JR: Flexible Distributed Programming in an Extended Java

Aaron W. Keen, Tingjian Ge, Justin T. Maris, and Ronald A. Olsson
Department of Computer Science
University of California, Davis
{keen,maris,olsson}@cs.ucdavis.edu, ge@informix.com

Abstract

Java provides a clean object-oriented programming model and allows for inherently system-independent programs. Unfortunately, Java has a limited concurrency model, providing only threads and remote method invocation (RMI).

The JR programming language extends Java to provide a rich concurrency model. JR provides dynamic remote virtual machine creation, dynamic remote object creation, remote method invocation, asynchronous communication, rendezvous, and dynamic process creation. JR programs are written in an extended Java and then translated into standard Java programs. The JR run-time support system is also written in standard Java.

This paper describes the JR programming language and its implementation. Some initial measurements of the performance of the implementation are also included.

1 Introduction

Java [24] has proven to be a clean and simple language for object-oriented programming. Even so, the standard Java concurrency model is rather limited. It provides threads, a primitive monitor-like mechanism, and remote method invocation (RMI). Though these features are useful, they offer little flexibility in the design and implementation of concurrent programs.

Our work provides a richer and more flexible concurrent programming model for Java. Our approach is to extend Java with the concurrency model provided by the SR concurrent programming language [5, 4]. The result is a new language, a superset of Java, which we call JR. JR adapts the following features from SR: dynamic remote virtual machine creation, dynamic remote object creation, remote method invocation, dynamic process creation, support for rendezvous, and asynchronous message passing. JR takes a novel object-oriented approach to synchronization.

The contributions of this work are the overall design and implementation of JR and how JR resolves the tension between inheritance and concurrency. JR has been designed to integrate the SR concurrency model with Java in a manner that retains the "feel" of Java. We expect that JR will be useful as a research and teaching tool. The results of this research will also be beneficial to the future design and implementation of other concurrent programming languages. In particular, we expect that JR's object-oriented approach to synchronization should be useful.

This research was supported in part by the National Science Foundation grant CCR-9527295.

The rest of this paper is organized as follows. Section 2 provides motivation and relevant background. Section 3 presents an overview of the JR programming language. Section 4 discusses inheritance in JR. Section 5 discusses the implementation of our initial prototype of JR and Section 6 evaluates its performance. Section 7 discusses related work. Finally, Section 8 covers future work on JR and concludes this paper.

2 Motivation and Background

Often described as simple and elegant, the Java programming language [24] has quickly gained in popularity due, in part, to its object-oriented programming model and system-independent nature. It is also a (fairly) secure language with respect to its strong type checking and memory management. Unfortunately, as noted above, Java's concurrency model is not very flexible.

In this section, we describe in detail shortcomings in Java's concurrency model. We then present the SR programming language, which has a rich concurrency model, but lacks many of the features that have made Java so popular. SR is not object-oriented*, is not as secure, and is not as portable as Java. Our research provides a richer and more flexible concurrent programming model for Java. Our approach (Section 3) extends Java with the concurrency model provided by SR using a novel object-oriented approach.

Our overall approach extends the Java language, rather than augmenting Java with library calls for synchronization. This approach has potential advantages because synchronization is represented as actual language primitives [10, 41]. This higher-level approach can reduce development time for applications, increase portability, lead to better optimizations, and simplify reasoning about programs (verification).

2.1 Shortcomings in Java's Concurrency Model

Standard Java/RMI's concurrency model has two significant shortcomings:

- It provides remote objects with limited support for dynamic creation.
- It provides only a single synchronization primitive — remote method invocation — for distributed programming.

Standard Java/RMI's static virtual machine (VM) model allows a program to communicate with any number of remote objects. Each of these remote objects must be initialized prior to communication. These objects are typically initialized through some external means (i.e., manual execution of a server program) often requiring a setup procedure before the program proper begins. This static model means that programs cannot adapt themselves to their input. For example, one might want to run a distributed Java RMI program for computing Fourier coefficients on a variable number of processors. Unfortunately, that cannot be specified *easily or automatically* within a Java RMI program. (Although it is possible to write such a program in Java RMI, it requires manual interaction.)

Standard Java/RMI's only synchronization primitive for distributed programming is the remote method invocation. The limitations of languages that provide only one form of synchronization

*SR is *object-based*: it does have dynamic modules (resources) accessed via pointers (capabilities), but it lacks inheritance and virtual methods.

mechanism have been widely discussed in the literature [39, 3, 31, 21, 23]. As one example, having both synchronous and asynchronous message passing is desirable. Synchronous message passing (such as RMI) is very useful, especially for programming client/server interactions in a familiar style (e.g., procedure call syntax and semantics). However, asynchronous message passing is also useful. First, it can be used to avoid *remote delay* in which a server, in processing a request, invokes an operation in another server that might delay [31]. Asynchronous message passing can be used to invoke the remote operation whenever it is necessary to prevent the first server from delaying. In a language that provides only synchronous message passing, extra processes must be employed to avoid remote delay; this often complicates problem solutions. Asynchronous message passing is also useful whenever it is not necessary to delay the invoker of an operation. For example, it can be used to program pipelines of filter processes, where it is most natural for the producer to continue after sending a message to the consumer.

