Journal of Parallel and Distributed Computing **60**, 1223–1237 (2000)
doi:10.1006/jpdc.2000.1651, available online at http://www.idealibrary.com on IDEAL®

Java Class Broker—A Seamless Bridge from Local to Distributed Programming

Zvi Har'El

Department of Mathematics, Technion—Israel Institute of Technology, Haifa 32000, Israel
E-mail: rl@math.technion.ac.il

and

Zvi Rosberg

Radware Ltd., Atidim Technological Park, Building 1, Tel-Aviv 61131, Israel
E-mail: ZviRo@radware.com

Received November 15, 1999; revised March 10, 2000; accepted March 23, 2000

Distributed object programming is significantly more complex than programming a local host and requires highly skilled developers. Current distributed middleware for distributed programming is hard to use mainly because its programming model and runtime support are quite different from those of local programming. For instance, the local reference and the remote reference to the same object are of different types and therefore are not assignable to the same set of references. Hence, a remote reference cannot always replace a local reference as a parameter in a method invocation. Furthermore, a local object cannot be referenced remotely, unless it has been first converted into a remote object. Another distributed programming obstacle with current middleware is that access to classes and resources residing across the distributed environment is not as natural and transparent as with local programming, where all resources are in the same classpath. The *Java* language introduces a new scope where distributed object programming can become as easy and simple as local programming. In this paper we identify the main distinctions between local and distributed *Java* programs and present new middleware which achieves this goal. The proposed middleware converts any *Java-enabled* host into a *Java peer*, which can share its classes and object instances and interact with other *Java peers* in a manner which almost reflects a *single system image* to the user. © 2000 Academic Press

1. INTRODUCTION AND PRELIMINARIES

Distributed object programming differs from local object programming in its syntax, semantics, deployment, and runtime support. As a result, even with current

0743-7315/00 $35.00
Copyright © 2000 by Academic Press
All rights of reproduction in any form reserved.

frameworks, distributed object programming is significantly more complex than local programming. For instance, the local reference and the remote reference to the same object are of different types and therefore are not assignable to the same set of references. Hence, a remote reference cannot always replace a local reference as a parameter in a method invocation. Furthermore, a local object cannot be referenced remotely, unless it has been first converted into a remote object. Another distributed programming obstacle with current middleware is that access to classes and resources residing across the distributed environment is not as natural and transparent as with local programming, where all resources are in the same classpath. The wide-spread *Java* language [JLS96] introduces a new scope where distributed object programming can be greatly simplified. In this paper we present a new pure and light framework which exploits Java's reflection and class loading to make distributed programming as easy as local programming. The proposed framework converts any *Java-enabled* host into a *Java peer*, which can share its classes and object instances and interacts with other Java peers in a manner which almost reflects a *single system image* (SSI) [AFT99]. Throughout this paper we explain how close to an SSI our new framework is.

To be definite we confine ourselves to Java [JLS96] and its class libraries [JCL98] and begin with a definition of local and distributed programs. Then we proceed by identifying nine major distinctions between local and distributed programs. Related projects are discussed at the end of this section. The new framework, referred to as *ClassBroker for Java*, is presented in Section 2, and its core features are explained in Section 3. The project status and our implementation experience are given in Section 4.

1.1. *Local and Distributed Programs*

A *program* is a collection of objects and execution threads. A *local program* is a program which is executed in a single Java virtual machine (JVM), whereas a *distributed program* is a program which is executed in multiple JVMs.

As Java objects are accessed only by reference, we use the two following key notions to distinguish between local and distributed programs. A *local reference* to a Java object is a pointer to a (*master*) *object* in the current JVM, which is implemented by the JVM (see [JLS96, Sect. 4.3.1]). Note that a local reference is *unique*; i.e., if *obj1* and *obj2* reference the same object, then (*obj1* = = *obj2*) returns true. Furthermore, in *local method calls*, parameters and return values are passed only by reference.

