Ajents: Towards an Environment for Parallel, Distributed and Mobile Java Applications

Matthew Izatt, Patrick Chan
Department of Computer Science
York University, Toronto, Ontario, M3J 1P3
{izatt,y-chan}@cs.yorku.ca

Tim Brecht
Department of Computer Science
University of Waterloo, Waterloo, Ontario, N2L 3G1
brecht@cs.uwaterloo.ca

Abstract

The rapid proliferation of the World-Wide Web has been due to the seamless access it provides to information that is distributed both within organizations and around the world. In this paper, we describe the design and implementation of a system, called Ajents, which provides the software infrastructure necessary to support a similar level of seamless access to organization-wide or world-wide heterogeneous computing resources.

Ajents introduces class libraries which are written entirely in Java and that run on any standard compliant Java virtual machine. These class libraries implement and combine several important features that are essential to supporting distributed and parallel computing using Java. These features include: the ability to easily create objects on remote hosts, to interact with those objects through either synchronous or asynchronous remote method invocations, and to freely migrate objects to heterogeneous hosts. While some of these features have been implemented in other systems, Ajents provides support for the combination of all of these features using techniques that permit them to operate together in a fashion that is more transparent and/or and less restrictive than existing systems.

Our experimental results show that in our test environment: we are able to achieve good speedup on a sample parallel application; the overheads introduced by our implementation do not adversely affect remote method invocation times; and (somewhat surprisingly) the cost of migration does not greatly impact the execution time of an example application.

1 Introduction

One of the compelling reasons for implementing distributed and parallel applications in Java is that compiled Java programs produce byte code which may be executed or interpreted on any machine that implements the Java Virtual Machine (JVM) [14]. This frees the programmer from being concerned with problems due to differences in architecture that traditionally plague heterogeneous distributed and parallel applications. Additionally, the standardized virtual machine provides for new opportunities to migrate objects between heterogeneous hosts. Moreover, the Java security manager provides functions which are especially important in a distributed system where untrusted user programs are granted permission to execute on unrelated hosts.

Ajents [1] is a collection of Java classes and servers (also written in standard Java) designed to provide a seamless, integrated environment for implementing distributed, parallel and mobile Java applications. No modifications are made to the Java language and no preprocessors, special compilers, or special stub compilers are required. Remote instances of objects can be created without modifying their source code (e.g., to extend `UnicastRemoteObject` as is required for standard RMI use) and without the need to create stubs and skeltons for these classes (i.e., there is no need to run `rmic`). This means that objects for which only byte-codes (`.class` files) are available can be instantiated on remote hosts. If the classes are serializable they can even be migrated. Most importantly, Ajents can be executed on any standard Java virtual machine.

With Ajents, one can permit users who do not otherwise have access to outside systems to utilize these systems, while Java security features are used to protect hosts which provide access to their resources. This is done by running a relatively small and simple server (called the Ajents server). It acts as a point of contact for applications that wish to utilize or access resources on that system.

The combination of the Ajents server and class libraries greatly simplifies the task of writing distributed Java applications by supporting several key features not currently supported in Java:

1. **Remote Object Creation:** While Java supports remote method invocation there is no support for an object on one machine to simply and easily create an object (e.g., using `new`) on a remote host. While it is possible to invoke methods of `remote` objects that have been statically created on a remote machine, there is no way for an object to actually create an object on a remote machine from a local machine. Ajents provides facilities for the remote creation and referencing of standard Java objects.

[1] The name Ajents was chosen because we believe that this environment would make a good tool for building mobile agents using Java.

2. **Remote Classloading**: In order to be able to instantiate objects on any remote host, Ajents implements remote class loading. When creating or migrating objects, Ajents includes the relevant byte-code, allowing the remote server to define and load the class dynamically.

3. **Asynchronous Remote Method Invocation (Futures)**: Although Remote Method Invocation (RMI) is supported in Java [27], all invocations are performed synchronously. Ajents supports the ability to overlap communication with computation by providing asynchronous remote method invocation. The invoking object is then free to continue execution until a point at which a returned value is needed (if any).

4. **Object Migration**: One of the key components of Ajents is its ability to migrate objects to multiple heterogeneous hosts. In an environment where resources are being loaned and shared by a number of users, we believe that it is important to be able to migrate an object while it is executing. Ajents is able to accomplish this without a preprocessor, and without modifying the virtual machine, compiler or stub compiler. This is implemented using checkpointing, roll back and restarting mechanisms. Thus, if desired, any *serializable* object can be migrated while executing by interrupting its execution, moving the most recently checkpointed state of the object and restarting the currently executing method. These mechanisms not only permit an object's execution to be interrupted but will also prove useful in the future when we hope to add fault tolerance and persistence. Because migration relies on the serialization facilities provided in Java, we are unable to migrate objects that are not serializable (e.g., threads and AWT objects).

We believe that continued improvement in execution speeds of Java applications due to just-in-time compiler technology and the combination of features provided in Ajents make Ajents an excellent basis for developing powerful distributed, parallel and mobile Java applications.

The remainder of the paper is organized as follows. Section 2 discusses related work. Section 3 describes the architecture of Ajents and the mechanisms used to implement remote object creation, remote class loading, asynchronous remote method invocation, and object migration. In Section 4 we evaluate the performance of several of Ajents key components and the performance of some applications that have been implemented using Ajents. Section 5 discusses some of the issues related to the current design and implementation and describes possibilities for future work. The paper is concluded with a summary, in Section 6.