Several languages that incorporate multiple synchronization primitives have been designed and implemented, e.g., Concurrent C [22], Lynx [40, 41], StarMod [16], and SR [5, 4]. Some work [7] has shown that an implementation of such a language (in [7], SR) can provide several synchronization mechanisms at a reasonable cost. Having multiple synchronization primitives proved useful in programming different upcall program structures [6]. Additional work [36] has shown that having asynchronous message passing is desirable — for simpler *and* faster code — even if a language provides a rendezvous mechanism.

Note that Java does provide a primitive monitor-like mechanism in addition to RMI (although, some contend that Java does not really support monitors [12]). Monitor methods can be invoked remotely via RMI. However, this use of monitors would support (directly) only centralized servers not other paradigms useful in distributed programming, such as replicated workers, bag of tasks, probe/echo, broadcast, etc.[2]

2.2 The SR Concurrent Programming Language and its Concurrency Model

The SR concurrent programming language [5, 4] provides a variety of mechanisms for writing parallel and distributed programs. The notions of virtual machines, resources, and operations are central to SR's concurrency model. The notion of a *virtual machine* (VM) is used for distributing a program onto physical machines. Each VM resides on one physical machine. VMs are created dynamically. SR's primary modular component is the resource. Instances of resources are dynamically created. Processes execute within a particular resource instance and have (shared) access to variables and operations within that instance. An operation can be considered a generalization of a procedure and enable processes to interact. Given that VMs, resources, and operations are created dynamically, SR uses *capabilities* for each. A capability acts as a pointer and can be assigned to variables and passed as parameters, thus permitting, for example, dynamic communication paths.

An operation (like a procedure) has a name, can take parameters, and can return a result. It can be invoked in two ways: synchronously by means of a call statement or asynchronously by means of a send statement. An operation can be serviced in two ways: by a procedure-like object called a proc or by `in` statements. (SR's `in` statement combines and generalizes aspects of Ada's accept and select statements.) This yields the following four combinations:

Invocation	*Service*	*Effect*
call	**proc**	procedure call
call	**in**	rendezvous
send	**proc**	dynamic process creation
send	**in**	message passing

A guard on an `in` statement can also contain a *synchronization* expression and a *scheduling* expression. The former specifies which invocations are acceptable; the latter specifies the order in which to service acceptable invocations. These expressions can reference the invocation parameters. SR's `reply` statement allows a servicing process to send an "early reply" to its invoker, after which the invoking and servicing processes both continue their executions. SR's `forward` statement defers replying to an invocation and instead passes on this responsibility to another operation.

3 The JR Programming Language

To remedy the shortcomings of Java's concurrency model discussed in Section 2.1, we have designed a new language, a superset of Java, which we call JR. JR adapts the following features from SR (see Section 2.2): dynamic remote virtual machine creation, dynamic remote object creation, remote method invocation, dynamic process creation, support for rendezvous, and asynchronous message passing. In JR, Java classes take the place of SR resources and Java methods take the place of SR procs.

JR provides SR-like operations in a novel object-oriented fashion. Figure 1 shows the general inheritance hierarchy of operation classes. As indicated in the figure, an operation is represented by the (abstract) base class Op; this base class declares abstract methods for invoking and servicing an operation (`call`, `send`, and `in`). The two possible ways to service invocations — via a proc or via `in` statements — are, then, two classes, ProcOp and InOp, derived from the Op base class. The ProcOp and InOp classes define the methods for invoking and servicing an operation. For example, the InOp class defines these methods such that their actions apply to a queue of invocations declared local to the class. Conceptually (as well as in the implementation, see Section 5), the JR statements for invoking and servicing an operation are translated to invocations of methods in the operation's class.

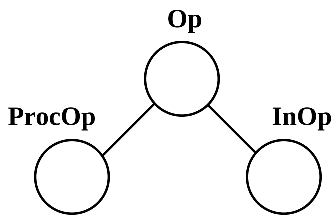

Figure 1: General JR Operation Inheritance Hierarchy

The rest of this section describes the concurrent programming features provided by the JR programming language. As a running example, we demonstrate a simple solution to the readers/writers problem.

```
 1 public class rwMain {
 2   public static void main(String [] args) {
 3     op void done(); // create a local op
 4
 5     // parse command line arguments to determine
 6     // #readers, #writers and Server destination host
 7     ...
 8
 9     // create on specified host
10     cap vm servVM = new cap vm() on args[0];
11     cap JRobject Server serv =
12       new cap JRobject Server() on servVM;
13
14     for (int i = 0; i < readers; i++)
15       new Reader(r_iters, serv, done);
16     for (int i = 0; i < writers; i++)
17       new Writer(w_iters, serv, done);
18
19     int waitfor = readers + writers;
20
21     // wait for each R and W to signal completion
22     while (waitfor > 0) {
23       receive done(); // abbreviated "in" statement
24       waitfor--;
25     }
26   }
27 }
```

Figure 2: Readers/Writers main class in JR

3.1 Dynamic Remote Virtual Machine Creation

JR eliminates Java/RMI's requirement for external setup and interaction with the program (see Section 2.1). Instead, a JR program can dynamically create remote virtual machines upon which remote objects can be instantiated.