A *global reference* to a Java object is a pure Java object which points to its master object, regardless of the master object location. Invoking the master object methods via its global reference is referred to as making *remote method calls*. There are various ways to pass parameters and return values in remote calls. We confine ourselves to Java's remote method invocation (RMI) scheme [RMI], where objects are passed and returned only by copy. Observe, however, that for all referencing practice, passing a global reference by copy is similar to passing its master object by reference. The reference type, though, is different. Note also that the RMI scheme requires that the parameters and return values of remote calls be *serializable* (see [JCL98, p. 1508]). Roughly speaking, a *serializable object* is an object which can be converted into a byte stream and be reconstructed back into a copy of the object. A crucial property of a *serializable Java object* is that the first nonserializable class in the object class hierarchy must have a constructor which accepts no parameters; otherwise an *InvalidClassException* is raised during deserialization.

1.2. Distinctions between Local and Distributed Programs

The distinctions between local and distributed programs may depend on the underlying distributed program framework. Some frameworks may hide some of the distinctions while others may add new ones. To be concrete, we take Java's RMI framework [RMI] as the frame of reference to specify the distinctions.

Global References. The *predominant distinction* is that a distributed program contains global references besides the intraJVM references of a local program.

Field Referencing. The *second distinction* is that an object field *f* can be referenced by the expression *obj.f* if *obj* is a local reference, but not if *obj* is a global reference. This holds true for RMI, as well as for other frameworks which do not recompile the application or modify the JVM.

Class Instantiation. The *third distinction* is the syntax by which an object is instantiated. A local instantiation of *MyClass* is done with the statement *MyClass cl* $=$ *newMyClass*(...). A remote instantiation of *MyClass* must be done through a special remote method call.

Callback Thread Context. The *fourth distinction* is the way the thread context is preserved in recursive callbacks. A *recursive callback* is a call *obj1.foo*(...) made from *obj2*, which subsequently follows a call *obj2.moo*(...) made from *obj1*, before the latter returns. If *obj1* and *obj2* are local references, then method *foo*(...) is executed by the same thread as method *moo*(...). If the references are global, then in RMI (and in any other framework we are aware of), they are executed in different threads. Observe that if *obj1* and *obj2* are global references and *foo*(...) and *moo*(...) are synchronized methods, then a deadlock will occur.

Locking Enforcement. The *fifth distinction* is the locking mechanism. When a lock is acquired by a JVM thread, all the threads in that JVM follow the local locking policy. This policy is not enforced on the program threads which are executed in other JVMs. To obtain a global locking policy for a distributed program, the framework must be enriched with new lock types. Note that without global locks, distributed programs may be deadlocked in certain scenarios. For example, assume two interacting JVMs where a mutual exclusion lock is acquired in each one of them. Then, when a remote call is made from each one to the other, requesting the locks again, the deadlock will occur.

Class Loading. Class and resource loading is the *sixth distinction.* A JVM has a *system class loader* which loads classes and resources from the system classpath. In addition, local programs can define one or more class loaders which can load classes from other locations. When a class object is first referenced from an object, say *obj*, the referenced class object is loaded by the class loader which has loaded the class object of *obj*. A class loader recognizes only the classes which it loads. In a local call, the parameter classes, which are resolved by the caller, are necessarily known by the called object—similarly with return value classes. In a remote call, this may not be true as the two JVMs use different class loaders. Resources, such as images and files which are used by a parameter which is passed in a remote call, may also not be available in the receiving JVM. This situation would not happen in local calls.

Reference Assignment. The *seventh distinction* is between the local and global reference types. A *global reference* is implemented by an object and assumes its type, which is different from its master object type. Thus, it may not be assignable to the types to which its master object is assignable. In RMI, and in most other frameworks, the global reference object implements one or more interfaces which its master object implements. Hence, it is assignable to those interfaces, but not to the master object type. As a result, passing by reference parameters (and return values) in remote calls may result in class cast exceptions.

Exception Handling. The *eighth distinction* is exception handling. As exceptions in remote calls may occur in the remote JVM; they do not percolate up automatically in the method call hierarchy all the way to the JVM which has made the remote call. Furthermore, the stack trace data (which is stored in a native data structure) is not passed to the calling JVM.

Garbage Collection. The *ninth distinction* is garbage collection. In a local program an object is unreachable if it is no longer referenced from any reachable local object. In a distributed program, an object may have global references from other JVMs, and therefore still being reachable in a distributed program context. This requires us to extend the local native garbage collection mechanism to a distributed one. A closely related distinction is the behavior of system exit. In a local program it terminates the application and cleans all its resources. In a distributed program it terminates only the JVM where the system exit is called.