2 Related Work

Much of the impetus for the design of Ajents comes from experience with ParaWeb [4], a Java based environment for parallel computing. ParaWeb was designed and implemented before remote method invocation [31] and object serialization [23] were added to Java. Hence, objects communicate through the awkward sockets interface.

Other systems developed since that time also leverage the Java virtual machine to provide distributed or parallel computing platforms on heterogeneous computing environments [7, 3, 1, 6, 20, 32, 25, 24, 16, 19, 29].

JavaParty [20] introduces the `remote` class modifier to the Java language. Adding this new keyword to the class definition denotes that the class should be used in a distributed fashion. To accomplish this, JavaParty uses a preprocessor which converts `remote` classes into pure Java code with RMI hooks. The change to the language is designed to simplify RMI programming, placing the burden of creating and handling remote proxies upon the preprocessor. This greatly simplifies the programming task, but results in increased complexity in the actual Java code produced by the preprocessor. However, the main drawback of this approach is that it introduces modifications to the Java language and therefore necessitates the use of the preprocessor. The mechanisms supplied by Ajents permits users to create remote objects without using a special compiler, stub compiler or preprocessor. In fact, Ajents is able to support the creation of remote objects for which only the byte-code is available. Such objects can be created remotely and their methods may be invoked synchronously or asynchronously. However, in order for them to be eligible for migration the original objects must be serializable (and if they aren't it's relatively simple for a programmer to extend the original object so that it is serializable).

Javelin [6] and SuperWeb [1] implement a "global computing infrastructure" which brings together three types of entities: clients, brokers and hosts. *Clients*, who seek computing resources, register with a *broker* and submit their work in the form of an applet. *Hosts*, who are willing to donate resources, contact the broker and run the applets. This work emphasizes methods for bringing together clients and hosts, and focuses on ways of bartering for CPU time. Bayanihan [24] also uses applets to supply a framework for volunteer computing using Java. Javelin++ [16] extends the work originated in Javelin by implementing Java applications and by more closely examining issues related to scheduling and scalability.

The goal of Charlotte [3, 11] is to support distributed shared memory on top of the Java virtual machine. It provides classes which encapsulate the behavior of a distributed shared memory system. All accesses to shared classes must be done through the provided interface using function calls (e.g., `myInt.get()`). Charlotte provides an abstraction of DSM, providing fine-grained support for distributed work.

Other work has considered mechanisms for implementing asynchronous remote method invocations, ARMI [21] and Active-RMI [10]. Both of these use a modified Java RMI stub compiler (`armic`) to implement asynchronous remote method invocations. In ARMI if the user chooses to use `armic` (instead of the usual `rmic`), all stub classes created will be modified so that (synchronous) RMI calls are made by a separate thread resulting in asynchronous behavior for the calling object. The user is provided with the option of allowing all RMI calls within a class to be either synchronous or asynchronous, but not both. Active-RMI implements asynchronous RMI calls and provides ways to block the invoking object (in order to essentially synchronize the call) and to also check if the result of an RMI is available without blocking. The main disadvantage with this approach is that a separate thread will be created in order to perform an asynchronous call, even when synchronous behaviour is desired. In contrast Ajents provides two distinct interfaces for synchronous and asynchronous method invocations and it does not require the use of a modified stub compiler. Active-RMI also differs from other systems in that remote objects are active objects (i.e., each object has an independent thread). While earlier versions of Ajents

[5] provided support for active objects, they are not longer supported because they can not be migrated (since it is not possible to capture the execution state of a thread in a standard JVM).

Agent-oriented Java systems, some examples of which include Mole [25] and Aglets [13], are designed to support Java-based autonomous software agents. Remote agents may be created, and are provided with mechanisms for mobility and communication. Generally these agent-oriented systems concentrate upon the independent movement of objects, while providing less support for interaction and control by the object's creator. In such systems, in order for agents to behave intelligently, yet independently, their behavior must be explicitly programmed. For example, it is typical for such systems to require a stop method to be called, before migrating an object. Therefore, each agent must be programmed to handle a stop method in which it prepares to migrate, including ensuring that its state does not change. The agent must also be programmed to handle a restart method after migration is complete. This type of intelligent behavior places the burden upon the programmer, rather than the system. Ajents, on the other hand is targeted towards easily building distributed object applications and as a result emphasizes programming simplicity and object control. Unlike existing agent-oriented systems, Ajents' objects are not expected to exert self-control, or have knowledge of the system.

ObjectStore has developed Voyager which provides support for remote object creation, asynchronous remote method invocation, and migration [19]. Although Voyager is a substantial product that includes functionality beyond that provided in Ajents (e.g., multicast and name spaces), the main advantages of Ajents is that we are able to create objects remotely without requiring the use of a stub compiler and we provide support for immediate migration, delayed migration and migration with checkpointing and rollback, while Voyager only supports immediate and delayed migration. In our view Ajents also provide interfaces that are more seamless to use than those supported by Voyager (especially those for migration).

Another system, ABC++ [2], which is a library designed for concurrent object-oriented parallel programming in C++, has influenced the design of Ajents' interfaces for remote object creation and asynchronous RMI. ABC++ does not support object migration, nor does it include special features for heterogeneous computing.

