

A Bytecode Translator for Distributed Execution of “Legacy” Java Software

Michiaki Tatsubori¹, Toshiyuki Sasaki^{1*}, Shigeru Chiba^{1,2**}, and Kozo Itano¹

¹ University of Tsukuba, Japan

² PRESTO, Japan Science and Technology Corporation

Abstract. This paper proposes a system named Addistant, which enables the distributed execution of “legacy” Java bytecode. Here “legacy” means the software originally developed to be executed on a single Java virtual machine (JVM). For adapting legacy software to distributed execution on multiple JVM, developers using Addistant have only to specify the host where instances of each class are allocated and how remote references are implemented. According to that specification, Addistant automatically transforms the bytecode at load time. A technical contribution by Addistant is that it covers a number of issues for implementing distributed execution in the real world. In fact, Addistant can adapt a legacy program written with the Swing library so that Swing objects are executed on a local JVM while the rest of objects are on a remote JVM.

1 Introduction

Object-oriented distributed software can be developed with various programming tools and environments. For example, a number of object request brokers have been proposed[8,9,10,21], just to mention a few, and they allow programmers to easily make an object accessed by a remote host through a network. The programmers only have to define the interface of the object in an interface definition language. Another example is to use a distributed programming language like Emerald[3]. Such a language provides language constructs for creating objects on remote hosts, migrating them to another host, and so on.

However, these programming tools and environments are mainly for developing new distributed software from scratch; they are not for adapting “legacy” software to distributed execution on multiple hosts. Here, “legacy” means that the software was originally developed with intent to be executed on a single host. The existing tools or environments are not helpful in modifying the legacy software so that part of the software can be executed on a remote host. The programmers have to manually modify the source text of the program to follow a programming conventions specified by the tools, or to use special language constructs. This modification takes long time and it is error-prone. It is even impossible if the program text is not available or modifiable. Practical demands

* Currently, Hitachi Ltd., Japan.

** Currently, Tokyo Institute of Technology, Japan.

for adapting legacy software to distributed execution will never disappear. While there is already a number of such legacy software, programmers will continue to develop legacy software since non-distributed software is easier to develop than distributed software.

To support distributed execution of legacy software written in Java[7], we have developed a system named *Addistant*. Addistant helps developers modify legacy Java programs to run on multiple Java virtual machines (JVM)[14]. It performs:

- Letting developers specify where to allocate the instances of each class among multiple hosts, in a policy file separated from the original program. All the instances of a class must be subject to the same allocation policy. Since real software contains a large number of objects, it is not realistic to individually specify where each object is allocated.
- Translating the bytecode of the legacy Java software according to the specification above so that specified classes are executed on the JVM running on a remote host. Addistant does not need source code for the translation. The translated bytecode is the regular Java bytecode. No custom JVM is needed for execution. And,
- Delivering the translated bytecode to remote JVM. This delivery is also performed by the runtime system of Addistant.

The translation by Addistant has been implemented by a synthesis and re-engineering of ideas found in existing programming tools and environments for distributed software. It is based on the proxy-master model, in which a proxy object forwards method invocations to a remote object through a network although the generation of the classes for proxy objects automatically managed by Addistant; it is hidden from the developers. A technical contribution by Addistant is rather that it covers all the issues that we encounter if applying the proxy-master model to real software development in Java. For example, since the JVM does not allow modifying the bytecode of the system classes at load time, the proxy-master model cannot be implemented with only a well-known straightforward translation, which requires the bytecode translation of all related classes including the system classes. To avoid these problems, Addistant provides multiple implementation approaches, which developers can choose for each class.

A typical application of Addistant is to apply functional distribution to a legacy Java program so that some modules of that program are executed on a remote host suitable for the functionality of those modules. For example, Addistant can be used to adapt a legacy program using the Swing class library[22], which is Java’s graphical user interface (GUI) library, so that GUI objects are executed on a host in front of the user while other objects are on a remote high-performance host. The resulting program produced by Addistant achieves good performance. Although the same effects can be achieved by using the X Window System[18], which enables the program to show windows on a remote display, our experiments showed that Addistant could achieve better response time of the GUI than the X Window System. This is because the X Window

System implements distribution at the level of runtime library and thus it needs network communication for every drawing primitive. On the other hand, Addistant implements distribution by translating a whole program including both library code and user code. This higher-level distribution significantly reduces the amount of network communication. This fact suggests that a distributed program developed with a program translator can give better performance than one with a runtime library.

In the rest of this paper, Section 2 presents the architecture of Addistant. Section 3 describes Java-related implementation issues. In Section 4, we show how Addistant can be used for adapting legacy software using the Swing class library to distributed execution. Section 5 discusses related work. Finally, section 6 concludes the paper.

2 Addistant

Addistant is a Java programming tool for adapting legacy software, which was developed with intent to be executed on a single JVM, to distributed execution so that some objects of that software are executed on a remote host. This adaptation is performed by a bytecode translator at load time. This section first mentions design issues of the tools like Addistant, and then presents how Addistant deals with them.

2.1 Design Goal

Unlike developing distributed software from scratch, adapting legacy software written in Java to distributed execution needs special tool support. Without such tool support, programmers would have to read the program of that software and modify it so that some objects should be allocated on a remote host and method invocations be specially treated as if they are across a network. Since manual modification is troublesome and error-prone, a programming tool should automate this modification.

