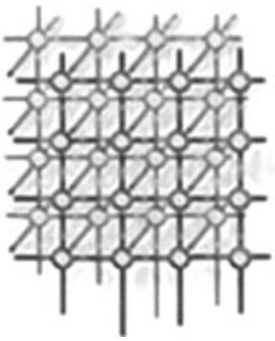
CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
Concurrency Computat.: Pract. Exper. 2008; **20**:1195–1224
Published online 22 January 2008 in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/cpe.1264



Babylon: middleware for distributed, parallel, and mobile Java applications

Willem van Heiningen[1], Steve MacDonald[2] and Tim Brecht[2,*,†]

[1]*Integrative Biology, Hospital for Sick Children, Toronto, Ont., Canada*
[2]*David R. Cheriton School of Computer Science, University of Waterloo, Waterloo, Ont., Canada*

SUMMARY

Babylon is a collection of tools and services that provide a 100% Java-compatible environment for developing, running and managing parallel, distributed and mobile Java applications. It incorporates features such as object migration, asynchronous method invocation, and remote class loading, while providing an easy-to-use interface. Additionally, Babylon enables Java applications to seamlessly create and interact with remote objects, while protecting those objects from other applications by implementing access restrictions and separate namespaces. The implementation of Babylon centers around *dynamic proxies*, a feature first available in Java 1.3, that allow proxy objects to be created at runtime. Dynamic proxies play a key role in achieving the goals of Babylon. The potential cluster computing benefits of the system are demonstrated with experimental results, which show that sequential Java applications can achieve significant performance benefits from using Babylon to parallelize their work across a cluster of workstations. Copyright © 2008 John Wiley & Sons, Ltd.

Received 8 August 2006; Revised 10 July 2007; Accepted 11 July 2007

KEY WORDS: synchronous and asynchronous remote method invocation; distributed middleware; dynamic proxies; object migration; remote class loading

1. INTRODUCTION

The Java language [1] has many features that facilitate distributed systems programming. Java's built-in security, threading, and dynamic class loading support can greatly simplify the development

*Correspondence to: Tim Brecht, David R. Cheriton School of Computer Science, University of Waterloo, 200 University Avenue West, Waterloo, Ont., Canada N2L 3G1.
†E-mail: brecht@cs.uwaterloo.ca

Contract/grant sponsor: The Ontario Research and Development Challenge Fund
Contract/grant sponsor: Natural Sciences and Engineering Research Council of Canada
Contract/grant sponsor: Morgan Stanley Dean Witter

of distributed applications. Furthermore, Java applications are compiled into a machine-independent representation called bytecodes, which can be run on any machine that runs the Java virtual machine (JVM). Java also supports remote method invocation (RMI), which can hide much of the complexity of communication with objects residing in other JVMs, possibly on other machines.

Nevertheless, Java and Java RMI do not include support for many important features for distributed object programming, such as dynamic remote object creation, asynchronous remote method invocation, remote object migration, and remote object administration facilities.

Babylon version 2.0 (simply 'Babylon' for the rest of this paper) overcomes these limitations by providing programmers with Java classes and interfaces for remote object creation, interaction and administration [2–4]. Babylon 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.

- Babylon provides a general programming model for distributed objects that can be used to efficiently implement a wide variety of applications with varying communication and structural needs.
- Babylon uses *dynamic proxy objects* in Java to provide transparent access to remote worker objects. Clients can use these dynamic proxies to invoke methods on worker objects using the standard Java method invocation syntax.
- Babylon uses a novel asynchronous method invocation technique based on *asynchronous tickets*, implemented with dynamic proxies. Method invocations using asynchronous tickets are syntactically identical to local method invocations, but are executed asynchronously. These tickets allow both synchronous and asynchronous remote method invocations to co-exist in an application without requiring special invocation syntax.
- Babylon provides two forms of remote object creation. Clients can create a new remote object instance based only on a programmer-specified class name, or take an existing locally created object and turn it into a remote object. Unlike many existing distributed systems, Babylon also supports nested remote object creation and nested remote object invocation.
- Babylon supports remote class loading, which means that clients do not need login access to the machines that host their remote objects. Unlike many existing distributed systems, Babylon also allows the execution of remote objects from many different clients simultaneously. Furthermore, objects and classes are not shared with other clients unless explicitly made available.
- Babylon supports the migration of idle remote objects between remote hosts. It also provides a novel form of migration called *safe-point migration* that can be used to migrate executing remote objects using a checkpointing and rollback protocol.
- Babylon 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 Babylon 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 and server thread information.

DOI: 10.1002/cpe

Babylon meets its implementation goals by using *dynamic proxies*, a new feature introduced in Java 1.3, that allow proxy objects to be created at runtime. Dynamic proxies can fulfill all of the same requirements as proxies generated by special stub compilers, such as `rmic` for Java RMI, but without the extra compilation tools and steps. These proxies are used to support remote method invocation and remote object creation. An important benefit of using dynamic proxies is that they reduce the requirements needed for a class to produce remote objects compared with standard Java RMI. This makes it possible to create remote objects from classes where source code is not available, including the Java standard class library, provided the remaining requirements are met.

Babylon provides all of these features while maintaining 100% Java compatibility, and can be used on any platform that supports a version of the JVM that is 1.3 or newer. We believe that the performance results and the combination of features provided in Babylon make it a powerful system for distributed application developers.

This paper is primarily a summary of [2] and combines and expands upon the two shorter papers [3,4]. It is organized as follows. Section 2 discusses background and related research. Section 3 describes the overall system architecture of Babylon, and Section 4 describes the important features of Babylon using a two-dimensional heat diffusion program as a running example. The performance results for two applications are presented in Section 5. The paper concludes with Section 6.

2. BACKGROUND AND RELATED WORK

2.1. Related research projects

There are a large number of commercial and academic Java-based distributed computing projects in various stages of development. The objectives and underlying technologies of each of these projects vary significantly. Some focus on the emerging grid, while others are designed for very specialized groups of computational problems.

2.1.1. History of Babylon

Much of the design for the original implementation of Babylon [5], referred to here as Babylon v1.0, came from experiences with Ajents [6] and ParaWeb [7]. Babylon v1.0 built on these systems to provide mechanisms for remote object creation, migration, and remote I/O, but lacks a flexible mechanism to create and interact with remote objects. Babylon v1.0 also suffers from deficiencies in other key areas such as remote class loading and object migration. However, the most serious drawback of Babylon v1.0 is its reliance on a non-standard remote method invocation interface that resembles method invocation with Java reflection. This interface is awkward to use and error prone because it prevents normal compile-time checks, such as invoking a non-existent method, passing an incorrect number of arguments to a method, or passing arguments of the wrong type. Examples of synchronous and asynchronous remote method invocations, using explicit futures [8], in Babylon v1.0 are shown in Figure 1. The second version of Babylon, described in this paper, builds on the strengths of Babylon v1.0 and addresses most of the shortcomings described above.

DOI: 10.1002/cpe

```
 1 try {
 2   // Synchronous call.
 3   reply = (String) Babylon.rmi(
 4     obj,     // remote obj
 5     "ask",   // the method
 6     question // argument
 7   );
 8   // Run when obj.ask(question)
 9   // completes.
10 } catch( // exception ) {
11     // Exception handling
12 }
```

(a)

```
 1 // Asynchronous call - use future in place
 2 // of actual return value.
 3 Future f = (String) Babylon.armi(obj,"ask",
 4                                  question);
 5 // Code concurrent with obj.ask(question)
 6 try {
 7   // Get result.  Blocks until available.
 8   String reply = (String) f.get();
 9 }catch(babylon.core.RemoteExecException e){
10   //  Target method threw an exception.
11 }
```

(b)

Figure 1. Babylon v1.0 remote method invocation, calling `obj.ask(question)`: (a) synchronous remote method invocation and (b) asynchronous remote method invocation.

2.1.2. Programming model and implementation

There are three aspects to invoking methods in a distributed object system:

- the mechanism for invoking remote methods, whether synchronous or asynchronous;
- the problem of distinguishing synchronous and asynchronous method calls; and
- how to obtain the results of asynchronous calls.

We will examine various approaches to each of these aspects.

Method invocation: There are two basic options for invoking methods in a distributed object system. The first is a *reflective interface*, where the user specifies the constituent parts of a method call and the system invokes the correct remote method. The second is a *proxy-based scheme*, where methods are invoked on placeholder objects local to the client. The proxy then invokes the correct remote method. The proxy-based scheme can be further broken down into two options based on how the proxy objects are created. *Static proxies* are generated by a special stub compiler at compile time, based on information supplied by the developer. *Dynamic proxies* are constructed on demand at runtime. As of version 1.3, Java has built-in facilities for creating dynamic proxies, which are discussed in detail in Section 2.2.

Many distributed object systems, especially those predating dynamic proxies in Java, relied on either reflective interfaces or static proxies. For reflective interfaces, the most common approach was to specify the name of the remote method along with an array of arguments [5–7,9–12]. One exception is Reflective RMI [13], which uses descriptor objects to specify the method signature and parameters for a method call in a more structured way. A reflective interface has the advantage of not requiring a stub compiler or remote interfaces, making it simpler to create remote objects. However, it suffers from the disadvantages noted above for Babylon v1.0.

Other systems use statically generated proxies. Since Java has a remote method invocation facility, the existing stub compiler (`rmic`) is normally used. Static proxies require the classes that may create remote objects to be identified during design since it is not feasible to generate stubs for all classes. Further, most static proxy systems require that these classes export a remote interface for clients. Static proxies add an extra step to the compilation process, but in return provide a useful abstraction for calling remote methods. Remote calls are syntactically identical to local calls since they appear

DOI: 10.1002/cpe

as local calls on proxy objects. Furthermore, proxies allow the compiler to verify that such calls are done correctly.

