USING JAVA TO DEVELOP DISCRETE EVENT SIMULATIONS

M. Pidd & R.A. Cassel
Department of Management Science
Lancaster University
Lancaster LA1 4YX
UK

ABSTRACT

The rapid development of the Internet and the introduction of Java as a programming language provide new possibilities for discrete event simulations. This paper surveys some of these developments and discusses the aspects of Java that make it attractive for developing such simulations. It then describes the development of two simulation systems in Java, one a straightforward application of the three-phase approach and the other a system that supports distributed client-server applications. The limitations and advantages of using Java in this way are discussed.

KEYWORDS

SIMULATION, COMPUTERS

September 1998, Revised March 1999

USING JAVA TO DEVELOP DISCRETE EVENT SIMULATIONS

BACKGROUND

In operational research, one of the most commonly used methods is discrete simulation[1]. Anecdotal evidence suggests that most such simulations are carried out using computer packages of the type described as Visual Interactive Modelling Systems[2] (VIMS), or Simulators[3]. These systems enable relatively unskilled users to develop useful simulations in a short period of time without the need for detailed computer programming. However, even these systems were actually programmed by developers skilled in languages such as C, C++ or Pascal. In addition, there remain systems that cannot be sensibly simulated in this way, either because the application logic is too detailed or obscure for the simulators, or for other reasons - such as the need to run an extremely fast or large simulation. Thus, there is still a need to develop experimental systems from scratch in general-purpose programming languages - if only to show proof of concept.

Java - the language

Java is a relatively recent programming language, commonly associated with the world-wide-web. It shares many features with other modern programming languages and may thus be used to develop discrete event simulations and other applications. Java was developed around 1991 by James Gosling within Sun Microsystems and was originally known as Oak. It was intended as a small language (that is, it would be undemanding in terms of resources) for use in small portable electronic devices. The idea was to develop a language that could be used to make these intelligent by programming them. Its renaming as Java is already a modern legend, happening when Sun developers discovered that there already was a language called Oak and so they retired to a local coffee house to brew a new name.

1995 saw the public announcement of Java by Sun Microsystems and, since then, its growth in popularity has been rapid. The explosion of the world-wide-web, prompted by the work of Tim Berners-Lee (now the leader of W3C, the World Wide Web Consortium[4]) at CERN, led to the rapid adoption of Java as a language well-suited to the creation of web pages with dynamic content. For more discussion of its

suitability for web pages with dynamic content, see books such as Flanagan[5] and Deitel and Deitel[6].

Java has many features that make it an attractive programming languages and it also has some less endearing aspects. Its attractive features, from a discrete event simulation perspective, are as follows.

1. It is fully object oriented: thus all types created by a programmer are defined as classes. A class is an encapsulated set of methods and data, with only the methods defined for the class being able to manipulate the data. Because the data is encapsulated it is much more difficult for unanticipated effects to occur when a program is enhanced. When a programmer creates a variable, it is an instance of a class and is usually known as an object. Object orientation first appeared in SIMULA[7], a discrete simulation system, and was popularised in SmallTalk. Pidd[8] provides an extensive discussion of the advantages of object orientation in discrete simulation. With careful design, an object oriented approach enables a programmer to develop new classes that inherit features from existing classes, thus saving program development time. It also means that class libraries can be developed, from which pre-defined classes can be used in an application.

2. Extensive packages: a package is a class library which provides useful classes that may be used directly or extended. Some of these classes come with the Java language itself and others are developed by third parties. Examples of in-built classes that are useful for discrete event simulation include:
   - *awt* (the abstract windowing toolkit), which provides classes for developing graphical user interfaces that are independent of the operating system in use.
   - *net*, which contains classes that support network applications
   - *util*, a set of utility functions, including random number generation
   - *rmi*, with classes for remote method invocation, used when an application is distributed across several computers.

   To develop a discrete event simulation system, the programmer develops a simulation package - that is, a set of classes which support discrete simulation.

3. A familiar syntax: the syntax of Java is based on that of C++, which in turn stemmed from C, both of which are widely used programming languages. One feature of C and C++ , missing in Java, is the explicit use of pointers and the

manipulation of their addresses. Anyone who has needed to debug a large program with complicated and slippery pointer structures is unlikely to mourn their passing.

