On Patterns for Practical Fault Tolerant Software in Java

Golden G. Richard III
Shengru Tu

Department of Computer Science
University of New Orleans
New Orleans, LA 70148
Email: {golden, shengru}@cs.uno.edu

Abstract

Fault tolerance is important for both sequential and distributed software, and particularly so for long-running applications. The ability to stop an application and restart it, with minimal lost work, is especially useful. If components of the application can be restarted on arbitrary hosts, so much the better. In this paper, we explore Java's potential to support fault tolerant software design. We note that while there are "deficiencies" in these facilities, a little creativity can still yield solutions that would be quite difficult in other programming environments.

The main contributions of the paper are several preliminary fault tolerant programming design patterns, aimed at easing the burden of application programmers who must write completely portable fault tolerant applications. This class of applications is growing, as Internet-domain software becomes increasingly prevalent. We expect that with further development the proposed patterns will be applicable to a wide range of sequential and parallel applications[*].

1. Introduction

Fault tolerance is important for both sequential and distributed software. By fault tolerant software, we mean the ability of software to handle (primarily) processor failures, as opposed to the ability of software to handle design faults. In particular, the ability to restart application components (or entire applications) from previous states is important for a number of reasons. Often-quoted reasons are power failure—a long running application's life is snuffed out because of a power outage—and hardware failure. But another reason occurs more frequently—the application's user wishes to stop running the application for a period of time, but without losing the results that have been computed so far. This might be necessary to free resources temporarily for another application, or during a system reboot. Support for heterogeneous environments is also highly desirable, given the increasing prevalence of heterogeneous networks of workstations and Internet-domain software.

While considerable theoretical work has been done to support development of fault tolerant software, application programmers interested in writing portable, fault tolerant software without a considerable time investment currently have few practical solutions available. The ideal is fully transparent, portable, minimally restrictive fault tolerance facilities that are easy for application programmers to use. The reality currently falls short. We very briefly survey some representative systems, since the body of work is extremely large. For sequential code, checkpointing facilities, such as those provided by Condor [5] and libckpt [7], allow applications to be checkpointed at arbitrary positions and then restarted, but carry significant restrictions. The most serious restriction is that they are targeted almost exclusively at UNIX platforms. Another restriction is that they are generally not portable even among different flavors of UNIX, since process representation varies considerably. Unfortunately, checkpointing UNIX processes is inherently a very difficult task, since process state is not nicely encapsulated. Another limitation is that frequently-used UNIX facilities such as signals, sockets, pipes, and *fork()/exec()* dynamic process creation are taboo. Programs that meet the requirements for checkpointing emerge rather "sterile".

For distributed code, the problems are even more pronounced. Many application-transparent techniques have been proposed for both Distributed Shared Memory (DSM) and message-passing systems, including checkpointing-based systems [e.g., 4, 12, 14] and systems based on combinations of synchronous and asynchronous checkpointing and message logging [e.g., 3, 8, 9]. Many of the systems have never been fully implemented, likely because some of the fundamental facilities (particularly checkpointing) are themselves very difficult to implement. Applications programmers could use (some) proposed tools for writing fault tolerant software if efficient, portable implementations were

[*] This work is supported in part by LEQSF grants 1998-01-RD-B-18 and 1997-00-RD-A-28.

available. But a grasp of the theory necessary for implementation of the more complex systems is difficult without extensive study of the area. Often researchers who are capable of such implementations must "move on" to other projects.

Some implemented systems to support fault tolerant applications (e.g., CoCheck [10], based on PVM) are finicky and non-portable. Naturally, there are many exceptions—Linda/FT-Linda [14] and Calypso, part of the Milan Project [6], exhibit good performance and have been ported to several platforms. However, their applicability to highly-heterogeneous environments, such as the Internet, remains an issue.

As an aside, we note that proper implementation of many of the application-transparent techniques, both for sequential and distributed code, would be much more feasible with proper operating system support. A major hurdle that implementers face in creating tools for fault tolerant programming is trying to get commonly used operating systems (prevalent versions of Unix, and, even more challenging, non-Unix operating systems like Windows NT) to behave in ways that were not envisioned by the architects. Even so, operating system-independent solutions are better for a broad class of applications, including Internet computing. And completely portable solutions will allow everyone to get more sleep. To this end, we are currently investigating the use of Java for development of fault tolerant software.