1.3. A Portable and Seamless Framework with Polymorphic Global References

We believe that a Java distributed framework should be *portable*. That is, it should be made available to and should connect any arbitrary set of JVMs, and it should run applications without recompilation. Such portability is particularly important if Web browsers are part of the JVM network. A distributed JVM which transparently distributes local programs, on the other hand, is not portable in that sense. Its objective, however, is different from that of a portable distributed framework, i.e., to accelerate program execution. In a portable framework, arbitrary objects and threads which are placed in separate JVMs can interact with each other. Note also that parallelism and distribution in a distributed JVM are limited to the underlying cluster, whereas in a portable framework, it is unlimited. Note also that permitting recompilation is not aligned with a truly portable framework, although it can eliminate some distinctions between local and distributed programs. If recompilation is done statically, then classes which have not been recompiled cannot be used during runtime. These may include standard class libraries and classes which are generated on-the-fly. If recompilation is done during class loading, then it requires a special purpose class loader, whose setting is restricted by the security manager.

A *seamless* framework is another useful requirement—that is, a framework in which any class and resource in the distributed domain is transparently accessible from anywhere in the distributed program. In particular, any class can be instantiated anywhere and from everywhere, and its methods can be invoked remotely. Observe that when parameters are passed by reference in remote invocations (without "remote enabling" the called object), a global reference must be assignable to its master object type. We refer to this property as *polymorphic global references*.

Seamless and *reference polymorphism* are essential for collaborative computing with existing class libraries and for automatic program partitioning and load balancing—for the former, because it makes distributed programming possible to ordinary developers, and for the latter, because it facilitates the algorithms.

1.4. Related Frameworks

The basic Java framework for distributed programming is RMI [RMI], which is part of the Java class library. As such it may serve as a frame of reference to other frameworks. RMI remote objects must be "remote-enabled," a procedure which includes a remote interface definition, pregeneration and placement of *stub* (Java 1 and 2) and *skeleton* (Java 1 only) classes, and special code segments to export remote objects. An RMI stub is used as a remote reference to its master object whose type is in a different class hierarchy, and therefore not assignable to its master object type. It is a *plain proxy*; namely, it just delegates the method calls to its master object. Parameters and return values (including stubs) are passed by copy using Java's serialization. Runtime loading of stubs, skeletons, method parameters, and return value classes can be done *only* from a prespecified Web server codebase. Images and files, or other classes, cannot be loaded by the RMI runtime. With RMI, remote objects are created and exported only in the current JVM, and their remote references are obtained from a different JVM by a naming service.

Another framework which is supported in the Java class library is IDL, the Java language interface to CORBA. As a comprehensive multilingual standard for most of the distributed services, CORBA presents a quite complex framework which is far from being close to a single system image.

Java// [CKV98] is a pure Java framework whose objective is similar to ours. With Java//, each remote object owns its execution thread, and parameters and return values are passed by reference (except for primitives, final classes, and objects which may throw exceptions) using *future objects* (see definition there). The future objects technique may inflict a significant performance penalty in light to medium platforms and it prevents proper exception handling. Among the distinctions above which are not addressed by Java// are callback thread context, locking enforcement, class loading, exception handling, and garbage collection. Other differences between our framework and Java// are the following. The transport layer of Java// is above Java's RMI, whereas ours is proprietary and is built to address the *callback thread context* distinction, to optimize performance, and to fully control the socket creation. The latter makes it easy to use secure sockets and reroute messages. Another difference is that object creation syntax in Java// does not enable to resolve an ambiguous constructor upon remote creation of an object.

A commercial framework for distributed programs which generates remote references on-the-fly is Voyager [Vo]. Unlike our framework, Voyager does not address the distinctions callback thread context, reference assignment, and exception handling and only partially address the distinction class loading.

JavaParty [PhZe97] is another framework for parallel distributed computing with Java. The main difference between JavaParty and our framework is that JavaParty uses a Remote modifier which is not part of the Java language. Consequently, JavaParty is not portable and seamless since the new modifier requires a specific compiler and a specific distributed JVM. As in Java//, JavaParty's transport layer is also built over RMI, whose shortcomings are expressed by the authors.

Another parallel distributed computing project is *Do*! [LaPa97], which adds new classes to support parallel programming semantics. Do! uses a preprocessor to transform a parallel program into a distributed program by mapping objects to processors and adding remote creation commands. Do! uses a standard JVM and RMI to distribute the processing over a TCP/IP network.