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

3 Ajents Architecture

In this section, we describe the design and implementation of Ajents. We describe those features of the system that a programmer uses in order to implement a distributed, parallel or mobile application. An object must be created (Section 3.1), on an Ajents server (Section 3.3), where it will be instantiated based upon its class file (Section 3.2). From there, methods of the object can be invoked (Section 3.4), and the object may be migrated to other servers (Section 3.5).

3.1 Remote Object Creation

One of the more serious limitations with respect to implementing distributed applications using the current definition of Java (JDK 1.1) is that although support is provided for remote method invocation there is no support provided for a local object to easily create an object on a remote host. In other words, while it is possible to invoke methods of existing objects that have been statically created and are already known to exist on a remote machine there is no way to create an object on a remote machine. (Figure 1 demonstrates the ease with which remote objects can be created in Ajents by using `Ajents.new` rather than `new`.)

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

In Ajents, remote object creation is supported by client and server side class libraries. The key mechanisms used for remote object creation are implemented inside the Ajents server which is implemented in Java and executes within a Java virtual machine on the remote host. This enables any machine which is running the Ajents server to act as a target for remote object creation. Figure 1 shows an example of how a user creates a remote object. The user first obtains a reference to an Ajents scheduler object by calling `Ajents.register()`. The scheduler object is later consulted in order to obtain an available server (note that the actual scheduler may reside on a different host). Ajents servers only contact the scheduler object when they are started and stopped, because scheduling is currently done in a round-robin fashion. We are investigating improved scheduling techniques that involve more frequent communication from the Ajents server to update load information.

Four parameters are passed to the `Ajents.new` method. These represent (1) the fully qualified class name, (2) a user-specified name for the object, (3) the location of the source code, and (4) an Ajents server upon which the object will be created (in this case obtained from the scheduler).

Once a user calls `Ajents.new`, the method call is translated into an RMI call to the remote Ajents server. Here, using Java reflection mechanisms [28], the requested class can be loaded into a `Class` object at the remote host and then instantiated. The Ajents server then adds the new object to its object table, and returns a remote reference to the user. Further details related to class loading are provided in Section 3.2.

The object returned to the user (of type `AjentsObj`) is actually a proxy object. This is done to shield the user from