To support the creation of remote virtual machines, JR provides a new data-type: virtual machine capabilities. Remote virtual machine creation has the following form:

```
cap vm <vmName> = new cap vm() [on <machineName> | on <vmVar>];
```

The optional "on" clause can be used to specify the host on which the new virtual machine is to be created. The default host is the physical machine of the instantiating process. Line 10 in Figure 2 demonstrates the creation of a virtual machine to house the server object in our readers/writers solution.

3.2 Dynamic Remote Object Creation

JR provides the ability to populate remote virtual machines with new objects through dynamic remote object creation. Remote objects are created using the familiar "new" expression provided by Java. However, a reference to a remote object in JR is stored in a JRobject capability rather than

```
public class Reader {
  int iters;
  cap JRobject Server serv;
  cap void () done;

  protected op void start();

  public Reader(int iters,
                cap JRobject Server serv,
                cap void () done) {
    this.iters = iters;
    this.serv = serv; this.done = done;
    send start();  // start the client thread
  }

  protected void start() {
    while (iters > 0) {
      iters--;
      call serv.readRequest();
      ... // read
      call serv.readRelease();
    }
    // tell main this thread has finished
    send done();
  }
}
```

Figure 3: Reader class (Writer is similar)

```
 1 public class Server {
 2   public op void readRequest();
 3   public op void readRelease();
 4   public op void writeRequest();
 5   public op void writeRelease();
 6   protected op void start();
 7
 8   public Server() {
 9     // create a thread to execute the start method
10     send start();
11   }
12   protected void start() {
13     int nw = 0, nr = 0;
14
15     // on each iteration, service
16     // a R or W request or release
17     while (true) {
18       in readRequest() st (nw == 0) { nr++; }
19       [] writeRequest() st (nw == 0) && (nr == 0)
20          { nw++; }
21       [] readRelease() { nr--; }
22       [] writeRelease() { nw--; }
23       ni
24     }
25   }
26 }
```

Figure 4: Readers/Writers Server in JR

a standard Java reference. The instantiation of a remote object has the following general form:

```
cap JRobject <class name> <var name> = new cap JRobject
  <class name>(<actual arguments>) [on vmRef];
```

An example of the creation of our remote readers/writers server can be seen on lines 11 and 12 in Figure 2.

A JRobject capability provides the interface through which a remote object may be manipulated. This interface is the set of operations defined by the object, e.g., lines 2-6 in Figure 4. Operations are discussed in Section 3.3.

3.3 Operations and Operation Capabilities

In a standard RMI program, remote objects "export" a communication interface that defines the methods that may be invoked remotely. In JR, a remote object's communication interface is defined through a general abstraction called an operation. The general form of an operation declaration is:

```
<modifiers> op <return type> <opname>(<formal argument list>) [exceptions];
```

In JR, an operation definition consists of an operation declaration and an implementation. An operation declaration defines the signature (i.e., formal parameter types, return type, and exception types) of the operation and adds the operation to the specification of the class. Unlike SR, JR allows overloaded operations in much the same way that Java allows overloaded methods. An operation's implementation is defined by either a method with a matching signature or a set of `in` statements that service the operation. An operation that is implemented by a method is called a ProcOp. Each invocation of a ProcOp is "serviced" by executing the body of the method associated with the ProcOp. In Figure 4, for example, the `start` operation declared on line 6 is implemented by the method defined on line 12.

An operation that is implemented by a set of `in` statements is called an InOp. By default, an operation that is declared without a corresponding method is considered to be an InOp. The declaration of an InOp implicitly defines an implementation that consists of an invocation queue. When an InOp is invoked, an invocation is placed in the operation's invocation queue until an `in` statement services the invocation. Each invocation is only serviced by a single `in` statement, which executes the body of code that corresponds to the arm servicing the operation. As such, the actual implementation of an InOp is provided by a set of `in` statements that service the operation. In Figure 4, for instance, the `in` statement on lines 18 – 23 services invocations of the operations defined in lines 2 – 5.

As in SR, operations in JR can be passed as arguments to methods, returned as results from methods, and assigned to variables through the use of operation capabilities. A JR operation capability is declared as follows:

```
cap <return type> ([<formal argument types>])
  [throws <exception types>] <variable name list>;
```

An operation capability will typically refer to an operation with a matching signature. In general, however, an operation capability may refer to an operation with less specific formal argument types and a more specific return type[†].