Java is a platform-independent language that runs on virtually every commonly used, modern operating system. It provides a rich set of tools for developing both sequential and distributed applications and includes powerful features that should be useful for the development of fault tolerant applications. Moreover, Java features to support fault tolerant programming are available in standard form on every Java platform. With the advent of Just In Time (JIT) compilers for Java, even performance issues, which have been a major Java criticism, have been largely abated. In informal tests, we have measured Java performance very close to that of optimized C/C++ using the Java Performance Pack (JPP), which includes the JIT compiler.

In this paper, we present our initial experiences with Java as a portable solution for developing completely portable fault tolerant programs. We discuss several Java-based fault tolerant programming patterns. These patterns allow development of fault tolerant Java solutions for a large class of sequential and distributed applications. A summary of the rest of the paper follows. In Section 2, we briefly discuss *object serialization*, the most important Java facility for developing fault tolerant software, and motivate our approach. In Section 3, we introduce several design patterns for developing fault tolerant applications in Java. Finally, in Section 4, we present our conclusions and outline future work.

2. Java Features for Fault Tolerant Applications

Perhaps the most powerful Java feature for fault tolerant programming is object serialization, which can read and write entire objects graphs to and from streams. During object serialization, a snapshot of the object, along with every directly and indirectly referenced object, is made and converted into a stream of bytes. This snapshot records the value of every non-transient/non-static variable in the involved objects. The stream of bytes can later be used to recreate an equivalent object graph. Such a powerful facility makes it feasible for application programs to make their own object checkpoints. These checkpoints are readable on any Java platform that supports serialization—they are architecture-independent.

The precondition to write an object to or read an object from a stream is that every object directly or indirectly referred to by this object must implement the *Serializable* interface. The majority of the classes in the Java API are serializable. A simple example, taken from the Javasoft Web site, illustrates writing and reading objects:

```
// Serialize date to file.
FileOutputStream out =
  new FileOutputStream("tmp");
ObjectOutputStream s = new
  ObjectOutputStream(out);
s.writeObject(new Date());
s.flush();

// De-serialize date from file.
FileInputStream in = new
  FileInputStream("tmp");
ObjectInputStream s = new
  ObjectInputStream(in);
Date date = (Date)s.readObject();
```

At first glance, it seems that object serialization is such a powerful facility that fault tolerant programming in Java will be trivial. However, a serious restriction is that the class *Thread* is not serializable. This means that the position(s) of execution within a Java program cannot be captured directly with object serialization. This is unlikely to change in the near future [11].

One approach to the threads problem is to modify a JVM (Java Virtual Machine) to dump the entire state of a Java application on demand. The problem with this approach is that portability, a very attractive feature of Java, is sacrificed, since there are a number of JVM's and each would have to be modified. Another problem is that applications are only resumable on the same type of JVM, which further destroys portability.

Java does not provide sufficient functionality to support completely application-transparent fault tolerance. But with the use of appropriate fault tolerant programming design patterns, Java provides a viable platform for developing high quality, portable, fault tolerant applications. Thus, we propose to implement fault-tolerant software in Java at the application level. Inevitably, this will complicate the applications and require careful design by the programmer. To shield this extra complexity, we suggest a simple set of design patterns for programmers to follow. When these patterns become mature, we can then construct a set of frameworks that shield the application programmer from the Java code necessary for fault tolerance. These patterns are discussed in the next section.

3. Java Design Patterns for Fault Tolerant Programming

We explore three fault tolerant programming design patterns in this section. The first is appropriate for applications with significant in-core state. The second is appropriate for some client/server applications and uses RMI. The third is Linda-like and appropriate for "bag-of-tasks" applications. All Java code was written and tested under JDK 1.1.6. The patterns are not mature, but are surprisingly powerful—especially given the small amount of Java code that is required. Complete examples appear in [16]; due to space limitations, we provide only an example for the first pattern in the Appendix.

3.1 A Pattern for Idempotent, State-ful Applications

We first consider a pattern for idempotent applications with non-trivial, in-core state. The idempotency requirement is necessary because object serialization is used to capture application state, and upon a crash/stop and restart, some portion of the application will be reexecuted. This is a typical requirement for checkpointing-based fault tolerant applications. A large class of tasks such as graphics rendering and information retrieval are idempotent, though important tasks like database updates must be excluded.