Although a number of researchers have been proposing Java-based distributed languages[12,17,19], those languages are not suitable for this purpose. Using such a distributed language, the programmer needs to obtain the source code of the program, which is usually unavailable if supplied by a third party. Moreover, she has to modify the program to use special syntax provided by that language. For example, in case of a language proposed by Nagaratnam[17], a regular Java statement for creating an object:

```
Frame f = new Frame("The Great Encyclopedia");
```

must be replaced with a statement:

```
Frame f = remotenew Frame("The Great Encyclopedia");
```

using special syntax `remotenuw`. She has to obtain the source code and edit all such statements.

Existing object request brokers (ORB)[4,8,9,10,21] are not suitable as well. They are mainly for making legacy software as a component of larger distributed software. To use such an ORB for distributing some modules of that software to a remote host, a programmer has to manually split the software into several modules and modify the program so that interactions among the modules are subject to the protocols of the ORB. For example, the Java RMI requires that all remote method invocations be performed through interface types. Suppose a method `show()` is called on a remote instance `f` of a class `Frame`. First, the programmer must declare a new interface `DistributedFrame` and modify the declaration of the class `Frame` so that the class `Frame` implements the interface `DistributedFrame`. Then she has to substitute `DistributedFrame` for occurrences of the class name `Frame` in the program. Also, she has to care about a number of issues such as remote object creation and polymorphism.

An ideal tool for adapting legacy software to distributed execution must provide the following features:

- **Remote reference:** The tool must hide implementation details of remote-object references from the programmers. The programmers should not have to modify the program so that remote references in the program follow a particular protocol specified by the tool.
- **Policy of object allocation:** The tool must allow the programmers to easily specify whether each object is allocated on a local host or a remote host. Since the programmers may not know details of the program of legacy software, the object allocation should be specified at an appropriate abstract level.
- **Program delivery:** The tool must be able to automatically deliver the program of modules to a remote host if those modules are executed on that host.

In the rest of this section, we first focus on the implementation of the remote reference. Then we describe how the users of Addistant specify the policy of object allocation. We also describe how Addistant implements the program delivery.

2.2 Remote Reference

Addistant implements remote references by bytecode translation at load time. To run the translated software, no custom JVM is needed; Addistant only needs that the regular JVM is running on every host.

Addistant employs the proxy-master model, which is also known as the Remote Proxy pattern[6,20], so that a remote method can be transparently invoked with the same syntax as a local method. In this model, an object whose methods can be invoked from a remote host is associated with an object called proxy existing on that remote host. For distinction, we call the former object master.

A proxy provides the same set of methods as its master and delegates every method invocation to its master. It encapsulates details of network communication necessary for the remote method invocations.

Unfortunately, any single implementation approach of the proxy-master model cannot deal with all kinds of classes. Each approach covers only the classes satisfying the criteria peculiar to that approach. Since we design a programming tool for legacy software, which someone else may have written, we cannot choose a single approach and enforce the criteria on the whole program. For example, one of the approaches needs to modify the declaration of the class of master objects. Since the JVM does not accept modified system classes, if an instance of a system class is a remote object, that approach cannot be used. A different approach must be used for that case.

To avoid this problem, Addistant provides several different approaches for implementing the proxy-master model. It currently provides four approaches: *replace*, *rename*, *subclass*, and *copy*. The developers can choose one from the four for each class of master. The differences among the four approaches are mainly how a proxy class is declared, how caller-side code, that is, expressions of remote method invocations, is modified, and how a master class is modified. The four approaches cover most of cases in practical development according to our experiences with the Swing library. To choose one from these four approaches, the developers must know whether a given class of master meets some of the following features or not:

- **Call by reference:** The master object must be passed to a remote method as a parameter in the call-by-reference manner. It cannot be passed in the call-by-value manner.
- **Heterogeneity:** A variable with that class type must be able to hold both local and remote references. Some kinds of master objects do not require this feature. For example, all the instances of a GUI class would exist on the same host in front of the user. If so, all the references to those instances are local on that host while they are remote on the other host. In this case, local and remote references do not coexist on a single host.
- **Unmodifiable bytecode:** The implementation must be done without transforming the unmodifiable bytecode. This is required as JVM prohibits the developers from modifying or replacing the bytecode of the system classes such as `java.util.Vector`. This feature is divided into three sub-features: the class declaration of the master objects (*original class*) is unmodifiable, other master classes accessing the master objects (*referrer classes*) are, or other master classes creating the master objects (*factory classes*) are, respectively.

The remainder of this subsection presents details of the four approaches, and conditions in which the approaches can be used. The summary of the conditions is listed in Table 1.

Replace Approach. The first approach is the *replace* approach. It is available unless the heterogeneity feature is required or the original class is unmodifiable.

Table 1. Applicability of the four approaches. The mark of x ([x]) indicates that the approach is (probably) unavailable in case the feature is required.

	Replace	Rename	Subclass	Copy
Call by reference				x
Heterogeneity	x	x		
Unmodifiable bytecode of original class	x		[x]	[x]
referrer classes		x		
factory classes		x	x	

Developers should apply this approach to non-system classes whose masters are allocated at the same host.