4. Support for multi-threading: a thread, sometimes known as a lightweight process, is a segment of computer program that can be executed relatively independently of the rest. A program that contains threads may thus be run asynchronously and on more than a single computer at the same time, each thread being run as a lightweight process on one of the computers. Java thus supports some of the basic requirements for distributed processing.

5. Highly portable: in common with earlier languages such as UCSD Pascal (compiled into an intermediate form known as P-code) and some versions of BASIC, Java is neither interpreted nor compiled in a true sense. Instead, each class is translated into a machine independent 'byte-code'. This byte code is then translated into executable code by a *Java Virtual Machine* (JVM) written for the particular hardware and software in use on the target computer. Thus programs may be written on one system and run on another, even if the latter is of a quite different type.

Sadly, all is not sweetness and light, and there are a few problems with Java, of which the following are most evident. Firstly, the use of multi-threading places the user in the hands of the operating system in a rather starker way than is normal, even for programs that are not explicitly multi-threading. That is, the user cannot always guarantee the order in which segments of program will run, for the control of the threads is handled by the operating system. For most applications, this does not really matter, but it makes Java unsafe for mission-critical work. The second snag is, paradoxically, the absence of pointers. There are times when they are useful, especially to support some types of run-time binding. There are ways round this problem, but they can get rather complicated. Thirdly, interpreted programs tend to run slower than those that are compiled. Hence many Java programs are slow, but the increasing availability of Just-In-Time compilers is changing this. Finally, like C++ in its early days, Java is a moving target. In 1995 Sun announced version 1.0 of the Java Development Kit (JDK). In 1997 this became version 1.1, version 1.2 was in beta test in late 1998 and became v2.0 when finally released. Each version adds features to the language, making it bigger every time.

Program development in Java

As discussed above, one attractive feature of Java is its machine independence achieved through the use of byte codes and JVMs. These allows program development to proceed as shown in figure 1, which assumes that the developer has access to Java packages containing pre-written classes that will form part of the program under development.

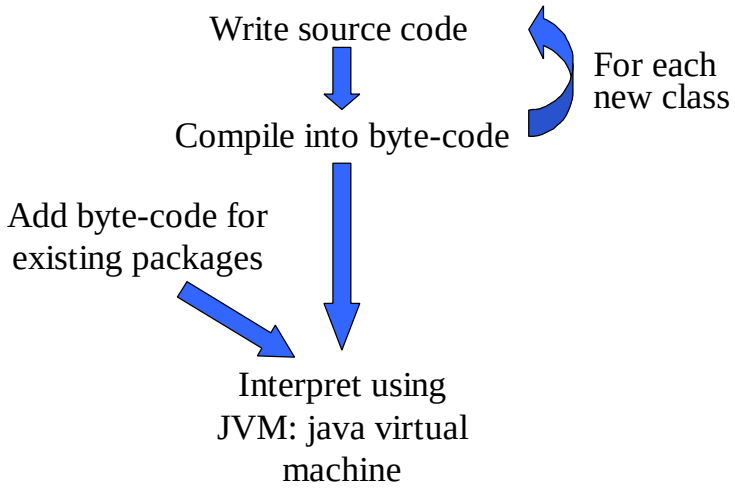

Figure 1: Java: program development

As with all program development, the process is actually cyclic and the program is gradually refined over time. Within this overall cycle, the rough development sequence is to begin with the definition of the classes needed for the application, including the use of classes that already exist. The application classes are written as encapsulated methods and data in the Java language, using any text editor. Once written, each class can be read by the Java system (known as the JDK) and translated into byte code. Each class normally sits in its own file and, once translated into byte code, carries the .class suffix. With careful design, a Java program can be written as a set of relatively independent classes.

It is inevitable that the user's Java program will make use of classes from other packages, if only for input/output. As with C and C++, these packages are linked to the user's classes. However, in the case of Java, this is usually done at run-time via the Java Virtual Machine (JVM). The JVM takes the byte code for the user's classes, the byte code from other packages and links these, translating the final set into an executable form. There is no need for the translations into byte code and then into an

executable form to be carried out on the same computer or the same type of computer. To run a Java program on a computer requires a JVM that has been written for that computer and its operating system. JVMs exist for a wide range of computers and are under development for hand-held devices such as those based on Windows CE computers and Psion's EPOC 32 system.