Some new distributed objects use dynamically generated proxies created at runtime. These proxies were used in the implementation of Babylon, RMIX [14], and Java ARMI [15]. Before the advent of dynamic proxies, systems such as ProActive [16,17] implemented their own equivalent facility. Dynamic proxies are attractive because they can reduce the requirements for creating classes that produce remote objects compared with normal Java RMI [4]. They also obviate the need for extra stub compilers and the need to identify classes that can create remote objects at design time.

Babylon uses the dynamic proxy facility in Java to create proxies for remote objects at runtime. This allows Babylon to meet its goals of complete Java compatibility without requiring extra preprocessing; not even the `rmic` stub compiler is needed. In addition, Babylon reduces the set of requirements for creating classes that can produce remote objects compared with normal Java RMI [4]. A class need only implement an interface exporting client methods to produce remote objects. This interface does not include any references to remote methods or exceptions as needed in Java RMI; a standard Java interface for the class is all that is needed. In addition, the class must be serializable if objects are to be migrated or exported. As long as these requirements are met, Babylon can create remote objects from a class even if the code is not available.

Asynchronous method invocations: The basic implementation of asynchronous remote method invocation in Java, launching a synchronous Java RMI in a separate thread and synchronizing with futures, was first described in [18] and has been used extensively since then.

One serious problem that has plagued distributed object systems is allowing both synchronous and asynchronous remote method invocations to co-exist in the same program. The difficulty is in identifying which semantics to apply for a given method call. Reflective systems can simply use a different library method for different calls, as done in Babylon v1.0 (see the `Babylon.rmi()` versus `Babylon.armi()` method calls in Figures 1(a) and (b)).

For systems based on proxies, allowing synchronous and asynchronous remote method invocations to co-exist is more difficult. One strategy is to have both synchronous and asynchronous interfaces for an object [14]. The asynchronous interface can be identified by requiring its name to follow a specific convention, and the method name can even encode more information about the desired method invocation semantics (asynchronous, asynchronous with callback or one-way remote method invocation). It is possible to implement this strategy with dynamic Java proxies, without extra preprocessing stages, by using Java reflection to determine whether the invoked method exists in an asynchronous interface for the remote object [14]. The main weakness with this approach is that the user must be careful to obey the expected naming conventions; errors cannot be caught until runtime. Unfortunately, method signatures change slightly for different interfaces. For example, an asynchronous method must return a future object and not the original return type. This property makes it difficult to get these conventions correct. Other systems use extra keywords or language constructs [15,19–22], which require an extra preprocessor or a new compiler. Yet others use implicit rules, based on the method return type and the presence of checked exceptions, to decide on the semantics to use for a method call [16]. Implicit rules add an extra cognitive burden to the developer, who must now be aware of how these rules will affect the execution of their application code.

DOI: 10.1002/cpe

Babylon uses a novel mechanism called *asynchronous tickets* to allow both forms of remote invocations to co-exist. These tickets are a different type of dynamic Java proxy. Now, the choice of synchronous versus asynchronous method invocation is dictated by the kind of proxy object on which the client invokes the method, as we will explain later. These tickets use dynamic Java proxies, which preserve the benefits of proxies but do not require language extensions, preprocessing tools, or class and interface naming conventions.

Obtaining asynchronous results: Once an asynchronous method is launched, a mechanism is required to enable the programmer to obtain its results, either the return value or a thrown exception.

The most common approach is to have asynchronous methods return a *future* [8], an object that represents the outstanding results of an asynchronous remote method call. This future is returned immediately, before the remote method executes. When the future is accessed (or *resolved*), by calling a method on it, the future checks whether the remote method has returned its results. If not, the method call is blocked until the result is available.

The use of futures introduces a subtlety in application code. The programmer must separate asynchronous calls from accesses to the results to achieve parallelism. This is not a natural style of programming and generally requires that legacy code be modified before it can be used to achieve good performance. However, it is interesting to note that failing to separate asynchronous calls from result accesses introduces only a performance problem and not a correctness problem; the program will run correctly, but with either little speedup or possibly even a slowdown [23].

There are two forms of futures. With *explicit futures*, similar to those used in Babylon, asynchronous methods return a future object in place of the original return type. This future object must be explicitly resolved using a method call to obtain method results. Explicit futures make it difficult to reuse existing code since the return types of methods are changed and require additional resolution code. With *transparent futures*, asynchronous methods appear to return their normal type. However, they instead return a proxy that encapsulates the results. When a method is invoked on the transparent future, the proxy blocks waiting for results. Transparent futures avoid the need for explicit resolution but introduce additional subtleties. For example, a method cannot return `null` in such a system because a future proxy is always returned; a method must always return a result object, where instance variables in that object must then contain any return values [24]. Transparent futures must also respond correctly to identity operators such as `==` and `instanceof` [25]. This is complicated by the fact that a future is a separate object of a different class than the actual return type; hence, these operators may return unexpected results that violate transparency and expose the future to the programmer. An analysis and transformation system is presented in [25] that correctly applies these operators to result objects and not futures, maintaining transparency.

Two other variants of futures are described in [26], called *lazy RMI* and *future-based RMI*, which reduce communication overhead in grid systems. These variants are shown in Figure 2. They create a remote object for method results on the server and return the remote object reference to the client rather than the serialized results. Lazy RMI is intended for synchronous remote method invocation and is illustrated in Figure 2(b). If the method result is simply passed as an argument to a second remote method (i.e. the client does not use the results), then the server for the second call issues a remote method invocation request to the first server to access the results. This saves having to return results to the client only to forward them onto another server. If the two remote servers are close together, lazy RMI can reduce communication costs assuming the remote reference is

DOI: 10.1002/cpe

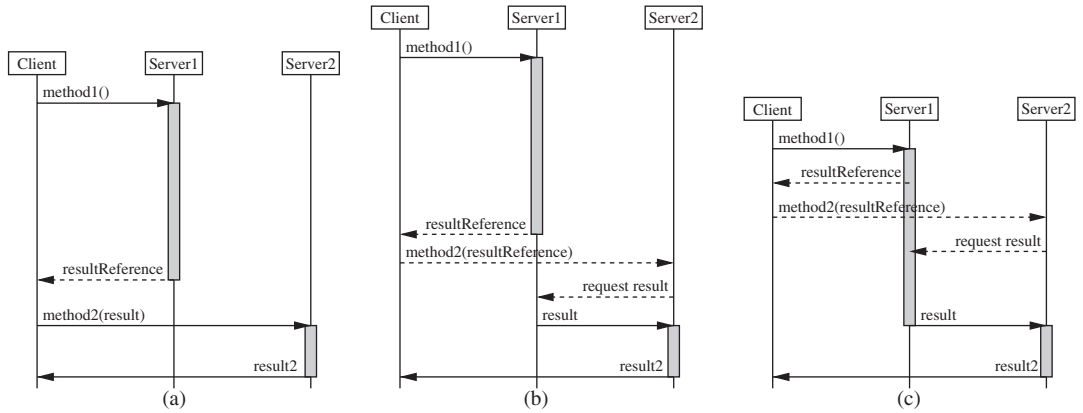


Figure 2. Timing diagrams for remote method invocation variations, from [26]: (a) normal RMI; (b) lazy RMI; and (c) future-based RMI.

smaller than the result object. Future-based RMI adds asynchrony to lazy RMI by creating and returning the remote reference (now to a future) as soon as possible, even before the remote method completes. The future is resolved by the server of the second remote call (or, more generally, by the server that invokes a method on the future) rather than by the client. Future-based RMI is shown in Figure 2(c).

In addition to using futures, *listener objects* are sometimes used to return the results of remote method invocations [13,27]. A one-way remote method invocation is made to the server. When the server completes the method invocation, it makes a one-way remote method invocation back to the client, invoking a method on a listener object registered for the call. This callback method processes the results. The drawback of this approach is that the application code must be reorganized so that the listener object executes any code that requires the results from the remote call, which can make it difficult to incorporate legacy code into the distributed version of the program.

In Java, it is also possible for a method to throw an exception rather than return a result. Handling exceptions thrown in asynchronously called methods is difficult because the client continues to execute. When the exception arrives, the client may no longer have sufficient context to handle it in a meaningful way. With listener objects, a callback method for processing exceptions can be invoked [13]. For futures, there are two general options depending on whether future resolution is explicit (like Figure 1(b)) or implicit (as done in ProActive). For explicit resolution, exceptions are normally rethrown at the client when the future is resolved. For implicit remote method invocation, ProActive uses two strategies. First, methods that throw exceptions are not executed asynchronously by default. Second, the programmer can explicitly run such methods asynchronously, but only within a `try-catch` block that includes extra ProActive calls to force the call to be resolved before the `try` block is exited; hence, the appropriate `catch` block can immediately handle any exceptions. These rules are covered in more detail later in this section.

Babylon uses future objects that must be explicitly resolved. Explicit resolution provides the user with complete control when synchronizing with remote calls, especially for methods that do not have a return value (i.e. `void` methods). At resolution time, any exceptions are thrown.

DOI: 10.1002/cpe

2.1.3. Other related systems

Other research efforts have focused on supporting more specific distributed system structures such as:

- master worker (Javelin [28–30], Ninflet [31], Charlotte [32]);
- branch and bound (Javelin, JICOS [33]);
- divide and conquer (Satin [34]);
- SPMD programming for scientific applications (HP Java [35], Spar [36]); and
- a variation on SPMD, based on distributed collections of objects (ADAJ [10], *Do*! [37]).

Babylon does not limit itself to a particular computational model and provides support for general remote object interaction using the standard Java method invocation syntax. As a result, distributed applications that require more complex object interactions, such as a grid-based heat diffusion computation, can be written using Babylon. In contrast, such an application cannot be easily or efficiently implemented using the aforementioned systems.

