Babylon v2.0:
Support for Distributed, Parallel and Mobile Java Applications

by

Willem van Heiningen

A thesis
presented to the University of Waterloo
in fulfilment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2003

I hereby declare that I am the sole author of this thesis.

I authorize the University of Waterloo to lend this thesis to other institutions or individuals for the purpose of scholarly research.

Willem van Heiningen

I further authorize the University of Waterloo to reproduce this thesis by photocopying or by other means, in total or in part, at the request of other institutions or individuals for the purpose of scholarly research.

Willem van Heiningen

The University of Waterloo requires the signatures of all persons using or photocopying this thesis. Please sign below, and give address and date.

Abstract

This thesis describes the design and implementation of Babylon v2.0. Babylon v2.0 is a 100% Java compatible framework for building parallel, distributed and mobile applications in Java. Babylon v2.0 incorporates features like object migration, asynchronous method invocation and remote class loading while providing an easy-to-use interface that enables seamless interaction with remote objects and hides the complexities of remote messaging protocols that are normally large part of distributed systems programming. The potential cluster computing benefits of Babylon v2.0 are demonstrated by the evaluation results which show that sequential Java applications can achieve significant performance gains by using Babylon v2.0 to parallelize their work across a cluster of workstations. Intuitive interfaces, ease of use, support for multiple simultaneous users, and services and features that facilitate the development and administration of distributed systems make Babylon v2.0 a unique and powerful system for distributed systems programmers.

Acknowledgements

I would like to acknowledge and thank some of the people I've had the privilege of meeting, working with, and hanging out with during my time at the University of Waterloo.

These people include Tim Brecht for his enthusiasm, patience and all-around outstanding supervision during the somewhat longer than normal duration of my degree; Mom and Dad for their endless encouragement, loving support and the many generous donations to the "get-Will-through-university" fund; the McLaughlin's for acting as my surrogate family and providing me with lots of love, home-cooked meals, bottles of scotch, pain-killers, and many other basic necessities of grad student life; Gabe and Ken for helping me drink those bottles of scotch and for being some of the best friends a guy could ask for; Zheng for her wise, sweet and sometimes painfully honest advice and discussions, and for sticking around until I finished (that one goes for Gabe as well); Yvonne for her great friendship and the many wonderful discussions we had; Dave E., Chrispy (and Tim again) for keeping the hammer cocked on the slopes of Chicopee; my office mate, Sonia, for putting up with the mess on my desk as well as the regular phone calls from parents and friends; Matt, Mike, Dana, Kin-Fai, Tom, Sehoon, Tom, Anwar, Dennis, Jin, Chato and all the other Masters and PhD students that helped make my experience at Waterloo unforgettable; the computer science staff for helping out with all those things that a confused and bewildered Masters student needs help with; the readers, Ken and Steve, for their helpful comments and, most importantly, their signatures; and last, but certainly not least, Janet for her endless encouragement and understanding, and for being an inspiration to me.

Contents

1 Introduction **1**
1.1 Motivation . . . 1
1.2 Objectives . . . 2
1.3 Contributions . . . 4
1.4 Outline of Thesis . . . 5

2 Background and Related Work **7**
2.1 Babylon v1.0 . . . 7
2.1.1 System Overview . . . 7
2.1.2 Remote Object Creation . . . 8
2.1.3 Synchronous Remote Method Invocation . . . 9
2.1.4 Asynchronous Remote Method Invocation . . . 11
2.1.5 Object Migration . . . 12
2.1.6 Remote Class Loading . . . 14
2.1.7 Remote I/O . . . 15
2.2 Related Work . . . 15
2.2.1 JavaParty . . . 15
2.2.2 Javelin . . . 16
2.2.3 Globus and the Java Commodity Grid Kit . . . 16
2.2.4 ProActive . . . 17
2.2.5 Ninflet . . . 18
2.2.6 Charlotte . . . 18
2.2.7 Voyager . . . 19
2.2.8 Java/DSM . . . 19
2.2.9 Java-Based Mobile Agent Systems . . . 20
2.3 Discussion . . . 20

3 Distributed Object Programming with Babylon v2.0 **22**
3.1 Introduction . . . 22
3.2 The Worker JAR . . . 23
3.3 The Application Policy File . . . 23
3.4 Initialization . . . 24
3.5 Dynamic Proxies . . . 24
3.5.1 Remote Worker Object Creation . . . 25
3.5.2 Asynchronous Remote Method Invocation . . . 27

3.6 Worker Object Lookup 30
3.7 Summary 30

4 Babylon Architecture **31**
4.1 Communication Model 31
4.2 Worker Object Migration 32
4.2.1 Migration Issues 33
4.2.2 Safe Mode 35
4.2.3 Safe Migration Points 35
4.2.4 Reference Updating 37
4.3 Fault-Tolerance in Babylon v2.0 37
4.4 Class Loading in Babylon 38
4.4.1 JVM Class Loading Semantics 38
4.4.2 Class Loading in Babylon v1.0 39
4.4.3 Class Loading in Babylon v2.0 41
4.5 Summary 42

5 System Evaluation **43**
5.1 Reducing Performance Variance during Benchmarking 43
5.2 Testbed 44
5.3 Micro Benchmarks 44
5.3.1 Remote Method Invocation 44
5.3.2 Worker Object Migration 45
5.4 Application Benchmarks 46
5.4.1 Matrix Multiplication 47
5.4.2 Heat Diffusion 51
5.4.3 Serial Matrix Multiplication with Migration 55
5.5 Discussion 56

6 Conclusions and Future Work **58**
6.1 Conclusions 58
6.2 Future Work 60

A Default Policy File **61**

B Code Samples **62**
B-1 Dictionary Interface and Implementation 62
B-2 Asynchronous Remote Method Invocation 63

C Sample Babylon v2.0 Application - Heat Diffusion **64**
C-1 GridClient.java 64
C-2 Grid.java 66
C-3 GridDiffuser.java 67
C-4 GridDiffuserImpl.java 68

D Sample Babylon v2.0 Application - Matrix Multiplication **73**

D-1 TestMatrixMult.java 73
D-2 Matrix.java 74
D-3 MultiplyMatrix.java 76
D-4 MultiplyMatrixImpl.java 76

E Garbage Collection Tuning Parameters **77**

List of Figures

1.1 Babylon Remote Method Invocation Comparison 3

2.1 Principal Components of the Babylon v1.0 Framework 8
2.2 Remote Object Creation in Babylon v1.0 9
2.3 Remote Method Invocation on a Worker Object in Babylon v1.0 10
2.4 Asynchronous Remote Method Invocation on a Worker Object in Babylon v1.0 12

3.1 Principal Components of the Babylon v2.0 Framework 23
3.2 Worker Object Creation and Method Invocation in Babylon v2.0 26
3.3 Asynchronous Remote Method Invocation on a Worker Object in Babylon v1.0 28
3.4 Asynchronous method invocation in Babylon v2.0 29

4.1 Babylon v2.0 Worker Object Communication Model 32
4.2 Immediate Migration in the Presence of Concurrent Method Invocations 34
4.3 Adding Safe Migration Points to Worker Object Code 36
4.4 Failed Class Loading Attempt 39

5.1 Babylon v2.0 Application Benchmark Speedups 57

A.1 Default Policy File 61

B.1 Interface Dictionary 62
B.2 Class DictionaryImpl 62
B.3 Asynchronous method invocation in Babylon v2.0 63

List of Tables

5.1 Remote Method Invocation Performance (ms) 45
5.2 Worker Object Migration Performance (ms) 46
5.3 Babylon v1.0 and v2.0 Matrix Multiplication Speedup and Execution Time (ms) 48
5.4 Babylon v2.0 Matrix Multiplication Speedup and Execution Time (ms) 48
5.5 Babylon v2.0 Matrix Multiplication Result Analysis (2048x2048) 50
5.6 Babylon v2.0 Heat Diffusion Speedup and Execution Time using 100 Iterations (ms) 53
5.7 Babylon v2.0 Heat Diffusion Result Analysis (4096x4096) 53
5.8 Babylon v2.0 Time for Matrix Multiplication including Migrations (ms) 55

Chapter 1

Introduction

The goal of this thesis is to build an easy-to-use, flexible, multi-user distributed Java object environment to support cluster computing and provide the basis for future research projects in the areas of scheduling, fault-tolerance and performance in distributed computing environments.

This goal has been realised through the design and implementation of the Babylon v2.0 distributed object system. Babylon v2.0 is a collection of tools and services that builds on experiences with Babylon v1.0 to provide a 100% Java compatible environment for developing, running and managing parallel, distributed and mobile Java applications. The potential cluster computing benefits of Babylon v2.0 are demonstrated by the evaluation results which show that sequential Java applications can achieve significant performance gains by using Babylon v2.0 to parallelize their work across a cluster of workstations. Intuitive interfaces, ease of use, support for multiple simultaneous users, and services and features that facilitate the development and administration of distributed systems make Babylon v2.0 a unique and powerful system for distributed systems programmers.

This chapter first provides the motivation for the creation of a environment for distributed Java objects. We then outline the objectives and contributions of the thesis and conclude with an outline of the remaining chapters.

1.1 Motivation

Traditionally, the Internet has been used primarily for the dissemination and sharing of information among dispersed groups of people. Tools such as web browsers, electronic mail, instant messaging, and file sharing applications are commonly used to access and share this information. More recently, an emerging vision of the Web is that of an unparalleled source of interconnected computational resources accessible to users anywhere on the Internet. This vision – of a massive supercomputer spanning the globe – is the impetus for the creation of Babylon.

Many existing applications have the potential for enormous performance gains if they could somehow harness even a fraction of the computing resources available on office LANs, organizational intranets, and the Internet. However, leveraging these untapped resources is not necessarily a simple or straightforward task. Most existing applications are designed for a single-processor and are often written for a specific platform. Furthermore, building a distributed or parallel application from the ground up using existing tools and services often requires learning complicated new APIs or constraining oneself to a particular coding or computational model.

The Java language [22] has many features that facilitate distributed systems programming. Java's built-in security, threading and dynamic class loading support can greatly simplify the development of distributed applications. Furthermore, Java applications are compiled into a machine independent representation called bytecodes that can be run on any machine that runs the Java Virtual Machine (JVM). This platform independence makes Java an attractive choice for transcending the complexities of multi-platform application development.

Java also supports remote messaging protocols such as Remote Method Invocation (RMI) and the Common Object Request Broker Architecture (CORBA), which hide the complexities of communication with objects residing in other JVMs, possibly on other machines. Nevertheless, building a distributed application using either of these protocols still requires a significant amount of code to implement the distributed layer. The task of distributing and managing remote objects across several machines introduces further complications. Additionally, the process of converting a single-processor application into a parallel or distributed application using either of these protocols would be both non-trivial and time consuming.

The original implementation of Babylon [26], referred to here as Babylon v1.0, was an important initiative to make distributed computing resources more widely available to developers. Babylon v1.0 is a collection of support classes and server components for building and running parallel, distributed and mobile applications. Much of the design for Babylon v1.0 came from experiences with Ajents [27] and ParaWeb [11]. Babylon v1.0 built on these systems to provide mechanisms for remote object creation, migration and remote I/O support while maintaining 100% Java compatibility. By building on top of Java's RMI, Babylon v1.0 takes advantage of Java's built-in features to provide a environment for building distributed applications that can utilize some of the computing power sitting idle on LANs, organizational intranets, and the Internet.

Unfortunately, Babylon v1.0 lacks several important distributed systems attributes including separate class namespaces for clients, remote object administration facilities, support for nested remote object creation, remote object sharing and state information to track client sessions. Additionally, object migration in Babylon v1.0 was not optimized and relied on Java features that have been deprecated since v1.0's implementation. However, the most serious drawback of Babylon v1.0 is its reliance on a non-standard remote method invocation interface that is awkward to use and error-prone because it prevents normal compile-time method invocation checks.

1.2 Objectives

The objective of this thesis is to design and build a environment for creating and running distributed Java applications. The environment must be capable of supporting multiple, simultaneously executing applications written by different users in a clustered workstation environment. Other required features include the ability to seamlessly interact with remote objects, migrate remote objects between machines and share remote objects with other users. The system will be used to investigate parallel and distributed application performance on workstation clusters and will serve as the basis for future research projects which will explore fault-tolerance, scheduling and performance issues in distributed computing environments.

Babylon v1.0 provides a good basis for building such a environment. However, Babylon v1.0 is deficient in several key areas. Most importantly, Babylon v1.0's remote method invocation mechanisms must be made more user-friendly and transparent. To illustrate some of the interface problems in Babylon v1.0, consider the code segment in Figure 1.1, in which the method `ask(String question)`, which returns a `String` and throws an `OracleException`, is invoked on a remote object.

In Babylon v1.0, a method on a remote object must be invoked using the provided `Babylon.rmi()`

method. This approach is significantly different from the standard syntax for method invocations in Java and bears more of a resemblance to method invocation using Java reflection. It does not allow the use of primitive type method arguments and prevents the validation of the method name as well as the number and the types of the arguments by the compiler. Thus, programmers using Babylon v1.0 must keep track of local and remote objects in their code so that they can use the appropriate method invocation mechanism. Furthermore, unlike standard Java method invocation, Babylon v1.0 uses the run-time type of the arguments to determine which method to use when invoking an overloaded method on a remote object.

The static `Babylon.armi()` interface method, which is used to make asynchronous remote method invocations in Babylon v1.0, suffers from the same drawbacks. As with `Babylon.rmi()`, incorrect use of method names and/or arguments will not be detected until run-time, making it difficult and time consuming to debug distributed applications.

```
String theQuestion = "What is the average wind-speed velocity of a sparrow?";
String theAnswer;

/* Invoking method ask() on a worker in Babylon v1.0. */
try {
   theAnswer = (String) babylon.core.Babylon.rmi (
      workerId,      // Worker identifier.
      "ask",         // String representation of method.
      theQuestion    // Argument to method.
   );
}
catch (babylon.core.RemoteExecException remoteEx) {
   if (remoteEx.getTargetException() instanceof OracleException) {
      // Exception handling code can go here.
   }
}
```

Figure 1.1: Babylon Remote Method Invocation Comparison

Furthermore, the design of the class loading architecture on Babylon v1.0 servers does not correctly support more than a single client and normally requires a server restart between successive client connections. This is because each Babylon v1.0 server uses a single classloader instance for all clients and, as a result, servers load all classes from all clients using the same class namespace. If a client tries to create a remote object of a class with the same name as a class previously loaded by a different client, an instance of the previously loaded class will be created by the server. Not only can this produce incorrect results and difficult to trace bugs, it may also compromise security because clients can create instances of classes loaded by other clients. Furthermore, if a programmer changes a class after it has been loaded by a particular server, that server must be restarted to clear its class namespace so that the new version of the class can be used. Clearly, this is not practical in an environment where someone else may be supplying the CPU and running the server. Babylon v1.0's class loading design also prevents client classes from being garbage collected because the class loader that defined them is never be unloaded.

The Java language has evolved significantly since the development of Babylon v1.0, introducing new features that could be exploited to improve the system while deprecating others that the original implementation relied on. For instance, remote object migration in Babylon v1.0 uses the deprecated `Thread.stop()` method to stop executing threads when a remote object is being migrated. Deprecated methods should be avoided because they may not be supported in future Java versions. The design of object migration in Babylon v1.0 is also flawed in other ways and can, in some cases, lead to remote objects which continue executing at their original location even after they have been migrated, or result in migrated remote objects with an inconsistent state. These are just a few examples of the major issues that prevent us from using Babylon v1.0 in its current form.

Unlike many other existing distributed systems, the new system must be able to support a wide range of applications and should not be limited to a specific computational model. For instance, many existing systems only support applications that adhere to the master-worker (Charlotte [7], Ninflet [51]) or branch-and-bound (Javelin [36]) computational model. In the master-worker paradigm [54], a master coordinates and distributes tasks to a number of workers. Each task is completely independent and so there is no communication or synchronization required between workers. As soon as a worker completes its assigned task, it returns the result to the master and becomes ready to accept a new task.

The branch-and-bound paradigm [38] can be used to solve optimization problems without necessarily searching the entire solution space. In this model, one maintains a record of the cost of the currently best solution found thus far. This cost is used to eliminate other partial solutions whose intermediate costs are already worse than the cost of the best discovered solution. However, there are many problems that cannot be structured as a master-worker or branch-and-bound application.

The objective of this thesis is to build a general purpose clustered computing environment that is not limited to a particular computational model. Babylon v1.0 can be used as the basis but it must be significantly modified and extended to fix its flaws and provide more powerful and flexible mechanisms and features for distributed object computing on workstation clusters. Furthermore, our goal is to ensure that the new system is written entirely in Java without the use of language extensions or special preprocessors so that it will be possible to use it on any platform that supports the JVM.

1.3 Contributions

This thesis describes the design and implementation of Babylon v2.0. Babylon v2.0 builds on the strengths of Babylon v1.0 and fixes many of its shortcomings. Babylon v2.0 also contributes several new features and approaches in the area of Java-based distributed and parallel systems without the use of special Java language extensions, preprocessors or compilers.

- The awkward and error-prone remote method invocation interface of Babylon v1.0 has been replaced with a transparent remote object invocation mechanism that uses proxy objects. As a result, invoking a method on a remote object in Babylon v2.0 is syntactically identical to a local method invocation. The new approach also permits the use of primitive-type remote method parameters.

- Babylon v2.0 uses a novel asynchronous method invocation technique based on asynchronous tickets. An asynchronous ticket is a “one-time” proxy object that can be used to make one method invocation on a remote object. Methods invocations using asynchronous tickets are syntactically identical to local method invocations but are executed asynchronously.

- Babylon v2.0 adds a new remote object creation mechanism which takes an existing locally created object and turns it into a remote object by copying it to a server. Babylon v2.0 also supports Babylon v1.0’s old remote object creation mechanism which creates a new remote object instance based on a programmer specified class name. Unlike v1.0 and many other existing systems, Babylon v2.0 supports nested remote object creation.

- Babylon v2.0 includes a completely redesigned class loading architecture. Each client is now assigned a private class loader and namespace which prevents naming conflicts and allows the JVM to garbage collect client classes when a client exits.

- Babylon v2.0 clients can now create persistent remote objects that are not garbage collected when the client exits but remain usable until they are explicitly unexported. References to persistent remote objects can be obtained and used by other clients via new remote object lookup methods.

- Babylon v2.0 includes a new remote object migration mechanism which no longer relies on deprecated methods and addresses the consistency issues that can affect migrated remote objects in Babylon v1.0. Furthermore, Babylon v2.0's optimized implementation of remote object migration has reduced migration times by almost 40% for small remote objects when compared with Babylon v1.0.

- Babylon v2.0 provides basic programmer-controlled fault-tolerance via a client initiated checkpointing mechanism that enables clients to write checkpoints of remote objects to disk and load these checkpoints back onto a different server in the event of a failure.

- A new Babylon v2.0 server interface enables server administrators to view which remote objects are running on a server and migrate them to a different server if desired. The interface also provides basic server metrics, server thread information and the ability to unexport persistent remote objects.

- Performance evaluation results show that the implementation of Babylon v2.0 is as fast or faster than Babylon v1.0 across a variety of clustered computing experiments. Experiments also demonstrate that Babylon v2.0 performs well with larger parallel applications that do not adhere to the simple master-worker or branch-and-bound computation model.

In addition to these enhancements and new features, Babylon v2.0 maintains the strengths and benefits of the first version. Babylon is still 100% Java compliant and can be used on any platform that supports the JVM. Furthermore, Babylon v1.0's remote I/O facilities are still supported and have been made more accessible to clients in Babylon v2.0 because their use no longer requires remote objects to extend a special Babylon class. We believe that the solid performance results and the combination of new and old features make Babylon v2.0 a unique and powerful system for distributed application developers.

1.4 Outline of Thesis

We now provide an outline of the remainder of the thesis. Chapter 2 describes the design and implementation of Babylon v1.0 in detail and explains how the system has changed in Babylon v2.0. It then concludes with a survey of related Java-based distributed object systems and compares and contrasts these systems with Babylon v2.0.

Chapter 3 covers key Babylon v2.0 features and primitives from a programmer's perspective. Several examples illustrate the design changes and the new Babylon v2.0 programming interface. Chapter 4 describes Babylon v2.0's architecture and explains the motivation behind the design decisions.

Chapter 5 presents the results of experiments that examine the performance of various Babylon v2.0 operations and demonstrate that performance has not been adversely affected by a major design overhaul and the addition of several new features. Chapter 6 presents the conclusions of the thesis and discusses the strengths and weaknesses of Babylon v2.0. Also described are potential areas of future work.

Appendix A provides a listing of the contents of the default Babylon v2.0 policy file. Appendix B provides the code for a class used in several examples throughout the thesis. Appendix C includes a more complete example of asynchronous remote method invocation and remote object lookup in Babylon v2.0. Appendices

D and E provide the complete source for the heat diffusion and matrix multiplication test applications, respectively. Appendix F discusses the JVM tuning parameters used to conduct the experimental evaluation of Babylon v2.0.

Chapter 2

Background and Related Work

Babylon v2.0 builds on many of the ideas and concepts of Babylon v1.0. This chapter begins with an overview of Babylon v1.0 and describes its design and implementation. A substantial number of other Java-based distributed and parallel systems have also been developed in recent years but the design and range of problems addressed by each varies substantially. Relevant Java-based distributed and parallel projects in the area are described in the second part of this chapter.

2.1 Babylon v1.0

Babylon v1.0, a distributed object system developed by Matthew Izatt, is a collection of Java classes that facilitate the creation of distributed and parallel applications. It is written entirely in Java and supports features such as remote object creation, asynchronous remote method invocation, object migration and remote I/O facilities.

Babylon v1.0 provides the basis for many of the ideas in Babylon v2.0. Although the implementation has changed considerably, many of the features and underlying mechanisms of the first version remain an intrinsic part of Babylon v2.0. This section describes the design and implementation of Babylon v1.0 and highlights its strengths and weaknesses.

2.1.1 System Overview

A number of the ideas in Babylon v1.0 came from two earlier distributed Java libraries called Ajents and ParaWeb. Babylon v1.0 built on experiences with these systems to provide improved mechanisms for creating and using remote objects. Clients using the Babylon v1.0 distributed object framework can create distributed objects, called *worker objects*, on remote hosts. A local object, called a *worker identifier*, holds a remote reference to a worker object and is used by a client to interact with the worker.

Babylon v1.0 includes a server component that must be running on each machine designed to host worker objects. The server provides a point of contact on the remote host and manages the worker objects running on the specified machine. Babylon v1.0 also includes a scheduler component that is used by clients to obtain server references. These components and their associations are displayed in Figure 2.1. Babylon v1.0 relies on Java's Remote Method Invocation (RMI) [23, 45] for the underlying communication mechanism. RMI greatly simplifies the implementation of Babylon by providing built-in support for distributed garbage collection and a simple mechanism to interface remote components without having to use low-level socket communication.

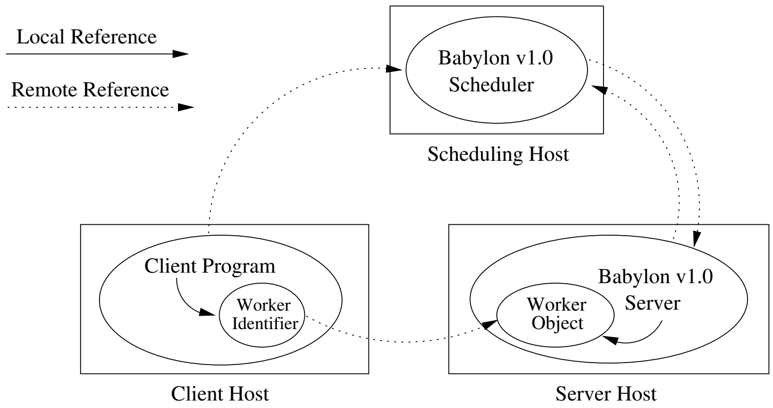

Figure 2.1: Principal Components of the Babylon v1.0 Framework

Babylon v2.0 uses a similar structure and also uses RMI as the underlying communication mechanism. A scheduler provides clients with references to servers and the servers host worker objects for clients. However, unlike the first version, Babylon v2.0 servers can host multiple worker objects, and multiple servers can be running on a single machine. This enables servers to better take advantage of multiple CPUs in a multiprocessor system. Several of the application experiments in Chapter 5 take advantage of the new server and worker object configurations made possible by this enhancement. Babylon v2.0 servers also provide a new command-line interface that can be used to administer the server and to manage the worker objects residing on it.

2.1.2 Remote Object Creation

Babylon v1.0 uses remote instantiation to create worker object instances on Babylon v1.0 servers. Clients create worker objects with the static `Babylon.remoteNew()` method. Figure 2.2 provides an example of remote object creation using this method. The first argument to `Babylon.remoteNew()` is the fully qualified class name (FQCN) of the worker object, represented as a Java string. The other arguments are a client-specified instance name (used for worker object lookups), the location of a Java Archive (JAR) [12] file containing the classes used by the worker object, the destination Babylon v1.0 server (obtained by querying the scheduler for an available server using the `AvailServer()` method), and an optional remote I/O object. When a client calls the `Babylon.remoteNew()` method, a remote object creation request with the specified class name is sent to a Babylon v1.0 server. The server creates a new instance of the specified class using a default no-argument constructor and returns a remote reference to the instance. The only restriction on remote instantiation is that worker objects must include a no-argument constructor.

The `remoteNew()` method returns an object, called a worker identifier, which encapsulates information about the the remote object that has just been created. The worker identifier is used to identify the target worker object for any future operations such as remote method invocations and migration requests.

Babylon v2.0 also supports worker object creation using remote instantiation but provides a second mechanism for creating worker objects called *exporting*. Exporting takes an existing locally created object and turns it into a remote worker object by copying it to a Babylon v2.0 server. This mechanism preserves the state of the original object and requires that the original object and any subordinate objects be serializable.

```java
BabylonObj workerId = null; // The worker identifier.

try {
   workerId = Babylon.remoteNew(
      "uw.babylon.Dictionary", // The worker's FQCN
      "myDictionary",          // An instance name for the worker
      "./dictionary.jar",      // The location of the JAR file holding the classes
      scheduler.AvailServer(), // The target server
      remoteIO                 // A remote I/O reference
   );
}
catch (FileNotFoundException fileEx) {
   // Error loading JAR file.
}
catch (RemoteInstantiationException instEx) {
   // Remote object creation failed for some other reason.
}
```

Figure 2.2: Remote Object Creation in Babylon v1.0

This technique permits clients to turn third-party object instances into worker objects or initialize and work with objects locally before turning them into worker objects. Unlike remote instantiation, worker objects created by exporting do not require a default constructor. A further addition in Babylon v2.0 is support for nested remote object creation. In other words, a worker object running on a remote Babylon v2.0 server can use `Babylon.remoteNew()` or `Babylon.export()` to create another worker object. This is not possible in v1.0 and most other existing Java-based systems.

