Proceedings of the 1998 Winter Simulation Conference
D.J. Medeiros, E.F. Watson, J.S. Carson and M.S. Manivannan, eds.

THREE PHASE SIMULATION IN JAVA

M. Pidd and R.A. Cassel

Department of Management Science
Lancaster University
Lancaster LA1 4YX, U.K.

ABSTRACT

Recent years have seen great interest in the use of Java as a language for developing computer simulations and in the development of methods that utilise the inherently distributed and co-operative nature of the world-wide web. This paper builds on both of these themes and discusses the development of three phase simulation models in Java. The first stage of the work was to use Java simply as a language for model development, much as any other language might be used. The second phase was to use the RMI functions of JDK 1.1 to develop client:server approaches to simulation modelling. Both of these stages are discussed here, with an indication of the problems that were faced and overcome.

1 BACKGROUND

Since its proposal and development by Sun MicroSystems, Java has been accepted as a language for general purpose use as well as one that is suitable for distributed computing on the Internet. Its attractive features are well-known and include the following.

1. Fully object oriented: meaning that it requires the user to program in terms of:
   - class mechanisms: variable types are declared as objects which may inherit part of their definition from previously declared object types.
   - encapsulation: an extension of program modularisation in which data and the functions which change that data are kept together in the same place.
   - polymorphism: in which the same instruction may be implemented quite differently by members of different classes.

   As discussed in more detail in Pidd (1995).
1. Extensive class libraries (known as packages): such as *awt* (for user interface design), *net* (for network based applications, *util* (for utility functions, such as hash tables) and *rmi* (for remote method invocation). These packages free the programmer from the need to re-invent the wheel and may be extended for particular applications.
2. Syntax based on C++: which was itself based on C, meaning that many programmers are already familiar with its keywords. However, as with spoken languages, there are 'false friends' in which a C++ concept means something rather different in Java.
3. Supports multi-threaded applications: which enables a single server to easily support a number of clients accessing the same software simultaneously in an economical way.
4. High portability: through the concept of a Java virtual machine (jvm), in which tokenised code may be run on any computer that can execute this code through an interpreter known as the jvm.

Given the now widespread notion that the network is the computer it clearly makes sense to investigate the use of Java for discrete simulation. Others have been engaged in this same quest. For example Buss and Stork (1996) developed a simple simulation system known as SimKit; Healy and Kilgore (1997) report on the development of Silk, a Java-based process simulation language; Kreutzer et al (1997) use Java to demonstrate a layered approach to simulation software development; Ferscha and Richter (1997) discuss aspects of Java use to develop conservative distributed simulation methods; Page et al (1997) focus on web-based simulation using Java's remote method invocation.

This paper describes the gradual development of a discrete simulation library using the three phase approach suggested by Tocher (1963) and discussed in detail in Pidd (1995 and 1998). In this approach, a simulation executive performs a repeated cycle of three tasks which allow it to simulate parallelism, whilst avoiding deadlock, as follows.

- A phase: the executive finds the next event, and advances the clock to the time in which this event is due;

- B phase: executes all the B activities (the direct consequences of the scheduled events). which are due at that time;
- C phase: the executive tries to execute all of the C activities (any actions whose start depends on resources and entities whose states may have changed in the B phase).

These three phases are repeated until the clock reaches the end of the simulation.

2 WRITING SIMPLE SIMULATIONS IN JAVA

It is remarkably easy to write discrete simulation applications in Java once familiar with the syntax of the language and with the ideas of object orientation. Thus, the first phase of this project was the development of straightforward three phase simulation libraries that execute much as would a library written in any other programming language. A number of these libraries were developed, each with different features but with many aspects in common.

These implementations define an abstract class, usually known as *GEntity*, which is used to create a dynamic record of the current states and other information about entities within the simulation. For a particular simulation, the user creates a separate class for each entity class in the simulation, these classes being specialisations (thus, descendents) of the abstract *GEntity class.* Thus, for example, any instance of a simulation entity class may be scheduled for a future event or released, using the *Schedule*() and *Release*() methods of this *GEntity* class.

The entity records themselves may be handled in many different ways, but one very convenient approach is to collect them into a *vector* as provided by the Java *util* package. A *vector* is a growable array - that is, its size need not be declared in advance. Thus entity records may be added to the *Details* vector whenever new entities are created and may be removed from there (with care) when entities are destroyed. The *vector* provides direct access, thus enabling fast entity scheduling. An *Executive* class controls the *Details vector* and uses it to execute the three phase cycle, discussed above, in the main simulation loop.