Suppose that the class of a master object is `Widget`. Since the heterogeneity feature is not required, all the references to the `Widget` objects are either local or remote. Therefore, Addistant uses the original `Widget` class for local references on one host while it generates another version of the `Widget` class for remote references on the other hosts (Table 2). This version corresponds to a proxy class for the original `Widget` class. Addistant sends the bytecode of this proxy class to the host where there are remote references to `Widget` masters.

Table 2. The proxy class for a class `Widget` used by each approach. Here, we assume that the original `Widget` is a subclass of `Object`.

	Replace	Rename	Subclass
Proxy class	<code>Widget†</code>	<code>WidgetProxy</code>	<code>WidgetProxy</code>
Superclass of proxy	<code>Object</code>	<code>Object</code>	<code>Widget</code>
Variable type for proxy	<code>Widget†</code>	<code>WidgetProxy</code>	<code>Widget</code>

†A different version of the `Widget` class.

Rename Approach. The second approach is the *rename* approach. The replace approach is not available if the declaration of the original class is not modifiable. The rename approach can be used in that case although it requires that the referrer classes and the factory classes are modifiable. As the replace approach, the rename approach is not available if the heterogeneity feature is required. Developers should apply this approach to classes like `java.awt.Window`.

In the rename approach, Addistant generates a proxy class of the original class `Widget` with a different name such as `WidgetProxy`. Then Addistant uses that proxy class for remote references. It modifies the bytecode of all the referrer classes on the hosts where references to the `Widget` objects are remote so that all the occurrences of the original class name `Widget` are replaced with the proxy class name `WidgetProxy`. Addistant does not modify the other referrer classes on the host where references to the `Widget` objects are local.

Addistant also modifies the factory classes if they are used on the host where references to the `Widget` objects are remote. Since the `Widget` objects must be

created on the other host, Addistant also replaces all the occurrences of `Widget` with `WidgetProxy` in the bytecode of the factory classes. For example, it translates the following statement:

```
Frame w = new Frame();
```

into this statement:

```
FrameProxy w = new FrameProxy();
```

The latter statement creates a proxy object, which requests a remote host to create a master object.

Subclass Approach. The third approach is the *subclass* approach. It is available even if the heterogeneity feature is required. Developers should apply this approach to classes like `java.util.Vector`.

In this approach, a proxy class `WidgetProxy` is a subclass of the original class `Widget`. Both local and remote references have the reference type to `Widget` so that they can coexist on the same host. If a reference is local, it points to a `Widget` object. If a reference is remote, it points to a `WidgetProxy` object.

However, as the rename approach, this approach needs to modify the factory classes if they are used on the host on which references to master objects are remote. Furthermore, this approach may require that the original class is modifiable. First, if the original class is a `final` class or it includes a `final` method, it must be modified to be a non-`final` class and to include no `final` method. Otherwise, the proxy class cannot be a subclass of the original class or override methods declared in the original class. Second, if the constructor of the original class causes inappropriate side-effects and fails to create an object, Addistant must add to that class another constructor performing nothing so that the constructor of the proxy class can call it. Remember that a constructor must call a constructor of the super class in Java. For example, the original constructor of the class `Widget` may access a local graphic device. If it is called by the constructor of `WidgetProxy` (since `WidgetProxy` is a subclass of `Widget`), it may throw an error because of the absence of the graphic device on the host where the `WidgetProxy` object is created.

Note that the subclass approach does not require that the original class is modifiable if the original class is not a `final` class and the constructors do not cause inappropriate side-effects. For instance, the bytecode of a system class `java.io.File` is unmodifiable. Since that class is used by other system classes, the referrer classes are also unmodifiable. Thus, even if the heterogeneity feature is not required, either the replace or rename approaches cannot be used for `java.io.File`. On the other hand, the subclass approach can be used for that class.

Copy Approach. The last is the *copy* approach. This approach can be used for primitive types such as `int` and classes like `java.lang.String`, instances of which are immutable.

If the copy approach is chosen for a class *C*, a remote reference to an instance of *C* cannot exist. If a local reference to an instance of *C* is passed to a remote method, Addistant makes a *shallow* copy of that instance on the remote host. A local reference to that copy is passed to the method. Thus, the copy approach cannot be used if a reference must be passed to a remote method in the call-by-reference manner. However, the copy approach does not need to modify bytecode at all.

Addistant also provides a slightly different version of the copy approach: the *write-back* copy approach. If this approach is chosen, the contents of the copy passed to the remote method are written back to the master object after executing that remote method. For example, suppose that the write-back copy approach is chosen for an array of `byte`. Then in the following code:

```
byte[] buf = ... ;
inputstream.read(buf);
```

the call to `read()` on a remote object `inputstream` makes a copy of `buf` on the remote host. A local reference to that copy is passed to `read()`. Since the write-back copy approach is chosen, the contents of that copy are written back to `buf` after executing `read()`. Therefore, the byte data read from the input stream are eventually stored in `buf`.

2.3 Object Allocation

Addistant allows the developers to specify a policy of object allocation for each class. It does not allow to use a different policy for each object because Addistant is a tool for modifying legacy software; it is not realistic for the developers to specify a policy for every occurrence of “`new`” (the operator of object creation) appearing in a program, which someone else may have written.

