JR: Flexible Distributed Programming in an Extended Java

AARON W. KEEN, TINGJIAN GE, JUSTIN T. MARIS, and RONALD A. OLSSON
University of California, Davis

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, based on that of SR. JR provides dynamic remote virtual machine creation, dynamic remote object creation, remote method invocation, asynchronous communication, rendezvous, and dynamic process creation. JR's concurrency model stems from the addition of operations (a generalization of procedures) and JR supports the redefinition of operations through inheritance. 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.

Categories and Subject Descriptors: D.3.3 [**Programming Languages**]: Language Constructs and Features—*Concurrent programming structures*

General Terms: Design, Languages

Additional Key Words and Phrases: Concurrency, concurrent object-oriented programming, Java, SR

1. INTRODUCTION

Java [Gosling et al. 1996] 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 [Andrews et al. 1988, Andrews and Olsson 1993]. 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,

Authors' addresses: A. W. Keen, T. Ge, J. T. Maris, and R. A. Olsson, Dept. of Computer Science, University of California, Davis, Davis, CA 95616.
This research was supported in part by the National Science Foundation grant CCR-9527295.
A preliminary version of this paper was presented at the 21st IEEE International Conference on Distributed Computing Systems (ICDCS 2001), held in Phoenix, Arizona, in April 2001.

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. JR can certainly be used in development, but it has limited tool support (e.g., IDE's, profilers, etc.) because of its extended syntax. 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.

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. Further discussion of JR can be found in [Keen 2002].

2. MOTIVATION AND BACKGROUND

Often described as simple and elegant, the Java programming language [Gosling et al. 1996] 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 providing a package of classes for synchronization and communication (e.g., as RMI does). This approach has potential advantages because synchronization is represented as actual language primitives [Black 1985; Scott 1991]. 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.

*The JR implementation and a collection of example programs are available at `http://www.cs.ucdavis.edu/~olsson/research/jr/`.

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

— It provides only a single synchronization primitive — remote method invocation — for distributed programming. (Java also has a socket package, but we consider sockets to be too low-level.)

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 (e.g., 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 mechanism have been widely discussed in the literature [Scott 1983; Andrews and Olsson 1986; Liskov et al. 1986, Gehani and Roome 1988, Gehani and Roome 1990]. 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 [Liskov et al. 1986]. 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 [Gehani and Roome 1989], Lynx [Scott 1987; Scott 1991], StarMod [Cook 1980], and SR [Andrews et al. 1988, Andrews and Olsson 1993]. Some work [Atkins and Olsson 1988] has shown that an implementation of such a language (in [Atkins and Olsson 1988], SR) can provide several synchronization mechanisms at a reasonable cost. Having multiple synchronization primitives proved useful in programming different upcall program structures [Atkins 1988]. Additional work [Olsson 1990] 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 [Brinch Hansen 1999] contend that Java does not really support monitors). 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

```
resource Main()
  import Philosopher, Servant
  var n, t: int
  read(n); read(t)
  # create the Servant and Philosophers
  var s: cap Servant
  s := create Servant(n)
  fa i := 1 to n ->
    # (returned resource cap not needed)
    create Philosopher(s, i, t)
  af
end
```

Fig. 1. Main resource in multiple-resource Dining Philosophers written in SR.

tasks, probe/echo, broadcast, etc.[Andrews 1991]

2.2 The SR Concurrent Programming Language and its Concurrency Model

The SR concurrent programming language [Andrews et al. 1988, Andrews and Olsson 1993] 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.

Resource instances are created via the `create` statement, which returns a resource capability for the newly created resource instance. A resource capability contains operation capabilities for all operations defined in the resource's specification; their values are assigned to those operations just created in the newly created resource instance. As an example, Figures 1 and 2 present a version of Dining Philosophers. It employs centralized control, but it places philosophers and servants in different resource instances. The main resource consists entirely of "initial" (top-level) code. This code is executed when an instance of the resource is created; in the case of `Main`, its initial code is executed when the program begins execution. `Main`'s initial code creates one instance of the `Servant` resource and `n` instances of the `Philosopher` resource, passing to each philosopher its integer identity (`i`), the capability for the servant (`s`), and the number of iterations it should execute (`t`). Each philosopher requests forks from the servant, eats, and later releases forks. (The code inside the `Servant` resource is discussed below.)

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.)

‡An operation may also be synchronously invoked as an expression without the call keyword.

```
resource Philosopher
  import Servant
body Philosopher(s: cap Servant; id, t: int)
  process phil
    fa i := 1 to t ->
      # get forks from Servant instance s
      s.getforks(id)
      # eat
      write("Philosopher", id, "is eating")
      # release forks back to Servant s
      s.relforks(id)
      # think
      write("Philosopher", id, "is thinking")
    af
  end
end
```

```
resource Servant
  op getforks(id: int) # these ops are invoked
  op relforks(id: int) # by Philosophers
body Servant(n: int)
  process server
    var eating[1:n] := ([n] false)
    do true ->
      in getforks(id)
          st  not eating[(id mod n) + 1]
          and not eating[((id-2) mod n) + 1] ->
              eating[id] := true
      [] relforks(id) ->
              eating[id] := false
      ni
    od
  end
end
```

Fig. 2. Philosopher and servant resources in multiple-resource Dining Philosophers written in SR.

```
op writeToFile(prio, data: int)
...
in writeToFile(prio, data) by prio ->
  # perform write operation
ni
```

Fig. 3. An SR scheduling expression to service invocations in order of priority.

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. To illustrate, consider the `in` statement in the `Servant` resource in Figure 2. The first arm of the server's `in` statement uses a synchronization expression, introduced by `st` (such-that), to accept an invocation of `getforks` from philosopher `id` only when that philosopher's neighbors are not eating. Note how the synchronization expression references the invocation's parameter, `id`. Figure 3 gives an example of a scheduling expression that services invocations in order of priority. When the input statement executes, the pending invocations are examined and the longest pending invocation that minimizes the scheduling expression (`prio` in this example) is selected for servicing.