Countable resources, that is, simulation objects that need not be distinguished individually and separately, are conveniently represented in a *Resource* class. Thus simulation resources are represented as classes of type *Resource* to which members may be added, deleted, occupied and freed as the simulation proceeds.

Java comes with the *Random* package as a subset of *Java.util* and this provides simple random number generation and a few routines for random variate generate. This can be easily extended into a *Sample* class to provide whatever sampling routines are needed in the simulation.

If required, a further *Trace* class may be added to enable the provision of traces and run-time reports as text files for debugging.

The above elements are strictly machine independent. The simulation can remain machine independent but with a GUI by the use of *Java.awt*, the abstract windowing tool-kit. This provides support for a number of graphical features that should run on any computer with a *jvm*. Further refinements may be made to the GUI by using application development tools, such as the Borland JBuilder or Microsoft J++, but this may restrict the use of the software to particular hardware and software configurations. Two simulations that illustrate this straightforward approach to three phase simulations in Java may be downloaded from the following URL:

//www.lancs.ac.uk/staff/smamp/MPCSMS4.html

3 THE JUST SYSTEM

The JUST (Java Ubiquitous Simulation Tool) system was developed for simulations to be run in a distributed client-server mode. Thus, the simulation modeller would define her simulation entities, resources, activities etc.. and would be the client that communicates with the simulation executive that runs on a remote server. This means that, once the user has finished creating the model, she does not have to upload the model to the server, nor has to download the simulation executive to her computer. The same executive can then be used to run many simulations, simultaneously if required.

Some class files of the JUST system have to reside in both computers. Classes like *GEntity*, *SimList* and *BEvent* have to be in the client as well as in the server, because both the executive and the model have to recognise these classes. Other classes as Queue and Resource need only be in the client, since the executive never manipulates them. However, since these classes are part of the library and are ready to be used by the user, they are stored in the server so that the user can download them before use.

The classes that stay in just one computer are those directly related to the executive (on the server side) and those strictly related to the model (on the client side). On the server side, the main class is the *Manager*. The *Manager* is responsible for running the three phase executive. It maintains an event list, schedules the next events, runs the A, B and C phases and keeps the clock(s).

On the client side, there are a few important classes: *InputEvents*, *Simulation*, *CActivities* and the new entities created by the user. The *InputEvents* class creates a window for the user to enter the parameters of the simulation. The *Simulation* class is responsible for creating the instances of the basic classes that are going to be used by the model. These basic classes are mainly the queues and the resources of the simulation. The *CActivities* class contains all the methods that describe the C activities of the

model. Finally, the simulation entities are sub-classes of the *GEntity* class that implement the B activities of the model.

Some of these classes located in just one of the computers will be called by the other computer using Remote Method Invocation. Therefore, they have to implement an interface to declare the methods to be called remotely. Though the classes themselves need not be downloaded or uploaded, the interfaces that they implement must be, so that the computer which calls the remote methods can understand them.

4 IMPLEMENTING THE JUST SYSTEM

4.1 Multi-threading

To support multiple simultaneous use, a distributed simulation system needs to be multi-threaded, otherwise a user may have to wait until the simulation of another is finished before she can run her own simulation. The use of threads allows more than one simulation to be run at the same time by dividing a computer program into semi-independent threads (or lightweight processes) that may be separately executed.

All versions of the Java Development Kit (JDK) provide a built-in *Thread* class to start and stop threads and to set priorities amongst competing threads. Any other class can extend this class and may thus be run as a thread. However, Java does not permit multiple inheritance, which means that no user defined class can extend the *Thread* class whilst also extending another class. As a way round this problem, Java provides the *Runnable* interface which permits any class to provide the body of a thread, thus permitting a class to extend another whilst being implemented as a thread.

In the JUSTsystem, the executive of the system is run as a thread. Hence, the *Manager* class implements the *Runnable* interface. This allows the *Manager* class to extend the *UnicastRemoteObject* class, which is necessary for Remote Method Invocation.

All threaded instances of the *Manager* class share the same priority, thus, if the instance with control of the processor relinquishes this control, this instance will go to the end of the queue and wait until its turn comes again. However, to make sure that all the instances of Manager are running “concurrently”, a call of the *yield*() method was introduced after each of the A, B and C phases. This means that, after each one of the three phases, the object with control of the processor will always yield to the next object in the queue.