The basic approach of the first pattern is to use object serialization to periodically capture the state of the computation. When the application is started, an attempt is made to initialize the state by loading the object instance from an object checkpoint file. If the application is stopped by the user or has experienced a system crash, the application's state can be recovered automatically when the application is restarted.

Since class *Thread* is not serializable, objects containing threads are not serializable by default. This problem can easily be "solved" by the addition of the clause *implements Serializable* to the class's specification. We define no additional methods—this serves only to identify the semantics of being serializable. For example, if a class *MyClass* is derived from the *Thread* class, *MyClass* becomes serializable with the following specification:

```
class MyClass extends Thread
          implements Serializable { }
```

However, "checkpoints" created by serialization of *MyClass* objects are likely to be useless, because all we have done is allow the serialization process to ignore the state of threads in the object. Even more troubling, checkpoints will be taken whenever *MyClass.writeObject()* is called, regardless of the position of execution in any threads in *MyClass*. This occurs whether serialization is induced by a direct call to *writeObject()* or by an implicit call during serialization of an enclosing object. This is likely to result in inconsistent variable values in the checkpointed object(s) after deserialization, since threads must be restarted from the beginning of execution (or not at all).

A simple solution to this problem is to inhibit execution of *writeObject()* for a threaded object until any threads are in a state that is easily recoverable—for example, after the *run()* methods of more deeply nested threaded objects have completed. Java's built-in synchronization methods can be used for this purpose. By tagging the *run()* and *writeObject()* methods of each threaded object *synchronized*, an attempt to *writeObject()* will be inhibited until the *run()* method completes. This is a direct effect of Java's built-in monitor-based synchronization facility. Thus:

```
public synchronized void run()
   { /* compute */ }

public synchronized void
 writeObject(ObjectOutputStream out)
 throws … { …
  out.defaultWriteObject();
}
```

Unfortunately this simple approach is severely limiting, since serialization of an object with nested, threaded objects will not complete until all *run()* methods terminate. In many scenarios, *run()* methods are coded to be non-terminating, or to carry on a very long-running iterative computation. A little more work needs to be done in order to accommodate these situations:

- The programmer must identify points at which *run()* methods leave the object in a consistent state; e.g., after a loop iteration. *wait()* calls inserted at these points will release the object's lock and allow object serialization to proceed.
- The programmer must modify *run()* methods so that they can continue work based on the values of non-static/non-transient class variables (these are saved when the object is serialized) should the *run()* method be started from scratch or a checkpoint.
- *readObject()* should restart any threads, as they will not be automatically restarted upon deserialization.

To illustrate these points, a partial definition of a class that computes a row of pixels, appropriate for a simple graphics application, follows. The timed *wait()* call after each pixel value is computed will allow a blocked serialization attempt to proceed; the one nanosecond wait interval is the shortest that can be specified. The *readObject()* method calls *defaultReadObject()* to deserialize the object, then *this.start()* to restart the *run()* method:

```
class Row extends Thread implements
  Serializable {
  private Color row[];
  private int start=0;
  …
 public synchronized void run() {
  for (int i=start; i < row.length;
    i++) {
    row[i]= /* pixel value */
    start = i;
    // allow serialization
    try { wait(0,1); }
    catch (Exception e) {};
  }
 }

 private synchronized void
   readObject(ObjectInputStream in)
   throws … {
   …
   in.defaultReadObject();
   …
   this.start();
 }
```

A complete example appears in the Appendix.

3.2 A Pattern for Fault Tolerant Client/Server Applications

We have also investigated use of object serialization-based checkpointing under Java's RMI (Remote Method Invocation) facility, in the context of a simple distributed rendering application. The application consists of a dispatcher and an arbitrary number of worker processes, executing on arbitrary computers. The dispatcher is an RMI server and the workers are clients of the dispatcher. This architecture minimizes the amount of RMI setup required. The RMI interface to the dispatcher is:

```
public interface RmtDispatcher
                     extends Remote {
  public Job getJob(Job preferred)
             throws RemoteException;
  public void handin(int row,
                     Image section)
             throws RemoteException;
}
```

The dispatcher maintains a table of complete and incomplete sections of the image being rendered. When the dispatcher has all the sections of the image available, an image file is generated. When a worker become idle, it requests a job from the dispatcher and is assigned a section of the image that has not yet been rendered.

