there are no `goto` statements in Babylon.

RObject.rmi()

On the remote host, the `RObject` handles the RMI request through the following sequence. Initially it checks to see if the actual remote object still exists on this host, or if it has been migrated elsewhere. If the remote object has been migrated, all that will remain is a "forwarding address" in the form of a reference to another `RObject` on another server. This will be returned to the `Babylon` class through a `MovedException`. If the object still exists on the current remote host and checkpointing is active, a checkpoint is done. Following checkpointing, the method name is used to call the appropriate method, using Java reflection techniques[9]. If the method throws an exception, it is not caught within the `RObject.rmi()` method, thus getting returned to the `Babylon.rmi()` method which will handle the exception. Finally, once the method completes execution, the copy of the object created for checkpointing purposes is deleted, since the actual copy of the object is once again in a static state. Deletion of the checkpointed copy will occur even if an exception was thrown by the method.

Babylon.rmi() catch clause

The `Babylon.rmi()` method has a number of `catch` clauses to handle the following possible exceptions.

MovedException: This indicates that the object has been moved and the reference used to make the method invocation is no longer valid. The exception returns a new reference to the object. The new reference is used to update the user's `BabylonObj` and then the remote method is reinvoked using the new updated reference.

InvocationTargetException: This indicates that an exception has occurred within the method(e.g., `ClassCastException`), and the exception should be returned to the user. The exception is embedded in a `RemoteMethodException`, and returned to the user. More details about how and why this is done are found

[9] This was originally done in Ajents and is not further described here.

in Section 3.5.2.

MethodNotFoundException: This indicates that the method requested by the user did not exist. It is returned to the user program, where it may be caught, if desired.

InterruptedException: This indicates that the method was interrupted. This is the approach used to halt execution of the thread executing the remote method. Such interruption is required for immediate migration. Thus, it also indicates that the remote object no longer exists on this remote host, as it has been interrupted and is being migrated elsewhere. The `Babylon.rmi()` method is repeated so that it will find the object's new location (through multiple `MovedException`'s if necessary, as in Figure 3.7), and then re-invoke the remote method. Note that a `MovedException` is not used in this case because at the time the method was interrupted, migration had not yet begun.

It should be noted that this design is effective in reducing the remote method invocation overhead imposed by Babylon as much as possible. Only a single RMI call is required for successful method invocations. Additionally, the use of exceptions for passing information between the originating and the remote host only occurs in uncommon situations (i.e., migration requests, unsuccessful method invocations). In other words, Babylon has been optimized for the common case which is the execution requests that are successful. While the minority of cases which require special handling will be less efficient as a result of this compromise, these situations already produce overhead in excess of a standard RMI call, so additional Babylon overhead will not impact these situations significantly. Thus we believe this to be an efficient implementation of RMI.

3.5.2 Exception Handling in RMI

Although synchronous and asynchronous remote method invocations are supported in Ajents, the Ajents' versions of RMI and ARMI do not return exceptions in remote methods to the user. Babylon is able to incorporate exception handling into RMI and ARMI by using the existing Babylon architecture for the remote invocation of methods.

There are two main concerns in designing an exception handling mechanism.

- The exceptions should be produced in a way which is similar to the standard Java model for exception handling. This allows the user to handle exceptions in a familiar manner.

- The exception model should not force the user to handle all possible exceptions if their method throws a limited subset of exceptions. This is a concern because the Java language requires advance declaration of all potential exceptions. Since `Babylon.rmi()` may call a method which may potentially throw any exception, Babylon can not be certain, in advance, which exceptions might be thrown.

Babylon addresses these concerns by creating a single exception type `RemoteMethodException` which extends `RuntimeException`. `RuntimeException` is a class of exceptions which are not required to be explicitly handled. Unlike the standard case where potential exception throwing code must be located within a `try/catch` block, code which throws a `RuntimeException` need not have a `try/catch` block. The Babylon class `RemoteMethodException` contains, as a member variable, the exception object which was thrown. This permits exceptions to be handled by simply accessing the exception within the `RemoteMethodException` through a `getTargetException()` method. `RemoteMethodException` is similar to the standard Java API `InvocationTargetException`, with the exception of the former being a subclass of `RuntimeException`. By subclassing `RuntimeException`, Babylon provides the user with the option of catching exceptions. If the user is aware that the method being called can not produce exceptions, then no `try/catch` block is necessary. However, if exceptions may occur, they can be handled in a familiar manner. A disadvantage of this approach is that it places an additional burden upon the programmer to be aware of potential exceptions, without relying upon the compiler to double check that all possible exceptions have been handled.

The method of returning exceptions thrown during a synchronous remote method invocation was detailed in Section 3.5.1. Returning an exception to a user program when the exception was thrown during an asynchronous remote method invocation is more challenging. Initially, the user calls `Babylon.armi()` to execute a method of a remote object. This method (using the technique implemented in Ajents) spawns a separate thread which executes the remote method invocation, and provides a `Future` object in which the return value is stored. The steps executed on the remote host, identical for rmi or armi calls, involve `RObject.rmi()` processing the method invocation as previously described in Section 3.5.1. In the case when an exception is thrown, `RObject.rmi()` throws an exception rather than returning a return value. This exception is caught by the thread which invoked the remote method, and the exception is stored in place of the return value within the `Future` object. At some point later, the user calls `Future.get()`. This method checks to see if the object contained within the future is an exception. If it is an exception, the exception is thrown, for the user to catch. Otherwise, the object returned by the remote method is returned from the `Future.get()` method, as shown in Figure 3.4.

One drawback of this design is that exceptions caused by methods invoked asynchronously are only caught when the `get()` method is invoked upon the future object. An alternative approach is to attempt to return an exception to the thread which invokes `Babylon.armi()` (which may not be the same thread as invokes `future.get()`). However, this is technically challenging. Since Java lacks signal handling methods, there is no simple way to throw arbitrary exceptions at arbitrary points in time, which would not be expected by the code being executed at that time. Even worse, it is possible the thread may no longer exist by the time the method completes.

3.5.3 Notes on Garbage Collection

While the Java remote method invocation specification [42] describes procedures for the garbage collection of distributed objects, Babylon can not blindly rely upon these techniques, but rather must assist them to be as effective and efficient as possible. As mentioned in Section 2.1, the Java virtual machine keeps track of all references on any hosts to remote objects. As Babylon features objects that are both mobile, and potentially long-lived, the abilities of the garbage collection system to keep references properly updated may be stressed. The default distributed garbage collection system was built for a virtual machine which does not provide migration or other features provided by Babylon. Additionally, it is important that memory is reclaimed by servers in a timely fashion when an object departs (either through migration, or the completion of the client program).

Passing remote object references between machines involves increased activity in the garbage collector when compared to the passing of standard objects. Not only must the reference be serialized, transfered, and deserialized, but the object reference count must be updated. Therefore, the machine upon which the object lives must also be contacted in order to update this count. Babylon endeavors to minimize the number of times this task must be completed. Object references are transfered only when required. This occurs solely during migration, when the object is moved, and references to the new location must be made available. However, by doing a lazy update of references, as described in Section 3.4, these updates only occur when and if required.

In addition to minimizing reference passing, Babylon also tries to ensure that dangling references are eliminated. This is a key step after migration. A careful elimination of unused references is necessary because of the forwarding system used by Babylon migration. Since the `RObject` is still required in order to forward remote references (as described in Section 3.4), it can not be garbage collected until all references to it are no longer in use. However, the RObject contains other variables which use up memory and/or contain references to additional objects. Eliminating the now unused references is done by nulling out all variable references contained within the `RObject` *from* which the remote object was migrated. By assigning references to null, the garbage collector is then immediately able to determine that the objects which were formerly in use by the `RObject`, are now unused, and available for collection. Thus, eliminating these references is necessary in order to reclaim memory, and to keep accurate reference counts for other objects.

By keeping these considerations in mind, Babylon is able to assist the distributed garbage collection mechanism in the efficient completion of its task.

Chapter 4

Remote I/O

Babylon relies upon Java security measures, such as a security manager object, to prevent remote objects from affecting hosts in an undesired manner. Operations disallowed by the security manager include all standard Java forms of I/O, including console, file, and socket operations. Thus, interaction with objects executing upon remote hosts is limited to method invocations. In addition to limiting the usefulness of Java based distributed systems, a failure to provide I/O facilities also makes debugging distributed and mobile programs more complicated. To our knowledge, this limitation exists in all other existing Java based systems, including all the systems described in Chapter 2.

Babylon includes features designed to allow input and output to be processed by remote objects, using an API that is similar to standard forms of Java I/O. Additionally, from the perspective of the user on the originating host, the remote object produces output and requests input as if it were actually executing upon the originating host.

4.1 Issues in Remote I/O

As can be seen in the coverage of related work in Chapter 2, there has been little attention paid to equipping distributed Java systems with remote I/O features. In fact, providing user programs with the ability to perform I/O operations remotely has received little attention in general [10]. We present here some of the issues that need to be addressed in implementing remote I/O, and some related work.

As identified by Dick [9], there are two issues that must be addressed by remote I/O systems: identifying the origin of the data, and data input and output synchronization. Data origin identification is an issue when multiple remote objects are producing data into a single buffer on the originating host. For debugging, and other purposes, users may wish to identify which remote host was the source of the output. While Dick and others (such as PVM [14]) transparently append source-host information, Babylon leaves this task up to the user, providing a method which will return the object's current host name, so that objects may choose whether to append origin information[1]. The other issue is the synchronization of input from a single source to multiple remote objects requesting input, and the reverse, synchronizing output from multiple remote objects into a single destination on the originating host. For

[1]Remote objects would implement the following example: `Babylon.println(this.getHost() + OutputString);`

reasons of simplicity, Babylon does not address this issue, leaving all synchronization to the user. We note, however, that this does not cause any potential system dangers, because the underlying Java I/O methods are themselves thread-safe. An additional, often unwritten, issue that must be addressed is ease of use. Babylon addresses this by ensuring that all Babylon I/O methods perform identically to their Java I/O counterparts. As will be discussed, this is done through the use of wrapper classes.

The reader is referred to Dick [9] for a lengthier discussion of other distributed and parallel I/O systems; we summarize other approaches here. PVM [14], a widely used library for parallel programming, provides only console output facilities for remotely executing jobs, input to these jobs is not allowed. MPI [39], a popular message passing interface for parallel programming, does not define any standard for remote I/O, although newer standards attempt to address this issue. For both PVM and MPI, deficiencies lie in their lack of an easy to use interface, and lack of support for I/O on arbitrary user-defined objects. Dick [9] addresses these concerns through an extension to the C++ `iostream` classes. Remote Unix [27], an implementation of a cluster load-sharing facility, implements I/O facilities similar to Babylon, in that it intercepts all I/O method invocations made by the remote object and transparently moves them onto the originating host. Another cluster load-sharing facility Utopia [48] addresses console I/O issues in the same manner as Remote Unix, but eliminates problems with remote file I/O by requiring that all remote hosts share a common file system.