Recall that lines 2 – 6 in Figure 4 demonstrate the declarations of the set of operations supported by the server object in our example. An explicit, although simple, use of an operation capability can be seen on line 24 in Figure 3. This capability stores a reference to the done operation defined on line 3 of the main class in Figure 2. The done operation is used by each of the reader threads and writer threads to notify the main class that the thread has completed.

3.4 Asynchronous Message Passing

JR supports asynchronous communication via a `send` statement. If an operation invoked by `send` is serviced by a method (i.e., the operation is implemented by a method), then a new thread is created to execute the method. If the operation is serviced by an `in` statement, then a message is created to store the arguments of the invocation. This message is then added to the invocation queue for the corresponding `in` operation.

4 Operations and Inheritance

In JR, a derived class may modify the implementation of its inherited specification by redefining the implementation of its inherited methods and operations. An inherited method's implementation is modified, as in standard Java, by redefining the method. In general, JR allows a subclass to

[†]Type α is less specific than type β if type β can be implicitly cast to type α

redefine the implementation of an inherited operation as either a ProcOp or an InOp, regardless of the operation's implementation in the superclass. Redefinition of an operation's implementation requires an explicit redeclaration of the operation in the subclass only if the redefinition changes the operation from an InOp to a ProcOp or vice-versa. Otherwise, an explicit redeclaration of the operation is not required.

The notation $\mathrm{Op}_1 \rightarrow \mathrm{Op}_2$ means that the superclass defines the operation as an Op_1 and the subclass is redefining the operation to be an Op_2.

1. ProcOp → ProcOp

   A redefinition from a ProcOp to a ProcOp corresponds directly to a method redefinition in standard Java. The subclass can simply redefine the method associated with the operation. Such a redefinition allows a subclass to specialize the operation implementation.

2. InOp → InOp

   The implementation of an InOp is not actually redefined but rather extended. Any `in` statements that "service" an inherited InOp are added to the set of `in` statements that implement the operation. A subclass may explicitly redeclare an InOp as an InOp by explicitly redeclaring the operation. This redeclaration allows a subclass to relax access restrictions but does not create a separate invocation queue.

3. ProcOp → InOp

   A ProcOp may be redefined as an InOp in a subclass by explicitly redeclaring the operation and not defining a signature-compatible method. The signature-compatible method that would have been inherited from the superclass is ignored.

   Figure 5 shows a subclass that redefines a ProcOp to be an InOp. This example demonstrates how such a redefinition can be used to distribute the servicing of the operation's invocations without changing the client. The subclass redefines the operation to be an InOp, creates Worker objects, and passes a capability to the InOp (created automatically) to the `init` method of each Worker object. Each Worker object repeatedly executes an `in` statement to service invocations on the InOp's queue. These Worker objects can be located on an arbitrary set of physical machines as specified by the `remoteHosts` array.

4. InOp → ProcOp

   An InOp may be redefined as a ProcOp in a subclass by both redeclaring the operation and defining a signature-compatible method.

   Figure 6 shows a subclass that redefines an InOp to be a ProcOp. This example demonstrates how such a redefinition can be used to filter the invocations of the operation, reducing the amount of work done by the Worker objects. `Server` defines the `serv` operation as an InOp on line 3. `FilterServer` redefines the `serv` operation to be a ProcOp on lines 28 – 32.

   Each invocation of the operation defined in the subclass is routed through the method associated with the ProcOp to determine whether or not the invocation will be passed on to a Worker object. If the invocation is not rejected by the filter, then the subclass uses a `forward` statement to pass responsibility for servicing the operation to the InOp defined in the parent class (`Server`). Each Worker object repeatedly executes an `in` statement to service the InOp defined in the `Server` class; this is the operation that is passed to the `init` method of the Worker object on line 45 in Figure 6.

```
public class Server {
  // operation declaration
  public op int serv(int i);

  // signature-compatible method
  public int serv(int i) {
    // compute and return value
  }
}

public class BagServer extends Server {
  // redeclaration of operation
  public op int serv(int i);

  public BagServer(cap vm [] remoteHosts) {
    // initialize Worker objects
    for (int i = 0; i < remoteHosts.length; i++) {
      cap JRobject Worker w =
        new cap JRobject Worker() on remoteHosts[i];
      send w.init(serv);
    }
  }
}

public class Worker {
  // operation declaration
  public op void init(cap int (int) server);
  public op void init(cap int (int) server) {
    while (true) {
      in server(int i) {
        // compute and return value
      }
      ni
    }
  }
}
```