Two levels of fault tolerance are provided in the application. Both the dispatcher and individual workers periodically make checkpoints to avoid lost work. A worker's checkpoint covers the current job only. When a worker crashes and restarts, it can query the dispatcher to validate the local checkpoint. If its previously assigned work has not been completed by another worker, it can resume the interrupted computation. Otherwise, the local checkpoint is ignored and the worker can be assigned a different section of the image. If the server (the dispatcher) is down, a client catches a *RemoteException* exception when it attempts to invoke a method of the server. The exception handling will let the client delay for a period of time (*delayTime*) and look up the remote object from the naming service again until the dispatcher becomes available. A sketch of the worker (the client) class is shown below.

```
 public class Worker extends Thread{
  …
  RmtDispatcher d;
  public Worker(){
    …
    d = (RmtDispatcher)
           Naming.lookup(serverURL);
  }
  public void run(){
    …
    try {
      while (more_rows) {
        Job j = d.getJob(myPrefer);
        …
        d.handin(lineNum, imgSection);
```

```
      }
    } catch (RemoteException e) {
        sleep(delayTime);
        d = (RmtDispatcher)
           Naming.lookup(serverURL);
        this.start();
  } }
  …
}
```

3.3 A Pattern for Fault Tolerant Bag-of-tasks Applications

Finally, we have implemented a simple fault tolerant "object space" server in Java that provides a global, associative object repository similar to the Linda model. The object space server accepts *in*, *rm*, *rd*, and *reg* requests from an arbitrary number of Java clients. The *reg* operation registers a new Java class on the server and must be executed once for each distinct class. *reg* transmits the Java bytecode *.class* file to the server, where it is stored in the server's local filesystem. The *in(obj, name)*, *rm(name)*, and *rd(name)* methods add an object *obj* with an associated *name*, retrieve and delete a matching object, and retrieve a copy of a matching object from the server, respectively. The *name* need not be unique; when multiple objects with the same name exist on the server, *rm* and *rd* retrieve an arbitrary one. Object serialization is used to transfer objects between clients and the object space server. A number of coarse-grained parallel/distributed algorithms, including many rendering algorithms for computer graphics, are naturals for this pattern.

Using the object server makes coding simple "bag of tasks" algorithms very straightforward. The abstract class *Work* has a single abstract method *exec()*. The programmer writes a new class definition (e.g., *MyWork*) that extends the *Work* class by defining fields which contain a description of a single task and the result of the computation, and by implementing an appropriate *exec()* to carry out the computation. A master task creates *MyWork* instances, and sends them to the server with name "task". Default clients running on a number of other machines repeatedly request "task" objects, execute the *exec()* methods, and then rename the objects "done" and send them back to the server. To retrieve the results of the bag-of-tasks computation, the master task requests the appropriate number of "done" objects.

The standard Linda model for bag-of-tasks applications is not adequate for fault tolerant applications, and several researchers have enhanced the model to support fault tolerance [e.g., 2, 14]. We extended the basic object space server in several ways to support fault tolerant Java applications. First, the server can be replicated on an arbitrary number of machines. A classic tradeoff develops between resilience of the object space and loss of performance. A study of the effects on performance is underway, but certainly the overhead of replicated servers will depend heavily on the traffic seen by the primary object server. Second, the object space can optionally be made persistent, so the object space is mirrored on stable storage. This is easily done using checkpointing based on object serialization, as discussed in the previous section.

However, making the server resilient to failure is not sufficient to render "bag-of-tasks" applications fault tolerant. If a process retrieves an object from the object server then immediately fails, the partial computation corresponding to the retrieved object will never be performed, likely resulting in an application deadlock. We tried several mechanisms for handling client failures, but currently support a single straightforward one. The object space servers are multithreaded, with a single thread devoted to each live client. As a client executes *rm* requests, its dedicated thread in the server logs the objects that have been removed from the object space. Death of the client results in a broken TCP/IP socket connection, which is detected by the thread. When this occurs, the logged objects are returned to the space. A thread's object log is cleared when the client executes an *in* request.

The current implementation is unable to handle overlapping failures of the clients and server and lacks sophisticated facilities such as transaction support. However, it is implemented in a mere 1100 lines of Java. This clearly shows Java's potential for developing fault tolerant applications. A version that handles overlapping failures is in progress.

4. Conclusion

In this paper we have explored the use of three simple design patterns for Java and discussed our initial experiences with these patterns. All the patterns use object serialization, which is the ultimate enabler for application-oriented fault tolerance in Java. We have not yet formalized them because much more research is needed to evaluate their use. Overall, we believe Java is a promising tool for the development of portable, fault tolerant sequential and distributed applications, especially applications for the Internet domain.