As noted previously, SR defines the notion of a *virtual machine* (VM) as its language mechanism for distributing a program onto physical machines. Like resource instances, VMs are created dynamically via the `create` statement. The program in Figures 1 and 2 executes on a single virtual machine and, therefore, on a single physical machine. It can be easily modified, though, so that each philosopher executes on a different virtual machine, each on a different physical machine. Only `Main`'s loop needs to be changed, for example, as shown in Figure 4. The `on` clause in the first `create` specifies the physical machine on which the VM is to be created;

```
fa i := 1 to n ->
var vmcap: cap vm
  vmcap := create vm() on host[i]
  create Philosopher(s, i, t) on vmcap
af
```

Fig. 4. Modified `Main` loop for multiple-VM Dining Philosophers written in SR.

the value returned by this `create` statement is a capability for the newly created virtual machine. The `on` clause in the second `create` specifies the virtual machine on which the resource instance is to be created. The array `host` contains physical machine names (as strings; its initialization is not shown).

SR provides abbreviations for commonly occurring uses of operations. For example, the `process`[§] keyword (used in Figure 2) is defined as an abbreviation for an operation declaration, a proc, and one or more sends to that proc by the resource's initial code. Other abbreviations include a `receive` statement, which is a simple form of `in`, and semaphore declarations and `P` and `V`, which are simple forms of operation declarations and `send` and `receive` statements.

SR also provides a few other statements that deal with invoking or servicing operations. SR's concurrent invocation statement provides a way to start several invocations at the same time; it terminates when all its invocations have completed. 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 5 depicts a conceptual inheritance hierarchy for operations. As indicated in the figure, all operations are derived from a base class Op; this base class provides (abstract) methods for invoking (`call` and `send`) and servicing (`inni`[¶]) operations. The two possible ways to service invocations — via a method or via `inni` statements — give rise to two subclasses of Op: ProcOp (borrowing SR terminology) and InOp. The ProcOp and InOp classes define the actual methods to support 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. The actual implementation of operations is discussed in Section 5.

[§]SR's processes do not correspond to OS processes, but rather to threads in SR's run-time system.
[¶]The `inni` statement is JR's version of SR's input statement. `inni` is the concatenation of SR's `in` and `ni` (see Section 2.2) to prevent clashes with the commonly used name `in`.

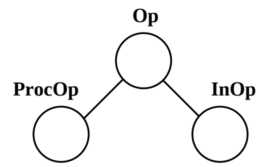


Fig. 5. General JR Operation Inheritance Hierarchy.

```
public class Main {
  public static void main(String [] args) {
    int n, t;
    // get n and t from args
    ...
    // create the Servant and Philosophers
    remote Servant s = new remote Servant(n);
    for (int i = 0; i < n; i++)
      new remote Philosopher(s, i, t)
  }
}
```

Fig. 6. Main class for Dining Philosophers solution in JR.

```
public class Philosopher {
  remote Servant s;
  int id, t;
  public Philosopher(remote Servant s,
   int id, int t) {
    this.s = s; this.id = id; this.t = t;
  }
  process phil() {
    for (int i = 0; i < t; i++) {
      // get forks from Servant instance s
      s.getforks(id);
      // eat
      System.out.println("Philosopher " +
          id + " is eating");
      // release forks back to Servant s
      s.relforks(id);
      // think
      System.out.println("Philosopher " +
          id + " is thinking");
    }
  }
}
```

```
public class Servant {
  public op void getforks(int);
  public op void relforks(int);
  int N;
  boolean [] eating;
  public Servant(int n) {
    this.N = n;
    eating = new boolean[N];
    for (int i = 0; i < N; i++ ) {
      eating[i] = false;
    }
  }
  process server () {
    while (true) {
      inni void getforks(int id) st oktoeat(id) {
              eating[id] = true
      }
      [] void relforks(int id) {
              eating[id] = false
      }
    }
  }
  protected boolean oktoeat(int id) {
    return !eating[((id+N)+1)%N] &&
           !eating[((id+N)-1)%N];
  }
}
```

Fig. 7. Philosopher and Servant classes in Dining Philosophers solution in JR.

Before discussing the individual features of JR, we provide a glimpse forward through a simple JR solution to the Dining Philosophers problem discussed in Section 2.2. The JR solution, presented in Figures 6 and 7, is very similar to the SR solution, including the use of an input statement to synchronize access to the forks.‖ As with the SR solution, the Philosophers and Servant can be created on different virtual machines using "on" clauses.

‖Note that, in the interest of clarity, exception handling is not shown in the following solutions.

```
public class Philosopher extends Thread {
  Servant s;
  int id, t;
  public Philosopher(Servant s, int id, int t) {
    this.s = s; this.id = id; this.t = t;
  }
  public void run() {
    for (int i = 0; i < t; i++) {
      // get forks from Servant instance s
      s.getforks(id);
      // eat
      System.out.println("Philosopher " +
          id + " is eating");
      // release forks back to Servant s
      s.relforks(id);
      // think
      System.out.println("Philosopher " +
          id + " is thinking");
    }
  }
  public static void main(String [] args) {
    // get id and iterations from args
    // get Servant from rmiregistry
    Servant s = (Servant)Naming.lookup("naming string");
    // create a philosopher thread
    (new Philosopher(s, id, t)).start();
  }
}
```

Fig. 8. Philosopher class for Dining Philosophers solution in Java/RMI.

Figures 8 and 9 present a Java/RMI solution to the Dining Philosophers problem. The presented solution is written such that the Philosophers and Servant may reside on different machines. In such a situation, the Servant and each Philosopher are separate programs that must be initiated separately. This example also demonstrates a simple use of the naming service from which the philosophers acquire a remote reference to the Servant object. Fork synchronization is split between the methods `getforks` and `relforks`, which use Java's `synchronized` support. Even for such a relatively simple synchronization algorithm, the programmer must "manage" the wait queue (through the use of `wait` and `notifyAll`). Furthermore, more complicated synchronization algorithms (e.g., waiting philosophers are granted the right to eat in order of priority) may require additional queue management or the implementation of additional synchronization objects.

Note that the given solution uses command-line arguments to determine the number of philosophers to which the servant must attend. A safer, though more complicated, solution might use a registration system to avoid a set limit.

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.

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 references to virtual machines. Remote virtual machine creation has the form:

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

```
public interface Servant
  extends Remote {
  public void getforks(int id)
    throws RemoteException;
  public void relforks(int id)
    throws RemoteException;
}
```

```
public class Servant_impl
 extends UnicastRemoteObject
 implements Servant {
  int N;
  boolean [] eating;
  public Servant_impl(int n)
   throws RemoteException {
    this.N = n;
    eating = new boolean[N];
    for (int i = 0; i < N; i++) {
      eating[i] = false;
    }
  }
  public synchronized void getforks(int id)
   throws RemoteException {
    while (!oktoeat(id)) {
      this.wait();
    }
    eating[id] = true;
  }
  protected boolean oktoeat(int id) {
    return !eating[((id+N)+1)%N] &&
           !eating[((id+N)-1)%N];
  }
  public synchronized void relforks(int id)
   throws RemoteException {
    eating[id] = false;
    this.notifyAll(); // allow others to try
  }
  public static void main(String [] args) {
    // get number of philosophers from args
    Servant s = new Servant_impl(n);
    Naming.rebind("naming string", s);
  }
}
```

Fig. 9. Servant class for Dining Philosophers solution in Java/RMI.

