

# Exploiting Dynamic Proxies in Middleware for Distributed, Parallel, and Mobile Java Applications

Willem van Heiningen<sup>1</sup>, Tim Brecht<sup>2</sup>, and Steve MacDonald<sup>2</sup>

<sup>1</sup> Integrative Biology  
Hospital for Sick Children  
Toronto, ON Canada  
willem@sickkids.ca

<sup>2</sup>David R. Cheriton School of Computer Science  
University of Waterloo  
Waterloo, ON Canada  
{brecht, stevem}@uwaterloo.ca

## Abstract

*Babylon v2.0 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 like object migration, asynchronous method invocation and remote class loading while providing an easy-to-use interface. The implementation of Babylon v2.0 exploits dynamic proxies, a feature added to Java 1.3 that allows runtime creation of proxy objects. This paper shows how Babylon v2.0 exploits dynamic proxies to implement several key features without the need for special language or virtual machine extensions, preprocessors, or compilers. The resulting Babylon programs are portable across all Java virtual machines, and the development process is simplified by removing the extra steps needed to invoke external stub compilers and incorporate the generated code into an application. This simplification also allows remote objects to be created for any class that supports an interface to its methods, even if source code is not available.*

## 1 Introduction

The Java language [6] has many features that facilitate distributed systems programming. Java's built-in security, threading and dynamic class loading support can greatly simplify the development of distributed applications. Furthermore, Java applications are compiled into a machine independent representation called bytecodes that can be run on any machine that runs the Java Virtual Machine (JVM). 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 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 v2.0 overcomes these limitations by providing programmers with Java classes and interfaces for remote object creation, interaction, and administration [22]. It contributes several new features and approaches in the area of Java-based distributed and parallel computing. Most importantly, it does so without the use of special language or virtual machine extensions, preprocessors, or compilers. Instead, Babylon v2.0 uses *dynamic proxies*, a new feature introduced in Java 1.3, that allows proxies to be created at runtime. Dynamic proxies can fulfill all of the same responsibilities as proxies normally generated by special stub compilers, like `rmic` for Java RMI. Thus, Babylon v2.0 can provide a Java-based solution without extending any of Java components and without adding extra development or compilation steps normally needed for stub compilers.

Specifically, this paper describes how dynamic proxies were used to implement remote object creation and remote method invocation. Remote objects can be created in one of two ways, by creating an object on a remote machine or by *exporting* an existing, serializable local object to another machine. Both forms of remote object creation use a dynamic proxy as the local placeholder for the remote object. Remote method invocations in Babylon can be done synchronously or asynchronously. The latter uses a novel technique based on an *asynchronous ticket*. Both method invocations are syntactically identical to local invocations, but with altered semantics. These semantics are achieved using different dynamic proxies to handle method calls.

An important contribution of this work is the reduction in the requirements needed for a class to produce remote objects. Babylon v2.0 only requires that the class implement an interface exporting client methods and be serializable.

This makes it possible to create remote objects from classes where source code is unavailable, including the Java standard class library, provided these requirements are met.

## 2 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.

The original implementation of Babylon [11], referred to here as Babylon v1.0, was an important initiative to make distributed computing resources more widely available to developers. Much of the design for Babylon v1.0 came from experiences with Agents [12] and ParaWeb [3]. 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 use 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.

Reflective RMI (RRMI) [21] adopts a reflective approach to remote method invocations. The user creates a descriptor object specifying the desired method using its name or signature, then creates a set of actual parameters for the call and requires the method be invoked by the runtime. The method can be run synchronously or asynchronously. Like Babylon v1.0, this is a non-standard method syntax that precludes compile-time checks. Synchronization with asynchronous methods can be achieved using a future-like object [8] called a *completion handle* or by registering a listener object to be invoked when the computation is complete and results (or thrown exception) are returned.

Several other existing systems, such as JavaParty [9] and Java/DSM [25], use a modified JVM or special preprocessors and language extensions to implement a functional distributed Java framework. Babylon v2.0 provides support for writing general distributed programs without requiring a modified JVM implementation, new keywords or other custom language extensions.

Other systems, such as Javelin [13], Charlotte [2], Ninplet [18], and Satin [24], only support distributed systems that can be formulated as a specific process structure, like master-worker, branch-and-bound, and divide-and-conquer.