Figure 5: Redefining an operation to distribute its implementation

```
public class Server {
  // operation declaration
  public op long serv(int i);

  public Server() {}    // empty constructor
  public Server(cap vm [] remoteHosts) {
    // initialize Worker objects
    for (int i = 0; i < remoteHosts.length; i++) {
      cap JRobject Worker w =
        new cap JRobject Worker() on remoteHosts[i];
      send w.init(serv);
    }
  }
}

public class Worker {
  // operation declaration
  public op void init(cap long (int) server);
  public op void init(cap long (int) server) {
    while (true) {
      in server(int i) { /* compute and return value */ } ni
    }
  }
}

public class FilterServer extends Server {
  // redeclaration of operation
  public op long serv(int i);
  public long serv(int i) {
    if (filter(i)) return DEFAULT; // or exception
    else forward super.serv(i);  // forward to Worker objects
  }

  boolean filter(int i) {  // simple filter
    return (i < 0);
  }

  public FilterServer(cap vm [] remoteHosts) {
    super();
    // initialize Worker objects
    for (int i = 0; i < remoteHosts.length; i++) {
      cap JRobject Worker w =
        new cap JRobject Worker() on remoteHosts[i];
      // reuse super operation
      send w.init(super.serv);
    }
  }
}
```

Figure 6: Redefining an operation to serialize invocations through a filter

5 Implementation

The current JR implementation extends the Java compiler available in SUN's JDK, Version 1.2.1. The JR translator converts JR programs into standard Java programs that are supported by the JR run-time system. The JR run-time system is also implemented in standard Java. This section discusses the implementation of each of the features provided by JR.

5.1 JR Virtual Machines

In the current implementation of JR, remote virtual machines are created by contacting a centralized virtual machine manager called JRX. JRX uses rsh to contact the remote host and execute the JR virtual machine (jrvm) program. The JR virtual machine is a small Java program that implements an interface with which other jrvm's communicate to create objects on the physical machine.

5.2 Remote Objects

JRobject capabilities are implemented as Serializable objects that contain references to operations. These references are operation capabilities that also implement the Serializable interface. A JRobject capability contains an operation capability for each operation in the class' interface. JRobject capabilities mimic the inheritance hierarchy of the classes with which they are associated.

5.3 Operations and Operation Capabilities

Figure 7 shows the actual inheritance hierarchy of operation classes in JR. This hierarchy is a specialization of the inheritance hierarchy in Figure 1.

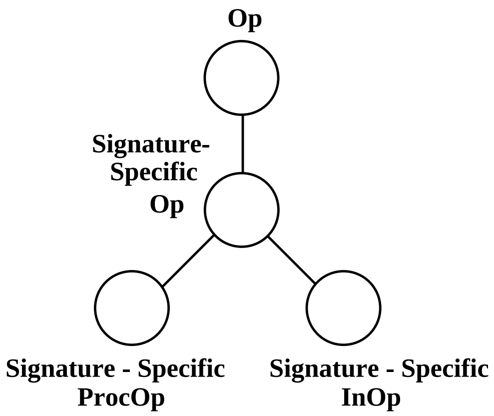

Figure 7: Actual JR Operation Inheritance Hierarchy

Each proc operation is implemented as a separate ProcOp object defined within the class that declares the operation. This is done so that a ProcOp object may be associated with a private method within the class definition. Invocations of the operation are translated into invocations of the appropriate method (i.e., call, send, etc.) in the ProcOp object.

An `in` operation is implemented as an InOp object that contains a message queue to store the arguments for each invocation. An invocation of an InOp is translated into an appropriate method invocation on the corresponding object. All operations implement the RMI Remote interface so that the methods can be invoked from remote hosts.

5.4 Asynchronous Message Passing

The current implementation of the JR run-time system is built using RMI. As such, the send statement is not truly asynchronous in the traditional sense. A send is implemented as an RMI invocation of the send method in the object that corresponds to the operation being invoked. An InOp's send method places a message containing the actual arguments into the invocation queue and then returns. A ProcOp's send method spawns a new thread to execute the method associated with the ProcOp and then returns, releasing the invoking process.

5.5 Inheritance

In Java, a method invocation causes a dynamic lookup to determine the actual method to invoke. However, this lookup is not done when accessing a data field. Each operation in JR is implemented by a signature-specific Op object, so the generated Java code must provide support for dynamic lookup of operations. This support is provided by access methods used to retrieve the appropriate operation object.

5.6 Remote Class Loading

The dynamic class loading described in the Java RMI specification [42] allows for class files to be loaded from either the local CLASSPATH or from a predefined URL. The JR run-time system requires only that necessary class files for the program be accessible through the CLASSPATH at the originating host (where the program is initially executed). When a remote object is created, the necessary class files are retrieved from the JRX object on the originating host through a custom class loader. This reduces the amount of setup required by the user and eliminates the need for a web server to provide file access.

6 Performance Results

Though performance has not been a primary goal in this initial phase of the JR project, our performance results demonstrate that method invocations in JR incur little overhead compared with equivalent invocations in standard RMI. Each of the following experiments was conducted on a cluster of 450 MHz Intel Pentium II workstations connected with a 10 Mbps Ethernet network. All experiments were conducted using the Linux port of SUN's JDK, Version 1.2.2.

