Outline

Figures

1 INTRODUCTION

Emerging multi-computer architectures that incorporate hundreds to hundreds of thousands of independently operating computers offer massive increases in computing power. Exploiting these new machines, however, involves writing programs that can be executed as a large number of interacting, asynchronous processes. Currently, this is possible only for very narrowly defined problems. The application of multi-processors to more general problems awaits new approaches to the synchronization of very large numbers of executing program components, or cooperating processes.

One new approach that appears to have great promise is Virtual Time, an optimistic synchronization scheme described by David Jefferson and Henry Sowrizal [Jefferson82] and [Jefferson85]. This document presents our own version of Virtual Time, implemented in the Jade distributed programming environment [Jade85].

Virtual Time imposes on conventional inter-process communication a temporal coordinate system, supplied by user processes, as the basic means of synchronization. Jefferson's paper proposes the Time Warp mechanism for realizing Virtual Time; it offers a new method for achieving high concurrency, free of deadlock and starvation, that suggests a re-thinking of synchronization in distributed systems.

The architecture of our Virtual Time system, called Tipc, consists of several levels. This paper is mainly concerned with the function, design, and implementation of the kernel level, although some issues in the design of an outer level are also presented. A key goal of our design is to provide a multi-lingual implementation of Virtual Time. The same kernel will be used for different programming languages; language specific issues are handled in the aforementioned outer level. An outline of these system levels, including the Jade level, is given in the rest of this chapter. Chapter Two addresses the issues of inter-process communication protocols and semantics, Chapter Three presents the overall structure and design rationale of our implementation, and Chapter Four describes the implementation in detail. Chapter Five deals with debugging and experimentation, and Chapter Six presents our conclusions.

1.1 The Jade Environment and Jipc

Jade provides an integrated set of tools which support the development of distributed systems. The Jade environment can be described in terms of four functional levels: a hardware level, a kernel level, a programming level, and a prototyping level. The facilities available at each level include those provided at lower levels.

The hardware level consists of a heterogeneous network of computers and network links. Vaxes and Suns running Unix 4.2BSD, connected by 10 Mbs Pronet and Ethernet, provide the host resources. The Jade workstations connect to the hosts via a 1 Mbs Omninet. There is no local disk storage for workstations, so booting and downloading are done from the hosts.

The Jade kernel level consists of Jipc, the Jade Inter-Process Communication facility. Jipc is a synchronous message-passing protocol supported among processes residing on any of the Vaxes or workstations. Jipc is based on the Thoth [Cheriton79] communications protocol which includes blocking "send", "receive", "receive_any", and non-blocking "forward" and "reply" primitives. Some of the differences between Jipc and Thoth are a result of the requirements for the Unix version, for example, the support of Unix processes which can enter and leave a Jipc system. Although Thoth supports shared

memory for a "team" of processes, Jipc processes cannot share memory: all interaction among processes must be done by passing messages.

A Jipc message contains a sequence of typed data elements. The types supported include: integer, real, character, string, atom, byte-block, and process id. The basic communication sequence is illustrated in Figure 1. Two message transfers are involved in this interaction, one for the send and one for the reply. Interfaces to Jipc are available for five programming languages: Ada, C, Lisp, Prolog, and Simula. The Tipc system is implemented in the C interface to Jipc on the workstations and the hosts. Later, Tipc interfaces could be built for other languages.

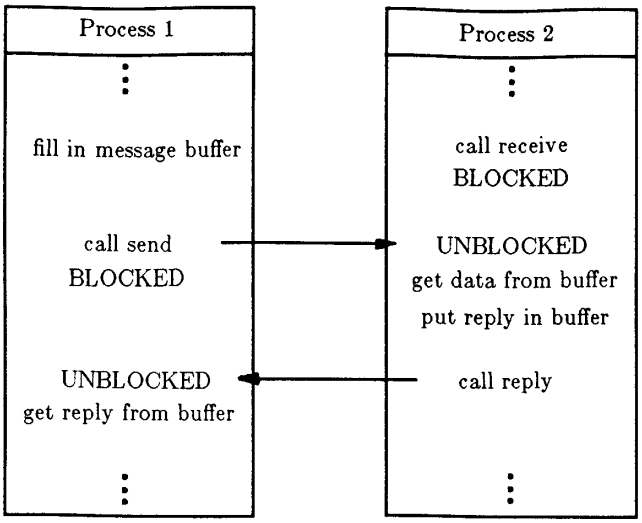


Figure 1. A Jipc Message Interaction.

The tools provided at the programming level include a hierarchical, recursive 2 1/2 dimensional graphics package, a cross compiler for the Jade workstation, downloading facilities, and a window system that has a program interface. A monitoring system is used to observe the interactions among Jipc processes; there are tools for textual traces, graphical animation of inter-process events, and control of execution order [Joyce86].

The prototyping level supports the modelling and simulation of target distributed systems, including the target embedded computer system. Execution of the distributed programs under development by the modelled target system can be simulated, including the portions of the system that are external to the embedded computer system. Performance information can be collected, displayed, and manipulated interactively. Prototyping in Jade has been described in [Lomow & Unger 85].

1.2 Virtual Time and Time Warp

Proposals have been made to coordinate distributed systems by defining a system of clocks, logical or physical, and labelling the events in a distributed system with timestamps, i.e., values from these clocks [Lamport 78]. Then the events in the distributed system must occur in increasing time order, i.e., a process cannot proceed to its next event until it knows that no other event with an earlier timestamp can occur. This often leads to complicated algorithms for synchronization, large amounts of communication, and less concurrency because processes must wait until they have at least one message from each potential sender process. These methods have been called "conservative" synchronization schemes. The work of Peacock, Wong, and Manning [Peacock79] and Chandy and Misra [Chandy81] are representative of this approach.

Virtual Time is a scheme that executes processes in coordination with an imaginary global virtual clock that ticks virtual time. Virtual Time provides a temporal coordinate system used to measure computational progress and define synchronization. A virtual clock is similar to Lamport's clock system [Lamport78] in that every event is labelled with a clock value from a totally ordered set (the virtual time scale). The difference is that there is an underlying mechanism which makes all the events appear to occur in increasing time order despite the fact that, in real time, some event with a later timestamp may occur before one with an earlier timestamp. From the program's point of view, however, events will occur in the correct order.

One mechanism for Virtual Time is called Time Warp, which relies upon process lookahead and rollback as the fundamental way to force events to be processed in non-decreasing virtual time order. In such a system each process is constantly gambling that no event will occur with an earlier time stamp than that of its next event. If the process loses that gamble, it will be forced to roll back to deal with the late event, otherwise it proceeds ahead without waiting. This "optimistic" synchronization scheme provides greater potential concurrency while maintaining an appropriate ordering of events.