Babylon v2.0 does not limit itself to a particular computational model and provides support for general remote object interaction using standard Java method invocation syntax. As a result, distributed applications that require more complex object interactions, such as a grid-based heat diffusion computation, can be written using Babylon v2.0. In contrast, such an application can not be implemented using the aforementioned systems.

ProActive [1, 10] provides similar services to Babylon v2.0, using similar techniques that do not require JVM changes or external tools. As well, it provides a group communication abstraction. ProActive generates proxies for remote objects at runtime using bytecode engineering libraries as it predates Java's dynamic proxies. This capability is used to create new remote objects and export local ones. ProActive can execute remote methods synchronously or asynchronously, but the choice is made automatically by the system based on implicit rules based on the method return type and presence of checked exceptions. Asynchronous calls return future objects [8], proxies that resolve to a final value on a method call, which transparently implement wait-by-necessity. The proxy classes generated by ProActive are subclasses of the originals. This provides polymorphism between local and remote objects and obviates the need for interfaces, but limits the classes that can be used to create remote objects or used as return types in methods. Final classes and methods, including arrays, cause problems. In addition, a remote object can only be run 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 [23].

In contrast, Babylon v2.0 uses an explicit approach to selecting synchronous versus asynchronous method invocations, which provides more control over both method invocation semantics and inter-object synchronization. Dynamic proxies in Java cannot be subclasses of an existing class, so delegation is used. This requires interfaces, but does not suffer from problems with final classes and methods. Babylon v2.0 uses Java RMI semantics at the server, where each remote method is run in a separate thread. This allows fewer remote methods in the grid-based heat diffusion application, but adds thread synchronization to Babylon v2.0 application code [23].

Babylon v2.0 uses Java RMI in its dynamic proxies. Since the remote method call is encapsulated by these proxies, it would be possible to use an alternate protocol or message passing layer like Ibis [10]. Ibis provides a high-performance, portable message passing layer that can use different protocols and networking hardware. Ibis performance results also show that object serialization in Java are a performance bottleneck, which is relieved using a byte-

code rewriter to generate Java serialization code. The use of this rewriter is counter to the stated goals of Babylon, which is to avoid extra preprocessing or compilation. Instead, Babylon could use the serialization optimizations presented in [19], which optimize serialization costs and reduce message sizes using a drop-in replacement for standard Java serialization.

## 2.2 Dynamic Proxies in Java

A proxy is an object that stands in for another, acting in place of the original [5]. 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 the stub and corresponding skeleton 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 adds its own functionality.

An example use of a dynamic proxy is shown in Figure 1. The proxy is created using the `newProxy()` method. It creates a new dynamic proxy that implements all of the interfaces for the argument object, and dispatches them against a new instance of the `MyProxy` class. The resulting proxy can be downcast to any of the interface types implemented by the original object. All method calls on the proxy object are forwarded to the `invoke()` method, where the invocation is represented as a `Method` object indicating the called method and an array of `Object` for the arguments. This proxy simply prints out information before and after the method is called on the original object.

## 3 Remote Object Creation

**Creation without Dynamic Proxies** Without dynamic proxies, creating remote objects must be done in one of two ways. First, an object already running at a server can export *factory methods* [5] that create new remote objects. This is the most common approach, with advantage of being simple to implement. The disadvantages of this approach are twofold. First, all potentially remote objects must have RMI stubs generated using `rmic`, meaning the object must be implemented as a remote object. This process is described in Section 4.1. Among the requirements is that the object must support an interface to its public methods which must

extend the `java.rmi.remote` interface. Second, this solution requires that object servers start running with factory objects already available.

Second, Java can create *activatable objects* that are instantiated when invoked. This removes the need to start a Java RMI program with factory objects already running. However, creating an activatable object adds more steps to the creation of a program. In addition to the above problems, the activatable object must extend the class `java.rmi.activation.Activatable`. Further, an extra setup program is required to register the implementation of this object with the RMI registry and RMI daemon processes so instances can be created at runtime.

One problem neither approach handles is converting a local object to a remote one. This could be done by explicitly creating a new remote object using the local one as an argument. This is only possible if the class has been written as a remote object but instantiated as a local one.