Although Babylon's emphasis on Java-based parallel and distributed objects addresses much different areas than the systems above, it shares the same concerns as those systems: ensuring that remote objects have access to I/O facilities that look and act the same as if the objects were executing locally.

4.2 Design

The design of Babylon's I/O subsystem can be described from different points of view. We begin with an example of a user-created remote object which takes advantage of remote I/O, then describe how the object hierarchy from Section 3.2 is altered in order to provide these features. Finally, we discuss the issues involved in this design.

4.2.1 A Short Programmer's Guide

Figure 4.1 is an example of a remote object which includes a method programmed to read data from a file, do some computation and then print output to the console. In order to take advantage of Babylon's I/O features, remote objects must extend `RemoteObj`. This is shown in line 1. To access I/O, a `RemoteIO` reference is returned by the `getIO()` method (line 4). The `RemoteIO` object referred to by this reference lives on the originating host, and includes methods to create new instances of each type of

I/O supported by Babylon. In the figure, a new `RemoteBufferedReader`[2] object is created (on the originating host), with a file name on the originating host passed as a parameter, which will serve as the input source (line 5). The `RemoteBufferedReader` object reads from the file (line 6) which is then processed (line 7). The output produced is then printed to console through `io.println()` (line 8). Console actions are done directly through the `RemoteIO` object. The reasons why console actions are done directly through the `RemoteIO` object while file actions require their own objects will be explained in Section 4.3.

```
1  public class SimpleObj extends RemoteObj implements Serializable {

2   public void useFile() {
3    String line, output;
4    RemoteIO io = getIO();
5    RemoteBufferedReader rbf = io.newRemoteBufferedReader
                                 ("/home/izatt/SampleFile");
6    while ((line = rbf.readLine()) != null){
7     output = doComputation(line);
8     io.println(ouput);
      }
    }
   }
```

Figure 4.1: Code Example of Remote I/O Use in a Remote Object

4.2.2 Object Hierarchy Design with I/O

As mentioned, in order to access Babylon's I/O facilities, remote objects must extend `RemoteObj`. This class provides access to a remote reference which refers to an I/O object (`RemoteIO`) created upon the originating host during Babylon's initialization.

[2] `Babylon.io.RemoteBufferedReader` is a wrapper class around `java.io.BufferedReader`.

By using the `getIO()` method (inherited from `RemoteObj`), remote objects can access this `RemoteIO` object and the I/O facilities contained within. The `RemoteIO` object is a Java RMI object which provides a point of contact for access to all of Babylon's I/O systems (console, files and sockets).

Extending `RemoteObj` adds another layer to the object hierarchy displayed in Figure 3.6. Figure 4.2 shows the object hierarchy design for those objects which extend `RemoteObj`. The `BabylonObj` and `RObject` objects serve the same purposes as described in Section 3.2, as a user held proxy, and Java RMI object, respectively. The object hierarchy changes begin at the user object level. The remote object created by the user (represented by `RemoteObj` in the figure), inherits a reference to the `RemoteIO` object located upon the originating host. It is through this `RemoteIO` object that all contact with Babylon's I/O system occurs.

In order to best emulate the functionality of regular Java I/O, Babylon creates wrapper classes around the basic input and output classes offered by Java. The key difference is that Babylon's wrapper classes are defined as Java RMI objects. An example implementation of a Babylon wrapper class is shown in Figure 4.3. All objects created using these wrapper classes execute upon the originating host once created, only providing a remote reference to the user, through which they may be accessed. Standard Java provides an interface for users to access each class's methods and Babylon copies this by providing a similar interface, accessible through remote,

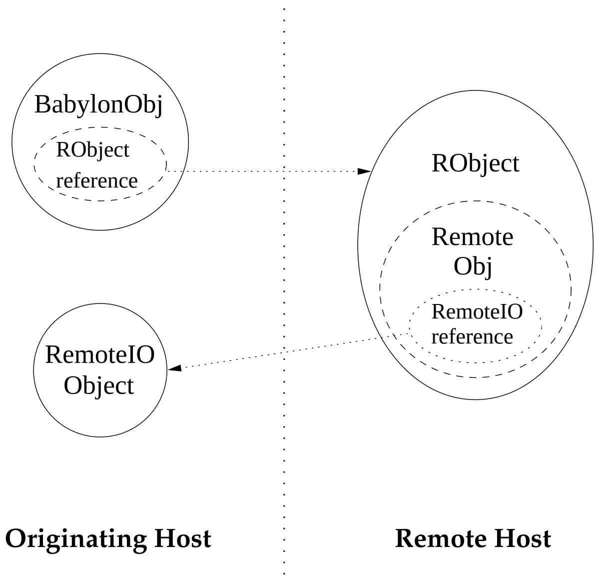

Figure 4.2: Object Hierarchy Design with I/O.
Arrows represent the referencing structure, with the source being the remote reference, and the destination being the referenced object.

rather than local references. This is discussed in more detail in the next section.

Figure 4.4 provides a demonstration of how the internal wrapper objects are created and accessed. In the figure, a remote object wishes to access files, and thus wishes to create a `RemoteBufferedReader` object. To accomplish this, 4 steps are taken. (1) The `RemoteIO` reference, located upon the remote host, invokes a `newRemoteBufferedReader()` method on the `RemoteIO` object, located upon the originating host. (2) This method creates a `RemoteBufferedReader` object and returns a remote reference to it. (3) This reference is returned to the remote object to provide it with a point of contact with the `RemoteBufferedReader` object. (4) The reference may then be used to directly contact the `RemoteBufferedReader` object and obtain services from it.

```
public interface RemoteBufferedReader extends Remote {
  public int read() throws RemoteException, IOException;
  public String readLine() throws RemoteException, IOException;
}


public class RemoteBufferedReaderImpl extends UnicastRemoteObject
               implements RemoteBufferedReader, Serializable {

   BufferedReader in;

...

  public synchronized int read() throws
                      RemoteException,IOException {
    return in.read();
  }

  public synchronized String readLine() throws
                      RemoteException,IOException {
    return in.readLine();
  }
}
```

Figure 4.3: Code Example of a Babylon I/O Wrapper Class.
This is Babylon.io.RemoteBufferedReader which is a wrapper around java.io.BufferedReader. The code shows both the interface class, and the implementation class.

4.2.3 Babylon's I/O Design Issues

The decision to use RMI to support I/O within Babylon was made for a couple of reasons. The most important is that remote method calls provided by RMI are fairly simple to use. In accessing I/O, objects operating on remote hosts will wish to contact sub-systems (file, terminal or network) operating on their originating host, and that contact will occur synchronously (i.e., through method invocations). These

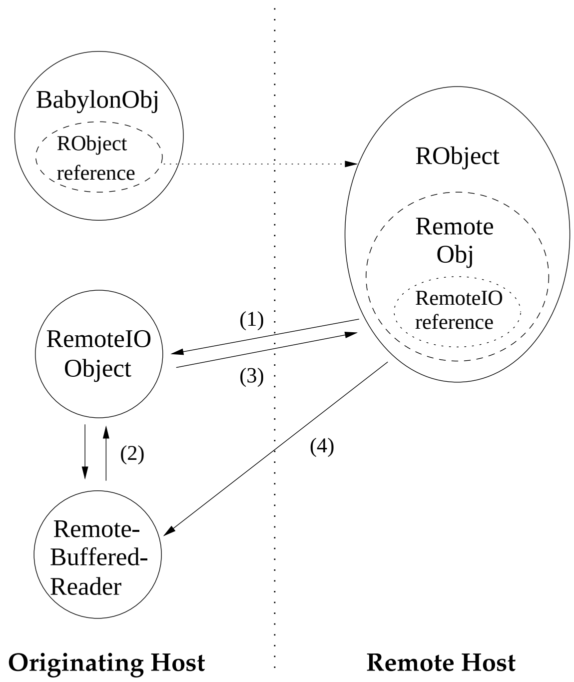

Figure 4.4: Remote I/O Design

requirements are precisely those for which RMI was designed and implemented. A second key reason involves the location transparency of remote RMI references. This is vital because the remote object, which contains a remote reference to an I/O object on the originating machine, could itself be migrated. With RMI references, the reference is still valid, even after the remote object (along with the `RemoteIO` reference) gets migrated. An alternative architecture might require us to design and implement a method to ensure that after migration, remote I/O references still refer to the correct location.

Babylon's I/O subsystem is designed to satisfy ease of use, security, and transparency concerns. The issues of ease of use, and transparency have been partially demonstrated in Figure 4.1. The object programmer is aware of the remote nature of the I/O being performed, and must expend only a small additional amount of effort altering class names, and accessing the `RemoteIO` object in order to create direct I/O facilities. However, the amount of extra programming is small. Once I/O objects are created (such as `RemoteBufferedReader`), the techniques for accessing them are as simple and direct to use as the techniques in the `java.io` classes. Additionally, once they are created, accessing remote I/O classes is done transparently. No additional programming effort is required and security is provided by using the security manager to disallow any object from accessing I/O without going through Babylon's specified interface.

One important consequence of requiring objects to extend `RemoteObj` in order to access I/O is that objects that want to use Babylon's I/O features can not extend any other objects. This problem is a consequence of Java's lack of support for multiple inheritance. An alternative method would be to have objects include a member variable containing the features of `RemoteObj`, rather than inherit from it. Technically, this approach is feasible, but it has other advantages and disadvantages. Inheritance is generally simpler to use and program. Since the parent object (`RemoteObj` in this case) is known by the Babylon system, the system can access member variables and independently set and change them. This means that the `RemoteIO` reference can be initialized by the system without user intervention. If this reference were held inside a member variable of the remote object, the remote object would have to include code to explicitly complete all initialization currently handled internally by the `RemoteObj` object. Thus inheritance hides the internal structure of `RemoteObj` object features, resulting in simpler programming of remote objects.

While the `RemoteObj` object is currently used primarily for storing the reference used to access Babylon's I/O features, it could easily be expanded to provide other features. Any system features which must be accessed by the remote object may be added using this approach, without affecting previously written objects, or adding complexity to user objects. For example, Babylon also provides an interface through the `RemoteObj` class which allows a remote object to access data about its current location. This, and other future additions are simple to implement using the current design, and are reflected to the user object through a changed API. However, if user objects were forced to implement and initialize their own references to data provided by Babylon, every addition to the Babylon API would result in an increase in the amount of code required on the part of the user.