Babylon v2.0 also includes several new properties that can be used to configure individual worker object behaviour. For instance, clients can now specify whether their remote worker objects should have *private* or *public* access. Private worker objects can only be used by the client that created them. Public worker objects, on the other hand, can be used by any client using the Babylon v2.0 object framework. Babylon v2.0 also includes a new worker object registry which can be used to obtain references to other client's public worker objects based on their instance name. Babylon v1.0 has no such registry and worker objects always have private access.

Like Babylon v1.0, Babylon v2.0 works closely with the distributed garbage collector which automatically garbage collects remote worker objects that are no longer referenced by any clients. Worker objects that are automatically garbage collected when they are no longer referenced are called *transient* worker objects. Babylon v1.0 only supports transient worker objects. Babylon v2.0 adds the option of creating *persistent* remote worker objects that remain live as long as the Babylon v2.0 server hosting them is running or until it is explicitly unexported by the server administrator.[1] Persistent worker objects can be used to provide a continuously available service to all clients. For example, a persistent public worker object with a well-known instance name could provide a very efficient and fast matrix multiplication algorithm. Any client could simply lookup the worker object and use it to perform their matrix multiplication without having to implement their own matrix multiplication algorithm, thus enabling Java-based application servers.

2.1.3 Synchronous Remote Method Invocation

Babylon v1.0 clients can invoke methods on worker objects using the static `Babylon.rmi()` method. Figure 2.3 provides an example of remote invocation in Babylon v1.0. The `Babylon.rmi()` method takes

[1] A persistent worker object can be explicitly removed from a Babylon v2.0 server with the `Babylon.unexport()` method or by using the administrative command-line interface on the server hosting the worker object.

three parameters, a worker identifier that identifies the target worker object, the name of the target method in the form of a Java string, and an `Object` array holding the arguments to the target method. This method also has several overidden versions to support passing the arguments to the target method as individual parameters instead of as `Object` array. The result of the remote method invocation is returned as an `Object` and must be cast to the appropriate type. Babylon uses Java's serialization facilities to pass arguments and return values to and from the worker object. Consequently, the return value and the arguments to the target method must be serializable.

This invocation mechanism has three serious drawbacks: (i) using strings to identify the target method is awkward, error-prone and does not provide transparent access to worker objects, (ii) method arguments are passed as an `Object` array which prevents the use of primitive type method arguments and (iii) it inhibits compile time checking of the method name (whether or not the given method actually exists), the method arguments (whether or not the number and types of the arguments match the method signature) and the return type.

If the remote method throws an exception during an invocation, the exception is caught by `Babylon.rmi()` and wrapped in a `babylon.core.RemoteExecException` before being rethrown to the client. A client can attempt to catch the `babylon.core.RemoteExecException` which includes a method, `getTargetException()`, to retrieve the original exception. It should be noted that `babylon.core.RemoteExecException` extends `java.lang.RuntimeException` and is thus not required to be caught in the `catch` clause of `Babylon.rmi()`.[2] Therefore, a client can decide to ignore all exceptions thrown by a worker object without any complaints from the compiler. Another disadvantage is that a client cannot directly catch exceptions thrown by a remote method. Instead, it must catch the `babylon.core.RemoteExecException`, retrieve the original exception and examine it to determine which exception was actually thrown.

```
Object arg = new String("obnubilate");
try {
   /* Invoking method getDefinition() on a worker in Babylon v1.0. */
   String definition = (String) Babylon.rmi (
      workerId,         // Worker identifier
      "getDefinition", // String representation of method
      arg               // Argument to method
   );
}
catch (babylon.core.RemoteExecException ex) {
   // The target method threw an exception.
   if (ex.getTargetException() instanceof
    babylon.examples.WordNotFoundException) {
      // The word was not found.
   }
}
```

Figure 2.3: Remote Method Invocation on a Worker Object in Babylon v1.0

Babylon v2.0 radically improves this interface by using dynamic proxies [49], which became part of the Java standard in v1.3, to provide transparent access to worker objects based on Java interfaces. The remote object creation methods in Babylon v2.0 return dynamic proxies for newly created worker objects. Clients can use these dynamic proxies to invoke methods on worker objects using standard Java method invocation syntax. Unlike Babylon v1.0, remote method invocation in Babylon v2.0 permits the use of primitive type

[2]Exceptions that extend `java.lang.RuntimeException` form a special class of Java exceptions that can be thrown by methods but are not required to be caught by the caller.

method arguments and ensures type checking of the target method and the arguments at compile time. A dynamic proxy to a worker object can safely be passed as an argument or returned as a result in any local or remote method invocation.

Dynamic proxies are similar to Java RMI stubs in that they make remote objects available via the Java interfaces they implement but differ from RMI stubs in two important ways: (i) dynamic proxies are generated dynamically at run-time instead of using a special stub compiler and (ii) the interfaces used by Babylon v2.0 for worker objects don't need to extend the `java.rmi.Remote` interface. However, it is important to remember that, like RMI, Babylon v2.0 worker objects need to implement a client-defined interface and will be accessible to clients only via the methods defined in this interface.

Since Babylon uses Java RMI as the underlying communication mechanism, parameters and return values are passed to and from worker objects using standard RMI semantics. In other words, primitive type and object parameters are passed by value using object serialization but remote object parameters (including Babylon dynamic proxies) are passed by reference. Another side-effect of using RMI as the underlying communication mechanism is that a new thread is started at the server by the RMI runtime for each method invocation on a worker object. Programmers should be mindful of this fact, especially when creating public worker objects, because several clients may be invoking methods on the worker object concurrently. The methods of such worker objects should be synchronized or capable of safely handling multiple concurrently executing threads.

Another Babylon v2.0 enhancement is the addition of stateful remote method invocations. Each remote method invocation in Babylon v2.0 includes context information that identifies the calling client. This context information is used by servers to authenticate the caller and to restrict invocation access to private worker objects. This is crucial in an environment where clients can obtain references to worker objects belonging to other clients. In some cases, clients may not want others to invoke methods on their worker objects. Stateful remote method invocation is used to ensure that when a client creates a private remote object, it is the only client that can invoke methods on that object.

2.1.4 Asynchronous Remote Method Invocation

Network latency in a distributed application environment can incur significant overhead and reduce overall application performance. One way to reduce the impact of network latency is to overlap communication and computation [46]. Asynchronous remote method invocations allow an application to continue working while a remote method invocation completes. Overlapping computation and communication in this way can improve application response time and increase overall performance.

Babylon v1.0 supports asynchronous remote method invocations with the static `Babylon.armi()` method. This method is non-blocking and returns a `babylon.core.Future` object which acts as a container for the return value of the invoked method. The `Future` object can be queried at any point to retrieve the method's actual result using the future's `get()` method. The future object resolves on the return value of the target method and `get()` will block if the return value is not yet available.

Parameters and return values passed to the worker object using asynchronous remote method invocation adhere to standard RMI semantics. The parameters to the `Babylon.armi()` method are the same as the parameters to the `Babylon.rmi()` method and, consequently, `Babylon.armi()` suffers from the same deficiencies. Namely, primitive type method arguments are not supported and errors such as invoking a non-existent method, passing an incorrect number of arguments, or passing arguments of the wrong type are not detected at compile-time. Figure 2.4 provides an example of asynchronous remote invocation in Babylon v1.0.

```
Object arg = new String("omphaloskepsis");

/* Invoke method getDefinition() asynchronously in Babylon v1.0 (non-blocking) */
Future future = Babylon.armi (
   workerId,        // Worker identifier
   "getDefinition", // String representation of method
   arg              // Argument to method
);

// Other code that executes concurrently with getDefinition() goes here.

try {
   // Retrieve the result from the Future (blocks if result is not yet available).
   String definition = (String)future.get();
}
catch (babylon.core.RemoteExecException ex) {
   // The target method threw an exception.
}
```

Figure 2.4: Asynchronous Remote Method Invocation on a Worker Object in Babylon v1.0

Babylon v2.0 uses a novel technique based on special proxy objects called *asynchronous tickets* to support asynchronous remote method invocation. An asynchronous ticket can be used to make the next method invocation on a worker object asynchronous. The method is invoked on the ticket using standard Java invocation syntax but the invocation completes asynchronously. Applications can continue running normally while the invocation completes.

When a client invokes a method on an asynchronous ticket, a service thread is started on the client which handles the remainder of the invocation. The client thread returns immediately while the service thread performs a normal synchronous remote method invocation for the requested method. Parameters and return values passed to the worker object using asynchronous remote method invocation follow standard RMI parameter passing semantics. The method's return value can be requested from the ticket at a later time. Like Babylon v1.0, Babylon v2.0 asynchronous remote method invocation resolves on the return value of the target method. Resolving on the return value ensures that any other side-effects have completed. This approach is 100% Java compatible and uses Java's standard method invocation syntax which can be checked at compile-time. Asynchronous tickets are described in detail in Section 3.5.2.

2.1.5 Object Migration

A key feature of Babylon v1.0 is the ability to freely move remote objects from one Babylon server to another. Object mobility can be used to support dynamic load balancing (i.e., move a remote object from a heavily loaded server to a lightly loaded server), fault-tolerance (i.e., move a remote object from a faulty server to a stable server) or to exploit server locality (i.e., move a remote object to a closer server with lower communication latency).

Babylon v1.0 supports three types of migration for remote objects. *Idle migration* takes an idle remote object and moves it to a new server. This form of migration can only be performed on worker objects that are not actively executing methods. Babylon v1.0 also supports two types of migration for worker objects that are not idle. *Delayed migration* allows executing methods to complete before migrating the remote object. No new remote method invocations can be started on the worker object once delayed migration is pending.[3] *Immediate migration* of running objects is also supported using checkpointing and rollback. This

[3]Idle migration is really just a special case of delayed migration. However, we differentiate between the two in this thesis for

form of migration interrupts an executing method and migrates the object immediately using the most recent checkpoint of the worker state. The interrupted method is then transparently restarted on the remote object at its new location.

Babylon v1.0 uses location forwarding [20] to provide relocation transparency for mobile objects. Location forwarding maintains forwarding information for a mobile object at each of the object's former locations, essentially creating a chain leading from the object's original location to its current location. When a Babylon v1.0 client tries to use a stale reference for a worker that has been migrated to a different server, the outdated remote reference is transparently updated by traversing the chain of forwarding locations.

Unfortunately, location forwarding fails if any of the intermediate forwarding locations become unreachable (i.e., an intermediate server failure or a network outage) before all the stale references for the worker object have been updated. Babylon v2.0 uses location forwarding but also stores the most current worker object location information in a worker object registry implemented in the Babylon scheduler. The worker object registry, normally used by Babylon v2.0 clients to lookup references to other client's worker objects, is used to update stale worker references if the new location of a worker object cannot be determined using location forwarding. Both strategies update stale client references transparently. Location forwarding is still used as the primary update method because of its inherent distributed nature which reduces the load on the scheduler.

Immediate migration in Babylon v1.0 is also hampered by three other deficiencies: (i) it can fail unexpectedly if more than one simultaneous remote method invocation is executing on the target remote object when migration is requested, (ii) it relies on the recently deprecated `stop()` method of the `java.lang.Thread` class to interrupt executing methods and (iii) it does not provide a mechanism to programmatically defend against the *checkpoint consistency* problem.[4] In Babylon v2.0, immediate migration has been replaced with a new form of migration called safe-point migration. Safe-point migration is designed to address each of the above mentioned problems.

To address the issue pertaining to immediate migration in the presence of multiple simultaneous invocations (this problem is discussed in detail in Section 4.2.1), Babylon v2.0 provides a special mode of execution for worker objects called *safe mode*. Safe mode enforces single threaded execution for a worker object and forces concurrent method invocation requests to be executed sequentially in the order of their arrival. Safe-point migration is only supported for worker objects running in safe mode.

There is currently no replacement for the deprecated `stop()` method and, consequently, no way to stop a Java thread without some cooperation from the running thread [47]. Several papers investigate the problem of thread migration and thread termination in Java but all propose solutions based on special preprocessors or custom JVM extensions [10, 43, 53]. However, one of the objectives of the Babylon v2.0 system is to avoid the use of preprocessors and language extensions so that Babylon v2.0 may be used on any machine supporting the JVM. With these restrictions in mind, Babylon v2.0 supports safe-point migration in the following way: Worker objects that require the ability to be immediately migrated must include calls to `Babylon.setMigrationPoint()` in their worker object code at points where safe-point migration can safely be performed. Normally, `Babylon.setMigrationPoint()` does nothing and simply returns. However, if safe-point migration is pending, this method will throw a `babylon.core.BabylonThreadDeath` object as an exception. Unless caught by the worker object code, this exception will propagate up to the server. When the server catches the exception, it knows that the

consistency with Matthew Izatt's description of Babylon v1.0 and to simplify the discussion of the implementation details.

[4]The checkpoint consistency problem [15] is a well-known problem that can occur with rollback-recovery protocols when a distributed object interacts with an external entity which cannot be rolled back. This can produce an inconsistent system state if the distributed object is rolled back to a state prior to the interaction with the external entity.

worker thread has been stopped and that the worker object can safely be migrated. A worker object can also catch the `BabylonThreadDeath` exception if it needs to perform any cleanup tasks before migration occurs, provided the exception is rethrown when cleanup tasks complete.

The above approach provides a 100% Java compatible and thread safe mechanism for stopping worker threads and, if required, gives worker objects the opportunity to perform cleanup tasks or other recovery tasks to defend against the checkpoint consistency problem before migration occurs. For instance, a worker object could catch a `BabylonThreadDeath` exception after setting a safe migration point and use this opportunity to close open I/O connections or undo an operation that affects an external component. However, the drawback of this approach is that the source code of the worker object must be available so that calls to `Babylon.setMigrationPoint()` can be added at safe migration points.

2.1.6 Remote Class Loading

Unlike most programming languages, Java provides a very flexible class loading mechanism that only finds and loads classes at run-time when they are actually needed [32, 55]. Normally, the virtual machine looks for class data in the form of "class files" that reside in the filesystem. However, developers can customize how the virtual machine finds and loads classes by implementing their own custom class loaders.

Babylon v1.0 uses custom class loaders to load worker object classes over the network. The class files for a worker object must be placed in a JAR file whose location is specified as an argument to the `Babylon.remoteNew()` method when the client creates the worker object. The JAR file is transmitted to the target server along with the remote object creation request. As a result, Babylon v1.0 servers do not need local filesystem access to the class files of the worker object and clients can run their worker objects on remote servers without requiring login access to the server machine. JAR files are used for class file transmission in order to reduce the number of messages required to obtain the class data for the worker object from the client. All the required classes are downloaded with a single message as opposed to a separate message for each required class. For worker objects that use many classes, this approach can greatly reduce the transmission overhead.

However, Babylon v1.0 servers use a single custom class loader instance to load classes for every client. It should be noted that once a class has been loaded and defined by a class loader, no other classes with the same name can be loaded by that class loader. This introduces several problems in a system that is designed to be used by many different clients. Firstly, no two clients can ever use a class with the same name. Secondly, Babylon v1.0 servers need to be restarted every time a client changes his/her classes otherwise the new class definitions cannot be loaded. Thirdly, since classes in the JVM are only garbage collected when the class loader that defined them is garbage collected, unused worker object classes in Babylon v1.0 can never be garbage collected.

The Babylon v1.0 class loading mechanism also introduces a security problem because clients have full access to each others classes on any given server. One client could potentially create an instance of a class that was written and loaded by a completely different client because every client shares the same class namespace. This could result in unexpected behaviour for the client that created the class if methods of the class modify a shared (static) variable or have some other side-effect that could affect other instances of the same class. This type of behaviour is unacceptable in a distributed object framework designed to host remote objects for many different clients.

Babylon v2.0 servers create a new instance of a custom class loader for each client. Each class loader manages its own namespace. Providing separate namespaces for each client solves the Babylon v1.0 class loading issues described above: (i) Different clients can safely use identical names for their classes without causing naming conflicts. (ii) Clients can change their classes and simply restart their application to load and use the new class definitions. (iii) When a client application completes, the class loader and the classes loaded by the client can be garbage collected. (iv) Clients no longer share a single class namespace and, consequently, no longer have access to each other's classes.

2.1.7 Remote I/O

Babylon v1.0 includes a mechanism for performing I/O operations with worker objects using server-side callbacks. This technique works by creating an I/O server inside the client which can be remotely referenced and used by a worker object using RMI. Babylon v1.0 provides wrapper classes for many standard Java I/O classes. These wrapper classes are essentially RMI servers with interfaces that more or less match their standard Java I/O counterparts. A worker object that needs to perform console, file or socket I/O can obtain a remote reference to the appropriate wrapper object and use it instead of the normal Java I/O class. For instance, a worker object that wants to write log information to a file on the client host can do so by obtaining a remote reference to a `babylon.io.RemotePrintWriter` instance residing in the client application. `RemotePrintWriter` is a wrapper class for `java.io.PrintWriter` and can be used by the worker object to write the log entries using equivalent wrapper methods.

Worker objects that want to use remote I/O in Babylon v1.0 must extend the `babylon.core.RemoteObj` class. This class holds a remote reference to the I/O server in the client application. A worker object that extends `babylon.core.RemoteObj` can call `getIO()` to retrieve this reference and use it to perform I/O operations. Babylon v2.0 has maintained Babylon v1.0's support for remote I/O. However, Babylon v2.0 worker objects no longer need to extend `babylon.core.RemoteObj` to be able to use this feature. Babylon now provides the static `Babylon.getIO()` method which can be called from any worker object to obtain a remote reference to the client's I/O server.

2.2 Related Work

There is currently a large number of commercial and academic Java-based distributed computing projects in various stages of development. However, the objectives and underlying technologies of each project vary significantly. Some focus on the emerging grid while others are designed for very specialized groups of computational problems. This section provides brief descriptions of some of the more relevant Java-based distributed computing projects and explains how they relate and compare to Babylon v2.0.

2.2.1 JavaParty

JavaParty [24, 40], developed at the Institute for Program Structures and Data Organization in Karlsruhe, Germany, is a companion to Java that provides distributed application developers with tools to convert stand-alone multithreaded applications into distributed applications capable of executing on a cluster of workstations. The main focus of JavaParty research is optimizing remote method invocations. JavaParty uses locality detection and an optimized RMI implementation called KaRMI [39] to achieve fast remote method invocations on target objects that are local to the caller.

JavaParty introduces a new keyword, `remote`, that allows developers to identify the classes of objects whose instances should execute remotely. Developers must annotate the class declarations of the objects they wish to distribute with the `remote` keyword and compile the classes using the provided JavaParty compiler. This greatly simplifies the task of programming distributed objects but comes at the expense of Java compatibility. This is notably different from Babylon v2.0, which has maintained 100% Java compatibility and uses no special preprocessors or language extensions. Babylon v2.0 also supports the remote execution of objects whose source code is not available. To use JavaParty, the source code for the classes of all objects that need to execute remotely must be available so that they can be annotated with the `remote` keyword.

Unlike Babylon v2.0, JavaParty does not support remote class loading and, consequently, remote JavaParty virtual machines must have access to all the application class files through the local filesystem. Without remote class loading, users of the system need access to all the machines that might potentially host their remote objects and copy the application class files to those machines. This would be impractical in any large-scale distributed environment. Babylon v2.0 supports remote class loading, meaning that users do not need to copy their class files to server machines because the servers automatically load the application classes from the user's machine over the network.

2.2.2 Javelin

Javelin [14, 36, 37], an ongoing project at the University of California in Santa Barbara, is a Java-based software system that provides a framework for building large-scale distributed applications. Development on Javelin began in 1997 and early versions, namely Javelin and Javelin++, supported master-worker distributed applications written as Java applets. Javelin 2.0 abandoned the applet-based programming framework and added support for problems that could be formulated as branch-and-bound computations. The most recent version, Javelin 3, includes an advanced eager scheduling algorithm that provides fault-tolerance.

Eager scheduling works by assigning distributed tasks to hosts until every task has been scheduled at least once. At this point, redundancy is introduced by rescheduling any tasks that have not yet completed. This process continues until every task completes. This form of aggressive scheduling can mask both node and link failures because tasks that do not finish for whatever reason are eventually rescheduled on different hosts. Scheduling is not the focus of Babylon v2.0 but Javelin's scheduling and scalability research may be very applicable to future Babylon projects.

However, Javelin only supports applications that adhere to the the master-worker or branch-and-bound computational models. Programmers must implement their Javelin programs according to the programming patterns defined for each model. This is very restrictive and makes it difficult to port existing applications to Javelin. Babylon v2.0, on the other hand, is a much more general distributed object framework that can host a broad range of distributed and parallel applications and provides transparent access to remote objects. Babylon v2.0 also supports many other features such as object migration, communication between distributed objects and remote I/O facilities which are not provided in Javelin.

2.2.3 Globus and the Java Commodity Grid Kit

Globus [1, 2, 19, 18] is a comprehensive toolkit for developing large-scale distributed applications and environments. It is implemented as a set of UNIX shell scripts and C programming APIs that provide secure authenticated access to remote computing resources. The toolkit includes tools and services for remote job submission, resource discovery, user management, fault detection and security. The Globus toolkit can be used to transform a cluster of computers networked together into a virtual supercomputer.

The Globus Resource Allocation Manager (GRAM) is a Globus service for submitting jobs and for monitoring job progress. Job information needed by GRAM, such as the location of the executable and command-line arguments, can be described using the Resource Specification Language (RSL). Globus also supports resource discovery using the Metacomputing Directory Service (MDS) which is based on the Lightweight

Directory Access Protocol (LDAP), an existing directory service standard. All these services are built on top of the Grid Security Infrastructure (GSI) to provide secure access to remote resources.

The Java Commodity Grid Kit (CoG) [56, 57] is a Java library whose API maps Java classes and methods to most of the Globus Grid services normally available to clients. This toolkit provides grid access for clients that are not necessarily running Globus software. Clients can use the toolkit's GRAM package to submit jobs or the MDS package to search for available nodes. It should be noted that the Java CoG also includes graphical user interface components for job and node management as well as resource discovery.

Although the Java CoG has made the grid more accessible to application developers, the interfaces are still relatively low-level and require a thorough understanding of grid middleware and the services that expose the grid. Furthermore, expressing job submission parameters and requirements in RSL can be cumbersome and complicated. Consequently, turning a regular Java application into a Java Grid applications still requires significant code changes and work. Babylon attempts to minimize this impact while still providing many of the features associated with distributed computing. The Java CoG toolkit could eventually be used to grid-enable Babylon programs but this feature has not been implemented in the current version.

2.2.4 ProActive

ProActive [4, 5, 8] is a Java library that provides tools and services for parallel and distributed application development. It is written entirely in Java and does not require any modifications or extensions to the virtual machine. Developers can use ProActive to remote-enable existing objects, invoke methods on remote objects and migrate idle remote objects between hosts. ProActive also supports a transparent group communication mechanism that may be applicable to future Babylon v2.0 projects in the area of fault-tolerance and remote object replication.

ProActive uses Apache's Byte Code Engineering Library (BCEL) [13] to dynamically create proxies for remote objects at runtime. BCEL is used to read in a worker object's class file, convert it to a new proxy class that is a subclass of the worker object's class and write it back out as a new class file, essentially creating a new class at runtime. All the public methods of the worker object's class are overridden in the dynamically generated proxy class and retrofitted with a custom implementation that delegates the invocation to the target remote object. The proxy preserves the worker object's class definition and therefore clients can use the proxy in the same way that they would use a local instance of the worker object.

Remote methods invocations using ProActive proxies are always executed asynchronously unless the method's return type cannot be subclassed (i.e., it returns a primitive type or a final class) or if the method throws a checked exception.[5] Remote methods that fit this description are always executed synchronously.

In Babylon v2.0, method invocations on remote objects are synchronous by default. If an application wants to make an asynchronous invocation it must first obtain an asynchronous ticket for the target remote object. Methods invoked on the asynchronous ticket are typed and are always executed asynchronously. Babylon's approach clearly distinguishes between synchronous and asynchronous remote method invocations and works with all methods, including those that return primitive-type objects or throw exceptions. However, the disadvantage of Babylon's asynchronous method invocation approach is that extra steps are required to obtain the asynchronous ticket and retrieve the method result.

Unfortunately, ProActive does not support separate client namespaces for classes on ProActive servers. This introduces many of the same class loading problems present in Babylon v1.0. For instance, ProActive servers must be restarted every time a worker object class changes because the previous version of the class

[5] A checked exception is an exception that is declared in the `throws` clause of the method declaration.

cannot be unloaded from the server. Another problem is that no two clients can ever use a class with the same name. Babylon v2.0 has been carefully designed to avoid these problems. It also supports additional features which are not supported by the ProActive library such as safe-point migration and access restrictions for remote objects.

2.2.5 Ninflet

Ninflet [51] is another attempt at building a Java framework to leverage the unused computational power of idle computers on the Internet. It was built to support object-based programming and takes advantage of many of the existing features of the Java language such as security, remote method invocation and object serialization. The components of the Ninflet system are similar to those in Babylon and include servers, clients and a scheduler. Ninflet servers host objects for clients and must register themselves with a scheduler. The scheduler manages the servers and provides clients with servers to host their objects.

Ninflet only supports master-worker applications and provides template classes for writing Ninflet objects that conform to this model. Application programmers must write worker objects by subclassing the provided `Worker` class and overriding certain methods. Similarly, master objects must be written by subclassing the provided `Master` class. Ninflet does not support any other computational models which seriously limits the types of applications that it can host. Furthermore, porting an existing application to Ninflet would require significant code modifications to conform to the supported design pattern.