Two further controls over thread execution are given to the user, who may pause the execution of the threads and/or resume it. Any time the user pauses her simulation, a flag is set on the *Manager* and the simulation is paused as soon as the in-progress A, B or C phase is complete. This avoids problems that may occur were the simulation to be paused during any of the three phases.

4.2 Reflection

The Reflection API is provided in JDK 1.1, though not in JDK 1.0, and it provides methods to return the fields, methods and constructors defined by a class. This makes classes, in effect, self-aware and enables a program to check the members of a class without knowing the class type in advance. Even more usefully, class methods and constructors can be invoked at run-time rather than during compilation, thus the server need not know the methods of the client in advance. Reflection provides a way around two problems that were present in JDK 1.0.

The first is that, for security reasons, Java does not have pointers and does not allow pointers arithmetic. This is mainly to avoid the user making serious mistakes when dealing with the memory. However, this can bring some problems when programming a three-phase simulation library, because it means that Java does not allow pointers to functions (or methods). When an entity is committed to a future event in a simulation, one parameter that must be passed when the event is scheduled, is something to identify the event. In C or C++ this is easily achieved via a function pointer. The executive, which handles the scheduling, thus need not know the identity of the function at compile time. The lack of pointers meant that this had to be 'programmed round' in JDK 1.0. However, the Reflection API of JDK 1.1 allows the executive to invoke methods that it did not know existed. Therefore, it is possible to pass the name of a method, find it in a particular class, and invoke it.

The second use for the Reflection API in the JUST system solves a problem about object oriented, three phase simulation discussed on Pidd (1995). A three phase simulation uses two types of activity as its building block and these are known as As and Bs (sometimes as A and B activities, sometimes as A and B events). Each B may be directly scheduled on an event calendar and occurs as a result of an event triggered by an entity (e.g. end of service in a queuing system). Each C is a possible contender for action once the Bs have been completed at a clock time (for example, a service may start if the server is now free and a customer is waiting). Bs depend only on the state of the entity that triggers its execution, whereas Cs are inherently co-operative - that is, they may involve several classes of entity. This is why the three phase approach is so effective in the simulation of highly interactive systems.

It is clear, therefore, that Bs should be methods that belong to the classes representing the entities that trigger their execution. The problem is that in an object oriented simulation, if entity classes descend from an abstract *GEntity* class, then this abstract class must contain methods to be overloaded in the 'proper' entity classes.

Thus, at worst, the system provider must provide a whole set of dummy Bs to be overloaded by the application. The Reflection API tackles this head on by allowing the *Manager* to determine the B that is due to be executed. In this sense, JDK 1.1 is an advance on C++.

It is, however, still the case that the Cs cannot belong to any particular entity class unless the simulation is implemented with a single active entity and the rest passive when a C is executed. Thus, in the JUST system, Cs are represented by a separate *CActivities* class.

4.3 Remote Method Invocation

If the simulation executive is to sit on one computer and the application classes on another, then the simulation system must provide some way in which the different classes may communicate with one another. The Java Remote Method Invocation (RMI) framework of JDK 1.1 provides the layers necessary for Java objects to communicate with each other using normal method calls, even if the objects are running in virtual machines on opposite sides of the world (Hughes et al (1997)). This enables the development of client-server applications, since the programmer need not worry about complex communication protocols between the applications. Thus RMI is at the heart of the JUST system.

Anecdotal evidence suggests that many people have found Java RMI rather difficult to implement, though as more literature appears, this is easing things somewhat. The JUST system makes use of the *java.rmi.Remote* interface and is implemented in the following classes which extend the *java.rmi.server.Unicast.RemoteObject* class.

- On the server side: *Manager* and *GEntity*.
- On the client side: *GEntity* and *CActivities*.

The actual entity classes of the simulation application also implement the *java.rmi.Remote* interface but are direct descendent classes (extensions) of the *GEntity* class.

In addition, the basic simulation classes to represent objects such as countable resources, queues and lists implements the *java.io.Serializable* interface so as to enable them to be passed as parameters to remote methods.