This process of class development, linkage and translation can be automated to some degree. Hence there are systems around such as Borland JBuilder[9], Microsoft J++[10], and Symantec Café[11] that include integrated program development environments and extensive packages to support this process in a smooth way.

Inside discrete event simulations

Pidd[2] provides a detailed account of the inner workings of discrete event simulation programs. For present purposes, figure 2 gives a summary of the four main features. It shows that such simulations are controlled via some form of user interface, nowadays usually graphical and windowed. Two distinct roles lie behind the figure. The first is that of the system developer who must provide a simulation executive (or control program) which ensures that the events of the simulation and their consequent activity occurs at the right simulated time and in the right sequence. This executive is wholly independent of any particular application and controls the simulation, much as a computer operating system controls user programs and other applications. The system developer must also provide a simulation library to provide support for features such as random sampling, specialised debugging, specialised input/output and animation.

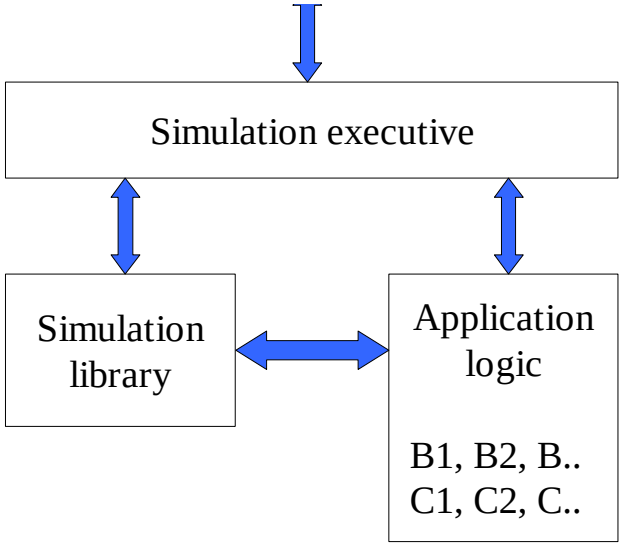

Figure 2: Inside a discrete event simulation

The second role is that of simulation analyst, who takes the components provided by the system developer to implement a simulation model of some application or other. Thus, the analyst must write code of some kind to specify the unique entities and their logic so as to capture the essential features of the application in a simulation model. This application logic can be expressed in several different ways. In the examples discussed later in this paper, a three-phase approach is used.

If an object oriented approach is employed by both developer and analyst, then the application logic will be written as a set of classes. In addition, the analyst may choose to extend the simulation library by adding classes that are found to be useful in more than one application.

Thus, in Java, the developer may make use of in-built classes to develop the simulation executive and may employ other classes to produce a user interface and the simulation library. The analyst adds new classes to develop a package suited to the particular application and may also extend the simulation package provided by the developer.

A SIMPLE SIMULATION SYSTEM IN JAVA

Recent years have seen a number of simulation systems developed in Java. Examples include that of Buss[12], whose event-based system can run simple simulations over the internet, of Zeigler and colleagues who have implemented DEVS[13] in Java, and of Page et al[14] who use remote method invocation (discussed later in this paper) to support distributed processing. Healy and Kilgore[15] describe the development of Silk,

a process-interaction based system written in Java. Kreutzer et al[16] show how Java can be used to simulate queuing networks. Ferscha and Richter[17] discuss the use of Java to develop co-operative distributed simulations. There are a number of world-wide-web pages[18,19] that bring together some of these developments. Also related to discrete simulation, Wilcox and Pooley[20] describe a prototype system that enables distributed users to co-operate over the internet in experimenting with a single simulation model as a form of distributed group decision support.