In Time Warp, each process maintains its own virtual time, known as the Local Virtual Time (LVT). As the system executes, process states are saved; when a process loses its gamble and is forced to roll back, a previous state is restored and all intermediate actions affecting other processes are undone by the use of anti-messages. The Global Virtual Time (GVT) measures the progress of the system. It represents the minimum of the times of all processes and events in the system, and so may be taken as a steadily advancing "floor" on the computation. No process can roll back previous to GVT; therefore, space taken up by copies of obsolete states can be incrementally reclaimed.

1.3 A Time Warp Implementation on Jade - Tipc

This document describes a new, potentially multi-lingual, implementation of Time Warp that is built on the Jipc [Jade85] synchronous protocol. Tipc is a version of Time Warp that provides essentially all of the Jipc primitives, plus additional primitives for manipulating virtual time. Tipc is implemented in two levels, a kernel level, called the T_kernel, and a Save_Restore level, that supports the saving and restoration of process states in different programming languages and for different computing hardware.

The T_kernel level provides the following:

1. Facilities for process management, i.e., to create, kill, start, and terminate processes.

2. Facilities for processes to communicate via primitives for sending, receiving, and replying to

messages. Facilities are also provided for manipulating messages.

3. Facilities for processes to supply and to interrogate the values of their virtual clocks.

4. The computation of GVT, which is defined as the lower bound of all virtual times shown by all virtual clocks.

5. Information about each process, including process state information and queues of input and output messages.

6. Part of the process rollback mechanism, including primitives for processes that cannot be rolled back. The rollback mechanism is divided between the T_kernel and Save_Restore levels.

7. Mechanisms for process scheduling, error handling, memory management and flow control.

The T_kernel level provides a Jipc-like inter-process communication protocol, called Tjipc, in which all messages are implicitly labelled with a virtual time. Tjipc also supplies primitives for specifying or interrogating virtual time.

The major functions of the Save_Restore level are to:

1. Provide user process state saving and restoration, the difficult part of rollback. User process states are saved at certain points and restored when rollback is required. (These functions will be different for different machines and languages.)

2. Hide the Time Warp mechanism from user processes, so that they are not aware that execution may have been repeatedly rolled back and brought forward.

3. Provide services for handling irrevocable events, e.g., input/output to a device such as a CRT screen or a printer.

Splitting Tipc into the T_kernel and Save_Restore levels was one of the major decisions made. To save user process state efficiently is a difficult job, dependent on the machine and operating system on which Tipc is running as well as the language in which Tipc processes are written. Since Jade is a heterogeneous and multi-lingual environment, we have a Time Warp system with a T_kernel implemented in C for all machines in the Jade environment. On top of the T_kernel, there is the Save_Restore layer which takes care of user process state saving and restoration (which may be implemented in different languages, thus handling state saving in different ways). The disadvantage of this scheme is that the rollback mechanism has to be split between the T_kernel and the Save_Restore layers, making the implementation more difficult.

1.4 Time Warp Applications

The problem of how to generate virtual times is equivalent to the problem of how to coordinate a distributed system: it varies with the requirements of synchronization. In a Time Warp system, virtual time is the resource used to coordinate processes and define synchronization. However, Tipc only guarantees that events will occur, or at least appear to occur, in virtual time order - it does not care

where and how the virtual times are generated. It is up to the programmers of Tipc systems to specify and supply the virtual times. Therefore, systems of virtual clocks must be built on top of Tipc. These clocks can be specified by the programmer or provided in a package. The virtual clock could be a real time clock or derived from it, or a logical clock which ticks in accordance with some logical steps.

Higher level primitives such as procedure calls can be added to languages supporting concurrent processes by using Tipc. The system of virtual clocks may be built into these higher levels or it may be left for the user program to construct and assign clock values values to processes through primitives at the language level.

2 INTER-PROCESS COMMUNICATION PROTOCOLS AND SEMANTICS

The goal of Tipc is to provide an inter-process message passing facility that implements Virtual Time. The Jade inter-process message passing facility, Jipc, represents a synchronous communication protocol capable of supporting Time Warp semantics. Each Tipc primitive maps semantically to a Jipc primitive, with the difference that Time Warp semantics are also transparently provided in the Tipc primitive. Tipc also includes some primitives for manipulating and interrogating virtual time.

2.1 The Tipc and Jipc Protocols

Tipc provides a uniform communication protocol that resembles the Jipc protocol. The primary difference between the two is that Tipc also encorporates the idea of virtual time, thus it provides primitives, and extra arguments to primitives, for manipulating virtual time. It thus becomes possible to run user programs on either Jipc or Tipc, almost without change to the programs. This can be useful, for example, to debug distributed systems using rollback.

A detailed description of the Jipc protocol is given in [Jade85]. In addition to the standard Jipc primitives, Tipc includes:

1. *An additional argument for process creation.* This argument is used by the caller (the child process's father) to specify the virtual time interval between the time the creation primitive is called and the time the child process starts to exist.

2. *A primitive for advancing a process's virtual time.* Once a process initializes, its virtual time can advance only by it either calling this primitive or by receiving a message with a virtual send time greater than its virtual time when the receive primitive was called.

3. *A primitive that enables a process to read its current virtual time.* This primitive can be used to support simulation or other applications.

4. *A primitive for handling irrevocable actions,* such as printing characters on a CRT screen. This type of action cannot be rolled back and therefore, must be suspended temporarily until global virtual time advances through the time the action is supposed to take place. The caller of this primitive will be blocked until global virtual time advances past the virtual time of the call.

5. *An asynchronous send primitive.* Jipc is a synchronous (blocking) communication protocol which was primarily designed for reliable inter-process communication. The distinguishing feature of Tipc is its reliance on process lookahead and rollback as the fundamental synchronization mechanism, thus it is no longer necessary to use process blocking for synchronization. More important is that a blocking send will reduce concurrency (high concurrency is one of the major design goals of a Virtual Time system). A blocking send always leads to the waiting of a process.

6. *A primitive for remote process creation.* This was added to handle the situation in which many processes execute the same code on a single workstation. When the first process is created, the code will be downloaded from the host machine and any subsequent processes executing the same program will share that code. It would be impractical to download the same code each time a process is created because downloading is expensive. This primitive is only valid for Jade workstations.

7. *A primitive for a process to terminate or exit.* When a process has completed its actions, it must call this primitive to terminate itself. This is due to the fact that Tipc schedules user processes on a voluntary basis; there is no way to stop a process unless it relinquishes control to the kernel.

2.2 Tipc Semantics

The Tipc protocol is basically equivalent to Jipc with Virtual Time incorporated. Both the semantics of Jipc and the semantics of Virtual Time are observed. For example, the nonblocking search primitive in Jipc returns the process identification if there is a process, at the time (real time) the call is made, with the same name as the argument, otherwise it returns a null process identification. The analogous primitive in Tipc returns an identification if there is a process which has the given name and started at a virtual time less than or equal to the virtual time at the moment the call is made, and which is still alive at the virtual time the call is made, even though the process might start to exist in real time after the real call time.

