Jade's IPC Kernel for Distributed Simulation

J.G. Cleary, G.A. Lomow, B.W. Unger and Zhonge Xiao
Department of Computer Science
The University of Calgary,
2500 University Dr, N.W. Calgary,
Alberta, Canada. T2N 1N4

Abstract

An implementation of virtual time using Jefferson's Time Warp mechanism is discussed. The context for the implementation is a multi-lingual distributed programming environment with an underlying message passing system, called Jipc, which is accessible to Ada, C, Lisp, Prolog, and Simula programs. The intention is that distributed programs written using Jipc can be simulated using virtual time with only small changes to the original source code. The system is layered so that the different languages and different roll-back mechanisms can be supported.

1 Introduction

Virtual time is a paradigm for organizing distributed and asynchronous systems. The programmer or user of such a system sees virtual time as advancing steadily just as ordinary Newtonian time does. The behaviour of a distributed program can be reasoned about in terms of this virtual time.

The naive implementation of such a scheme would have processes wait until all possible messages sent at a particular time have been received or it is known that none were sent. This requires that all processes periodicly synchronize and wait for each other. Thus little or nothing is gained by having a distributed or parallel system, the processes in fact compute serially waiting on each other to finish.

Time Warp

Jefferson and Sowizral [1982, 1983, 1985] and Sowizral [1985] propose the *Time Warp* mechanism which allows the user to see a steadily advancing virtual time and which allows considerable parallelism and overlap between computations by different processes. The basic idea is that each local process continues computing, locally advancing its own version of time (Local Virtual Time or LVT) until a message arrives which is apparently in its past. Its LVT is then rolled back to the messages receipt time and the computation is restarted and continues forward.

For this to work it must be possible to restore old states of the program and restart. There are a number of ways of the doing this. One is to take periodic snapshots of the entire program and label them with the LVT of when they were taken. Any messages sent by a program during a period which was rolled back must also be undone. This is accomplished by sending *anti-messages* which annihilate with the originals and which may cause further rollbacks of other processes.

A concept central to this scheme is *Global Virtual Time (GVT).* GVT is (approximately) the minimum of the LVTs of the processes in the system. It is easily shown that no process can be forced to roll back earlier than GVT. Thus it can be used for garbage collecting old copies of process snapshots, and for determining when irrevocable actions such as printing a line can be done. A process that wants to print must wait until GVT advances past the time it wanted to do the printing and then do it.

Jade's IPC

This paper describes an implementation of virtual time and Time Warp within an already existing distributed programming environment, Jade [Unger, 1984], [Jade, 1984]. Jade provides a message passing protocol, Jipc [Neal, 1981] which is accessible to a number of high level languages (Ada, C, Lisp, Prolog, and Simula). It is based on the Cheriton's THOTH system [Cheriton, 1979a,b]. Jipc provides facilities for message passing by; send, receive, forward, and reply primitives. These are provided in the form of a blocking send, non-blocking reply and forward, and both blocking and non-blocking receive. Facilities are provided for process creation and destruction as well as process identification and naming. All messages consist of buffers which are sequences of simply typed items (integer, float, character, string). Jade currently runs under UNIX 4.2 bsd and on a number of stand alone workstations.

2 Jade Virtual Time System

The intention of the Jade Virtual Time System is to provide all the facilities of Jipc on top of virtual time implemented using Time Warp. This should enable programs to be written in one of the Jade languages and then run using virtual time with almost no changes. For example, for the purpose of simulating a real time process. The major difference will , of course, have to be the addition of hold statements within the program to advance the local time.

As shown in Figure 1, the Virtual Time System is to be implemented in a number of layers. The outermost layer is the interface to the programmer. It will be provided in a number of the Jade languages, and will appear identical to the standard Jipc implementation language with the addition of primitives for advancing local time.

The next layer deals with saving and restoring local process states. It is separated out so that different implementations can be tried. Three implementations are currently envisioned using: Prolog backtracking; snapshots of complete C programs; and snapshots of user defined sets of variables within a DEMOS-like simulation language [Birtwistle, 1979].