The developers can declare that all the instances of a class are allocated on a specific host. If a host *D* is specified for a class *C*, an expression “`new C()`” (create an instance of *C*) executed on any host is interpreted as that an instance of *C* is created on the (probably remote) host *D*. On the other hand, if any host is not specified for the class *C*, an expression “`new C()`” executed on a host *D'* is interpreted as that an instance of *C* is locally created on the host *D'*.

The declaration by the developers is written in a policy file, which Addistant reads at startup time. The policy file is written in an XML-like syntax. For example, a declaration below:

```
<import proxy="rename" from="display">
  java.awt.*
</import>
```

means that all the instances of classes included in the `java.awt` package are allocated on a host specified by a variable `display`. Remote references to these instances are implemented with the rename approach. The variable `display` is

bound to a real host name at run time. If the “**from**” attribute is not given, the instances of a class *C* are allocated on a host where an expression “**new C()**” is executed.

Note that `java.awt.*` means all the classes included in the `java.awt` package. It does not mean sub-packages of `java.awt`, such as `java.awt.image` because sub-packages are irrelevant to the parent package with respect to the language semantics. For example, the access rights of a class in a sub-package are equivalent to ones in other packages than the parent package of that sub-package. To specify all the classes and sub-packages in `java.awt`, `java.awt.-` should be used.

Besides all classes included in a package, all subclasses of a class in a package can be specified. For example, a declaration below:

```
<import proxy="rename" from="display">
    subclass@java.awt.Component
</import>
```

means that the rename approach is used for all the subclasses of the class `Component`, including `Component` itself. To specify only the subclasses excluding the parent class, `exactsubclass` should be used instead of `subclass`.

Some implementation approaches of remote references restrict policies of object allocation. Since the replace and rename approaches require that local and remote references do not coexist, the “**from**” field must be specified so that instances are created on the same host. On the other hand, the copy approach does not allow the developers to specify the “**from**” field since it does not deal with remote references.

2.4 Bytecode Delivery

Addistant provides a mechanism for automatically distributing bytecode from a host to other hosts. The users have to only run a class loader of Addistant on every host. If a program starts on a host *A* and creates an object on a remote host *D*, the class loader on the host *A* sends necessary bytecode to the class loader on the host *D* so that the object can be created on the host *D*. If the bytecode must be modified, it is modified by the class loader on the host *A* before it is sent to the host *D*. If the bytecode is of system classes, the class loader on the host *D* loads it from a local file system instead of the host *A*.

Although the regular class loader of Java fetches bytecode on demand, the class loader of Addistant may fetch the bytecode of certain classes in advance. For example, suppose that the rename approach is specified for all subclasses of a class *C*. If the class loader of Addistant loads a class *U*, it must read the bytecode of the class specified by every name appearing in the bytecode of *U* and examine whether each class is a subclass of *C*. If so, the class name must be replaced with the name of the proxy class. Thus, while the class loader of Addistant loads a class *U*, it may fetch a number of other classes as well as the class *C* and subclasses of *C*.

3 Implementation Issues

3.1 Single System Image

There are several implementation issues for keeping the semantics of the Java language in distributed program execution, that is, providing a single system image with multiple JVMs. This sub section describes how Addistant deals with those issues.

Remote Field Access. Although a naive implementation of the proxy-master model cannot support remote field accesses, Addistant translates a field access at the bytecode level into a static method invocation on that class and thereby enables remote field accesses. Suppose that a class `Point` declares a field `x` and the field is accessed as follows:

```
Point p;
.. = p.x ..
.. p.x = 100 ..
```

If remote references to `Point` objects are implemented by the rename approach, the code above is translated into the code below:

```
PointProxy p;
.. = PointProxy.read_x(p) ..
.. PointProxy.write_x(p, 100) ..
```

The static methods `read_x()` and `write_y()` implement the remote field accesses. They are declared in the proxy class produced by Addistant.

The translation above must be applied to all the remote field accesses. Therefore, Addistant cannot deal with remote field accesses embedded in the unmodifiable bytecode, for example, the bytecode of the system classes.

Equality between Remote References. Addistant preserves the semantics of the equality operators such as “==” and “!=” with respect to remote references. To do that, Addistant maintains a table of proxy objects on every host so that there exists only a single proxy object referencing to each master object. Addistant gives a unique identifier to every master object and sends this identifier when a reference to the master object is passed as a parameter across the network to a remote method. Then it looks up the corresponding proxy object in the table and passes a reference to that proxy object to the destination method. If the proxy object is not found in the table, Addistant creates and registers it in the table.

Self Deadlock Avoidance. In Addistant, any host can invoke a method on a remote object and receive a method invocation from a remote object. Therefore, a remote method call from a host A to a host D may cause another method call back from D to A . In this case, the latter method call must be handled by the same thread that requested the former method call on the host A . Otherwise, a deadlock may occur if the methods are synchronized ones.