As defined in [Jefferson85], the semantics of virtual time are simple. If an event A has a virtual time less than that of event B, then the execution of A and B must be scheduled so that A appears to be completed before B starts. This semantic rule is satisfied if the following conditions are observed.

1. If event A comes before event B in a process, then event A has a virtual time less than that of event B. If the events are the receipt of messages, then all messages directed to a process are processed in non-decreasing virtual receive time order and all messages sent out by a process are sent in non-decreasing virtual time order.

2. The virtual send time of a message is less than or equal to its virtual receive time.

3. All messages received by a process through any non-selective receive primitives (e.g., receive_any) are processed in nondecreasing send time order.

4. In a process all messages coming from another process are processed in nondecreasing send time order, no matter through what receive primitives the messages are received.

5. If the event is process creation, then the virtual time at which the child process comes to exist is greater than or equal to the virtual time of that event.

6. If one process successfully searches for another, the virtual time at the completion of the search is greater than or equal to the virtual time at which the process being searched for starts to exist.

Messages passed between processes are stamped with their send time, the virtual time at the moment the message is sent. The virtual receive time of a message is the virtual time when the message is actually received by the receiver process. Generally, in a real system it is impossible to specify the virtual time when a message must be received, so the virtual receive time of a message is decided on the receiving end of the communication; it equals the maximum of the virtual send time and the receiver's virtual time at the moment the message is received. This principle is derived from conditions one and two, and it can be generalized. Any "waiting" event in terms of virtual time will result in the increase of the waiting process's virtual time. The waiting process's virtual time at the conclusion of the event will be the virtual time of that event.

Condition one speaks of the "happened before" relation of events in one process. Condition two is about the order of two closely related events: the send and the receive of a message in two processes involved in communication. Similar to condition two, conditions four and five are about the order of two events, one which is the cause and another the consequence. Conditions three and four specify the order of events (the receipt of messages) in one process which are the consequences of events happening in other processes.

Since the receive time of a message is derived from conditions one and two, conditions three and four are necessary for satisfying the semantics of virtual time. For example, suppose process A calls "receive any" at virtual time 90, and a message with send time 100 arrives first and is received by A. According to conditions one and two the message is received at time 100. Then A calls "receive any" again and a message with send time 95 arrives, so it would be received at time 100. Conditions one and two still hold, but the receive order is wrong in terms of virtual time. With condition three, the message with send time 95 would be processed first and the semantics of virtual time satisfied. Without condition four, messages sent by one process and directed to another process could be received not in nondecreasing send time order if the send order is not preserved by the network on the receive end. Again this would conflict with the semantics of virtual time.

Another example shows that the messages directed to a process are not necessarily received in nondecreasing virtual send time order: suppose process A calls "receive from process B" at time 90 and receives a message with send time 100 from process B, thus increasing the virtual time of process A to 100. Meanwhile process C sends a message at time 95; this message will not be received until process A calls "receive from C" or "receive any" afterwards. In this example, in our implementation, the message with earlier send time is received later because the receiver desired to receive a specific message first.

2.3 Limitations of the Implementation

Certain limitations are present in Tipc that result from the requirement to only use Jipc for communication between processes. These include the lack of a facility to interrupt a running Tipc process, and potential deadlock due to the use of Jipc's blocking send and receive calls.

Other difficulties result from splitting the rollback mechanism between the T_kernel and Save_Restore levels. Maintaining consistency between the two levels makes the connection between levels complicated, because some feedback is required from the Save_Restore level to the T_kernel level.

3 SYSTEM ARCHITECTURE

Tipc is implemented in two levels, the T_kernel, which resides on each machine, or node, and the Save_Restore level. The T_kernel comprises several system processes and a subroutine package. The Save_Restore level provides access to state saving and restoration routines. The routines in the Tipc subroutine package are a combination of their Tjipc counterparts and the state saving and restoration procedures.

Tipc can be executed on any number of nodes provided that each node supports Jipc and the T_kernel. The T_kernel will connect these nodes into a logical ring that is used in the computation of GVT. An overview of the Tipc architecture, along with the rationale for the major design decisions is presented in this chapter.

3.1 The System Structure on a Single Node

A Tipc system consists of several levels, i.e., User, Tipc, and Jipc, as shown in Figure 2. In the figure, the bold box represents a process, the directed lines indicate an invocation, and the directed double lines show the directions in which Jipc messages are sent.

The T_kernel Level

The T_kernel level consists of a Local Control Process (LCP), a package of routines providing T_kernel primitives (Tjipc), and the Creator, a system process responsible for process creation. There is one LCP process and one Creator process per machine. The user process interacts with the LCP by calling routines in the Tjipc subroutine package.

One of the main functions of the LCP is to provide input message buffering for all user processes on its machine. There are two reasons for this. First, and most important, is that it is a common situation for a user process that some messages "in the future" arrive because their senders have proceeded ahead in time, while some messages arrive "from the past" because their senders have lagged. Only an intermediate process can provide the buffering for future messages and keep the historical record for past messages. If a message from the past does arrive, some processed messages have to be "unreceived" and the process has to roll back to a point in time previous to the time of the message from the past. All messages with virtual send time equal to or greater than GVT are kept in an input queue maintained by the LCP. Second, since Tipc has an asynchronous (nonblocking) send, an intermediate process is necessary to provide message buffering.

Another function of the LCP is to maintain information on processes and to schedule processes. In any scheme, a kernel process is needed. In this design, a single LCP is adopted instead of one buffering process for each user process plus another kernel process. Demanding large amounts of space is one of the problems with Time Warp systems. The scheme of multiple buffering processes would make the space problem even worse and would unnecessarily increase system overhead on process swapping and communication.