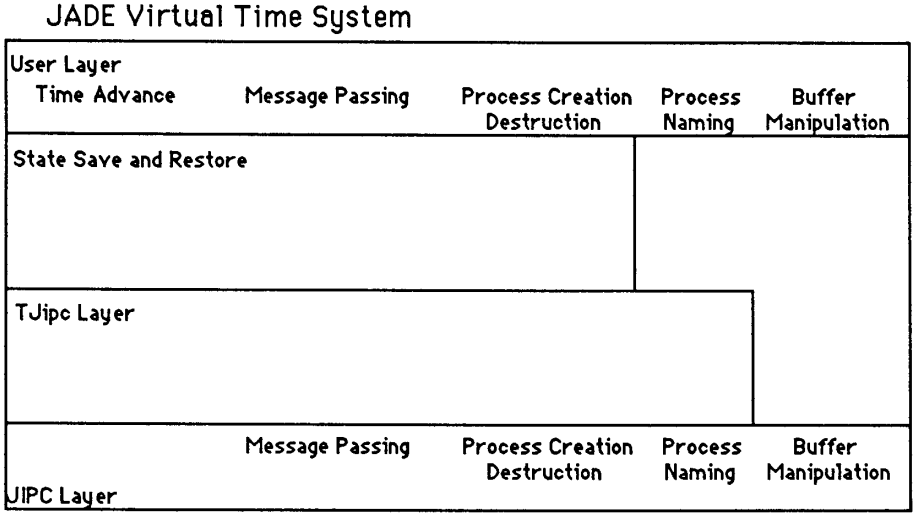

Figure 1 Layers of Virtual Time Implementation

The third layer, called the TJipc system, contains the bulk of the implementation. It provides all of the Time Warp mechanism excluding state saving and restoration. Facilities are provided for queueing messages in virtual time order, detecting when a rollback is necessary, generating and processing anti-messages and computing GVT. TJipc is implemented using Jipc itself.

User Layer

The user layer is an emulation of Jipc itself together with some mechanism for advancing local time. The time advance, message passing, and process creation and destruction are all implemented by calling routines at the state save/restore level. For example when a time advance is done it is necessary to advance the local time to be used in the next snapshot. Similarly, when a message receive is done a rollback may be forced by the arrival of a message at an earlier virtual time. This must be tested for by the rollback mechanism after completion of the receive (pseudo-code for this action is given below).

Some other functions such as finding the name of a process cannot involve any changes in LVT and are implemented directly by TJipc routines. Still other actions such as adding items to a buffer are implemented directly as Jipc routines.

State Save Restore

The next lower layer implements rollback and state saving and calls the TJipc layer to implement maintenance of time as well as message passing. This organization has been chosen so that a number of different state saving mechanisms can be experimented with. For example, T-Concurrent Prolog [Cleary, 1985] is to form the user layer in one implementation. The intention there is to use the underlying Prolog backtracking as a rollback mechanism. This obviously is very language dependent. Snapshots of entire executing programs are also very hardware and operating system dependent. Thus, we did not want to be tied to a single mechanism.

This level is aware of the ebb and flow of virtual time. For example, each snapshot needs to be labeled with the current LVT. LVT is affected from two directions; by calls from the user level to advance local time, and by times returned from the TJipc level. For example a 'hold(Time)' call by the user will cause the local value of LVT to advance and will result in a call to the TJipc routine 'tj_advance_time' to report this change.

A call to receive a message will be implemented by a call to TJipc to do the receive and a check when this has completed on the new value of LVT. This may have advanced because the message was in the receiver's future (in virtual time) or it may have been moved back because of the arrival of a message from the past which should cause a rollback. If a rollback has been caused then an appropriate snapshot has to be selected, the state of the program restored and TJipc informed of the new LVT (there may not have been a snapshot at exactly the right time so an earlier snapshot may have been used). The code for receive is approximately as follows:

```
receive:
        Old_LVT := LVT;
        tj_receive;
        tj_get_time(LVT,GVT);
        if Old_LVT > LVT
        then
                retrieve a snapshot with time ≤ LVT;
                LVT := time of snapshot;
                tj_set_back_time(LVT);
                start executing from old snapshot;
        else
                return
```

'tj_receive' does a receive at the TJipe level. tj_get_time then obtains the new LVT and GVT after the message has been received. If this is in the past then the snapshot is retrieved and LVT set to the snapshot time. Finally TJipe is informed of the new time using tj_set_back_time. All the user level message passing routines as well as process creation and destruction are implemented in this way.

3 Jade Virtual Time Implementation

The bulk of the implementation occurs at the TJipe level. The following sections describe the various data structures and routines seen at the user level and how they are mapped onto Jipe constructs via TJipe.

Buffers

Every message consists of a buffer containing simple items such as integers, floating point numbers, characters, and strings. The buffers at the user level are identical to those in Jipe [Jade, 1984]. So, all the Jipe routines for manipulating buffers can be used directly. TJipe needs to send extra information with each user message. For example the LVT at which it was sent and whether it is an anti-message or not. This is sent as a separate message buffer preceding the one seen by the user. Using two buffers greatly simplifies the process of keeping the semantics of the user and Jipe level identical. For example, if the information about LVT were packed into the front of a buffer then the maximum possible message length seen by the user would differ from pure Jipe.