This section describes a simple simulation system, written in Java, that employs the three-phase approach developed by Tocher[21]. In the three-phase approach, the simulation analyst develops the application logic as a set of Bs and Cs. The Bs are used to represent activity that is Bound to occur at some specified time and which can, therefore, be scheduled by the simulation executive on its event calendar. The Cs are used to represent other activity that cannot be so scheduled, since it may depend on the availability of resources, or on the state of other simulation entities. For example, the start of a job on a machine in a factory is usually represented as a C, since it requires a machine and job to be available and may also require other resources. By contrast, the end of the processing of the job is usually modelled as a B, since once started it is possible (in statistical terms) to say how long the job will take. In general, though with a few exceptions, resources are consumed in the Cs and released for re-use in the Bs.

The tasks undertaken by the three-phase executive are thus as follows.

- Scheduling entities for Bs that are due to happen at some time in the simulated future. In addition, it will need to have some way to de-schedule entities whose planned future changes due to exogenous events.
- Managing an event calendar that enables control of the entities of the simulation. The event calendar shows when the Bs are due and the executive ensures that they are executed when they become due in simulated time.
- Enforcing priority systems when an entity could engage in several Cs at the same time. For example, which job should be started on Machine A if there is a choice available?

The simple Java implementation

This first implementation shows how Java may be used like any other programming language to develop a three-phase discrete event simulation system. It makes no use of Java as a language to support distributed or world-wide-web based processing, nor does it make explicit use of multi-threading. It does, however, illustrate a few of the useful features of Java. As mentioned previously, this simple system provides a simulation package that others may use to develop their own simulation applications. Like all Java packages, it may be extended or incorporated in other applications. Its main classes are as follows.

- Executive: as discussed above, the simulation executive has three main tasks, all based around an event calendar that it must manage. The event calendar is, in effect, a list of future events (the Bs) which is usually maintained in a chronological sequence. Thus, those events due nearest to the current simulation time are at the head of the list and those due latest are at the end of the list. There are well-known techniques for managing the event list in an effective way, but the simple Java application discussed here uses the *Vector* class that comes as part of the *java.util* package. A *Vector* is simply a growable one-dimensional array whose members can belong to any Java class. Thus, elements may be added and removed by using the methods that form part of the *Vector* class. Unless used with care, the *Vector* class manipulations can be rather slow, but that was not an important issue in implementing this simple simulation package.

- Entities: as in the C++ systems discussed in Pidd[8], entities in the application should extend abstract class, *GEntity*, which maintains the minimal state information needed to represent a simulation entity. The data members of *GEntity* are shown in figure 3 and each of these fields employs one of the primitive types (long, int, etc) of Java. The protected keyword indicates that the data will be accessible to a method of the *GEntity* class or by a method of one of its descendents. The private keyword, indicates that *EntCounter* may only be accessed via a method of *GEntity*. Because *GEntity* is an abstract class, it will have no members, but will be used as the basis of the application classes that represent the actual entities of the simulation.

```
public abstract class GEntity
// General entity class for three phase simulation
{
    public String Name;
    protected long TCell = 0;
    protected long PrevTime = 0;
    protected boolean Avail = true;
    protected long Util = 0;
    private static int EntCounter = 0;
    protected int EntNum;
    protected Activity NextB;
```

Figure 3: GEntity class definition

- Resources: resources are countable items that are effectively identical and which do not need to be individually tracked during the execution of the simulation. Examples might be the number of passengers waiting at a bus-stop or the number of parts waiting to be processed in a factory. In the simple Java system, these are represented by the *Resource* class.
- Queues: entities often need to join queues to wait for resources or for other entities and it thus makes sense to create a class *Queue* to represent this. In effect, the *Queue* is a circular linked list in which the tail of the list is linked to the head. *GEntities* and their descendents may be added to a *Queue* in whatever queue discipline is desired.
- Trace: since the simple Java system makes no attempt at an impressive user interface, some form of trace is needed to help the user check the events as they occur. The *Trace* class is used to create a file to which events are written as they occur, and these may be checked later to see if proper event sequencing and execution has occurred.
- Random sampling: most discrete event simulations include random events that are modelled by random sampling algorithms using pseudo-random numbers. Rudimentary sampling comes as one of the Java classes built-in to the JDK's

java.util package, and the simple simulation system extends this to make it more usable.