Additionally, the use of inheritance allows Babylon to determine whether remote objects are going to be using the features contained in `RemoteObj`. By using Java's `instanceof` method, Babylon is able to determine whether an object has extended `RemoteObj`. This can be of use in determining whether features like I/O are going to be used, and making optimizations based upon that knowledge. For example, if Babylon knows that remote I/O will not be used, then it does not need to instantiate a `RemoteIO` object for the corresponding remote object.

The following sections (Sections 4.3 - 4.5) describe the challenges found in implementing this design for the types of I/O that are supported.

4.3 Console I/O

Supporting reads from and writes to the console introduces challenges that are different from file accesses because of the atomic nature of the console. While multiple objects may simultaneously open the same file, and a single object may hold multiple different references to a single file, Java does not allow this behavior with the console. Only a single point of access to the console is possible per Java virtual machine. In other words, even if multiple objects hold a reference to the console, they would be sharing the same reference, while with files, they each could be holding a unique reference. This is represented by the static variables `System.in`, `System.out` and `System.err`, which provide user programs with access to standard input, output, and error respectively.

Babylon duplicates this behavior by allowing access to the console through the `RemoteIO` object. Although multiple `RemoteIO` objects may exist upon a single host (Babylon creates one for each remote object), the `RemoteIO` class contains static references to the System variables, thus preventing multiple instances of `RemoteIO` from each trying to bind a new I/O stream onto the console. Thus, only single instances of console variables can ever exist under Babylon, mirroring the situation in standard Java. Finally, whereas file I/O requires instantiation of a remote wrapper class, console I/O avoids this step by using the `RemoteIO` object, thus, reducing overhead.

4.4 File I/O

Supporting file I/O is straightforward, once the over-all I/O design is considered. This is because there are no file system access issues that arise in remote situations as opposed to local situations. In most parallel and distributed systems, issues arise surrounding synchronization of multiple, potentially remote, processes attempting to access a single file. However, standard I/O in Java is already designed to work correctly in a multi-threaded environment. By designing access to the file system through wrapper classes designed to be accessed remotely, file access through Babylon occurs in the same manner as in standard Java. Multiple accesses to the same file, by the same or different objects, is allowed in both standard Java and in Babylon. In both cases, the underlying virtual machine and operating system take care of synchronization issues.

As a result of this design, Babylon supports only independent (rather than shared) file access. That is, if multiple remote objects wish to access the same file, they may do so, but each must separately open the file and use its own file pointer. They may not share a single file pointer. Thus, if it is important to synchronize writes to a file, some external synchronization technique must be used.

In Figure 4.5, we provide an example of how Babylon is able to ignore file synchronization issues. The figure demonstrates how two remote objects are each holding a reference to a RemoteBufferedReader object on the originating Host (1). Each RemoteBufferedReader holds a reference to some file on the originating host (2). File system integrity is maintained by the operating system and the Java virtual machine. Meanwhile, any additional synchronization between the objects is left to the programmer, rather than enforced by Babylon. Synchronization is the type of task for which enabling socket use by remote objects might be required.

4.5 Socket I/O

Supporting socket I/O in Babylon involves supporting the following key attributes. Remote objects must be able to open sockets which connect to arbitrary hosts. The reverse must also be true. Remote objects must be able to create sockets which accept connections from arbitrary hosts. In addition, socket connections should remain valid even if the remote object migrates.

The challenge in creating sockets is that there exists an external host in addition

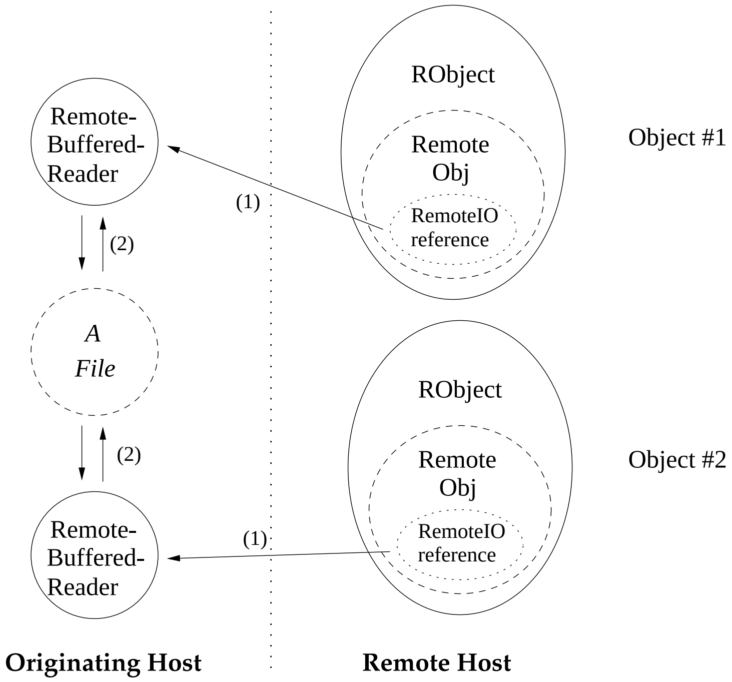

Figure 4.5: File I/O Diagram.
This diagram demonstrates two remote objects accessing the same file, each through its own RemoteBufferedReader object.

to the originating and remote hosts. This third host should be able to open a socket, and communicate with a remote object, without any knowledge of the "remote" or migratory nature of the object it is communicating with. In Babylon, it is possible for a remote object to create a socket, wait for a connection from a third party, communicate through the socket, migrate, and then continue communicating. In addition, the third party is unaware that the remote object has migrated, or even that the remote object does not live on the originating host. Thus, there are no special requirements on the part of the program running on the external host.

It is the overall design of remote I/O in Babylon that makes it possible to ensure that sockets work correctly even after migration. Sockets in Babylon are implemented identically to the other forms of I/O. That is, wrapper classes have been created around `java.net` classes, where the the Babylon versions are remote (RMI) versions with identical methods. This forces the actual socket to be created on the originating host, which is why third party transparency is possible. In order to allow reading from and writing to the socket, we reuse the same I/O stream wrapper classes used for files and demonstrated in Section 4.2.1 (e.g., `RemoteBufferedReader`). In this case, the stream source for the `RemoteBufferedReader` would come from a socket, instead of a file.

The first problem to solve is the question of which host does a third party contact in order to set up the socket. By making this the originating host, a single, stationary host serves as the point of contact for the remote object. This ensures that the third party is unaware of migration, since the socket always remains connected to the originating host. A second problem solved revolves around the remote object's ability to access the socket on the originating host even after the object has been migrated. However, because RMI is used, the remote object retains an RMI reference to the socket wrapper class. This RMI reference operates correctly after migration, thus ensuring that access to the socket continues. It is these problems that led to rejection of the possibility of having sockets connect directly to the remote host. Such a design would not be able to transparently handle migration because the sockets would need to be closed upon one host and then re-established on a second host after migration. Clearly, this solution would not be transparent, as the third party would have to become involved in re-establishing the socket connection.

Figure 4.6 provides a diagramatic example of how two objects communicate with each other through a socket. Initially, object #1 creates a server-socket using Babylon's `RemoteServerSocket` class. Object #2 connects to this socket using Babylon's `RemoteSocket` class (1). Note that object #2 uses the originating host as the host to which it's socket will connect, and object #1's server-socket will report that it has been connected to by a socket on the originating host. Finally, the sockets may exchange information (2,3) in the same manner as if they were regular Java sockets.

4.6 I/O Limitations

While Babylon's design supports the forms of remote I/O described, there are some limitations placed upon I/O by this design.

While remote objects that use I/O can be migrated, and their I/O references will continue to work correctly after migration, this is not always the case with one form of migration. The immediate migration of executing objects can cause problems when used in concert with I/O. The problem here is identical to the checkpoint consistency problem (previously referred to in Section 3.4). The problem arises if a remote ob-

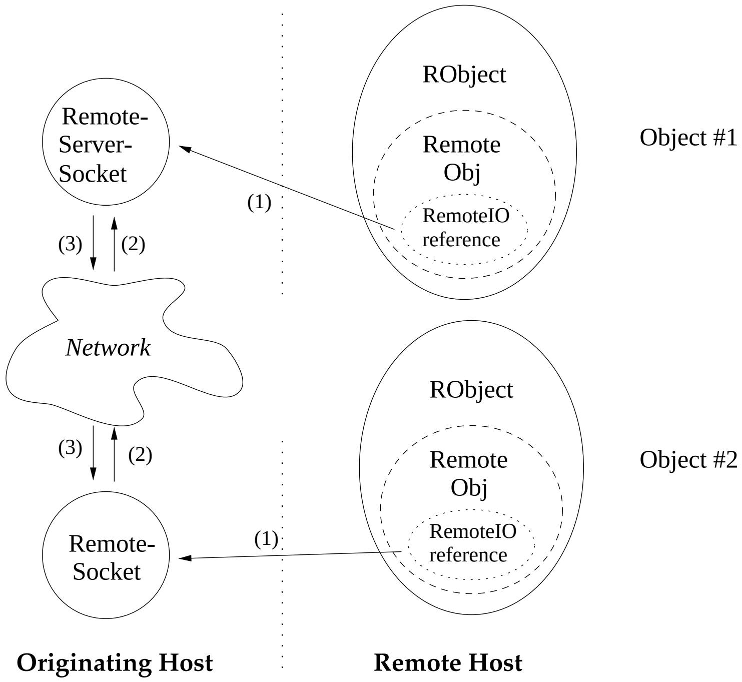

Figure 4.6: Socket I/O Diagram.
This diagram demonstrates two remote objects communicating through sockets.

ject is invoking a method which involves I/O, and gets interrupted, migrated, and restarted. The I/O may then be repeated, potentially causing a consistency problem if such repetition is unexpected.

Another drawback with the current implementation of remote I/O is the use of RMI. While using RMI simplifies the implementation of remote I/O, it also means that additional RMI overheads are added to every I/O access. Thus, it is important for programmers to keep this in mind, and choose larger data transfer sizes where possible.

Babylon's I/O support is partly provided through wrapper classes based upon standard Java classes. This means that for every type of I/O stream that Java provides, an equivalent Babylon wrapper class must be created. The current implementation of Babylon does not include a complete set of wrappers; a sufficiently large enough subset[3] is supported in order to successfully test and demonstrate console, file and socket implementations. As demonstrated in Figure 4.3, completing the full set of wrapper classes is a straightforward exercise. In fact, it could be generated automatically.

