CONCURRENCY: PRACTICE AND EXPERIENCE
Concurrency: Pract. Exper., Vol. **10**(11–13), 999–1005 (1998)

JWarp: a Java library for parallel discrete-event simulations

PEDRO BIZARRO*, LUÍS M. SILVA AND JOÃO GABRIEL SILVA

Departamento de Engenharia Informática, Universidade de Coimbra, Polo II, 3030 Coimbra, Portugal
(e-mail: bizarro@dsg.dei.uc.pt)
(e-mail: luis@dei.uc.pt)
(e-mail: jgabriel@dei.uc.pt)

SUMMARY

Java is a very promising language for use in the simulation of physical models due to its object-oriented nature, portability, robustness and support for multithreading. This paper presents JWarp, a Java library for discrete-event parallel simulations. It is based on an optimistic model for synchronization of the simulation entities: the Time Warp mechanism. We introduce the main features of the library and discuss some of the implementation details. ©1998 John Wiley & Sons, Ltd.

1. INTRODUCTION

In several areas like engineering, computer science, economics and military applications, simulation is used to study the behaviour of complex models. The execution of some of these simulation models can be a very time-consuming task. For statistical reasons, it might be necessary to simulate a model for quite a long time, or to perform the same simulation several times with different parameter values.

A possible solution to reduce the execution times of long-running simulations is by using multiple processors operating in parallel[1]. A typical simulation model involves several components or entities. By exploiting this inherent model of parallelism, it would be possible to speed up the performance of the simulations by decomposing these components through several processors. Every simulation model is a specification of the corresponding physical model and is composed of a set of states and events. In a discrete event simulation, the state of the system only changes at discrete points in simulated time. A natural decomposition strategy can result in an object-oriented system design, where an object corresponds to some component of the real system and is represented by a computational task that is assigned to a processor for execution. In this way, every component of the model is simulated by a logical process (LP). A discrete-event simulation requires the existence of multiple LP entities, a time-ordered event list holding time stamped events to be processed in the future, a global discrete clock that indicates the current simulation time and a set of state variables that define the state of the simulation. The simplest way to manage the event list would be based on a centralized strategy. The list of events would be managed by a single process (master), and there would be a pool of slave processes running on the parallel system that would execute those events in a concurrent way. However, the existence of a centralized queue of events would represent a bottleneck to the simulation, thereby clearly reducing the potential for parallelism.

*Correspondence to: Pedro Bizarro, Departamento de Engenharia Informática, Universidade de Coimbra, Polo II, 3030 Coimbra, Portugal.

CCC 1040–3108/98/110999–07$17.50

Received 28 February 1998
Revised 4 May 1998

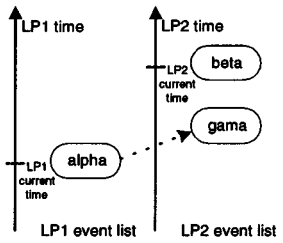

Figure 1. The problem of causality errors

The most effective way of conducting parallel simulations is to eliminate the globally shared-event list and use a completely distributed list of events. Each LP will be assigned to a processor that maintains its own local simulation clock (LVT – local virtual time), a local event list and a set of state variables. Events are modelled as timestamped messages, which are exchanged between the physical objects of the application (LP).

However, the schemes that follow a distributed strategy would require some synchronization protocols to make sure the events are processed in a consistent order by all the LP entities. These synchronization protocols may increase the costs of communication between processors. Nevertheless, they have deserved considerable attention from the parallel simulation research community[2].

In order to understand the main issue behind the use of distributed event lists let us look at Figure 1. It represents the temporal execution of two logical processes (LP1 and LP2). The LP1 entity is processing event *alpha*, while LP2 is processing event *beta*. The execution of event *alpha* generates a new event (*gamma*) that is sent to LP2. Event *gamma* has a lower timestamp than event beta, and thus should have been consumed before. Due to the asynchrony of the LP entities it was not possible to ensure a consistent order in the processing of events, thereby resulting in a *causality error*[1].

The synchronization protocols have been broadly classified as *Conservative* or *Optimistic*[3]. Both schemes are based on the sending of messages carrying some causality information.