The first experiment, taken from [29], demonstrates the time needed to invoke an empty method in a remote object. Table 1 shows the results of repeatedly invoking an empty ProcOp in JR and an empty method using standard RMI. The remote method takes as an argument a single object that contains an array of a specified number of integers.

The performance differences demonstrated in Table 1 are attributable to method invocation overhead inherent in the current implementation of JR. A remote method invocation begins by

Object Size	(*ints*)	1	$1k$	$10k$	$100k$
JR	(ms)	2.42	5.93	53.8	497.5
JDK RMI	(ms)	2.11	5.52	51.8	497.2

Table 1: Time to invoke an empty proc operation in JR and an empty method using standard RMI.

invoking the call method of the operation capability. The operation capability call method invokes the call method of the ProcOp. This invocation transmits the parameters to the remote host using RMI. At the remote host, the ProcOp call method invokes the actual user-defined method.

R/W/RI/WI		20/10/3/3	50/15/3/3	100/30/3/5
JR	(ms)	5257.6	7266.6	17570.7
JDK RMI	(ms)	4190.7	5663.7	8802.2

Table 2: Time to complete execution of all iterations (RI and WI) for all readers (R) and writers (W).

Table 2 shows the results of multiple executions of the readers/writers program using both JR and RMI. As Figure 4 shows, the readers/writers server uses `in` statements to service invocations of the different InOps associated with requesting and releasing read/write access.

The overhead associated with invocations of an InOp can be attributed primarily to object creation. In order to support return statements within the body of an `in` statement arm, there must be a mechanism to return the value to the invoker. Each invocation of an InOp currently requires that an object be created to support this return statement. In addition, a message object is created to place the invocation arguments into the message queue.

Number of coefficients		10000	100000	1000000
Sequential Java	(s)	79.0	851.5	12953.0
JR (1 Server)	(s)	79.4	863.7	13057.6
JR (2 Servers)	(s)	43.5	433.1	6566.9
JDK RMI (1 Server)	(s)	77.4	784.5	12131.6
JDK RMI (2 Servers)	(s)	45.1	392.0	7190.6

Table 3: Time to calculate the first n coefficients of the function $(x+1)^x$ defined on the interval [0,2].

Table 3 compares the performance results of the standard sequential version of the Java Grande *Fourier* Benchmark [17] against distributed versions written in JR and RMI.

The distributed versions of the program divide the computation equally among the available servers. The JR program uses asynchronous message-passing to initiate each computation and then collects the results using an `in` statement. The RMI version uses threads to concurrently initiate invocations of the remote method and to collect the results. These results demonstrate that distributed JR programs scale well.

7 Related Work

Other extensions to Java have modified its concurrency model to include, for example, asynchronous communication, distributed shared memory, and active agents. None of these extensions provide the flexibility of operations, capabilities, and `in` statements. Moreover, many of the previous extensions still require the user to manually start remote programs.

Ajents [29] provides remote object creation, asynchronous RMI, and object migration through a collection of Java classes. The Ajents project makes no modifications to the Java programming language or the run-time system. Creation of remote objects and invocation of methods within remote objects (both synchronous and asynchronous) is done through the **Ajents** class. The arguments to **Ajents.new()**, **Ajents.rmi()**, and **Ajents.armi()** include a String specifying the type of the object to create or the method to invoke. Without a preprocessor, it is not possible to statically determine if an object can be created or if the method being invoked actually exists.

JavaParty [37] extends Java through the addition of the **remote** keyword to provide transparent remote objects. JavaParty includes a translator that converts JavaParty programs into standard Java programs. This translation includes converting the **remote** classes into the corresponding RMI support classes simplifying the work of the programmer.

In [34], Java is extended to include a **remotenew** expression that allows for the instantiation of remote objects on specified hosts. The **remotenew** keyword is mapped to a new opcode which requires an extended virtual machine. The implementation restricts remote method arguments to primitive types whereas JR allows any Serializable or Remote object as an argument.

Asynchronous remote method invocations are provided by [38] by using the **armic** stub/skeleton generator instead of the standard rmic. Unfortunately, all remote method invocations are asynchronous. The programmer is provided access to return values through a mailbox.

A socket implementation of asynchronous message passing is discussed in [26]. This implementation allows a single object to be passed as a message between two threads and provides support for a conditional receive (a very limited `in` statement). Support for rendezvous is provided via a blocking request at the "client" and a request/reply pair at the "server".

Both Java/DSM [43] and Charlotte [8] extend Java to include mechanisms for distributed shared memory. Javelin [15], SuperWeb [1], and ParaWeb [11] seek to exploit the potential for parallel computation using the World Wide Web. Communicating Java Threads [27] extends the concurrency model of Java by providing communication between threads based on the CSP [28] paradigm. Support for data-driven objects in Java is discussed in [30].