**Creation with Dynamic Proxies** Babylon v2.0 provides two methods for creating remote worker objects which can be used by clients to perform distributed operations. The

```
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 ite.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 1. Example dynamic proxy from [16].

```

1 // Matrix interface
2 public interface Matrix {
3     public Matrix computeInverse();
4 }
5
6 // MatrixImpl class
7 public class MatrixImpl
8     implements Matrix, Serializable {
9     public Matrix computeInverse() {
10         . . .
11     }
12 }
13
14 // Babylon v2.0 code using above Matrix.
15 try {
16     // Initialize Babylon.
17     Babylon.initApplication("schedulerhost");
18 } catch(Exception e) {
19     // Initialization failed.
20 }
21
22 // Create local instance of a matrix object.
23 Matrix matrix = new MatrixImpl();
24
25 // Export matrix to remote Babylon server.
26 try {
27     matrix = (Matrix) Babylon.export(matrix,
28         "MyMatrix", Matrix.class, "matrix.jar");
29 } catch(Exception e) {
30     // export failed
31 }
32
33 // Call computeInverse() on remote worker.
34 Matrix matrixInverse = matrix.computeInverse();
35
36 // Lookup reference to different matrix worker.
37 try {
38     matrix = (Matrix)Babylon.lookup("DiffMatrix",
39         Matrix.class);
40 } catch(Babylon.InstanceNotBoundException e) {
41     // Unable to locate the given matrix object.
42 }
43
44 // Move the object to a new host
45 Babylon.migrate(matrix, targethost);
46
47 // Get asynchronous ticket for Matrix worker.
48 Matrix asynch_matrix =
49     (Matrix) AsynchTicket.newTicket(matrix);
50
51 // Call method computeInverse() asynchronously.
52 asynch_matrix.computeInverse();
53
54 // Other code that executed concurrently with
55 // computeInverse() can go here.
56
57 // Retrieve the invocation result (blocks if
58 // result is not yet available).
59 try {
60     matrixInverse = (Matrix)
61         AsynchTicket.getResult(asynch_matrix);
62 } catch(RemoteExecException ex) {
63     // The target method threw an exception.
64 }

```

**Figure 2. Babylon v2.0 Code Example**

Babylon.remoteNew() method takes a class name as one of its arguments and creates a new instance of the given class on a remote Babylon server. Alternatively, the Babylon.export() method can be used to distribute an existing local instance of a serializable object to a remote Babylon server. This technique preserves the state of the local object. However, since the local object is copied to the Babylon v2.0 server, only serializable objects can be turned into worker objects using this technique. Line 27 in Figure 2 shows how an object is exported in Babylon v2.0.

Both methods return a 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. Providing transparent access is a significant enhancement over Babylon v1.0, as shown later.

Worker object look up functionality is also provided with the help of a worker object registry implemented in the Babylon scheduler. The registry maintains a record of all the worker objects in the Babylon v2.0 system. Clients can look up references to worker objects based on the worker object's instance name and the interface it implements. The static method Babylon.lookup(String instanceName, Class workerInterface) 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. Line 38 in Figure 2 demonstrates how to use the Babylon.lookup() method.

Perhaps the biggest benefit to using dynamic interfaces with respect to remote object creation is it reduces the number of requirements that an object must fulfill to be remote. In Babylon v2.0, an object can be made remote if the following two criteria are met. The first requirement is that the class must implement an interface that exports methods to be accessed by clients. In fact, the method can implement several interfaces, and the generated proxy will implement all of them. (This introduces a problem if different interfaces contain the same method signature but expect different implementations.) Unlike Java RMI, these interfaces do not have to extend any other interface. The second requirement is that the class must implement the Serializable interface (as must all of its instance variables) so that it can be marshaled and sent to a remote machine. All of this is accomplished completely within Java and does not require any external tools, preprocessors, or compilers.

This improvement is shown in Figure 2. The interface Matrix is given on lines 1 to 4, a normal Java interface that does not extend any special Babylon-specific or remote interfaces. An implementation class that implements this interface appears at lines 6 through 12. It only implements the Matrix and Serializable interfaces. Outside of the

Serializable interface, this class has no artifacts that suggest it will be run in a distributed environment. However, Babylon v2.0 can use this class to create remote objects. Furthermore, it should be possible to create remote versions of objects for which source code is unavailable, such as the Java standard class library, provided they meet the simpler requirements for distribution. This is not possible with Java RMI because of the requirements for creating remote objects. Further, these remote objects contain other artifacts of their distributed execution, such as requiring methods to throw `RemoteException`.

## 4 Remote Method Invocations

### 4.1 Synchronous RMI

**Synchronous RMI without Dynamic Proxies** Without dynamic proxies, synchronous remote method invocation can be built with Java RMI. The benefits are that remote method calls look like local calls and that RMI proxies permit compile-time checking of method calls. The main problem with RMI is that the class must meet a set of conditions before it can be used with Java RMI. First, the class needs to implement an interface to export methods to clients, which must extend the `java.rmi.remote` interface. Second, any class initialization must call the `exportObject()` method in the `java.rmi.server.UnicastRemoteObject` class, which may throw the checked exception `RemoteException`. This requires all constructors and initialization methods to indicate 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 also throw `RemoteException`. Only if all of these conditions are met can we use the `rmic` stub compiler to generate static proxies and use the class in an RMI program.

Unless the designer of a class created it with RMI in mind, converting the class to create remote object requires access to the source code. We cannot subclass the original class because the new class must also be a subclass of `UnicastRemoteObject`, and Java lacks multiple inheritance. Inserting the call to `exportObject()` requires the source code. We could instead try delegation, creating a new remote object that holds an instance of the class we wish to make remote. However, the delegate object will not be substitutable for the original object; it will not be a subclass of the original (the delegate must be derived from `UnicastRemoteObject`) and it cannot use any of the interfaces defined for the original (since they will not be remote interfaces, provided the original class defines any interfaces). As a result, application code written to use the original object will need to be rewritten to use the delegate.