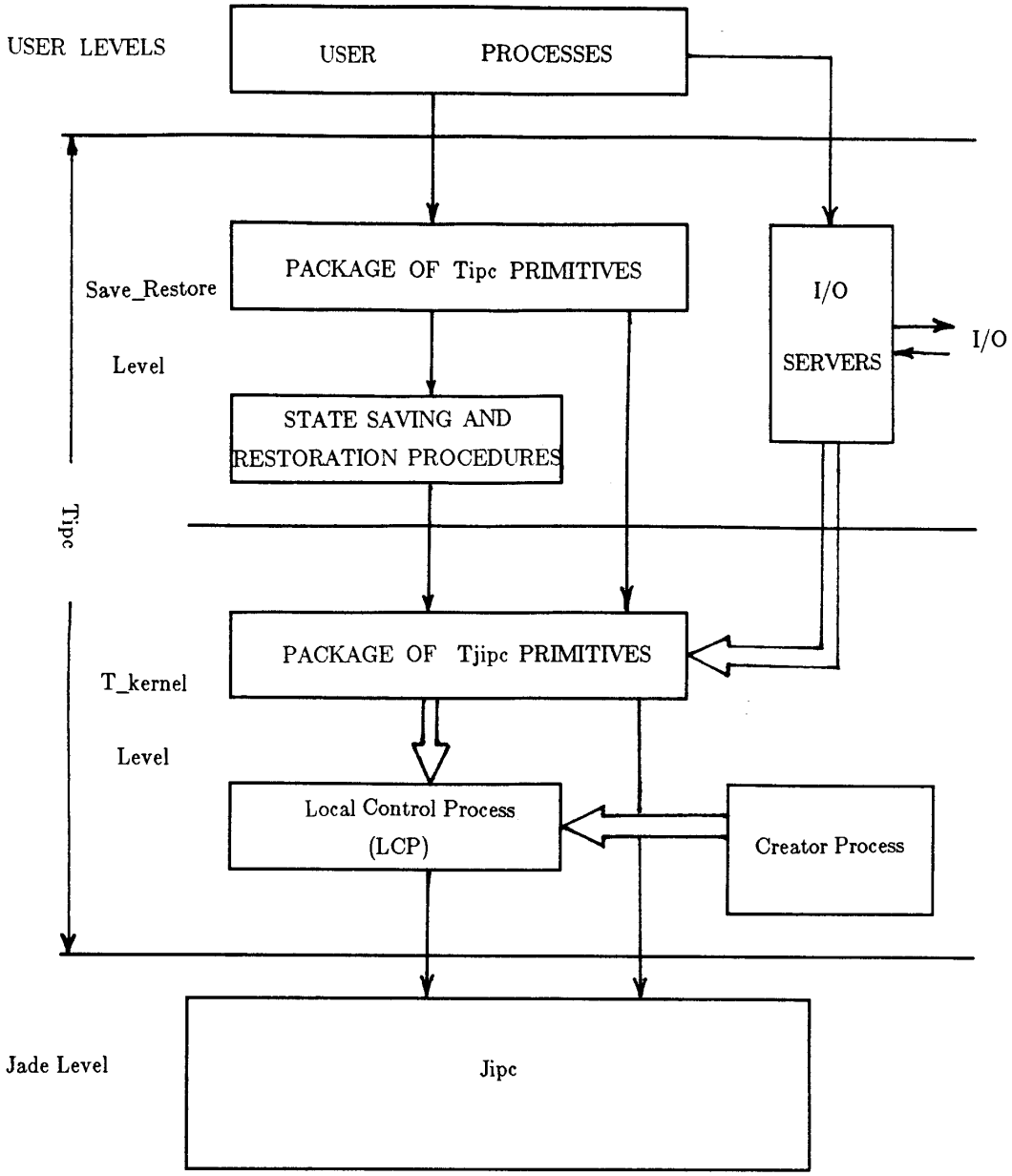


Figure 2. Tipc System Levels On A Single Site.

It is necessary to save all the output messages (those sent by a process) with virtual send time equal to or greater than global virtual time in an output queue for each user process because the user process may roll back and some messages sent previously may have to be cancelled by sending anti-messages. The output queues are maintained by the Tjipc package, i.e., in the user processes themselves, so that each user process handles its own Output Queue. We admit that it is more desirable to keep both input and output queues in the LCP for easy access. Unfortunately, this is impossible. If the output queues were maintained in the LCP, the LCP would need to send messages on behalf of user processes. But the LCP is not able to send messages to other processes which may also send to it (e.g., other LCPs), otherwise the LCPs would fall into deadlock. Also, extra message passing would be involved.

All the T_kernel primitives are implemented in the Tjipc subroutine package, providing a timestamped communication facility between Tipc processes. Within all Tjipc routines, except one for getting the LVT, eventually a Jipc message of some type, called the local control message (LCM), is sent to the LCP to notify it of the event, e.g., sending a message or presenting a request to receive a message. The calling process will then be blocked until it gets the Jipc reply message from the LCP. In the meantime, any other process may resume execution, depending on the scheduling strategy adopted by the LCP.

The Creator is a TW system server process which accepts requests from the LCP to create user processes. Any process creation request from a user process becomes a message directed to the Creator so that process creation is treated as a message event, which is beneficial to a consistent rollback mechanism. As well, it puts the burden of process creation on a dedicated process, freeing the T_kernel from that time consuming task.

The Save_Restore Level

The main function of this level is to implement user state saving and restoration. It also provides higher level services for irrevocable actions such as I/O operations. Two procedures are available:

1. *The state saving procedure* is called whenever user state needs to be saved. By the user state we mean all the data area of a user process including the program counter, the stack area, the stack pointer, the values of registers, global variables, and all dynamically allocated memory.

2. *The check and restoration procedure* is called after most Tjipc primitives to check if rollback is required. This is done by getting the newest virtual time of the caller and comparing it with the last virtual time, which has been kept by Tipc. If a rollback request is pending (i.e., new_LVT $<$ last_LVT), a previously saved user state with virtual time equal to or less than the newest value will be restored. Once a state is restored, program execution resumes at a point where that state was saved as if the execution had never gone beyond that point. The rollback mechanism is totally hidden from user processes.

Higher level primitives for irrevocable operations are provided through special server processes. These server processes run on top of Tjipc and process requests sent as Tjipc messages. The servers' behaviour

follows the pattern:

```
repeat forever:
{ wait for GVT (global virtual time) to advance by calling a special
  Tjipc primitive for irrevocable operations;

  get all requests (messages) with virtual send time less than GVT;

  service the requests;
}
```

The point is that rollbacks can't occur for processes providing irrevocable services. The servers can also be viewed as the link between the processes in the Tipc system and the outside world. More than one server may exist to provide different services.

3.2 The System Structure Across Multiple Nodes

The above processes and functions are replicated on each node of a multi-processor, or on each individual computer in a network. The entire Tipc system is built on top of Jipc, there being no other way to communicate between processes. All nodes in a Tipc system form a logical ring around which the message for estimating GVT circulates. The head of the ring is a system process called the Global Controller, which can reside on any machine. In this process and in the LCP on each machine in the ring, a pointer variable points to the next member (an LCP) of the ring. On system startup, the pointer in the global controller points to itself, since the ring is initially empty. When the first Tipc user process comes into existence on a machine not already in the ring, the LCP on that machine will send a message to the global controller and join the ring. After the last Tipc user process actually dies on a machine, the LCP sends another message to the global controller and leaves the ring. The ring may become empty again; this can be a perfect indication of the termination of the whole system.

Figure 3 shows the distributed structure of a Tipc system. Small circles represent processes while the two large concentric circles delimit the various levels. Directed lines show the directions in which messages are sent. All messages among processes in this system are Jipc messages.