[3]Specifically, the following `java.io` classes have `Babylon.io` equivalents: `BufferedReader`, `ObjectInputStream`, `ObjectOutputStream`, `PrintWriter`, `ServerSocket`, `Socket`.

Chapter 5

Scheduling

Although Babylon's primary contribution is made in providing a wide range of features for use by distributed objects, Babylon would be useless without mechanisms to connect user objects requiring resources with remote hosts offering resources, in the form of a scheduling system. A complete scheduling system consists of both mechanisms for controlling and policies for governing the allocation of resources, and would have to implement client-server connection mechanisms, as well as handling relationships between multiple scheduling servers. In addition, the scheduling infrastructure would have to scale properly, potentially on an Internet-wide scale.

Developing a complete scheduling system is beyond the scope of this thesis. Instead, Babylon implements a basic scheduling substructure intended to function adequately for the purposes of testing the features implemented in Babylon thus far. The current Babylon scheduling system implements a simple scheduling server, as well as mechanisms to provide references to available Babylon servers that have identified themselves to the scheduling server. In addition, Babylon's scheduling system seamlessly and transparently handles the dynamically changing availabilities of Babylon servers.

In this chapter we discuss the implementation of the simple scheduling system and the areas in which additional scheduling research and development is required. Section 5.1 is an overview of the different scheduling issues that must be considered by distributed systems such as Babylon, and provides references to other work in areas relevant to Babylon. As well, we discuss how other scheduling research and systems can be combined with Babylon. Section 5.2 discusses the scheduling infrastructure currently implemented in Babylon. The current infrastructure operates on a very basic level, with a simple scheduling server acting as a single point of contact for clients and Babylon servers. The Babylon scheduling server currently implements a simple round-robin strategy for distributing requests among available servers.

5.1 Issues in Scheduling

Babylon contributes two main features to the area of load-balancing systems: the ability to create objects on arbitrary remote hosts, and the ability to migrate these objects to other arbitrary remote hosts. In order to make object location and relocation transparent, these abilities are entirely controlled by internal Babylon structures, with no information provided by the user. Thus, in order for load-balancing to be successful, Babylon's scheduling system must define three main policies and support the infrastructure to implement these policies.

I. A *location policy*, which defines what host should be selected as the host for objects being created and/or migrated.

II. A *migration policy*, which defines what objects should be migrated and at what point.

III. An *information-sharing policy*, which defines what information should be shared between Babylon servers and the scheduling servers, and how accurate this information must be[1].

While Babylon does not define or implement any of these policies, we provide a brief discussion of work related to these issues.

Utopia [48] is a load sharing facility designed for large heterogeneous distributed systems. The system is designed to distribute tasks among a group of eligible machines, given some *a priori* knowledge of the task. Utopia implements a system whereby machines are divided into virtual clusters, so that scheduling is done accurately and efficiently within clusters, but less accurately and efficiently between clusters. While this design works quite well, it relies upon significant user input and set-up, which are designed to be minimal in Babylon. Utopia defines both location and information-sharing policies, but explicitly avoids allowing migration or defining

[1]The classic trade-off here is between gathering large amounts of information and sharing it frequently, which is very accurate but inefficient, versus minimizing sharing, which is efficient but inaccurate.

a migration policy.

The justification for disallowing migration in Utopia draws from a paper by Eager, Lazowska and Zahorjan [12], which shows that the benefits of preemptive migration for load balancing are small, compared with schemes which load balance only upon the start of a task. However, these results have been challenged by Downey and Harchol-Balter [11] who argue that Eager *et. al* used a weak model to describe the workload encountered by distributed load-balancing systems. Downey and Harchol-Balter also complete a study [18], demonstrating the "performance benefits of preemptive, implicit load balancing strategies that assume no *a priori* information about processes". This study describes very closely the workload that Babylon can expect. All load-balancing in Babylon is done by the system, with no user input to provide insight as to resource requirements such as CPU or I/O usage. As well, Babylon has no future job knowledge, such as arrival times, to allow schedulers to make more informed decisions.

More recent work examines in greater depth the ways in which objects should be scheduled for migration. FarGo [19] is a system which implements the idea of *complets*. A complet is a grouping of objects which have close relationships. FarGo allows the designation of relationships among complets so that not only are objects within a complet always migrated together, but the user may define multiple complets which must always be or are suggested to be, co-located.

A final issue in deciding what host to locate objects on involves deciding how to compare the load on heterogeneous hosts, where all of the resources on the hosts may be incomparable. Utopia [48] relies upon user input in order to define which clusters are high-powered CPU clusters, and which are designed for heavy I/O use. Amir *et. al* [2] define algorithms for converting multiple heterogeneous resources into a single numeric value which defines the availability of a host, and the cost of putting additional objects on that host. They demonstrate that this method can be used to minimize the cost among a group of heterogeneous hosts, such as a collection Babylon servers forming a cluster.

5.1.1 Babylon Information

As a result of Babylon's design, useful information about object size and lifetime can be provided to any Babylon scheduling server in order to make appropriate scheduling decisions. Because all remote method invocations are intercepted, and because the Babylon server is also involved in handling all object creation and migration requests, Babylon could keep track of a number of statistics related to the remote objects residing on the Babylon server. The following information could be collected for use by a scheduling policy. Note that collection of these statistics is not currently implemented. In order to determine the best destination for newly created or migrating objects, Babylon could measure the following items.

- In order to determine the least loaded server through a measure of computational demands being placed by individual objects upon the Babylon servers, Babylon could measure:
  - the number of remote objects executing methods per unit time.
  - the amount of time (percentage or aggregate) a remote object spends executing methods.
- In order to determine the relative memory requirements being placed upon the Babylon server, Babylon could measure the number of objects residing on a Babylon server at any time, or the number of objects resident over a period of time. A larger number of objects resident may indicate greater memory usage, although this hypothesis requires studying. As well, for migrated objects, it is possible to determine their actual memory footprint by checking the size of the serialized object at the time of migration.
- In order to provide a measure of object lifetimes, the aggregate amount of time a remote object has resided within the Babylon system could be measured. This would serve as a guide to the likelihood that this object will continue to reside on the server in the future (and could be compared relative to other objects).
- To consider the effects of migration, Babylon could measure the number of times individual objects have been migrated. This could serve as a guide in

deciding which objects should be migrated, if the Babylon server needs to reduce the number of objects resident upon it. Objects which have previously been subjected to migration costs are better candidates for future migration since the relative cost of migration versus object lifetime is less for these objects.

While this list is not complete, in our opinion it represents the some of the most important statistics Babylon could collect, and acts as guide to the type of measurements that can be made obtained using Babylon. Downey and Harchol-Balter [18] conclude that process lifetime distribution, a measurement which Babylon can simulate by tracking object lifetime distribution, can be used as the deciding measure for process migration. That is, when choosing a job to migrate, always choose the one which has lived the longest. This is because the longest job is most likely to be the one which will live the longest into the future and because it is the job which will suffer the least (relatively) consequences from migration overhead.

It would also be possible for Babylon to require certain information about Babylon servers be obtained as part of start-up (e.g., as part of a configuration file). For example, with knowledge of which Babylon servers have greater memory or computational resources, objects which are memory or CPU intensive can be allocated to these servers. This would be similar to the implementation of Utopia, which relies upon user input to define which hosts serve what purposes, and which types of hosts objects should have an affinity for.

5.2 Design

Ajents, from which Babylon descended, lacked sophisticated scheduling mechanisms. Objects found servers by examining a file and then trying to connect to the server. This was considered unacceptable for Babylon and thus a scheduling system was implemented, with the primary design goal being a simple, functional scheduling system. The design revolves around a scheduling server which is a point of contact, providing references to available Babylon servers upon request. The scheduling server maintains an updated internal database of available servers which is used to fulfill requests for available servers. In order to allow access to be completed through remote method invocations, the scheduling server is implemented as a Java RMI object. This offers simple transparent access to the remotely-located scheduling server.

The scheduling server makes the following methods publicly available to clients:

`availServer()`: This method returns a remote reference to an available Babylon server. Currently, the available Babylon servers are stored in a list, and round-robin ordering is used to decide which available server is returned for use by the calling application. Improved scheduling policies would be implemented by modifying this method.

`registerServer(BabylonServer reference)`: This method is used by Babylon servers in order to add themselves to the list of available servers maintained by the scheduler. As part of the initialization of every Babylon server, this method is invoked.

`unRegisterServer(BabylonServer reference):` This method is used by Babylon servers in order to remove themselves from the list of available servers maintained by the scheduler. This method is invoked as part of the shut-down procedure of every Babylon server.

`getServerReference(String serverName):` This method returns a reference to an available Babylon server if one matches the host name. This allows for objects to request creation on, or migration to, a specific Babylon server by specifying a host name.

In order to ensure that all server arrivals and departures are known to the scheduling server, the `registerServer()/unRegisterServer()` steps are built in to the Babylon servers. In order to contact the scheduling server, Babylon servers require a parameter upon start-up representing the host name of the scheduling server. Note that currently, if a Babylon server should fail, the scheduling server would not be informed. This is a limitation imposed by the lack of fault tolerance features implemented in the current version of Babylon.

In order to enable scalable and fault tolerant scheduling in the future, all data contained in the scheduler (and thus, the scheduling server object itself) is serializable, and thus migratable. This allows for references to available Babylon servers, and Babylon server state information to be migrated to another scheduler, should a scheduler become overloaded. Scalable scheduling has been considered by many other systems, such as Javelin++ [29] and Utopia [48]. A scheme similar to that implemented in Utopia could be implemented, whereby all scheduling information is replicated onto a backup scheduler. Thus, if the primary scheduler fails, the backup will be capable of replacing it immediately. However, the current implementation of Babylon does not include any scalable scheduling or fault tolerant infrastructure.

Chapter 6

System Evaluation

6.1 Performance Evaluation

In order to demonstrate the practical relevance of Babylon we have evaluated the performance of Babylon using several micro-benchmarks and three simple distributed applications. These results show that the overheads introduced in using Babylon are not prohibitive.

6.1.1 Testbed

All experiments were conducted using a cluster of 8 SUN Ultra-1 workstations as Babylon servers. Each is equipped with a 143 MHz UltraSparc CPU and 64 MB of memory, and runs Solaris 2.5. The client (originating host) is an additional SUN Ultra. However, it is located on a different subnet from the Babylon server cluster. This is done so that the client does not share a file system with the Babylon servers, thus requiring class files to be remotely loaded at the time of remote object creation or migration. All machines are connected with a 10 Mbps Ethernet network. All experiments were conducted using SUN's JDK, Version 1.1.6. This version includes a just-in-time compiler which was activated for all experiments. Checkpointing was not activated (the Babylon default), except where specified.