In contrast, Babylon v2.0 is not limited to a specific computational model and supports distributed application development using normal object interactions. Remote objects in Babylon v2.0 do not need to extend custom classes or adhere to special programming patterns. This can greatly simplify distributed application development and also reduce the effort required to turn an existing stand-alone application into a distributed application.

Ninflet also includes a web-based administration interface that enables server providers to manage the servers running on their machines and migrate idle remote objects to different machines. However, Ninflet does not support many other useful distributed object programming features such as asynchronous remote method invocations, safe-point migration and remote object lookups. Babylon provides these features and also supports transparent remote object interactions which do not limit it to a particular computational model.

2.2.6 Charlotte

Charlotte [6, 7], a metacomputing project started at New York University, attempts to harness the power of idle computers on the Internet using Java's applet technology. Charlotte provides classes that support distributed shared memory (DSM) semantics on top of the standard JVM. DSM semantics are activated in special sections of code that execute concurrently called *parallel routines*.

A parallel routine must be encapsulated in a class that extends Charlotte's `Droutine` class. The `drun()` method of the `Droutine` class must be overridden in the programmer's class and should implement the body of the parallel routine. The `Droutine` class is somewhat analogous to a Java thread and the `drun()` method corresponds to a thread's `run()` method. Programmers install parallel routines in their code using the `addRoutine(Droutine parallelRoutine, int numTasks)` method. The parameters to this method are an instance of a parallel routine and the number of instances of the routine that should execute concurrently. Parallel routines are executed in parallel on all machines participating in the computation.

Charlotte also uses an eager scheduling algorithm that provides fault-tolerance by scheduling tasks redundantly. To ensure result and memory consistency in the presence of multiple executions of a particular task, Charlotte uses a two-phase idempotent execution strategy (TIES). The first phase of TIES stores modifications to shared data in a temporary buffer. After all tasks complete, the second phase examines the data for consistency and commits the modifications to the shared data region idempotently.

Charlotte uses an interesting distributed programming approach that is well-suited for the dynamic nature of the Web. It makes use of existing Internet infrastructure such as HTTP servers and web browsers running applets to accomplish its goals. However, Charlotte programs must conform to the parallel routine program structure and must be implemented as a Java applet. This approach is not very transparent or flexible because programmers must adhere to both the applet API and Charlotte's parallel routine structure. Charlotte also lacks distributed object programming features available in Babylon v2.0 including object mobility and transparent remote object access. Furthermore, Babylon v2.0 avoids many of the complexities of shared memory access by using remote method invocations for remote object communication.

2.2.7 Voyager

ObjectSpace's Voyager [42] is a comprehensive commercial Java-based distributed computing environment. Voyager supports a wide range of features from asynchronous remote method invocation to remote object migration. Voyager is based on the Object Request Broker (ORB) architecture but does not limit itself to a specific remote messaging standard. It supports not only CORBA but also RMI, the Distributed Component Object Model (DCOM) and the Simple Object Access Protocol (SOAP). As a result, Voyager can act as the "glue" between a variety of distributed applications using whichever underlying remote messaging standard is appropriate.

Voyager, like Babylon v2.0, exports objects based on the interfaces they implement. A remote object can then be accessed transparently via a dynamically generated proxy that implements the interface of the remote object. Voyager also provides support for many other features such as idle and delayed migration of remote objects, remote class loading, object naming services and group communication. However, Voyager does not support safe-point migration.

However, Voyager's support for asynchronous method invocation is very similar to the approach used in Babylon v1.0. A special method, `Future.Asynch()`, must be used to invoke a method asynchronously on a remote object. The target method is specified as a Java string and the arguments to the remote method are passed as an `Object` array to the `Future.Asynch()` method. As is the case in Babylon v1.0, this approach inhibits compile-time checking of both the method and its arguments. Babylon v2.0 resolves this issue and permits type checking for asynchronous method invocations at compile time. Babylon v2.0 also supports file and socket remote I/O facilities and an safe-point migration mechanism which are not provided by Voyager.

Despite supporting a number of features not provided in Babylon (i.e, group communication, CORBA and DCOM support), Voyager remains a commercial application whose implementation details are not available. This makes it impossible to perform a comprehensive comparison of the implementation and does not allow us to examine design differences and tradeoffs. Furthermore, the proprietary nature of Voyager would make it difficult to use it as a basis for future research projects in the area of distributed systems.

2.2.8 Java/DSM

Java/DSM [59] uses a modified JVM to implement a distributed shared memory (DSM) system across networks of workstations. It has been implemented on top of the Treadmarks [3] DSM implementation which provides the memory consistency algorithms to support parallel computing in a distributed environment.

However, Treadmarks is built as a UNIX library and relies on many operating system features for its implementation. Consequently, the modified JVM used for Java/DSM is also limited to the architectures supported by the Treadmarks system.

2.2.9 Java-Based Mobile Agent Systems

Mobile agent systems such as Aglets [29] and Mole [9] can be used to create mobile Java objects that can move autonomously from one host to the next. A mobile agent framework includes servers that must be running on machines designed to host the agents and provides mechanisms for agent mobility and communication. The intelligent and independent behaviour normally exhibited by mobile agents must be carefully added within a well-defined agent programming framework. This framework is event driven and provides methods that must be overridden to implement the agent functionality.

For instance, mobile agents written using Aglets must extend the `com.ibm.aglet.Aglet` class and the agent's independent and intelligent behaviour must be implemented by overriding certain methods defined in the Aglet Software Development Kit (ASDK). The `onArrival()` method can be overridden to implement operations that must be performed every time the agent arrives at a new location. However, the main computational loop of the agent must be implemented by overriding the `run()` method.

The focus of mobile agent systems is on the movement of objects and not on the interaction and control of them. As a result, these systems do not provide an effortless remote object creation procedure or a transparent communication mechanism. In contrast, Babylon v2.0 is designed to facilitate distributed programming by providing transparent remote object interaction and flexible remote object creation facilities. The emphasis in Babylon v2.0 is on programming simplicity and on special features that simplify distributed application development, not on building independent and autonomous objects that exhibit intelligent behaviour.

2.3 Discussion

The Babylon v1.0 distributed object environment was an important initiative to make distributed resources more widely available to application developers. However, it lacked a flexible and transparent mechanism to create and use remote objects and suffered from deficiencies in key areas such as RMI, remote class loading and object migration. Babylon v2.0 builds on the strengths of Babylon v1.0 and addresses many of the shortcomings.

Several existing systems, such as JavaParty and Java/DSM, use a modified JVM or special preprocessors and language extensions to implement a functional distributed Java framework. One of the objectives of Babylon v1.0 was to create a 100% Java compatible system that does not rely on any custom extensions. Babylon v2.0 continues with this objective and has addressed the problems of v1.0 while providing several new features without requiring a modified JVM implementation, new keywords or other custom language extensions.

Others systems, such as Javelin, Charlotte and Ninflet, only support distributed applications that can be formulated as master-worker or branch-and-bound computational problems. Babylon v2.0 does not limit itself to a particular computational model and provides support for general remote object interaction using standard Java method invocation syntax. As a result, distributed applications that require more complex object interactions such as the grid computation-based heat diffusion benchmark used in Chapter 5 can be written using Babylon v2.0. In contrast, this benchmark could not have been implemented using the aforementioned systems.

Another important Babylon v2.0 enhancement is the addition of separate namespaces on servers for clients. This is a crucial feature in a multi-user environment where several different clients may be running remote objects on a single server at the same time. Many existing systems such as ProActive, JavaParty and the initial version of Babylon do not provide separate class namespaces for clients. In these systems, servers must be restarted every time a client changes any of his/her classes and no two clients can ever use classes with the same name. In addition to supporting multiple class namespaces, Babylon v2.0 also provides access restrictions for remote objects based on context information transmitted with remote method invocations. Access restrictions can be used by a client to prevent other clients from using its worker objects or, alternatively, to share access to the worker object with other clients.

Babylon v2.0 also provides basic fault-tolerance in the form of checkpointing. Clients can make checkpoints of worker objects and write them to the local client disk. In the event of a server failure, the object can be read from the disk and loaded onto a different server. Furthermore, to our knowledge, no other existing distributed Java system supports safe-point migration and remote I/O facilities. In this regard, Babylon provides a unique combination of features to assist distributed application development.

Chapter 3

Distributed Object Programming with Babylon v2.0

Babylon v2.0 facilitates distributed object programming by providing programmers with classes and interfaces for remote object creation, interaction and administration. Most of the Babylon v2.0 distributed object programming primitives are accessible via the static methods of the `babylon.core.Babylon` utility class. This chapter covers the key Babylon v2.0 features and primitives from a programmer's perspective.

3.1 Introduction

A typical Babylon v2.0 application consists of a client program and one or more remote objects running on Babylon v2.0 servers. A client interacts with a remote object, called a worker object, using a local proxy object that transparently delegates requests to the worker object and returns the worker object's results back to the client.

A Babylon v2.0 server provides a virtual environment for running and managing one or more worker objects. The server is the point of contact on the remote machine for clients. Babylon v2.0 also includes a scheduler component that keeps track of Babylon v2.0 servers. The scheduler is used at worker object creation time to locate a server that can host the new worker object and also acts as a worker object directory that can be used to lookup references to live worker objects.

Babylon v2.0 schedulers and servers can run on any machine that supports the JVM. An instance of the Java RMI registry, which can be started using the `rmiregistry` command, must be running on the machine hosting the scheduler. A scheduler can be started with the command `java babylon.sched.SchedulerImpl`. Once the scheduler is up and running, one or more Babylon v2.0 servers can be started with the command `java babylon.server.RemoteObjectServerImpl <scheduling host>`. The scheduler's hostname must be specified as a command-line argument. At startup, a server queries the RMI registry on the specified host to obtain a reference to the scheduler. The server then contacts the specified scheduler to register itself. The scheduler adds the server to its list of available servers so that clients can begin using it to host their worker objects. Unlike Babylon v1.0, Babylon v2.0 supports the presence of multiple servers on a single machine. Babylon v2.0 also supports multiple worker objects within a single server. The ability to run several workers or servers on a single machine can be used to make more effective use of multiple CPUs in clustered multiprocessor environments.

Figure 3.1 illustrates the principal components of the Babylon v2.0 environment and their associations in a sample scenario. Each component is running in a separate JVM, possibly on a different machine. In the figure, two separate worker objects are being accessed by a client program via local proxy objects. Although the worker objects in the figure are running in separate Babylon v2.0 servers, they could also have been running in the same server.

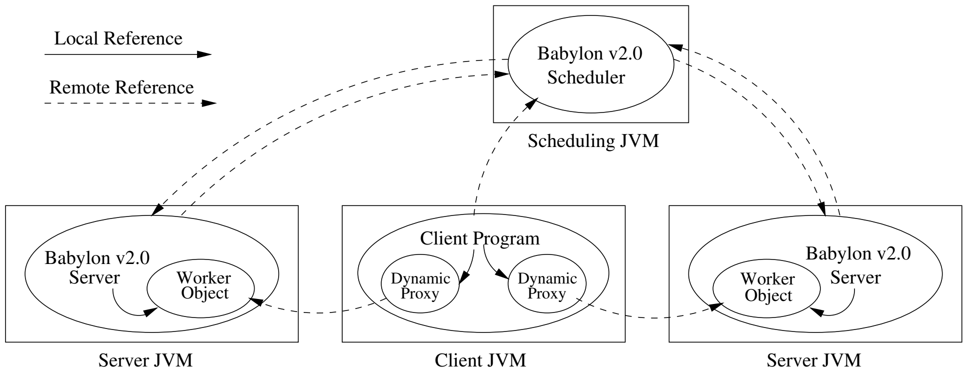

Figure 3.1: Principal Components of the Babylon v2.0 Framework

3.2 The Worker JAR

Normally, Java applications require filesystem access to the class files of objects that they instantiate. However, Java provides a mechanism to extend class loading behaviour so that classes can be loaded from other locations, such as over the network. Like Babylon v1.0, Babylon v2.0 uses a custom class loading strategy to load classes for worker objects over the network in the form of a JAR file.

The worker JAR is a client-supplied JAR file containing the class files needed to create a particular worker object. Babylon v2.0 servers need this information when they instantiate the worker object because they may not have filesystem access to the worker object's class files. Clients specify a worker JAR when they create a worker object using either of Babylon v2.0's worker object creation methods (i.e., `Babylon.remoteNew()` or `Babylon.export()`). The JAR file is automatically delivered to a target Babylon v2.0 server when the worker object creation request is sent.

JAR files can significantly reduce the number of network transactions required to obtain the class data for a worker object. With the use of JAR files, all of the worker object class data is transmitted to a server in a single network transaction as opposed to a separate transaction for each class. This approach can significantly reduce network transmission overhead for worker objects that require many classes. The class data inside the JAR file is also normally compressed, thereby reducing the transmission overhead even further. A similar approach is used by Java applets to reduce the transmission overhead of the applet classes from the web server to the client's machine [55].

3.3 The Application Policy File

In Babylon v2.0, clients can use a policy file to assign access controls and other configuration options to worker objects. For instance, a client can use a policy file to specify whether their worker objects should be public or private, or whether the worker objects should be transient or persistent. Normally, Babylon v2.0 uses a default policy file which is included with the Babylon v2.0 distribution. Appendix A shows the contents of the default policy file and provides a brief description of the policy file format. However, if non-default policies are required, clients must write their own policy file and specify the policy file location at initialization when the `Babylon.initApplication()` method is called (this is described in more detail in Section 3.4).

Currently, the policy file format is implemented as a flat file which specifies global application policies. Global policies are adequate for small client applications with few worker objects but lack flexibility when it comes to larger applications with many different worker objects. In the future, we hope to use a hierarchical structure to enable developers to specify more fine grained policy rules using the policy file. The hierarchical structure will enable clients to associate policies with individual worker objects or with groups of worker objects based on their class.

3.4 Initialization

Babylon v2.0 applications must call `Babylon.initApplication(String schedulerAddress, String jarFile, String policyFile)` to initialize the Babylon v2.0 run-time. This method begins by establishing a connection to the scheduler using the address specified by the `schedulerAddress` parameter. The `schedulerAddress` parameter can be either the IP address or the hostname of the scheduler's machine.

The `jarFile` and `policyFile` parameters are optional. The `jarFile` parameter specifies the default JAR file to use for the creation of remote worker objects. If this parameter is not defined, clients must specify a worker JAR each time they create a worker object by using the appropriate remote object creation method. The `policyFile` parameter specifies the policy file that should be used by the application. If no policy file is given, Babylon v2.0 uses the default policy file (see Appendix A).

The `initApplication()` method also creates an identifier which uniquely identifies the client. Client identifiers are used to enforce access restrictions for private worker objects. When a worker object is first created, the identifier of the creating client is associated with the worker object. This is referred to as the worker object's *control identifier* and uniquely identifies the client that created it. Furthermore, every time a client makes a remote method invocation on a worker object, the invoking client's identifier is transmitted with the invocation request to the target worker object. This is referred to as the *invocation identifier* of the method invocation request. If the control identifier of a private worker object matches the invocation identifier of a method invocation request, the operation is allowed.

3.5 Dynamic Proxies

Babylon v2.0 uses a technique that leverages Java's dynamic proxy classes [49] in order to better support synchronous and asynchronous remote method invocations on worker objects. Dynamic proxies are used to present a location-transparent local interface to remote objects such as worker objects. An important benefit of dynamic proxies is that it becomes possible to create a proxy object for a worker object at run-time which preserves the interface of the worker object.

Babylon v2.0 uses Java's dynamic proxy framework to generate proxies for worker objects. The proxy objects implement the same interface as the worker objects they represent and can be used like a local instance of the object. Methods invoked on the proxy are transparently forwarded to the worker object and the results of the invocation are propagated back. The following sections explain how to create and use worker objects in Babylon v2.0 and demonstrate the differences between Babylon v1.0 and v2.0.

3.5.1 Remote Worker Object Creation

Remote object creation is the process of creating an instance of an object on a remote server and making this instance available to clients. Babylon v1.0 supports remote object creation using *remote instantiation*. This form of remote object creation, which was described in Section 2.1.2, creates a new worker object instance on a remote Babylon v1.0 server based on a fully-qualified class name provided by the client.

Babylon v2.0 maintains support for remote object creation using remote instantiation. The static `Babylon.remoteNew()` method in Babylon v2.0 is similar to the Babylon v1.0 equivalent except that the class of the worker object is no longer specified as a string. In Babylon v2.0, the class of the worker is specified as a `java.lang.Class` object whose existence can be checked at compile-time. For instance, to create a `examples.DictionaryImpl` worker object in Babylon v1.0, the fully qualified class name must be specified as the string `"examples.DictionaryImpl"` to the `Babylon.remoteNew()` method. Much like the check for method names, the existence of the named class can only be verified by the server at run-time and not at compile time. In Babylon v2.0 the class of the worker is simply specified as `babylon.examples.DictionaryImpl.class`.[1]

This approach requires the presence of a no-argument default constructor. Babylon v2.0 can ensure the class exists at compile time, but cannot verify the existence or accessibility of the no-argument default constructor. The constructor may not exist (because another constructor with arguments is defined) or it may not be possible to execute the constructor (because it's defined as private, which is done in the Singleton pattern).

Remote object creation using remote instantiation is useful for creating worker objects from scratch but cannot be used to turn an existing local object into a worker object which preserves the state of the local object. For instance, consider a client that is holding a reference to a local object. This object may have been obtained as the return value of a local method invocation or may simply be an object that was created by the client and required some local initialization. Suppose the client wants to turn this object into a worker object to make it available to other processes. In Babylon v1.0, there is no way to take a local instance of an object and move it to a Babylon v1.0 server so that it can act as worker object.

To address this issue, Babylon v2.0 supports a second technique for creating worker objects, called *exporting*. Exporting takes a local instance of an object and copies it to Babylon v2.0 server where it becomes a worker object. This technique preserves the state of the local object. However, since the local object is copied to the Babylon v2.0 server, only objects that are serializable can be turned into worker objects using this technique.

Figure 3.2 contains a code excerpt that exports a `DictionaryImpl` worker object to a Babylon v2.0 server. To use exporting, clients must first create a local instance of the object they want to export, which we call the *archetype*. The archetype and the interface implemented by the archetype are passed as arguments to the `Babylon.export()` method. The `Babylon.export()` method transmits a copy of the archetype over the network to the destination Babylon v2.0 server. The method returns a proxy object that implements the interface of the worker object and can be used just as one would use the local instance.

The example in Figure 3.2 is based on the `DictionaryImpl` class and `Dictionary` interface listed

[1] As of Java 1.1, programmers can obtain the `Class` object for any loaded class or interface by adding `.class` to the end of the class or interface name in their code.

in Appendix B-1. The code excerpt begins by creating the archetype (line 2) for the desired worker object. The archetype is exported to a Babylon v2.0 server on line 6. A reference to a dynamic proxy implementing the same interface as the archetype is returned to the application. This proxy can be used to invoke any of the methods defined in the `Dictionary` interface on the remote worker object (line 21).

Even though worker objects may implement several interfaces, Babylon v2.0 only supports the specification of a single interface for a dynamic proxy. Furthermore, this interface must be specified explicitely as an argument to the `Babylon.export()` or `Babylon.remoteNew()` method (line 8). Alternatively, the interfaces implemented by the worker object could also be determined dynamically at run-time and automatically associated with the dynamic proxy. This would eliminate the need for the explicit specification of the worker object's interface(s) by the programmer. Future versions of Babylon will likely support both mechanisms.

It should also be noted that the class must implement an interface in order to used as a worker object, which could potentially limit the use of legacy code for which source is unavailable. This could be fixed either by using a bytecode assembler (such as Jasmin) to add an interface into the class definition or by using a system that allows for run-time interface compliance checking.

```
 1 // Create a local instance of the worker object.
 2 Dictionary diction = new DictionaryImpl();
 3
 4 /* Export the object to a Babylon v2.0 server. */
 5 try {
 6    diction = (Dictionary) babylon.core.Babylon.export(
 7       diction,                          // The object to be exported
 8       Dictionary.class,                 // The interface for the proxy
 9       "myDictionary",                   // An instance name for the worker
10       "./dictionary.jar"                // The location of the JAR file
11    );
12 }
13 catch (java.io.IOException ioEx) {
14    // Error locating JAR file.
15 }
16 catch (babylon.core.RemoteInstantiationException instEx) {
17    // Remote object creation failed for some reason.
18 }
19
20 try {
21    String definition = diction.getDefinition("sciolism"); // Remote method invocation
22 }
23 catch (WordNotFoundException ex) {
24    // The word was not found.
25 }
```

Figure 3.2: Worker Object Creation and Method Invocation in Babylon v2.0

After exporting an archetype and obtaining the dynamic proxy for the corresponding worker object, the client should release the original reference to the archetype. Once released (this can be done by setting the reference to null or to another object as on line 6 of Figure 3.2), the archetype can be garbage collected by the JVM. In the example in Figure 3.2, the archetype reference is simply overwritten by the dynamic proxy reference (line 6). This is a convenient way to release the archetype reference but still use the same variable for subsequent interactions with the worker object.

However, it should be noted that the archetype is still a perfectly valid local object even after it has been used to create a worker object. Turning an archetype into a worker object does not modify the archetype in any way. If desired, clients can continue to use the archetype as a local object but it is important to remember that any interaction with archetype will not affect the worker object and any interaction with the worker object will not affect the archetype. They are two different and completely independent objects.

On line 21, the `getDefinition()` method is invoked on the proxy. The `getDefinition()` method, which is defined in the `Dictionary` interface, takes a `String` argument and returns a `String` object. The proxy forwards the invocation to the worker object and, when the worker object completes the invocation, the return value is propagated back to the client. It is important to note that Babylon v2.0 remote method invocation is syntactically equivalent to normal Java method invocation. As a result, Babylon v2.0 permits the use of primitive type method arguments and ensures type checking of the method and the number and types of the arguments at compile-time. Furthermore, exception handling for Babylon v2.0 remote method invocation is also equivalent to standard Java (line 23).

Several forms of `Babylon.remoteNew()` and `Babylon.export()` are available in Babylon v2.0. The form used in Figure 3.2 automatically contacts the scheduler to find an available server for the worker object. However, clients can also use different forms of these methods that permit them to explicitly specify the destination Babylon v2.0 server as an argument. Other forms of `Babylon.remoteNew()` and `Babylon.export()` allow clients to specify an instance name for the worker object being created. The instance name is needed to make worker objects available to other processes. Clients can lookup references to other client's worker objects based on their instance name using the `Babylon.lookup()` method.

Babylon v2.0 also supports nested remote object creation and nested remote method invocation. This enables worker objects to create other worker objects using Babylon's remote object creation methods and to invoke methods on them. Nested worker objects inherit the control identifier of the worker object that created them. The Babylon v2.0 environment also keeps track of invocation identifiers across nested remote method invocations. For instance, if a client, A, invokes a method on a worker object and that this method makes a nested invocation on a method of a different worker object then the the invocation identifier of the nested method invocation will still represent client A. This behaviour is important in order to correctly implement access restrictions for nested worker objects.

For example, consider a public worker object, W_{A1}, created by client A, which then creates a nested private worker object, W_{A2}. W_{A2} will inherit the control identifier of W_{A1} and only threads of invocation originating from client A (i.e., threads whose invocation identifier represents client A) will be able to invoke methods on W_{A2}.

3.5.2 Asynchronous Remote Method Invocation

The static `Babylon.armi()` method is used to perform asynchronous remote method invocations on worker objects in Babylon v1.0. Babylon v1.0 uses future objects to support return values for asynchronous invocations. A `babylon.core.Future` object returned by the `Babylon.armi()` method acts as a temporary container for the result of a remote method invocation. When the result is needed, the programmer can use the `get()` method of the `Future` object to retrieve it.

Figure 3.3 demonstrates an asynchronous method invocation using Babylon v1.0. In this example, the `getDefinition()` method is invoked asynchronously on a worker object (line 2). Behind the scenes, the `Babylon.armi()` method creates a service thread and delegates the rest of the invocation to this thread. Once the service thread is started, the method returns and the remainder of the invocation occurs asynchronously. The result of the method is retrieved later from the `Future` object (line 8).

As with synchronous remote method invocation, asynchronous remote method invocation in Babylon v1.0 is affected by many of the same problems regarding compile-time type checking and primitive type method arguments. In order to resolve these issues, Babylon v2.0 uses a technique based on special proxy objects called asynchronous tickets. Asynchronous tickets serve two purposes. First, they act as a proxy

```java
/* Invoke method getDefinition() asynchronously in Babylon v1.0 */
babylon.core.Future future = babylon.core.Babylon.armi (
   workerId,          // Worker identifier
   "getDefinition",  // String representation of method
   "pedantic"         // Argument to method
);

// Other code that executes concurrently with getDefinition() can go here.

try {
   // Retrieve the result of getDefinition().
   String definition = (String) future.get();
}
catch (babylon.core.RemoteExecException ex) {
   // The target method threw an exception.
   if (ex.getTargetException() instanceof WordNotFoundException) {
      // The word was not found.
   }
}
```

Figure 3.3: Asynchronous Remote Method Invocation on a Worker Object in Babylon v1.0

which implements the interface of the worker object and can be used to invoke a method on the worker object using standard Java invocation syntax. Secondly, they act as a future object which can be used to retrieve the result of the invocation at later time.

The example in Figure 3.4 demonstrates how asynchronous remote method invocation is accomplished in Babylon v2.0. Clients can obtain an asynchronous ticket for any worker object with the static `AsynchTicket.newTicket()` method (line 3). The parameter to the `AsynchTicket.newTicket()` method is the proxy object of the target worker object. Each time this method is invoked, a new asynchronous ticket is created that can be used to make a single asynchronous remote method invocation (line 6). A `MultipleInvocationException` is thrown if more than one invocation is made on an asynchronous ticket. The result of the invocation can be retrieved with the static `AsynchTicket.getResult()` method, which takes the asynchronous ticket as its parameter (line 10).

Any exceptions that are declared in the `throws` clause of the target method's declaration must still be caught when the method is invoked asynchronously (line 8). This is required by the compiler. However, any exceptions that are actually thrown by the target method will only be returned to the client when the `getResult()` method is invoked (line 14) and not when the asynchronous invocation is made (line 7).