The **Conservative approach**[4] strictly avoids the possibility of any causality error ever occurring. This is achieved by stopping each process until the system is sure that no other event will be scheduled by any other LP with a timestamp smaller than the one in the top of the local list of events. This method introduces some blocking on the execution of processes and restricts the potential for parallelism. Besides, it is prone to the occurrence of deadlock and thus requires a deadlock detection and recovery scheme.

The **Optimistic approach** tries to exploit all the potential parallelism available in the simulations. The Time Warp mechanism is a well-known optimistic approach based upon the Virtual Time paradigm[5]. It relies upon a scheme for causality error detection and a recovery scheme based on a rollback technique. An optimistic LP advances simulation and its local virtual time as far as possible.

An event scheduled in some LP with a timestamp in the local past relative to the local virtual clock is said to be out of chronological order or to be the *straggler* message. It will force the LP entity to roll back to the most recently saved state in the simulation history consistent with the arrival of that event message. LP will then restart the simulation at that point, thereby correcting the causality error.

In order to allow this rollback operation every LP entity is forced to save its simulation state from time to time. All the messages that were sent previously after that instant of time should be undone. This is achieved by sending some sort of anti-messages to annihilate the original messages. If these ones were already consumed by the destination processes they will be forced to roll back as well to a previous saved state. It was proved that the protocol will not roll back until the beginning of the simulation and always ensures some forward progress for the computation[5].

The major drawback of the Time Warp approach is the need to save each process state periodically[6]. To free up some of the used memory the simulation system calculates a time limit, called Global Virtual Time (GVT)[7] beyond which no process is required to roll back and thereby the system can perform some garbage collection scheme to free up some of the unused data structures.

2. THE JAVA OPTION

Java has received tremendous hype in the past few years. In fact, it has several advantages over other languages and it fits particularly well with this kind of programming. The features of Java that facilitate our implementation were:

- communication-centric – Java has some built-in classes for network communication and to solve the problem of portability: no *little-endian vs. big-endian* dilemmas. All data types are well defined and consistent in both size and binary representation across the JVM implementations
- object-oriented – it allows the exchange of objects (events) between processes and ensures the modularity of code
- multithreaded – allows splitting the program into separate and cleaner modules providing greater concurrency
- serialization mechanism – gives support to checkpoint and recovery schemes of the Time Warp approach.

A comprehensive list of computing platforms has been enhanced with the support of the Java Virtual Machine (JVM)[8]. Since Java programs are entirely portable across the systems that have a JVM we will be able to execute parallel simulations in heterogeneous systems, comprising networks of personal computers running a Microsoft Windows operating system or clusters of workstation machines running some flavor of Unix. All this will be possible with a simulation tool like JWarp. Programmers are not required to change any line of code of their simulations since Java provides the necessary support to deal with the heterogeneity.

3. JWarp ARCHITECTURE

Figure 2 presents JWarp's architecture. The left side highlights the message flows (lines), threads (ovals) and buffers (rectangles). The right side shows what goes on inside the buffers. It will be used to explain rollback and garbage collection operations. As can be seen, threads have short names that represent the data storages they connect. Message flows are represented with full lines and other information flows with dotted lines.

Events arrive at every `LP` by being first received in `cs2ib`, placed in `IB`, received in `ib2iq` and placed in `IQ`. Outgoing events are placed in `OQ` by `LP`, received by `oq2ob`,