While all of our performance testing was done in a homogeneous environment, Babylon does successfully run on a variety of platforms[1].

6.1.2 Benchmarks

In this section, we demonstrate the overheads incurred by each of Babylon's main features. This is done through a series of benchmarks, wherein individual Babylon features are tested.

Remote Method Invocations

We expect that Babylon will be commonly used to distribute long-lived objects among available machines, with remote execution requests occurring far more frequently than object creation or migration requests. Thus, it is vital that Babylon is able to perform RMI requests efficiently. Table 6.1 shows the results of our RMI request benchmark. A remote method was invoked repeatedly, passing an integer array of a specified size as a parameter (the size of this array is changed for each experiment). The results show the average time to complete a single remote method invocation (and thus

[1]Babylon has been tested on: Solaris, IRIX, Linux and Windows.

differences may not be statistically significant). These results demonstrate that while there is overhead involved in making remote method invocations using Babylon, it is not significantly greater than that incurred when performing a regular Java RMI call.

Parameter Array Size	(ints)	1	1k	10k	100k
Time per RMI - Babylon	(ms)	9	11	46	400
Time per RMI - JDK RMI	(ms)	4	7	42	397

Table 6.1: Remote Method Invocation Times

The results in Table 6.1 demonstrate that the design of Babylon's remote method invocations does not add significant overhead. By performing only a single Java RMI call, and handling error situations through exceptions, Babylon's remote method invocations approach a simple Java RMI call in efficiency. This is in contrast to Ajents' design, which involves multiple Java RMI calls in order to preemptively determine if non-standard conditions (such as object migration) exist.

Migration

A major feature of Babylon is the ability to migrate remote objects. Table 6.2 shows the times to migrate remote objects of various sizes. In conducting this experiment, we migrated an object which is composed of a single integer array (the size of this array is changed for each experiment). The object was migrated 64 times using 8 different machines (the 8 machines were used repeatedly in the same order). These results were then averaged to produce the results shown in the table.

Object Size	(ints)	1	1k	10k	100k
Time per migration	(ms)	334	334	397	769

Table 6.2: Remote Object Migration Times

We can see by comparing these results with those in Table 6.1 that there is a significant increase in cost for object migration when compared with a remote method invocation. This can be accounted for by several factors. Migrating an object requires that objects to be serialized, transfered to the target host and then deserialized. References to the object then need to be updated, including those needed by the Babylon server, the client, and the garbage collector. Additionally, when the object is migrated to a new host, classloading may be required. One final factor involved is that migration involves multiple RMI requests. An available Babylon server must be provided, either by contacting the scheduler, or by resolving a host name provided by the user. The object must then be contacted in order to obtain a reference to the object's current server. This current server must be sent a migration request and the destination server must also be contacted in order for it to prepare to receive the object. Each of these steps involves an RMI, and the passing of parameters, each of which must be serialized/deserialized.

Despite the number of fixed costs inherent in the method used for migration, the results in Table 6.2 are reasonable. Migration is an activity which should not occur frequently, and thus the overhead involved in migration should not be a major inefficiency.

Checkpointing

As previously described in Section 3.4, the use of checkpointing in Babylon enables the use of the checkpointing and rollback form of migration. However, the checkpointing of objects preceding a remote method invocation has an obvious negative impact on performance. Table 6.3 displays the results of a simple checkpointing benchmark designed to provide an idea of the costs involved in checkpointing. For this benchmark, an empty remote method was invoked repeatedly on an object whose sole member variable is an array of integers (the size of this array is changed for each experiment). The first two rows show the average time to complete a single remote method invocation, with checkpointing enabled, and without checkpointing enabled. The last row in the table shows the time difference between these two experiments. It represents the overhead added to an RMI call when checkpointing is enabled.

Object Size	(ints)	1	1k	10k	100k
No-Checkpointing	(ms)	6	6	6	6
Checkpointing	(ms)	8	9	19	115
Checkpointing Cost	(ms)	2	3	13	109

Table 6.3: Checkpointing Performance Results

The results presented in Table 6.3 show that the overhead of Babylon's checkpointing method is not extravagant, and as expected the overhead increases with the size of the object being checkpointed. Since the costs are considerably lower than migration costs (as shown in Table 6.2), the same relatively coarse grained applications whose execution times are not significantly impacted by migration would also not be significantly impacted by checkpointing. Our target applications are those whose method invocations made through Babylon will have a running time measured in seconds and minutes (as in the matrix multiplication case), and thus the overheads due to checkpointing will not be significant for these cases. However, checkpointing will have a considerable impact when there are a large number of invocations on methods with relatively short execution times.

A potentially more prohibitive cost of checkpointing is memory usage. Since checkpointing creates a complete copy of an object, memory usage for the checkpointed object effectively doubles. This can be an issue when dealing with large objects, such as matrices where object size is measured in megabytes.

Object Output

In order to demonstrate that the overheads involved in remote I/O are not severe, we conducted an I/O benchmark. Table 6.4 shows the results of our object output benchmark. In conducting this experiment, we save a remote object, which is composed of a single integer array (the size of this array is changed for each experiment), from memory to disk on the originating host. Note that in the Babylon version, this means that the object resides in memory on a remote host, but is saved to disk on the originating host. Thus it must be contacted, a save-to-disk method invoked, and then the integer array object must be transferred by Babylon to the originating host to be saved. These extra steps, not required on the local version, account for the overhead presented in the table. Measured in Table 6.4 is the time required to invoke a method of the object which saves the object to disk. In the Babylon case, this method is remotely invoked. The final line in Table 6.4 reports the difference between the Babylon version, and a local version.

Object Size	(ints)	1	1k	10k	100k
Local	(ms)	1	2	13	110
Babylon Remote	(ms)	25	25	70	500
Overhead	(ms)	24	23	57	390

Table 6.4: Object Output Results

The overhead presented in Table 6.4 are greater than, but on the same order of magnitude, as those presented in Table 6.1 for RMI. In each case, the majority of the cost is due to the serialization and transfer of the integer array. In this case, the Babylon version involves an RMI to request that the object saves itself to disk, as well as an additional RMI when the remote object transfers its data to the originating host. In addition, this benchmark involves disk overhead. Thus, the results presented in Table 6.4 approximately match expected estimates computed using the data in Table 6.1.

6.1.3 Applications

Parallel Matrix Multiplication

Table 6.5 shows the results of experiments conducted using a simple (non-blocked) matrix multiplication benchmark. While this is clearly not the most efficient implementation of matrix multiplication, the same implementation is consistently used in all cases in order to provide a fair comparison. The times reported measure the time to complete the matrix multiplication. In addition, the time required in order to transfer input and output data to/from the Babylon servers is included, where applicable. The first two rows of this table show the sequential execution times of versions of this program written in C and in Java (the columns show the results for different matrix sizes in integers). We note that across all matrix sizes the execution times of the Java version compares reasonably well with results obtained using C. The Java version is slower, by roughly a factor of 2. Although the Java version is slower, we believe that for a number of programmers and coarse-grained applications the benefits obtained from the ease with which Babylon programs can be implemented and executed across a wide variety of platforms will outweigh the costs of decreased execution times (relative to applications implemented in C).

The third row of Table 6.5 shows the execution time of the parallel version of the matrix multiplication program when executing using one remote server. By comparing the second and third rows of the table we see that the overheads incurred by Babylon, in switching from a simple sequential matrix multiply to a single-server Babylon version, is low.

Matrix Size	(N)	200	500	640	800	1024
Sequential C Time	(s)	4.0	67	139	280	601
Sequential Java Time	(s)	7.5	131	286	574	1272
Time on 1 Server	(s)	7.7	134	292	580	1305
Time on 8 Servers	(s)	1.5	20	40	78	172
Speedup		5.1	6.7	7.3	7.4	7.6

Table 6.5: Matrix Multiplication: Execution Time Comparison and Speedup
Speedup is the time on 8 servers versus sequential Java time.

The last two rows of Table 6.5 show the execution times for each matrix size when using all eight servers, and the corresponding speedups. The speedups obtained in these experiments are quite typical of those obtained in similar loosely coupled environments. While speedup is acceptable for smaller problem sizes, it continues to improve as the problem size grows, with a speedup of 7.6 obtained using 8 machines and a matrix size of 1024 x 1024 integers. That the speedup approaches linear as the problem size grows indicates that Babylon's overhead is largely a fixed cost versus computation and memory overhead costs.

Serial Matrix Multiplication with Migration

A key feature of Babylon is that we seamlessly integrate object migration and the execution of remote methods. Babylon was built to support objects which may be long-lived and as a result might be migrated many times during their lifetimes. While previous experiments show the cost of remote method invocation and object migration in isolation, it is also important to consider the overheads incurred by real applications that utilize these facilities. Therefore, we conducted experiments in which a sequential matrix multiplication is run and after completing a portion of its computation (in this case one eighth), all three matrices and the computation are migrated to a new machine[2]. This is repeated eight times. Matrix multiplication was chosen as a benchmark application because it is both data and computationally intensive. That is, it runs for a reasonable amount of time but also needs a significant amount of data to be migrated.

The results of these experiments, shown in Table 6.6, compare the execution times of the sequential matrix multiplication (done on a single Babylon server) with a sequential matrix multiplication forced to migrate to eight different machines. The first two rows of the table show the execution times for different problem sizes, while the last row shows the inflation factors for the different problem sizes (i.e., the ratio of the execution time including the migrations over the execution time without the migrations). The results show that even for matrices of moderate size the overheads associated with migrating the matrix objects a number of times does not significantly increase the total execution time. We found these results to be surprisingly good, especially because the computations involving a 1024 x 1024 matrix of integers must serialize, transfer and deserialize at least 12 MB of data during each migration (4 MB

[2]The source code for this application is provided in Appendix B.

for each of the three matrices, assuming four bytes for each integer). Admittedly, these objects are not very complex and may be faster to serialize and deserialize than more complex objects. However, even with an increase in object migration times the advantages of migration will still outweigh the disadvantages for some applications.