ProActive [16,17] provides services similar to Babylon, using similar techniques that also do not require JVM changes or external tools. ProActive generates proxies for remote objects at runtime using bytecode engineering libraries as it predates dynamic proxies in Java. This capability is used to create new remote objects and export local ones. ProActive can execute remote methods synchronously or asynchronously. This choice is not made by the programmer, but rather is based on implicit rules that consider the return type of the method and the presence of checked exceptions. These rules, which are evaluated at runtime, are as follows:

- If the method does not throw a checked exception and the return type is either a class that can be subclassed or `void`, the method is called asynchronously.
- If the method throws a checked exception or the return type is a primitive type or cannot be subclassed (i.e. is final), the method is called synchronously by default.
- In certain cases, it is possible to override the default to execute a method synchronously. Specifically, if the method throws a checked exception and the return type is a class that can be subclassed (or `void`), the user can introduce some concurrency if the method is called within a `try/catch` block. ProActive includes primitives that allow the method to execute asynchronously, but these primitives force the method to complete before the `try` block ends (to ensure any exceptions can be handled). This requires three library calls that must be at specific locations in the application code, and provides limited parallelism. An example of this is shown in Figure 3, with the added code at lines 14, 22, and 27. The figure also documents the requirements on the use of these primitives.

Asynchronous calls in ProActive return a future object, which is a proxy that resolves to a final value when a method is called on it.

All proxy classes generated by ProActive, for both remote objects and futures, are subclasses of the originals that override all public methods. Thus, proxy objects can be substituted for original objects. This provides polymorphism between local and remote objects and between local return values and futures, and obviates the need for interfaces. It also allows future objects to transparently implement wait by necessity. However, it limits the classes that can be used to create remote objects or that can be used as return types in asynchronous remote methods. Final classes and methods, including

DOI: 10.1002/cpe

```
// Method that throws an exception.  ProActive will execute
// this method synchronously by default.
public class MyActiveObject
{
  Object myMethod() throws ApplicationException
  {
    . . .
  }
}

// Call myMethod() asynchronously instead, overriding default.
// This must be called before try block starts, with the
// exceptions that could be thrown.
ProActive.tryWithCatch(ApplicationException.class);
try {
  Object result = activeObject.myMethod();

  // Method runs asynchronously instead.
  continueComputing(result);

  // Wait for myMethod() to complete in case an exception was thrown.
  ProActive.endTryWithCatch();
} catch (ApplicationException ae) {
  handleException(ae);
} finally {
  // Must add this call in a finally block, as first line.
  ProActive.removeTryWithCatch();
}
```

Figure 3. Overriding synchronous method execution for exceptions in ProActive, from examples in [24].

arrays, cannot be used as they cannot be subclassed or overridden. In addition, a remote object can run only one method at a time, which can make some applications difficult to implement efficiently. For example, a grid-based heat diffusion application requires a separate remote method invocation for each iteration to allow edge data to be exchanged, increasing communication costs [3].

In contrast, Babylon uses an explicit approach for selecting synchronous versus asynchronous method invocation and future resolution, providing more control to the user. The choice is made on the basis of the type of dynamic proxy that the client uses. If the proxy is an asynchronous ticket the method call is asynchronous, otherwise it is synchronous. This explicit approach provides the user with more control over the execution of a Babylon program, more than is possible with implicit approaches such as ProActive. Users have complete control over the semantics used for method calls and the timing of future resolution. The downside is that it is more difficult to reuse existing code in a Babylon application. Proxies and asynchronous tickets must be explicitly created and used, and asynchronous tickets must be explicitly resolved.

Fully transparent solutions, such as ProActive, are intended to promote the reuse of existing code, but it has been noted that this is not possible in general [38]. Specifically, even if the asynchronous nature of method calls is fully hidden, it is still necessary to separate method calls from accesses to their results in order to achieve parallelism and improved performance. It is not natural to write sequential programs in this manner; hence, the application code will need to be changed. Further, exception handling in a fully transparent asynchronous system causes problems since

DOI: 10.1002/cpe

the exception can arrive after the caller has exited the `try/catch` block intended to handle the problem. ProActive deals with this problem by running such methods synchronously by default, complicating their rules for method execution. In addition, Babylon uses Java RMI semantics at the server. Each remote method is run in a separate thread. This allows Babylon to use fewer remote messages in the grid-based heat diffusion application, but adds thread synchronization to the Babylon application code [3]. Since concurrency is explicit in the code, the need for thread synchronization is easier to see. In more transparent systems, it may not be obvious which activities are concurrent and which are not, making it more difficult for the user to properly implement synchronization. Since fully transparent solutions are problematic, a *semi-transparent* approach was proposed in [38] and is offered by Babylon.

2.1.4. Other features in distributed object middleware

Some of the research systems noted above include other features to support distributed object programming. While a number of these features can also be found in Babylon some features are not part of the current version, although future versions would benefit from their addition.

ProActive includes a transparent group communication facility, as do the projects based on distributed collections (ADAJ and *Do*!).

Babylon includes three different forms of remote object migration that allow objects to move at runtime, discussed further in Section 4.5. Migration is also included in ProActive, Ninflet and JavaSymphony [12], although these projects do not support all of the forms found in Babylon.

Several of the middleware systems are expanding into the grid domain, where fault tolerance is a key concern. The migration facilities already implemented in Babylon, which also include a checkpointing facility, are an important part of fault tolerance. However, Babylon does not currently include facilities for detecting a failed host and automatically restarting its objects. Such facilities are included in the latest version of Javelin and Charlotte. ProActive now includes a checkpointing and restart facility [39]. However, this facility relies heavily on the asynchronous sequential processes model on which ProActive is based. In particular, the facility assumes that an active object has only one thread associated with it, which serially executes method requests. It is not clear how well this algorithm would generalize to the thread-per-invocation semantics employed by Java RMI (and hence employed by Babylon). The group communication mechanism in ProActive may also be part of a solution to this problem.

Another problem in grid-based systems is describing the architecture of the available computing resources. In particular, it is difficult to describe the geographical distribution of machines, noting which resources are co-located or geographically close (and often have reduced communication costs) versus those that are distant from one another (and often incur increased communication costs). With the grid, the relative capabilities or availability of resources is also useful information. This information is important for managing locality and load balancing. JavaSymphony and HiMM [40] allow users to construct abstract architectural descriptions. These descriptions are then used to improve the scheduling of objects to servers. The Satin system for divide-and-conquer programs uses a work-stealing algorithm that distinguishes between local and remote clusters to efficiently balance the workload in systems with varying communication costs [34].

To date, Babylon has been targeted at applications running on a local cluster and not the wider-area grid. As a result, it has not included these extra facilities.

DOI: 10.1002/cpe

Another important feature of Babylon is the use 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. ProActive also supports such namespaces, following the OSGi platform specifications for distributed Java systems [41]. Many other existing systems, such as JavaParty and Babylon v1.0, do not provide these separate class namespaces for clients. Without a namespace, servers must be restarted each time a client changes any class and no two clients can ever use classes with the same name. Babylon does not assume that users have login access, much less administrative privileges, on server machines. In addition to supporting multiple class namespaces, Babylon and ProActive provide access restrictions for remote objects based on context information transmitted with remote method invocations. Objects and classes can be shared, but only if the user explicitly requests it.

2.1.5. Java RMI improvements

Babylon uses Java RMI in its dynamic proxies. Since all of the details of a remote method call are encapsulated in these proxies, it would be possible to use an alternative remote method invocation implementation without any impact on the user. Several alternatives already exist, which address the weaknesses in the Sun implementation of remote method invocation. One of the largest bottlenecks in Java RMI is the serialization process that converts objects into a sequence of bytes suitable for network transmission. The generic implementation of this process relies on expensive Java reflection, native methods, and data copying. A number of systems devote effort to reducing these costs.

The Manta compiler translates a Java program into native code [42]. As a result, it includes a new implementation of remote method invocation that removes many bottlenecks and supports serialization by generating customized code in each class. Unfortunately, this solution is not suitable for grid environments because it uses a non-standard Java runtime. Based on Manta, Ibis implemented many of the same ideas but in Java, including the use of a bytecode rewriter to insert customized serialization routines [43]. The Ibis implementation of remote method invocation also reduces remote method invocation costs by ensuring that class information is sent to each server only once.

KaRMI is a drop-in replacement for Java RMI [44]. Again, it implements a more streamlined implementation of the remote method invocation classes. Improved serialization is offered to classes implementing the `uka.transport.Transportable` interface. The developer can implement the serialization methods or can use a special tool to generate them.

The Jaguar project uses *Pre-serialized Objects* (PSOs) to reduce serialization costs [45]. The JVM runtime is extended to store PSOs in memory in their serialized format, suitable for transmission without further processing. However, PSOs must be stored in special memory regions called *containers*. A PSO can contain a reference to any object in its JVM. However, references to objects outside the container for a PSO (either a non-PSO object or a PSO in a different container) will not be serialized and transmitted to remote JVMs, much like the way static data are not serialized by standard Java serialization. Thus, managing references will take extra effort to ensure that the complete state for a PSO is stored as PSOs in the same container. Finally, not all objects can be stored as PSOs; only subclasses of `Jaguar.PSO` have that capability.

DOI: 10.1002/cpe

The use of these solutions is counter to the stated goals of Babylon, which is to avoid changing the JVM, adding extra preprocessing stages, or adding extra compilation steps. However, there is nothing that should preclude developers from using these solutions in conjunction with Babylon to improve performance. Babylon could use the serialization optimizations presented in [46]. This work provides a drop-in replacement for the standard Java serialization classes that exploits homogeneity in object types to reduce the size of serialized data without the need for any external tools.

2.2. Dynamic proxies in Java

A proxy is an object that stands in for another, acting in place of the original [47]. The proxy implements the same interface as the original object, which allows the proxy to be used where the original is expected.

Proxies are used in distributed object systems to fulfill the same roles as client-side stubs in remote procedure call systems. The proxies marshal arguments before sending the data to the remote server holding the remote object. In Java RMI, these proxies are generated using the `rmic` stub compiler, which generates class files containing client-side and server-side stub code.