To use this simple Java package, the user must define entity classes that extend the *GEntity* class and must write Bs and Cs to represent the logical interaction between the entity classes. For example, the code fragment shown in figure 4 is from the definition of a *Personal Enquirer* class for the Harassed Booking Clerk simulation discussed in Pidd[2]. This shows that the *PersCust* class extends the general entity (*GEntity*) class (see earlier) by adding two new data fields of type *Queue*; two counters *Number* and *Counter*; average values for the service time and inter-arrival time to be used as the basis for random sampling in the simulation; two probability distributions, *interArrTime* and *serviceTime*, which can be bound at run-time to define the actual distributions used; plus the two Bs *Arrive* and *EndServe*, which belong to this class.

```
public class PersCust extends GEntity
{
    public static Queue QIn;
    public static Queue QOut;
    private int Number;
    private static int Counter = 0;
    private static int IAT;
    private static int ServeTime;
    private static Sample interArrTime, serviceTime;
    public static Arrive nextArrival;
    public static EndServe serviceOver;
```

Figure 4: PersCust class definition

In addition, some initialisation code is needed to set up the simulation and finalisation code may be needed to analyse and display the final output. One problem that must be faced when writing three-phase simulations in an object oriented way is, what to do about the Cs? The Bs are no real problem, since they belong to specific entity classes and are entered on the event calendar when the entity in question is scheduled for some future B. Thus, the Bs become methods of the relevant entity classes. The problem is that the Cs may belong to no single entity class since a C may involve the co-operation of more than one class of entity. However, the further

problem is that methods need, in Java, to belong to a class and cannot be free-standing.

The solution adopted was to define a Java *interface* called *Activity*. A Java interface is a special type of abstract class, in which all data members (if there are any) are constants. Hence, in the Java code that defines the *GEntity* class shown in figure 3, its type is shown as an *Activity*. In this way, the Cs are implementations of the *Activity* interface and they belong to the application class, rather than to a single entity class.

The system described here in outline was implemented with relatively little effort using free software tools and Sun Microsystems JDK 1.0. It is available for download via the world-wide-web from the site with the URL http://www.lancs.ac.uk/staff/smamp/MPCSMS4.html.

Lessons learned from the simple simulation system

The most obvious lesson learned from the developing the above system was that it was not difficult, certainly not for someone who had developed similar systems in other programming languages. Like all programming languages, Java has its quirks, some of which are to be exploited and some of which are to be avoided. However, the task was not too difficult and the resulting system, though by no means a commercial product, works well enough for student use. Linked to this, one pleasant realisation was that even JDK 1.0 came with packages that were of great help in developing discrete event simulations. In particular, the *java.util* package provides the *Vector* growable array and the *Random* class has a simple random number generator.

The other lessons were a little less positive. The first was that portability is still a light at the end of the tunnel unless only the most basic facilities of Java are used. For example, *java.awt* provides simple facilities for graphical user interfaces with fill-in forms and the like. However, these look rather crude when compared with the objects provided in systems such as Visual C++ or Visual Basic, both of which run only on PCs. To counter this, products such as J++ and JBuilder provide extensive extra windowing objects, but these may only run on PCs under MS Windows and not, for example, on UNIX systems. The second lesson, discussed in more detail later, is that the language is still a moving target.

TOWARDS DISTRIBUTED SIMULATION IN JAVA

This section describes the development of a rather more sophisticated three phase simulation package in Java. Whereas the system described above used Java much like any other programming language, the system described here (known as JUST - the Java Ubiquitous Simulation Tool) goes much further. In particular, it was developed to investigate how Java could be used to support distributed simulation. Before describing JUST, it is perhaps worth discussing why such a tool may be worthwhile.

The first reason is the oft-quoted slogan, originating with Sun Microsystems, that 'the network is the computer'. Today, many computers are inter-connected in some way or other. They may be desk-top workstations that are permanently wired into a local area network or intranet. They may have direct or dial-up access to the internet. They may be laptops that are periodically docked into a LAN. They may be palmtops that communicate via infra-red or GSM phones. Whatever the linkage, they represent distributed computational power that can be harnessed in different ways. Apart from the problem of communication bandwidth, it may not matter where the actual computation takes place when a user is connected to others. Having simulation tools to exploit this future seems worthwhile.