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

// Create a remote object on an available server
AjentsObj obj = Ajents.new("ajents.tests.NewObj", "NewObj_1",
                 "/home/ajents/tests/myobjs.jar", sched.AvailServer());
```

Figure 1: Example of a remote object creation.

the internal complexities of Ajents and Java RMI. As well, it allows Ajents to maintain control over the actual remote reference, which is required in order for migration to occur asynchronously and independently of any user actions (if desired).

The range of objects that may be submitted to Ajents is limited only by the restriction that in order to take advantage of Ajents' migration features, submitted objects must be serializable. While the mechanisms implemented within the Ajents class libraries rely upon RMI to execute remote methods, the submitted objects need have no knowledge of this. Thus objects that were not originally designed or intended to operate as remote objects can be treated as remote objects in Ajents. This means that existing objects for which only byte-code (i.e., no source) is available, can be remotely created, used and even migrated (provided the migrating objects are serializable) and that no stubs or skeletons need to be generated (using `rmic`).

3.2 Remote Class Loading

The dynamic class loading described in the Java RMI specification [27] requires that all class files be located either locally, in the `CLASSPATH`, or at a pre-defined URL. This was deemed insufficient for Ajents, where our goal is to allow objects to be created on arbitrary participating machines. In order to utilize remote resources, Ajents users are not required to place their class files on a WWW server nor are they expected to have accounts on, or physical access to all participating hosts in order to pre-load a copy of the byte-code. Therefore, in order to ensure complete portability we transfer the Java byte-code to the remote host at the time of object creation. To accomplish this, Ajents requires that all necessary byte-codes be placed into a Java archive file (JAR), which is then transferred along with the remote object creation request. Multiple class files are archived in order to reduce the number of messages needed to obtain all necessary class files (rather than using individual messages for each class file). During migration a copy of the JAR file is included with the object so that the target server is able to also obtain the class files (if necessary). This is done by extending Java's default class loader to load class files from other sources (e.g., the `AjentsObj` on the host where the object is being created or the Ajents server on the host from which a migrating object is being moved). Since the Ajents class loader is just an extension of the default class loader the same rules are applied for class loading with packages being used to ensure that names are distinct.

3.3 Ajents Server

The role of the Ajents server is to provide a point of contact for creating objects on the host on which the Ajents server resides. In the same way that a client World-Wide-Web browser is not able to access information stored on systems that are not executing an HTTP server, Ajents is not able to utilize resources of systems that are not running an Ajents server. The Ajents server is responsible for security, authorization, and authentication. In addition, it implements policies that control who is permitted to create objects on the local machine and when.

The Ajents server also keeps track of objects currently residing on it. This allows the server to track, limit, and move objects when resources become heavily utilized. It also permits the server to keep track of the location of migrated objects (this is explained in detail in Section 3.5).

3.4 Asynchronous RMI

It is commonly known that the performance of many distributed and parallel applications is hindered by network latencies. A common technique for improving performance is to overlap communication with computation. Existing techniques for remote method invocation [27] cause the execution of the requesting object to be suspended until the method has completed. This delay is incurred regardless of when or if the invoking object requires the return value. Ajents provides a means for performing asynchronous remote method invocations, thus enabling applications to overlap communication and computation.

In Ajents, when the user performs an asynchronous RMI, a separate thread is created which performs the RMI (a possible optimization would be to pre-fork a pool of threads that would be reused, thus reducing overheads due to thread creation). As a result, the original object is only blocked during the creation of the thread, and thereafter continues execution. Meanwhile, the new thread performs a regular synchronous RMI.

To support return values, we use the concept of a future. In essence, our `Future` object is a temporary receptacle for the return value. The return value (which may be an object) is held inside the `Future` object until a `get()` request is made by the program. When a call to `get()` is made, the result of the asynchronous RMI is returned to the calling object. If the result is not yet available, the object will block until the result is available.

Figure 2 contains code fragments that show how someone programming with Ajents would perform synchronous and asynchronous remote method invocations. Note that in order to support both synchronous and asynchronous remote method invocations within the same object and without modifying the Java language, compiler, or the Java RMI stub compiler, we support synchronous RMI calls using the `Ajents.rmi()` method and asynchronous RMI calls using the `Ajents.armi()` method. Each of these methods is overloaded to support a variable number of arguments, up to a predefined maximum number of arguments. Our current implementation supports up to ten arguments and can easily be modified to support an arbitrary but fixed number of arguments.

In order to handle exceptions, we propagate all exceptions from an asynchronous remote method invocation through the `Future` object back to the user program. This allows the method that is processing the result of the method invocation to handle whatever exceptions occurred (including invalid methods or parameters). Figure 2 also shows how the user program is notified of and handles exceptions when getting return values from the asynchronous RMI calls. In the case of synchronous calls the exception is propagated back to the point where the `Ajents.rmi()` method is invoked.

3.5 Object Migration

An important function that is not readily available in many systems designed for implementing distributed applications is the ability to easily migrate computation, especially in heterogeneous environments.

Migrating processes causes serious problems in a network of workstations environment where multiple individual workstations are shared among a number of users. The main problem in such environments occurs when the owner of the workstation desires sole usage of their workstation, only to find that its resources are being utilized by someone else. Current approaches to this problem include suspending the intruding job until the user is no longer using the machine or in rare instances migrating the process to another machine of the same architecture.

Ajents provides a simple and relatively effective means for migrating Java objects between different operating systems and even different architectures. Objects are permitted to migrate themselves or to have other objects initiate their migration. Some of the mechanisms required in order to migrate Java objects are partially provided in current Java implementations in the form of object serialization [26], class loaders and the standardization of the Java virtual machine.

Ajents supports three forms of object migration: the immediate migration of idle objects, the delayed migration of executing objects, and the immediate migration of executing objects using checkpointing and roll back.

The migration of idle objects is fairly simple since we do not need to concern ourselves with the possibility that they will be modified during or after migration. This can be accomplished by using the JDK's `readObject()` and `writeObject()` serialization methods in combination with socket connections. The more technically challenging aspect of implementing migration involves ensuring that remote method invocations, both synchronous and asynchronous, continue to operate correctly before, during and after migration (this is discussed in more detail later in this section).

Ajents supports delayed migration, where a migration request gets completed only once the object is idle. To accomplish this, anytime a method of an object is executed, a flag is set. When a migrate call is made, it checks the flag, and calls `wait()` if the flag is set. Upon completion of the method invocation the object is idle so the flag is cleared, and `notify()` wakes up the thread that invoked migrate. Once the object is idle, we follow the same procedure as in the case of an idle object.

Delayed migration is supported because Java does not provide for the ability to save and restore the state of an executing object (e.g., there is no access to the program counter or stack). Therefore, it is not possible to fully support the migration of actively executing objects without modifying the implementation of the virtual machine (i.e., the interpreter) [22] or without preprocessing [8] the Java source code. As a result, Ajents supports the immediate migration of objects which are currently executing methods by using checkpointing and roll back. Ajents is able to accomplish this by checkpointing the object before beginning each remote method invocation, interrupting the executing method when migration is requested, migrating the checkpointed version of the object and then re-executing the method call using the checkpointed object state. Checkpointing involves storing the state of the object prior to a method invocation (this is done by serializing the object and storing it in memory), keeping track of the method being invoked, the parameters being passed to that method, and the thread in which the object is executing. If a migration request is later received during the execution of a method, the server is able to suspend execution of the thread, and move the saved state of the object to the new host. Then using the saved method name and parameters, the Ajents server on the remote host continues execution of the object by re-invoking the method that was called immediately after checkpointing. (Issues related to checkpoint consistency are discussed in Section 5.)

Control over whether checkpointing occurs lies in the hands of the user program which has remotely created the object and wishes to call its methods. The user may choose to enable `Ajents.setCheckPoint(object,true)` or disable `Ajents.setCheckPoint(object,false)` checkpointing for each object. Or they may wish to override the current setting prior to individual method invocations. By default, checkpointing is not initially activated. This flexibility allows for the object to be checkpointed only at key function calls, or for checkpointing to be avoided when consecutive read-only functions are being executed. Since the overhead involved in creating a duplicate copy of larger objects can be significant, the user control over checkpointing is vital to Ajents' efficiency. As part of our future work we plan to investigate methods for providing the programmer with more and finer-grained control over when checkpointing occurs (see Section 5). Furthermore, the overhead of checkpointing is only needed (and justified) for long-running jobs. For these cases, checkpointing occurs relatively infrequently, and substantial roll backs can be incurred without significantly impacting overall execution time.

We now outline each of the steps involved in migration using checkpoint and roll back in more detail:

1. Checkpointing (making a copy of the state of an object), is done using Java serialization techniques. This creates a deep copy of the object, meaning all member variables are themselves copied, rather than just their references. Ajents does this by intercepting each remote method invocation (at the server side) in order to checkpoint the object before allowing the method to execute. In addition, using reflection, Ajents is able to store a copy of the method name and its parameters in case the method is interrupted by a migration request and the method invocation needs to be restarted after a migration. Finally, Ajents stores a reference to the thread which will be executing the method (in order to interrupt the thread if necessary).

2. Upon receiving a migration request the Ajents server interrupts the currently executing thread. This interrupt causes an exception, which is then caught by Ajents instead of being returned to the user. Ajents is thus able to gain control of the execution of the thread (i.e., suspend execution) and can then proceed with object migration.

3. Once execution of the thread has been halted, the object is in a static state, however, there is no way to