Processes

In Jipe the world consists of distinct processes with their own names (not necessarily distinct) and process ids (which are unique). Each process at the user level actually corresponds to two Jipe processes. A process in which the users code runs and a separate 'buffer' process which queues messages incoming for the user process. The user processes may share buffer processes. At one extreme there need be only one buffer process in the entire system, at the other extreme there will be one buffer for each user process. In practice it is expected that all the processes on a single CPU will share a single buffer process. That is there will be one buffer process per CPU.

To see why these additional buffer processes are necessary, recall that Jipe sends are blocking. That is, if the destination (receiving) process does not receive the message immediately then the sender blocks until the destination does an appropriate receive. In particular if two processes send to each other simultaneously then they will both be blocked forever in a deadlock. In a normal Jipe system this is a bug and must be expunged. However, in TJipe such a deadlock might be caused by messages at different virtual times, which if they were running in a 'real' situation would never occur simultaneously. Also,

another message might arrive and force one of the processes to roll-back and take a different course. But because it is blocked it will never see the message which causes the rollback. This now is a bug in the virtual time system itself not the users program. The standard technique for solving such problems within blocking send systems is to send not to the process itself but to a buffer process. The buffer receives all incoming messages, as well as requests for new messages from the user process. Provided all sends are directed only to the buffer processes and the buffer processes never do sends and never run out of memory, it is easy to show that deadlock can never occur. Figure 2 shows a typical configuration of processes using this scheme.

A user level send then consists of the following sequence at the TJipc and Jipc levels:

send(To_process_id):
the destination process id is examined and its buffer process
is determined;
a Jipc message is sent to the buffer process giving the LVT
of the sender and the ultimate destination;
a null Jipc reply is returned;
the actual user message buffer is sent to the buffer process;
the buffer process places the message on the appropriate
input queue for the destination (this may involve deleting
a message if an anti-message is present or causing a rollback
in the destination process);
eventually the destination user process will send a Jipc
message to its buffer process requesting the next incoming message;
the users buffer will be dequeued and passed as a reply to the
destination user process;
an extra buffer will be sent to the destination user process
giving the new values of its LVT and GVT.

A number of points are apparent from this sequence. First each message passing action at the user level translates to a number of Jipc messages. Second most of the queueing and handling of matters such as annihilating anti-messages is handled within the buffer processes themselves. The only data maintained within the user processes are copies of the current values of GVT and LVT. No messages or other queues

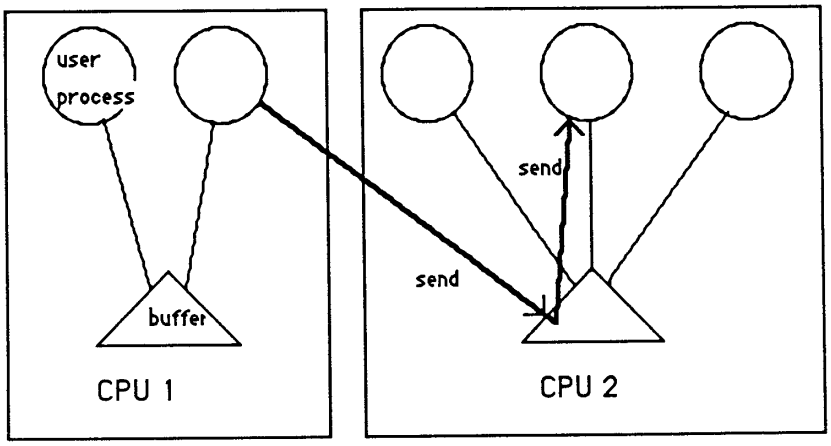

Figure 2 Use of Buffer Processes in TJipc

are maintained within the user process.

The other main function of the buffer processes is the calculation of GVT. In a distributed system this involves a broadcast protocol where all processes LVTs must be checked periodicly [Jefferson, 1983].

Time

Time at the user level is represented by simple integers. However, it is necessary to supplement this at the save/restore and TJipc levels. In the TJipc implementation it is necessary to order the queues in strictly increasing time order (this is to ensure correct rollback and annihilation of messages). Messages which are sent at the same virtual time must be distinguished somehow. This is done by adding a counter to each time. Messages sent by a single process at the same time are sequence numbered in order of generation. All message queueing within TJipc uses the composite of time plus sequence counter.

Servers

An irrevocable action in a Time Warp system, for example printing a character on paper, cannot be done until GVT advances past the time when the action was requested. Then the request will never be rolled back. To accommodate this type of request such actions as printing are handled by 'servers'. The servers receive messages requesting a service and wait until GVT advances before actually executing them. A single TJipc routine 'tj_time_wait()" is provided to allow servers to be constructed. When called it blocks until GVT advances or until LVT is rolled back. A server then receives messages, queues them, and does a tj_time_wait() until the GVT moves past the queued message arrival times when the request can be serviced.