Asynchronous tickets simulate asynchronous behaviour by creating a client-side service thread that handles the actual invocation in much the same way as is done in Babylon v1.0. The service thread performs a synchronous remote method invocation of the target method on the worker object. This enables the client thread to return immediately and continue executing while the invocation completes asynchronously. Programmers should also be aware that a separate thread is started on the server by the RMI run-time for each method invocation request. If programmers make more than one simultaneous asynchronous method invocation on a worker object, these methods will execute concurrently (unless the worker object is running in safe mode). In these cases, it is important that programmers make their worker objects thread safe or synchronize access to methods that should not be executed concurrently.

Because an asynchronous proxy also acts as a future for the invocation's return value, it can only be used to invoke a single method on the worker object. If multiple invocations were permitted on a single asynchronous ticket, the ticket would also need to store multiple return values. It would be impractical to determine which of the return values was desired when the application tried to retrieve the result of an invocation. Therefore, a new ticket must be obtained for each asynchronous remote method invocation made by a client. However,

```
1 /* Obtain a new asynchronous ticket */
2 Dictionary asynch_ticket = (Dictionary) babylon.core.AsynchTicket.newTicket(diction);
3
4 /* Invoke method getDefinition() asynchronously (non-blocking). */
5 try {
6    asynch_ticket.getDefinition("logomachy");
7 }
8 catch (Exception e) {}
9
10 // Other code that executes concurrently with getDefinition() can go here.
11
12 /* Retrieve the invocation result (blocks if result is not yet available). */
13 try {
14    String translation = (String) babylon.core.AsynchTicket.getResult(asynch_ticket);
15 }
16 catch (babylon.core.RemoteExecException ex) {
17    // The target method threw an exception.
18    if (ex.getTargetException() instanceof WordNotFoundException) {
19       // The word was not found.
20    }
21 }
```

Figure 3.4: Asynchronous method invocation in Babylon v2.0

this can be quite easily accomplished using an array of asynchronous tickets as is demonstrated in the more complete asynchronous invocation example presented in Appendix B-2.

It is also important to note that Babylon v2.0 returns a dummy value when a client makes the actual asynchronous invocation of the worker object's method using the asynchronous ticket. This is because the real return value is not immediately available. The real return value is retrieved using the `AsynchTicket.getResult()` method. The dummy value corresponds to the Java specification default for the method's return type. This value should simply be ignored (line 6).

Below are brief descriptions of the static methods of the `babylon.core.AsynchTicket` that are used to create new asynchronous tickets and retrieve results from asynchronous remote method invocations.

`Object newTicket(Object workerProxy)`

This method creates and returns a new asynchronous ticket for the worker object associated with the given `workerProxy` parameter. The `workerProxy` parameter can be any proxy object obtained using the `Babylon.export()` or `Babylon.remoteNew()` remote object creation methods. The resulting asynchronous ticket implements the same interface as the `workerProxy` and can be used to make a single asynchronous method invocation on the worker object associated with the `workerProxy` parameter.

`Object getResult(Object asynchTicket) throws RemoteExecException`

This method returns the result of the asynchronous method invocation performed using the asynchronous ticket specified by the `asynchTicket` parameter. It resolves on the return value of the target method. If the result is not yet available, this method blocks until the invocation completes and the result becomes available. If the remote method throws an exception, this method will throw a `RemoteExecException` containing the remote method exception. The original exception can be retrieved by calling `getTargetException` on the `RemoteExecException`.

`boolean isAvailable(Object asynchTicket)`

This method tests if the result of the asynchronous method invocation performed using the asynchronous ticket specified by the `asynchTicket` parameter is available.

3.6 Worker Object Lookup

Worker object lookup functionality is provided with the help of a new worker object registry implemented in the scheduler. The registry maintains a record of all the worker objects in the Babylon v2.0 system. Whenever a client creates or migrates a worker object, the registry is updated to reflect the most recent location of the worker object. Clients can lookup references to worker objects based on the worker object's instance name and the interface it implements. The static `Babylon.lookup(String instanceName, Class workerInterface)` method provides this feature. After locating the worker object with the specified instance name and interface from the worker object registry, the `Babylon.lookup()` method returns a dynamic proxy that can be used to invoke methods on the given worker object. The code example in Appendix B-2 demonstrates how to use the `Babylon.lookup()` method.

3.7 Summary

A major Babylon v1.0 deficiency was the clumsy and error-prone interface used for remote method invocation. Babylon v2.0 addresses this shortcoming and uses dynamic proxies to provide transparent synchronous and asynchronous method invocations. The use of dynamic proxies resolves the compile-time type checking issues of Babylon v1.0 and enables the use of primitive type method arguments. Babylon v2.0 also provides a new remote object creation mechanism called exporting on top of the existing remote instantiation mechanism provided in Babylon v1.0. These features will greatly simplify distributed object programming using Babylon.

In the next chapter, we present the architecture of Babylon v2.0 and provide a detailed explanation of the multi-layered communication model used for client to worker object interaction. The chapter also includes a discussion of the issues surrounding migration in Babylon v1.0 and how these issues have been addressed in Babylon v2.0.

Chapter 4

Babylon Architecture

This chapter describes the architectural design of the Babylon v2.0 framework. Babylon v2.0 uses a multi-layered communication model to provide location and access transparency for worker objects. The model also supports features such as local and nested worker object creation, object mobility and remote class loading. This chapter describes the various components that comprise the Babylon v2.0 framework and how they work together to provide these features.

4.1 Communication Model

The multi-layered communication model of Babylon v2.0 hides the details of remote object communication from the programmer. Several Babylon v2.0 components work together to provide access transparency and location transparency. At the client side, relocation transparency and method invocation delegation are handled by the `babylon.core.ClientAdapter` class. At the server side, the `babylon.server.RemoteAdapter` class provides a point of contact on the server for the worker object. These components are displayed in Figure 4.1.

Figure 4.1 also shows the reference structure of the Babylon v2.0 communication model. The components in the figure work together to delegate method invocation requests from the client to the worker object. The following paragraphs describe the components involved in this delegation process.

On the client host, the *client application* holds a reference to a *dynamic proxy* object for each worker object it has created. The remote object creation methods in Babylon v2.0 return dynamic proxies that should be used by the client to invoke methods on the corresponding worker objects. Each dynamic proxy object implements the *worker interface* of the associated worker object and clients can invoke methods defined in the worker interface on the dynamic proxy, just as they would invoke these methods on a local instance of the worker object.

Babylon v2.0 uses Java's dynamic proxy framework to create the dynamic proxy objects used by clients. The `java.lang.reflect.Proxy` class provides static methods that can be used to dynamically generate proxy classes and instances based on one or more interfaces supplied by the programmer. The generated proxies implement the supplied interfaces and dispatch method invocations to an invocation handler object which is also supplied by the programmer at proxy creation. The invocation handler is an object that implements the `java.lang.reflect.InvocationHandler` interface. Babylon v2.0 uses a `babylon.core.ClientAdapter` instance, called the *client adapter*, as the invocation handler for the dynamic proxy of a worker object.

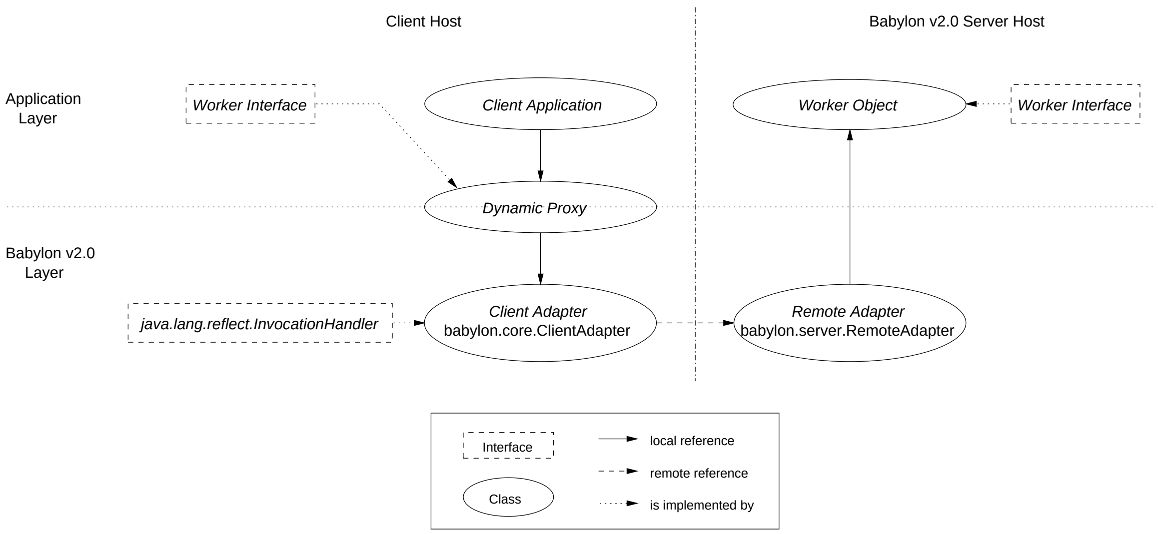

Figure 4.1: Babylon v2.0 Worker Object Communication Model

A client adapter is the client-side access point to the worker object and is where a remote reference to the worker object's `babylon.server.RemoteAdapter` is stored. In addition to forwarding method invocation requests to the remote adapter (which then forwards them to the worker object), one of the client adapter's key functions is to transparently update the `babylon.server.RemoteAdapter` reference if the worker object is migrated to a different server. This reference updating strategy permits worker object migration that is transparent to the clients that use them. The reference updating strategy is covered in detail in Section 4.2.4.

A client adapter cannot reference a worker object directly. All interaction with a worker object must occur through an instance of `babylon.server.RemoteAdapter`, called the worker object's *remote adapter*. A remote adapter is a server-side entity created for a worker object on a Babylon v2.0 server. Each worker object has a unique remote adapter instance which serve as the point of contact on the server for that worker object. Remote adapters use reflection to invoke requested methods on the actual worker object instance. The remote adapter is also responsible for checkpointing the worker object state when it is running in safe mode, preparing the worker object for garbage collection when it is no longer referenced by any clients and performing migration when it is requested. These responsibilities are described in more detail in the following sections.

4.2 Worker Object Migration

Three types of migration are supported by Babylon v2.0. Idle migration takes an idle worker object (one that is not executing any methods) and moves it to a new server. If a worker object is actively executing one or more methods, delayed or safe-point migration can be used. Delayed migration prevents new method invocations from starting while allowing in-progress methods to finish executing. Once all the in-progress methods finish executing, the object becomes idle and migration can occur. Finally, safe-point migration uses a checkpointing and rollback protocol to perform migration at programmer specified safe migration points. In this form of migration, a checkpoint of the worker object, which was created when the worker was idle, is migrated to the destination server when execution reaches a safe migration point.

Babylon v2.0 supports worker object migration with the static `Babylon.migrate(Object workerProxy, boolean useSafePoint)` method. The parameters to this method are (i) the proxy object for the target worker object and (ii) a `boolean` flag which indicates whether or not safe-point migration is desired. Unless safe-point migration is specified, Babylon v2.0 will simply perform idle or delayed migration depending on whether or not the worker object is idle or executing.

Worker object migration works by taking a snapshot of a worker object's data state, known as a checkpoint, and transmitting this checkpoint to a new Babylon v2.0 server. Using the checkpoint, the destination server constructs a new worker object with the same data state as the original worker. Java's object serialization [48] mechanism is used to create a checkpoint of the state of a worker object. Consequently, only objects that are serializable can be migrated and the resulting checkpoints are also limited by any limitations of Java's object serialization. For instance, static and transient variables cannot be serialized and, consequently, their values would not be preserved across migrations.

However, Java object serialization does have limitations. Java security restrictions prevent Java programs from accessing the JVM's execution state. Consequently, elements of the execution state such as stack contents and program counters cannot be examined or serialized. Since many elements of the execution state are part of a running Java thread, threads cannot be serialized. This restriction seriously limits the effectiveness of safe-point migration because worker objects can only be checkpointed when they are idle. We next discuss the implications of this limitation in more detail.

4.2.1 Migration Issues

Although object migration was supported in Babylon v1.0, it was not optimized and the implementation was deficient in several areas. Firstly, six RMI calls were required to perform a single worker object migration (idle, delayed or immediate). RMI overhead is considerably higher than local method invocation overhead and, when used excessively, can become a bottleneck in a distributed application [21, 28]. Babylon v2.0 object migration has been redesigned to reduce the number of RMI calls required to migrate a worker object from one server to another. Instead of six, Babylon v2.0 requires only three RMI calls for the migration of a worker object. This has resulted in a performance improvement of almost 40% for the migration of small worker objects. An experimental performance evaluation of the costs of migration is conducted in Chapter 5.

In Babylon v1.0, a client can enable checkpointing for a worker object by calling the `Babylon.setCheckpoint(`*`workerId`*`, true)` method using the worker identifier of the target worker object. When this option is enabled, a checkpoint of the state of the target worker object is created prior to every method invocation. Once checkpointing has been enabled, a client can request immediate migration. When immediate migration is requested, the most recently started executing method is interrupted and the most recent worker object checkpoint is migrated to the destination server. The interrupted method is then transparently restarted on the worker object at the destination server with the object re-executing code that had been executed between the time of the last checkpoint and the actual migration.

However, several methods may be executing concurrently on the worker object when the checkpoint request is made. Unfortunately, Babylon v1.0 does not enforce single threaded execution of method invocations on a worker object when checkpointing is enabled. This can lead to the following problem. Consider the scenario in Figure 4.2 in which a worker object with checkpointing enabled is executing two method invocations, `m1()` and `m2()`, concurrently. Since checkpointing is enabled, a new checkpoint of the worker object is created prior to the start of each invocation. However, checkpoint `C2`, triggered by the start of method invocation `m2()`, is created while the worker object is already in the process of executing method `m1()`. This checkpoint may suffer from inconsistencies and may not be valid because the worker object was not idle when it was created.

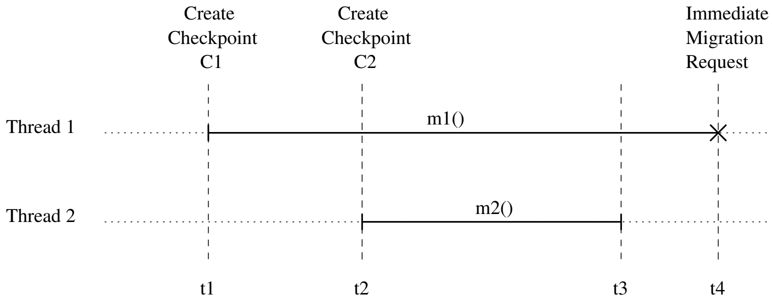

Figure 4.2: Immediate Migration in the Presence of Concurrent Method Invocations

There are also other factors that complicate immediate migration in the presence of multiple concurrently executing threads. These problems all stem from the fact that it is not possible to serialize the state of a running thread. For instance, consider the immediate migration request in Figure 4.2 at time t_4. At this point, method `m2()` has already completed (at time t_3) and the result of the invocation has been returned to the client that made the call. However, method `m1()` must be interrupted before immediate migration can proceed.

In this scenario, it may be tempting to simply ignore checkpoint `C2` and restart both method invocations (`m1()` and `m2()`) on the destination server using checkpoint `C1`. Eventually the new worker object instance on the destination server will reach the state it was in before migration was requested. However, the re-execution of method `m2()` on the destination server may result in a different outcome than on the original server if the method uses a non-deterministic variable such as a random number or the time of day for a computation, or if either method accesses a shared (static) variable. This could lead to an inconsistent system state because the client that made the initial invocation of method `m2()` has already obtained a result which will differ from the new result produced by the method. Thus, simply restarting concurrent method invocations on the worker object at the destination server can lead to a global inconsistent state.

There is also no mechanism in Babylon v1.0 to defend against the checkpoint consistency problem [15]. The checkpoint consistency problem can occur in rollback-recovery protocols when a worker object interacts with an external entity which cannot be rolled back. This can produce an inconsistent system state if the worker object is rolled back using a checkpoint that was created before the interaction with the external entity. After the rollback, the state of the external entity will still reflect the interaction but the state of the worker object will not.

Furthermore, as of JDK v1.2, several methods of the `java.lang.Thread` class have been deprecated because they are inherently unsafe and can potentially destroy the integrity of the thread model [47]. The list of deprecated methods includes the `stop()`, `suspend()` and `resume()` methods. The elimination of the `stop()` method leaves the Java language with no convenient way to stop an executing thread without some cooperation from the thread itself. Unfortunately, immediate migration in Babylon v1.0 relies on the `stop()` method of the `java.lang.Thread` class to interrupt executing methods.

We now discuss how Babylon v2.0 addresses the problems discussed above and propose a safe-point migration mechanism that enforces single threaded execution of methods on worker objects and defines special points in the worker object code, called *safe migration points*, where safe-point migration can be performed.

4.2.2 Safe Mode

Babylon v2.0 provides a new mode of execution for worker objects called safe mode. Safe mode addresses the consistency issues associated with migration in the presence of concurrent method invocations which were described in the previous section. Safe-point migration in Babylon v2.0 is only supported for objects running in safe mode. Safe mode for a particular worker object can be activated or deactivated with the static `Babylon.setSafeMode(Object workerProxy, boolean enable)` method. The `enable` flag indicates whether safe mode should be activated or deactivated. Several things occur when a worker object starts executing in safe mode. First, the worker object becomes single-threaded and only a single method invocation is allowed to be executing at any time. This element is crucial for correct safe-point migration behaviour and was overlooked in the design of Babylon v1.0 object migration.

Secondly, a checkpoint of the worker object state is created prior to the start of each method invocation. This ensures that the worker object always has a recent checkpoint which can be used if safe-point migration is requested. Nevertheless, it should be noted that the creation of a checkpoint of the worker object prior to each method invocation can incur significant overhead. Safe mode is therefore disabled by default and must be enabled by the programmer if safe-point migration of a worker object is desired.

4.2.3 Safe Migration Points

Safe migration points address two of the immediate migration issues defined in Section 4.2.1. First, they provide of a safe mechanism for stopping a running Java thread and, secondly, they provide a mechanism to programmatically avoid the checkpoint consistency problem.

At any point during the execution of a method on a worker object, the method may be executing a synchronized section of code that cannot be interrupted or may have just performed an interaction with an external entity that cannot be rolled back. In either case, interrupting the method and rolling back to a checkpoint is unsafe and may result in an inconsistent system state. Safe migration points are programmer-defined points in the worker object code where safe-point migration can be performed without resulting in an inconsistent system state. Programmers identify safe migration points in their worker object code with calls to the static `Babylon.setMigrationPoint()` method.

In Babylon v2.0, worker objects that need to be immediately migrated must include calls to the static `Babylon.setMigrationPoint()` at safe migration points in the source code of their worker objects. Normally, this method does nothing and simply returns. However, if safe-point migration is pending, this method will throw a `BabylonThreadDeath` object as an exception. The `BabylonThreadDeath` object propagates up the stack until it is caught by the remote adapter for this worker object. When the remote adapter catches the exception, it knows that the executing method has been stopped and that safe-point migration can be safely performed.

The `BabylonThreadDeath` exception is a runtime exception which does not need to be caught in the worker object code. However, the exception will still cause the `finally` blocks of any enclosing `try` statements to be executed. If the worker object needs to perform cleanup tasks or undo an operation on an external entity before safe-point migration is performed, the programmer can perform cleanup tasks in the finally clause or catch the exception explicitly and perform cleanup tasks at that point. The exception must be rethrown when the cleanup task completes so that the remote adapter is still aware that safe-point migration can be performed. The example in Figure 4.3 demonstrates how the code for a worker object could be annotated to enable safe-point migration.

```
 1 public class WorkerObject implements java.io.Serializable
 2 {
 3    public int m1(int iterations)
 4    {
 5       for (int i = 0; i < iterations; i++) {
 6          doSomeWork();
 7
 8          // Migration can safely occur here.
 9          Babylon.setMigrationPoint();
10       }
11
12       interactWithExternalEntity();
13
14       try {
15          for (int i = 0; i < iterations; i++) {
16             doSomeMoreWork();
17
18             // Migration can safely occur here but we have to undo the
19             // interaction with the external entity before it can happen.
20             Babylon.setMigrationPoint();
21          }
22       }
23       catch (BabylonThreadDeath threadDeath) {
24          // Undo the interaction with the external entity and rethrow exception.
25          undoExternalInteraction();
26          throw threadDeath;
27       }
28    }
29 }
```

Figure 4.3: Adding Safe Migration Points to Worker Object Code

In this example, method `m1()` of the worker object `WorkerObject` is annotated with two calls to `Babylon.setMigrationPoint()`. At the first safe migration point (line 9), safe-point migration can be performed without the need for any cleanup. If safe-point migration is pending when `Babylon.setMigrationPoint()` is called at line 9, a `BabylonThreadDeath` exception will be thrown. The exception will propagate up the stack until it is caught by the remote adapter for the worker object. Once the remote adapter catches the exception, it will migrate the checkpoint that was created before the invocation of `m1()` to a new server and restart the method at the new location.

The second safe migration point (line 20) is established after an interaction with an external entity that cannot be rolled back (line 12). If a migration request occurs at this time, the `BabylonThreadDeath` exception will be caught (line 23) so that the interaction can be undone before migration occurs. After performing the necessary steps to undo the interaction, the exception is rethrown (line 26) so that it propagates to the remote adapter which will proceed with the migration. It should be noted that example in Figure 4.3 is a very simple program for which safe migration points can be easily determined. However, determining the location of safe migration points in large applications with many interactions with external entities can be a non-trivial task. All the same, safe migration points are only required for for worker objects that require this form of migration. Other forms of migration such idle and delayed can still be performed without requiring safe migration points or safe mode.

Although defining safe migration points to worker object code requires some programmer intervention, Babylon v2.0 attempts to make this task as simple as possible for the programmer. Apart from the addition of occasional calls to `Babylon.setMigrationPoint()`, this mechanism requires no other intervention from the worker object programmer. Furthermore, our mechanism is 100% Java compatible and guarantees that migration is only performed at points in the worker object code that will not result in an inconsistent system state. Nevertheless, the drawback of this approach is that the source code of the worker object must be available.

4.2.4 Reference Updating

Babylon v1.0 uses a reference update strategy know as location forwarding [20]. Location forwarding maintains forwarding information at each of the worker object's former locations which identifies the destination of the migration for the worker object. This forwarding information essentially creates a chain leading to the current location of the worker object.

For instance, when a Babylon v1.0 client attempts to use a worker identifier with a stale worker object reference, the Babylon v1.0 server that used to host the worker object will throw a `babylon.core.MovedException` containing a reference to the worker object's next location. This exception is caught at the client side by the Babylon v1.0 framework which extracts the new location and updates the worker object reference. If the object moved more than once, subsequent use of the remote reference will yield a new `babylon.core.MovedException` containing a reference to the next location. Eventually, the latest remote reference will be obtained and the operation will proceed as requested. The problem with this technique is that it will not work if any of the intermediate servers become unreachable before all clients have a chance to update their references. An unreachable intermediate server would break the chain of forwarding locations and the reference to the object could not be retrieved.

Babylon v2.0 also uses location forwarding as the primary mechanism to update stale worker object references. However, Babylon v2.0 provides a backup mechanism to locate worker objects if any of the intermediate forwarding locations become unreachable. This functionality is provided with the help of the object registry implemented in the scheduler. The registry maintains a record of all the worker objects in the Babylon v2.0 system and is also used to provide worker object lookup functionality to clients.

When a worker object cannot be located using location forwarding, the worker object registry is used to determine the most recent worker object location. The failed operation is retried once the proxy's remote reference to the worker object has been updated using the reference obtained from the registry. This is all performed by the client adapter and is completely transparent to the client application. However, location forwarding is still used as the primary update method because of its inherent distributed nature.

4.3 Fault-Tolerance in Babylon v2.0

Unlike Babylon v1.0, Babylon v2.0 provides basic programmer-controlled fault-tolerance for worker objects using checkpointing. Clients can call the static `Babylon.getCheckpoint(Object workerProxy)` method to obtain a checkpoint of a worker object. The argument to this method is the proxy object of the worker object that should be checkpointed. The method returns a `babylon.core.Checkpoint` object which is the checkpointed copy of the worker object. If the worker object is not idle when a checkpoint is requested, the executing methods are given a chance to complete before a checkpoint of the worker object is created. However, no new methods on the worker object can be started once the creation of a checkpoint has been requested. New method invocations are queued and are only allowed to start once the checkpoint has been created and returned to the requesting client.

The `babylon.core.Checkpoint` class includes methods to read and write checkpoints to and from disk. A client can write a checkpoint to disk using the checkpoint's `writeToFile(String filename)` method. A checkpoint written to disk in this manner can be read from disk using the static

`Checkpoint.readFromFile(String filename)` method. In the event of a server failure, a previously saved checkpoint can be read from disk and exported to a server using the `Babylon.loadCheckpoint(Checkpoint cp)` method. This enables clients to mask server failures and continue executing from the point where the last checkpoint was made. However, the burden of requesting checkpoints and saving them to disk is currently placed on the programmer. A more transparent fault-tolerant framework is likely to be a future Babylon research project.

4.4 Class Loading in Babylon

A class loader is an object that is responsible for loading the classes required by a Java application [30, 31, 35, 55]. The JVM comes with two built-in class loaders that load class data from "class files" residing in the local filesystem. Applications can also create user-defined class loaders that load classes using a custom strategy.

Babylon v1.0 and v2.0 servers use such user-defined class loaders to dynamically load classes for worker objects whose class files do not reside in the local file system. These user-defined class loaders load worker classes from a codebase provided by the client. This section begins with an overview of Java's class loading semantics followed by a description of the extended Babylon v1.0 and v2.0 class loading mechanisms.

4.4.1 JVM Class Loading Semantics

The Java virtual machine contains two built-in class loaders: the bootstrap class loader and the system class loader. When asked to load a class, these class loaders typically convert the fully-qualified name of the class into a file name and attempt to locate and read the corresponding class file from the local filesystem. The bootstrap class loader loads the Java run-time and the classes of the standard Java API from pre-defined locations in the filesystem. The system class loader loads the application classes from the filesystem by searching the directories specified in the CLASSPATH environment variable. Applications that require different class loading behaviour can create user-defined class loaders that load classes from other locations. User-defined class loaders must extend the `java.lang.ClassLoader` class included in the standard Java API.