Matrix Size	(N)	200	500	640	800	1024
Time with 1 Server	(s)	7.7	134	292	580	1305
Time with 8 Servers	(s)	20.3	165	344	666	1402
Inflation Factor		2.64	1.23	1.18	1.15	1.07

Table 6.6: Serial Matrix Multiplication Times with and without Migration Overheads

We also believe that an inflation factor of 2.64 for the 200 x 200 matrix is not excessively large, considering that an additional job added to the same machine would result in an inflation factor of 2 (assuming round-robin scheduling with no overhead).

We expect that this test represents a worst case scenario: a computationally intense job using large objects. In the majority of cases, the number of migrations per unit time will be less than in our simulation. Objects will often be idle, allowing migration to occur between RMI calls, with negligible effect on execution times. We believe that the 7% inflation achieved for the longer running matrix multiplications is a positive result, and it justifies our belief that Babylon will perform well in environments requiring both efficient execution and mobility from objects.

File Display

As an example of the performance of Babylon's remote I/O facilities, a version of the common UNIX utility `cat` was implemented in Java. Our version is designed to echo text files, reading input and writing it to the console, line by line. While this is not the most efficient method of designing such a program, it is the most natural way of dealing with text files. Table 6.7 presents the results of this experiment. As the Babylon version runs remotely, the 6 ms overhead per line reflects the two remote method invocations that must be made per line. Read from file, and write to screen are both remote methods.

File Size	150 lines (6427 bytes)	
Local	total	85 ms
	per line	0.5 ms
Babylon	total	970 ms
	per line	6.5 ms

Table 6.7: File Display Results

While these results clearly indicate that remote I/O in Babylon can not compare in performance to local I/O, we believe they are reasonable. I/O is still very valuable for debugging purposes, and for human interactive use, where performance requirements are not as stringent. In these areas (debugging and human interaction), there is no comparable way of completing tasks remotely without I/O features such as those provided by Babylon. Thus, remote I/O is not a replacement for, nor in competition with, local I/O. It is a feature which provides an important function, rather than one which improves the speed of high performance computing.

6.2 Discussion

Having detailed the design and implementation of Babylon, and presented a range of performance results, we now discuss issues relating to the Babylon project as a whole.

The current design and implementation has been greatly influenced by our overriding goals to produce a system that is as transparent as possible while ensuring that we do not modify the language or use preprocessors. We want to ensure that Babylon, and applications that use Babylon, are compatible with existing Java tools, and the Java virtual machine. As a result, compromises have been made in other areas.

Babylon's Java compatibility is achieved by making the synchronous and asynchronous RMI interface less transparent than approaches used by SUN's RMI [46] or JavaParty [34]. The mechanisms provided by Babylon for creating and interacting with remote objects (`Babylon.{new(), rmi(), armi(), migrate()}`) are clearly different from similar mechanisms for local objects. This requires the application programmer to keep track of which objects are local and which are remote and to ensure that the proper interface is used for remote objects. However, this lack of transparency has its advantages, since it may be helpful for programmers to remember which method invocations are remote when performance is an issue.

An additional disadvantage of the Babylon RMI interface is that the method and parameters passed as parameters to the `Babylon.rmi()` call can not be checked at compile time, nor can the types of the parameters that are to be passed to the specified method. As a result it is not possible to detect errors that might otherwise be detected at compile time, such as invoking a non-existent method of an object, or invoking a method with incorrect arguments types, or an incorrect number of arguments. Unfortunately, in Babylon such problems can only be detected at run-time (Babylon throws an exception appropriate for the error). However, we believe that these tradeoffs are warranted in order to maintain 100% Java compatibility.

While we believe the overheads introduced by Babylon are reasonable, it is clear that the total costs are still high compared to existing high performance distributed computed systems such as Treadmarks [23]. As can be seen by examining the overheads incurred when invoking remote methods (Table 6.1) and performing object migration (Table 6.2), Babylon is clearly designed for use with coarse-grained applications. Even with improvements in Java virtual machine implementations, it is unlikely that Babylon will be used for fine-grained computations in a clustered workstation environment.

Although costs are relatively high at the present, we foresee the potential for significant improvements in Babylon's performance. The introduction of, and improvements to, just-in-time compiler technology has greatly improved Java performance in recent years, as shown by a comparison between ParaWeb and Ajents [20]. It is also possible that dynamic and/or hot-spot compilation techniques will further reduce the gap in execution times. Another large source of overhead in Babylon is its heavy reliance on RMI. While Java RMI is a very useful tool, it has significant costs. In addition to the method invocation costs, every RMI involves the serialization of parameters. Object serialization times are also an issue in migration. The performance of Babylon could be significantly improved through more efficient RMIs and/or improved object serialization [33, 30, 28, 24]. Recently published research by Phillippsen et al. [33] shows that object serialization takes between 25% and 50% of the time needed for a remote invocation and that this time can be improved by 81% to 97% [33]. Another paper by Phillippsen et al. [30] implements techniques to improve RMI times by a median of 45%. There is also the potential for improvements in Babylon's implementation. While some features, such as RMI, have been tuned for performance, other features, such as object migration are untuned. For example, it may be possible to reduce the number of messages and/or remote method invocations being used to implement migration.

Our personal experience with programming benchmarks and test applications have provided insight into the ease of programming for Babylon. While the use of a non-transparent interface requires minor extra coding and debugging, the interface is still relatively easy to use for experienced Java programmers. For example, a simple client-server socket-based Knock-Knock joke program from Sun's Java tutorial [6] was ported to Babylon in a very short time (roughly 10 minutes), with minimal changes required. No changes were required to the client side, while two types of changes were required for the server. One, a "bootstrap" program (about 5 lines long) was written to remotely create the remote Knock-Knock server object, and to remote invoke its "run" method. Two, the server object was slightly edited to change all socket and stream classes to remote socket and stream classes[3].

While Babylon supports a significant number of features, it is not free of limitations. Checkpointing and rollback are only seriously viable for methods which perform self-contained computation. Methods which alter external objects, through method invocations or I/O, are at risk from consistency problems, should they be interrupted, a checkpointed copy migrated, and rolled back.

Another limitation with respect to immediate migration involves remote methods which create additional threads. In order for immediate migration to proceed, Babylon halts threads related to remote method invocations. However, Babylon is only aware of the single thread that is executing a remote method invocation, and does not know about any additional threads that might be created during the execution of this method. Thus, during immediate migration, Babylon is not able to halt any

[3]For example, changing the declaration "`ServerSocket serverSocket;`" to "`RemoteServerSocket serverSocket;`".

additional threads that might have been created. A possible solution, currently unimplemented, would involve requiring remote objects to create threads using a Babylon specific interface. Babylon would then be able to retain a reference to these threads, and halt all threads related to a remote object, once an immediate migration request is made.

Chapter 7

Conclusions and Future Work

7.1 Conclusions

Babylon is a system which provides constructs designed to aid distributed object programming in Java. To provide these constructs, Babylon improves upon features included in class libraries initially created for Ajents, and implements a number of new features.

Babylon makes three main contributions to the area of Java-based distributed object programming.

- Unlike a number of other existing systems, Babylon is 100% Java compatible. No non-standard keywords, compilers, stub compilers or virtual machines are required. No changes to the Java language specification or the Java virtual machine are required.

- Babylon allows any Java object to be used in a distributed context and remote methods to be invoked upon that object. No special programming techniques are required including no added keywords, exceptions or variables.

- Babylon includes a significant number of features, which, to our knowledge, have not been combined into a single Java-based distributed environment. These features include extensions to features originally included in Ajents, as well as newly implemented features. Features added to the original version of Ajents [7] include:

  - Ajents requires class files to be located upon any host where the object might reside, either because it is created there, or migrated there. Babylon provides a remote class loading mechanism, so that class files need only be located upon the originating host, and the system ensures that these class files reach the appropriate destination.

  - Babylon adds support for exceptions to synchronous and asynchronous remote method invocations. This is not supported in Ajents.

  - Babylon adds support for the immediate migration of executing objects. Ajents only supports the migration of idle objects.

  Features newly implemented for Babylon are as follows.

  - Babylon implements a new internal object design, and a new remote method invocation design.

- Babylon implements the use of checkpointing, rollback and restart as a means of immediately migrating objects.
- Migration in Babylon occurs transparently. References to remote objects continue to work after migration, and remote methods which were interrupted by migration are transparently re-invoked.
- Babylon provides a means of communication with remote objects through console and file I/O, and socket communication. Remote I/O works correctly with remote objects before and after migration.
- Babylon provides basic infrastructure for a scheduling system, including the ability to seamlessly handle the arrival and departure of servers.

In order to access Babylon features such as migration and remote I/O, objects are required to implement or extend certain classes. These requirements are summarized in Table 7.1. Object creation, and remote method invocation (synchronous and asynchronous) can be performed using any Java object. This includes objects for which no source is available. Only remote objects that implement `serializable` can be migrated. Using Babylon's remote I/O facilities requires that the object extend `RemoteIO`. For remote creation, RMI, migration and remote I/O, Babylon provides a special application programming interface which must be used in order to access those features.

Babylon supports features that have not, to our knowledge, been previously im-

Babylon Feature	Object Restrictions
Object Creation and RMI	None
Object Migration	must implement `serializable`
Remote I/O	must extend `RemoteIO`

Table 7.1: Object Restrictions in Order to Utilize Babylon Features

plemented in a Java-based distributed system. In addition, Babylon's combined range of features is more extensive than any other Java-based distributed system that we are aware of. Finally, we have shown that the mechanisms used to implement these features perform in a reasonable fashion, and that future improvements in Java technology will positively impact Babylon. Thus, we believe that Babylon provides an excellent basis for building distributed object systems in Java.

7.2 Future Work

Babylon provides support for distributed object programming in Java. However, it is certainly not a finished product. There are additional research areas which would continue our work towards creating a system which autonomously allocates resources to objects while retaining ease of use and reasonable performance. The major areas available for future improvements and research related to Babylon are listed below.

Performance Improvements: Babylon has not been extensively tuned. Thus, we can not definitively state, for example, which parts of the migration process are slowest, and whether the most efficient programming techniques have been used in every case. An in-depth analysis, as well as adoption of new, more efficient Java routines for RMI [30, 28, 24] and object serialization [33] could result in greatly improved performance.

Scheduling: While Babylon provides a basic scheduling infrastructure, no scheduling policies are implemented or tested. Policies are required which can distribute the work equitably among the available processors, despite the limited information available. Not only is there the classic scheduling problem of unknown future job requirements, but future resource availability are also unknown. A second scheduling issue involves analyzing different approaches to hierarchical structuring, in order to learn how to effectively group large collections of servers and to handle object placement requests effectively and efficiently. Efficient scheduling mechanisms are vital in order for Babylon to scale over a large network.