Dynamic proxies were introduced into Java in version 1.3. Given a list of interfaces, it is now possible to construct a proxy at runtime, without the use of extra stub compilers or other tools. This proxy dispatches any methods to an object acting as the *invocation handler*, which processes the method invocations and can add extra functionality if desired.

An example of a dynamic proxy is shown in Figure 4. This proxy simply prints out information before and after a method is called on the original object. The proxies are created using the `newProxyInstance()` method (line 7). The first argument is a class loader, which defines the namespace in which the proxy will reside. The second argument is an array of interfaces that the new proxy will implement. The returned proxy can be safely downcast to any of these interface types. Note that dynamic proxies are constructed for Java interfaces to preserve substitutability without requiring the proxy to be a subclass of an existing class. The last argument is an *invocation handler*. All method calls on the generated proxy are forwarded to the `invoke()` method on this handler (line 17), where the invocation is represented as a `Method` object indicating the called method and an array of `Objects` for the arguments.

3. SYSTEM ARCHITECTURE

Figure 5 illustrates the principal components of the Babylon environment and their associations in a sample scenario. Section 4 contains examples demonstrating some of the features of the system and their use.

A typical Babylon application consists of a client program and one or more remote objects running on Babylon 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.

In Figure 5, 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 servers, they could also have been running in the same server.

DOI: 10.1002/cpe

```
 1 public class MyProxy implements
 2        java.lang.reflect.InvocationHandler {
 3   private Object receiver;
 4
 5   public static Object newProxy(Object obj) {
 6     Class c = obj.getClass();
 7     return java.lang.reflect.Proxy.
 8            newProxyInstance(c.getClassLoader(),
 9                             c.getInterfaces(),
10                             new MyProxy(obj));
11   }
12
13   private MyProxy(Object obj) {
14     this.receiver = obj;
15   }
16
17   public Object invoke(Object proxy, Method m,
18               Object[] args) throws Throwable {
19     Object result;
20     try {
21       System.out.println("Before " + m.getName());
22       result = m.invoke(receiver, args);
23     } catch (InvocationTargetException ite) {
24       // If method threw exception, rethrow.
25       throw e.getTargetException();
26     } catch (Exception e) {
27       // Handle other invocation problems
28     } finally {
29       System.out.println("After " + m.getName());
30     }
31     return(result);
32   }
33 }
```

Figure 4. Example of a dynamic proxy from [48].

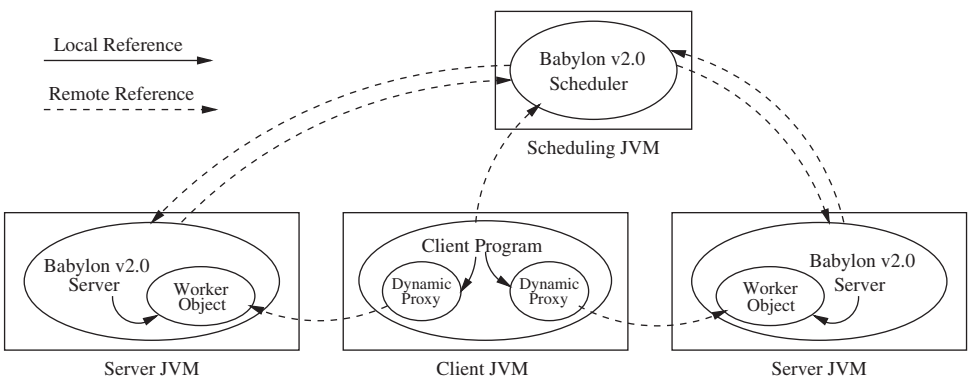


Figure 5. Principal components of the Babylon framework.

DOI: 10.1002/cpe

A Babylon server must be running on each machine that is designed to host worker objects. The server provides a virtual environment for running and managing one or more worker objects and is the point of contact on the remote machine for clients. Once a server is started, any user can create objects on it and can invoke methods on those objects. Users do not require login access to the server machine to use a Babylon server process.

Babylon includes a scheduler component that has two important responsibilities. First, it tracks a list of available Babylon servers. When a new remote worker object is created, the scheduler locates a server that can host it. Second, the scheduler also maintains a name registry service that can be used to look up references to existing worker objects. A Babylon system is bootstrapped using the `rmiregistry` service provided by Java, to allow servers and clients to locate the scheduler. However, this is a basic name service. Babylon provides its own name service to ensure that a client cannot access classes and objects from other clients without permission.

At startup, a server queries the `rmiregistry` service on a specified host to obtain a reference to the scheduler. The server then contacts that 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 supports the presence of multiple servers on a single machine. Babylon 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 and multicore environments.

Babylon relies on Java RMI [49,50] for the underlying communication mechanism. Remote method invocation greatly simplifies the implementation of Babylon by providing built-in support for distributed garbage collection and a simple mechanism to interface with remote components without having to use low-level socket communication. However, Babylon uses dynamic Java proxies [48] rather than the static proxies generated by the `rmic` stub compiler. These proxies are generated at runtime and are used to support object creation and method invocation features not present in Java RMI, as described in later sections.

Babylon facilitates distributed object programming by providing programmers with classes and interfaces for remote object creation, interaction, and administration. Most of the Babylon-distributed object programming primitives are accessible via the static methods of the `babylon.core.Babylon` utility class.

4. BABYLON FEATURES

To demonstrate some of the important features of Babylon, we use portions of code taken from a heat diffusion application. The heat diffusion application 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 [51] 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 neighboring cells using data from iteration $i - 1$.

In our implementation, N worker objects are used to compute the temperature of a surface after 100 iterations. We use a decomposition strategy to partition the original array into N blocks, each

DOI: 10.1002/cpe

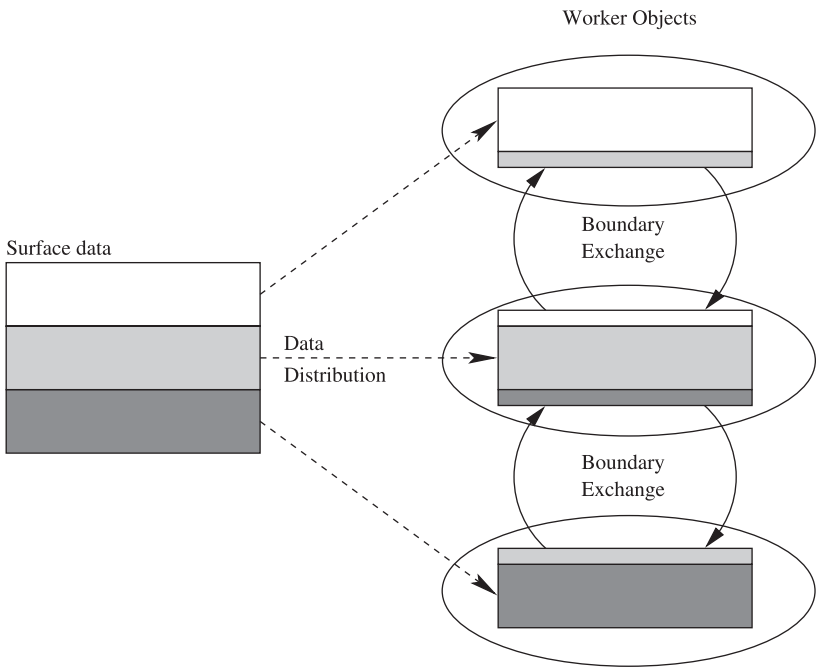


Figure 6. Babylon grid diffusion example—application structure.

consisting of 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 neighboring 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. This structure is shown in Figure 6.

Selected portions of the heat diffusion code appear in Figures 7 and 8. The code in Figure 7 creates worker objects, initializes them, launches the computation and gathers the results. The code in Figure 8 is the main execution loop for each worker in the heat diffusion computation. Each worker repeatedly obtains edge data from adjacent workers and then computes the new temperature for each element. After a fixed number of iterations, the results are returned to the object that launched the computation. We will refer to the code in these two figures for the rest of this section.

4.1. Remote object creation

Remote object creation is the process of creating an instance of an object on a remote server and making this object available to clients. Clients access this new remote object using the returned local proxy object. Babylon provides two methods for creating remote worker objects that can be used by clients to perform distributed operations: `Babylon.remoteNew()` and `Babylon.export()`.

The `remoteNew()` method is illustrated in line 9 in Figure 7, where it is used to create remote worker objects of type `GridDiffuserImpl`, one per processor. The first argument is the class

DOI: 10.1002/cpe

```
 1 public void startHeatDiffusion() {
 2   try {
 3     Babylon.initApplication("schedulerHostName", "Grid.jar", null);
 4   } catch (Exception e) { /* handle exception */ }
 5   GridDiffuser gd[] = new GridDiffuser[nprocs];
 6   Grid grid = new Grid(matrix_size, matrix_size);
 7   try {
 8      for (i = 0; i < nprocs; i++) {
 9        gd[i] = (GridDiffuser) Babylon.remoteNew(GridDiffuserImpl.class,
10                                                 GridDiffuser.class,
11                                                 "GridSection" + i,
12                                                 "Grid.jar");
13      }
14   } catch (Exception e) { /* handle exception */ }
15
16   // Provide references to adjacent grid workers for edge exchange.
17   setAdjacencies(gd, nprocs);
18
19   // Start diffusing... first prepare space for the asynchronous tickets.
20   GridDiffuser gd_asynch[] = new GridDiffuser[nprocs];
21
22   for (i = 0; i < nprocs; i++) {
23     grid_section[i] = grid.getMyRows(i, nprocs);
24
25     // Calling diffuse() asynchronously on each worker object.
26     gd_asynch[i] = (GridDiffuser) AsynchTicket.newTicket(gd[i]);
27     gd_asynch[i].diffuse(grid_section[i], MAX_ITERATIONS);
28   }
29
30   // Now retrieve the results from each worker.
31   float sub_result[][][] = new float[nprocs][][];
32   for (i = 0; i < nprocs; i++) {
33     try {
34       sub_result[i] = (float[][])AsynchTicket.getResult(gd_asynch[i]);
35     } catch (RemoteExecException t) { /* handle exception */ }
36   }
37 }
```