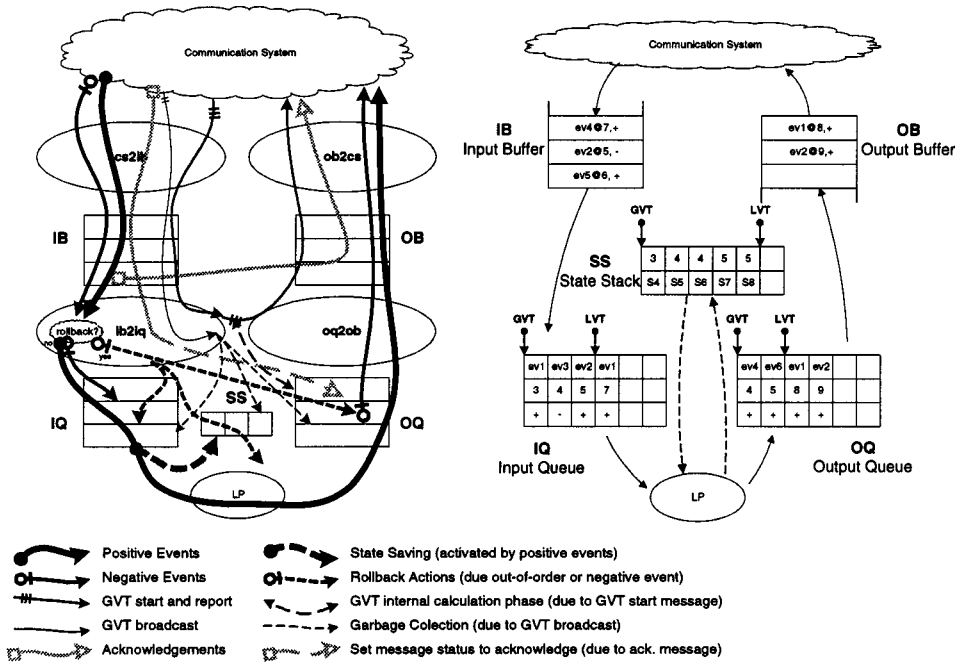

Figure 2. On the left, the flow of internal messages, threads and buffers; on the right the buffer behaviour with $GVT = 3$ *and* $LVT = 5$

placed in `OB`, received by `ob2cs` and sent into the network.

LP state variables (defined by the programmer) are saved from time to time in the State Stack (`SS`).

Although all the buffers have been represented equally in Figure 2 they do not have the same behaviour. In JWarp, when a buffer is asked to retrieve the next event it can do one of two things: retrieve, return and delete the message; or just retrieve and return. Buffers `IB` and `OB` delete retrieved messages, while `IQ` and `OQ` do not. Events are maintained in `IQ` and `OQ` because whenever a rollback happens, already consumed events (those in `IQ` before the `LVT` pointer) must be consumed again. Likewise, sent events (those in `OQ` before the `LVT` pointer) must be kept because there could be a need to send anti-messages, which are generated from the normal ones. Thus, fetching an event in `IQ` or `OQ` means only to retrieve a copy of it and move forward the `LVT` pointer.

Note that although the pointers are called `LVT` and `GVT` they do not store LVT and GVT time values. They are just a reference in the array buffers. Buffers `IB` and `OB` do not need to keep any of its messages. All the information needed for a rollback is stored in `IQ`, `OQ` and `SS` between each one's `GVT` and `LVT` pointers.

The threads `cs2ib`, `oq2ob` and `ob2cs` are just running an infinite cycle fetching data from one side and placing it in the other. The real brain of the operations is performed by thread `ib2iq`: it will detect messages out-of-order and causality errors; it will command state restoration and anti-message sending; it will process GVT calculation requests; and it will acknowledge every received message.

If there were no straggler messages the JWarp internal behaviour would be the following:

1. Message arrives at `cs2ib`.
2. Message is placed in `IB` **in arriving order**.
3. It is fetched by `ib2iq`.
4. A corresponding acknowledge message is put in `OB` by `ib2iq`.
5. Acknowledge message is sent by `ob2cs`.
6. `ib2iq` puts the received message in `IQ` **ordered by simulation time**.
7. Depending on the checkpoint frequency, the `LP`'s state is saved in `SS`.
8. Just after the state saving the message finally arrives to `LP`. `LVT` is updated to a new value: the incoming message processing (or simulation) time.
9. `LP` processes the message and responds by sending 0, 1 or more messages, to one or more recipients, that are placed in the `Output Queue` **in arriving order**.
10. Messages are then fetched from `OQ` and placed in `OB` by `oq2ob`.
11. They are finally sent over the network if they are remote events (to be processed in another `LP`) or placed in `IQ` if they are local events.

On the acknowledge message receiving side, the incoming acknowledge message is cross-checked with the messages in `OB` to find its counterpart. When it finds it, it changes its status from `unacknowledged` to `acknowledged`. Messages need to be acknowledged due to the GVT calculation, as shown in [9].