Its meaning is similar to VM creation in SR; e.g., as in Figure 4. The optional "on" clause can be used to specify the host on which the new virtual machine is to be created. This host can be specified as either a Java String, in which case the new VM is created on the host specified by the string, or as another VM variable, in which case the new VM is colocated on the host of the specified VM. The default host is the physical machine of the instantiating process. Line 10 in Figure 10 demonstrates the creation of a virtual machine to house the server object in our readers/writers solution.

A JR virtual machine is a thin layer of services provided in addition to those of the underlying Java virtual machine. Specifically, JR provides, at the language-level, references to virtual machines and support for their dynamic creation. Furthermore, JR's virtual machines service remote requests to create objects, thus providing support for dynamically created remote objects.

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 `remote` reference rather than a standard Java reference. The instantiation of a remote object has the general form:

```
remote <class name> <var name> = new remote <class name> (<actuals>) [on vmRef];
```

```
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    vm servVM = new vm() on args[0];
11    remote Server serv =
12      new remote ServerImpl() 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    JR.exit(0);
27  }
28 }
```

Fig. 10. Readers/Writers main class in JR.

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

  public Reader(int iters,
                remote Server serv,
                cap void () done) {
    this.iters = iters;
    this.serv = serv; this.done = done;
  }

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

Fig. 11. Reader class (Writer is similar).

An example of the creation of our remote readers/writers server can be seen on lines 11 and 12 in Figure 10.

A remote object reference provides the interface through which a remote object may be manipulated. This interface is the set of operations declared by the object's class or interface, e.g., lines 2 – 5 in Figure 12. Operations are discussed in Section 3.3.

```
1 public interface Server {
2   public op void readRequest();
3   public op void readRelease();
4   public op void writeRequest();
5   public op void writeRelease();
6 }
```

```
7 public class ServerImpl implements Server {
8   public op void readRequest();
9   public op void readRelease();
10  public op void writeRequest();
11  public op void writeRelease();
12
13  public ServerImpl() {
14  }
15
16  // create the servicing thread
17  protected process start() {
18    int nw = 0, nr = 0;
19
20    // on each iteration, service
21    // a R or W request or release
22    while (true) {
23      inni void readRequest() st nw==0 { nr++; }
24      [] void writeRequest() st nw==0 && nr==0 { nw++; }
25      [] void readRelease() { nr--; }
26      [] void writeRelease() { nw--; }
27    }
28  }
29 }
```

Fig. 12. Readers/Writers server interface and implementation in JR.

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 by the set of operations declared in the object's class. An operation declaration has the general form:

```
<modifiers> op <return type> <opname> (<formals>) [exceptions];
```

In JR, an operation definition consists of a 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 `inni` statements that service the operation. An operation 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.

An operation implemented by a set of `inni` 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 `inni` statement services the invocation. Each invocation is only serviced by a single `inni` 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 `inni` statements that service the operation. In Figure 12, for instance, the `inni` statement on lines 23 – 26 services invocations of the operations defined in lines 8 – 11.

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. In JR, an operation capability is a reference to an operation; the "cap"

```
abstract class A
{
   public abstract op void foo();
   public abstract op void bar();
}
```

(a) Abstract class.

```
class B extends A
{
   public op void foo();

   public op void bar();
   public void bar()
   { inni void foo() { ... } }
}
```

(b) Concrete subclass.

Fig. 13. Concrete redefinition of abstract operations.

keyword is required to simplify parsing. A JR operation capability is declared as follows:

```
cap <return type> (<formal types>) [<exception types>] <variables>;
```

An operation capability can only refer to an operation with a matching signature.

Recall that lines 2 – 5 (8 – 11 in the implementation) in Figure 12 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 21 in Figure 11. This capability stores a reference to the done operation defined on line 3 of the main class in Figure 10. The `done` operation is used by each of the reader threads and writer threads to notify the main class that the thread has completed. The `done` operation and explicit "JR.exit(0)" are not required in this example because of JR's support for termination detection (see Section 5.7), but are included to demonstrate the use of operations, capabilities, the `receive` statement, and explicit termination.

JR also allows operations to be declared abstract with the restriction that the defining class be declared abstract. An abstract operation is neither an InOp nor a ProcOp since, by definition, it has no implementation. The implementation of an abstract operation in JR is defined in a subclass (making the operation concrete) just as the implementation of an abstract method in Java is defined in a subclass. Figure 13 demonstrates the definition of two abstract operations, in Figure 13 (a), that are made concrete in the extending class, in Figure 13 (b). Operation `foo` is implemented as an InOp by the subclass whereas operation `bar` is implemented as a ProcOp.

Unlike SR, JR allows overloaded operations in much the same way that Java allows overloaded methods. In Java, when an overloaded operation is invoked, the number of actual arguments and the types of these arguments (more precisely, the type of the reference in the case of object arguments) are used to determine which instance of the overloaded operation is invoked. However, in JR, this resolution mechanism is complicated by the fact that operations and methods can accept operations, which themselves may be overloaded, as arguments. JR extends the resolution mechanism to handle such a situation.

Consider the invocation of the overloaded method `foo` with the overloaded operation `f` as its argument on line 12 of Figure 14. The resolution mechanism first determines the most specific[**] definition of `foo` for which any definition of `f` is a

[**]Type β is more specific than type α if type β can be implicitly cast to type α. For capability types, β is more specific than α if α's argument types are more specific and its return type is less specific (i.e., contravariant in the argument types and covariant in the return types, see

```
 1 static void foo(cap void(int) c)
 2 { System.out.println("foo int"); }
 3
 4 static void foo(cap void(long) c)
 5 { System.out.println("foo long"); }
 6
 7 static op void f(int);
 8 static op void f(long);
 9
10 public static void main(String [] args)
11 {
12   foo(f);                 // prints foo long
13   foo((cap void(int)) f); // prints foo int
14 }
```

Fig. 14. Invocation of an overloaded method with an overloaded operation as an argument.

valid argument (i.e., type compatible). For the invocation on line 12, the selected definition of `foo` is that starting on line 4 because `cap void (long)` is more specific than `cap void (int)`. Next, the most specific definition of `f` that can be passed as an argument to the resolved `foo` is selected. In the invocation on line 12, the `f` operation defined on line 8 is passed as the argument to `foo`. The selection strategy can be modified through the use of explicit casts and, of course, the use of operation capabilities (which are uniquely named). Line 13 in Figure 14 demonstrates the use of an explicit cast to invoke the `foo` method defined on line 1 with the `f` operation defined on line 7 as its argument. The explicit cast is used to “select” the desired `f` operation and to restrict the potential matches for `foo`.