A notable feature of the structure is that the LCPs, except in one situation, only receive messages. If two LCPs try to send messages to each other simultaneously, or more precisely, if LCP A sends a message to LCP B before A receives a message from B, and the message reaches B after B sends its message to A, these two LCPs will fall into deadlock since each is waiting for a reply from the other. Because of the possibility of deadlock, the LCP is constantly waiting for any messages directed to it. On receiving a message, it sends back a reply as quickly as possible, does whatever processing is necessary, then goes back to wait for the next message. It never blocks waiting for a particular message, so it never falls into deadlock.

The only exception is the GVT message, the message circulating around the ring in a fixed direction for the estimation of GVT. The LCP receives the GVT message, replies immediately, calculates the lowest local virtual time on its machine, then sends the message on to the next LCP in the ring. Although the GVT message path forms a closed circle, there is only one GVT message at any given time, so it cannot cause deadlock.

Besides the GVT message, the LCP expects two other types of messages.

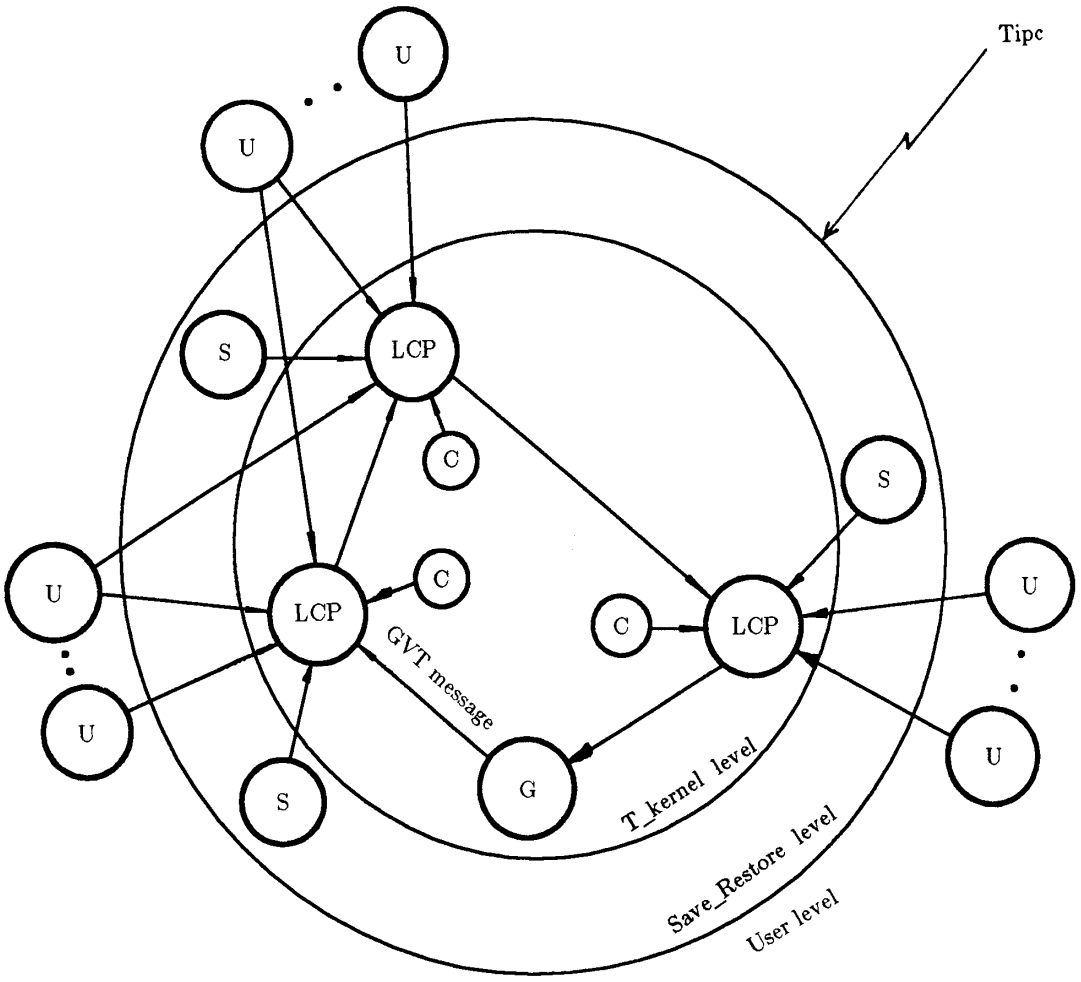


Figure 3. The Distributed Structure Of Tipc.

U - User Process
S - Server Process
C - Creator
LCP - Local Control Process
G - Global Controller

1. *Messages for communication among user processes,* which are buffered by the LCP, if necessary, or request messages for such things as process creation or interrogation. Usually, the LCP sends a reply for this type of message immediately.

2. *Messages from local user processes to request a message or to notify the LCP that some event has happened in the user process.* We call this type of message a Local Control Message. The LCP does not reply to this message until it is the calling process's turn to resume execution, thus giving the LCP scheduling control.

3.3 The Tipc Inter-Process Communication Sequence

An interaction between Tipc processes is realized by a series of Jipc messages. A message is not directly sent to the destination Tipc process, instead it is sent to the LCP on the machine on which the destination process resides. The message is buffered by that LCP. When the destination process wants to receive a message, it sends a local control message of type "receive" or "receive-any" to its LCP whereupon the LCP will pass the message back to it by a Jipc reply message.

Figure 4 gives two typical communication sequences. In this interaction process one sends a message to process two. A series of Jipc messages are exchanged as follows: For the nonblocking send, process one sends the real message to the LCP on the same machine as process two and gets replied to immediately, then sends a local control message of type "send" to notify its LCP of the send. So that process one can continue executing, its LCP will reply to it immediately, provided that there is no other user process ready to run with an earlier virtual time. Otherwise, process one will be suspended until it becomes the process with the lowest virtual time. The real message (labelled with the receiver and its virtual send time) received by the destination LCP is put in the input queue of process two. When process two sends a local message of type "receive" to get the message, the LCP passes the real message to process two in its reply. Also, the LCP retains a copy of the message in the input queue for process two, stamped with the virtual receive time and marked with "received" to distinguish it from unprocessed messages. There is a potential wait for process two.