Causality errors are detected when the `ib2iq` thread places a new event in buffer `IQ`. Causality errors may be caused by just two things: an arrival of a positive message timestamped in the past, or an arrival of a negative message timestamped in the past when its positive counterpart had been consumed already. Every time a causality error occurs the LP must rollback. Rollback consists in restoring the state (fetched from `SS`), send anti-messages, if necessary, and adjusting the pointers `LVT` in the buffers (this will lead to re-simulating some events is `IQ`). Restoring the state is achieved by inverting the checkpoint operation. The messages that need to be undone i.e. to send their corresponding negative version, are the ones sent after the new restored state.

In a rollback operation, the system adjusts `OQ`'s `LVT` pointer to the proper place and switches signs of all the messages between the new `LVT` pointer position and the old `LVT` pointer position (this corresponds to the messages that were sent before and must be undone). Messages that were kept after the old `LVT` pointer were never sent and are simply deleted (see Figure 3). Unlike normal messages, the anti-messages are deleted from `OQ` after being sent because they will never be undone.

Checkpointing is obtained by converting all declared data-structures, together with the LVT value, into a byte stream, and storing this byte stream into a Java hashtable.* Restoring the state corresponds to the opposite operation, i.e. transforming the byte stream into variables and LVT. Both operations use the object serialization facility of Java.

The GVT algorithm that was used in JWarp was the one described in [9]. It finds a lower bound to the GVT value. A real GVT value is impossible to obtain due to communications delays and in-transit messages. In-transit messages are the ones sent by one process but not received by the other. GVT calculation is vital to the good behaviour of the simulation. It is the only way to prevent memory starvation, since the garbage collection scheme is attached to the calculation of GVT. One LP is defined as the GVT master. From time to time, it will broadcast a `GVT start` message that must be replied with a `GVT report` message

*The Java VM does not allow one to obtain the execution state of the threads and therefore one must force rollbacks and restores to occur always in the same execution place inside the cycle.

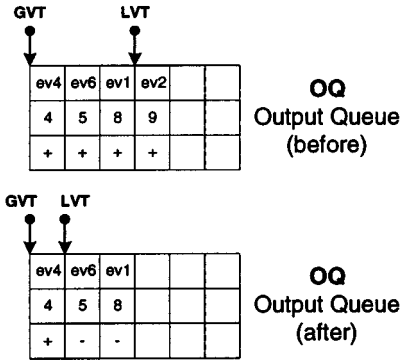

Figure 3.

stating each LP's choice to the next GVT value (calculated through its own LVT and the acknowledged and unacknowledged messages). The GVT master calculates the minimum of all GVT proposals and announces that value in a `GVT broadcast` message. The `GVT broadcast` message will start the garbage collection operation in each LP. That involves removing data from `IQ`, `OQ` and `SS` and to adjust their internal `GVT` pointers.

4. RELATED WORK

The Time Warp concept was proposed by Jefferson[5], and it was implemented into the TWOS (Time Warp Operating System)[6]. Several parallel simulation languages have also appeared in the last decade: OLPS[10], Maisie[11], ModSim[12], SCE[13], Sim++[14] and YADDES[15]. Another approach has been followed by other researchers, who decided to implement the parallel simulation system as a runtime library written in C++: examples include WARPED[16], SPEEDES[17] and HASE++[18]. The first simulation libraries in Java, SimJava[19] and SimKit[20], only supported sequential simulations. Shortly after, PDES Java libraries appeared: JTED[21], following the conservative approach, and Formax[22] following a Web-based optimistic approach.

5. CONCLUSIONS

Time Warp is a relatively complex simulation protocol but it has been proved a very effective technique for running complex asynchronous simulations[23,24]. We foresee that with an implementation in Java the use of Time Warp could become more widespread within the research community and it can be used for educational purposes.

Java is a very suitable language for building programming libraries. Its most distinguishing features are object serialization, platform independence, object-orientation, support for network communications, modularity and easy debugging.

REFERENCES

1. R. M. Fujimoto, 'Parallel discrete event simulation', *Commun. ACM*, **33**(10), 30–53 (1990).
2. Y. B. Lin and P. Fishwick, 'Asynchronous parallel discrete event simulation', *IEEE Trans. Syst. Man Cybern.*, **26**(4), 397–412 (1995).