We are striving to provide a middle ground between the ideal of fault tolerance that is completely transparent and fault tolerance issues which must be dealt with "from scratch" (and with great pain!), by the application programmer [1]. The "patterns" we have discussed are applicable for a wide range of sequential and distributed applications, but are not fully mature. Future work includes investigating additional patterns, formalization, and performance studies. We also need to further investigate use of the patterns with highly interactive applications. Ultimately, we hope to provide sophisticated software frameworks that can further reduce the labor expended by the applications programmer.

Acknowledgments

Thanks to Abdul Altalhi for implementation of the object space software and to Michael Ledet for helpful discussions.

References

[1] F. Adelstein, G. G. Richard III, L. Schweibert, R. Parent, M. Singhal, "A Distributed Graphics Library System," *Software Practice and Experience*, vol. 24, no. 4, April 1994.

[2] S. Cannon, D. Dunn, "A High-level Model for the Development of Fault Tolerant Parallel and Distributed Systems," University of Utah Computer Science TR # A0192, August 1992.

[3] S. Kanthadai, J. Welch, "Implementation of Recoverable Distributed Shared Memory by Logging Writes," *Proc. of the 16th International Conference on Distributed Computing Systems*, May 1996.

[4] R. Koo, S. Toueg, "Checkpointing and Rollback Recovery for Distributed Systems," *IEEE Transactions on Software Engineering*, January 1987.

[5] M. Litzkow, T. Tannenbaum, J. Basney, M. Livny, "Checkpoint and Migration of UNIX Processes in the Condor Distributed Processing System," University of Wisconsin-Madison Computer Science TR # 1346, April 1997.

[6] D. Mclaughlin, S. Sardesai, P. Dasgupta, "Calypso NT: Reliable, Efficient Parallel Processing on Windows NT Networks," cs.nyu.edu/milan/milan/index.html.

[7] J. Plank, K. Li, "Libckpt: Transparent Checkpointing under Unix," *Proc. of the USENIX Winter Technical Conference*, January 1995.

[8] G. G. Richard III, M. Singhal, "Complete Process Recovery: Using Vector Time to Handle Multiple Failures in Distributed Systems," *IEEE Concurrency*, Summer 1997.

[9] G. G. Richard III, M. Singhal, "Using Logging and Asynchronous Checkpointing to Implement Recoverable Distributed Shared Memory," *Proc. of the 12th IEEE Symposium on Reliable Distributed Systems*, October 1993.

[10] G. Stellner, "Resource Management and Checkpointing for PVM," *Proc. of the 2nd European PVM Users Group Meeting*, 1995.

[11] Thread Serialization FAQ, URL: www.javasoft.com.

[12] Y-M. Wang, W. K. Fuchs, "Lazy Checkpoint Coordination for Bounding Rollback Propagation," *Proc. of the 12th Symposium on Reliable Distributed Systems*, October 1993.

[13] D. Bakken, R. Schlichting, "Supporting Fault-Tolerant Parallel Programming in Linda," IEEE *Transactions on Parallel and Distributed Systems*, March 1995.

[14] K. Wu, W. Fuchs, "Recoverable Distributed Shared Virtual Memory," *IEEE Transactions on Computers*, April 1990.

[15] J. Kiniry, D. Zimmerman, "A Hands-On Look at Java Mobile Agents", *IEEE Internet Computing*, July/August 1997.

[16] Fault Tolerant Programming in Java, URL: www.cs.uno.edu/~golden/FTJava/index.html.