In [KBW], a different design and implementation which extend Java with parallel constructs and do not require modification to Java compilers and JVMs are used. In this framework, remote objects are created through their predefined proxy classes, which are generated offline by a special compiler. The constructors and the methods of remote classes in [KBW] accept only objects of a special type, *Message*, and pass them only by copy. Moreover, remote methods are executed asynchronously, and cannot return a value. Again, distribution and remote method invocation are built on top of RMI, and to run a parallel application over a network, a native interpretability framework (Converse) is needed.

A recent interesting distributed JVM implementation over a cluster is the cJVM project reported in [AFT99]. Two other distributed JVM projects which are reported in the literature are Java/DSM [YuCo97] and Hyperion [MMH98]. The objective of the distributed JVMs is to execute unaltered pure Java applications (only) more efficiently than in a single host. The differences between a distributed JVM and a portable framework as ours is explained in Section 1.3 above.

2. THE CLASS BROKER FOR JAVA

We motivate and explain our ClassBroker programming model by considering the following fictitious but illustrative distributed concert example. *Maestro M*. from host *Mland* wishes to orchestrate an international concert to be broadcast from host *Bland*, in which the performing orchestras, leading voices, and choruses will be switched during the performance upon his scepter command. The performers are all located throughout several *MusicCenters*.

The classes which implement the concert are already defined and specified as follows. The *Concert*, *Orchestra*, and *Chorus* classes have all public constructors and methods:

```
public class Concert {
   public Concert(Orchestra orch, LeadingVoice lv, Chorus cho) {...}
   public void start() {...}
   public void stop() {...}
   public void switch(Orchestra orch, LeadingVoice lv, Chorus cho) {...}
}
public class Orchestra {
   public Orchestra(Instrument piano,... Instrument violin) {...}
}
public class Chorus {
   public Chorus(Voice tenor,... Voice bass) {...}
}
```

The leading voices, however, as expected, have private constructors and can be instantiated only through their private managers, whose constructors, on the other hand, are public. The opera singers and their private managers, e.g., are defined by the following pair of classes:

```
class PrimaDona implements LeadingVoice {
   private PrimaDona (String name) {....}
   public boolean pleaseDo (Request req) {
        return false;
   }
}
public class PrivateManager {
   public PrivateManager (String name, Long commission) {...}
   public PrimaDona getPrimaDona(Double dollars) {....}
}
```

The Maestro local application (where all classes, objects, and threads are in the same JVM) is given by

```
public class Maestro {
   public static main(String args[ ]) {
      LeadingVoice maria=
         (new PrivateManager (“Maria”), 10000)).getPrimaDona(100000);
      Concert con=Concert(new Orchestra(...), maria, new Chorus(...));
      con.start();
      sleep(5000);
      LeadingVoice faverotti=
         (new PrivateManager (“Luciano”), 20000)).getPrimaDona(200000);
      con.switch(null, faverotti, null);
     ....
   }
}
```

Since performers cannot be cloned and the Maestro wants to conduct the distributed concert from its host only, a distributed concert would not be possible without the support of the following framework.

2.1. Facilitating a Seamless and Polymorphic Framework with Smart Global References

To address the restrictions imposed by the performers and the maestro, the following devices are needed:

1. *RemoteCreator*: A factory to create objects in remote hosts. The returned global references should be assignable to their master objects' type, e.g., an API to create from Mland a Concert instance in Bland, as well as PrivateManager, Orchestra, and Chorus instances in any other host.
2. *TypeTranslator*: A mechanism to transparently translate a global reference into a local one, and vice versa. For example, give that *g_pm* is a global reference to a remote PrivateManager object, *g_pm.getPrimaDona*(...) should return a global reference to a *PrimaDona* object, rather than a local one. Furthermore, as with the RemoteCreator, the global reference should also be assignable to PrimaDona.