Figure 7. Babylon grid diffusion example—starting the computation.

of the worker implementation, followed by the class exporting the client interface used to construct the dynamic local proxy. The third argument is a user-defined name that is registered with the Babylon name service so that clients can find remote objects. The last argument is a Java Archive (JAR) file containing the code and other files needed for the `GridDiffuserImpl` class since we do not assume that remote servers share a file system. The archive is sent to the machine on which the new remote object is created. This process can be optimized by specifying a JAR file as the second argument to the call to `Babylon.initApplication()` in line 3. This forwards the JAR file to Babylon servers at application startup; hence, `remoteNew()` need not supply the archive. While Figure 7 includes the JAR file in both `initApplication()` and `remoteNew()`, it only needs to be specified in one place. The JAR file is used as a convenient packaging mechanism; future versions of Babylon may use the remote class loading capabilities in Java and may optimize communication by caching class information sent to other Babylon servers so that it need not be

DOI: 10.1002/cpe

```
 1 public void diffuse(float[][] partition, int endIteration) {
 2   float[][] temp;
 3   float[][] srcGrid = initLocalGrid(partition);
 4   float[][] targetGrid = new float[srcGrid.length][srcGrid[0].length];
 5   int current_Iteration = 0;
 6   while (current_iteration < end_iteration) {
 7
 8     // Make sure we have the updated values for the remote edges
 9     // before we compute the diffusion for this iteration.
10     // Block until other worker completes last iteration if necessary.
11     getRemoteEdgesFromAdjacentWorkers(srcGrid);
12
13     // Compute the temperature diffusion for this iteration.
14     diffuse(targetGrid, srcGrid);
15
16     // We're done. Swap the source and target grid.
17     float[][] temp = srcGrid;
18     srcGrid = targetGrid;
19     targetGrid = temp;
20
21     // Update local edge array and iteration. Wake waiting threads.
22     setLocalEdges(++current_iteration);
23   }
24 }
```

Figure 8. Babylon grid diffusion example—main worker code.

```
1 gd[i] = (GridDiffuser) Babylon.export(new GridDiffuserImpl(),
2                                       GridDiffuser.class,
3                                       "GridSection" + i,
4                                       "Grid.jar");
```

Figure 9. Babylon grid diffusion example—exporting local objects.

transmitted multiple times [17]. Further, JAR and class files can be cached at the server and be reused between client applications to further improve performance.

The `export()` method takes a local object and uses it to create and initialize a new remote object that is moved to an available server. Figure 9 shows an example of exporting. This is the object creation code from line 9 in Figure 7, rewritten to export a locally created instance of `GridDiffuserImpl`. The `export()` method uses four parameters: the local object to be exported, an interface implemented by the class of that object (used to construct a dynamic Java proxy for the new remote object), a user-defined name registered with the Babylon name service and the JAR file containing the required class files. Similar to `remoteNew()`, the JAR file can instead be specified in the call to `Babylon.initApplication()` and removed from calls to `export()`.

Both `remoteNew()` and `export()` methods are overloaded to remove arguments that are not needed for each invocation. For example, the name of the remote object and the JAR file are not always needed and may be omitted.

Both methods return a dynamic Java proxy that can be used to transparently invoke methods on the newly created worker object. These proxies provide transparent access to distributed objects by implementing the same interfaces as their distributed counterparts, in this case the interface

DOI: 10.1002/cpe

specified as the second argument to `remoteNew()` or `export()`. Providing transparent access is a significant enhancement over Babylon v1.0, which used a clumsy method invocation interface.

Java RMI does not provide remote object creation. Instead, it relies on one of two methods to create object. The first method is to use *factory objects* [47]. Object servers are started with a factory object loaded, where that object exports methods to create new remote objects. The second method is to use *activatable objects* that are instantiated when invoked. This requires extra code above that already needed to create an RMI-compliant class. An activatable object must extend the class `java.rmi.activation.Activatable` and a setup program must be written to register the object implementation with the `rmiregistry` service. Since both of these methods use Java RMI, both still require the use of the `rmic` stub compiler.

The use of dynamic proxies makes it easier to use Java classes to create remote objects by reducing the requirements on the class compared with Java RMI [4]. In Babylon, a class need satisfy only one requirement: it must implement one or more interfaces that export methods to clients. Unlike Java RMI, the interfaces need not extend the `java.rmi.Remote` interface and the methods do not have to throw `java.rmi.RemoteException`. If the object is to be exported or migrated, the class must also implement the `Serializable` interface. In addition, there are no extra compilation or runtime steps needed to create remote objects. If these requirements are met, then the class can be used to create remote objects even if source code is not available. This could allow objects created from classes in the Java standard class library to be created as remote Babylon worker objects. For example, many of the collection classes in Java meet these requirements and could form remote objects in a Babylon application.

In contrast, Java RMI requires classes to have three characteristics. First, the class must implement the `Serializable` interface and an interface that extends the `java.rmi.Remote` interface that exports methods to clients. Second, all objects must call the `exportObject()` method in the `java.rmi.server.UnicastRemoteObject` class when created, which may throw the checked exception `RemoteException`. This requires all constructors and initialization methods to indicate that the exception may be thrown. This condition is normally met by making the class a subclass of `java.rmi.server.UnicastRemoteObject`. Third, all methods in the interface and implementation must throw `RemoteException`. Only after all of these requirements are met can the `rmic` stub compiler be used to generate static proxies and the class be used in a program that uses remote method invocation. If a class was not written with remote method invocation in mind, then converting the class to create remote objects will require access to the source code, particularly to satisfy the second requirement.

Worker object lookup functionality is also provided with the help of a worker object registry implemented in the scheduler. The registry maintains a record of all worker objects in the Babylon system. Clients can look up references to worker objects based on the instance name of the worker object 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. This feature is not used in our code example. Using the names registered in the call to `remoteNew()` (line 9 in Figure 7) or `export()` (Figure 9), a worker object could use the code in Figure 10 to find its neighbors. In our example, these remote references

DOI: 10.1002/cpe

```
 1 public void findNeighbours(int whoAmI, int numPartitions) {
 2   int above = whoAmI - 1;
 3   int below = whoAmI + 1;
 4
 5   if (above >= 0) {
 6     this.top = (GridDiffuser) Babylon.lookup("GridSection" + above,
 7                                              GridDiffuser.class);
 8   }
 9   if (below < numPartitions) {
10     this.bottom = (GridDiffuser) Babylon.lookup("GridSection" + below,
11                                                 GridDiffuser.class);
12   }
13 }
```

Figure 10. Babylon grid diffusion example—object lookup.

are initialized using a remote mutator method called by the `setAdjacencies()` method (line 17 of Figure 7).

Babylon provides each client with a separate namespace for their objects. This namespace allows different clients to use the same names for their classes and objects without having to worry about naming clashes. Furthermore, the same client can restart their application using newer versions of their classes without problems. In systems without namespaces, the object servers would have to be restarted to prevent a server from reusing older class definitions. The namespace also provides an opportunity to share objects and classes by placing them into a shared portion of the namespace. This sharing is completely under user control as they must explicitly share objects and classes.

4.2. Synchronous remote method invocation

Babylon supports synchronous remote method invocation using the dynamic proxies returned by the remote object creation methods described in Section 4.1 (that is, those proxies returned by the calls to `remoteNew()` and `export()`). These dynamic proxies fulfill the same responsibilities as the static proxies generated by `rmic` in Java RMI. They stand in for the remote objects, allowing programmers to call methods using the normal method invocation syntax. They also permit primitive types to be used as arguments and return types, and allow the method call to be checked at compile time, unlike the syntax used by Babylon v1.0 and Reflective RMI. Similar to their static counterparts, dynamic proxies can be passed as method arguments or returned as method results.

The dynamic proxies in Babylon use Java RMI in their implementation of method invocation. As a result, they inherit the same argument-passing semantics. Remote objects are passed by reference, whereas local objects are passed by value. Also, each method call is run in a separate thread at the receiving server by default, which provides intra-object parallelism within a Babylon server. Programmers need to be mindful of this fact and must take steps to ensure that methods are thread safe.

Dynamic proxies are similar to Java RMI stubs in that they make remote objects available via the Java interfaces they implement, but differ from Java RMI stubs in two important ways. First, dynamic proxies are generated dynamically at runtime instead of using a special stub compiler. Second, the interfaces used by Babylon for worker objects neither need to extend the `java.rmi.Remote`

DOI: 10.1002/cpe

```
 1 public void setAdjacencies(GridDiffuser[] partitions, int nproc) {
 2   // If only one worker, it has no neighbours so do nothing.
 3   if (nproc > 1) {
 4     // Handle top and bottom separately.
 5     // setAdjacentWorkers() is a remote mutator method for workers
 6     // above and below the current one.
 7     gd[0].setAdjacentWorkers(null, gd[1]);
 8     gd[nproc - 1].setAdjacentWorkers(gd[nproc - 2], null);
 9
10     for(int i = 1;i < nproc - 1;++i) {
11       gd[i].setAdjacentWorkers(gd[i - 1], gd[i + 1]);
12     }
13   }
14 }
```

Figure 11. Babylon grid diffusion example—setting adjacencies.

interface, nor do they need to throw `java.rmi.RemoteException`. However, it is important to remember that similar to Java RMI, Babylon worker objects need to implement a client-defined interface and these objects will be accessible to clients only via the methods defined in this interface.

As an example of synchronous method invocation, Figure 11 provides the implementation of the `setAdjacencies()` method used in line 17 in Figure 7. This method calls the remote mutator method `setAdjacentWorkers()` on each worker to provide remote references to neighboring worker objects, which are used during the boundary exchange. These calls, in lines 7, 8 and 11, mirror local method invocations. The proxy objects (instances of `GridDiffuser`) were created using `remoteNew()` (line 9 in Figure 7), so the remote invocations run synchronously.