Following on from this, Java was designed with network computing in mind. The facilities needed are not add-ons, but were part of the core design of the language. Thus, from a research and potential application perspective, it presents an attractive opportunity of exploiting network computation for discrete simulation. Parallel and Distributed Simulation (PADS) is a well-developed subject in its own right and offers technologies that have been exploited for large scale and real-time simulations, especially in the defence sector. Fujimoto[22] gives a lucid description of some of the issues to be faced in this type of simulation.

Distributed simulation can mean a number of different things, as follows. First it can mean that a single computer program is distributed across several processors. The processors may be within a single compute box or may be interconnected in an external network. Secondly it can refer to client-server applications in which part of the simulation program acts as a server (most likely the simulation executive and library) that is used by a number of clients, which may represent the application logic. There is no particular reason why a single server cannot simultaneously serve multiple

different clients, each being a different simulation application. Finally, the term could refer to multiple simulation replications of the same simulation program (*n* identical clients, but with different random number streams), all served by the same executive.

The JUST system, as described here, supports multiple simultaneous client-server applications with co-operative processing. Thus, the executive sits on the server and client programs connect, using the server to control their operation. This approach may have advantages for simulation applications in which the Bs and C are compute-intensive. It may also be appealing for software vendors, since it permits software metering - that is, income generated on a pay-by-use basis.

THE JUST SYSTEM

The JUST system and distributed processing

The JUST system utilises a number of features of Java that support distributed processing. The first is the idea, mentioned earlier, of multi-threading. A multi-threaded program may handle different tasks, apparently simultaneously. If the program is run on a single processor, then this concurrency is apparent or virtual, since what is actually happening is that each thread has brief control of an extremely fast processor. If the program is run across multiple processors, then this multi-threaded concurrency can be real. Languages such as C and C++ support concurrency to some degree, but it forms a basic part of Java - which, as discussed earlier, can lead to problems unless care is exercised. In the case of the JUST system, multiple instances of the simulation executive can be created as objects that are members of the *Manager* class. Each instance may be used to control a different three-phase simulation program.

The second feature used by the JUST system is known as *reflection*, which is a concept that can be hard to understand. In conventional object-orientation, run-time binding is used as one way of writing a general-purpose program without knowing exactly what objects the program will need to work with. For example, a drawing program may need to draw different shapes and in different colours. It may also need to be extended to cope with new shapes, colours and combinations of these. Since the methods (functions) of the programs are only linked to the objects at run-time, this is known as run-time binding. In languages such as C and C++, this is usually achieved via pointers, but these are not available in Java and reflection is used instead.

Reflection was provided in JDK 1.1 and enables classes to 'reflect' that is, to look inside themselves. A method for an unknown class can be executed by getting the unknown class to reflect and to invoke its internal methods. In this way, a very safe form of run-time binding can be achieved. Reflection also gets round one of the basic problems of object orientation discussed in Pidd[8]. This is that member functions of descendent classes must have prototypes in their base class if they are to be invoked via their base class, as is needed in run-time binding. For example, if a method *Rotate* is to be called for a new graphics class (say a shape known as a *Squodge*) then this can only be achieved via run-time binding if the base class of *Squodge* (say, *Shape*) has a prototype function with the same name *Rotate*. This makes it hard to allow a simulation executive to call unknown methods for unknown classes without some considerable jiggery-pokery in most languages. Reflection makes it rather simple.

The third feature used by JUST is the Remote Method Invocation (RMI), which is provided by the java.rmi package. RMI supports communication by enabling a method sitting on one computer to be invoked by a call from another computer without worrying too much about the detail. For example, consider a computer known as *computer1* that contains a class called *class1,* which itself has a method called *method1*. Suppose that a class on another computer, *computer2*, wishes to call this method, To do this using RMI, *computer2* must locate the *class1* that owns the method in the registry of *computer1*. When this is done, the method on *computer1* can be invoked by *computer2* as if it were in its own registry, making its use very simple and safe.

Taken together, reflection and RMI enable client server applications to be created in a relatively painless way. Combined with multi-threading, they allow system developers to exploit the idea that the network is the computer.