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 `inni` 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 `inni` operation.

4. OPERATIONS AND INHERITANCE

4.1 Java Inheritance Background

A Java class definition can be viewed as consisting of a specification and an implementation (a Java interface consists of only a specification). The specification of a class defines the external interface “exported” by instances of that class. An instance of a class is used by invoking instance methods defined in the specification of the class. The implementation of a class defines the actions taken when a method in the specification is invoked (i.e., the implementation defines the statements that are executed upon invocation).

The actual syntactic definition of a class in Java does not differentiate between the specification and the implementation of the class. Each declaration of a non-abstract method adds the method to the specification of the class and defines the implementation of that method. Abstract methods (as well as interfaces) can be used to declare a specification apart from the implementation, but this requires that

[Pierce 2002] for additional discussion).

the defining class be declared abstract as well. Ultimately, the implementation of the method is defined in conjunction with the specification of some class.

In Java, a new class may be derived from an existing class to create a subclass. Through this derivation, the subclass inherits the specification of its parent class. By default, a subclass also inherits its parent's implementation of the specification. A subclass can extend the inherited specification through the addition of methods. Similarly, a subclass can modify the implementation of its inherited specification by redefining the inherited methods.

4.2 Operation 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 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 `inni` statements attempting to service (or receive from) an operation define its "implementation", though only a single `inni` statement will service a given invocation. As such, any additional `inni` statements "servicing" an inherited operation are added to the set of `inni` statements defining its implementation. 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.

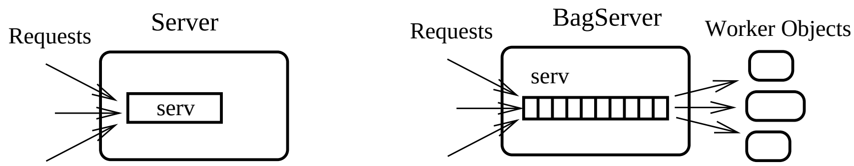
(3) ProcOp → InOp

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.

(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.

Redefinition of a ProcOp to be an InOp can be used to distribute the servicing of the operation's invocations without changing the client. Figure 15 graphically depicts both the original client-server structure, which uses a ProcOp, and the



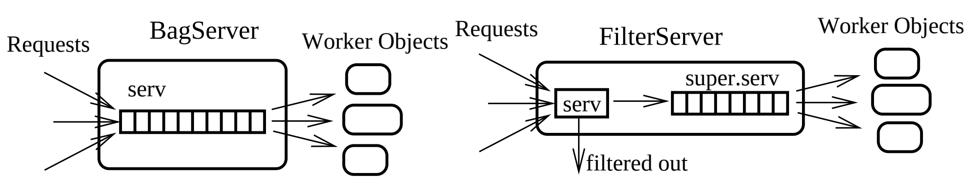
(a) Centralized server. (b) Distributed server.

Fig. 15. Distribution of servicing through redefinition of operation in subclass BagServer.

```
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 Worker {
  // operation declaration
  public op void init(cap int (int) srvr);
  public void init(cap int (int) srvr) {
    while (true) {
      inni int srvr(int i) {
        ... /* compute and return value */ }
    }
  }
}
```

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

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

Fig. 16. Redefining an operation to distribute its implementation.



(a) Distributed server. (b) Filtered distributed server.

Fig. 17. Filtering of invocations through redefinition of operation in subclass FilterServer.

new bag of tasks structure that results from redefining the inherited operation to be an InOp. With the original operation, each invocation results in a new thread being created (at the server) to service the invocation. With the redefined operation, each invocation is handled by an extant Worker object, which was created at program startup and which may be located on a separate host. Figure 16 shows the definition of the subclass `BagServer` and the redefinition of the operation. `BagServer` redefines the operation `serv` to be an InOp, creates Worker objects, and passes a capability for `serv` to the `init` method of each Worker object. Each Worker object repeatedly executes an `inni` statement to service invocations on `serv`'s queue. These Worker objects can be located on an arbitrary set of physical machines as specified by the `remHosts` array.

It is possible to filter the invocations of an operation by redefining an InOp to be a ProcOp. Figure 17 graphically depicts each server configuration. In the first

```
1 public class FilterServer extends BagServer {
2   // redeclaration of operation
3   public op long serv(int i);
4   public long serv(int i) {
5     if (filter(i))
6       return DEFAULT; // or exception
7     else // forward to Worker objects
8       forward super.serv(i);
9   }
10
11  boolean filter(int i) { // simple filter
12    return (i < 0);
13  }
14
15  public FilterServer(vm [] remHosts) {
16    super();
17    int i; // initialize Worker objects
18    for (i = 0; i < remHosts.length; i++) {
19      remote Worker w =
20        new remote Worker() on remHosts[i];
21      // reuse super operation
22      send w.init(super.serv);
23    }
24  }
25 }
```

Fig. 18. Redefining an operation to serialize invocations through a filter.

server configuration, a `BagServer` as shown in Figure 17 (a), each invocation is serviced by an extant Worker object. Redefinition of the operation, as shown in Figure 17 (b), allows for the filtering of invocations in order to reduce the amount of work done by the Worker objects. Figure 18 shows the definition of the subclass `FilterServer` that redefines an InOp to be a ProcOp. `BagServer` defines the `serv` operation as an InOp on line 23 of Figure 16. `FilterServer` redefines the `serv` operation to be a ProcOp on lines 3 – 9.

Each invocation of `serv` 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 invocation to the InOp `serv` defined in the parent class (`BagServer`). Each Worker object repeatedly executes an `inni` statement to service the InOp `serv` defined in the `BagServer` class; this is the operation capability that is passed to the `init` method of the Worker objects on line 22 in Figure 18.