Synchronous methods can be used to implement the synchronization needed for an application. This synchronization may also exploit the fact that each remote method call is run in a separate thread can use the existing Java thread synchronization methods (noting the subtleties of using thread synchronization within a Java RMI program detailed in [52]). An example of this is the boundary exchange implemented by the worker objects in Figure 8. One difficulty with the exchange of edges stems from the use of Jacobi iteration in the computation. The values used to compute the diffusion in iteration i are those obtained from iteration $i - 1$. A worker requesting remote edges from an adjacent worker must wait until that worker completes its current iteration. This synchronization is accomplished using the thread synchronization facilities in Java. The worker calls a remote accessor method on a neighbor to retrieve the edge values. This accessor blocks the calling thread until the edges are updated. Since the method call is synchronous, the remote client is also blocked. Once the current iteration is complete, these blocked threads are woken (in line 22 of Figure 8) and return the edge data, allowing adjacent workers to proceed with their next iteration.

Another Babylon feature is stateful remote method invocations. Each remote method invocation in Babylon 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 only the client who can invoke methods on that object.

DOI: 10.1002/cpe

4.3. Asynchronous remote method invocation

Network latency in a distributed application environment can incur significant overhead and reduce overall application performance. One way of reducing the impact of network latency is to overlap communication and computation [53]. 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 uses a novel technique based on proxy objects called *asynchronous tickets* to support asynchronous remote method invocation. An asynchronous ticket is created using an existing synchronous proxy object. The ticket is a different type of dynamic proxy that 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 executes.

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 remote method invocation parameter-passing semantics. The return value of the method can be requested from the ticket at a later time.

Babylon 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. Any exceptions that were thrown during the execution of the asynchronous method are thrown when the return value is resolved.

The workers in our heat diffusion application are started with asynchronous tickets to allow all of them to work in parallel. Asynchronous tickets for the workers are created in line 26 in Figure 7. These proxies, not those created with `remoteNew()`, are then used to call the `diffuse()` method in line 27. After the loop exits, all of the worker objects are executing and the `startHeatDiffusion()` method continues execution. To obtain results from asynchronous tickets, the `AsynchTicket.getResult()` method is used, with the ticket as a parameter (line 34). Calls to the `getResult()` method must be enclosed in a `try-catch` block that handles any exceptions that may be thrown by the method invoked on the ticket.

Previous work has considered mechanisms for implementing asynchronous remote method invocations [15,19,22]. These approaches either introduce new keywords and then utilize a preprocessor or require the use of a modified stub compiler. In contrast, our asynchronous tickets are completely compatible with standard Java, compilers and runtime systems, and do not require any preprocessing or a modified stub compiler. Dynamic Java proxies provide a way for both synchronous and asynchronous remote method invocations to co-exist in the same application without such modifications. Furthermore, both synchronous and asynchronous remote method invocations are indistinguishable from local method calls; the only difference is the type of object on which the method is called (local object, synchronous proxy, asynchronous ticket). Otherwise, the syntax mirrors local method invocations.

DOI: 10.1002/cpe

Babylon explicitly distinguishes between synchronous and asynchronous method calls using asynchronous tickets. This feature provides the user with control over how methods are invoked, rather than requiring users to program to implicit runtime rules as with ProActive. Furthermore, asynchronous tickets provide control over synchronization with asynchronous methods. Specifically, ProActive asynchronously executes methods with a return type of `void` (unless they throw a checked exception), but these methods produce no future and so synchronization is not possible. For methods with side effects on the state of a remote object, these semantics may not be desired. Although additional code is needed, Babylon fully supports this synchronization.

4.4. Remote class loading

Unlike most programming languages, Java provides a very flexible class loading mechanism that finds and loads classes at runtime only when they are actually needed [54,55]. Normally, the virtual machine looks for class data as class files that reside in the file system. However, developers can customize how the virtual machine finds and loads classes by implementing their own custom class loaders.

Babylon 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 either when the client creates the worker object or when a Babylon program is initialized. The JAR file is transmitted to the target server along with the remote object creation request. As a result, Babylon servers do not need local file system 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. The JAR file is a convenient package for this transmission; individual classes or files could also be transmitted.

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. Furthermore, JAR files are compressed, reducing the size of the data transmitted between client and server.

To support multiple clients without requiring server restarts between successive client connections, Babylon 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 many of the class-loading issues experienced by systems such as ProActive, JavaParty and Babylon v1.0. Some of the advantages of Babylon class loading are the following:

- Different clients can safely use identical names for their classes without causing naming conflicts.
- To load and use the new class definitions, clients can change their classes and simply restart their application. Without namespaces, the Babylon server processes would need to be restarted each time the application is run. This is important since we do not assume users have login access to servers, much less administrative privileges. We envision the provisioning of server clusters that can be readily used by anyone with an application written using Babylon.
- When a client application has completed, the class loader and the classes loaded by the client can be garbage collected. This allows servers to run for long periods of time as unneeded classes will not consume memory.

DOI: 10.1002/cpe

- Clients no longer share a single class namespace and, consequently, no longer have access to classes created by other users by default. Classes must be explicitly made available to other users if desired.

4.5. Object migration

A key feature of Babylon 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). Migration may be done manually. For example, a system administrator could migrate objects off a machine, upgrade the software or hardware, and move the object back onto the machine. It could also be done automatically by the Babylon scheduler by noticing that faster machines have become available and migrating long-running jobs from slower machines to faster machines. Defining and comparing scheduling policies for making decisions about automatic migration are the subject of ongoing work.

Worker object migration is performed by taking a snapshot of a worker object's data state, known as a checkpoint, and transmitting this checkpoint to a new Babylon server. Consequently, only objects that are serializable [56] can be migrated.

Babylon stores the most current worker object location information in a worker object registry in the Babylon scheduler. In addition, *location forwarding* is used [57]. Forwarding information for a mobile object is stored at each of the object's former locations, creating a chain leading from the original location of the object to its current location. Stale worker references are updated transparently using this information. The combination of these two mechanisms ensures that calls to a migrated object cannot be lost.

Babylon supports three types of migration: idle migration, delayed migration, and safe-point migration. Idle migration takes an idle worker object (one that is not executing any method) and moves it to a new server immediately. If a worker object is actively executing one or more methods at the time migration is requested, then delayed or safe-point migration can be used. Delayed migration prevents new method invocations from starting while allowing in-progress methods to complete. Once the in-progress methods complete and the object becomes idle, it is migrated along with the queue of delayed method invocations.

Safe-point migration uses a checkpointing and rollback protocol to perform migration at programmer-specified safe migration points. Safe-point migration is supported only for worker objects running in *safe mode*. First, 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. Second, 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.

Worker objects that require the ability to be immediately migrated using safe-point migration 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

DOI: 10.1002/cpe

the worker object code, this exception will propagate up to the server. When the server catches the exception, it knows that the 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 are completed. Another approach would be to provide a listener object in the call to `setMigrationPoint()`, which could provide a method to perform the required cleanup if migration is pending. This would eliminate the possibility of incorrectly handling the `BabylonThreadDeath` exception.

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 [58] 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.

4.6. Remote I/O

Babylon 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 remote method invocation. Babylon provides wrapper classes for many standard Java I/O classes. These wrapper classes are essentially remote method invocation 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. Babylon 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.

5. SYSTEM EVALUATION

In this section, we examine the performance of a simple remote method invocation micro-benchmark and two parallel application benchmarks, matrix multiply and heat diffusion. The experiments were conducted using 16 Intel Xeon-based servers. Seven of these machines contain 2.8 GHz Xeon processors, whereas the remaining nine contain 2.4 GHz Xeon processors. All hosts contain 512 KB of L2 cache, 1 GB of memory, Intel e1000 gigabit Ethernet cards, and are connected with a 24 port HP Procurve 2824 gigabit switch. All experiments were conducted using version 1.5.1 of the Java Runtime Environment, and all systems were running Linux. Babylon requires only JVM support for dynamic proxies, which is standard starting from version 1.3.

DOI: 10.1002/cpe

Table I. Remote method invocation performance (ms).

	Argument size (integers)						
Application	1	1k	10k	100k	1M	10M	100M
Babylon RMI (ms)	0.59	0.64	1.42	7.02	58.91	590.41	5786.02
JDK RMI (ms)	0.40	0.44	1.22	6.67	58.45	575.44	5672.25

5.1. Remote method invocation

During the implementation of Babylon, a significant rewrite of Babylon v1.0 RMI 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 an effort was made to minimize the overall remote method invocation overhead and to optimize the transmission of invocation context information.

Table I presents performance measurements obtained using Babylon remote method invocation and the standard remote method invocation facilities provided by the JDK. In each experiment, we perform several (from 10 to 100 000 depending on the size of the argument object) remote method invocations on a worker object and compute the average amount of time it takes to perform a single method invocation. 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 1000 `int`s, '10k' represents an array containing 10 000 `int`s, etc.) and the return value is an integer indicating the size of the array. All timings were conducted between 2.8 GHz machines and 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.

The results in Table I compare the average time of a single remote method invocation using Babylon with that of standard Java remote method invocation. The results show that while the additional layers of indirection present in the Babylon framework do introduce a small overhead, the difference in performance becomes insignificant relative to the total invocation time as the size of the argument grows. In a realistic coarse-grained Babylon application (which is what Babylon is designed to support), this performance difference will have little, if any, impact on the overall application performance.

5.2. Parallel application benchmarks

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 subproblems that can be solved independently by worker objects. The participating workers do not need to communicate with each other to synchronize or share data. A distributed implementation of the matrix multiplication benchmark is used to evaluate the performance of a typical master–worker computation using Babylon. The sequential baseline implementation is a simple, standalone matrix multiplication application (without remote method

DOI: 10.1002/cpe

invocation or domain partitioning) that uses the same core matrix multiplication algorithm as the distributed implementation.