Appendix

```
/* example of first pattern */
import java.lang.*;
import java.io.*;
import java.awt.*;
import java.util.*;
public class Render {
  public static void main(String[] args) {
    if (args.length != 3) {
      System.out.println("Usage: Render <size>
        <imgfilename> <cpfilename>");
      System.exit(-1);
    }
    RenderingTh t = new RenderingTh(args[2],
      args[1], Integer.parseInt(args[0]));
    t.start();
  }
}
public class RenderingTh extends Thread {
  int rownum;
  int size;
  String ckFile;
  String imgFile;
  GRow[] gra;
  public RenderingTh(String ckFile,
    String imgFile, int size) {
     this.ckFile = ckFile;
     this.imgFile = imgFile;
     this.size = size;
     gra = new GRow[size];
  }
  public void run() {
    FileOutputStream fos=null;
    System.out.println("Initializing, writing
        PPM header.");
    try {
      fos = new FileOutputStream(imgFile);
      fos.write(new
        String("P6\n").getBytes());
      fos.write(new String(size +" "+size+
        "\n").getBytes());
      fos.write(new
        String("255\n").getBytes());
    }
    catch (Exception ioe) {
      System.out.println("Error writing PPM
        header.");
      System.exit(-1);
    }
    try {
      ObjectInputStream in = new
        ObjectInputStream(new
          FileInputStream(ckFile));
      rownum = in.readInt();
      for (int i = 0; i < rownum; i++) {
        gra[i] = (GRow)in.readObject();
      }
      System.out.println("Checkpoint allows
        recovery up to row " + rownum);
    }
    catch(Exception rcpe) {
      rownum = 0;
      System.out.println("No valid checkpoint,
        starting from row 0.");
```

```java
  }
  for (int k = rownum; k < size; k++) {
    gra[k] = new GRow(k, size);
    gra[k].start();
    if (((k+1) % 10) == 0) {
    try {
      System.out.println("Writing new
        checkpoint to temporary file.");
      ObjectOutputStream out =
        new ObjectOutputStream(new
          FileOutputStream("$$CHECKPOINT$$"));
      out.writeInt(k);
      for (int j = 0; j < k; j++) {
        out.writeObject(gra[j]);
      }
      out.flush();
      out.close();
      File newCP = new File(ckFile);
      System.out.println("Delete of old
        checkpoint " + (newCP.delete() ?
        "succeeded." : "failed."));
      File oldCP = new File("$$CHECKPOINT$$");
      System.out.println("Rename of new
        checkpoint file " +
        (oldCP.renameTo(newCP) ?
         "succeeded." : "failed."));
      System.out.println("Checkpoint " +
        ((k+1)/10) + " complete.");
    }
    catch (Exception wcpe) {
      System.out.println("Failed to make
        checkpoint " + ((k+1)/10) + ".");
    }
  }
}
System.out.println("Writing image.");
for (int k=0; k < size; k++) {
  System.out.println("Writing pixels for row "
    +k+".");
  gra[k].writePixels(fos);
}
System.out.println("DONE.");
try {
    fos.close();
}
catch (Exception ce) {
  System.out.println("Error closing PPM
      file.");
      System.exit(-1);
    }
  }
}
class GRow extends Thread implements
  Serializable {
  private int id;  // our row #
  private Color row[];  // row of pixels
  private boolean complete=false;
  private int start=0; // start position
  private boolean paused=false;
  public GRow(int id, int len) {
    this.id = id;
    row = new Color[len];
  }
  private void render() {
    for (int i=start; i < row.length; i++) {
      row[i]=new Color(i % 256, id % 256,
        (i + id) % 256);
      start = i;
      // allow object serialization
      try {
        paused=true;
        wait(0,1);
        paused=false;
      }
      catch (Exception e) {};
    }
  }
  public synchronized void run() {
    System.out.println("Beginning row " +
      id + ".");
    render();
    System.out.println("Row " + id +
      " complete.");
    complete=true;
    notifyAll();
  }
  public synchronized void
    writePixels(FileOutputStream fos) {
    while (! complete) {
      try { wait(); }
      catch (InterruptedException ie) { }
    }
    System.out.println("Writing pixels for row
      " + id + ".");
    for (int i=0; i < row.length; i++) {
      try {
        fos.write((byte)row[i].getRed());
        fos.write((byte)row[i].getBlue());
        fos.write((byte)row[i].getGreen());
      }
      catch (Exception e) {
        System.out.println("Error writing
          pixels for row "+id+".");
      }
    }
  }
  private synchronized void
    writeObject(ObjectOutputStream out)
      throws IOException,
      ClassNotFoundException {
    if (paused) {
      System.out.println("Row "+id+"
      interrupted at position "+start+".");
    }
    try {
        out.defaultWriteObject();
    }
    catch(Exception e){
      System.out.println(id +": writeObject
        failed.");
    }
  }
  private synchronized void
    readObject(ObjectInputStream in)
      throws IOException,
             ClassNotFoundException {
    try {
      in.defaultReadObject();
      System.out.println("Row " + id + " will
        resume at position " + start + ".");
      this.start();
    }
    catch(Exception e){
      System.out.println(id +
            ": readObject failed.");
    }
  }
}
```