4.3 Redefinition Considerations

JR requires that a subclass explicitly redeclare an operation if the subclass redefines an inherited InOp as a ProcOp or an inherited ProcOp as an InOp. Such a redeclaration is required to statically determine that the operation has been redefined and to reduce the potential for erroneous code. Such a redeclaration is required as a statement of intent and an aid to the compiler for static analysis.

Imagine that an explicit redeclaration were not required to redefine a ProcOp as an InOp. Instead, assume that an operation is implicitly redefined as an InOp if the operation is serviced by an `inni` statement. Full program analysis would be required to statically determine that the operation has been redefined. However, such an analysis is not sufficient when capabilities are used within `inni` statements. This "redefinition" approach is unsatisfactory because all such redefinitions cannot

```
1 class A
2 {
3    public op void foo();
4 }
5
6 class B extends A
7 {
8    public op void foo();
9    public void foo()
10   { ... }
11 }
```

(a) Redefinition of InOp as ProcOp.

```
1 {
2    ... // use of operations
3    A a = new B();
4    B b = new B();
5
6    inni void b.foo() // compile-time error,
7                      // b.foo is a ProcOp
8      { ... }
9    [] void a.foo() // run-time error,
10                   // a refers to a B object
11     { ... }
12 }
```

(b) Erroneous uses.

Fig. 19. Erroneous uses of ProcOps in an `inni` statement.

be discovered at run-time. Furthermore, allowing such a "redefinition" could lead to hard-to-find errors if a programmer accidentally "redefines" the wrong operation in an `inni` statement.

An InOp may be redefined to be a ProcOp if the subclass explicitly redeclares the operation and defines a signature-compatible method. If an `inni` statement attempts to service the operation through a reference to the subclass, then a compile-time error will be raised (see Figure 19 (b), line 6). If an `inni` statement attempts to service the operation through a reference to the superclass, then a run-time error will be generated (see Figure 19 (b), line 9). This explicit redeclaration is required to reduce the potential for accidental redefinitions and to make the change in invocation servicing semantics explicit.

JR currently makes a strict distinction between operations and methods . An inherited operation may not be redefined as a method and an inherited method may not be redefined as an operation. This distinction is required because the concurrency constructs introduced in JR (e.g., the *send* statement) apply only to operations and capabilities and it is desirable to statically check improper use. If a subclass is allowed to redefine an inherited operation as a method, then a *send* to that method will fail at run-time because the method will not have been properly translated to support asynchronous invocation. Moreover, since the set of operations defined in a class directly translates to the interface exported by remote instances of that class (see Section 3.2), allowing a standard method to redefine an operation is equivalent to removing the operation from the remote specification. As such, this restriction is consistent with the behavior of Java because a subclass is not permitted to remove declarations from its inherited specification.

Redefining an inherited method to be an operation is not supported for implementation reasons. An operation is implicitly defined to include the RemoteException exception in its throws clause. This inclusion is to support the use of operations as interfaces to remote objects. As such, redefining the method to be an operation would add an exception to the throws clause of the subclass's "method". Java does not permit the addition of exceptions to the throws clause of redefined methods. Support for such a redefinition would require either modifying the language and library such that all methods throw the RemoteException exception or creating a proxy to handle the RemoteException exceptions thrown by operations. We consider modifying the Java library for this purpose unacceptable. Creating a proxy for each operation reduces the performance of operation invocations and requires that the proxy code handle the communication exceptions in a general manner. Communication exceptions are currently thrown to the user code (through the RemoteException) so that the programmer may handle them appropriately.

4.4 Inheritance Anomaly

In concurrent object-oriented languages, there exists a conflict between inheritance and synchronization constraints that often requires redefinition of inherited methods in order to maintain the integrity of concurrent objects. This conflict has been deemed the *inheritance anomaly* [Matsuoka and Yonezawa 1993]. The complaints regarding the inheritance anomaly are focused on functionally unnecessary redefinition of methods and a violation of encapsulation. The proposals, discussed in [Matsuoka and Yonezawa 1993], that seek to resolve the inheritance anomaly attempt to reduce the amount of redefinition required. Rather than requiring the redefinition of the implementation of a method, many of the proposals allow the redefinition, in some form, of the synchronization constraint. However, the encapsulation argument does not seem to be resolved since the synchronization code in the subclass often takes advantage of knowledge of the synchronization code used in the parent class.

JR inherently suffers from the inheritance anomaly because it allows for the reception of messages within the body of a method, i.e., JR's inni statement appears within executable code. If the synchronization strategy is redefined in a subclass, then some methods may require redefinition in order to change the operations from which they receive. This redefinition is required even if the functionality of the method is to remain the same. The same is true of Java programs that use synchronized blocks (as opposed to synchronizing the entire method) within the bodies of methods (e.g., to emulate semaphores).

We view synchronization, though a cross-cutting concern, as an integral part of the implementation of certain methods; without proper synchronization the "implementation" of a method may be simply incorrect. Consider, as a simple example, the implementation of a hashtable versus that of a thread-safe hashtable (i.e., one that uses synchronization to support simultaneous access by multiple threads). Though the inherent functionality of each is essentially the same, the implementation of the thread-safe hashtable necessarily requires proper synchronization. The implementation of the thread-safe hashtable could certainly be simplified through the reuse of the basic hashtable, but such reuse by means of implementation inheritance would require method redefinition to support synchronization. We feel that the drawbacks of the inheritance anomaly are more than compensated for by the flexibility provided by servicing operations within method bodies and is needed to provide the support argued for in Section 2.1.

JR does, however, introduce difficulties relating to the inheritance anomaly beyond those discussed above. An input statement can service (i.e., receive from) both operations and capabilities. Since these capabilities may refer to operations from different objects, different classes, and even different machines, the task of determining the code affected by a change in synchronization policy requires that the programmer follow the dynamic call graph. This problem can be eliminated by restricting input statements such that only the operations defined in the object (or class) in which (the method containing) the input statement is executing can be serviced. Such a restriction would not prevent the invocation of operations through operation capabilities. This alternative will be explored as the language gains use and matures.

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 interesting parts of 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 which plays a role similar to that of SR's SRX [Andrews and Olsson 1993]. 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. The JR virtual machine then uses RMI to contact JRX and register itself as ready to receive requests. Remote objects are subsequently created by contacting the JR virtual machine directly through RMI.

5.1.1 *Security Implications.* The use of rsh for virtual machine creation requires that systems allow remote execution of programs through a relatively insecure protocol. As such, we are exploring alternatives to allow virtual machine creation without sacrificing machine security. Execution of each virtual machine is still, however, subject to the security policy files for the user at the remote site. Communication between virtual machines is currently implemented using RMI and is not encrypted.