The heat diffusion benchmark was used to evaluate the performance of a communication-intensive Babylon application that could not have been efficiently realized using the master–worker computation model. As a result, it could not have been easily written or executed using many existing distributed systems (e.g., Babylon v1.0, Javelin, Ninflet and Charlotte) since they do not support the more general programming model required to implement this type of application. Javelin implements an efficient parallel branch-and-bound structure based on the master–worker structure with work stealing, which does not match the structure of the heat diffusion program. Charlotte is based on a fork/join style of parallelism with distributed shared memory provided between spawned processes. However, updates to the shared memory regions are visible only to the main process on a join operation; hence, every iteration of heat diffusion would require a new set of processes to be forked, which may be expensive. Ninflet, also based on the master–worker structure, does not appear to have a mechanism that allows workers to communicate with each other. The boundary exchange between workers would have to be implemented with intermediary objects, which would also have to implement the necessary synchronization, to avoid making each iteration a separate remote method call. A ProActive version of the heat diffusion application would suffer from the limitation that only one method can be running in a remote object at a given time. This means the edges cannot be exchanged as shown in Figure 8, since the remote accessor method cannot execute as long as the `diffuse()` method runs. Instead, each iteration of the diffusion must be a separate remote method call, making the code awkward and increasing communication costs. In contrast, more general systems like Babylon and even Java RMI can take advantage of thread synchronization facilities to help synchronize the processes involved in heat diffusion, as explained in Section 4.2. The sequential baseline implementation is a simple, standalone diffusion application (without remote method invocation or domain partitioning) that uses the Jacobi iterative algorithm to compute the final temperature distribution across the surface.

The speedup results for the matrix multiplication and heat diffusion benchmarks are summarized in Figure 12. Each machine runs a single Babylon server; each server contains a single worker object; and all objects run in parallel. Speedups are computed by comparing against the sequential version of each application while executing on the 2.8 GHz machine. The straight dotted line represents perfect speedup.

The speedups obtained for the 2048×2048 matrix multiplication experiments are quite good. With a 2048×2048 matrix, the granularity of the computation can be kept large enough and the network is fast enough to provide good speedup (14.8 on 16 processors).

To validate our speedup results, we conducted a detailed analysis of data distribution costs incurred by the multiplication of two 2048×2048 matrices and compare our speedup with an upper bound computed using Amdahl's Law. The communication costs of distributing the matrix data to the worker objects increase proportionally with N. This is primarily because the entire matrix B must be transmitted 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, there can be a significant amount of data which may have a considerable impact on the resulting speedup.

DOI: 10.1002/cpe

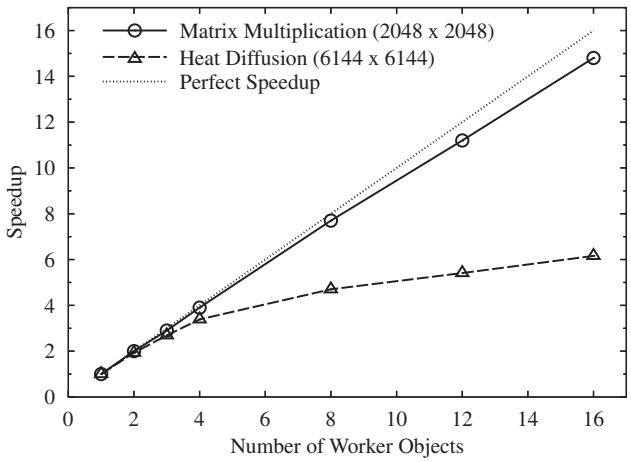


Figure 12. System evaluation results.

The total amount of network traffic generated with N worker objects for a b byte matrix ($b = 16.0$ MB for a 2048×2048 matrix) is $(A + B + C) \times N$, where $A = b/N$, $B = b$ and $C = b/N$. In the case of 16 worker objects 288 MB of data is transmitted. The process of distributing this large amount of data from the master to the worker objects causes the algorithm efficiency to drop slightly for configurations using 16 workers.

To explore the impact of data distribution on matrix multiplication performance more explicitly, we measured the raw matrix data transmission time by calling a dummy matrix multiplication routine on each worker object, which simply returned the matrix to be multiplied. Using 16 worker objects we measure the transmission time to be 2537 ms.

The total time to actually perform the matrix multiplication (including transmission overheads and computation time) is 56 476 ms. The total sequential execution time was 838 998 ms. Using the time to perform only the communication and the total sequential execution time, we can obtain the fraction of the sequential computation that is not executed in parallel ($f = 0.003030$).

The data transmission time enables us to compute an estimate of the fraction of our computation that is executed sequentially. This fraction ($f = 0.003030$) is computed by dividing the matrix transmission time 2537 ms by the total sequential computation time 838 998 ms. On the basis of f, we compute an upper bound on the possible speedup using Amdahl's Law [59]. In this case an upper bound on the speedup is 15.3, which is quite close to the actual speedup of 14.8.

Although the speedup values obtained in the heat diffusion benchmark are smaller than those obtained in the matrix multiplication benchmark, the results are still promising and indicate that speedup can be achieved despite the communication-intensive nature of the problem. In fact, similar experiments in [46,60,61] yield speedups in the range of 2.5–4.9 on eight processors and 3.5–6.5 on 16 processors. Speedups obtained using Babylon are 4.3 using eight worker objects and 5.7 using 16 worker objects. This suggests that Babylon can be just as effective at running communication-intensive applications as other conventional systems.

DOI: 10.1002/cpe

6. CONCLUSIONS

Babylon incorporates features such as remote object creation and migration, remote object access restrictions, separate namespaces for clients, and dynamic class loading while providing an easy-to-use interface that works seamlessly with existing Java code. Worker objects created using Babylon can be accessed transparently using dynamic proxy objects. Babylon also introduces a novel asynchronous method invocation technique based on proxy objects called asynchronous tickets. The overall result is a unique and powerful system that gives developers the necessary tools and services for building powerful cluster computing applications.

Babylon provides all of its features without requiring special preprocessors or extensions to either the JVM or the Java language. Although the system uses Java RMI, it does not require even the `rmic` stub compiler. Instead, it relies on the dynamic proxy facility that is now standard in Java. Another result of using dynamic proxies is a reduction in the requirements for a class to create remote objects. Babylon only requires the class to be serializable and implement some interface that exports methods for clients. If these conditions are met, a class can create remote objects even if the developer does not have the source code.

The performance evaluation results are promising and indicate that applications can use Babylon to distribute objects across multiple hosts in order to execute quite efficiently in parallel. Experiments show that reasonable speedups can be obtained both for 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 can be used effectively to build and run clustered computing applications.

ACKNOWLEDGEMENTS

The authors thank Matthew Kennedy for helping us to get Babylon and our applications running on our experimental testbed. We gratefully acknowledge Morgan Stanley Dean Witter, the Ontario Research and Development Challenge Fund, and the Natural Sciences and Engineering Research Council of Canada for financial support for portions of this project. This paper has also benefitted from the comments and suggestions of the anonymous reviewers.

REFERENCES

1. Gosling J, Joy B, Steele G, Bracha G. *The Java Language Specification* (2nd edn). Addison-Wesley: Reading, MA, 2000.
2. van Heiningen W. Babylon v2.0: Support for distributed parallel and mobile Java applications. *Master's Thesis*, David R. Cheriton School of Computer Science, University of Waterloo, Waterloo, Ont., August 2003.
3. van Heiningen W, Brecht T, MacDonald S. Babylon v2.0: Middleware for distributed, parallel, and mobile Java applications. *Proceedings of the 11th International Workshop on High-level Parallel Programming Models and Supportive Environments*. IEEE Computer Society Press: Los Alamitos, CA, 2006.
4. van Heiningen W, Brecht T, MacDonald S. Exploiting dynamic proxies in middleware for distributed, parallel, and mobile Java applications. *Proceedings of the 8th International Workshop on Java for Parallel and Distributed Computing*. IEEE Computer Society Press: Loss Alamitos, CA, 2006.
5. Izatt M. Babylon: A Java-based distributed object environment. *Master's Thesis*, York University, Toronto, July 2000.
6. Izatt M, Chan P, Brecht T. Ajents: Towards an environment for parallel, distributed and mobile Java applications. *Concurrency*: *Practice and Experience* 2000; **12**(8):667–685.