```
1 // Invoke method obj.ask(question)
2 try {
3     answer = (String) Babylon.rmi(
4         obj,        // remote obj
5         "ask",      // the method
6         question    // argument to method
7     );
8 } catch( // exception ) {
9     // Exception handling code
10 }
```

Figure 3. Babylon v1.0 RMI

In contrast to RMI, the syntax used by Babylon v1.0, shown in Figure 3, resembles Java Reflection [7]. The syntax is clumsy and completely different than standard Java method invocation, making it awkward to use. Worse, this syntax prevents compile-time detection of errors such as invoking a non-existent method, passing an incorrect number or types of arguments to a method. Also, this syntax cannot accept primitive values; wrapper classes must be used.

**Synchronous RMI with Dynamic Proxies** The remote object creation methods in Babylon v2.0 return dynamic proxies for newly created worker objects. Clients can use these dynamic proxies to invoke methods on worker objects using standard Java method invocation syntax. Remote method invocation in Babylon v2.0 permits the use of primitive type method arguments and ensures type checking of the target method and the arguments at compile time. A dynamic proxy to a worker object can safely be passed as an argument or returned as a result in any local or remote method invocation. Line 34 in Figure 2 provides an example of method invocation on a worker object.

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

Since Babylon uses Java RMI as the underlying communication mechanism, parameters and return values are passed to and from worker objects using standard RMI semantics. In other words, primitive type and object parameters are passed by value using object serialization but remote object parameters (including Babylon dynamic proxies) are passed by reference. Another side-effect of using RMI as the underlying communication mechanism is that

a new thread is started at the server by the RMI runtime for each method invocation on a worker object. Programmers should be mindful of this fact, especially when creating public worker objects, because several clients may be invoking methods on the worker object concurrently. The methods of such worker objects should be thread-safe.

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

## 4.2 Asynchronous RMI

Network latency in a distributed application environment can incur significant overhead and reduce overall application performance. One way to reduce the impact of network latency is to overlap communication and computation [15]. 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.

**Asynchronous RMI without Dynamic Proxies** Asynchronous RMI without dynamic proxies is usually done by adding threads to Java RMI. The programmer can create a new thread for each method invocation. Obtaining results requires that the program join with this thread before gathering results. The developer will have to decide how to handle any exceptions (whether to deal with them in the new thread or rethrow them in another thread).

In Babylon v1.0, the syntax for asynchronous calls is similar to that in Figure 3, with three differences. First, the static method `Babylon.armi()` is used. Second, the return value of this method is an object of type `Future` that represents the results [8]. The results are obtained using the `get()` method on the `Future` object, which returns an object that can be downcast to the correct type. Any exceptions during the asynchronous invocation are thrown when the `get()` method is called. As with synchronous calls, this syntax is clumsy and prevents compile-time checks.

**Asynchronous RMI with Dynamic Proxies** Babylon v2.0 uses a novel technique based on special proxy objects called *asynchronous tickets* to support asynchronous remote

method invocation. An asynchronous ticket can be used to make the next method invocation on a worker object asynchronous. The method is invoked on the ticket using standard Java invocation syntax but the invocation completes asynchronously. Applications can continue running normally while the invocation completes.