3. A. Ferscha and S.Tripathi, 'Parallel and distributed simulation of discrete-event systems', *Technical Report*, University of Vienna, 1995.
4. K. M. Chandy and J. Misra, 'Distributed simulation: a case study in design and verification of distributed programs', *IEEE Trans. Softw. Eng.*, **SE-5**(5), 440–452 (1979).
5. D. R. Jefferson, 'Virtual time', *ACM Trans. Program. Lang. Syst.*, **7**(3), 404–425 (1985).
6. D. Jefferson, *et al.*, 'Distributed simulation and the time warp operating system', *Proceedings of 11th ACM Symposium on Operating Systems Principles*, **21**(5), 77–93 (1987).
7. S. Bellenot, 'Global virtual time algorithms', *Proceedings of SCS Multiconference on Distributed Simulation*, Vol. 22, No. 2, January 1990, pp. 122–127.
8. The Java Oasis, 'The Java Oasis: Java Developers Kit (JDK) – Index', `http://www.oasis.leo.org/java/development/jdk/00-index.html`.
9. Behrokh Samedi, 'A distributed algorithm to detect a global state of a distributed simulation system', *IFIP Conference on Distributed Processing*, October 1987.
10. M. Abrams, 'The object library for parallel simulations (OLPS)', *Proceedings Winter Simulation Conference*, San Diego, California, December 1988, pp. 210–219.
11. R. L. Bagrodia, V. Jha and J. Waldorf, 'The Maisie environment for parallel simulation', *Proceedings 27th Annual Simulation Symposium*, California, 1994, pp. 4–12.
12. J. West and A. Mullarney, 'ModSim: a language for distributed simulation', *Proceeedings of the 1988 SCS Multiconference on Distributed Simulation*, Vol. 4, (2), 1994, pp. 235–257.
13. D. H. Gill and F. X. Maginnis, 'An interface for programming parallel simulations', *Proceedings of the SCS Multiconference on Distributed Simulation*, Vol. 21, (2), Tampa, Florida, March 1989, pp. 151–154.
14. D. Baezner, G. Lomow and B. Unger, 'A parallel simulation environment based on time warp', *Int. J. Comput. Simul.*, **4**(2), 183–207 (1994).
15. B. R. Preiss, 'The YADDES distributed discrete-event simulation specification language', *Proceedings of the SCS Multiconference on Distributed Simulation*, Vol. 21, (2), Tampa Florida, March 1989, pp. 139–144.
16. D. Martin, P. Wilsey and T. McBrayer, 'The WARPED time-warp simulation kernel', *Technical Report*, University of Cincinnati, USA, 1994.
17. J. S. Steinman, 'SPEEDES: synchronous parallel environment for emulation and discrete-event simulation', *Proceedings of Parallel and Distributed Simulation Conference*, 1991, pp. 95–103.
18. F. Howell, 'HASE++: a discrete-event simulation library for C++', *Technical Report*, Department of Computer Science, University of Edinburgh, Available at: `http://www.dcs.ed.ac.uk/home/hase/projects/hase++.html`
19. SimJava Homepage, `http://www.dcs.ed.ac.uk/home/hase/simjava/simjava-1.1/`.
20. SimKit Homepage: `http://www.cpsc.ucalgary.ca/~adi/simkit/workshop/index.htm`.
21. J Cowie, 'JTED: parallel discrete-event simulation in Java', *Proceedings of ACM 1998 Workshop on Java for High Performance Network Computing*, February 1998, pp. 251–254.
22. B. Halderen and B. Overeinder, 'Formax: Web-based distributed discrete event simulation in Java', *Proceedings of ACM 1998 Workshop on Java for High Performance Network Computing*, February 1998, pp. 113–122.
23. F. Wieland, L. Hawley, A. Feinberg, M. DiLoreto, L. Blume, *et al.*, 'The performance of distributed combat simulation with the time warp operating system', *Concurrency: Pract. Exp.*, **1**(1), 35–40 (1989).
24. M. Presley, M. Ebling, F. Wieland and D. Jefferson, 'Benchmarking the time warp operating system with a computer network simulation', *Proceedings SCS Distributed Simulation Conference*, 1989, pp. 8–13.