DOI: 10.1002/cpe

7. Brecht T, Sandhu H, Talbot J, Shan M. ParaWeb: Towards world-wide supercomputing. *Proceedings of the Seventh ACM SIGOPS European Workshop*. ACM Press: New York, September 1996; 181–188.
8. Halstead R. Multilisp: A language for concurrent symbolic computation. *ACM Transactions on Programming Languages and Systems* 1985; **7**(4):501–538.
9. Chaumette S, Vignéras P. A framework for seamlessly making object-oriented applications distributed. *Proceedings of Parallel Computing 2003*. Elsevier: North-Holland, Amsterdam, 2003; 305–312.
10. Felea V, Toursel B. Methodology for Java distributed and parallel programming using distributed collections. *Proceedings of the 4th Workshop on Java for Parallel and Distributed Computing*. IEEE Computer Society Press: Los Alamitos, CA, 2002.
11. Gilmore S, Palomino M. Babylonlite: Improvements to a Java-based distributed object system. *Proceedings of the 4th CaberNet Plenary Workshop*, 2001; 1–4.
12. Fahringer T, Jugravu A. JavaSymphony: A new programming paradigm to control and synchronize locality, parallelism and load balancing for parallel and distributed computing. *Concurrency and Computation*: *Practice and Experience* 2005; **17**(7–8):1005–1025.
13. Thiruvathukal G, Thomas L, Korczynski T. Reflective remote method invocation. *Concurrency*: *Practice and Experience* 1998; **10**(11–13):911–925.
14. Kurzyniec D, Sunderam V. Semantic aspects of asynchronous RMI: The RMIX approach. *Proceedings of the 6th International Workshop on Java for Parallel and Distributed Computing*. IEEE Computer Society Press: Los Alamitos, CA, 2004.
15. Taveira W, de Oliveira Valente M, da Silva Bigonha M, da Silva Bigonha R. Asynchronous remote method invocation in Java. *Journal of Universal Computer Science* 2003; **9**(8):761–775.
16. Attali I, Caromel D, Guider R. A step toward automatic distribution of Java programs. *Fourth International Conference on Formal Methods for Open Object-based Distributed Systems*. Kluwer: Amsterdam, The Netherlands, September 2000; 141–161.
17. Huet F, Caromel D, Bal H. A high performance Java middleware with a real application. *Proceedings of the Supercomputing Conference*. IEEE Computer Society Press: Los Alamitos, CA, 2004.
18. Raje R, Williams J, Boyles M. Asynchronous remote method invocation (ARMI) mechanism for Java. *Concurrency*: *Practice and Experience* 1997; **9**(11):1207–1211.
19. Raje R, Williams J, Boyles M. An asynchronous remote method invocation (ARMI) mechanism for Java. *ACM Workshop on Java for Science and Engineering Computation*. ACM Press: New York, June 1997.
20. Keen A, Ge T, Maris J, Olsson R. JR: Flexible distributed programming in an extended Java. *ACM Transactions on Programming Languages and Systems* 2004; **26**(3):578–608.
21. Haumacher B, Philippsen M. Exploiting object locality in JavaParty, a distributed computing environment for workstation clusters. *Proceedings of the Ninth Workshop on Compilers for Parallel Computers*, June 2001; 83–94.
22. Falkner K, Coddington P, Oudshoorn M. Implementing asynchronous remote method invocation in Java. *Proceedings of the Parallel and Real-time Systems Conference*. Springer-Verlag: Berlin, November 1999.
23. Szafron D, Schaeffer J. An experiment to measure the usability of parallel programming systems. *Concurrency*: *Practice and Experience* 1996; **8**(2):147–166.
24. ProActive. *ProActive On-line Documentation*. INRIA Sophia Antipolis, 2005. http://proactive.inria.fr/release-doc/html/index.html [15 January 2008].
25. Pratikakis P, Spacco J, Hicks M. Transparent proxies for Java futures. *Proceedings of the 19th Annual ACM Conference on Object-oriented Programming Systems*, *Languages*, *and Applications*. ACM Press: New York, 2004; 206–223.
26. Alt M, Gorlatch S. Adapting Java RMI for grid computing. *Future Generation Computer Systems* 2005; **21**(5):699–707.
27. Sysala T, Janeček J. Optimizing remote method invocation in Java. *Proceedings of the 13th International Workshop on Database and Expert Systems Applications*. IEEE Computer Society Press: Los Alamitos, CA, 2002; 29–35.
28. Neary M, Cappello P. Advanced eager scheduling for Java-based adaptive parallel computing. *Concurrency and Computation*: *Practice and Experience* 2005; **17**(7–8):797–819.
29. Neary MO, Phipps A, Richman S, Cappello PR. Javelin 2.0: Java-based parallel computing on the Internet. *Proceedings of the Sixth International European Parallel Computing Conference* (*Lecture Notes in Computer Science*, vol. 1900), Springer: Berlin, 2000; 1231–1238.
30. Christiansen B, Cappello P, Ionescu M, Neary M, Schauser K, Wu D. Javelin: Internet-based parallel computing using Java. *ACM 1997 Workshop on Java for Science and Engineering Computation*. ACM Press: New York, June 1997.
31. Takagi H, Matsuoka S, Nakada H, Sekiguchi S, Satoh M, Nagashima U. Ninflet: A migratable parallel objects framework using Java. *ACM 1998 Workshop on Java for High-performance Network Computing*. ACM Press: New York, 1998; 151–159.
32. Baratloo A, Karaul M, Kedem ZM, Wyckoff P. Charlotte: Metacomputing on the web. *Future Generation Computer Systems* 1999; **15**(5):559–570.
33. Cappello P, Coakley C JICOS: A Java-centric network computing service. *Proceedings of the 17th IASTED International Conference on Parallel and Distributed Computing and Systems*. ACTA Press: Calgary, Alberta, Canada, 2005; 510–515.

34. van Nieuwpoort R, Maassen J, Kielmann T, Bal H. Satin: Simple and efficient Java-based grid programming. *Scalable Computing*: *Practice and Experience* 2005; **6**(3):19–32.
35. Lee H, Carpenter B, Fox G, Lim S. HP Java: Programming support for high-performance grid-enabled applications. *Parallel Algorithms and Applications* 2004; **19**(2–3):175–193.
36. van Reeuwijk C, Kuijlman F, Sips H. Spar: A set of extensions to Java for scientific computation. *Concurrency and Computation*: *Practice and Experience* 2003; **15**(3–5):277–297.
37. Launay P, Pazat JL. Easing parallel programming for clusters with Java. *Future Generation Computer Systems* 2001; **18**(2):253–263.
38. Vignéras P. Transparency and asynchronous method invocation. *On the Move to Meaningful Internet Systems 2005*: *CoopIs*, *DOA*, *and ODBASE* (*Lecture Notes in Computer Science*, vol. 3760). Springer: Berlin, 2005; 750–762.
39. Baude F, Caromel D, Delbé C, Henrio L. A hybrid message logging-CIC protocol for constrained checkpointability. *Proceedings of Europar 2005* (*Lecture Notes in Computer Science*, vol. 3648). Springer: Berlin, 2005; 644–653.
40. Di Santo M, Frattolillo F, Ranaldo N, Russo W, Zimeo E. Programming metasystems with active objects. *Proceedings of the 5th International Workshop on Java for Parallel and Distributed Computing*. IEEE Computer Society Press: Los Alamitos, CA, 2003.
41. OSGi Alliance. *OSGi Service Platform*, *Core Specification*, *Release 4*, 2005. http://www.osgi.org [15 January 2008].
42. Maassen J, van Nieuwpoort R, Veldema R, Bal H, Kielmann T, Jacobs C, Hofman R. Efficient Java RMI for parallel programming. *ACM Transactions on Programming Languages and Systems* 2001; **23**(6):747–775.
43. van Nieuwpoort R, Maassen J, Wrzesinska G, Hofman R, Jacobs C, Kielmann T, Bal H. Ibis: A flexible efficient Java based grid programming environment. *Concurrency and Computation*: *Practice and Experience* 2005; **17**(7–8):1079–1107.
44. Philippsen M, Haumacher B, Nester C. More efficient serialization and RMI for Java. *Concurrency*: *Practice and Experience* 2000; **12**(7):495–518.
45. Welsh M, Culler D. Jaguar: Enabling efficient communication I/O in Java. *Concurrency*: *Practice and Experience* 2000; **12**(7):519–538.
46. Tan K, Szafron D, Schaeffer J, Anvik J, MacDonald S. Using generative design patterns to generate parallel code for a distributed memory environment. *Proceedings of the Ninth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. ACM Press: New York, 2003; 203–215.
47. Gamma E, Helm R, Johnson R, Vlissides J. *Design Patterns*: *Elements of Reusable Object-oriented Software*. Addison-Wesley: Reading, MA, 1994.
48. Sun Microsystems, Inc. Dynamic proxy classes, 1999. http://java.sun.com/j2se/1.3/docs/guide/reflection/proxy.html [15 January 2008].
49. Grosso W. *Java RMI*. O'Reilly: Cambridge, Massachusetts, 2001.
50. Sridharan P, Rieken B, Peterson L. *Advanced Java Networking*. Prentice-Hall: Englewood Cliffs, NJ, 1997.
51. Incropera FP, DeWitt DP. *Introduction to Heat Transfer* (4th edn). Wiley: New York, 2001.
52. Haumacher B, Moschny T, Reuter J, Tichy W. Transparent distributed threads for Java. *Proceedings of the 5th International Workshop on Java for Parallel and Distributed Computing*. IEEE Computer Society Press: Los Alamitos, CA, 2003.
53. Strumpen V, Casavant TL. Exploiting communication latency hiding for parallel network computing: Model and analysis. *International IEEE Conference on Parallel and Distributed Systems*. IEEE Computer Society Press: Los Alamitos, CA, December 1994; 622–627.
54. Lindholm T, Yellin F. *The Java Virtual Machine Specification* (2nd edn). Addison-Wesley: Reading, MA, 1999.
55. Venners B. *Inside the Java 2.0 Virtual Machine*. McGraw-Hill: New York, January 2000.
56. Sun Microsystems, Inc. Java object serialization specification, 2001. http://java.sun.com/ [15 January 2008].
57. Fowler RJ. Decentralized object finding using forwarding addresses. *PhD Thesis*, University of Washington, Seattle, WA, December 1985 (Department of Computer Science, *Technical Report TR85-12-1*).
58. Damani OP, Garg VK. How to recover efficiently and asynchronously when optimism fails. *IEEE International Conference on Distributed Computing Systems*. IEEE Computer Society Press: Los Alamitos, CA, May 1996; 108–115.
59. Amdahl G. Validity of the single processor approach to achieving large-scale computing capabilities. *Proceedings of the AFIPS 1967 Joint Computer Conference*, vol. 30. AFIPS Press: Reston, Virginia, 1967; 483–485.
60. Markus S, Kim SB, Pantazopoulos K, Ocken AL, Houstis EN, Wu P, Weerawarana S, Maharry D. Performance evaluation of MPI implementations using the parallel ELLPACK PSE. *The Second MPI Developer's Conference*. IEEE Computer Society Press: Silver Spring, MD, 1996; 162–169.
61. MacDonald S, Anvik J, Bromling S, Schaeffer J, Szafron D, Tan K. From patterns to frameworks to parallel programs. *Parallel Computing* 2002; **28**(12):1663–1683.

DOI: 10.1002/cpe