As usual we want to make this as transparent as possible at the user level. In particular, the programmer of a server should not need to be aware of GVT and its advances. A simple wrapper routine which hides the local queueing from the user will do this. The programmer writing the server thinks then that he is writing a simple server which receives messages which can be serviced immediately.

Discussion

Differences from the Jade IPC Protocol

As far as possible the Jade virtual time system emulates Jade and the Jipc protocol exactly. Apart from the provision of time one further facility has been the addition of an asynchronous send primitive. Standard Jipc provides only a blocking send, where the sender blocks until the destination process explicitly does a receive call.

The reasoning here was that there is no correct way of providing a truly non-blocking send. It is always possible that when the send is attempted there will be no space in the system queues to store the buffer. The send would then have to block anyway. The result is the introduction of a system dependent blocking which will occur only very occasionally, making debugging the system more rather than less difficult. The standard way of achieving a non-blocking send in Jipc is for the user write his own buffering process which queues messages.

In TJipc however the buffering processes are already provided as part of the underlying implementation. Also if the system queues are full when a non-blocking send is attempted the message can be sent back as an anti-message. This causes a roll-back and a later attempt to resend the message. So, for economy of user implementation effort and for efficiency the non-blocking send is built into TJipc.

Differences from Jefferson's Virtual Time System

The Jade virtual time system differs in a few points from the system described by Jefferson. As mentioned earlier time is not assumed to advance between message sends, so a number of messages can be sent at the same time. At the TJipc level counters are added to ensure that the time plus counter combination always advances.

If a program is in an infinite loop, the correct semantics of Time Warp require that it still be possible to rollback the program. Without some additional facilities this is not possible in TJipc. The problem is that it is not possible to detect a rollback unless a request has been made to the TJipc buffer process for values of GVT and LVT. If a program is in an infinite loop it will not be making such requests.

This can be solved in interpreters such as Prolog by checking the value of LVT every so many interpreter steps. In other languages it might be possible to put in a timer interrupt which did the check every so often. Both these solutions introduce a significant overhead, and it can be taken as a criticism of the underlying Jipc protocol that a more elegant solution is not possible.

Acknowledgements

We would like to thank the members of the JADE project at the University of Calgary for providing a stimulating environment for this work and for constructive criticisms of this paper. Both the authors and the JADE project have been supported by grants from the Natural Sciences and Engineering Research Council of Canada.

References

Birtwistle, G.M. (1979) *Demos - A System for Discrete Event Modelling On Simula.* MacMillan.

Cheriton, D.R., Malcolm, M.A., Melen, L.S., and Sager, G.R. (1979a) "Thoth: a portable real-time operating system" *Communications of the Association for Computing Machinery, 22* (2) 105-115, February.

Cheriton, D.R. (1979b) "Multi-Process Structuring and the Thoth Operating System" *Ph.D. Thesis, Department of Computer Science,* University of Waterloo.

Cleary, J.G., Unger, B., and Goh, K.S. (1985) "Discrete event simulation in prolog" *Proc. SCS Conference on AI, Graphics, and Simulation,* San Diego, California.

Jade (1984) "Jade User's Manual" Jade Research Report J84/1/1, Department of Computer Science, University of Calgary, September.

Jefferson, D. and Sowizral, H. (1982) "Fast Concurrent Simulation Using the Time Warp Mechanism, Part I: Local Control" Technical Note N-1906AF, The Rand Corporation, Santa Monica, California, December.

Jefferson, D.R. and Sowizral, H. (1983) "Fast Concurrent Simulation Using the Time Warp Mechanism, Part II: Global Control" Technical Note, The Rand Corporation, Santa Monica, California, August.

Jefferson, D.R. and Sowizral, H.A. (1985) "Fast Concurrent Simulation Using the Time Warp Mechanism" *Proc. of the SCS Distributed Simulation Conference,* San Diego, January.

Neal, R., Lomow, G.A., Peterson, M., Unger, B.W., and Witten, I.H. (1984) "Experience with an interprocess communication protocol in a distributed programming environment" *CIPS Session 84 Conference,* Calgary, Alberta, May.

Sowizral, H.A. (1985) "The Time Warp Simulation System and its Performance" *Proc. of the SCS Distributed Simulation Conference,* San Diego, January.

Unger, B.W., Birtwistle, G.M., Cleary, J.G., Hill, D.R., Lomow, G.A., Neal, R., Peterson, M., Witten, I.H., and Wyvill, B.L.M. (1984) "Jade: A Simulation and Software Prototyping Environment" *Proc SCS Conference on Simulation in Strongly Typed Languages,* San Diego, California, February.