Class loaders are organized in a hierarchical fashion. The bootstrap class loader is at the root of the class loader hierarchy and is an ancestor of all other class loaders. Every user-defined class loader instance is assigned a parent class loader. The default parent for a user-defined class loader is the system class loader whose parent is the bootstrap class loader.

Classes are located using a delegation model. When the virtual machine encounters a symbolic reference to a new class, it calls the `loadClass(String className)` method of the class loader that loaded the referencing class. This class loader first checks its local cache of previously loaded classes for the requested class. If the requested class is found in the cache, the class loader simply returns the corresponding `java.lang.Class` object. If the class loader cannot find the requested class in its cache, it requests the class from its parent class loader before attempting to load the class from the filesystem or some other class source. The parent class loader follows the same procedure and propagates the request to its parent if it cannot find the requested class in its cache. A class loader will only attempt to load a class from its class source if its parent loader, and by extension all its ancestors, have failed to load the class. Failure to load a class is indicated by throwing a `java.lang.ClassNotFoundException`. This delegation model prevents classes from being loaded more than once.

To help clarify the class loading delegation model, Figure 4.4 illustrates a failed class loading attempt for a class called `MyWorker`. In this example, the virtual machine initiates the class loading delegation by invoking the `loadClass(String className)` method of a user-defined class loader called `RemoteClassLoader`. The requested class is not available and thus the request propagates all the way to the bootstrap class loader and back down as each class loader fails to locate or load the class.

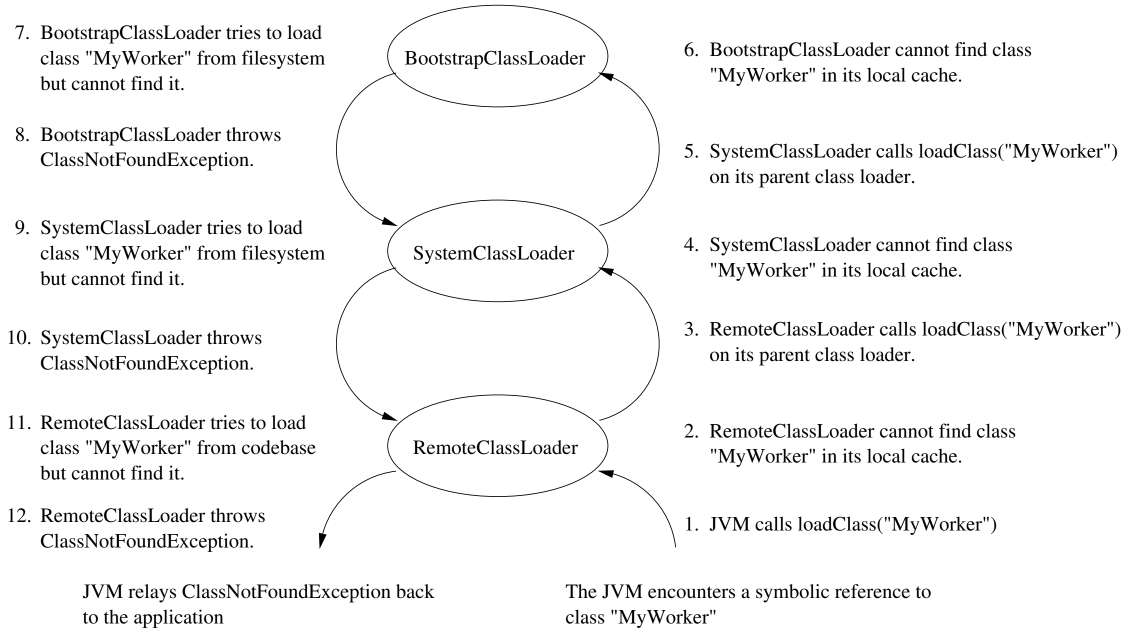

Figure 4.4: Failed Class Loading Attempt

The class loader that loads a particular class is referred to as the defining loader for that class. Applications can retrieve the defining loader of a class by calling the `getClassLoader()` method on a `java.lang.Class` instance corresponding to the given class. The virtual machine uses the defining loader of a given class to load new classes referenced by that class. For example, if the virtual machine encounters a symbolic reference to a class, say `MyWorker`, while executing a method of another class, say `MainClass`, then the virtual machine will try to load the referenced class, `MyWorker`, from the class loader that loaded the referencing class, `MainClass`.

An implication of this class loading delegation model is namespace separation at the class loader level. This namespace separation allows the existence of different definitions for a class of the same name as long as each has been loaded by a different class loader. Applet class loading provides a good example of this namespace separation because applets from different sources are normally loaded by different class loaders. As a result, two different applets may use classes with identical class names but since they reside in separate namespaces they will not interfere with each other.

Although both versions of Babylon use user-defined class loaders for worker object class loading, the implementations and loading mechanics differ significantly. These differences are described in the following two sections.

4.4.2 Class Loading in Babylon v1.0

At startup, a Babylon v1.0 server has no knowledge of the worker objects that it will be required to host and is thus unable to load the worker classes using the system class loader. Babylon v1.0 servers use a custom class loader (which is part of Babylon), called `babylon.core.RemoteClassLoader`, to load these classes. In order to instantiate client-defined worker objects, a `babylon.core.RemoteClassLoader` instance loads the worker's classes dynamically from the supplied codebase. This codebase is automatically transferred to the Babylon v1.0 server when a client creates a new remote object.

In Babylon v1.0, the `babylon.core.RObject` class plays a similar role to the `babylon.core.RemoteAdapter` class in Babylon v2.0. Instances of the `RObject` class act as remote adapters for worker objects in Babylon v1.0. A Babylon v1.0 worker object can only be accessed using the associated `RObject` instance. Since the `RObject` class is a standard part of Babylon v1.0 it is loaded automatically at server startup by the system class loader.

Babylon v1.0 clients use the static `Babylon.rmi()` method to invoke methods on worker objects. Any arguments for the target method are passed as arguments to the `Babylon.rmi()` method. These arguments, along with the method name, are passed to the target worker object's `RObject` instance using a remote method invocation. At the server side, the RMI runtime uses the class loader that defined the `RObject` instance to unmarshal any method arguments. Recall that the class loader that defined `RObject` is actually the system class loader which has no knowledge of client-defined classes. Because of this namespace mismatch, the unmarshaling fails if any of the target method parameters are classes defined by the client.

This problem would disappear if we could somehow force the `babylon.core.RObject` class to be loaded by the `babylon.core.RemoteClassLoader` instead of the system class loader. Babylon v1.0 accomplishes this using three steps:

1. The `babylon.core.RObject` class is subclassed by a dummy class, called `babylon.core.R2Object`, which inherits all the functionality of `babylon.core.RObject`.

2. The name of the `babylon.core.R2Object` class file is changed from `R2Object.class` to `R2Object.clas` so that the system class loader cannot find it.

3. The `babylon.core.RemoteClassLoader` is retrofitted with the functionality to load classes whose class file names end in `.clas` so that it can load the `babylon.core.R2Object` class file from the filesystem.

When a Babylon v1.0 server uses `babylon.core.RemoteClassLoader` to load the `babylon.core.R2Object` class, the class will be loaded by the `babylon.core.RemoteClassLoader` because none of the parent class loaders will be able to find it. As a result, the RMI run-time will unmarshall client-defined arguments at the Babylon v1.0 server using the appropriate class loader.

However, Babylon v1.0 servers use a single instance of `RemoteClassLoader` to load classes for every client. This class loading strategy is adequate for simple single-client cases but introduces serious garbage collection, naming, and security issues in more complicated scenarios. For instance:

- Garbage Collection

  A Babylon v1.0 server loads the classes of every client that it services using the same `babylon.core.RemoteClassLoader` instance. The definition of each loaded class remains in memory even after the client that used it exits. This occurs because classes are only garbage collected when the class loader that defined them is garbage collected. However, Babylon v1.0 servers use a single class loader instance which is never garbage collected.

- Naming

Since there is only one `babylon.core.RemoteClassLoader` instance per Babylon v1.0 server, different clients cannot use classes that have the same name. This behaviour becomes especially problematic when a single client creates a worker object on a server, quits, changes the worker object code and then tries to re-create that object on the same server. The Babylon v1.0 server will not load the new worker object class definition because it has already loaded and will continue to use the original definition of that class. As a result, Babylon v1.0 servers must be restarted every time clients change their classes.

- Security

  All clients share a single namespace on a Babylon v1.0 server and, consequently, clients have access to the classes loaded by other clients. This can be a security problem because one client could potentially use another client's classes in malicious ways. For instance, a client could modify a static variable of a class which is being used by another client.

4.4.3 Class Loading in Babylon v2.0

The key difference between Babylon v1.0 and Babylon v2.0 class loading is that Babylon v2.0 servers no longer use a single class loader instance per server. Babylon v2.0 servers create a new instance of a custom class loader for each client. When a client connects to a server for the first time, the server creates a new instance of `babylon.server.ClientClassLoader`. This class loader will be in charge of loading all the classes that are needed by the server to import worker objects for the given client. The server saves a reference to the class loader instance so that it can be retrieved and reused if the same client tries to create another worker object on the same server.

Each `ClientClassLoader` manages its own namespace. A `ClientClassLoader` cannot see beyond the boundaries of it's own namespace. It only knows about the classes it loaded and the classes loaded by the class loaders in its delegation chain. This shields and protects applications from each other's classes and circumvents naming conflicts between identically named classes belonging to different clients. Separate class namespaces also protect a client's classes from malicious use by another client. Furthermore, clients can now change their worker object classes and simply restart their application to load and use the new class definitions. Finally, because each application has its own instance of `ClientClassLoader`, a class loader and the classes loaded by it can be garbage collected when the associated client application exits and the class loader is no longer used.

Java 2 also introduced the notion of a context class loader. Context class loaders were added to get around the limitations of the parent-child class loading mechanism of Java 1.1. Context class loaders make it possible to load new classes using a different class loader than the one that defined the loading class. Every thread in a Java 2 JVM has an associated context class loader which is initially set to the context class loader of the parent thread. However, the context class loader of a thread can be set to a different class loader using the static `Thread.setContextClassLoader()` method.

Java RMI uses context class loaders to load the classes of arguments passed to remote methods. When Babylon v2.0 creates a remote adapter for a worker object, it sets the context class loader of the RMI thread that listens for incoming RMI connections for that remote adapter to be the `ClientClassLoader` instance that loaded the corresponding worker object. When a client adapter invokes a method on a remote adapter to request an operation, the RMI run-time will use the context class loader to load the classes for any arguments passed to the method. This class loader knows about the client's classes and will be able to properly load them. As a result, Babylon v2.0 no longer needs to use class name mangling to force worker object classes to be loaded by a particular class loader.

4.5 Summary

Significant modifications in the design and implementation of Babylon have been made to accommodate the new transparent method invocation mechanisms for worker objects, improved worker object migration techniques and a redesigned class loading architecture. When compared with v1.0, the changes have resulted in significant improvements in both the interface and functionality.

In the next chapter we explore some of the performance issues related to the new implementation. Our results show that Babylon v2.0 offers these benefits with little or no impact on overall performance.

Chapter 5

System Evaluation

This chapter presents the results of experiments designed to evaluate the performance of Babylon v2.0. Microbenchmarks were used to measure the performance of basic Babylon v2.0 operations and application benchmarks were used to test overall Babylon v2.0 performance using larger and more realistic applications. Where possible, the experiments were performed using both the new and old versions of Babylon. The results show that despite many changes and significant new features in Babylon v2.0, performance is as good or better than Babylon v1.0. The chapter concludes with a brief discussion of the experimental results.

Before presenting the experimental results, we provide a brief discussion of some of the variables that can introduce performance variations in Java test environments and the methods that were used to address and reduce these variations.

5.1 Reducing Performance Variance during Benchmarking

Producing consistent and reproducible performance measurements for complex Java applications can be a difficult and challenging task. Non-deterministic activities such as garbage collection and just-in-time compilation can introduce performance variations between otherwise identical program executions. This is especially true in an environment like Babylon, where a single JVM instance is used to repeatedly execute a number of applications.

To eliminate inconsistencies that could be introduced by variable network load, all of our experiments are conducted using a controlled and isolated test network. The test network is on a separate subnet and is isolated from the rest of the university network to prevent interference from other traffic.

The performance variations introduced by just-in-time compilation are more difficult to control. Fortunately, these variations tend to disappear after the application has been running for a period of time [44]. The application eventually reaches a relatively steady state in which the more frequently used classes have all been loaded and just-in-time compilation has had a chance to compile the application “hot spots” to native code. A steady Babylon server state was achieved for the experiments in this chapter by running the test applications a number of times initially until performance variations were reduced to a minimum.

Unfortunately, garbage collection can still produce performance variations even after an application has reached a steady state. In most conventional single processor applications, a new JVM is started each time the application runs. Consequently, the initial conditions are roughly equivalent and garbage collection occurs at approximately the same point during each execution. Babylon servers, on the other hand, have normally been running for a period of time before an experiment is started in order to give each server time to reach a steady state. As a result, the initial Babylon server state is not necessarily identical between successive executions of our experiments and garbage collection may not always occur at predictable points. The presence of such sporadic collections can produce deviations in the resulting performance data.

To produce a more consistent initial state, a garbage collection is requested before each experiment by calling `System.gc()` on all servers participating in the experiment. `System.gc()` suggests to the JVM that a major garbage collection may be necessary. A major garbage collection should reclaim any memory that may still be in use by discarded objects from previous experiments. In our experiments, we observed that a major garbage collection was indeed triggered every time this method was called. Using command-line arguments to the JVM, the Babylon server's JVM was also tuned to minimize collection frequency during the experiments. The command-line arguments that were used to tune the JVM are described in detail in Appendix E. In addition to the above techniques, performance variations for the heat diffusion experiment in Section 5.4.2 were further reduced by doing ten consecutive executions and choosing the fastest one (this was done to compare the speedups obtained using different numbers of processors).

5.2 Testbed

The experiments in this chapter are conducted using a cluster of 8 dual CPU (500 MHz Pentium III) PCs running Red Hat Linux 7.1 2.96-98 connected via a 100 Mbps switch. Each of these machines contains 256 MB of RAM and is running the J2SE platform, version 1.3.1_02.

Unless otherwise noted, experiments are launched from a separate dual CPU (1 GHz Pentium III) PC called `grand` with 2 GB of RAM running Red Hat Linux 8.0 3.2-7 also connected to the 100 MB switch. This machine runs the J2SE platform, version 1.4.1_01.

5.3 Micro Benchmarks

Microbenchmarks are used to evaluate the performance of remote method invocations and worker object migration in Babylon v2.0. The experiments in this section are largely reproductions of experiments performed by Matthew Izatt in his thesis on Babylon v1.0. The original test applications have been rewritten to work with Babylon v2.0 and executed on the new test network using a wider range of test parameters than were possible in the original experiments. The results demonstrate that despite significant design changes and the addition of several new features, the performance of the tested operations is still equal to or better than their Babylon v1.0 counterparts.

5.3.1 Remote Method Invocation

During the implementation of Babylon v2.0, a significant rewrite of remote method invocation was required in order to add support for primitive-type remote method parameters and worker object access restrictions. Nevertheless, speed and efficiency remained a key requirement and every effort was made to minimize overall remote method invocation overhead and to optimize the transmission of invocation context information.

Table 5.1 presents the remote method invocation performance measurements using Babylon v2.0, Babylon v1.0 and the standard RMI facilities provided by the JDK. In each experiment, a client measures the total time it takes to make 10,000 remote method invocations on a worker object. The only argument to the invoked method is an object containing an integer array of a specified size (a size of "1" represents an array containing one `int`, "1k" represents an array containing 1,000 `ints`, "10k" represents an array containing

10,000 `ints`, etc) and the return value is an integer indicating the size of the array. The total time (call and reply) for 10,000 invocations is used to compute and report the average time for a single remote method invocation. All times are reported in milliseconds. It should be noted that remote method arguments must be serializable because Babylon uses Java serialization to transmit the arguments to worker objects.

Application	Argument Size (integers)					
	1	*1k*	*10k*	*100k*	*1M*	*10M*
Babylon v2.0 RMI (ms)	2.16	2.55	5.50	38.28	364.84	3693.10
Babylon v1.0 RMI (ms)	2.26	2.65	5.58	39.49	369.04	3717.17
JDK RMI (ms)	1.00	1.34	4.42	36.35	363.15	3677.30

Table 5.1: Remote Method Invocation Performance (ms)

The results in Table 5.1 demonstrate that Babylon v2.0 performance is approximately equivalent to Babylon v1.0 performance in all cases. The difference between RMI and Babylon v2.0 performance can be attributed to the additional layers of indirection present in the Babylon framework. However, the results also demonstrate that the performance difference becomes insignificant relative to the total invocation time for larger arguments. For instance, with a 100k integer argument, a Babylon v2.0 invocation is only 1.05 times slower than raw RMI and this value drops to 1.004 for a 10M integer argument. In a larger and more realistic Babylon v2.0 application, this performance difference will have very little, if any, impact on overall application performance. See Section 5.5 for a discussion of Java RMI versus Babylon v2.0 application performance.

5.3.2 Worker Object Migration

Another key Babylon feature is worker object migration. Idle migration performance is tested by exporting a worker object of a particular size and then migrating it to a different server seven times. Each of the eight Babylon servers used in this experiment is running on a different machine on the test network and each must load the client codebase (approx. 3 KB) before it can receive the migrated worker object. The total time for seven successive idle migrations is measured and used to compute the average time for a single migration.[1] Each migration is requested using the `Babylon.migrate()` method and runs to completion (i.e., the `migrate()` method returns) before the next migration is requested. This process is repeated seven times and the total time from the first request to the last successfully completed migration is recorded.

The migration performance results for Babylon v2.0 and v1.0 are presented in Table 5.2. The migrated object is a simple worker object whose only data state is a Java integer array of a specified size (as was done for the previous RMI experiments). All times are reported in milliseconds. The table also shows an RMI comparison factor. The comparison factor measures the ratio of the time required for a worker object migration to the time required to perform a remote method invocation using an argument of the same size as the migrated worker object (see Table 5.1 for remote method invocation benchmark results).

Although object migration was supported in Babylon v1.0, the implementation was not very efficient and required six RMI calls to perform a single worker object migration. The implementation of migration in Babylon v2.0 has been redesigned and the number of required RMI calls has been reduced from six to three.

[1] Classloading issues in Babylon v1.0 prevented us from running the experiment with more than seven subsequent migrations because Babylon v1.0 servers cannot receive an object of the same name more than once. Although these class loading issues have been resolved in Babylon v2.0, seven migrations were used in all experiments for consistency.

Application	Object Size (integers)				
	1	*1k*	*100k*	*1M*	*10M*
Babylon v2.0 Migration (ms)	76	79	106	423	3754
RMI Comparison Factor	35.19	30.98	2.77	1.15	1.02
Babylon v1.0 Migration (ms)	128	130	177	506	3954
RMI Comparison Factor	56.63	49.06	4.48	1.37	1.06

Table 5.2: Worker Object Migration Performance (ms)

This has resulted in significant performance improvements for small worker objects. For instance, the time required for the migration of a worker object containing 1000 integers (*1k*) has been reduced from 130 ms in Babylon v1.0 to 79 ms in Babylon v2.0, an improvement of almost 40%.

Nevertheless, worker object migration is still significantly slower than the time required to perform a remote method invocation using an object of the same size as an argument. To better understand the high inflation factor for small worker objects, one needs to look at the extra overhead associated with worker object migration. There are several important differences between a remote method invocation and the migration of a worker object. First, migration involves much more than just the transmission of the worker object to the target server. Before the migrated worker object can be received, the targer server must obtain the codebase from the original server and load it. Furthermore, after the worker object has been successfully instantiated at the target server, the worker instance on the original server must be marked as migrated so that it can be garbage collected when the next garbage collection occurs.

These steps, combined with the transmission costs for the codebase, control context and the actual worker object, all contribute to a high inflation factor for small worker objects. However, this overhead drops significantly for larger worker objects (i.e., 100k integers) and is relatively small for worker objects larger than 1M integers. At this point, the codebase transmission and setup costs are almost completely overshadowed by the cost of transmitting the actual worker object.

5.4 Application Benchmarks

Although microbenchmarks are useful for measuring the performance of specific Babylon v2.0 operations, they are not necessarily good at predicting overall application performance. Consequently, we have conducted several experiments using larger and more realistic applications. Note that each test application still focuses on a particular Babylon v2.0 feature and the resulting measurements provide valuable insights into both the performance of the feature under consideration as well as overall system performance.

The experiments in this section have been conducted on a cluster of dual CPU workstations as described in Section 5.2. The simplest configuration for distributing the worker objects to the workstations is to start a single Babylon v2.0 server on each machine and run a single worker object on each of those servers. However, this type of configuration does not take full advantage of both of the CPUs present on each workstation.

Therefore, in addition to the "one worker object per Babylon server per test machine" configuration described above, we have also experimented with other configurations that make better use of all available CPUs. For example, one such configuration runs two worker objects in each Babylon server. In this scenario, Java threads are used for parallelization on each dual CPU system. Other configurations run two Babylon servers on each dual CPU machine. In this case, each Babylon server (and by extension, each worker object) is running in a separate JVM and thus can be scheduled on different CPUs by the operating system.

These experiments take advantage of both of the CPUs in the systems in our test environment and have enabled us to run parallel test applications with 16 worker objects using only eight machines. Where possible, the experiments include measurements for different worker/server/machine configurations and provide some insight into how well a Java application can be parallelised using dual CPU systems running Linux and which configurations provide the best results.

We use a *[worker/server/machine]* labelling convention to describe the different configurations in our experiments. The *worker* value denotes the total number of worker objects participating in the computation and, consequently, also denotes the degree of parallelism. The *server* and *machine* values denote, respectively, the number of Babylon servers and the number of test machines participating in the computation. If the *worker* and *server* values are equal, the experiment was performed using one worker object per Babylon server. If the *server* value is two times the *worker* value, the experiment was performed using two worker objects per Babylon server. The same rule applies to the *server* to *machine* ratio.

For example, *[16w/16s/8m]* describes a configuration with 16 worker objects each running on a separate Babylon server (16 Babylon servers in total) running on 8 machines. Therefore, there are two Babylon servers per test machine. On the other hand, *[16w/8s/8m]* describes a configuration with 16 worker objects running on 8 Babylon servers using 8 machines. In this case, two worker objects will be executing in each Babylon server but only one Babylon server will be running on each machine.

5.4.1 Matrix Multiplication

In this section, we consider a distributed implementation of matrix multiplication. Matrix multiplication is often used as a test application for distributed systems because it can be implemented using the master-worker design pattern. In other words, a matrix multiplication problem can be divided into sub problems that can be solved independently by worker objects. The participating workers do not need to communicate with each other to synchronize or share data.

Table 5.3 and Table 5.4 present the results of a distributed implementation of a matrix multiplication benchmark using Babylon v1.0 and Babylon v2.0, respectively. In this experiment two $M \times M$ matrices, A and B, are multiplied together using N worker objects. We use a block decomposition strategy to partition matrix A into N blocks each consisting of $\frac{M}{N}$ contiguous rows. Each block, along with a copy of matrix B, is transmitted over the network to a worker object running on a remote Babylon server. Each worker multiplies the provided matrices and returns the result back to the master program where they are combined to form the final result matrix.[2]

The sequential and distributed matrix multiplication implementations were timed for progressively larger square matrices. The sequential and distributed implementation use the same core matrix multiplication algorithm. The total execution time as well as the corresponding speedup are reported in Table 5.3 and Table 5.4 for each configuration and matrix size. The first column for each size denotes the speedup while the second column shows the total execution time. Measured times include the transmission of the matrices to the workers and the transmission of the results back to the master program. All times are reported in milliseconds. The speedup, S, for N worker objects is computed as the ratio of the time taken by the parallel Babylon implementation, $T_{par}(N)$, to the time taken by the sequential implementation of the algorithm, T_{seq}:

[2]This implementation of matrix multiplication has been used for consistency with Matthew Izatt's Babylon v1.0 experiments. However, it should be noted that this algorithm incurs relatively high data distribution costs and is not the most efficient algorithm available [17].

$$S(N) = \frac{T_{par}(N)}{T_{seq}}$$

It should be noted that a similar matrix multiplication experiment was performed by Matthew Izatt in his evaluation of Babylon v1.0. However, his experiments were only conducted using a limited number of configurations and did not include a detailed result analysis. This was largely because Babylon v1.0 only supports configurations involving a single worker per Babylon server per test machine. It should be noted that this limitation was another important motivating factor for the redesign and implementation of Babylon v2.0. The results obtained using Babylon v1.0 are shown in Table 5.3. The complete source for the matrix multiplication test application is given in Appendix D.