Suppose that a `Button` object and a `Listener` object exist on different hosts D (display host) and A (application host), respectively. The declarations of class `Button` and `Listener` are as follows:

```
class Button {
    Listener listener;
    synchronized void push() {
        listener.pushed(this);
    }
    synchronized ButtonState getState() { ... }
}

class Listener {
    void handlePush(Button button) {
        .. button.getState() ..
    }
}
```

If `push()` is invoked on the `Button` object, it calls `handlePush()` on the remote `Listener` object. Then `getState()` is called back on the `Button` object. If `push()` and `getState()` are executed by different threads, a deadlock occurs since the two threads try to lock the `Button` object at the same time. The deadlock never occurs if the two objects exist on the same host because all the methods are executed by the same thread.

In order to ensure the same thread executes all the methods called back, Addistant establishes a one-to-one communication channel between the thread executing `push()` on D (T_D^i) and the one executing `handlePush()` on A (T_A^i). This communication channel is stored in a thread local variable implemented with `java.lang.ThreadLocal`. A thread always uses the same channel for every remote method invocation and it waits for not only the result of the invocation but also another request of invocation from a remote thread sharing the same channel. When `handlePush()` calls `getState()`, the thread T_A^i sends a request of `getState()` to the remote thread T_D^i connected through the communication channel, which is the thread executing `push()` on D and blocking to wait for the result of `handlePush()`. The thread T_D^i invokes the requested `getState()` to send the result of `getState()` through the channel, and then it continues to wait for the result of the original `handlePush()`. Thereby, both `push()` and `getState()` are executed by the same thread T_D^i . A deadlock is avoided.

Distributed Garbage Collection. Addistant maintains a table of objects exported to a remote host. While there exists a proxy object on a remote host,

the master object is recorded in that table so that it is not garbage collected. If all the proxy objects are garbage collected, then the master object is removed from the table. If there are no other references to the master object, then the master object is garbage collected.

The table of proxy objects for checking the equality between remote references is implemented with the weak reference mechanism of Java[1]. An element of the table is a weak reference to a proxy object. Thus, the proxy object is garbage collected when the garbage collector determines that nothing except that table refers to the proxy object.

Currently, Addistant cannot collect all objects if remote references make cycles. Although several algorithms are known for dealing with distributed cycles, efficiently implementing those algorithms is not straightforward without modifying the JVM. For example, if using a distributed mark-sweep algorithm, we would need a mechanism for tracing object references. However, Java’s reflection API does not provide such a mechanism. We expect that weak references and object finalizers might help to solve this problem but implementation details are still open.

3.2 Bytecode Modification

Bytecode Translation Toolkit. One of the research aims of the development of Addistant was to examine the expressive power of Javassist[5], which is our toolkit for implementing a bytecode translator for Java. Unlike other similar toolkits, Javassist provides a source-level view of bytecode for the developers, who can manipulate bytecode without detailed knowledge of the bytecode specifications. Javassist is easier to use than other naive toolkits as a source-level debugger is easier to use than an assembly-level debugger. On the other hand, Javassist restricts the ability to modify bytecode. It does not allow bytecode modification that is difficult to express with a source-level view.

To show that the expressive power of Javassist is powerful enough to implement a real application, we have developed Addistant within the confines of the Javassist API (Application Programming Interface). No undocumented low-level API was used. All the bytecode modification that Addistant needs could be easily implemented with a source-level abstraction provided by Javassist.

Bootstrap Classes. If we use a command-line option provided by Sun’s JVM, we can modify the bytecode of system classes and have the JVM load the modified bytecode at bootstrap time. Hence using this option extends the range of the classes that the approaches provided by Addistant for implementing the proxy-master model are applicable to. However, we did not modify the bytecode of the system classes because Sun’s license terms prohibit the modification. Even if we could modify, consistently modifying the system classes is difficult since runtime systems such as a system class loader depends on the definition of the system classes.

4 Distributed Swing Applications

This section presents that Addistant can adapt legacy software using the Swing class library so that GUI objects are allocated on a remote host and the users can interact with the software through the GUI shown on a remote display. The Swing class library is a GUI library included in the standard Java runtime environment. Although the same effects can be achieved with the X Window system[18], Addistant can achieve better performance since drawing operations are directly performed on the host with a display. This is typical benefit of functional distribution. The X Window system needs network communication for every primitive drawing operation and hence communication overheads tend to be a performance bottleneck.

In this section, we first present a policy file for adapting legacy software using the Swing class library to distributed execution. Then we show the results of our performance measurement.

4.1 Policy File

The following is a typical policy file for adapting software using the Swing class library:

```
<policy>
  <import proxy="rename" from="display">
    subclass@java.awt.-
    subclass@javax.swing.-
    subclass@javax.accessibility.*
    subclass@java.util.EventObject          </import>
  <import proxy="rename" from="application">
    exactsubclass@java.io.[InputStream|OutputStream|Reader|Writer]
    exactsubclass@javax.swing.filechooser.* </import>
  <import proxy="subclass">
    subclass@java.util.[Dictionary|AbstractCollection|AbstractMap]
    subclass@java.util.BitSet              </import>
  <import proxy="writeBackCopy">
    array@-                                </import>
  <import proxy="replace" from="application">
    user@-                                  </import>
  <import proxy="copy">
    -                                        </import>
</policy>
```

Here, the variable `display` indicates the host where the GUI objects are allocated. The variable `application` indicates the other host where the rest of the objects are allocated. An `import` declaration listed above has a higher priority.