The current JR implementation uses but does not rely upon RMI. As such, future JR implementations can take advantage of improved or optimized communication frameworks. For example, [32] and [35] discuss more efficient versions of Java's RMI.

Java-SR [33] adds SR operations, asynchronous message passing, and dynamic resource creation to Java. Though a good initial effort, Java-SR has several shortcomings. Java-SR uses a preprocessor to translate programs into standard Java but adds little to the Java syntax. The result is a mix of syntactic extensions to define operations and exposed implementation details for interacting with operations. Java-SR does not truly integrate the SR concurrency model (e.g., no VMs) and Java.

Java-SR does not address the extension of the SR constructs in terms of the Java programming language.

More generally, numerous concurrent object-oriented languages have been proposed, e.g., as discussed in a recent survey [13]. These languages have various concurrency models, ways of expressing synchronization, etc. Some concurrent object-oriented languages take an object-oriented approach to synchronization (as does JR), including Simtalk [9], Actalk [14], GARF [25, 18], and BAST [19, 20]. SimTalk, for example, derives different kinds of monitor classes from a common base class. Although JR shares some features with some concurrent object-oriented languages, JR differs in its overall approach of building synchronization via the operation abstraction, its overall concurrency model, and its definition as an extension of Java.

Furthermore, the survey classifies the different approaches in object-oriented concurrent programming into three coarse categories [13]. The three categories are: the *library* approach, the *integrative* approach, and the *reflective* approach. The *library* approach provides class libraries that encapsulate concurrency components (e.g., Java threads are represented as objects). The *integrative* approach unifies concurrency concepts with object-oriented concepts (e.g., merging the notion of *object* and *process* to create the notion of an *active object*). The *reflective* approach uses reflection mechanisms to provide concurrency components at the meta-level.

JR takes an *integrative* approach in defining its concurrency model. JR provides operations as a general communication abstraction. Operations are defined as part of an object and each operation is associated with a specific object. Using operations, a JR programmer can create active objects, synchronized objects, and distributed objects. An object can be made active by using the `process` keyword to create a thread within the object. A JR programmer can use Java's `synchronized` keyword or InOps to synchronize invocations. As discussed in Section 3.2, JR provides dynamic remote object creation and JRobject capabilities to facilitate the distribution of objects.

8 Conclusion

The JR programming language integrates the SR concurrency model and the Java programming language. It does so via a novel approach that resolves the tension between inheritance and concurrency. JR provides a more flexible way to program distributed applications without great performance costs.

Further work will complete the integration of the SR concurrency model and the Java programming language. We are investigating a more generalized form of the `in` statement that will allow more control over the order in which invocations are serviced. We are also extending Java's exception model to handle exceptions that occur during asynchronous communication. This includes exceptions that are thrown by a method or `in` statement invoked via send, as well as exceptions thrown after a `reply` or `forward` statement (Section 2.2) has been executed.

Our goal for the initial implementation of the JR system has been to improve the concurrency model provided in an extended Java. Performance has not been a primary concern. Though the current implementation adds little overhead to RMI equivalent method invocations, future work will further optimize the JR run-time system. Performance can be improved by further tuning the JR run-time system but greater performance gains will come from optimizations done through static code analysis and specialized translations. Additional work will study the use of communication frameworks other than RMI to improve performance.

References

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

[2] G. R. Andrews. *Concurrent Programming: Principles and Practice*. Benjamin/Cummings, 1991.

[3] G. R. Andrews and R. A. Olsson. The evolution of the SR language. *Distributed Computing*, 1(3):133–149, 1986.

[4] G. R. Andrews and R. A. Olsson. *The SR Programming Language: Concurrency in Practice*. The Benjamin/Cummings Publishing Co., Redwood City, California, 1993.

[5] G. R. Andrews, R. A. Olsson, M. Coffin, I. Elshoff, K. Nilsen, T. Purdin, and G. Townsend. An overview of the SR language and implementation. *ACM Transactions on Programming Languages and Systems*, 10(1):51–86, January 1988.

[6] M. S. Atkins. Experiments in SR with different upcall program structures. *ACM Transactions on Computer Systems*, 6(9):365–392, November 1988.

[7] M. S. Atkins and R. A. Olsson. Performance of multi-tasking and synchronization mechanisms in the programming language SR. *Software – Practice and Experience*, 18(9):879–895, September 1988.

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

[9] Jean Bezivin. Some experiments in object-oriented simulation. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 394–405, 1987.

[10] A. P. Black. Supporting distributed applications: Experience with Eden. In *Proceedings of the Tenth ACM Symposium on Operating Systems Principles*, pages 181–193, 1985. In *ACM Operating Systems Review* 19(5).

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

[12] P. Brinch Hansen. Java's insecure parallelism. *ACM SIGPLAN Notices*, 34(4):38–45, April 1999.