Client-server simulations in the JUST system

The initial implementation of the JUST system supports client server applications as shown in figure 5. This assumes that there are at least two computers co-operating to execute a simulation model. One of these computers is the simulation server and the others (one or more) are simulation clients. In the terms used in figure 2, for discrete simulations in general, the simulation server plays host to the simulation executive

and (most of) the simulation library. The application logic sits on the client computer(s). The programs on all the computers must be written in Java, using the principles of the JUST system and should be already in byte code. Each computer must have a compatible Java Virtual Machine to translate the byte codes into executable forms. The computers must be linked by a network, which could be a local intranet or the global internet, such that they are able to communicate using standard protocols.

Figure 5: Client:server simulation in JUST

A simulation is initiated by a client calling the server (with a known machine address) in the same way as any computer calls another using internet technology. This call initiates the *Manager* class in the server, which is, in effect, the executive that will control the simulation. In object oriented terms, the call instantiates a new object in the *Manager* class and this object (sometimes known as an instance) will control a simulation run. Since the *Manager* object is to control simulation entities, then these entities on the client side must belong to classes that are descendents of the *GEntity* class. As in the case of the simple Java implementation discussed earlier, *GEntity* is an abstract class used to represent the basic features of any entity. Specific entity classes will extend the *GEntity* class. Thus both sides, client and server, need access to the entity class. The server needs access so that the *Manager* knows what it is controlling and the client needs it to define the entity classes that will comprise the simulation application.

In addition, as figure 5 makes clear, the client side needs two other classes. It needs the application entity classes that extend *GEntity* and it needs the Cs that are

used to represent co-operative activity in the simulation. As discussed earlier, most Cs cannot belong to a single entity class since they represent co-operation between different classes of entity. The Bs, on the other hand, are to be found within the entity class definitions, since they are specific to particular entity classes.

The JUST system, described here, is scaleable, in that there can be one or more clients running at the same time in co-operation with the same single server. The clients could be wholly different simulations - which would be the case if multiple users were connected to the same server, or they could be multiple instances of the same simulation model. In the latter case, this allows multiple replications to be run under different random number streams so as to gain representative samples.

The JUST system has also been implemented as distributed simulation system in which a single application has its application classes spread over a number of distributed computers. Whether this is worthwhile depends on the relative demands for computation on the networked computers and on the communication speeds of the network. However, in the past, distributed simulation has needed access to parallel computers or to rather clever communications programming. The use of Java on the internet makes this much easier.

IS THIS PRACTICAL, AND WHY BOTHER?

Anyone who has used the internet knows how slow it can be. In most cases there are two causes. The first is that the network links and nodes of the internet itself have limited bandwidth and thus cannot supply the data fast enough to satisfy most of us. Linked to this, the internet has enjoyed explosive growth and this has led to system overload. The second reason for slowness is that the local connection from a computer to the internet may be very slow, such as through a modem over a conventional phone line. Does this mean that a system such as the JUST system discussed here is hopeless in practice?

As far as the current internet is concerned, the answer is clearly, yes. It is far too slow for this type of application to be practicable. However, many readers of this journal will recall that only a decade ago, a computer with 1 Mbytes of memory was thought powerful. They may recall, too, that much of the British phone system was still based on pulse dialling with unreliable Strowger switches at the exchange. About a decade ago, a typical home modem supported data transfer at 30 bits per second. In

the last decade this has risen to almost 64 K bits per second. Supply tends to follow demand, and it seems likely that the bandwidth problem will be eased by new technologies.

As far as intranets are concerned, the communications speeds are already much higher and permit MBit data transfer routine. Applications distributed in this way may never run as fast as in a single compute box, but the situation is already much better than it is for the global internet. However, this form of client server technology does have two aspects that should, eventually, spill over onto the internet as well.

The first is that of control. Rather than have to distribute many versions of simulation software, this brings a return to the position in which applications can be centralised should that seem sensible. It would not be sensible for all applications, but it might be sensible for some. The main advantage is that of version control. Users can be sure what version they are using, as can help desks providing remote support when necessary. The second is the possibility of pay per use licensing rather than buying a licence in the present form. This of course, opens up the possibility of differentiated access deals for different types of user.