This policy file specifies that GUI objects are allocated on the `display` host and remote references to those objects are implemented with the rename approach. Any array type (`array@-`) is processed with the write-back copy approach. The instances of classes except the system classes (`user@-`) are allocated on the `application` host and remote references to them are implemented with the replace approach. The rest of the classes are processed with the copy approach.

4.2 Performance Measurement

For performance measurement, we used two host computers. One is a machine with a 500MHz PentiumIII and Linux 2.2. It is a display server for executing GUI objects. The other is a machine with a 440MHz UltraSparcII and Solaris 2.7. It is an application server for executing the other objects. We used the HotSpot JVM (JDK 1.3) for both machines. For connecting the two machines, we used two kinds of network: 100Base-TX full-duplex and 10Base-T half-duplex.

Remote Method Invocation. Before measuring the performance of a GUI, we compared the execution time of remote invocations of empty methods among Addistant (AD) and other Java-based object request brokers (ORB), which are HORB[10] version 2.0.1, Java RMI (JRMI) included in JDK 1.3, and Java Class Broker[8] (JCB) version 1.2. We changed the number and types of parameters and measured the elapsed time of each remote method invocation. We also changed the network connecting the two hosts.

Table 3. Elapsed Time (milliseconds) for a remote method invocation. AD indicates Addistant.

(ms)	(100Base-TX full-duplex)				(10Base-T half-duplex)			
	HORB	JRMI	JCB	AD	HORB	JRMI	JCB	AD
void f()	0.33	0.52	0.71	0.28	0.48	0.69	0.86	0.40
void f(int)	0.33	0.53	0.78	0.48	0.48	0.69	0.93	0.61
int f(int)	0.34	0.54	1.20	0.54	0.49	0.71	1.42	0.68
void f(int,int,int)	0.34	0.54	0.75	0.76	0.49	0.71	0.91	0.91
int f(int,int,int)	0.34	0.55	1.17	0.83	0.50	0.72	1.40	0.99
void f(String)	0.34	0.58	0.83	0.36	0.50	0.75	1.00	0.48
String f(String)	0.35	0.63	0.94	0.37	0.52	0.81	1.11	0.50
void f(String[])	0.58	0.87	1.26	0.66	0.77	1.08	1.46	0.85
String[] f(String[])	0.84	1.22	1.66	0.79	1.10	1.50	1.94	0.99
void f(byte[])	0.69	0.94	1.26	2.76	1.51	1.73	2.08	2.93

Table 3 lists the results. The results showed that Addistant achieved a comparable performance to other ORB except the case that a `byte` array was passed. This is because the parameter encoder/decoder of Addistant had not been tuned and because Addistant used the write-back copy approach for passing an array as a parameter although the other ORB did not write the updated contents of the array back after executing a method. In the measurement, all `String` type parameters included 10 ASCII characters. The size of `String` array was three. The size of `byte` array was 1024.

Window Drawing. To measure the performance of a GUI, we prepared three Java programs. The first one displays a single window (a `java.awt.Frame` object) containing no components. The second displays a single empty internal window (a `javax.swing.JInternalFrame` object) in a window (a `javax.swing.JFrame` object). The third displays a single internal window containing twenty buttons

(a `javax.swing.JButton` object) in a window. The size of the window is 600 by 600 while the size of the internal window is 500 by 500.

We compared the X Window system[18], Rawt[11], and Addistant by measuring the elapsed time that each program took for creating and drawing a window (and internal windows) on a remote display. The X Window system showed a window on a remote display by connecting a remote X server. Rawt is a GUI library that is compatible to the Swing class library but enables to show a window on a remote display. Addistant showed a window on a remote display by allocating GUI objects on the remote host with that display.

Table 4 listed the results. As a drawing image becomes more complex, Addistant showed better performance against Rawt because Rawt allocates only part of instances of the Swing classes on the remote host with a display and thus it needs a larger number of remote method invocations for drawing a window. On the other hand, Addistant allocates all instances of the Swing classes on the remote host and thus the interactions among the Swing objects are local method invocations. This is because Addistant is a general-purpose bytecode translator and it allows the developers to easily customize object allocation for maximizing performance. Rawt cannot do that since the implementation of Rawt is a black box.

Table 4. The elapsed time (seconds) for drawing a window.

	X Window	Rawt	Addistant
(100base-TX full-duplex)			
No components	0.005	0.041	0.044
1 internal window	0.156	1.814	0.276
20 buttons	0.873	17.599	0.955
(10base-T half-duplex)			
No components	0.006	0.041	0.045
1 internal window	0.612	1.988	0.281
20 buttons	1.895	21.322	0.971

Since the X Window system asynchronously executes an X server and an X client, the elapsed time listed in the table indicates the time needed for sending all the requests from the client to the server. It does not indicate the actual elapsed time of drawing a window. In fact, we observed that the response time of the GUI implemented on top of the X Window system was considerably slower than one on top of Addistant.