5.2 Remote Objects

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

5.3 Operations and Operation Capabilities

Figure 20 shows the actual inheritance hierarchy of operation classes in JR. This hierarchy is a specialization of the inheritance hierarchy in Figure 5. Each proc operation is implemented as a separate instance of a ProcOp class nested within the class that defined the operation. Thus, 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. This translation is very similar to the common technique of simulating a method reference (not directly representable in Java without using

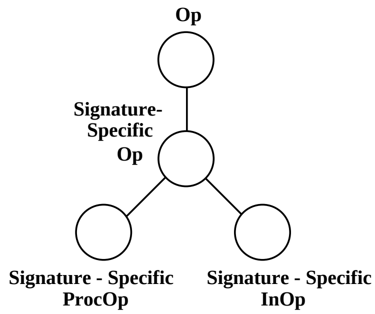


Fig. 20. Actual JR Operation Inheritance Hierarchy.

```
class A
{
   public op void foo(int);
   public void foo(int i)
   { ... }

   public void bar()
     throws java.rmi.RemoteException
   {
      foo(3);
   }
}
```

(a) JR code.

```
class A
{
   ... // implementation of ProcOp foo

   public void bar()
     throws java.rmi.RemoteException
   {
      JRget_op_foo_intTovoid().call(3);
   }
}
```

(b) Generated Java code.

Fig. 21. Translation of the invocation of a ProcOp.

reflection) as an object with a well-defined interface. Figure 21 shows an example of this translation.

An `inni` 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 via access methods used to retrieve the appropriate operation object an example of which can be seen on line 8 of Figure 21 (b).

5.6 Remote Class Loading

Dynamic class loading, as described in the Java RMI specification [Sun Microsystems 1997], 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 separate server to provide file access.

5.7 Termination Detection

The JR run-time system provides support for termination detection in programs that use only JR concurrency constructs. Specifically, the termination detection is not guaranteed for programs that directly use Java's threads, `synchronized` statement (or method attribute), or RMI. This restriction is due to the implementation of the termination detection at the user level in JR's run-time system.

The implementation relies upon instrumenting the generated Java code to track the creation and completion of threads, and the sending and receiving of messages. Java's concurrency constructs are not similarly instrumented as doing so would require modification of the class library or Java's virtual machine (which we are reluctant to do). Automatic termination detection is, however, optional and may be disabled for those programs in which Java's concurrency constructs are to be used.

With termination detection, use of the `done` operation (and explicity "JR.exit(0)") can be removed from Figures 10 and 11. The program would terminate once all of the readers and writers have completed their iterations (and the server blocks waiting for a request).

5.8 JR versus SR

In addition to the object-oriented features discussed in previous sections (which SR does not provide), JR supports all of the concurrent programming features of SR with the exception of the concurrent invocation statement and quantifiers in `in` statements (typically used to specify indices of operation arrays). These remaining features could, with reasonable effort, be added to JR but we have opted not to do so at this point.

6. PERFORMANCE RESULTS

A number of microbenchmarks have been used to study the performance of JR programs against equivalent RMI programs. The performance results demonstrate that remote method invocations in JR incur little overhead compared with equivalent invocations in standard RMI (upon which JR is built). Each of the following experiments was conducted on a cluster of 350 MHz Intel Pentium II workstations connected via a 10 Mbps Ethernet network. All experiments were conducted using

SUN's JDK Version 1.3.1 on Linux 2.4.2-2. For each of the experiments, the presented results are averages over multiple executions of the respective benchmarks.

The first experiment demonstrates the time needed to invoke an empty method. Table I shows the results of repeatedly invoking an empty ProcOp in JR and an empty method in Java. The method takes as an argument a single object that contains an array of a specified number of integers.

Object Size	(*ints*)	1	1k	10k	100k
JR	(μs)	0.3682	0.3685	0.3686	0.3694
Java	(μs)	0.0741	0.0746	0.0746	0.0758
JR time / Java time	(*ratio*)	4.97	4.90	4.94	4.87

Table I. Time to invoke an empty JR ProcOp and an empty Java method in a local object.

As evidenced by this experiment, a JR method invocation takes about five times longer than a standard method invocation. The current implementation of JR requires four additional method invocations to support dynamic dispatch on operations and causal ordering of messages. The overhead to support dynamic dispatch on operations could be reduced through implementation techniques similar to those used for methods (i.e., a virtual method table). Further optimization at the compiler level may reduce the overhead of local operation invocations by eliminating the invocations for causal ordering.

The next experiment, taken from [Izatt et al. 1999], extends the previous experiment to measure invocations of an empty method in a remote object. Table II 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.

Object Size	(*ints*)	1	1k	10k	100k
JR	(*ms*)	1.73	2.28	6.64	48.43
JDK RMI	(*ms*)	1.60	2.19	6.21	47.63
JR time / RMI time	(*ratio*)	1.08	1.04	1.07	1.02

Table II. Time to invoke an empty JR ProcOp and an empty RMI method in a remote object.

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

Table III shows the results of multiple executions of the Readers/Writers program from Section 3 using both JR and RMI. In this experiment, all readers and

[††]The differences are so minor, however, that instances of JR outperforming RMI, due to extraneous network traffic, have been observed.

writers are colocated on the same machine; the server that manages their synchronization resides on a separate machine. As Figure 12 shows, in the JR solution the Readers/Writers server uses `inni` statements to service invocations of the different InOps associated with requesting and releasing read/write access. The RMI solution in this experiment, however, uses a semaphore-like approach in solving the Readers/Writers problem. Therefore, Table III also includes performance results for a roughly equivalent JR semaphore-like solution.

R/W/RI/WI		20/10/3/3	50/15/3/3	100/30/3/5
JR (inni)	(ms)	3555.0	5544.6	14428.0
JR (semaphore)	(ms)	2765.0	4291.6	11692.2
JDK RMI	(ms)	2723.4	3971.6	10686.0
JR (inni) / RMI	($ratio$)	1.31	1.40	1.35
JR (semaphore) / RMI	($ratio$)	1.01	1.08	1.09

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

The JR semaphore-like solution uses JR's semaphore abbreviations. These abbreviations translate into sends and receives on an InOp. As such, this semaphore solution does not perform as well as the RMI solution because of the overhead associated with supporting general message passing. Further optimization of the InOp process fairness code should improve the performance of this solution. It should be noted, however, that the JR semaphore solution did outperform the RMI solution when the RMI server was restarted between each test. Restarting the server between tests reduced the effectiveness of Java's Just-In-Time (JIT) compiler for the RMI solution. The results shown in Table III for the RMI solution are for a persistent server.