There is one major drawback to the use of Java in this way. This is that the difficulty of controlling multi-threaded programs means that it is unsuitable for real-time simulation and for mission-critical work. Real-time simulations require computations to be done in a clearly defined way and within a specified (usually very small) time period. The distributed and multi-threaded nature of Java and its implementations on different hardware and software makes this impossible, at least at the moment. The same is true of mission critical work, which some might hesitate to conduct over public networks. It seems reasonable to assume that, before too long, there will be a successor to Java that overcomes these problems.

Like all general purpose programming languages, Java is not the ideal language in which to write discrete simulation programs. For rapid development of straightforward applications it still makes sense to use simulation languages such as MODSIM III[23] or VIMS such as Witness[24] . As discussed in Pidd[2], this is because simulation languages and VIMS come with ready-written simulation executives and libraries and with animated graphics well-suited to the run-time display of simulation output. Nevertheless, the development of the JUST Systems, plus the experience of

commercial developers such as Healy and Kilgore[15] suggests that Java, or something like it, might be the basis of some commercial simulation systems in the future.

In summary, the development of the JUST system, and that of other Java-based simulation systems around the world, demonstrates that distributed simulation is possible using straightforward internet technology. The same may be true of other OR-type computations, such as those required for optimisation and for heuristic approaches.

ACKNOWLEDGEMENT

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

REFERENCES

[1] Fildes R. and Ranyard J.C. (1997) Success and survival of operational research groups – a review. Jnl Opl Res Soc, 48, 4, 336-360.

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

[3] Law A.M. & Kelton W.D. (1991) *Simulation Modelling & Analysis (2nd edition).* McGraw-Hill International Edition.

[4] http://www.w3.org/Consortium/

[5] Flanagan D. (1997) *Java in a nutshell* (2nd ed) O'Reilly & Associates Inc, Cambridge, Mass.

[6] Deitel H.M. and Deitel P.J. (1997) *Java: how to program.* Prentice-Hall Inc, Upper Saddle River, NJ.

[7] Dahl O. & Nygaard K. (1966) *SIMULA - an Algol-based simulation language.* Comm ACM 9, 671-678.

[8] Pidd M. (1995) Object orientation, discrete simulation and the three-phase approach. *Jnl Opl Res Soc.*, 46, 362-374.

[9] Borland International (1998) http://www.borland.com/jbuilder/

[10] Microsoft Corporation (1998) http://www.microsoft.com/visualj/

[11] Symantec Corporation (1998) http://www.Symantec.com

[12] 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.

[13] Zeigler B.P. (1984) *Multi-facetted modelling and discrete event simulation.* Academic Press, New York.

[14] 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.

[15] Healy K.J. and Kilgore R. Introduction to Silk™ and Java-based simulation . In *Proceedings of the 1998 Winter Simulation Conference*. ed D.J. Medeiros, E.F. Watson, J.S. Carson and M.S. Manivannan, Grand Hyatt, Washington, D.C. December 13-16, 1998,

[16] 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

[17] 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.

[18] http://www0.cise.ufl.edu/~fishwick

[19] http://ms.ie.org/websim/survey.html

[20] Wilcox P.A. and Pooley R.J. (1998) Factory simulation across the world-wide-web using Java. In *Proc SCC Western Multi Conference*, San Diego, February.

[21] Tocher K.D. (1963) *The art of simulation.* English Univ Press, London.

[22] Fujimoto R.J. (1990) Parallel discrete event simulation. *Comm ACM*, 33, 10, 30-53.

[23] Goble G. & Wood B. (1998) MODSIM III: A Tutorial with Advances in Database Access and HLA SupportIn *Proceedings of the 1998 Winter Simulation Conference*. ed D.J. Medeiros, E.F. Watson, J.S. Carson and M.S. Manivannan, Grand Hyatt, Washington, D.C. December 13-16, 1998,

[24] Rawles I. (1998) The Witness toolbox – a tutorial**.** In *Proceedings of the 1998 Winter Simulation Conference*. ed D.J. Medeiros, E.F. Watson, J.S. Carson and M.S. Manivannan, Grand Hyatt, Washington, D.C. December 13-16, 1998,