Configuration [workers/servers/machines]	Matrix Size (integers)					
	512x512		*1024x1024*		*2048x2048*	
	$S(N)$	*Time (ms)*	$S(N)$	*Time (ms)*	$S(N)$	*Time (ms)*
Sequential Java	1.0	16,554	1.0	138,268	1.0	1,216,991
Babylon v1.0 [1w/1s/1m]	1.0	17,215	1.0	140,274	1.0	1,229,343
Babylon v2.0 [1w/1s/1m]	1.0	17,155	1.0	140,203	1.0	1,226,433
Babylon v1.0 [2w/2s/2m]	1.9	8,758	1.9	71,232	1.9	626,859
Babylon v2.0 [2w/2s/2m]	1.9	8,748	1.9	71,092	2.0	622,879
Babylon v1.0 [4w/4s/4m]	3.4	4,933	3.7	37,027	3.8	321,666
Babylon v2.0 [4w/4s/4m]	3.5	4,702	3.7	37,060	3.8	321,984
Babylon v1.0 [8w/8s/8m]	5.4	3,084	6.6	20,865	6.9	176,070
Babylon v2.0 [8w/8s/8m]	5.6	2,982	6.6	20,841	7.1	170,807

Table 5.3: Babylon v1.0 and v2.0 Matrix Multiplication Speedup and Execution Time (ms)

Configuration [workers/servers/machines]	Matrix Size (integers)					
	512x512		*1024x1024*		*2048x2048*	
	$S(N)$	*Time (ms)*	$S(N)$	*Time (ms)*	$S(N)$	*Time (ms)*
Sequential Java	1.0	16,554	1.0	138,268	1.0	1,216,991
Babylon v2.0 [1w/1s/1m]	1.0	17,155	1.0	140,203	1.0	1,226,433
Babylon v2.0 [2w/2s/2m]	1.9	8,748	1.9	71,092	2.0	622,879
Babylon v2.0 [2w/2s/1m]	1.8	8,966	1.9	72,552	1.9	643,459
Babylon v2.0 [2w/1s/1m]	1.8	9,292	1.8	75,036	1.9	651,570
Babylon v2.0 [4w/4s/4m]	3.5	4,702	3.7	37,060	3.8	321,984
Babylon v2.0 [4w/4s/2m]	3.4	4,822	3.7	37,827	3.7	328,000
Babylon v2.0 [4w/2s/2m]	3.4	4,890	3.6	38,079	3.5	346,263
Babylon v2.0 [8w/8s/8m]	5.6	2,982	6.6	20,841	7.1	170,807
Babylon v2.0 [8w/8s/4m]	5.4	3,054	6.5	21,225	6.9	177,144
Babylon v2.0 [8w/4s/4m]	5.4	3,039	6.5	21,306	6.5	186,236
Babylon v2.0 [16w/16s/8m]	6.2	2,690	9.2	15,062	11.5	105,655
Babylon v2.0 [16w/8s/8m]	6.3	2,626	9.0	15,349	10.8	112,183

Table 5.4: Babylon v2.0 Matrix Multiplication Speedup and Execution Time (ms)

After comparing the results in Table 5.3 and 5.4, one sees that Babylon v2.0 performs at least as well as Babylon v1.0 in all supported configurations. It is also interesting to note that moving from a sequential implementation of matrix multiplication to a distributed Babylon implementation using a single worker object introduces very little overhead (<4%) despite the added data distribution costs. For instance, the sequential matrix multiplication implementation takes 1,216,991 ms to multiply two 2048x2048 matrices. When the same multiplication is done using a single Babylon v2.0 worker object, the time required is 1,226,433 ms, an increase of only 9,442 ms (0.78%). This is a relatively small increase when one considers that in the Babylon v2.0 implementation, all the matrix data (48 MB in total) must be serialized and transmitted over the network between the master and the worker, which is not required in the sequential implementation.

Table 5.4 shows that when using Babylon v2.0 for a fixed number of workers, configurations that position one worker on each dual CPU machine (i.e, *[8w/8s/8m]*) perform slightly better than configurations that assign two workers to each machine, be it two workers per Babylon server (i.e., *[8w/4s/4m]*) or two Babylon servers per machine (i.e., *[8w/8s/4m]*). One factor contributing to the lower speedups in configurations that position two workers on each machine (i.e., *[8w/4s/4m]* and *[8/8/4]*) is that the machines in these configurations receive twice as much matrix distribution data as machines in configurations that position one worker object on each machine (i.e., *[8w/8s/8m]*). Furthermore, in configurations with two workers on each dual CPU machine, background JVM threads that could otherwise be run on the free CPU (such as the JIT compilation thread, the finalizer thread, the signal dispatcher thread and several RMI related threads), must now interrupt a worker object to run. Despite the slightly lower speedups in these configurations, the impact is relatively small suggesting that applications should always assign worker objects to every available CPU.

Another pattern that emerges from Table 5.4 and pertains to configurations that position two worker objects on each dual CPU machine is that it is slightly more efficient to run each of two worker objects in their own Babylon server and, consequently, two separate JVMs (i.e., *[8w/8s/4m]*) than it is to run the two worker objects in a single server using a single JVM (i.e., *[8w/4s/4m]*). This is likely due to differences in the way the operating system schedules two worker threads within a single Babylon server (two Java threads within a single JVM process) as opposed to the way it schedules two separate Babylon servers (two separate JVM processes). The results indicate that the latter is slightly more efficient.

Overall, the speedups obtained for the 2048x2048 matrix multiplication experiments are quite good until 16 worker objects are used. With a 2048x2048 matrix, the grain size can be kept large enough to produce significant speedups with four or eight workers. For instance, using a configuration with four workers (*[4w/4s/4m]*) we were able to achieve a speedup of 3.8 and with eight workers (*[8w/8s/8m]*), a speedup of 7.1. However, as the number of workers increases, so does the cost of distributing the matrix data to the workers. This overhead becomes especially evident in configurations that use 16 workers. For instance, the largest speedup we were able to obtain using 16 workers was 11.5 using the configuration *[16w/16s/8m]*.

To determine why the speedups are not better for 16 worker objects, we conducted a detailed analysis of data distribution costs incurred by the multiplication of two 2048x2048 matrices. These results are presented in Table 5.5. The communication costs of distributing the matrix data to the worker objects increases proportionally with N. This is primarily because matrix B must be transmitted in its entirety to each worker object participating in the computation. For example, if 16 workers are participating in the computation, the master program will need to send all of matrix B and a portion of matrix A a total of 16 times at the start of the computation so that each worker has the required data. For large matrices, this can be a significant amount of data and may have a considerable impact on the resulting speedup.

The "Data" column in Table 5.5 provides an estimate of the total amount of data distributed in each configuration (this amount includes the distribution of matrix A and B to the workers, and the return of the

Configuration	*Total Time (ms)*	*Actual Speedup*	*Data (MB)*	*Transmission Time*		*Parallel Time (ms)*	*Parallel Speedup*	*Serial Fraction*	*Speedup Bound*
				(ms)	*(% of Total)*				
Sequential	1,216,991	1.0	0	0	0.0%	1,216,991	1.0	0.000	1.0
[1w/1s/1m]	1,226,433	1.0	48.0	4,843	0.4%	1,221,590	1.0	0.004	1.0
[2w/2s/2m]	622,879	2.0	64.0	5,906	0.9%	616,973	2.0	0.005	2.0
[2w/2s/1m]	643,459	1.9	64.0	5,891	0.9%	637,568	1.9	0.005	2.0
[2w/1s/1m]	651,570	1.9	64.0	5,816	0.9%	645,754	1.9	0.005	2.0
[4w/4s/4m]	321,984	3.8	96.0	8,666	2.7%	313,318	3.9	0.007	3.9
[4w/4s/2m]	328,000	3.7	96.0	8,696	2.7%	319,304	3.8	0.007	3.9
[4w/2s/2m]	346,263	3.5	96.0	8,688	2.5%	337,575	3.6	0.007	3.9
[8w/8s/8m]	170,807	7.1	160.0	14,237	8.3%	156,570	7.8	0.012	7.4
[8w/8s/4m]	177,144	6.9	160.0	14,140	8.0%	163,004	7.5	0.012	7.4
[8w/4s/4m]	186,236	6.5	160.0	13,960	7.5%	172,276	7.1	0.011	7.4
[16w/16s/8m]	105,655	11.5	288.0	24,649	23.3%	81,006	15.0	0.020	12.3
[16w/8s/8m]	112,183	10.8	288.0	24,978	22.3%	87,205	14.0	0.021	12.2

Table 5.5: Babylon v2.0 Matrix Multiplication Result Analysis (2048x2048)

result matrix, C, back to the master).[3] For example, the multiplication of two 2048x2048 integer matrices using two remote worker objects requires a total of approximately 64 MB of transmitted data. This includes the distribution of the matrices to the worker objects and returning the result to the master program. In contrast, multiplying the same matrices using 16 worker objects requires 288 MB of transmitted data. The process of distributing this large amount of data from the master to the worker objects causes the algorithm efficiency to drop significantly for configurations using 16 workers.

To further explore the impact of data distribution on matrix multiplication performance, we measured the raw matrix data transmission times for each configuration by calling a dummy matrix multiplication routine on each worker object which simply returned the matrix to be multiplied. These results are presented in the "Transmission Time" column in Table 5.5 by showing both the total transmission time and the transmission time as a percentage of the total run time. By subtracting the transmission time from the total run time, we were able to produce an estimate of the parallel execution time (time spent on actual matrix multiplication) of the computation. The parallel time estimate was used to compute a new speedup value which should, in theory, be almost linear provided there are no other sequential components to the algorithm. The parallel time and corresponding speedup results are displayed in the "Parallel Time" and "Parallel Speedup" columns, respectively.

The data transmission results demonstrate that for configurations with 16 worker objects, almost 25% of the total run time is spent transmitting data. This is a very large portion of the computation and explains the large drop in efficiency observed in these configurations. Furthermore, when we compute speedup values using only the parallel portion of the execution, it becomes clear that transmission costs account for almost all of the observed speedup loss. For instance, for the configuration *[16w/16s/8m]* with 16 workers, the speedup of the parallel portion is 15.0, up from only 11.5. For the configuration *[8w/8s/8m]* with 8 workers, the speedup of the parallel portion is 7.8, up from 7.1.

The data distribution times computed for various configurations also agree with the remote method invo-

[3]The total amount of network traffic generated with N worker objects for a b byte matrix (b = 16.0 MB for a 2048x2048 matrix) was computed according to the following formula: $(A + B + C) \times N$, where $A = b/N$, $B = b$ and $C = b/N$. This amount does not include any serialization overhead.

cation benchmark times reported in Table 5.1. For instance, the remote method invocation results demonstrate that it takes approximately 3,693 ms to transmit a 38 MB (10M integer) argument. This suggests that it should take approximately 4,647 ms to transmit a 48 MB argument using Babylon v2.0. The experimentally obtained transmission time, presented in Table 5.5, indicates that it takes 4,843 ms to transmit 48.0 MB of matrix data for configuration *[1w/1s/1m]*, which is very close to the estimated time based on the remote method invocation benchmark results. The same holds true for the other tested configuration. This suggests that the remote method invocation results reported in Table 5.1 could be used to predict, with reasonable accuracy, the transmission times for the distribution of other amounts of matrix data needed for configurations that were not actually tested.

The data transmission times also enable us to compute an estimate of the fraction of our computation that will be executed sequentially. This fraction, f, is computed for each configuration by dividing the matrix transmission time for that configuration by the total sequential computation time. The results are displayed in the "Serial Fraction" column of Table 5.5. Based on f, we compute an upper bound on the possible speedup, S_{Amdahl}, using Amdahl's Law [58]:

$$S_{Amdahl} = \frac{1}{(f + \frac{1-f}{N})}$$

The maximum possible speedup for each configuration is displayed in the "Speedup Bound" column of Table 5.5. The theoretical speedup values computed using Amdahl's Law are in close agreement with the speedup values obtained in our experiments. For instance, the maximum possible speedup using eight workers was found to be 7.4. This is very close to our speedup of 7.1 obtained in configuration *[8w/8s/8m]*. Similarly, the maximum speedup using 16 workers is 12.3 whereas we obtained 11.5 in configuration *[16w/16s/8m]*. This interpretation of the results also supports the claim that matrix distribution costs are the principal factor limiting speedup in our implementation.

Amdahl's law can also be used to predict the maximum possible speedup for configurations that have not been tested. First, the amount of matrix data transmitted for a particular configuration must be computed based on the matrix size and the number of workers. The remote invocation results from Table 5.1 can then be used to compute the transmission time for distributing the specified amount of data. To predict transmission times we use a transmission rate of 10.33 MB/s based on the 10M column of Table 5.1.

Consider a configuration where 32 worker objects are used to solve a 2048x2048 matrix multiplication computation. In this configuration, 544.0 MB of matrix data will need to be distributed to the worker objects. This will take approximately 52,662 ms based on the remote method invocation results in Table 5.1. Using Amdahl's law, we can predict that the maximum possible speedup under these conditions is 13.7. In other words, doubling the number of workers from 16 to 32 would only increase the speedup from 11.5 to 13.7 under ideal conditions. This is a very small performance gain for such a substantial increase in the number of worker objects. In fact, when the speedup is computed for successive numbers of worker objects, it can be shown that the best possible speedup for this implementation of matrix multiplication is 13.8 using 28 worker objects. This is because the communication overhead increases with the number of participating workers and when more than 28 worker objects are used, the communication overhead begins to exceed the benefits of parallelization.

5.4.2 Heat Diffusion

The heat diffusion benchmark simulates heat transfer across a two-dimensional surface over time. In this simulation, the surface is discretized into an $M \times M$ two-dimensional array, T, and the Jacobi iterative method

[25] is used to compute the final temperature distribution of the surface. The array is initialized to an even temperature distribution and a constant heat source is applied to each edge of the array. Each iteration simulates heat diffusion across the surface over a small period of time. At each iteration, i, the value of every cell in the array is recomputed to be the average value of its four neighbouring cells:

$$T_i(m,n) = \frac{T_{i-1}(m,n-1) + T_{i-1}(m+1,n) + T_{i-1}(m,n+1) + T_{i-1}(m-1,n)}{4}$$

In our implementation, N worker objects are used to compute the temperature of a surface after 100 iterations. We use a block decomposition strategy to partition the original array into N blocks each consisting of $\frac{M}{N}$ contiguous rows. Each block is transmitted over the network to a worker object running on a remote Babylon server. Two block edges must be exchanged between neighbouring worker objects at each iteration. Once these data points have been exchanged, each worker can compute the updated temperature for all the cells in its block for the current iteration.

The distributed heat diffusion application is of interest because, unlike the matrix multiplication benchmark, it cannot be divided into sub tasks that can be solved independently by worker objects. Workers need to communicate shared edge values with neighbouring workers at each iteration. This has two important implications: (i) heat diffusion cannot be efficiently realised using the master-worker or branch-and-bound computational model and (ii) the problem incurs high-communication overhead because sub problems need to synchronize at each iteration.

Many existing distributed object frameworks (e.g. Javelin, Ninflet and Charlotte) only support applications that can be written as a master-worker or branch-and-bound problem. This limits their applicability to problems that can be formulated according to one of these two models. A motivating factor for the implementation of Babylon v2.0 was to provide a more flexible framework for building distributed applications that was not limited to a particular programming model. The heat diffusion benchmark is a good example of such an application. Furthermore, since workers need to synchronize with each other at each iteration, this benchmark provides both a measure of Babylon v2.0 performance under communication intensive conditions and a good test of Babylon's support for nested remote method invocations. The complete source for the heat diffusion test application is given in Appendix C.

The results for the heat diffusion benchmark are presented in Table 5.6. All times are reported in milliseconds. The total execution time and the corresponding speedup are reported for each configuration and grid size. The first column for each size denotes the speedup while the second column shows the total execution time. Measured times include the transmission of the initialized grid blocks to each worker object and the transmission of the results back to the master program. Worker object always use the remote method invocation facilities (and RMI by extension) for communication even if they are in the same server.

Although the speedup values obtained in this benchmark are smaller than those obtained in the matrix multiplication benchmark, the results are promising and indicate that speedup can be achieved despite the communication-intensive nature of the problem. In fact, we will show that the high communication overhead of this type of application would make it difficult to obtain much larger speedups on a cluster of workstations without using specialized communication hardware. Similar experiments in [34], [33] and [52] yield speedups in the range of 2.5 to 4.9 on eight processors and 3.5 to 6.5 on 16 processors. Speedups obtained using Babylon v2.0 are as high as 5.0 using 8 worker objects and 6.5 using 16 worker objects. This suggests that Babylon v2.0 can be just as effective at running communication-intensive applications as other conventional systems.

It is always important to use a large grain size in order to maximise the potential speedup of a parallel application. This is especially true for the heat diffusion benchmark, where the actual diffusion algorithm is

Configuration [workers/servers/machines]	Grid Size (floats)							
	512x512		*1024x1024*		*2048x2048*		*4096x4096*	
	$S(N)$	*Time (ms)*	$S(N)$	*Time (ms)*	$S(N)$	*Time (ms)*	$S(N)$	*Time (ms)*
Sequential Java	1.0	2,571	1.0	10,318	1.0	40,408	1.0	161,280
Babylon v2.0 [1w/1s/1m]	1.0	2,682	1.0	10,807	1.0	42,531	0.9	187,284
Babylon v2.0 [2w/2s/2m]	1.4	1,903	1.6	6,383	1.7	23,287	1.7	92,762
Babylon v2.0 [2w/2s/1m]	1.0	2,504	1.6	6,412	1.5	26,421	1.5	104,938
Babylon v2.0 [2w/1s/1m]	1.1	2,442	1.5	7,054	1.6	25,569	1.6	103,923
Babylon v2.0 [4w/4s/4m]	1.6	1,559	2.5	4,145	3.0	13,672	3.1	52,295
Babylon v2.0 [4w/4s/2m]	1.4	1,842	2.3	4,504	2.7	14,760	2.9	56,128
Babylon v2.0 [4w/2s/2m]	1.5	1,705	2.3	4,544	2.8	14,510	2.9	56,184
Babylon v2.0 [8w/8s/8m]	1.9	1,334	3.7	2,798	4.5	8,908	5.0	32,483
Babylon v2.0 [8w/8s/4m]	1.9	1,379	3.5	2,963	4.3	9,474	4.7	34,574
Babylon v2.0 [8w/4s/4m]	1.8	1,437	3.4	3,061	4.2	9,532	4.7	34,546
Babylon v2.0 [16w/16s/8m]	1.9	1,345	3.9	2,677	5.7	7,076	6.5	24,781
Babylon v2.0 [16w/8s/8m]	2.0	1,276	4.2	2,482	5.7	7,048	6.4	25,218

Table 5.6: Babylon v2.0 Heat Diffusion Speedup and Execution Time using 100 Iterations (ms)

relatively inexpensive in terms of computational complexity when compared to the costs of synchronization at each iteration. Under these circumstances, the amount of work done between synchronizations can easily become dominated by the synchronization overhead if the grain size is too small. In our experiments, the speedup obtained for the smallest tested grid size (512x512) is very low for all configurations because the granularity is simply too small. However, the speedup values improve significantly when the experiments involve larger grid sizes. For instance, the speedup achieved using eight worker objects and a 4096x4096 grid is as high as 5.0 compared with a speedup of only 1.9 for the same configuration and a 512x512 grid.

To investigate the details of the communication overhead for the heat diffusion experiment, we performed an analysis similar to the one presented in Table 5.5 for the matrix multiplication experiment. The details of this analysis for the 4096x4096 grid results are presented in Table 5.7.

Configuration	*Total Time (ms)*	*Actual Speedup*	*Distrib. Data (MB)*	*Worker Synch. Data (KB)*	*Estimated Trans. Time*		*Serial Fraction*	*Speedup Bound*
					(ms)	*(% of Total)*		
Sequential	161,280	1.0	0.0	0.0	0	0.0%	0.000	1.0
[1w/1s/1m]	187,284	0.9	128.0	0.0	12,391	6.7%	0.077	1.0
[2w/2s/2m]	92,762	1.7	128.0	32.0	12,691	13.7%	0.079	1.9
[2w/2s/1m]	104,938	1.5	128.0	32.0	12,691	12.1%	0.079	1.9
[2w/1s/1m]	103,923	1.6	128.0	32.0	12,691	12.2%	0.079	1.9
[4w/4s/4m]	52,295	3.1	128.0	32.0	12,691	24.3%	0.079	3.2
[4w/4s/2m]	56,128	2.9	128.0	32.0	12,691	22.6%	0.079	3.2
[4w/2s/2m]	56,184	2.9	128.0	32.0	12,691	22.6%	0.079	3.2
[8w/8s/8m]	32,483	5.0	128.0	32.0	12,691	39.1%	0.079	5.2
[8w/8s/4m]	34,574	4.7	128.0	32.0	12,691	36.7%	0.079	5.2
[8w/4s/4m]	34,546	4.7	128.0	32.0	12,691	36.7%	0.079	5.2
[16w/16s/8m]	24,781	6.5	128.0	32.0	12,691	51.2%	0.079	7.3
[16w/8s/8m]	25,218	6.4	128.0	32.0	12,691	50.3%	0.079	7.3

Table 5.7: Babylon v2.0 Heat Diffusion Result Analysis (4096x4096)

The layout of Table 5.7 is very similar to that of Table 5.5 in the matrix multiplication application section. The "Distributed Data" column contains the amount of grid data transmitted over the network in each configuration for a 4096x4096 grid between the master program and the worker objects. It is computed as the sum of the data distributed by the master program to the worker objects and the data returned by each worker object to the master program at the end of computation. Note that the grid is composed of Java `floats` which occupy four bytes each.

Each pair of neighbouring worker objects must exchange two rows of data at each iteration. Each row of data consists of 4096 Java `floats` so the amount of data exchanged at each iteration between a pair of neighbouring worker objects is $4096 \times 4 \times 2 = 32.0$ KB. This amount is reported in the column labelled "Worker Synch. Data". It should be noted that the total amount of data transmitted at each iteration is actually $32.0 \times (N - 1)$ KB because each of the $(N - 1)$ pairs of worker objects exchange two rows. However, we are only interested in individual exchanges when we compute the transmission time because, ideally, each pair of worker objects should exchange their boundary rows concurrently.

In the matrix multiplication experiment section, it was shown that the remote method invocation results in Table 5.1 can be used to make relatively accurate predictions of transmission times. Based on these results, we can compute estimates of the total transmission time required for the initial and final grid distribution and the data exchanges that occur at each iteration. To predict transmission times we use a transmission rate of 10.33 MB/s which is based on the 10M column of Table 5.1. The total transmission time is then computed as the time required to exchange two rows of data (32 KB) a total of 99 times (once for each iteration minus the last one) plus the time required for the initial and final grid distribution (128 MB). The computed estimates are reported in the column labelled "Estimated Transmission Time".

The data transmission times enable us to compute the serial fraction, f, of the computation. Based on f, we compute an upper bound on the possible speedup, S_{Amdahl}, using Amdahl's Law. The maximum possible speedup for each configuration is displayed in the column labelled "Speedup Bound".

We can see from these results that the actual speedup values in Table 5.6 are very close to the maximum possible speedup values computed using Amdah's Law. For instance, the maximum possible speedup using eight workers was found to be 5.2. This is very close to our speedup of 5.0 obtained in configuration *[8w/8s/8m]*. Similarly, the maximum speedup using 16 workers is 7.3 which is comparable to the speedup of 6.5 obtained in configuration *[16w/16s/8m]*. Based on these results, we can also predict the maximum speedup for other configurations that were not tested. For instance, the maximum speedup using 32 worker objects is 9.3.

Unlike the matrix multiplication experiment, the amount of data transmitted in the heat diffusion experiment is constant and does not depend on the number of worker objects (assuming the data exchanges between pairs of workers at each iteration occur concurrently). This means that, in theory, we should be able to obtain a speedup as high as 12.6 for sufficiently large N.[4] However, this theoretical bound is rather optimistic. It fails to take into account other sequential components such as software overhead to divide the problem into sub-problems, synchronization and network contention overhead when shared data rows are exchanged and other inherent serial components such as garbage collection and various system and scheduling related overheads. These factors will limit the actual speedup that can be obtained and will likely prevent it from becoming as high as 12.6.

[4]Based on Amdahl's Law, the maximum speedup of a computation as $N \to \infty$ is equal to $\frac{1}{f}$ for constant f.

5.4.3 Serial Matrix Multiplication with Migration

In this section, we conduct an experiment designed to investigate the impact of idle worker object migration on application performance similar to a Babylon v1.0 evaluation experiment performed by Matthew Izatt. Babylon v2.0 is designed to support persistent objects which may remain on Babylon servers for long periods of time. At the same time, the migration of a worker object from a particular Babylon server may be triggered if a faster server becomes available or if the current server needs to be shutdown for some reason (e.g., maintenance or upgrade). As a result, long-lived objests may need to be migrated several times during their lifetime. It is important that the impact of these migrations be as small possible on the overall run-time of the application. To investigate this impact, we compare the run time of the matrix multiplication benchmark running on a single Babylon server to one that is migrated across eight different servers during its execution.

In this experiment, a matrix multiplication worker object is exported to a Babylon server. In the non-migrating version of the benchmark, we simply measure the time required to multiply two matrices together on the server and to return the result back to the master. In the migrating version of the benchmark, one eighth of the matrix multiplication is performed on the worker object after which it is migrated to a new Babylon server. This process is repeated seven times, at which point the multiplication completes and the result is returned to the master. Measured times include the transmission of the matrix data to the worker object and the transmission of the result back to the master program. All times are reported in milliseconds. Table 5.8 presents the results of the benchmark with and without migration and includes an inflation factor which measures the increase in run time of the migrating version relative to the non-migrating version.

Application	Matrix Size (integers)			
	256x256	*512x512*	*1024x1024*	*2048x2048*
Matrix Multiplication without Migrations	1,657	17,177	140,203	1,226,433
Matrix Multiplication with 7 Migrations	3,372	20,768	153,264	1,296,189
Inflation Factor	2.04	1.21	1.09	1.06

Table 5.8: Babylon v2.0 Time for Matrix Multiplication including Migrations (ms)

The results in Table 5.8 suggest that the impact of migration on overall application performance is quite low, especially when one takes into consideration the amount of data and code that must be migrated during the course of each experiment. For instance, each migration in the 2048x2048 matrix experiment incurs overheads that include the transfer of the worker object (48 MB of matrix data), the client codebase and any serialization and deserialization overhead. The total amount of data transferred during the migrations totals more than 330 MB by the end of the experiment. Despite this communication overhead, the inflation factor for the 2048x2048 matrix case is only 1.06.

Even the 2.04 inflation factor for the 256x256 experiment is quite low when one considers that the run-time of an application could effectively double just by adding a second job to the same machine (assuming they have to share the same CPU in a round-robin fashion for the lifetime of the application). Moreover, a 256x256 matrix multiplication normally takes 1.6 seconds to complete without any migrations. Migrating a worker object seven times in 1.6 seconds is not a very likely usage scenario and suggests that the resulting inflation factor is a worse-case scenario. Many long-lived persistent objects also won't be actively executing methods all the time. A worker object could be migrated during such an idle period which would not incur any additional run time overhead.

It should be noted that the inflation factors obtained in this experiment are closely related to the type of application that was used. In this experiment the computation time takes $O(N^3)$ while the costs of migration are only $O(N^2)$. Consequently, the inflation factors could have been made arbitrarily small simply by making the matrix sizes sufficiently large. However, the matrix sizes used in this experiment are not unrealistic and we believe the resulting inflation factors still give valuable insight into the impact of worker object migration on application performance. Developers should be aware that other applications may exhibit different (higher or lower) inflation factors than those observed here.

The application migration benchmark results are quite promising and demonstrate that for longer running applications, occasional migrations will not have a significant impact on overall performance. The results also agree closely with the inflation factors reported by Matthew Izatt in his thesis on Babylon v1.0 where the same type of experiment was performed.