Observe that a RemoteCreator is not sufficient to get a global reference to a PrimaDona instance, since the latter has a private constructor. By adding TypeTranslator, this can be resolved as follows. First, a remote instance of PrivateManager is created by using the following APIs:

```
// Get a Broker instance to the LeadingVoice host
ClassBroker cb=ClassBroker.getBroker(....);
// Create a remote instance that returns a global reference
// which is assignable to PrivateManager
PrivateManager g_pm=(PrivateManager) cb.create ("PrivateManager", params);
```

The first call gets a factory instance to a specified remote host, and the second call uses its "remote-create" method to get a global reference, *g_pm*, to a PrivateManager instance which is assignable to a PrivateManager type. Now, with TypeTranslator, a global reference to a PrimaDona is obtained by *g_pm.getPrimaDona*(...). The type translation is done by our smart global reference. This translation role is built into it upon its creation. Without such type translation, *g_pm.getPrimaDona*(...) would return a clone of PrimaDona and not the "real thing." A simple way to build intelligence into the *g_pm* global reference is to replace its creation call

```
PrivateManager g_pm=(PrivateManager) cb.create("PrivateManager", params);
```

with a call

```
PrivateManager g_pm=(PrivateManager) cb.create("PrivateManager",
                                        params, "PrivateManagerReplacer");
```

Here, the PrivateManagerReplacer is an interface which specifies the type translation rules when the methods of *g_pm* are invoked; e.g., replace the return value type of *getPrimaDona*(...) with a global reference which is assignable to *LeadingVoice*. The PrivateManagerReplacer interface could also be used to specify parameter type translation. With the RemoteCreator and Reference Translator devices, the local Maestro program can be converted into a distributed one, without changing any code in the other hosts, as follows:

```
public class Maestro  {
    public static main(String args[ ] ) {
       // Step 1: Get a Broker instance to Maria's host
       ClassBroker cb_1 =  ClassBroker.getBroker(....);
       // Step 2: Create a remote instance which returns a global reference,
       //         assignable to PrivateManager
       PrivateManager g_pm=(PrivateManager)cb_1.create("PrivateManager",params,
                                                      "PrivateManagerReplacer");
       LeadingVoice g_maria =  (LeadingVoice) g_pm.getPrimaDona(100000);
       // Step 3: Get a Broker instance to an Orchestra host
       ClassBroker cb_2 =  ClassBroker.getBroker(....);
       // Step 4: Create a remote Orchestra which returns a global reference,
       //         assignable to Orchestra
       Orchestra g_orch = (Orchestra)cb_2.create("Orchestra",params);
       // Step 5: Create a local Chorus object
       Chorus cho = new Chorus(...);
       // Step 6: Get a global reference to the cho object,
       //         assignable to Chorus
       Chorus g_cho = cb_1.getGlobalReference("Chorus", cho);
       // Step 7: Get a Broker instance to the Bland host
       ClassBroker cb_3 =  ClassBroker.getBroker(....);
       Object params[ ]  = {g_orch, g_maria, g_cho};
       Concert g_con = (Concert) cb_3.create("Concert", params );
       g_con.start();
       sleep(5000);
       // Step 8: Get a Broker instance to the Luciano's host
       cb_1 =  ClassBroker.getBroker(....);
       // Step 9: Create a remote instance which returns a global reference,
       //         assignable to PrivateManager
       g_pm=(PrivateManager) cb_1.create("PrivateManager",params,
                                         "PrivateManagerReplacer");
       LeadingVoice g_faverotti =  (LeadingVoice) g_pm.getPrimaDona(200000);
       g_con.switch(null, g_faverotti, null);
       ....
       ....
     }
}
```

Observe that in the distributed program above, separate ClassBroker instances are used for different remote hosts. In addition, in Step 6, one of the ClassBroker instances is used to get a global reference which is assignable to Chorus, for an existing Chorus object. This API is necessary when a global reference to a local object should be passed, rather than a copy of it.

To summarize, after the initialization of ClassBroker objects, our portable, seamless, and polymorphic framework provides APIs such as *ClassBroker.create*(...) and *ClassBroker.getGlobalReference*(...), where the exact locations of the master objects, classes, and resources are transparent. Moreover, the classes may reside at any host participating in the distributed program, that is, to provide a single system image. A portable implementation as specified above cannot achieve a complete SSI, but can come very close to it. In the next section we describe and discuss how this is done with ClassBroker for Java.

3. CLASS BROKER CORE FEATURES

In this section we describe the core features of our ClassBroker framework which narrow the distinctions between local and distributed programs. Since we require a portable solution, all services must be implemented with pure Java classes.