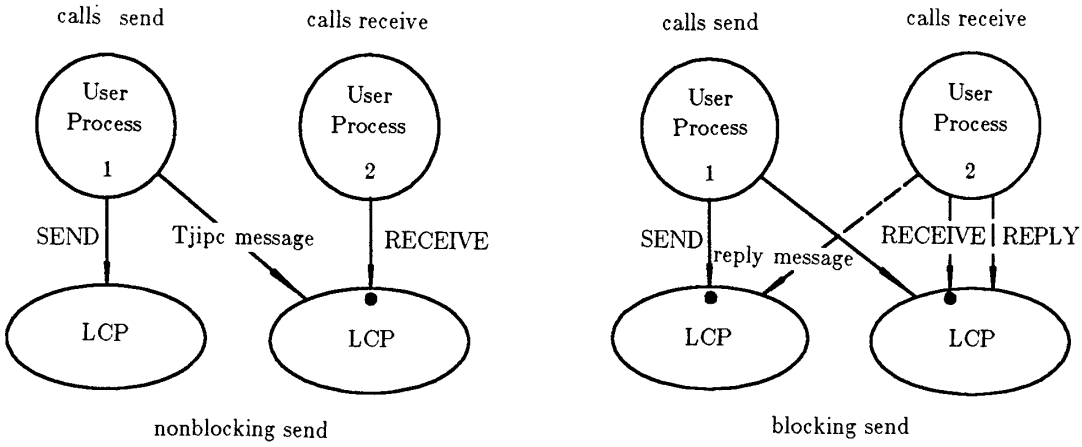


Figure 4. Interactions Between Tipc Processes.

The sequence for a blocking send is similar to the nonblocking send except that process one goes blocked waiting for the reply message from its LCP and a reply sequence is necessary. In fact, the reply sequence is similar to the send sequence except that the local message of type "send" is replaced by one of type "reply". The reply message is also copied into the input queue of process one in case rollback causes the reply message to be unreceived.

4 SYSTEM IMPLEMENTATION

This chapter is concerned with the internal structure of the Tipc system. The first five sections deal with local control on a node, and the following three sections cover the global control of the entire system. The last section presents one implementation of state saving and restoration.

4.1 T_kernel Data Structures

The T_kernel maintains information on Tipc user processes. Part of this information belongs to the LCP and part to the user process. Since this system is based on message passing, each part is accessible only to the process which owns it. The Process Control Block (PCB) and the input message queue for each user process are held in the LCP. The output queues are maintained by the Tjipc routine package (i.e., each Tipc process owns its output queue), but they are system maintained and not manipulable by the user. The Process Data (PD), system maintained user process data, are also held in the Tjipc package.

The only exception is that Tjipc contains the current process number, an integer variable of the LCP, which contains the index number to the PCB and the PD of the process currently in execution. For the workstation version of Tjipc, due to limited memory space, the Tjipc package must be reentrant code so that it can be shared by all user processes on the same machine. By making the current process number accessible (read only) to user processes, it became trivial to write reentrant code for the Tjipc package. This variable is used by Tjipc routines as an index to the PD and the output queue of the calling user process so that the user process accesses only on its own data area in the shared reentrant code.

The Tipc Process Id

Before one Tipc process can communicate with another, it must obtain the process id of the process it wishes to talk with. A process id uniquely identifies a Tipc process. Because a Tipc process is also a Jipc process and messages are buffered by the LCP (another Jipc process) before finally reaching the destination processes, a Tipc process id consists of

1. The Jipc process id of the process,

2. The Jipc process id of the LCP on the same machine as the process,

3. The local process number. (The local number is used for quick access to system maintained information about the process.)

A Tipc process sees the Tipc id as a predefined type of data: t_process_id. Given the name of a Tipc process, search primitives provided by Tipc can be used to acquire the Tipc id of that process. See

[Jade85] for the components of a Jipc id.

The Format of a Tipc Message

A transaction between Tipc processes is realized by a series of Jipc messages, as described in Section 3.3. A Tipc message is translated into two Jipc messages, the "header" and the "data". The header message contains the information needed by the T_kernel to properly handle the semantics of the message passing primitive being used. The data message travels from the sender process to the receiver's LCP; it carries the information the sender wants to pass onto the receiver. If the Jipc message format could accommodate the Tipc header message in the header of the Jipc message, it would not be necessary to carry the header in a separate message. The format of a header message is given in Figure 5.

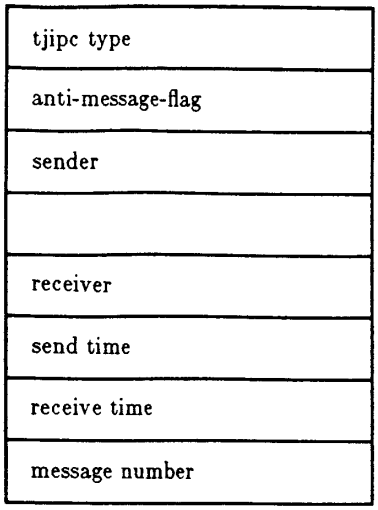


Figure 5. The Tipc Header Message Format.

There are several types of Tipc messages, differentiated by the first item in the header message. On receiving a Tipc header message, the LCP applies different processing procedures according to the type of the message. Following are the components of a header message. The "receiver" refers to the destination user process and the "sender" the process which sends the message. The "send time" is the local virtual time of the sender at the moment the message is sent. The "receive time" is the virtual time when the message is received by the receiver; it is assigned by the LCP on receiving the "receive" type local control message from the receiver. The message number is assigned by the sender and is unique among all messages coming from that sender. The "anti-message-flag" distinguishes the negative Tjipc message (or anti-message) from the positive Tjipc message (normal message). Anti-messages have exactly the same contents as their opposites, except that the anti-message-flag carries a value -1 (instead of +1 for a positive message), and are used in rollback to cancel normal messages sent previously. The data message contains whatever the sender process puts in it. Like an ordinary Jipc message, the information in a data

message is a sequence of typed items.

The Process Control Block and the Ready Queue

The LCP keeps track of information about application processes by maintaining an array of Process Control Blocks (PCBs). When a process initializes, or enters a Tipc system, an initialization message will be sent to the LCP by the "initialize" or "enter_system" routine and the LCP will assign a PCB to the new process. This PCB will contain all information on the process throughout the time the process remains in the Tipc system. Figure 6 lists the items in a PCB. A general description of the role the PCB plays is given below, details will be discussed where appropriate.

The "Jipc id" item contains the Jipc process id of the user process. The "index number" is used as the local process number of the process. All processes ready to execute have the "scheduled" flag in their PCB set and are linked together by two pointers in the PCB, the "successor" and the "predecessor", to form the Ready Queue, a two way linked list. The "successor" points to the next PCB, or NULL (for the last PCB in the queue). The "predecessor" points to the previous PCB, or NULL (for the first PCB). A pointer variable in the LCP points to the first PCB in the Ready Queue if there is at least one process ready for execution, otherwise it points to NULL.

PCBs are inserted into the queue in increasing local virtual time order. When a process has completed an event and notifies the LCP, the dispatcher routine of the LCP will wake the process up whose PCB is the first member in the Ready Queue, so that the process can resume its execution. The LCP does this by replying to the user process. The reply contains necessary data (notably the LVT and the data message sent from the other process). The reply message is handled by the Tjipc routines in the user process. Again, for Tipc, message passing is the only way for processes to exchange information.