5.5 Discussion

In this chapter, we examined the performance of basic Babylon operations and Babylon as a whole by running a several microbenchmarks and some larger and more realistic application experiments. The results demonstrate that Babylon v2.0 performs at least as well or better than Babylon v1.0 in all experiments. An optimized implementation of worker object migration has resulted in significant improvements for the migration of small worker objects and also maintains good migration performance for larger worker objects. Remote method invocations are also slightly faster than Babylon v1.0. Despite these improvements, simple remote method invocation on worker objects in Babylon v2.0 is still less efficient than normal Java RMI. However, we believe that the advantages of using Babylon in terms of ease of use and additional functionality outweigh the disadvantage of increased remote method invocation overhead. Additionally, for the types of computations that would benefit from parallelization, the overheads do not prevent us from obtaining reasonable speedup in the applications tested.

It is interesting to note that if the applications in this chapter had been written using standard Java RMI instead of Babylon v2.0, the results would have been almost identical to those obtained in our experiments. For instance, the speedup bound for a Java RMI implementation of the matrix multiplication application using a 2048x2048 matrix and eight worker objects is 7.4 based on the raw RMI performance results presented in Table 5.1. This bound is identical to the speedup bound obtained for the Babylon v2.0 version of the application. Similarly, the speedup up bound for Java RMI implementation of the heat diffusion application using a 4096x4096 grid and 16 worker objects is 6.3. This is almost identical to the 6.2 bound estimated for the Babylon v2.0 implementation of the application. This suggests that the added overhead of basic remote method invocations in Babylon v2.0 would have almost no impact when used in the context of larger and more realistic applications.

The application benchmark results are quite promising and demonstrate that sequential applications have the potential to achieve reasonable speedups if converted to parallel Babylon applications. The matrix multiplication benchmark was used to evaluate the performance of a typical master-worker computation using Babylon v2.0. The heat diffusion benchmark, on the other hand, was used to evaluate the performance of a communication-intensive Babylon v2.0 application that could not have been efficiently realised using the master-worker computation model. As a result, it could not have been easily written or executed using many existing distributed systems which do not support the more general programming model required to implement this type of application. Both applications were able to achieve significant speedups using 4, 8 and 16 worker objects. We also examined the performance of worker object migration in a large-scale application context and showed that migration does not have a significant impact on application performance.

The speedup results for the matrix multiplication and heat diffusion benchmarks are summarized in Figure 5.1. The best speedup results for each application are graphed in the figure. The straight dotted line represents perfect speedup. Overall, the results of the experiments in this chapter demonstrate that Babylon v2.0 can be quite effective in a clustered workstation environment, that significant speedups can be achieved using a wide range of applications, and that the performance of v2.0 is as good as or better than v1.0.

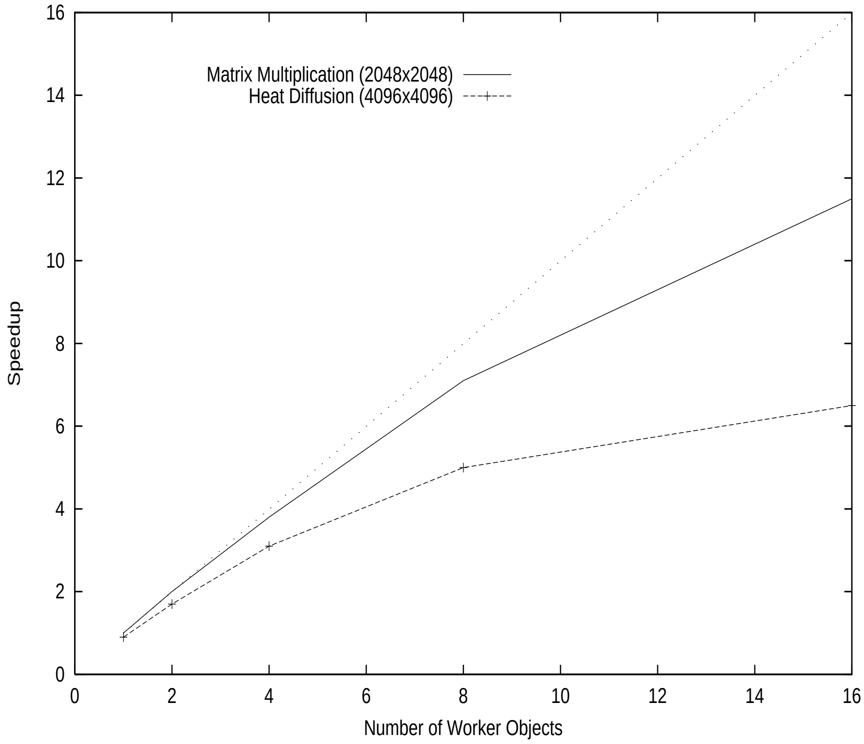

Figure 5.1: Babylon v2.0 Application Benchmark Speedups

Chapter 6

Conclusions and Future Work

This chapter presents concluding remarks and discusses some of the strengths and weaknesses of Babylon v2.0. Furthermore, although Babylon v2.0 can greatly facilitate distributed application development and administration, it also provides many opportunities for future work.

6.1 Conclusions

Babylon v2.0 is a framework for building parallel and distributed applications in Java. Babylon v2.0 incorporates features like object migration, asynchronous method invocation and dynamic class loading while providing an easy-to-use interface that works seamlessly with existing Java code. The result is a unique and powerful system that gives developers the necessary tools and services for building powerful cluster computing applications.

Babylon v2.0 provides two methods for creating remote worker objects which can be used by clients to perform distributed operations. The `Babylon.remoteNew()` method takes a class name as its argument and creates a new instance of the given class on a remote Babylon server. The `Babylon.export()` method distributes an existing local instance of a serializable object to a remote Babylon server. Both methods return a proxy that can be used to transparently invoke methods on the newly created worker object. Babylon v2.0 proxies provide transparent access to distributed objects by implementing the same interfaces as their distributed counterparts. Providing transparent access is a significant enhancement over Babylon v1.0, which used a clumsy method invocation interface that was completely different from the standard Java remote method invocation syntax and prevented compile-time detection of errors such as invoking a non-existent method, passing an incorrect number of arguments to a method, or passing arguments of the wrong type.

Babylon v2.0 also introduces a novel asynchronous method invocation technique based on proxy objects called asynchronous “tickets”. An asynchronous ticket is a proxy that can be used to make one asynchronous method invocation on a worker object. The method is invoked on the ticket using standard Java invocation syntax but the invocation is non-blocking. Applications can continue running normally while the invocation completes asynchronously and the ticket can be used to retrieve the method’s result at a later time. The advantage of this approach is that it is 100% Java compatible and that asynchronous method invocations can be checked at compile-time because they adhere to Java’s standard method invocation syntax. Several other distributed systems support asynchronous remote method invocations, however, almost all proposed solutions use either non-standard invocation syntax which cannot be checked at compile time [42] or require custom preprocessors and special language extensions which are not 100% Java compatible [41, 16].

Another contribution of Babylon v2.0 is the addition of stateful remote method invocations that enable us to implement basic worker object access restrictions. Developers can now create private or public worker objects. Private worker objects are only accessible to the client that exported the object whereas public worker objects can be looked up and used by any Babylon client. Public worker objects can also be made persistent so that they remain active even after the client that exported them exits. In other words, they won't be garbage collected when they are no longer referenced by any clients. Persistent worker objects can be used to provide general services to any Babylon client. Developers use a policy file to mark their worker objects as public, private, persistent or transient.

Babylon v2.0 also provides separate namespaces on Babylon servers for different clients. Many other systems, such as ProActive and Babylon v1.0, use a single namespace for all clients. Using a single namespace prevents the unloading of client classes when the client application exits. As a result, a server restart is required every time a client application changes so that the new class definitions can be loaded. Separate namespaces also provide security so that different clients do not have access to each other's classes. Furthermore, Babylon v2.0 works closely with the distributed garbage collection facilities provided by the JVM so that distributed objects that are no longer referenced by clients can be automatically garbage collected by the JVM (unless they are marked as persistent).

Worker object migration has also been completely redesigned in Babylon v2.0. Migration performance has been optimized and is now almost 40% faster than Babylon v1.0 for small worker objects. The implementation of immediate migration has also been rewritten to fix deficiencies present in the Babylon v1.0 implementation. The new design no longer relies on deprecated `Thread.stop()` method to stop running threads and provides a mechanism to programmatically defend against the checkpoint consistency problem.

Furthermore, support for nested remote method invocations expands the number and types of applications that can be written in Babylon v2.0. Most significantly, Babylon v2.0 can support a wide range of applications that many other existing systems cannot (some of which are limited to supporting only the master-worker computation model). This enables support for applications such as the heat diffusion experiment of Chapter 5 which requires communication between worker objects (such computations are not possible without nested invocations).

Performance evaluation results are also promising and indicate that sequential applications can achieve substantial performance gains by using Babylon v2.0 to distribute their objects. Experiments show that reasonable speedups can be obtained for both simple master-worker applications (e.g., matrix multiplication) and for more complicated and communication-intensive applications (e.g., heat diffusion). The experiments demonstrate that Babylon v2.0 can be used to effectively build and run clustered computing applications. To our knowledge, Babylon v2.0 is the only distributed Java object system that uses no special preprocessors or JVM extensions and provides all of these features.

It should be noted that although Java has improved and evolved significantly over the past years, it is still, in many ways, inadequate for solving problems that require finer grained control over thread behaviour and access to runtime elements. It is still difficult and often impossible to implement more advanced distributed system functions without using JVM extensions. This was most evident in Babylon's implementation of immediate migration. In order to adhere to our original goal of using no JVM extensions, preprocessors or a modified JVM, it was only possible to provide a crippled version of immediate migration which, unfortunately, requires some cooperation from the worker object to be migrated.

6.2 Future Work

In its current form, Babylon v2.0 is adequate for small grid cluster applications but lacks scheduling and fault-tolerance capabilities that would be required for large-scale deployment. A scheduling scheme that keeps track of active worker objects and available processors to implement load balancing would be extremely beneficial. Although other distributed systems projects (i.e., Javelin, Charlotte) focus on enhanced scheduling techniques, they don't seem to place much emphasis on how to track and use information about the execution environment in the context of a JVM (where little or no info about load/use is available).

Although Babylon v2.0 includes a basic checkpointing mechanism to save and load a snapshot of a worker object's state, it does not include a comprehensive fault-tolerance framework. Support for continued execution in the presence of some faults is crucial in a distributed system where arbitrary machines may crash or become disconnected from the network. A framework that provides worker object fault-tolerance using replication is an important enhancement that would make Babylon v2.0 more adequate for large scale deployment.

Furthermore, in the current implementation, a single scheduler instance is used to manage all the Babylon servers and worker objects in the system. This introduces a single point of failure and a potential bottleneck for environments with large numbers of servers and worker objects. A distributed scheduling scheme could be used in future versions to address some of these problems.

Although the Babylon v2.0 asynchronous method invocation interface has improved significantly, the implementation could still be further optimized. The current implementation starts a new thread at the client-side to handle each asynchronous invocation. This could become a serious bottleneck for applications that make a large number of asynchronous invocations. Techniques such as thread pooling or perhaps a non-threaded approach could be used to make asynchronous remote method invocation more efficient and scalable.

Furthermore, the policy file format for Babylon v2.0 is currently implemented as a flat file that specifies global application-wide policies. Although application-wide policies may be satisfactory for small distributed applications, they will not provide adequate flexibility when deploying larger applications with many distributed objects. For these types of applications a hierarchical policy file structure, which enables programmers to specify global system-wide policies that can be overridden by instance or class-specific policies, would be more appropriate.

Appendix A

Default Policy File

This section presents the contents of the default policy file used by Babylon v2.0. It is located in the `$BABYLON_HOME/bin/policy` directory of the Babylon v2.0 distribution. Lines beginning with `//` are comments and are ignored.

```
// Restrict remote method invocation access to the client's worker objects.
// [public, private]
// public -  enable other clients to invoke methods on worker objects
//           created by this client.
// private - only allow this client to invoke methods on the worker
//           objects it creates.
worker.access = private

// Specifies the lifetime of the object.
// transient = the worker is garbage collected as soon as there are no
//             remaining client references to it.
// persistent = the worker will persist even after the application
//              terminates and there are no remaining client references
//              to it. The worker will only be garbage collected if it
//              is explicitly unexported.
worker.lifetime = transient
```

Figure A.1: Default Policy File

Appendix B

Code Samples

B-1 Dictionary Interface and Implementation

The following code samples are used for various examples in this thesis. The `Dictionary` interface exposes two operations that can be performed on a dictionary object. The `DictionaryImpl` provides a sample implementation of the `Dictionary` interface.

```
package examples;

public interface Dictionary
{
   public String getDefinition(String word)
      throws WordNotFoundException;
}
```

Figure B.1: Interface Dictionary

```
package examples;

import java.io.*;
import java.util.*;

public class DictionaryImpl implements
   Dictionary, Serializable
{
   Hashtable dictionary;

   public String getDefinition(String word)
      throws WordNotFoundException;
   {
      String definition = (String) dictionary.get(word);
      if (definition == null) {
         throw new WordNotFoundException(word);
      }
      return definition;
   }
}
```

Figure B.2: Class DictionaryImpl

B-2 Asynchronous Remote Method Invocation

In the following code example, the definitions of ten words are obtained using a Babylon v2.0 worker object. The reference to the public `Dictionary` worker object is obtained using the `Babylon.lookup()` method. This example assumes that a `Dictionary` worker object with the instance name "myDictionary" has been previously created. The definition for each word is obtained using an asynchronous invocation of the worker object's `getDefinition()` method. The example demonstrates how clients can perform multiple invocations on a worker object concurrently and retrieve the results after all the invocations have been made.

```
Dictionary diction = null;

try {
   diction = (Dictionary) babylon.core.Babylon.lookup(
      "DictionaryServer",  Dictionary.class
   );
}
catch (babylon.core.InstanceNotBoundException e) {
   // A worker object with the instance name "DictionaryServer" does not exist.
}

// Allocate space for 10 asynchronous tickets.
Dictionary asynch_ticket[] = new Dictionary[10];

// Now get the definition of 10 words asynchronously.
for (int i = 0; i < 10; i++) {
   // Obtain a new asynchronous ticket for this invocation.
   asynch_ticket[i] = (Dictionary)babylon.core.AsynchTicket.newTicket(diction);

   try {
      // Invoke the getDefinition() method asynchronously using the ticket.
      asynch_ticket[i].getDefinition(word[i]);
   }
   catch (Exception e) {}
}

// Initialize space for the results of the method invocations.
String results[] = new String[10];

// Now retrieve all the results and store them in the results[] array.
for (int i = 0; i < 10; i++) {
   results[i] = (String) babylon.core.AsynchTicket.getResult(asynch_ticket[i]);
}
```

Figure B.3: Asynchronous method invocation in Babylon v2.0

Appendix C

Sample Babylon v2.0 Application - Heat Diffusion

C-1 GridClient.java

```
package babylon.tests.grid;

import babylon.core.*;
import babylon.io.*;
import babylon.util.Debug;

import java.util.Arrays;

public class GridClient
{
   public static void main(String args[])
   {
      if (args.length != 5)
      {
         System.err.println("To run grid diffusion test program:");
         System.out.println(
            "grid_test <scheduler name> <jar_file> <num processors> " +
            "<matrix size> <# iterations>"
         );
         System.exit(0);
      }

      // Parse the command line paramters.
      int nprocs = Integer.parseInt(args[2]);
      int matrix_size = Integer.parseInt(args[3]);
      int iterations = Integer.parseInt(args[4]);

      System.out.println("Running Heat Diffusion Test Program...");

      try {
         Babylon.initApplication(args[0], args[1], null);
      }
      catch (Exception e) {
         e.printStackTrace();
         System.exit(0);
      }

      // Create new grid.
      Grid grid = new Grid(matrix_size, matrix_size);


      /**** Starting Servers ****/

      long serverStartT, serverFinishT;
      serverStartT = System.currentTimeMillis();

      // Allocate space for the grid diffuser server references.
      GridDiffuser gd[] = new GridDiffuser[nprocs];
      float grid_section[][][] = new float[nprocs][][];
```

```
int i = 0;
int start_row = 0;
int end_row = -1;
int nrows = matrix_size / nprocs;

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

      grid_section[i] = grid.getRows(start_row, end_row);

      gd[i] = (GridDiffuser) Babylon.export(
         new GridDiffuserImpl(), GridDiffuser.class
      );

   }
}
catch (Exception e) {
   System.err.println("Remote creation error at server: " + i);
   e.printStackTrace();
   System.exit(-1);
}

// Initialize boundary constraints...
// Float boundary constraints indicate that a constant temperature
// of the given value is applied to that boundary.
Float topBoundary = new Float(100f);
Float bottomBoundary = new Float(100f);
Float leftBoundary = new Float(100f);
Float rightBoundary = new Float(100f);

Object borders[] = new Object[Grid.NUM_EDGES];

// Set the boundary constraints on each worker object.
if (nprocs == 1)
{
   borders[Grid.TOP] = topBoundary;
   borders[Grid.BOTTOM] = bottomBoundary;
   borders[Grid.LEFT] = leftBoundary;
   borders[Grid.RIGHT] = rightBoundary;

   // Set the boundary on the one worker.
   gd[0].setAdjacentGrids(borders);
}
else
{
   borders[Grid.TOP] = topBoundary;
   borders[Grid.BOTTOM] = null;
   borders[Grid.LEFT] = leftBoundary;
   borders[Grid.RIGHT] = rightBoundary;

   gd[0].setAdjacentGrids(borders);

   for (i = 1; i < (nprocs-1); i++)
   {
      borders[Grid.TOP] = gd[i-1];
      borders[Grid.BOTTOM] = null;
      borders[Grid.LEFT] = topBoundary;
      borders[Grid.RIGHT] = topBoundary;

      gd[i].setAdjacentGrids(borders);
   }

   borders[Grid.TOP] = gd[nprocs-2];
   borders[Grid.BOTTOM] = bottomBoundary;
   borders[Grid.LEFT] = topBoundary;
   borders[Grid.RIGHT] = topBoundary;


   gd[nprocs-1].setAdjacentGrids(borders);
}

serverFinishT = System.currentTimeMillis();
```

```
      /**** End Server Setup ****/


      /**** Start Diffusion ****/

      long execStartT, execFinishT;
      execStartT = System.currentTimeMillis();

      // Start diffusing... first prepare space for the asynchronous tickets.
      GridDiffuser gd_asynch[] = new GridDiffuser[nprocs];
      for (i = 0; i < nprocs; i++)
      {
         gd_asynch[i] = (GridDiffuser) AsynchTicket.newTicket(gd[i]);

         // Calling diffuse() asynchronously on each worker object.
         gd_asynch[i].diffuse(grid_section[i], 0, iterations);
      }

      // Now retrieve the results from each worker.
      float sub_result[][][] = new float[nprocs][][];
      for (i = 0; i < nprocs; i++) {
         try {
            sub_result[i] = (float[][])AsynchTicket.getResult(gd_asynch[i]);
         }
         catch (RemoteExecException t) {
            t.printStackTrace();
            System.exit(-1);
         }
      }

      execFinishT = System.currentTimeMillis();

      long serverSetupT = serverFinishT - serverStartT;
      long execTimeT = execFinishT - execStartT;
   }
}
```

C-2 Grid.java

```
package babylon.tests.grid;

public class Grid
{
   public static final String APP_ID = "GRID";
   public static final int TOP = 0;
   public static final int BOTTOM = 1;
   public static final int LEFT = 2;
   public static final int RIGHT = 3;
   public static final int NUM_EDGES = 4;

   private int rows = 0;
   private int columns = 0;

   private float[][] grid;

  /**
   * Create a new grid of the given size initialized with 0's.
   */
   public Grid(int rows, int columns)
   {
      this.rows = rows;
      this.columns = columns;
      grid = new float[rows][columns];
   }

   public float[][] getRows(int start, int end)
   {
      int sub_rows = (end - start) + 1;
      float[][] sub = new float[sub_rows][columns];
      for (int i = 0; i < sub_rows; i++) {
         for (int j = 0; j < columns; j++) {
            sub[i][j] = grid[start+i][j];
         }
      }
      return sub;
   }

  /**
```

```
   * Returns the identifier for the opposite edge of the edge specified by
   * the argument.
   */
   public static int oppositeEdge(int edge)
   {
      if (edge == TOP) {
         return BOTTOM;
      }
      else if (edge == BOTTOM) {
         return TOP;
      }
      else if (edge == LEFT) {
         return RIGHT;
      }
      else if (edge == RIGHT) {
         return LEFT;
      }

      throw new IllegalArgumentException(
         "oppositeEdge: Invalid edge value: " + edge
      );

   }

  /**
   * Simply returns a string representation of the given edge value.
   */
   public static String edgeToString(int edge)
   {
      if (edge == TOP) {
         return "TOP";
      }
      else if (edge == BOTTOM) {
         return "BOTTOM";
      }
      else if (edge == LEFT) {
         return "LEFT";
      }
      else if (edge == RIGHT) {
         return "RIGHT";
      }

      throw new IllegalArgumentException(
         "edgeToString: Invalid edge value: " + edge
      );
   }
}
```

C-3 GridDiffuser.java

```
package babylon.tests.grid;

public interface GridDiffuser
{
  /**
   * Initializes this GridDiffuser's grid to the given array. The iteration
   * state is set to 'iteration'.
   */
   public void setGrid(float[][] grid, int iteration);

  /**
   * Initializes references to the GridDiffuser workers that are adjacent
   * to this one.
   */
   public void setAdjacentGrids(Object[] adjacentGrids);

  /**
   * Performs a diffusion on the grid for 'iterations' iterations.
   */
   public float[][] diffuse(int iterations);
   public float[][] diffuse(float[][] grid, int curIteration, int iterations);

  /**
   * Returns the edge values for the given edge.
   */
   public float[] getEdgeValues(int edge, int iteration, float[] remoteEdgeIn);
}
```