```
try {
    // Make synchronous RMI call to the method setName
    // Parameters:  object reference, method name, and (optional) method parameter(s)
    Ajents.rmi(obj, "setName", "NewObj");
} catch (Exception ex) { ... }

// Make asynchronous RMI calls to the method getName
Future future = Ajents.armi(obj, "getName");

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

Figure 2: Examples of Synchronous and Asynchronous Remote Method Invocations

determine where the thread is executing or the state of the stack. As a result, we migrate the state of the object as it was at the time of the last checkpoint.

4. Following migration, we re-execute the interrupted method. This is done without user intervention, since Ajents already has all the information needed to restart the method invocation.

As mentioned previously, the Ajents server keeps track of the objects currently residing on it as well as maintaining a reference to the new home of any object that has been migrated. The reference to the new home of the object is maintained so that migration can be implemented in a way that is transparent to any object that has a reference to a remote object that has been migrated. This is in contrast to other approaches (e.g., [25]) where the user program receives an exception and must update any references to the remote object that has been migrated.

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

Updating remote references in this fashion is a form of lazy updating, since the references are not updated until they are used. An eager update approach would involve up-

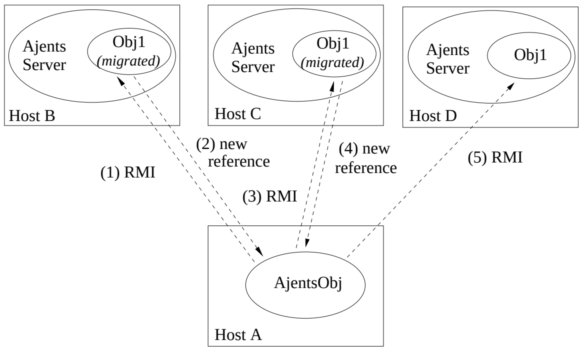

Figure 3: Remote Method Invocation After Third Party Migration

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

Object migration is accessible to the user using the syntax shown in Figure 4. In this example, checkpointing is enabled for the object in order to support migration during execution. The object is migrated, an asynchronous method is invoked and the object is migrated a second time. The return value is obtained from the future and a final RMI is performed on the migrated object. This example demonstrates that migration occurs transparently to the programmer. They may still use the same `AjentsObj` after migration and are unaware of whether the second migration occurred before or after completing the asynchronous remote method invocation.

Due to our reliance on the object serialization primitives in Java, there are some limitations on the type of objects which may be checkpointed and/or migrated by Ajents. Any object which is not serializable falls into this category, and may not be migrated by Ajents. This includes core Java API features such as threads and the AWT. We feel this limitation is reasonable since the serialization (and migration) of threads is impossible in the current JDK, and we can find

```
AjentsObj obj = Ajents.new(...);

// Parameters:  object, new host
Ajents.migrate(obj, sched.AvailServer());

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

// obj is checkpointed and method is invoked
Future future = Ajents.armi(obj, "method1");

// Migrate obj, possibly before armi completes
Ajents.migrate(obj, sched.AvailServer());

result1 = future.get();

// rmi on migrated obj
result2 = Ajents.rmi(obj, "method2");
```

Figure 4: Example of code performing object migration.

little reason to want to migrate any part of the AWT.

4 Performance Results

In order to demonstrate the practical relevance of Ajents we have evaluated the performance of Ajents using several micro-benchmarks and two versions of a simple (non-blocked) parallel application, matrix multiplication. Our findings demonstrate that the overheads introduced in using Ajents are not prohibitive. All experiments were conducted using a cluster of 8 SUN Ultra-1 workstations with 143 MHz UltraSparc CPU's acting as Ajents servers while another SUN Ultra was used as a client. All machines are connected with a 10 Mbps Ethernet network. However, the client does not share a file system with the Ajents servers requiring class files to be remotely loaded at the time of remote object creation or migration. All experiments were conducted using SUN's JDK, Version 1.1.6, which includes a just-in-time compiler.