[13] J.-P. Briot, R. Guerraoui, and K.-P. Lohr. Concurrency and distribution in object-oriented programming. *ACM Computing Surveys*, 30(3):291–329, September 1998.

[14] Jean-Pierre Briot. Actalk: A testbed for classifying and designing actor languages in the Smalltalk-80 environment. In S. Cook, editor, *Proceedings ECOOP'89*, pages 109–129, Nottingham, July 10-14 1989. Cambridge University Press.

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

[16] P. R. Cook. *mod–a language for distributed programming. *IEEE Transactions on Software Engineering*, 6(6):563–571, November 1980.

[17] Edinburgh Parallel Computing Centre. Java Grande Forum Benchmark Suite. http://www.epcc.ed.ac.uk/research/ javagrande/benchmarking.html.

[18] B. Garbinato and R. Guerraoui. Flexible protocol composition in Bast. In *Proceedings of the 18th International Conference on Distributed Computing Systems (ICDCS-18)*, pages 22–29, Amsterdam, The Netherlands, May 1998. IEEE Computer Society Press.

[19] Benoît Garbinato, Pascal Felber, and Rachid Guerraoui. Protocol classes for designing reliable distributed environments. In Pierre Cointe, editor, *ECOOP '96—Object-Oriented Programming*, volume 1098 of *Lecture Notes in Computer Science*, pages 316–343. Springer, 1996.

[20] Benoît Garbinato and Rachid Guerraoui. Using the strategy design pattern to compose reliable distributed protocols. In USENIX, editor, *The Third USENIX Conference on Object-Oriented Technologies and Systems (COOTS), June 16–19, 1997. Portland, Oregon*, pages 221–232, Berkeley, CA, USA, June 1997. USENIX.

[21] N. Gehani and W.D. Roome. Rendezvous facilities: Concurrent C and the Ada language. *IEEE Transactions on Software Engineering*, 14(11):1546–1553, November 1988.

[22] N. Gehani and W.D. Roome. *The Concurrent C Programming Language*. Silicon Press, Summit, NJ, 1989.

[23] N. Gehani and W.D. Roome. Message passing in Concurrent C: Synchronous versus asynchronous. *Software – Practice and Experience*, 20(6):571–592, June 1990.

[24] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Java Series. Sun Microsystems, 1996. ISBN 0-201-63451-1.

[25] Rachid Guerraoui, Benoît Garbinato, and Karim R. Mazouni. Garf: A tool for programming reliable distributed applications. *IEEE Concurrency*, 5(4):32–39, October/December 1997.

[26] S. J. Hartley. *Concurrent Programming: The Java Programming Language*. Oxford University Press, 1998.

[27] G. Hilderink, J. Broenink, Wiek Vervoort, and A. Bakkers. Communicating Java threads. In *WoTUG-20*, 1997.

[28] C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8), 1974.

[29] M. Izatt, P. Chan, and T. Brecht. Ajents: Towards an environment for parallel, distributed and mobile Java applications. In *ACM 1999 Java Grande Conference*, 1999.

[30] L. V. Kalé, M. Bhandarkar, and T. Wilmarth. Design and implementation of parallel Java with global object space. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications*, 1997.

[31] B. Liskov, M. Herlihy, and L. Gilbert. Limitations of remote procedure call and static process structure for distributed computing. In *Proceedings of 13th ACM Symposium on Principles of Programming Languages*, St. Petersburg, FL, January 1986.

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

[33] C. N. Mendis. Java-synchronising resources for concurrent and distributed programming. Master's thesis, University of Warwick, 1997.

[34] N. Nagaratnam and A. Srinivasan. Remote objects in Java. In *IASTED International Conference on Networks*, 1996.

[35] C. Nester, M. Philippsen, and B. Haumacher. A more efficient RMI for Java. In *ACM 1999 Java Grande Conference*, 1999.

[36] R. A. Olsson. Using SR for discrete event simulation: A study in concurrent programming. *SOFTWARE — Practice and Experience*, 20(12):1187–1208, December 1990.

[37] M. Philippsen and M. Zenger. JavaParty – transparent remote objects in Java. In *ACM 1997 Workshop on Java for Science and Engineering Computation*, 1997.

[38] R. Raje, J. I. William, and M. Boyles. An asynchronous remote method invocation (ARMI) mechanism for Java. In *ACM 1997 Workshop on Java for Science and Engineering Computation*, 1997.

[39] M. L. Scott. Messages vs. remote procedures is a false dichotomy. *ACM SIGPLAN Notices*, 18(5):57–62, May 1983.

[40] M. L. Scott. Language support for loosely coupled distributed programs. *IEEE Transactions on Software Engineering*, 13(1):88–103, January 1987.

[41] M. L. Scott. The Lynx distributed programming language: Motivation, design and experience. *Computer Languages*, 16(3/4):209–233, 1991.

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

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