To confirm our observation above, we conducted another experiment. We wrote a Java program that displays a button in a window and, if that button is clicked, then a graphic image (1148 by 778) is shown in the window. Table 5 listed the results of our experiment. We measured the elapsed time after the button was clicked by mouse until the image was shown. The time was measured by hand. 0.0 means that the response time was too short to measure. Since the Swing class library caches a drawn image, Rawt and Addistant responded quicker than the X Window to a mouse click at the second time. The X Window must

transfer the drawn image every time from the client to the server. Even at the first time, Addistant achieved the best performance if the network is 10Base-T since the X Window system and Rawt had to transfer a larger amount of data between the hosts. Table 6 listed the results of our measurement of the size of the data exchanged through a network during the above interaction. The X Window system needs a few megabytes whereas Addistant does less than a hundred kilobytes. The large amount of exchanged data can be a performance bottleneck.

Table 5. The response time (seconds) to a mouse click.

(sec.)	(100Base-TX full-duplex)			(10Base-T half-duplex)		
	X Window	Rawt	Addistant	X Window	Rawt	Addistant
1st	1.6	2.6	2.0	5.6	3.2	2.0
2nd	1.4	0.0	0.0	5.6	0.0	0.0

Table 6. The size of the data (Kbyte) exchanged through a network.

	X Window	Rawt	Addistant
1st	3493.57	116.20	81.88
2nd	3438.96	10.95	0.06

5 Related Work

5.1 Transparent Distribution

To run a Java program on a distributed environment, several extended Java virtual machines have been developed. These virtual machines such as cJVM[2], Java/DSM[23], and JESSICA[16] provide a single-machine image on several network-connected computers, that is, a workstation/PC cluster. Thus, multiple threads are executed in parallel as if they were running on a multi-processor machine with shared memory. These virtual machines do not need to modify a program at all to run it. A difference between Addistant and these virtual machines is that Addistant uses the standard JVM and hence it is mainly for functional distribution, where objects run on the most suitable host for the computation by the objects.

JavaParty[19] extended the Java language for parallel distributed computing. They introduced only the extended modifier “**remote**” for class declarations. Although the users of Addistant do not have to modify a program, the users of JavaParty have to append an extended modifier “**remote**” to a class declaration if an instance of that class is accessed through a remote reference.

There are a number of object request brokers for Java. Most of them, including the Java RMI[21], require that a remote object be accessed through an interface type. Thus, developers may have to largely modify programs if they adapt legacy software to distributed execution. Java Class Broker[8] avoids this problem by a technique similar to our subclass approach. However, it requires developers to modify a program to follow another programming convention. For example, the following regular Java program:

```
Frame f = new Frame("The Great Encyclopedia");
Button b = new Button();
f.add(b);
```

must be translated into a program using a runtime distribution manager object `objectBroker`:

```
Object[] params = {"The Great Encyclopedia"};
Frame f = (Frame) objectBroker.create("Frame", params);
Button b = (Button) objectBroker.getProxy("Button", new Button());
f.add(b);
```

5.2 Remote Display

The X Window System[18] enables a Java program to show a graphical output on the display of a remote host. Like Addistant, the X Window System does not require developers to modify their programs to use a remote display. However, as shown in Section 4, the X Window System is often less efficient than Addistant.

Rawt[11] is a GUI library that is compatible to the standard Java GUI library. If substituting Rawt for the standard library, developers can extend their programs without any other modifications to use a remote display for output. Underlying network communication is encapsulated by that library. Addistant can be regarded as a tool for semi-automatically producing a library like Rawt from the standard Java GUI library. Since the production by Addistant is based on both the library and user code, however, the resulting software can often achieve better performance than Rawt.

5.3 Aspect-Oriented Programming

With Addistant, developers describe a policy file for adapting software to distributed execution. This policy file can be considered as a separate description of a distribution aspect in the context of aspect-oriented programming (AOP). In this context, Addistant is a tool for *weaving* a Java program written for a single JVM and a description separately written about a distribution aspect.

Proposing a distribution aspect is not new. For example, D[15] provides an aspect language for distribution. However, it allows programmers to separately describe how a parameter is passed to a remote procedure whereas Addistant allows to describe where objects are allocated and how proxy objects are implemented. Furthermore, it seems that the design goal of D is to support the

development of distributed software from scratch. The goal of Addistant is to add a new aspect on existing software for adaptation. Thus, the description in a policy file is not a part of program text but rather *meta-level* instructions to modify an existing program.

6 Conclusion

This paper presented Addistant, which is a programming tool for adapting legacy Java software to distributed execution. Addistant performs this adaptation by bytecode translation at load time. No source code is needed for the adaptation. The users of Addistant have only to write a policy file for specifying where the instances of each class are allocated and how remote references to those instances are implemented. The users can select an implementation approach from the four provided by Addistant.

Although the four implementation approaches are not new, a contribution of this paper is that it reveals that letting developers select an implementation approach for each class is necessary for adapting legacy Java software in the real world to distributed execution. This paper presented several practical issues that we must consider for the adaptation. However, the ability of Addistant still has a few limitations. Although the developers using Addistant do not need to read or modify source code, they must have some knowledge of source code, for example, which class of objects should be allocated on a remote host. Moreover, Addistant provides only class-based distribution: all the instances of a class must be allocated on the same host. These limitations are acceptable in our GUI examples although it is an open question in other contexts.