5 THE OPERATION OF THE JUST SYSTEM

1. Start the server. When this starts, it calls the *Manager.bindManager* method, which starts the server's registry, and binds the *Manager* to it. After that, the server waits until a client calls it.
2. When a client is started, the *Simulation* class locates the server, looks up the *Manager* class registered in the server's registry, starts the client's registry, and binds the *CActivities* to the client's registry.
3. However, the entities are not bound to the registry by the *Simulation* class. Each time a new instance of an entity is created, the constructor of the *GEntity* class binds it to the registry.
4. Once this stage is reached the client side already knows where the *Manager* is, and if it needs a method of the *Manager* to be executed it merely calls this method as if it were local. On the server side it is similar, though with some discrepancies. Since the executive does not know beforehand what will be the next event and, therefore, does not know which entity will execute that activity, it must use the *Reflection* API, as explained earlier.

6 WHAT WERE THE PROBLEMS FACED?

When we started work on the JUST system, literature discussing RMI was rather slight. In addition, the general Java API documentation sometimes is confusing. Thus we learned about RMI by trying things out - tedious, but fruitful in the end! Some problems that we faced were as follows, and it is amazing how obvious the answers seem now!

QUESTION: Why doesn't the library run when both the client and the server are implemented on the same physical computer? ANSWER: Because, even when the client and the server reside in the same computer, it needs a network connection.

QUESTION: How to make the system start the computer's registry? ANSWER: The JDK has a class called *LocateRegistry* in the *java.rmi.registry* library with a method called *createRegistry*.

QUESTION: Which classes can stay in the client and which of them must go to the server? ANSWER: This depends on the application.

QUESTION: Do stubs and skeletons (see Page et al (1997) for a clear description of these) stay in the client or in the server? ANSWER: The stubs stay where the call for the remote method originates and the skeletons stay where the remote objects are located.

ACKNOWLEDGEMENTS

Ricardo A. Cassel's research is funded by the Brazilian Government through CAPES - Fundacao Coordenacao de Aperfeicoamento de Pessoal de Nivel Superior.

REFERENCES

Buss A.H. and Stork K.A. 1996. Discrete event simulation and the world-wide web using Java. In *Proceedings of the 1996 Winter Simulation Conference*. ed J.M. Charnes, D.J. Morrice, D.T. Brunner and J.J. Swain, 780-785, Coronado, CA, 8-11 December.

Ferscha A. and Richter M. 1997. Java based co-operative distributed simulation. In *Proceedings of the 1997 Winter Simulation Conference*. ed. S. Andradôttir, K.J. Healy, D.H. Withers & B.L. Nelson, 381-388, Atlanta, GA, 7-10 December.

Healy K.J. and Kilgore R.A. 1997. Silk™: a Java-based process simulation language. In *Proceedings of the 1997 Winter Simulation Conference*. ed. S. Andradôttir, K.J. Healy, D.H. Withers & B.L. Nelson, 475-482, Atlanta, GA, 7-10 December.

Hughes M., Hughes C., Shoffner M. and Winslow M. 1997 *Java network programming.* Manning Publications Co, Greenwich, CT.

Kreutzer W., Hopkins J and van Mierlo M. 1997. SimJAVA - a framework for modeling queueing networks in Java. In *Proceedings of the 1997 Winter Simulation Conference*. ed. S. Andradôttir, K.J. Healy, D.H. Withers & B.L. Nelson, 483-488, Atlanta, GA, 7-10 December.

Page E.H., Moose R.L. Jnr and Griffin S.P. 1997. Web-based simulation in SimJava using remote method invocation. In *Proceedings of the 1997 Winter Simulation Conference*. ed. S. Andradôttir, K.J. Healy, D.H. Withers & B.L. Nelson, 468-474, Atlanta, GA, 7-10 December.

Pidd M. 1995. Object orientation, discrete simulation and the three-phase approach. *Journal of the Operational Research Society* 46: 362-374

Pidd M. 1998. *Computer simulation in management science*. (4th ed) John Wiley & Sons Ltd, Chichester.

Tocher K.D. 1963. *The art of simulation*. English Universities Press, London.

AUTHOR BIOGRAPHIES

MIKE PIDD is Head of the Department of Management Science and Professor of Management Science at Lancaster University in the UK. He is best known for two books, *Computer simulation in management science* (now in its fourth edition) and *Tools for thinking: modelling in management science*; both published by John Wiley. He is active in researching simulation methods related to modularity and object orientation. He acts as consultant to a number of private and public sector organisations in Europe.

RICARDO A. CASSEL is a Ph.D. student in the Department of Management Science at Lancaster University, whose research focuses on distributed simulation in Java on the world-wide web. He graduated in electrical and electronic engineering from the Universidade Federal do Rio Grande do Sul, Brazil, and has experience of simulation work in logistics and manufacturing systems.