The JR solution that uses an `inni` statement did not perform as well as the other two solutions. The performance difference is attributable to the current implementation of the `inni` statement's fairness preserving semantics. Table IV shows the percentage of time spent executing code that pertains to the fairness semantics for the Reader/Writer experiment. As shown in the table, a large percentage of time is spent selecting the invocation to service from the currently pending invocations (e.g., readRequest).

Functionality		20/10/3/3	50/15/3/3	100/30/3/5
Invocation Sort	(%)	0.3	0.7	0.7
Select Invocation	(%)	22.2	23.9	25.5
Remove Invocation	(%)	0.5	1.5	1.3
Lock Acquire (Wait)	(%)	60.9	61.2	62.2

Table IV. JR (inni) Solution: Percentage of total execution time spent executing synchronization code for the Readers/Writers experiment.

Invocation selection takes a large percentage of the total execution time, in this program, because all invocations in an operation are examined until one is found that satisfies the arm's `st` clause. In this program, however, the `st` clause does not reference invocation arguments. Therefore, if the `st` clause is false, all invocations in the operation are examined needlessly. As an optimization for such cases, the `st` clause can be lifted out of the selection loop in the generated code.

Table V compares the performance results of the standard sequential version of the Java Grande Forum *Fourier* Benchmark [Edinburgh Parallel Computing Centre] against distributed versions written in JR and RMI.

Number of coefficients		10000	100000
Sequential Java	(s)	75.17	760.42
JR (1 Server)	(s)	68.01	768.37
JR (2 Servers)	(s)	32.29	379.85
JDK RMI (1 Server)	(s)	68.63	760.00
JDK RMI (2 Servers)	(s)	37.87	379.63
JR / RMI (1 Server)	$(ratio)$	0.99	1.01
JR / RMI (2 Servers)	$(ratio)$	0.85	1.00

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

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 `inni` statement. The RMI version uses threads to concurrently initiate invocations of the remote method and to collect the results.

7. RELATED WORK

7.1 Java Message Service

Java 2 Enterprise Edition provides a message service API, Java Message Service (JMS) [Sun Microsystems 2001], that allows point-to-point and publish/subscribe messaging. This support, however, is directed at enterprise solutions. As such, the message "destinations" exist outside of specific programs and are accessed through the JMS provided classes. Programs are provided, via some means, with the names of message destinations (in the point-to-point case, the name of a message queue). To use the message service, a program must create a connection to a specific destination before it can send or receive messages.

This message service is intended for loosely-coupled, enterprise-level solutions and is quite unwieldy for tightly-coupled concurrent programs. Message destinations are created and maintained using administrative tools and, as such, cannot truly be considered a language-level mechanism for distributed programming.

7.2 CORBA

CORBA's [The Object Management Group] Notification Service (an extension of its Event Service) provides push/pull messaging through event channels. In this general framework, a Supplier may push events to the event channel or wait until the event channel pulls events from the Supplier. Similarly, a Consumer may pull events from the event channel or allow the channel to push events to the Consumer.

CORBA's greatest strengths stem from its language and architecture independence. As such, CORBA is often used for loosely-coupled distributed programs. Unlike the Java Message Service discussed above, CORBA's Notification Service event channels need not be stand-alone programs. As such, the Notification Service can be used in writing tightly-coupled concurrent programs such as those targeted by JR. Even so, the support for asynchronous communication differs greatly in these two solutions.

The Notification Service can support asynchronous method invocation through events pushed to the clients. A client provides support for pushes by implementing the `PushConsumer` interface. As such, the single well-known push method defined in `PushConsumer` is the only method that can be invoked asynchronously, whereas JR allows any operation to be invoked asynchronously. Additionally, a `PullConsumer` can only attempt to pull from a single channel at a time, whereas in JR a client can attempt to receive from multiple (in) operations with a single input statement and, thereby, prioritize messages.

The events in CORBA's Notification Service are generic, meaning that any type of data (that satisfy the constraints placed on general remote invocations in CORBA) can be sent as events. Such flexibility requires that the consuming object essentially unmarshall the data sent as events. To address this problem, CORBA also provides typed events, which require agreed upon interfaces, and structured events, which map the event data to a well-known structure. Moreover, event types can be mixed such that a Supplier can push a generic event while a Consumer attempts to pull a structured event. JR supports messaging through operations that have signatures similar to those of methods. An event type is implicitly agreed upon as the signature of an operation and the run-time system manages the marshalling and unmarshalling of sent data.

CORBA's Notification Service also provides event filtering through filter objects that are registered with either an Admin object associated with the event channel or a Proxy object that sits between the Admin and the Consumer. Filters at a Proxy can filter events for a specific Consumer, whereas filters at an Admin can filter events for a group of Consumers. Filters are specified using a Filter Constraint Language and stored as strings in the running program. These filters provide functionality similar to that provided by synchronization expressions on input statements in JR. However, the filters cannot access variables local to a pulling consumer and they cannot change the order in which events are delivered. Moreover, JR includes support for comparing and selecting from pending messages in multiple operations.

7.3 Java Extensions

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 `inni` statements. Moreover, many of the previous extensions still require the user to manually start remote programs.

Ajents [Izatt et al. 1999] 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. (That is another drawback of the "package" approach described near the beginning of Section 2.)

JavaParty [Philippsen and Zenger 1997] 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 [Nagaratnam et al. 1996], 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 [Raje et al. 1997] 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 [Hartley 1998]. 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 `inni` statement). Support for rendezvous is provided via a blocking request at the "client" and a request/reply pair at the "server".

Both Java/DSM [Yu and Cox 1997] and Charlotte [Baratloo et al. 1996] extend Java to include mechanisms for distributed shared memory. ParaWeb [Brecht et al. 1996], SuperWeb [Alexandrov et al. 1997], and Javelin [Christiansen et al. 1997] seek to exploit the potential for parallel computation using the World Wide Web. Communicating Java Threads [Hilderink et al. 1997] extends the concurrency model of Java by providing communication between threads based on the CSP [Hoare 1978] paradigm. Support for data-driven objects in Java is discussed in [Kalé et al. 1997].

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, [Maassen et al. 1999] and [Nester et al. 1999] discuss more efficient versions of Java's RMI.