Table 1 shows the results of experiments conducted using a simple (non-blocked) matrix multiplication benchmark. While this is clearly not the most efficient implementation of matrix multiplication, the same implementation is used in all cases in order to fairly compare the different environments. The first two rows of this table show the sequential execution times of versions of this program written in C and in Java (the columns show the results for different matrix sizes in integers). We note that across all matrix sizes the execution times of the Java version compare reasonably well with results obtained using C. The Java version is slower, by roughly a factor of 2. These results are quite interesting when compared with previous experiments [4] in which the difference between the C and Java versions was much larger. This provides some evidence that, as would be expected, moving to a just-in-time compilation environment significantly improves execution times. It is also possible that hot-spot compilation techniques will further reduce the gap in execution times. Although the Java version is slower, we believe that for a number of programmers and coarse-grained applications the benefits obtained from the ease with which Ajents programs can be implemented and executed across a wide variety of platforms will outweigh the costs of decreased execution times (relative to applications implemented in C).

The third row of Table 1 shows the execution time of the parallel version of the matrix multiplication program when executing using one remote server. By comparing the second and third rows of the table we see that the overheads incurred by Ajents, in switching from a simple sequential matrix multiply to a single-server Ajents version, is low.

The last two rows of Table 1 show the execution times for each matrix size when using all eight servers, and the corresponding speedups. The speedups obtained in these experiments are quite typical of those obtained in similar loosely coupled environments. While speedup is acceptable for smaller problem sizes, it continues to improve as the problem size grows, with a speedup of 7.6 obtained using 8 machines and a matrix size of 1024 x 1024 integers.

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

Table 1: Sequential C, sequential Java, parallel Java execution times for 1 and 8 servers, and speedups obtained. Times are in seconds using a simple (non-blocked) multiplication of two NxN matrices of integers. Speedup is the time on 8 servers versus sequential Java time.

We expect that Ajents will be commonly used to distribute long-lived objects among available machines, with remote execution requests occurring far more frequently than object creation or migration requests. Thus, it is vital that Ajents is able to perform RMI requests efficiently. Table 2 shows the results of our RMI request benchmark. A remote method was invoked repeatedly, passing an integer array of a specified size as a parameter. The results show the average time to complete a single remote method invocation (and thus differences may not be statistically significant). These results demonstrate that while there is overhead involved in making remote method invocations using Ajents, it is not significantly greater than that incurred when performing a regular Java RMI call.

Object Size	(ints)	1	1k	10k	100k
RMI - Ajents	(ms)	9	11	46	400
RMI - JDK RMI	(ms)	5	8	44	400

Table 2: Time to invoke an empty remote method using the base JDK RMI and Ajents. Object size denotes the number of integers contained in the array that is the only data member of the object that is passed as a parameter to the remote method.

A major feature of Ajents is the ability to migrate remote objects. Table 3 shows the times to migrate remote objects of various sizes. In conducting this experiment, we migrated an object which is composed of a single integer array. The object was migrated 64 times using 8 different machines (the 8 machines were used repeatedly in the same order). These results were then averaged to produce the results given in the table. We can see from these results that there is a significant increase in cost for object migration when compared with a remote method invocation. This can be accounted for by a number of factors. Migrating an object requires that an object be serialized, transfered to the target host and then deserialized. References to the object then need to be updated, including those needed by the Ajents server, the client, and the garbage collector. Additionally, when the object is migrated to a new host, class loading may be required.

Object Size	(ints)	1	1k	10k	100k
Migration time	(ms)	334	334	397	769

Table 3: Average remote object migration times (in milliseconds) to migrate on object 64 times across 8 hosts. Object size denotes the number of integers contained in the array that is the only data member of the object being migrated.

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

The results of these experiments, shown in Table 4, compare the execution times of the sequential matrix multiplication (done on a single server) with a sequential matrix multiplication forced to migrate to eight different machines. The first two rows of the table show the execution times for different problem sizes, while the last row shows the inflation factors for the different problem sizes (i.e., the ratio of the execution time including the migrations over the execution time without the migrations). The results show that even for matrices of moderate size the overheads associated with migrating the matrix objects a number of times does not significantly increase the total execution time. We found these results to be surprisingly good, especially because the computations involving a 1024 x 1024 matrix of integers must serialize, transfer and deserialize at least 12 MB of data during each migration (4 MB for each of the three matrices, assuming four bytes for each integer). Admittedly, these objects are not very complex and may be faster to serialize and deserialize than more complex objects. However, even with an increase in object migration times the advantages of migration will still outweigh the disadvantages for some applications.

We consider these results to be encouraging, considering that an additional job added to the same machine would result in an inflation factor of 2 (assuming round-robin scheduling with no overhead).

As previously described the use of checkpointing in Ajents enables the use of the checkpointing and roll back form of migration. However, the checkpointing of objects preceding a remote method invocation has an obvious negative impact on performance. Table 5 displays the results of a simple checkpointing benchmark designed to provide a rough idea of the costs involved in checkpointing. For this benchmark, an empty remote method was invoked repeatedly on an object whose sole member variable is an array

N	200	500	640	800	1024
1 Server	7.7	134	292	580	1305
8 Servers	20.3	165	344	666	1402
Inflation	2.64	1.23	1.18	1.15	1.07

Table 4: Serial matrix multiplication times (in seconds) for NxN matrices without migration overheads on 1 server and with migration overheads across 8 servers. Inflation is the ratio of the time on 8 servers to the time on 1 server.

of integers (the size of this array is changed for each experiment). The first two rows show the average time to complete a single remote method invocation with checkpointing enabled and without checkpointing enabled. The last row in the table shows the time difference between these two experiments. It represents the overhead added to an RMI call when checkpointing is enabled.