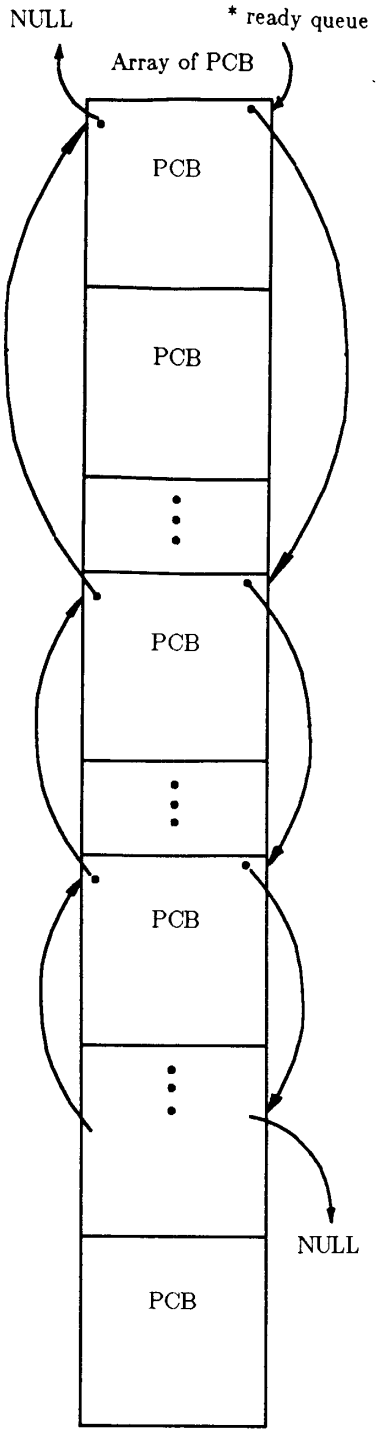
The "start time" contains the virtual time the Tjipc process was created or entered into the Tipc system. It equals the virtual time that the "create" or "enter" primitive was called plus the time interval passed as parameter to the primitive. "LVT" keeps the current LVT of the process and "last LVT" the virtual time of the last event in the process. The item "status" is probably the most important, it shows the current state of the user process. These states fall into two categories; a process may be either out of the Ready Queue waiting for a certain event, or in the Ready Queue waiting to execute. From the point of view of the LCP, a Tipc process may be in one of the process states detailed next.

The States of a Process

Two ready states are introduced so that the LCP can send a reply message containing a different type of data for each case.

1. *INIT:* The process has been created, or has entered into the Tipc system and initialization of its PCB and PD has completed. It is either in the Ready Queue waiting for its turn to start executing or is already executing.

2. *READY:* A process is in this state if it is not waiting for any events and is ready to resume executing. If a rollback condition for a process is detected by the LCP, that process will immediately be put in the Ready Queue, in this state, with the process's LVT set to the time it has to roll back to. It doesn't matter if the process was ready or waiting before the rollback request was encountered. When the time comes for the process to run, the reply message sent to it by the LCP will contain the LVT value less than the one kept in its PD in the Tjipc package.



* successor
* predecessor
start time
last LVT
LVT
kill time
rollback mark
status
scheduled
* current message
from (sender)
local process number being searched
* anti-in-messages
Jipc ID
local process number

Figure 6. The Process Control Block Format And Ready Queue Structure.

As mentioned earlier, this is the only way for the Tjipc routines to detect a pending rollback request.

A user process will stay in one of the above two states until it calls the next Tjipc primitive in its code. The rest are states of waiting for different events:

3. *RECEIVE:* A process enters this state when it calls the Tipc primitive to receive a message from a specific process and the message hasn't arrived yet. The "from" field in its PCB will contain the process id of the expected sender process. The LCP can recognize the expected message (when it comes) by the receiver and sender process ids (contained in the header message). When the message arrives, the waiting process will be put in the Ready Queue in the READY state with the "current message" field in its PCB pointing to the message so that the message can be passed to the receiver. The LVT field in the PCB is set to the virtual send time of the message if the send time is greater than the original LVT of the receiver.

4. *RECEIVE_ANY:* A process will be in this state when it calls the Tipc primitive to receive any message directed to it and no message has arrived at the time the call is made. Upon the arrival of any message directed to it, the state of the process changes to READY just like in the case of RECEIVE.

5. *REPLY:* A process which calls the Tipc blocking send primitive will be in this state. The "from" field in its PCB will contain the process id of the process the message is sent to. When the reply arrives, the state changes to READY, with the "current message" field pointing to the reply message. The LVT field is updated if the virtual time of the reply is greater than the original LVT.

6. *CREATE:* After a process calls any creation primitive, it stays in this state waiting for a reply as to whether the creation was successful or not and, if successful, the process id of the child process. The reply will come from the system Creator process which is the process the creation request message (a type of Tjipc message) is sent to by the Tjipc package. If the Creator successfully creates the child process, it passes the child's start time and the parent's id to the child by a Jipc message, and gets the child's local process number in the reply from the child. Then the Creator is ready to send a reply to the creation request message. This reply goes to the parent's input message queue in the LCP. When the reply arrives from the Creator, the parent will be woken up and put into the READY state. Its LVT will remain the same and the "current message" field points to the reply message from the Creator.

7. *SEARCHW:* A process which calls the routine to search for a process, and is willing to wait until it comes into existence, goes into this state if the searched for process does not already exist. It will stay in this state until the process being searched for comes into existence. Then the latter will send a message of some type to wake the searcher up. (The message is directed to the LCP on the machine of the searching process.) The LVT field in the searcher's PCB will be updated to the virtual start time of the searched-for process if the original LVT is less than that start time. The "from" and "index of being searched" in the searcher's PCB are used to hold the Jipc id and local number of the process being searched for.

8. *LEAVE:* By calling the leave primitive, a process can leave a Tjipc system. The process will not actually leave the system until GVT advances past the LVT at which the process wants to leave. In the mean time, the process stays in this state in case has to roll back. After the process leaves, its PCB will be released.

9. *DYING:* When the execution of a process finishes, it must call the exit primitive to notify the

LCP. The LCP will put the calling process in this state until GVT advances past the LVT of the process at the time of the call.

10. *GVT:* When a process calls the primitive which waits for GVT to advance, it is suspended in this state until GVT advances. The LVT of the process will be updated to the new advanced GVT if the original LVT is less than the new GVT. Usually, only server processes that are created for handling irrevocable operations may be in this state.

11. *HOLD:* This state is used by the LCP to control message flow. When a process tries to send a message, it might be suspended in this state if one of the following situations arise: the machine on which the receiver of the message resides is running out of memory; or the machine on which the sender process resides is running out of memory. If the process encounters one of these situations, it will stay in the HOLD state until GVT advances, then it will try again. It may still be suspended unless enough memory space is recovered through garbage collection.