3.1. Smart Global References

Since a standard JVM does not support global references we implement them with proxy classes. To resemble local references we require that they be implicit, unique, and symmetric. By *implicit* we mean that they should be created "on-the-fly" and "under-the-cover" as implied from the API context (e.g., upon a "remote-create"). By *unique* we mean that if *g_obj1* and *g_obj2* are two global references to the same object, then the Java expression (*g_obj1* = = *g_ob2*) returns true. By *symmetric* we mean that they preserve the required symmetry property of equality. Uniqueness not only keeps one reference object, but also preserves local JVM behavior of the " = = " operator. The latter property is motivated by the following common user programming style.

Let *g_el* be a global reference to an EventListener object instance in JVM1 which has been registered to a remote EventGenerator in JVM2 using the remote call *addListener*(*g_el*). A common way to manage listeners is to keep them in some data structure. When a *removeListener*(*g_el*) is called later, a while loop (*g_el* = = *listener*) over all listener objects is performed. (According to [JLS96], a reference is unique; thus using the " = = " operator rather than *gel.equals*(*listener*) is a natural choice.) If global reference uniqueness is not preserved, then *removeListener*(*g_el*) will not remove the listener which has been registered by *addListener*(*g_el*). The reason is that Java deserialization generates a new object for every object which is passed remotely.

The global reference polymorphism as defined in Section 1.3 above is another useful property which is motivated by the scenario given in the Maestro example above. This property is not free of implementation issues. As we implement this property by subclassing the master object class, the first issue is which one must implement all methods inherited from the master class, from that only a small subset may be required by the user. Another issue may pop up when a global reference object is instantiated, as its initializer calls one of its master object constructors. The latter may execute some undesirable code. A third issue may arise when a global reference object is deserialized (when passed by copy to another JVM). If the first nonserializable class in the class hierarchy of the master class does not contain a constructor with no parameters, an exception will be raised. Due to these three issues we also provide an option by which global references only implement a specified interface.

Observe that global references are generated during runtime. This may be forbidden in some applications, e.g., browsers. In such cases, class generation is transparently delegated to another host which runs ClassBroker and has no such limitation. The class is then loaded by the ClassBroker runtime class loader or by the applet class loader.

To summarize, the global references of the ClassBroker for Java are implicit, symmetric, multiple typed, unique (up to the global reference type), and smart (in the sense that they can be set to translate a local reference into a global one). As a result, the global references and reference assignment distinctions from Section 1.2 are substantially narrowed.

3.2. Class and Resource Loading

As global reference classes are generated on-the-fly and in a distributed environment, global references, as well as other objects, are passed from one JVM to another; their classes must be loaded into the receiving JVM. Besides classes, other remote resources such as images and files may be needed.

To resemble a SSI transparent class loader and address potential security manager denial, the loader we implemented satisfies the following properties:

1. *Joint-classpath*: All classes and resources in every JVM which participates in a distributed program should be accessible to all JVMs.
2. *Application-defined-class resolution*: A class which has been defined by an application class loader in one JVM should be made available to all other JVMs.
3. *Transparent delegation to Application ClassLoader*: If a class loader construction is denied, class loading should transparently be delegated to the underlying class loader (if it exits).
4. *Class prefetching*: In cases where an array of objects are passed in a remote call, its element Class object should be prefetched.
5. *Resource loading*: Resource files which are referred from any object should be made available to any JVM in the distributed program.

These properties represent an environment with a fully shared set of classes and resources. In distributed environments, however, there are cases where class sharing should be limited to a subset of JVMs, allowing different sessions to use different versions or implementations of the same class. ClassBroker supports both options and lets the user select between the two.

To summarize, the distributed class and resource loader above substantially narrows the class loading distinction from Section 1.2.

3.3. Thread Context Preservation in Recursive Remote Callbacks

To eliminate the callback thread context distinction from Section 1.2, the ClassBroker is using a patent pending mechanism which is built into the remote invoker layer. It ensures that every subsequent remote callback generated from a remote call before its return is switched to the thread and priority of the originating remote call. This feature liberates the applications from taking care of deadlocks which do not occur in local programs, hence making remote calls more similar to local calls.