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

Table 5: Times (in milliseconds) to invoke an empty remote method without checkpointing and with checkpointing. Object size denotes the number of integers contained in the array that is the only data member of the object being checkpointed. Checkpointing cost is the difference between the checkpointing and no-checkpointing cases.

The results presented in Table 5 show that Ajents' checkpointing function performs in a reasonably efficient manner and as expected the overhead increases with the size of the object being checkpointed. While the cost of checkpointing is a multiple of the base cost of a remote method call, we believe that it is still within a reasonable range. Since the costs are considerably lower than migration costs (as shown in Table 3), the same relatively coarse grained applications whose execution times are not significantly impacted by migration would also not be significantly impacted by checkpointing. Our target applications are those whose method invocations made through Ajents will have a running time measured in seconds and minutes (as in the matrix multiply case), and thus the overheads due to checkpointing will not be significant for these cases. However, checkpointing will have a considerable impact when there are a large number of invocations on methods with relatively short execution times.

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

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

5 Discussion

We now outline some of the issues related to the current design and implementation of Ajents.

Although the mechanisms and functionality provided by Ajents are sufficient to support several types of parallel, distributed, and mobile applications, the system has and will continue to evolve. The current design and implementation has been greatly influenced by our overriding goals to produce a system that is as transparent as possible while ensuring that we do not modify the language or use preprocessors. In addition we want to ensure that Ajents and applications that use Ajents are compatible with existing Java compilers, stub compilers, and Java virtual machines. These decisions and the choice of Java have influenced the design, implementation and performance of our system.

Ajents' Java compatibility is achieved by making the synchronous and asynchronous RMI interface less transparent than approaches used by SUN's RMI [31] or JavaParty [20]. The mechanisms provided by `Ajents.new()`, `Ajents.rmi()` and `Ajents.armi()` calls imply that the interface for creating and interacting with remote objects is different from the local objects. This requires the application programmer to keep track of which objects are local and which are remote and to ensure that the proper interface is used for remote objects. This may not be a large disadvantage since it may be helpful for programmers to remember which method invocations are remote when performance is an issue.

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

An alternate approach[2] which we plan to explore would be to deploy a modified stub compiler that would automatically generate an additional asynchronous version of each method within a remote object. For example, `getName_async` could be generated and used for asynchronous calls while `getName` could be used for synchronous calls. Although this would require the use of a modified stub compiler and would require source for the classes, it would enable signature and parameter type checking at compile time and permit the use of `ref.getName()` to invoke both local and remote methods.

As can be seen by examining the overheads incurred when invoking remote methods (Table 2) and performing object migration (Table 3), the current implementation of Ajents is primarily useful for fairly coarse-grained applications. Recent research [29] that makes use of high-performance techniques for RMI and serialization has demonstrated that significant speedups can be obtained using a number of Java applications over a cluster of wide-area systems. The performance of Ajents will be significantly improved by advances made in Java virtual machine implementations, especially more efficient object serialization and RMIs [12, 15, 17, 29] and improved hot-spot compilation and garbage collection techniques. By deploying improved serialization and RMI techniques and by tuning the existing implementation of migration we believe that Java (and Ajents) may some day be able to compete with MPI implementations (provided JVM performance continues to improve).

[2] This approach was suggested to us by Bill Pugh.

A potentially serious problem arises in Ajents and in other systems which implement checkpointing for distributed and parallel applications. The problem arises when a remote object `A`, invokes a method, `M`, of object `B` which modifies that object's state. If object `A` is checkpointed, then it invokes method `M` of object `B` and it is later migrated and restarted on a different server; `A`'s execution will be rolled back to the checkpoint and continued from that point. Thus method `M` of object `B` will be called twice (rather than once). This is called the checkpointing consistency problem and is a well known problem [18, 30]. One intractable approach to ensuring checkpoint consistency would be to checkpoint all objects that object `A` could possibly interact with each time object `A` is checkpointed. Unfortunately, it is not possible to keep track of and checkpoint all such objects because some objects may not be known by or controlled by Ajents (e.g., local objects, objects that are communicated with using standard SUN RMI or using a socket).

At this point our system can be used to implement relatively simple applications. In order to fully support more serious distributed, parallel and mobile applications we are continuing to investigate techniques for adding features and examining issues such as: performance in wide area networks; finer control over checkpointing (e.g., setting checkpointing per method); fault tolerance; remote I/O; and improved scheduling (currently scheduling is done in a round-robin fashion). In particular, remote I/O for interactive input, files and sockets is being examined by Izatt [9].

6 Summary

Ajents is a system designed to make the implementation of parallel, distributed and mobile Java applications capable of seamlessly utilizing heterogeneous computing resources across an Intranet or throughout the Internet. Our implementation of Ajents currently provides simple and efficient mechanisms for the creation of remote objects, synchronous and asynchronous remote method invocations, as well as support for object migration.