This paper also showed that Addistant could adapt a Java program using the Swing class library so that GUI objects could be allocated on a remote host with a display. This functional distribution with Addistant showed better response time of the GUI than the distribution with the X Window System and the Rawt class library. This fact suggests that library-level functional distribution could not give good performance since only the library code is split and distributed to multiple hosts. On the other hand, Addistant can split a whole program including both user and library code and then it can distribute objects so that the maximum performance could be obtained.

Acknowledgement

The authors thank anonymous reviewers for their advice on this paper. Kenichi Kourai and Yasushi Shinjo gave helpful comments on the drafts of this paper. This research was supported in part by the Japan Science and Technology Corporation.

References

1. Ken Arnold, James Gosling and David Holmes, Garbage Collection and Memory, Chapter 12, In *The Java Programming Language Third Edition*, Addison Wesley, pp.313–327, 2000.
2. Yariv Aridor, Michael Factor and Avi Teperman, CJVM: a Single System Image of a JVM on a Cluster, In *Proceedings of ICPP 99*, IEEE, 1999.
3. Andrew Black, Norman Hutchinson, Eric Jul, Henry Levy, and Larry Carter, Distribution and Abstract Types in Emerald, In *IEEE Transactions on Software Engineering*, Vol. SE-13, No. 1, IEEE, pp.65–76, 1987.
4. Denis Caromel, Wilfried Klauser and Julien Vayssi era, Towards Seamless Computing and Metacomputing in Java, In *Concurrency: Practice & Experience*, Vol. 10, No. 11-13, Wiley, pp.1043–1061, 1998.
5. Shigeru Chiba, Load-time Structural Reflection in Java, In *Proceedings of ECOOP 2000, LNCS 1850*, Springer Verlag, pp.313–336, 2000.
6. Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, *Design Patterns - Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1994.
7. James Gosling, Bill Joy, and Guy Steele, *The Java Language Specification*, Addison-Wesley, 1996.
8. Zvi Har’El and Zvi Rosberg, Java Class Broker - A Seamless Bridge from Local to Distributed Programming, In *Journal of Parallel and Distributed Computing*, Vol. 60, No. 11, Academic Press, pp.1223–1237, 2000.
9. Michael Hicks, Suresh Jagannathan, Richard Kelsey, Jonathan T. Moore, and Cristian Ungureanu, Transparent Communication for Distributed Objects in Java, In *Proceedings of the ACM 1999 conference on Java Grande*, pp.160–170, 1999.
10. Satoshi Hirano, HORB: Distributed Execution of Java Programs, In *Lecture Notes in Computer Science 1274*, Springer, pp.29–42, 1997.
11. IBM, *Remote Abstract Windowing Toolkit (RAWT)*, Online publishing, URI <http://www.s390.ibm.com/java/rawt.html>, 2000.
12. Laxmikant V. Kal e, Milind Bhandarkar, and Terry Wilmarth, Design and Implementation of Parallel Java with Global Object Space, In *Proceedings of PDPTA ’97, Conference on Parallel and Distributed Processing Technology and Applications*, 1997.
13. Sheng Liang and Gilad Bracha, Dynamic Class Loading in the Java Virtual Machine, In *Proceedings of OOPSLA ’98, ACM SIGPLAN Notices*, Vol. 33, No. 10, pp.36–44, 1998.
14. Tim Lindholm and Frank Yellin, *The Java Virtual Machine Specification*, Addison-Wesley, 1997.
15. Cristina V. Lopes and Gregor Kiczales, D: A Language Framework for Distributed Programming, In *Technical report SPL97-010*, pp.50–67, Xerox Palo Alto Research Center, 1997.
16. Matchy J. M. Ma, Cho-Li Wang, and Francis C. M. Lau, JESSICA: Java-Enabled Single-System-Image Computing Architecture, In *Journal of Parallel and Distributed Computing*, Vol. 60, No. 11, Academic Press, pp.1194–1222, 2000.
17. Nataraj Nagaratnam, Arvind Srinivasan, and Doug Lea, Remote Objects in Java, In *Proceedings of IASTED ’96, International Conference on Networks*, 1996.
18. Robert Scheifler and Jim Gettys, The X Window System, In *ACM Transactions on Graphics*, Vol. 5, No. 2, pp.79–109, 1986.
19. Michael Philippsen and Matthias Zenger, JavaParty - Transparent Remote Objects in Java, In *Concurrency: Practice & Experience*, Vol. 9, No. 11, Wiley, pp.1225–1242, 1999.

20. Hans Rohnert, The Proxy Design Pattern Revisited, In *Pattern Languages of Program Design 2*, Addison-Wesley, pp.105–118, 1995.
21. Sun Microsystems, Inc., *The Java Remote Method Invocation Specification*, Online publishing, URI <http://java.sun.com/products/jdk/rmi/>, 1997.
22. Sun Microsystems, Inc., *Java Foundation Classes*, Online publishing, URI <http://java.sun.com/products/jfc/index.html>, 2000.
23. Weimin Yu and Alan Cox, Java/DSM: A Platform for Heterogeneous Computing, In *Concurrency: Practice & Experience, Vol.9, No.11*, Wiley, pp.1213–1224, 1997.