When a client invokes a method on an asynchronous ticket, the underlying Babylon v2.0 runtime starts a service thread on the client which handles the remainder of the invocation. The client thread returns immediately while the service thread performs a normal synchronous remote method invocation for the requested method. Parameters and return values passed to the worker object using asynchronous remote method invocation follow standard RMI parameter passing semantics. The method's return value can be requested from the ticket at a later time.

Babylon v2.0 asynchronous remote method invocation resolves on the return value of the target method. Resolving on the return value ensures that any other side-effects have completed. This approach is 100% Java compatible and uses Java's standard method invocation syntax which can be checked at compile-time.

On line 48 of Figure 2, a new asynchronous ticket is obtained for the `Matrix` worker object used in the example. This ticket is used on line 52 to make an asynchronous invocation of the `computeInverse()` method. The method invocation is allowed to complete asynchronously and the result of the invocation is retrieved from the ticket using the static `getResult()` method on line 60.

Previous work has considered mechanisms for implementing asynchronous remote method invocations [14, 4, 20]. 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 run-time systems and does not require any preprocessing or a modified stub compiler.

## 5 Other Babylon Features and Performance

Babylon v2.0 also includes many other features to simplify the process of developing distributed, parallel, and mobile Java programs. These features are detailed in [22].

First, Babylon v2.0 supports remote class loading so clients do not need login access to the machines that host their remote objects. Furthermore, each application has a separate class loader with its own namespace, so different users can have classes with the same name without conflict. Second, Babylon v2.0 supports the migration of objects between servers using several different methods. One of these methods is safe-point migration, a novel technique that uses a combination of checkpointing and rollback. This migration can be done by the user or by a system administrator.

Third, Babylon v2.0 includes remote I/O to access I/O resources on remote machines.

We have conducted an extensive performance evaluation of Babylon v2.0 and have compared the performance with Babylon v1.0. We have found the performance of Babylon v2.0 is on par or better than that of v1.0 and that the extensive use of dynamic proxy objects in Babylon v2.0 does not degrade its performance [22].

## 6 Conclusions

This paper highlights the use of dynamic proxies in a middleware system for building distributed, parallel, and mobile Java applications. Dynamic proxies help in the implementation of remote object creation (specifically exporting an existing local object to a remote server) and remote method invocation (both synchronous and asynchronous).

One of the biggest benefits of using dynamic proxies is the reduction in the requirements needed to create remote objects. Classes need only implement any interface that exports methods and implement the `Serializable` interface. This opens the possibility of creating remote objects from classes we do not have source code, such as those in the Java class library, which is not possible with Java RMI.

## Acknowledgments

We gratefully acknowledge Morgan Stanley Dean Witter, the Ontario Research and Development Challenge Fund, and the National Sciences and Engineering Research Council of Canada for financial support for portions of this project. This paper has also benefited from the comments and suggestions of the anonymous reviewers.

## References

- [1] I. Attali, D. Caromel, and R. Guider. A step toward automatic distribution of Java programs. In *Fourth International Conference on Formal Methods for Open Object-Based Distributed Systems*, pages 141–161, 2000.