Our experimental results show that we are able to achieve quite good speedup using a relatively simple implementation of a parallel matrix multiplication application. Our micro-benchmarks show that the overheads introduced by our implementation do not adversely affect remote method invocation times. In addition, we demonstrate that Ajents is capable of migrating relatively large objects several times without significantly impacting the execution time of objects that are performing a non-trivial amount of computation.

Ajents is implemented as a collection of Java classes and can be executed on any standard Java virtual machine. Absolutely no modifications to the Java language are made and no preprocessors, special compilers, or special stub compilers are required. We believe that the use of standard Java and the combination of features that Ajents supports, makes it an effective tool for programmers writing parallel, distributed or mobile Java applications.

7 Acknowledgements

We thank the anonymous referees for their helpful comments which have helped to improve this paper. Brecht and Izatt wish to thank the the Natural Sciences and Engineering Research Council (NSERC) of Canada for a grant and scholarship (respectively) that partially supported this research.

References

[1] A. Alexandrov, M. Ibel, K. Schauser, and C. Scheiman. SuperWeb: Towards a global Web-based parallel computing infrastructure. In *11th International Parallel Processing Symposium*, April 1997.

[2] E. Arjomandi, W. O'Farrell, I. Kalas, G. Koblents, F.C. Eigler, and G. Gao. ABC++: Concurrency and inheritance in C++. *IBM Systems Journal*, 34(1):120–136, 1995.

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

[4] T. Brecht, H. Sandhu, J. Talbot, and M. Shan. ParaWeb: Towards world-wide supercomputing. In *European Symposium on Operating System Principles*, October 1996.

[5] P. Chan. Ajents: A parallel and distributed Java system. Master's thesis, York University, 1998. (in preparation).

[6] B. Christiansen, P. Cappello, M.F. Ionescu, M.O. Neary, K. Schauser, and D. Wu. Javelin: Internet-based parallel computing using Java. In *ACM 1997 Workshop on Java for Science and Engineering Computation*, June 1997.

[7] G.C. Fox and W. Furmanski. Towards Web/Java based high performance distributed computing – an evolving virtual machine. In *Fifth IEEE International Symposium on High Performance Distributed Computing (HPDC5)*, Syracuse, NY, August 1996.

[8] S. Fünfrocken. Transparent migration of Java-based mobile agents (capturing and reestablishing the state of Java programs). In *Proceedings of Second International Workshop on Mobile Agents (MA'98)*, September 1998.

[9] M. Izatt. Babylon: A Java-based distributed object environment. Master's thesis, York University. (in preparation).

[10] M. Karaorman and J. Bruno. Active-rmi: Active remote method invocation system for distributed computing using active java objects. In *TOOLS USA 1998*, August 1998.

[11] H. Karl. Bridging the gap between distributed shared memory and message passing. In *ACM 1998 Workshop on Java for Science and Engineering Computation*, February 1998.

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

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

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

[15] J. Maassen, R. van Nieuwpoort, R. Veldema, H. E. Bal, and A. Plaat. An efficient implementation of Java's remote method invocation. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP'99)*, May 1999.

[16] M.O. Neary, S.P. Brydon, P. Kmiec, S. Rollins, and P. Cappello. Javelin++: Scalability issues in global computing. In *ACM 1999 Java Grande Conference*, pages 171–180, June 1999.

[17] C. Nester, M. Phillippsen, and B. Haumacher. A more efficient rmi for java. In *ACM 1999 Java Grande Conference*, pages 152–159, June 1999.

[18] R.H.B. Netzer and J. Xu. Neccessary and sufficient conditions for consistent global snapshots. *IEEE Transactions on Parallel and Distributed Systems*, 6(2), February 1995.

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

[20] M. Phillippsen and M. Zenger. JavaParty - transparent remote objects in Java. In *ACM 1997 Workshop on Java for Science and Engineering Computation*, June 1997.

[21] 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.

[22] M. Ranganathan, A. Acharya, S. Sharma, and J. Saltz. Network-aware mobile programs. In *Proceedings of USENIX'97*, 1997.

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

[24] L.F.G. Sarmenta, S. Hirano, and S.A. Ward. Towards bayanihan: Building an extensible framework for volunteer computing using java. *Concurrency: Practice and Experience*, 10(11-13):1015–1019, 1998.

[25] M. Straßer, J. Baumann, and F. Hohl. Mole - A Java based mobile agent system. In *ECOOP '96 Workshop on Mobile Object Systems*, 1996.

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

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

[28] Sun Microsystems, Palo Alto, CA. *Java Platform 1.1 Core API Specification*, 1998. http://java.sun.com/-products/jdk/1.1/docs/api/packages.html.

[29] R. van Nieuwpoort, J. Maassen, H.E. Bal, T. Kielmann, and R. Veldema. Wide-area parallel computing in java. In *ACM 1999 Java Grande Conference*, pages 8–14, June 1999.

[30] Y-M. Wang. Consistent global checkpoints that contain a given set of local checkpoints. *IEEE Transactions on Computers*, 46(4), April 1997.

[31] A. Wollrath, R. Riggs, and J. Waldo. A distributed object model for Java. In *2nd Conference on Object-Oriented Technologies and Systems (COOTS)*, pages 219–231, Toronto, Ontario, June 1996.

[32] W.M. Yu and A.L. Cox. Java/DSM: a platform for heterogeneous computing. In *ACM 1997 Workshop on Java for Science and Engineering Computation*, June 1997.