C-4 GridDiffuserImpl.java

```
package babylon.tests.grid;

import java.util.Arrays;

import babylon.util.Debug;


public class GridDiffuserImpl implements GridDiffuser, java.io.Serializable
{
   int rows, columns;
   int remoteTop, remoteBottom, remoteLeft, remoteRight;
   int localTop, localBottom, localLeft, localRight;

   // Used to save the original grid.
   float[][] origGrid;

   // The two computation grids. At each iteration we compute the updated
   // temperature distribution of the srcGrid and place the result in the
   // targetGrid. The outer edges of these grids represent the neighbouring
   // edge values and must be update at each iteration. The four values in the
   // remoteEdgeState array indicate what iteration the values in the given
   // outer edge correspond to.
   float[][] srcGrid;
   float[][] targetGrid;

   // The outer edges of srcGrid always contain the edge values of the
   // neighbouring grid for a particular iteration. This variable keeps track
   // of that iteration. It hold a value for Grid.BOTTOM/LEFT/RIGHT/TOP.
   int[] remoteEdgeState;

   // Tracks the current iteration. Each of the four remoteEdgeState values
   // must be equal to curIteration+1 before we can compute the temperature
   // distribution for the current iteration.
   int curIteration = -1;


   // Stores information about the kind of edge at each side. See the
   // setAdjacentGrids method for more information on the type of information.
   Object[] adjacentGrids; // Holds a value for Grid.BOTTOM/LEFT/RIGHT/TOP.

   // Special array the holds a copy of each of the local edges. Access must
   // be synchronized because several threads may access this data (i.e.
   // neighbouring worker objects).
   float[][] localEdges;

   // Used to prevent a client from changing the adjacentGrids once the
   // diffusion computation starts.
   Integer diffuseLock = new Integer(1);

   public GridDiffuserImpl()
   {
      // Default constructor... client will have to call setGrid() to
      // initialize the grid.
   }


   public GridDiffuserImpl(float[][] grid, int curIteration)
   {
      setGrid(grid, curIteration);
   }

  /**
   * Initializes this GridDiffuser's grid to the given array. The iteration
   * state is set to 'iteration'.
   */
   public void setGrid(float[][] grid, int iteration)
   {
      this.origGrid = grid;
      this.rows = grid.length;
      this.columns = grid[0].length;

      remoteTop = 0;
      remoteBottom = rows+1;
      remoteLeft = 0;
      remoteRight = columns+1;
```

```
    localTop = 1;
    localBottom = rows;
    localLeft = 1;
    localRight = columns;

    // Allocate the space for the grids. Allow extra room for boundaries.
    srcGrid = new float[rows+2][columns+2];
    targetGrid = new float[rows+2][columns+2];

    // Populate srcGrid with the parameter grid.
    for (int i = 0; i < rows; i++) {
       for (int j = 0; j < columns; j++) {
          srcGrid[i+1][j+1] = grid[i][j];
       }
    }

    // Initialize the four values (Grid.TOP/BOTTOM/RIGHT/LEFT) of the
    // remoteEdgeState to currentIteration-1.
    remoteEdgeState = new int[Grid.NUM_EDGES];
    Arrays.fill(remoteEdgeState, curIteration-1);

    localEdges = new float[Grid.NUM_EDGES][];
    localEdges[Grid.TOP] = new float[columns];
    localEdges[Grid.BOTTOM] = new float[columns];
    localEdges[Grid.LEFT] = new float[rows];
    localEdges[Grid.RIGHT] = new float[rows];

    setLocalEdges(iteration);
 }

/**
 * Initializes references to the GridDiffuser workers that are adjacent
 * to this one.
 * @param adjacentGrids Array containing four elements (Grid.TOP, Grid.BOTTOM,
 *            Grid.LEFT and Grid.RIGHT) each representing the type of edge
 *            on the given side of this worker object.
 *            An edge of type Float represents a constant
 *            heat source of the specified temperature (no need to obtain
 *            neighbouring values at each iteration). An edge of type float[]
 *            represents a constant heat source of the specified temperatures
 *            each point (no need to obtain neighbouring values at each
 *            iteration). An edge of type GridDiffuser represents a
 *            neighbouring worker which must queried at each iteration
 *            for updated values (in this case we piggyback our updated
 *            edge values on that request). A null edge value indicates
 *            that there is a neighbouring worker but the updated edge
 *            values will be piggybacked on a request from that worker
 *            for our edge values.
 */
 public void setAdjacentGrids(Object[] adjacentGrids)
 {
    // First make sure there is no diffusion computation in progress by
    // grabbing the diffuseLock.
    synchronized (diffuseLock) {
       // Save the adjacentGrids values.
       this.adjacentGrids = adjacentGrids;

       // If the adjacent edges values are fixed (constant heat source
       // specified by Float or float[]) we save the values in the borders
       // of our grids for quick access.
       for (int d = 0; d < Grid.NUM_EDGES; d++)
       {
          if (adjacentGrids[d] != null)
          {
             if (adjacentGrids[d] instanceof float[])
             {
                saveRemoteEdge(srcGrid, (float[])adjacentGrids[d], d);
                saveRemoteEdge(targetGrid, (float[])adjacentGrids[d], d);
             }
             else if (adjacentGrids[d] instanceof Float)
             {
                saveRemoteEdge(srcGrid, (Float)adjacentGrids[d], d);
                saveRemoteEdge(targetGrid, (Float)adjacentGrids[d], d);
             }
          }
       }
    }
 }

 public float[][] diffuse(float[][] grid, int curIteration, int numIterations)
```

```java
   {
      setGrid(grid, curIteration);
      return diffuse(numIterations);
   }

  /**
   * Performs a diffusion on the grid for 'numIterations' iterations.
   */
   public float[][] diffuse(int numIterations)
   {
      // Grab the diffuseLock so that the adjacentGrid values can no longer be
      // modified.
      synchronized (diffuseLock) {
         int end_iteration = curIteration + numIterations;
         int i = curIteration;
         while (i < end_iteration) {

            // Make sure we have the updated values for the remote edges
            // before we compute the diffusion for this iteration.
            getRemoteEdgesFromAdjacentWorkers();

            // Compute the temperature diffusion for this iteration.
            diffuse();

            // We're done. Swap the source and target grid.
            float[][] temp = srcGrid;
            srcGrid = targetGrid;
            targetGrid = temp;

            // Update the local edge array and the iteration.
            setLocalEdges(++i);
         }

         for (i = 0; i < rows; i++) {
            for (int j = 0; j < columns; j++) {
               origGrid[i][j] = srcGrid[i+1][j+1];
            }
         }

         return origGrid;
      }
   }

  /**
   * Diffuses the srcGrid and put the results in the target grid.
   */
   private void diffuse() {
      for (int i = 1; i < rows+1; i++) {
         for (int j = 1; j < columns+1; j++) {
            targetGrid[i][j] =
               (srcGrid[i-1][j] + srcGrid[i+1][j] +
                   srcGrid[i][j-1] + srcGrid[i][j+1]) / 4;
         }
      }
   }

  /**
   * Returns the values for the given edge from this worker. This is the
   * method that is normally called by an adjacent worker object to get the
   * edge values from this worker. This method is meant to swap neighbouring
   * edges and therefore the caller must also include the values of his
   * edges using the remoteEdgeIn argument.
   * @param requestEdge The edge that is being requested by the caller. One of
   *            Grid.TOP/BOTTOM/RIGHT/LEFT
   * @param iteration Specifies the iteration that the edge should be.
   * @param remoteEdgeIn The value of the callers piggybacked edge.
   */
   public synchronized float[] getEdgeValues(int requestEdge, int iteration,
      float[] remoteEdgeIn)
   {
      if (iteration < curIteration) {
         throw new java.lang.IllegalStateException(
            "Requested iteration (" + iteration +
            ") is less than current iteration (" + curIteration + ")"
         );
      }

      // Wait until this worker object has reached the requested iteration.
      while (iteration > curIteration) {
         try {
```

```java
            wait();
        } catch (InterruptedException e) {}
    }

    // Save the piggybacked edge and update the value in remoteEdgeState for
    // appropriate side to reflect the newly updated remote edge.
    saveRemoteEdge(srcGrid, remoteEdgeIn, requestEdge);
    remoteEdgeState[requestEdge] = curIteration;

    // Wake up any threads waiting in getRemoteEdgesFromAdjacentWorkers()
    // for the piggybacked remote edges.
    notifyAll();

    return localEdges[requestEdge];
}

/**
 * Updates the values in the local remote edge array by querying adjacent
 * workers so that is consistent with the current iteration. This method
 * assumes the localEdges array has already been updated using
 * setLocalEdges().
 */
private synchronized void getRemoteEdgesFromAdjacentWorkers()
{
    for (int i = 0; i < Grid.NUM_EDGES; i++) {
        if (adjacentGrids[i] == null) {
            while (remoteEdgeState[i] != curIteration) {
                // Wait for the remote grid to update the value which
                // will piggybacked when it calls getEdgeValues.
                try {
                    wait();
                }
                catch (InterruptedException e) {}
            }
        }
        else if (adjacentGrids[i] instanceof float[]) {
            // This edge is an outer boundary so we can simply use the
            // existing values.
            remoteEdgeState[i] = curIteration;
        }
        else if (adjacentGrids[i] instanceof Float) {
            // This edge is an outer boundary so we can simply use the
            // existing values.
            remoteEdgeState[i] = curIteration;
        }
        else if (adjacentGrids[i] instanceof GridDiffuser ) {
            GridDiffuser gd = (GridDiffuser)adjacentGrids[i];
            float[] remoteEdge = gd.getEdgeValues(
                Grid.oppositeEdge(i), curIteration, localEdges[i]
            );
            saveRemoteEdge(srcGrid, remoteEdge, i);
            remoteEdgeState[i] = curIteration;
        }
    }

}


/**
 * Saves the edges of the grid into a separate array so that
 * they can be retrieved by adjacent grids using the getEdgeValues
 * method.
 */
private synchronized void setLocalEdges(int iteration)
{
    for (int i = 0; i < rows; i++) {
        localEdges[Grid.LEFT][i] = srcGrid[i+1][localLeft];
        localEdges[Grid.RIGHT][i] = srcGrid[i+1][localRight];
    }
    for (int j = 0; j < columns; j++) {
        localEdges[Grid.TOP][j] = srcGrid[localTop][j+1];
        localEdges[Grid.BOTTOM][j] = srcGrid[localBottom][j+1];
    }
    curIteration = iteration;

    // Wake up any threads waiting in getEdgeValues()...
    notifyAll();

}
```

```java
   /**
    * This utility method simple copies the value of 'value' into all the
    * cells of the outer edge of 'dest'. See also the overidden version below.
    */
   private void saveRemoteEdge(float[][] dest, Float value, int edge)
   {
      float floatValue = value.floatValue();
      if (edge == Grid.TOP) {
         for (int j = 0; j < columns; j++) {
            dest[remoteTop][j+1] = floatValue;
         }
      }
      else if (edge == Grid.BOTTOM) {
         for (int j = 0; j < columns; j++) {
            dest[remoteBottom][j+1] = floatValue;
         }
      }
      else if (edge == Grid.LEFT) {
         for (int i = 0; i < rows; i++) {
            dest[i+1][remoteLeft] = floatValue;
         }
      }
      else if (edge == Grid.RIGHT) {
         for (int i = 0; i < rows; i++) {
            dest[i+1][remoteRight] = floatValue;
         }
      }
      else {
         throw new IllegalArgumentException(
            "saveRemoteEdge: Invalid edge value: " + edge
         );
      }
   }

   /**
    * This utility method simple copies 'value' into the outer edge of 'dest'.
    * See also the overidden version above.
    */
   private void saveRemoteEdge(float[][] dest, float[] data, int edge)
   {
      if (edge == Grid.TOP) {
         for (int j = 0; j < columns; j++) {
            dest[remoteTop][j+1] = data[j];
         }
      }
      else if (edge == Grid.BOTTOM) {
         for (int j = 0; j < columns; j++) {
            dest[remoteBottom][j+1] = data[j];
         }
      }
      else if (edge == Grid.LEFT) {
         for (int i = 0; i < rows; i++) {
            dest[i+1][remoteLeft] = data[i];
         }
      }
      else if (edge == Grid.RIGHT) {
         for (int i = 0; i < rows; i++) {
            dest[i+1][remoteRight] = data[i];
         }
      }
      else {
         throw new IllegalArgumentException(
            "saveRemoteEdge: Invalid edge value: " + edge
         );
      }
   }
}
```

Appendix D

Sample Babylon v2.0 Application - Matrix Multiplication

D-1 TestMatrixMult.java

```
package babylon.tests.matmult;

import babylon.core.*;
import babylon.io.*;
import babylon.util.Debug;
import java.util.Date;

public class TestMatrixMult
{
   public static void main(String args[])
   {
      if (args.length < 4) {
         System.err.println("To run the matrix multiply application: ");
         System.err.println(
            "Usage: matmult_test <scheduler> <jar_file> <num servers> <matrix size>");
         System.exit(0);
      }

      // The number of worker objects.
      int nprocs = Integer.parseInt(args[2]);

      // The size of the matrices to be multiplied.
      int size = Integer.parseInt(args[3]);

      System.out.println("Running Matrix Multiplication Test Program...");
      try {
         // Initialize Babylon.
         Babylon.initApplication(args[0], args[1], null);
      }
      catch (Exception e) {
         e.printStackTrace();
         System.exit(0);
      }

      Matrix sub_a[] = new Matrix[nprocs];
      Matrix sub_result[] = new Matrix[nprocs];

      // Allocate space for dynamic proxies.
      MultiplyMatrix mm[] = new MultiplyMatrix[nprocs];

      long startServerT, finishServerT;
      long startExecT, finishExecT;

      int start_row = 0;
      int end_row = -1;
      int nrows = size / nprocs;

      Matrix a = Matrix.createRandomMatrix(size, size);
      Matrix b = Matrix.createIdentityMatrix(size);
      int bInt[][] = b.getArray();
```

```
      // Divide the original matrix into blocks of contiguous rows.
      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;
         }
         sub_a[i] = a.getSubMatrixByRows(start_row, end_row);
      }

      startServerT = System.currentTimeMillis();
      for (int i = 0; i < nprocs; i++) {
         try {
            // Export each server.
            mm[i] = (MultiplyMatrix) Babylon.export(
               new MultiplyMatrixImpl(), MultiplyMatrix.class
            );
         }
         catch (Exception ex) {
            ex.printStackTrace();
            System.exit(0);
         }
      }
      finishServerT = System.currentTimeMillis();

      startExecT = System.currentTimeMillis();

      // Allocate space for the asynchronous tickets.
      MultiplyMatrix mm_asynch[] = new MultiplyMatrix[nprocs];

      for (int i = 0; i < nprocs; i++)
      {
         // Get an asynchronous ticket for the worker object.
         mm_asynch[i] = (MultiplyMatrix) AsynchTicket.newTicket(mm[i]);

         // Invoke multiply() asynchronously.
         mm_asynch[i].multiply(sub_a[i], b);
      }

      for (int i = 0; i < nprocs; i++)
      {
         try {
            // Retrieve the result of the asynchronous invocation from the
            // ticket.
            sub_result[i] = (Matrix)AsynchTicket.getResult(mm_asynch[i]);
         }
         catch (RemoteExecException t) {
            t.printStackTrace();
            System.exit(0);
         }
      }

      finishExecT = System.currentTimeMillis();

      long serverSetupT = finishServerT - startServerT;
      long execTimeT = finishExecT - startExecT;
   }
}
```

D-2 Matrix.java

```
package babylon.tests.matmult;

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

public class Matrix implements Serializable
{
  int rows = 10;
  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 Matrix(int rows, int columns, int initialValue)
   {
      this.rows = rows;
      this.columns = columns;
      m = new int[rows][columns];
      for (int i = 0; i < rows; i++) {
         for (int j = 0; j < columns; j++) {
            m[i][j] = initialValue;
         }
      }
  }

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

  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() % 10);
    return result;
  }

  public Matrix getSubMatrixByRows(int startAtRow, int endAtRow) {
    if (endAtRow < startAtRow)
        return null;
    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 int[][] getRows(int startAtRow, int endAtRow) {
      int nrows = endAtRow - startAtRow + 1;
      int result[][] = new int[nrows][];
      for (int i=0; i<nrows; i++) {
         result[i] = m[i+startAtRow];
      }
      return result;
   }

   public int[][] getArray() {
      return m;
   }
}
```

D-3 MultiplyMatrix.java

```
package babylon.tests.matmult;

public interface MultiplyMatrix
{
   public Matrix multiply(Matrix a, Matrix b);
}
```

D-4 MultiplyMatrixImpl.java

```
package babylon.tests.matmult;

import java.io.Serializable;

public class MultiplyMatrixImpl implements MultiplyMatrix, Serializable
{
  public MultiplyMatrixImpl() {
  }

  public Matrix multiply(Matrix a, Matrix b) {
      int brows, rows, columns;
      Matrix result = null;
      int value;
      if (a.getColumns()== b.getRows()) {
         rows = a.getRows();
         columns = b.getColumns();
         brows = b.getRows();
         result =  new Matrix(rows, columns);
         for (int i = 0; i < rows; 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);
               }
               result.setValue(i, j, value);
            }
         }
      }
      return result;
   }
}
```

Appendix E

Garbage Collection Tuning Parameters

To minimize the overall impact of garbage collection on the experiments described in Chapter 5, several JVM properties and parameters were tuned to reduce the collection frequency and increase available memory. This section discusses these properties and how they were used to improve Babylon server performance in our experiments.

Memory for application objects in the Java 2 JVM is divided into two areas, the young generation and the tenured generation. When the JVM creates a new object, it gets allocated in the young generation. When the young generation fills up, a minor collection is triggered which cleans up the unreachable objects in this generation. After an object survives several minor collections it gets promoted to the tenured generation. When the tenured generation fills up it triggers a major collection which frees the memory used by the unreachable objects in this generation. Minor collections are much more frequent and are normally done quite quickly. Major collections, on the other hand, occur less often but require more time (which can result in longer pauses at application run-time). A complete discussion of garbage collection in the JVM is beyond the scope of this thesis but the reader is referred to [50] for a more detailed explanation.

One of the key factors affecting garbage collection frequency and, consequently, application performance, is the heap size. If the heap size is too small, frequent garbage collections will be triggered. To reduce the collection frequency, Babylon servers are launched with a large initial heap size of 512 MB and a matching maximum heap size (`-ms512m -mx512m`). The large initial heap size ensures that each server will have enough free memory to accommodate the memory requirements for every experiment. Additionally, the JVM does not have to resize the heap at run-time because the initial and maximum heap size are set to the same value at server startup.

Another important memory sizing parameter is the relative size of the tenured generation to the young generation. The default value for this parameter is 2, which sets the tenured generation to twice the size of the young generation. However, if an application creates many short-lived objects, increasing the size of the young generation will decrease the number of minor collections and can increase performance. Furthermore, since most short-lived objects will not survive a minor collection, very few objects get promoted to the tenured generation and thus the number of major collections does not increase significantly. For the Babylon servers in our experiments, this parameter is given a value of 1 (`-XX:NewRatio=1`) essentially setting the the tenured and young generation to the same size. The resulting young generation is 256 MB, which is still large enough to accommodate the memory requirements for every experiment. As a result, we are able to eliminate minor collections during our experiments and use the new generation exclusively for experiment data.

Finally, the distributed garbage collector (DGC) forces a major collection on both the client and server every 60 seconds to ensure that remote objects are cleaned up promptly. However, in our case, this frequency is much too high because remote objects are normally exported at the start of an experiment and should not be garbage collected until the experiment completes. Consequently, the value for this parameter was set to 60 minutes (`-Dsun.rmi.dgc.server.gcInterval=3600000`) which avoided frequent and unnecessary major collections.

Bibliography

[1] The Globus project. http://www-fp.globus.org/details/.

[2] R. J. Allan, D. R. S. Boyd, T. Folkes, C. Greenough, D. Hanlon, R. P. Middleton, and R. A. Sansum. Evaluation of Globus and associated grid middleware. Technical report, Central Laboratory of the Research Councils (CLRC) e-Science Centre, 2001. Available at http://esc.dl.ac.uk/StarterKit/evals.html.

[3] C. Amza, A. L. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, W. Yu, and W. Zwaenepoel. Treadmarks: Shared memory computing on networks of workstations. *IEEE Computer*, 29(2):18–28, February 1996.

[4] I. Attali, D. Caromel, and R. Guider. A step toward automatic distribution of Java programs. In *Fourth International Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS'00)*, pages 141–161. Kluwer Academic Publishers, September 2000.

[5] L. Baduel, F. Baude, and D. Caromel. Efficient, flexible and typed group communications for Java. In *Joint ACM Java Grande - ISCOPE 2002 Conference*, November 2002.

[6] A. Baratloo, M. Karaul, Z. M. Kedem, and P. Wyckoff. Charlotte: Metacomputing on the Web. In *The Ninth International Conference on Parallel and Distributed Computing Systems (PDCS-96)*, October 1996.

[7] A. Baratloo, M. Karaul, Z. M. Kedem, and P. Wyckoff. Charlotte: Metacomputing on the Web. *Future Generation Computer Systems*, 15(5):559–570, October 1999.

[8] F. Baude, D. Caromel, F. Huet, L. Mestre, and J. Vayssiere. Interactive and descriptor-based deployment of object-oriented grid applications. In *The Eleventh IEEE International Symposium on High Performance Distributed Computing (HPDC-11)*, pages 93–102, July 2002.

[9] J. Baumann, F. Hohl, K. Rothermel, M. Schwehm, and M. Straber. Mole 3.0: A middleware for Java-based mobile software agents. In *IFIP/ACM International Conference on Distributed Systems, Platforms and Open Distributed Processing (Middleware'98)*, pages 355–370. Springer Verlag, April 1998.

[10] S. Bouchenak and D. Hagimont. Zero overhead Java thread migration. Technical Report RT-0261, The French National Institute for Research in Computer Science and Control (INRIA), May 2002.

[11] T. Brecht, H. Sandhu, J. Talbot, and M. Shan. ParaWeb: Towards world-wide supercomputing. In *Proceedings of the Seventh ACM SIGOPS European Workshop*, pages 181–188, September 1996.

[12] M. Campione, K. Walrath, A. Huml, and The Tutorial Team. *The Java Tutorial Continued: The Rest of the JDK*. Addison-Wesley, December 1998.

[13] D. Caromel. *Byte Code Engineering Library (BCEL) Manual*. Apache Software Foundation, 2002. Available at http://jakarta.apache.org/bcel/.

[14] B. Christiansen, P. Cappello, M. F. Ionescu, M. O. 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.

[15] O. P. Damani and V. K. Garg. How to recover efficiently and asynchronously when optimism fails. In *IEEE International Conference on Distributed Computing Systems (ICDCS'96)*, pages 108–115, May 1996.

[16] K. E. Falkner, P. D. Coddington, and M. J. Oudshoorn. Implementing asynchronous remote method invocation in Java. In *The Sixth Annual Australasian Conference on Parallel and Real-Time Systems (PART'99)*, pages 22–34. Springer-Verlag, November 1999.

[17] I. Foster. *Designing and Building Parallel Programs: Concepts and Tools for Parallel Software Engineering*. Addison-Wesley Publishing Company, Inc., Massachusetts, 1995.

[18] I. Foster and C. Kesselman. *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann Publishers, July 1998.

[19] I. Foster, C. Kesselman, and S. Tuecke. The anatomy of the grid: Enabling scalable virtual organizations. *The International Journal of High Performance Computing Applications*, 15(3):200–222, Fall 2001.

[20] R. J. Fowler. *Decentralized Object Finding Using Forwarding Addresses*. PhD thesis, University of Washington, Seattle, Washington, December 1985. (Department of Computer Science Technical Report TR85-12-1).

[21] B. Goetz. Design for performance, Part 3: Remote interfaces. *JavaWorld*, March 2001. Available at http://www.javaworld.com/archives/index-jw-03-2001.html.

[22] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification, Second Edition*. Addison Wesley, June 2000.

[23] W. Grosso. *Java RMI*. O'Reilly, October 2001.

[24] B. Haumacher and M. Philippsen. Exploiting object locality in JavaParty, a distributed computing environment for workstation clusters. In *The Ninth Workshop on Compilers for Parallel Computers (CPC2001)*, pages 83–94, June 2001.

[25] F. P. Incropera and D. P. DeWitt. *Introduction to Heat Transfer, Fourth Edition*. John Wiley and Sons, Inc., August 2001.

[26] M. Izatt. Babylon: A Java-based distributed object environment. Master's thesis, York University, Toronto, July 2000.

[27] M. Izatt, P. Chan, and T. Brecht. Ajents: Towards an environment for parallel, distributed and mobile Java applications. *Concurrency: Practice and Experience*, 12(8):667–685, July 2000.

[28] M. B. Juric, I. Rozman, A. P. Stevens, M. Hericko, and S. Nash. Java 2 distributed object models, performance analysis, comparison and optimization. In *Seventh International Conference on Parallel and Distributed Systems (ICPADS'2000)*, pages 239–246. IEEE Computer Society Press, July 2000.

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

[30] R. Lee. *The JNDI Tutorial: Building Directory-Enabled Java Applications*. Addison Wesley, June 2000.

[31] S. Liang and G. Bracha. Dynamic class loading in the Java virtual machine. In *Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'98)*, pages 36–44, October 1998.

[32] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification, Second Edition*. Addison Wesley, 1999.

[33] S. MacDonald, J. Anvik, S. Bromling, J. Schaeffer, D. Szafron, and K. Tan. From patterns to frameworks to parallel programs. *Parallel Computing*, 28(12):1663–1683, 2002.

[34] S. Markus, S. B. Kim, K. Pantazopoulos, A. L. Ocken, E. N. Houstis, P. Wu, S. Weerawarana, and D. Maharry. Performance evaluation of MPI implementations using the parallel ELLPACK PSE. In *The Second MPI Developer's Conference*, pages 162–169. IEEE Computer Society Press, July 1996.

[35] G. McGraw and E. Felten. *Securing Java: Getting Down to Business with Mobile Code*. John Wiley and Sons, January 1999.

[36] M. O. Neary. *Scalability and Fault Tolerance in Global Computing*. PhD thesis, University of California, Santa Barbara, June 2002.

[37] M. O. Neary, A. Phipps, S. Richman, and P. R. Cappello. Javelin 2.0: Java-based parallel computing on the Internet. In *The Sixth International European Parallel Computing Conference (Euro-Par'2000)*, volume 1900, pages 1231–1238, August 2000.

[38] C. H. Papadimitriou and K. Steiglitz. *Combinatorial Optimization: Algorithms and Complexity*. Prentice Hall, January 1982.

[39] M. Philippsen, B. Haumacher, and C. Nester. More efficient serialization and RMI for Java. *Concurrency: Practice and Experience*, 12(7):495–518, May 2000.

[40] M. Philippsen and M. Zenger. JavaParty – transparent remote objects in Java. *Concurrency: Practice and Experience*, 9(11):1225–1242, November 1997.

[41] R. R. Raje, J. I. Williams, and M. Boyles. Asynchronous Remote Method Invocation (ARMI) mechanism for Java. *Concurrency: Practice and Experience*, 9(11):1207–1211, November 1997.

[42] Recursion Software, Inc. *Voyager ORB Developer's Guide, version 4.6*. Recursion Software, 2003. Available at http://www.recursionsw.com/products/voyager/voyager.asp.

[43] T. Sekiguchi, H. Masuhara, and A. Yonezawa. A simple extension of Java language for controllable transparent migration and its portable implementation. In *The Third International Conference on Coordination Models and Languages (Coordination'99)*, pages 211–226, April 1999.

[44] J. Shirazi. *Java Performance Tuning, Second Edition*. O'Reilly and Associates, Inc., California, January 2003.

[45] P. Sridharan, B. Rieken, and L. Peterson. *Advanced Java Networking*. Prentice Hall PTR, 1997.

[46] V. Strumpen and T. L. Casavant. Exploiting communication latency hiding for parallel network computing: Model and analysis. In *International IEEE Conference on Parallel and Distributed Systems (ICPDS'94)*, pages 622–627, December 1994.

[47] Sun Microsystems, Inc. Why are Thread.stop(), Thread.suspend(), Thread.resume() and Runtime.runFinalizersOnExit() deprecated?, 1999. Available at http://java.sun.com/j2se/1.4/docs/guide/misc/threadPrimitiveDeprecation.html.

[48] Sun Microsystems, Inc. Java object serialization specification, 2001. Available at http://java.sun.com/.

[49] Sun Microsystems, Inc. Java2 platform, standard edition, v1.4.0 API specification, 2002. Available at http://java.sun.com/.

[50] Sun Microsystems, Inc. Tuning garbage collection with the 1.4.2 Java virtual machine, 2003. Available at http://java.sun.com/docs/hotspot/gc1.4.2/index.html.

[51] H. Takagi, S. Matsuoka, H. Nakada, S. Sekiguchi, M. Satoh, and U. Nagashima. Ninflet: a migratable parallel objects framework using Java. In *ACM 1998 Workshop on Java for High-Performance Network Computing*, pages 151–159, 1998.

[52] K. Tan, D. Szafron, J. Schaeffer, J. Anvik, and S. MacDonald. Using generative design patterns to generate parallel code for a distributed memory environment. In *Proceedings of ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP 2003)*, pages 203–215, June 2003.

[53] E. Truyen, B. Robben, B. Vanhaute, T. Coninx, W. Joosen, and P. Verbaeten. Portable support for transparent thread migration in Java. In *International Symposium on Agent Systems and Applications/Mobile Agents (ASA/MA'2000)*, pages 29–43, September 2000.

[54] M. T. Vandevoorde and E. S. Roberts. Workcrews: An abstraction for controlling parallelism. *International Journal of Parallel Programming*, 17(4):347–366, August 1988.

[55] B. Venners. *Inside the Java 2.0 Virtual Machine*. McGraw-Hill Osborne Media, January 2000.

[56] G. von Laszewski, I. Foster, J. Gawor, and P. Lane. A Java commodity grid kit. *Concurrency and Computation: Practice and Experience*, 13(8-9):643–662, 2001.

[57] G. von Laszewski, J. Gawor, P. Lane, N. Rehn, and M. Russell. Features of the Java commodity grid kit. *Concurrency and Computation: Practice and Experience (Grid Computing Environments Special Issue)*, 14(13-15):1045–1055, 2001.

[58] G. V. Wilson. *Practical Parallel Programming*. MIT Press, Massachusetts, December 1995.

[59] W. M. Yu and A. Cox. Java/DSM: A platform for heterogeneous computing. *Concurrency: Practice and Experience*, 9(11):1213–1224, November 1997.