- [2] A. Baratloo, M. Karaul, Z. M. Kedem, and P. Wyckoff. Charlotte: Metacomputing on the Web. *Future Generation Computer Systems*, 15(5):559–570, October 1999.
- [3] T. Brecht, H. Sandhu, J. Talbot, and M. Shan. ParaWeb: Towards world-wide supercomputing. In *Proc. of the Seventh ACM SIGOPS European Workshop*, pages 181–188, 1996.
- [4] K.E. Falkner, P.D. Coddington, and M.J. Oudshoorn. Implementing asynchronous remote method invocation in java. In *Proc. the Parallel and Real-Time Systems Conference*, 1999.
- [5] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.
- [6] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification, 2nd Edition*. Addison Wesley, 2000.
- [7] D. Green. The reflection API, 2005. Available at <http://java.sun.com/docs/books/tutorial/reflect>.
- [8] R. Halstead. Multilisp: A language for concurrent symbolic computation. *ACM Transactions on Programming Languages and Systems*, 7(4):501-538, 1985.
- [9] B. Haumacher and M. Philippsen. Exploiting object locality in JavaParty, a distributed computing environment for workstation clusters. In *The Ninth Workshop on Compilers for Parallel Computers*, pages 83–94, June 2001.
- [10] F. Huet, D. Caromel, and H. Bal. A high performance java middleware with a real application. In *Proc. Supercomputing Conference*, 2004.
- [11] M. Izatt. Babylon: A Java-based distributed object environment. Master’s thesis, York University, Toronto, July 2000.
- [12] M. Izatt, P. Chan, and T. Brecht. Agents: Towards an environment for parallel, distributed and mobile Java applications. *Concurrency: Practice and Experience*, 12(8):667–685, July 2000.
- [13] M. O. Neary, A. Phipps, S. Richman, and P. R. Cappello. Javelin 2.0: Java-based parallel computing on the Internet. In *The Sixth International European Parallel Computing Conference*, volume 1900 of LNCS, pgs 1231-1238, 2000.
- [14] R. Raje, J. I. William, and M. Boyles. An asynchronous remote method invocation (ARMI) mechanism for Java. In *ACM 1997 Workshop on Java for Science and Engineering Computation*, June 1997.
- [15] V. Strumpfen and T. L. Casavant. Exploiting communication latency hiding for parallel network computing: Model and analysis. In *International IEEE Conference on Parallel and Distributed Systems*, pages 622–627, December 1994.
- [16] Sun Microsystems, Inc. Dynamic proxy classes, 1999. At <http://java.sun.com/j2se/1.3/docs/guide/reflection/proxy.html>.
- [17] Sun Microsystems, Inc. Java2 platform, standard edition, v1.4.0 API specification, 2002. At <http://java.sun.com/>.
- [18] H. Takagi, S. Matsuoka, H. Nakada, S. Sekiguchi, M. Satoh, and U. Nagashima. Ninflot: a migratable parallel objects framework using Java. In *1998 Workshop on Java for High-Performance Network Computing*, pages 151–159, 1998.
- [19] K. Tan, D. Szafron, J. Schaeffer, J. Anvik, and S. MacDonald. Using generative design patterns to generate parallel code for a distributed memory environment. In *Proc. ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 202-214, 2003.
- [20] W.F. Taveira, M.T. de Oliveira Valente, M.A da Silva Bigonha, and R. da Silva Bigonha. Asynchronous remote method invocation in java. *Journal of Universal Computer Science*, 9(8):761–775, 2003.
- [21] G. Thiruvathukal, L. Thomas, and T. Korczynski. Reflective remote method invocation. *Concurrency: Practice and Experience*, 10(11-13):911-925, 1998.
- [22] W. van Heiningen. Babylon v2.0: Support for distributed parallel and mobile Java applications. Master’s thesis, School of Computer Science, University of Waterloo, 2003.
- [23] W. van Heiningen, T. Brecht, and S. MacDonald. Babylon v2.0: Middleware for distributed, parallel, and mobile Java applications. In *Proc. 11th International Workshop on High-Level Parallel Programming Models and Supportive Environments*, 2006.

- [24] R. van Nieuwpoort, J. Maassen, T. Kielmann, and H. Bal. Satin: Simple and efficient Java-based grid programming. *Scalable Computing: Practice and Experience*, 6(3):19-32, 2005.
- [25] W. M. Yu and A. Cox. Java/DSM: A platform for heterogeneous computing. *Concurrency: Practice and Experience*, 9(11):1213–1224, November 1997.