3.4. Distributed Garbage Collection

One of Java's key features is its transparent garbage collection which frees the application from managing its memory. To narrow the garbage collection distinction, an SSI framework for distributed programs should also free the distributed application from managing its distributed memory. ClassBroker implements a transparent distributed garbage collection using a weighted-count algorithm with a dynamic total weight value. Note that a dynamic total weight value is needed to cope with the situation where the number of global reference copies exceeds its initial setting. A disposal of a global reference object is caught in its *finalize*() method. To make sure that *finalize*() is always called, ClassBroker runtime forces the *runFinalizersOnExit*(*true*) option.

3.5. Remote Exceptions and Their Stack Traces

Exception and *Error* objects are used to indicate that exceptional situations have occurred. Typically, these objects are freshly created in the context of the exceptional situation to include relevant information such as stack trace data. The stack trace contains a snapshot of the execution stack of its thread at the time it was created and is being filled by a native function into a native data structure. Exceptions can be thrown by the JVM or by the Java throw statement.

When an exception is thrown in a local program, it percolates up the method calling hierarchy until caught by a Java catch statement. If it is not caught, the thread which encounters it is terminated. In a distributed program, a remote method call may be initiated in one JVM, and the exception may occur in another. Our SSI exception handling delegates the exception, along with its stack trace, back to the thread which has initiated the remote call and rethrows it there. Moreover, the Exception type we throw is assignable to the type of the original exception, and its printStackTrace() method prints the original stack trace data. To implement these properties, ClassBroker generates on-the-fly subclasses of such exceptions, whose printStackTrace() method prints the remote stack trace data. This feature narrows the exception handling distinction from Section 1.2.

3.6. Remote Static Method Invocation

As stated above, the main objective of the ClassBroker is to provide a distributed framework which is as close as possible to a SSI framework. This also implies that if an object can be created locally, it should also be possible to create it remotely and to get a global reference to it. The following example demonstrates that smart global references is not sufficient. Consider the following version of class PrivateManager which is defined above:

```
public class PrivateManager1 {
      private PrivateManager1 (String name, Long commission) {...}
      public static PrimaDona getPrimaDona (Double dollars) {....}
}
```

PrivateManager1 has a private constructor and a static method to create PrimaDona objects. Therefore, the only way to remotely create a PrimaDona object is to support a remote call to a static method. To do so, ClassBroker provides the following two instance methods:

```
ClassBroker.invokeStatic(String method, String class, Object[] params);
ClassBroker.invokeStaticReturnRemote(String method, String class,
                                     Object[] params, String returnType);
```

The first API returns the object by copy and the second by (global) reference.

3.7. Most Specific Constructors and Methods

In local programs, calls to methods and constructors are done by specifying the method or the constructor name along with the parameters. Since the Java language supports polymorphism, this could result in ambiguity, since more than one method or constructor may meet the specified method and parameter types. In local programs, such ambiguity can be resolved during compilation by casting the parameters to the required types. In our distributed framework where global references are created during runtime, remote object creation and remote static method invocation ambiguity are resolved by providing optional APIs where the parameter types can be specified.

3.8. Nevertheless, Still Not SSI

In conclusion, ClassBroker eliminates six of the nine distinctions from Section 1.2. Three distinctions still remain: field referencing, class instantiation, and locking enforcement. The first two cannot be eliminated due to our portable framework requirement defined in Section 1.3, and the third one could be too costly performancewise. In this respect, our ClassBroker framework is not a fully SSI framework, but comes quite close.

3.9. Other Features

Very often, distributed programs may consist of applets running inside the Java security sandbox model. In Sections 3.1 and 3.2, we already discussed the restrictions imposed by the sandbox security model on two basic features, class generation and class loading. We also described the solutions taken by ClassBroker. There are other crucial methods which must be used by a framework implementation which are subject to denial by the applet SecurityManager or may throw exceptions if called more than once, e.g., method *setSocketFactory*(...), of class ServerSocket and method *setURLStreamHandlerFactory*(...) of class URL, which are called by the browser running the applet. An example of a function which a framework may wish to use and be denied by the applet SecurityManager is to open a socket to a host different than the codebase. In general, security exceptions can be avoided by using *signed applets*. However, since different browsers have different models for security privileges, this solution is not portable. Therefore, it is better to provide portable bypasses whenever possible. ClassBroker provides bypasses for setting secure sockets and for message rerouting to hosts other than the codebase.