Fault Tolerance: In any computing environment, there is the potential for hardware or software failure. Thus Babylon can not depend upon the absolute reliability of all Babylon servers. Fault tolerance is often a key requirement in distributed computing, and an investigation and application of fault tolerance schemes is required before Babylon can be considered complete. The checkpointing facilities provided in Babylon should provide a good base for implementing fault tolerance schemes.

Security: Babylon's reliance on Java's internal security mechanisms safeguards Babylon servers from malicious users. However, Babylon does not implement any security policy for remote objects. Currently, anyone may obtain a reference, and invoke methods upon remote objects. There is no authentication of the source of remote method invocation requests, neither are these requests encrypted while traveling over the network. As well, there is no protection for remote objects from malicious Babylon server hosts. The level of security required, and how to implement these security features is an important future step.

Remote Classloading Improvements: Babylon's remote classloading implementation meets basic requirements, but is not as efficient and flexible as possible. Aglets [25] implements a very similar classloading scheme, and many of their improvements could be adapted to Babylon. For example, Aglets caches an object's codebase to avoid repeated transfers, if the object migrates to the same host multiple times.

In this thesis, we have made significant contributions towards the development of distributed system features in Java and the integration of those features into a single Java-based distributed environment. We believe that Babylon forms a solid basis for future research into a complete system for distributed programming in Java.

Appendix A

Babylon User API

Listed here is the API for the methods which Babylon makes public for programmers to make use of. Babylon's internal API and structures are documented within the source code and not listed here for the sake of brevity.

These API's are also available at: http://www.cs.yorku.ca/~izatt/babylon/api

A.1 babylon.core.Babylon

public class Babylon extends java.lang.Object

A.1.1 Methods

```
public static java.lang.Object rmi(BabylonObj obj,
                                   java.lang.String method,
                                   java.lang.Object[] arguments)
                            throws MethodNotFoundException,
                                   java.lang.RuntimeException,
                                   RemoteExecException

public static java.lang.Object rmi(BabylonObj obj,
                                   java.lang.String method)
                            throws MethodNotFoundException,
```

```
                                   java.lang.RuntimeException,
                                   RemoteExecException

public static Future armi(BabylonObj obj,
                          java.lang.String method,
                          java.lang.Object[] arguments)

public static Future armi(BabylonObj obj,
                          java.lang.String method)

public static boolean migrate(BabylonObj obj,
                              java.lang.String host)
                       throws NotMovableException

public static boolean migrate(BabylonObj object,
                              RemoteObjectServer toServer)
                       throws NotMovableException

public static BabylonObj remoteNew(java.lang.String className,
                                   java.lang.String instanceName,
                                   java.lang.String jarFile,
                                   RemoteObjectServer ros,
                                   RemoteIO io)
                            throws RemoteInstantiationException,
                                   java.io.IOException,
                                   java.io.FileNotFoundException

public static Scheduler register()

public static Scheduler register(java.lang.String schedulerAddress)

public static void setCheckPoint(BabylonObj obj,
                                 boolean check)
                          throws java.rmi.RemoteException
```

A.2 babylon.core.Future

public class Future extends java.lang.Object

A.2.1 Methods

```
public java.lang.Object get()
                     throws MethodNotFoundException,
                            RemoteExecException,
                            java.lang.RuntimeException
```

A.3 babylon.core.RemoteObj

public class RemoteObj extends java.lang.Object implements java.io.Serializable

A.3.1 Methods

```
public final RemoteIO getIO()
```

```
public final java.lang.String getLocation()
```

A.4 babylon.sched.SchedulerImpl

public class SchedulerImpl extends java.rmi.server.UnicastRemoteObject implements Scheduler

A.4.1 Methods

```
public RemoteObjectServer AvailServer()
                               throws java.rmi.RemoteException
```

```
public boolean registerServer(RemoteObjectServer ros,
                              java.net.InetAddress addr)
                       throws java.rmi.RemoteException
```

```
public boolean unregisterServer(RemoteObjectServer ros,
                                java.net.InetAddress addr)
                         throws java.rmi.RemoteException
```

```
public RemoteObjectServer getServer(java.lang.String serverName)
                             throws java.rmi.RemoteException
```

```
public static void main(java.lang.String[] args)
```

A.5 babylon.io.RemoteIOImpl

public class RemoteIOImpl extends java.rmi.server.UnicastRemoteObject implements RemoteIO, java.io.Serializable

A.5.1 Methods

```
public void println(java.lang.Object obj)
             throws java.rmi.RemoteException

public void print(java.lang.Object obj)
           throws java.rmi.RemoteException

public java.lang.String readLine()
                          throws java.rmi.RemoteException,
                                 java.io.IOException

public RemotePrintWriter
newRemotePrintWriter (java.lang.String fileName)
               throws java.rmi.RemoteException,
                      java.io.IOException

public RemoteObjOutStream
newRemoteObjOutStream (java.lang.String fileName)
                throws java.rmi.RemoteException,
                       java.io.IOException

public RemoteObjInStream
newRemoteObjInStream (java.lang.String fileName)
               throws java.rmi.RemoteException,
                      java.io.IOException

public RemoteBufferedReader
newRemoteBufferedReader (java.lang.String fileName)
                  throws java.rmi.RemoteException,
                         java.io.IOException

public RemoteServerSocket
newRemoteServerSocket (int port)
                throws java.rmi.RemoteException,
```

```
                 java.io.IOException

public RemoteSocket newRemoteSocket(java.lang.String host,
                                    int port)
                             throws java.io.IOException,
                                    java.rmi.RemoteException

public RemoteSocket newRemoteSocket(java.net.InetAddress address,
                                    int port)
                             throws java.io.IOException,
                                    java.rmi.RemoteException
```

Appendix B

Sample Babylon Application - Serialized Matrix Multiply

Listed here is the source code for the serialized matrix multiply application used in Section 6.1.3.

B.1 Matrix.java

```
package babylon.tests.MatMul;

import java.io.Serializable;
import babylon.core.*;
import java.util.Random;

public class Matrix implements Serializable
{
  private final static int numberSize = 10;
  private int rows = 10;
  private int columns = 10;
  int[][] m;

  public Matrix() {
    m = new int[rows][columns];
  }
```

```
public Matrix(int rows, int columns)  {
  this.rows = rows;
  this.columns = columns;
  m = new int[rows][columns];
}

public int getValue(int x, int y)  {
  return m[x][y];
}


public void print() {
  for (int i = 0; i < rows; i++){
      for (int j = 0; j < columns; j++)
              System.out.print(m[i][j] + " ");
      System.out.println();
      }
}

public int getRows()  {
  return rows;
}

public int getColumns() {
  return columns;
}

public void setValue(int x, int y, int value) {
  m[x][y] = value;
}

public static Matrix createIdentityMatrix(int size) {
  Matrix result = new Matrix(size, size);
  for (int i = 0; i < size; i++)
      for (int j = 0; j < size; j++)
        if (i == j)
          result.setValue(i, j, 1);
      else
          result.setValue(i, j, 0);
```

```
    return result;
  }

  public static Matrix createRandomMatrix(int rows, int columns) {
    Matrix result = new Matrix(rows, columns);
    Random random = new Random();
    for (int i = 0; i < rows; i++)
        for (int j = 0; j < columns; j++)
          result.setValue(i, j, random.nextInt() % Matrix.numberSize);
    return result;
  }

  public Matrix getSubMatrixByRows(int startAtRow, int endAtRow) {
    int nrows = endAtRow - startAtRow + 1;
    Matrix result = new Matrix(nrows, columns);
    for (int i = 0; i < nrows; i++)
        for (int j = 0; j < columns; j++)
          result.setValue(i, j, m[i+startAtRow][j]);
    return result;
  }

  public Matrix getSubMatrixByColumns(int startAtCol, int endAtCol) {
    int ncols = endAtCol - startAtCol + 1;
    Matrix result = new Matrix(rows, ncols);
    for (int i = 0; i < rows; i++)
        for (int j = 0; j < ncols; j++)
          result.setValue(i, j, m[i][j+startAtCol]);
    return result;
  }
}
```

B.2 MultiplyMatrix.java

```
package babylon.tests.MatMul;

import java.rmi.server.*;
import java.rmi.*;
import java.io.Serializable;
import babylon.core.*;
```

```
public class MultiplyMatrix extends RemoteObj implements Serializable
{
  Matrix a,b,c;

  public MultiplyMatrix() {
  }

  public void setA(Matrix m) {
    a = m;
  }

  public void setB(Matrix m) {
    b = m;
  }

  public void createC() {
    c = new Matrix(a.getRows(),b.getColumns());
  }

  public Matrix getResult() {
    return c;
  }

  public void  multiply(Integer st,Integer fi) {
    int brows, rows, columns;
    int start = st.intValue();
    int finish = fi.intValue();
    int value;
    if (a.getColumns()== b.getRows()) {
        rows = a.getRows();
        columns = b.getColumns();
        brows = b.getRows();

        for (int i = start; i < finish; i++)
          for (int j = 0; j < columns; j++) {
            value = 0;
            for (int k = 0; k < brows; k++)
                value += a.getValue(i,k) * b.getValue(k,j);
            c.setValue(i, j, value);
          }
```

```
    }
  }


}
```