Java-SR [Mendis 1997] 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.

In a similar vein, though not related to Java, Wellings et al. [2000] discuss extending Ada 95 to more fully integrate object-oriented programming and Ada's protected objects. This discussion highlights the syntactic and semantic issues involved in making Ada's protected types extensible.

7.4 Concurrent Object-Oriented Languages

Numerous concurrent object-oriented languages have been proposed, e.g., as discussed in a recent survey [Briot et al. 1998]. 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 Actalk [Briot 1989], BAST [Garbinato et al. 1996; Garbinato and Guerraoui 1997], GARF [Guerraoui et al. 1997, Garbinato and Guerraoui 1998], and Simtalk [Bezivin 1987]. 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 [Briot et al. 1998]. 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 remote references 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.

The JR programming language provides a generalization of the input (`inni`) statement that provides greater control over the selection of invocations to service. Specifically, the extended input statement allows selection based on comparisons among the entire set of pending invocations (see [Keen 2002]). JR also extends Java's exception model to handle exceptions that occur during asynchronous communication (see [Keen and Olsson 2002]). This includes exceptions that are thrown by a method or `inni` 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. Even so, our experiments demonstrate that, for the application benchmarks, JR incurs little penalty over equivalent RMI programs. We believe that the ease of programming concurrent applications in JR, over using RMI, outweighs the incurred performance penalty. 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.

The JR programming language has been successfully used in courses on concurrent programming and for small to moderate-sized projects. Such projects range from games to a distributed file system simulation. Future work will explore the implementation of larger concurrent systems.

REFERENCES

Alexandrov, A., Ibel, M., Schauser, K., and Scheiman, C. 1997. Superweb: Towards a global web-based parallel computing infrastructure. In *11th International Parallel Processing Symposium*. 100–106.

Andrews, G. 1991. *Concurrent Programming: Principles and Practice*. Benjamin/Cummings Publishing Company, Inc., Redwood City, CA.

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

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

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

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

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

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

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

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

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

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

Briot, J.-P. 1989. Actalk: A testbed for classifying and designing actor languages in the Smalltalk-80 environment. In *Proceedings ECOOP'89*, S. Cook, Ed. Cambridge University Press, Nottingham, 109–129.

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

Christiansen, B. O., Cappello, P., Ionescu, M. F., Neary, M. O., Schauser, K. E., and Wu, D. 1997. Javelin: Internet-based parallel computing using Java. *Concurrency: Practice and Experience 9,* 11 (Nov.), 1139–1160.

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

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

Garbinato, B., Felber, P., and Guerraoui, R. 1996. Protocol classes for designing reliable distributed environments. In *ECOOP '96—Object-Oriented Programming*, P. Cointe, Ed. Lecture Notes in Computer Science, vol. 1098. Springer, 316–343.

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

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

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

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

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

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

Guerraoui, R., Garbinato, B., and Mazouni, K. R. 1997. Garf: A tool for programming reliable distributed applications. *IEEE Concurrency 5,* 4 (Oct./Dec.), 32–39.

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

Hilderink, G., Broenink, J., Vervoort, W., and Bakkers, A. 1997. Communicating Java Threads. In *WoTUG 20*. 48–76.

Hoare, C. A. R. 1978. Communicating sequential processes. *Communications of the ACM 21,* 8, 666–677.

Izatt, M., Chan, P., and Brecht, T. 1999. Ajents: Towards an environment for parallel, distributed and mobile Java applications. In *ACM 1999 Java Grande Conference*. 15–24.

Kalé, L. V., Bhandarkar, M., and Wilmarth, T. 1997. Design and implementation of parallel Java with global object space. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications*. 235–244.

Keen, A. W. 2002. Integrating concurrency constructs with object-oriented programming languages: A case study. Ph.D. thesis, Department of Computer Science.

Keen, A. W. and Olsson, R. A. 2002. Exception handling during asynchronous method invocation. In *Euro-Par 2002 Parallel Processing*, B. Monien and R. Feldmann, Eds. Number 2400 in Lecture Notes in Computer Science. Springer-Verlag, 656–660.

Liskov, B., Herlihy, M., and Gilbert, L. 1986. 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.

Maassen, J., van Nieuwpoort, R., Veldema, R., Bal, H. E., and Plaat, A. 1999. An efficient implementation of Java's remote method invocation. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 173–182.

Matsuoka, S. and Yonezawa, A. 1993. Analysis of inheritance anomaly in object-oriented concurrent programming languages. In *Research Directions in Concurrent Object-Oriented Programming*. 107–150.

Mendis, C. N. 1997. Java-synchronising resources for concurrent and distributed programming. M.S. thesis, University of Warwick.

Nagaratnam, N., Srinivasan, A., and Lea, D. 1996. Remote objects in Java. In *IASTED International Conference on Networks*.

NESTER, C., PHILIPPSEN, M., AND HAUMACHER, B. 1999. A more efficient RMI for Java. In *ACM 1999 Java Grande Conference*. 152–159.

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

PHILIPPSEN, M. AND ZENGER, M. 1997. JavaParty — transparent remote objects in Java. *Concurrency: Practice and Experience 9,* 11 (Nov.), 1225–1242.

PIERCE, B. C. 2002. *Types and Programming Languages*. The MIT Press, Cambridge, Massachusetts.

RAJE, R., WILLIAMS, J., AND BOYLES, M. 1997. An asynchronous Remote Method Invocation (ARMI) mechanism for Java. *Concurrency: Practice and Experience 9,* 11 (Nov.), 1207–1211.

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

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

SCOTT, M. L. 1991. The Lynx distributed programming language: Motivation, design and experience. *Computer Languages 16,* 3/4, 209–233.

SUN MICROSYSTEMS. 1997. *Java Remote Method Invocation Specification*. Sun Microsystems, Palo Alto, CA.

SUN MICROSYSTEMS. 2001. *Java Message Service Specification*. Sun Microsystems, Palo Alto, CA.

The Object Management Group. *The Common Object Request Broker*. The Object Management Group. `http://www.omg.org/`.

WELLINGS, A. J., JOHNSON, B., SANDEN, B., KIENZLE, J., WOLF, T., AND MICHELL, S. 2000. Integrating object-oriented programming and protected objects in Ada 95. *ACM Transactions on Programming Languages and Systems 22,* 3, 506–539.

YU, W. AND COX, A. 1997. Java/DSM: A platform for heterogeneous computing. *Concurrency: Practice and Experience 9,* 11 (Nov.), 1213–1224.