A distributed framework also needs a mechanism to restrict remote access from other JVMs. ClassBroker runtime consults with a preset RemoteAccessController object before any object instantiation or method (static and instance) invocation which has been initiated from a remote JVM. The RemoteAccessController class is an application implementation which enforces the remote access policy. A Remote AccessController object can be reset at any time. It can be set in a global context, namely, applied to all connecting JVMs, and can be set on a connection basis. The latter allows application of different policies to different connecting JVMs.

4. STATUS, IMPLEMENTATION EXPERIENCE, AND CONCLUSION

ClassBroker has a mature and complete implementation which can be downloaded from http://www.alphaworks.ibm.com/tech/jcbroker. It is part of the Java server platform shipped with the IBM AS/400 V4R5 operating system, and it is used as the distributed framework for a forthcoming IBM tool to manage AIX/6000 systems.

In our implementation we emphasized three major elements: bytecode size, performance, and being able to run as an unsigned applet. We restricted ourselves to an uncompressed bytecode of 200 Kbytes which can comfortably fit into a thin client (network) desktop. We achieved a size of about 160 Kbytes for applications and about 100 Kbytes for applets. This requirement forced very careful design and development of our own runtime compiler to generate global reference classes. Existing compilers were too large.

Fast generation of global reference classes was another requirement which convinced us to implement our own compiler, since existing ones either are too slow or are implemented in native code. The speed of our class generator compiler is comparable to that of the Symantec native compiler. Performance and thread context preservation upon recursive callbacks were the two main reasons for implementing our own transport layer for remote method invocation. An earlier attempt to build it on top of Java's RMI was not sufficiently fast. Another reason for using our transport layer is to have better control of how sockets are being opened and closed. This has a crucial impact on performance (especially for secure sockets) and on security violations when running as an applet.

The size, efficiency, complexity, and usability of existing frameworks for distributed programs have motivated us to specify and implement an alternative one. The beacons we have followed were openness and simplicity as expressed by the SSI concept. The ultimate and final criterion for every design and implementation decision was "How close would the distributed program be to its local version?" Secondary considerations, but still important, were portability, bytecode size, and performance.

The wide use of the Java language and Java-enabled Web browsers led us to select Java as the only supported programming language. This is not much of a restriction as even in enterprise environments, Java is taking the role of being a mediating layer to native applications.

ACKNOWLEDGMENTS

The authors thank their colleagues, Y. Aridor, O. Biran, M. Factor, and E. Farchi for valuable feedback during the design and testing of the ClassBroker framework.

REFERENCES

[AFT99] Y. Aridor, M. Factor, and A. Teperman, cJVM: A single image system of a JVM on a cluster, *in* "Proceedings of '99 IEEE International Conference on Parallel Processing, ICPP'99," to appear.

[CKV98] D. Caromel, W. Klauser, and J. Vayssiere, "Towards Seamless Computing and Metacomputing in Java," Technical Report, INRIA, 1998, *available at* http://www.inria.fr/sloop/javall.

[JCL98] M. Chan, R. Lee, and D. Kramer, "The Java Class Libraries," 2nd ed., Vols. 1 and 2, Addison–Wesley, Reading, MA, 1998.

[JLS96] J. Gosling, B. Joy, and G. Steele, "The Java Language Specification," Addison–Wesley, Reading, MA, 1996.

[KBW] L. V. Kale, M. Bhandarkar, and T. Wilmarth, "Design and Implementation of Parallel Java with Global Object Space," Technical Report, Department of Computer Science, University of Illinois, Urbana, IL.

[LaPa97] P. Launay and J.-L. Pazat, "A Framework for Parallel Programming in Java," Technical Report 1154 IRISA, December 1997.

[MMH98] M. MacBeth, K. McGuigan, and P. Halcher, Executing Java threads in parallel in a distributed memory environment, *in* "IBM Center of Advanced Studies Conference," Canada, November 1998.

[PhZe97] M. Philippsen and M. Zenger, JavaParty—Transparent remote objects in Java, *Concurrency*: *Practice Exper*. **11**, No. 9 (1997), 1125–1142.

[RMI] http://www.javasoft.com/rmi.

[Vo] http://www.objectspace.com/products/prodVoyager.asp.

[YuCo97] A. Yu and W. Cox, Java/DSM, A platform for heterogeneous computing, *in* "JCM 1997 Workshop on Java for Science and Engineering Computation," June 1997.