B.3 TestMatrixMult.java

```
package babylon.tests.MatMul;

import babylon.core.*;
import babylon.io.*;
import babylon.server.*;
import babylon.sched.*;
import java.rmi.*;
import java.util.Date;

public class TestMatrixMult
{
  public static void main(String args[]) {
    System.setSecurityManager(new RMISecurityManager());
    if (args.length != 3) {
      System.out.println("<scheduling server> " +
                         "<num processors> <matrix size>");
      System.exit(0);
    }

    Scheduler sched = Babylon.register(args[0]);

    int nprocs = Integer.parseInt(args[1]);
    int size = Integer.parseInt(args[2]);

    Matrix a, b,c;
    BabylonObj mm = null;


    long startServerT, finishServerT;
    long startExecT, finishExecT;

    startServerT = (new Date()).getTime();
```

```
try {
    RemoteIO io = new RemoteIOImpl();
    mm = Babylon.remoteNew(
            "babylon.tests.MatMul.MultiplyMatrix",
            "mm", "/cs/home/grad1/izatt/MatMul.jar",
            sched.AvailServer(),io);
}
catch (Exception ex) {
    System.out.println("Creation error");
}

b = Matrix.createIdentityMatrix(size);
a = Matrix.createRandomMatrix(size, size);
c = null;
int start_row = 0;
int end_row = -1;
int nrows = size / nprocs;

try {
  Babylon.rmi(mm, "setA", a);
  Babylon.rmi(mm, "setB", b);
  Babylon.rmi(mm, "createC");
}
catch (Exception ex) {
  System.out.println("error " + ex);
}

finishServerT = (new Date()).getTime();

for (int i = 0; i < nprocs; i++) {
  if (i >= size % nprocs) {
    start_row = end_row + 1;
    end_row = start_row + nrows - 1;
  }
  else {
    start_row = end_row + 1;
    end_row = start_row + nrows;
  }
}
```

```
    startExecT = (new Date()).getTime();
    for (int i = 0; i < nprocs; i++) {
      try {
        Babylon.rmi(mm, "multiply", new Integer(i*nrows),
                    new Integer((i+1)*nrows));
        if (i < nprocs-1) {
          System.out.println("moving..");
          Babylon.migrate(mm,sched.AvailServer());
        }
      }
      catch (Exception ex) {
        System.out.println("Execution " + i + ": error: " + ex);
      }
    }
    try {
      c = (Matrix) Babylon.rmi(mm,"getResult");
    }
    catch (Exception ex) {
      System.out.println("error " + ex);
    }
    finishExecT = (new Date()).getTime();

    long serverStartup = finishServerT - startServerT;
    long execTime = finishExecT - startExecT;
    System.out.println("Matrix size = " + size +
                       ", processors = " + nprocs);
    System.out.println("Time required for starting " +
                       nprocs + " servers:" + serverStartup);
    System.out.println("Time required for multiplication: " +
                       execTime);
    System.exit(0);
  }
}
```

Bibliography

[1] A. Alexandrov, M. Ibel, K. Schauser, and C. Scheiman, "SuperWeb: Towards a Global Web-Based Parallel Computing Infrastructure," in *11th International Parallel Processing Symposium*, April 1997.

[2] Y. Amir, B. Awerbucn, A. Barak, R. Borgstrom, and A. Keren, "An opportunity cost approach for job assignment in a scalabale computing cluster," in *Proceedings of the 10th International Conference on Parallel and Distributed Computing Systems (PDCS '98)*, pp. 40–53, October 1999.

[3] E. Arjomandi, W. O'Farrell, I. Kalas, G. Koblents, F. Eigler, and G. Gao, "ABC++: Concurrency and Inheritance in C++," *IBM Systems Journal*, Vol. 34, No. 1, pp. 120–136, 1995.

[4] A. Baratloo, M. Karaul, Z. Kedem, and P. Wyckoff, "Charlotte: Metacomputing on the Web," in *Proceedings of the 9th Conference on Parallel and Distributed Computing Systems*, 1996.

[5] T. Brecht, H. Sandhu, J. Talbot, and M. Shan, "ParaWeb: Towards World-Wide Supercomputing," in *European Symposium on Operating System Principles*, October 1996.

[6] M. Campione and K. Walrath, *The Java Tutorial.* Addison Wesley Developers Press, Sunsoft Java Series, 1998.

[7] P. Chan, "Ajents: A Parallel and Distributed Java System." Master's project, York University, 1998.

[8] B. Christiansen, P. Cappello, M. Ionescu, M. Neary, K. Schauser, and D. Wu, "Javelin: Internet-Based Parallel Computing Using Java," in *ACM 1997 Workshop on Java for Science and Engineering Computation*, June 1997.

[9] A. Dick, "Object-Oriented Distributed and Parallel I/O Streams," Master's thesis, York University, 1999.

[10] A. Dick, E. Arjomandi, and T. Brecht, "Object-Oriented Distributed and Parallel I/O Streams," in *13th Annual International Symposium on High Performance Computing Systems and Applications (HPCS '99)*, June 1999.

[11] A. B. Downey and M. Harchol-Balter, "A note on "The limited performance benefits of migratings active processes for load sharing"," Technical Report CSD-95-888, University of California, Berkeley, November 1995.

[12] D. L. Eager, E. D. Lazowska, and J. Zahorjan, "The limited performance benefits of migrating active processes for load sharing," *SIGMETRICS*, pp. 662–675, May 1988.

[13] S. Fünfrocken, "Transparent Migration of Java-based Mobile Agents (Capturing and Re-establishing the State of Java Programs)," in *Proceedings of the Second International Workshop on Mobile Agents (MA'98)*, September 1998.

[14] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Mancheck, and V. Sunderam, *PVM: Parallel Virtual Machine, A User's Guide and Tutorial for Networked Parallel Computing.* MIT Press, Cambridge Massachusetts, 1994.

[15] G. Glass, "Overview of Voyager: ObjectSpace's Product Family for State-of-the-Art Distributed Computing," Technical Report, ObjectSpace, 1999.

[16] J. Gosling, B. Joy, and G. Steele, *The Java Language Specification.* Addison Wesley Developers Press, Sunsoft Java Series, 1996.

[17] J. Gosling and H. McGilton, "The Java Language Environment," Technical Report, Sun Microsystems, http://java.sun.com, October 1995.

[18] M. Harchol-Balter and A. Downey, "Exploiting process lifetime distributions for dynamic load balancing," in *Proceedings of ACM SIGMETRICS '96*, May 1996.

[19] O. Holder, I. Ben-Shaul, and H. Gazit, "System support for dynamic layout of distributed applications," in *Proceedings of the 19th International Conference on Distributed Computing Systems (ICDCS '99)*, pp. 403–411, May 1999.

[20] M. Izatt, T. Brecht, and P. Chan, "Ajents: Towards an Environment for Parallel, Distributed and Mobile Java Applications," in *ACM 1999 Java Grande Conference*, June 1999. (Notwithstanding the title, this paper is based upon Babylon).

[21] Java Grande Forum, 1998-. http://www.javagrande.org.

[22] H. Karl, "Bridging the gap between Distributed Shared Memory and Message Passing," in *ACM 1998 Workshop on Java for Science and Engineering Computation*, February 1998.

[23] P. Keleher, A. Cox, S. Dwarkadas, and W. Zwaenepoel, "TreadMarks: Distributed Shared Memory on Standard Workstations and Operating Systems," in *Proceedings of the 1994 USENIX Technical Conference*, 1994.

[24] V. Krishnaswamy, D. Walther, S. Bhola, E. Bommaiah, G. Riley, B. Topol, and M. Ahamad, "Efficient Implementations of Java RMI," in *Proceedings of the 4th USENIX Conference on Object-Oriented Technologies and Systems (COOTS'98)*, April 1998.

[25] D. Lange and M. Oshima, *Programming and Deploying Java Mobile Agents with Aglets.* Addison Wesley, 1998.

[26] T. Lindholm and F. Yellin, *The Java Virtual Machine Specification.* Addison Wesley Developers Press, Sunsoft Java Series, 1996.

[27] M. Litzkow, "Remote Unix - Turning Idle Workstations into Cycle Servers," in *Proceedings of the Usenix Summer Conference*, pp. 381–384, June 1987.

[28] J. Maassen, R. van Nieuwpoort, R. Veldema, H. E. Bal, and A. Plaat, "An Efficient Implementation of Java's Remote Method Invocation," in *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP'99)*, May 1999.

[29] M. Neary, S. Brydon, P. Kmiec, S. Rollins, and P. Cappello, "Javelin++: Scalability Issues in Global Computing," in *ACM 1999 Java Grande Conference*, June 1999.

[30] C. Nester, M. Phillippsen, and B. Haumacher, "A More Efficient RMI for Java," in *ACM 1999 Java Grande Conference*, June 1999.

[31] R. Netzer and J. Xu, "Necessary and Sufficient Conditions for Consistent Global Snapshots," *IEEE Transactions on Parallel and Distributed Systems*, Vol. 6, No. 2, February 1995.

[32] ObjectSpace, *Voyager Core Technical 2.0 User Guide*, 1998.

[33] M. Phillippsen and B. Haumacher, "More Efficient Object Serialization," in *International Workshop on Java for Parallel and Distributed Computing*, April 1999.

[34] M. Phillippsen and M. Zenger, "JavaParty - Transparent Remote Objects in Java," in *ACM 1997 Workshop on Java for Science and Engineering Computation*, June 1997.

[35] R. Raje, J. I. William, and M. Boyles, "An Asynchronous Remote Method Invocation (ARMI) Mechanism for Java," in *ACM 1997 Workshop on Java for Science and Engineering Computation*, June 1997.

[36] M. Ranganathan, A. Acharya, S. Sharma, and J. Saltz, "Network-aware Mobile Programs," in *Proceedings of the 1997 USENIX Technical Conference*, 1997.

[37] R. Riggs, J. Waldo, and A. Wollrath, "Pickling State in Java," in *2nd Conference on Object-Oriented Technologies and Systems (COOTS)*, Toronto, Ontario, pp. 241–250, June 1996.

[38] P. Smith and N. Hutchinson, "Heterogeneous Process Migration: The Tui System," Technical Report, University of British Columbia, March 1997.

[39] M. Snir, S. Otto, S. Huss-Lederman, D. Walker, and J. Dongarra, *MPI: The Complete Reference.* MIT Press, Cambridge Massachusetts, 1996.

[40] M. Straßer, J. Baumann, and F. Hohl, "Mole - A Java based Mobile Agent System," in *ECOOP '96 Workshop on Mobile Object Systems*, 1996.

[41] Sun Microsystems, Palo Alto, CA., *Java Object Serialization Specification*, 1997.

[42] Sun Microsystems, Palo Alto, CA., *Java Remote Method Invocation Specification JDK 1.1*, 1997.

[43] Sun Microsystems, Palo Alto, CA., *Java Platform 1.1 Core API Specification*, 1998.

[44] Sun Microsystems, Palo Alto, CA., *Java Remote Method Invocation Specification JDK 1.2*, 1998.

[45] Y.-M. Wang, "Consistent Global Checkpoints that Contain a Given Set of Local Checkpoints," *IEEE Transactions on Computers*, Vol. 46, No. 4, April 1997.

[46] A. Wollrath, R. Riggs, and J. Waldo, "A Distributed Object Model for Java," in *2nd Conference on Object-Oriented Technologies and Systems (COOTS)*, Toronto, Ontario, pp. 219–231, June 1996.

[47] W. Yu and A. Cox, "Java/DSM: a platform for heterogeneous computing," in *ACM 1997 Workshop on Java for Science and Engineering Computation*, June 1997.

[48] S. Zhou, J. Wang, X. Zheng, and P. Delisle, "Utopia: a load-sharing facility for large heterogeneous distributed computing systems.," *Software - Practice and Experience*, Vol. 23, No. 2, pp. 1305–1336, December 1993.