The Input Queue

The LCP maintains an input message queue for each local user process. Figure 7 is an example of an input message queue. A two way linked list is used for the input queue to facilitate searching. The input queue is sorted in order of virtual send time. Each entry in the queue consists of eight fields plus the message text, a copy of the data message. The "receive time" field is left unfilled (with zero value in this implementation) until the message is actually received. Each incoming message goes into the input queue before it can be received by the Tipc process. The "antimsg" flag is used to distinguish normal messages and anti-messages. If there is a pair of opposite messages in the queue, they will annihilate each other. However, if the normal message has already been processed, the receiver process must roll back. The message remains in the PEND (pending) state until it is received (processed) by the user process. The "receive state" field represents how the user process obtained the message and is an important piece of information for the detection of rollback situations (see Section 4.4). A processed message can be identified by one of the following six receive states:

1. RCVANY - A non-selective receive primitive has been called.

2. RCV - A selective receive primitive has been called.

3. SEND - The receiver of a blocking send message has returned a reply message.

4. MAYRCV - No message from a specified process with a send time equal to or less than the caller's LVT was pending at the time the maybe-receive primitive was called.

5. MAYANY - No message with a send time equal to or less than LVT was pending at the time the maybe-receive-any primitive was called.

6. CREAT - The creator process has replied to a process creation request.

Messages of state MAYRCV or MAYANY are dummy messages. They are not real messages but created when the maybe-receive or maybe-receive-any call is made in the user process, and no message with a send time no later than the LVT is available at that time. If corresponding messages show up later, the dummy messages can be used to determine the right receive time, or the time that the receiver must roll back to. The receive time of a dummy message is the calling process's current virtual time. If the

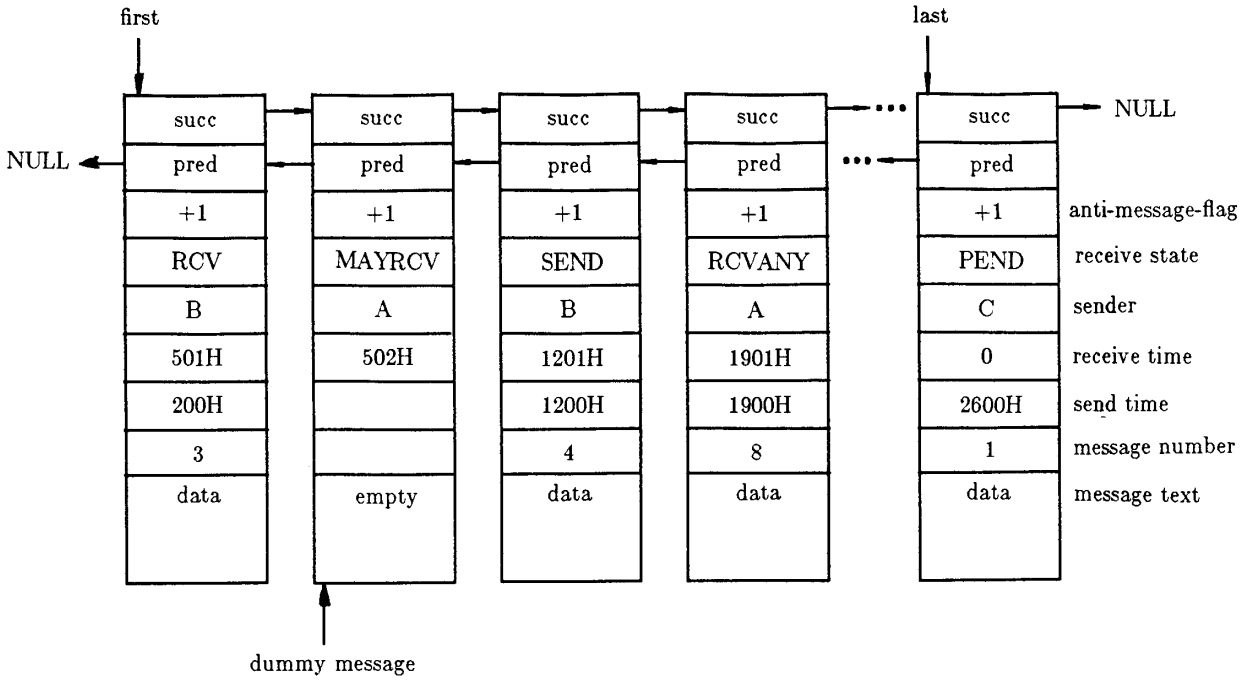


Figure 7. The Input Message Queue.

"maybe-receive" call succeeds, no dummy message is created.

Since more than one message can be received at the same virtual time, a "counter" is introduced into the representation of virtual time to record the order in which messages are received. Four bytes (a C long integer) are used to hold the virtual time variable: the first three bytes represent real virtual time and the last byte is the counter. The real virtual time is what user processes see, so the virtual time variable's value is 256 times larger than the real virtual time. As a matter of fact, the counter is used to count not only the receive events but all other events happening at the same virtual time. When the virtual time increases, the counter is cleared and starts from zero again. If the counter byte of the LVT is equal to zero, then no event happened at the time specified by the first three bytes of the LVT. If it is n, then n messages have been received (including dummy messages) at the same time providing no events other than message receipts occurred. This makes the receive time unique: no two messages directed to the same process have the same virtual receive time, which prevents some subtle problems from arising.

The send time of a message is the current LVT of the sender with the counter byte cleared, for the counter is only of use to its owner process. The receive time of a message equals one plus the maximum of the send time of the message and the current LVT of the receiver. Not considering rollback situations, the LVT of a receiver process is updated to the receive time each time a message is received (i.e., the receive time always increments by at least one). From now on, whenever "virtual time" is mentioned, it refers to the real virtual time represented by first three bytes of the LVT while "LVT" includes the counter byte. It would be wise to extend the counter from local to global and this change may be made in a future version of Tipc.

When a user process rolls back to an earlier virtual time, all the messages in its input queue with a receive time equal to or greater than the earlier time are "unreceived" by setting their "state" field back to a pending state and the "receive time" field to zero. Messages in the input queue are not necessarily in increasing receive time order. It is possible that a message with an earlier send time is received after one with a later send time, if a selective receive primitive is called.

The Output Queue

Output message queues are maintained in the user processes by the Tjipc package. An output queue keeps copies of recent messages sent by the user process. It is not a buffering queue like the input queues in the LCP, but is used merely to undo message sending by sending anti-messages when rollback occurs. Figure 8 gives an example of the output queue.

The output queue has a structure similar to the input queue. For each message sent out, a copy of it consisting of some information fields plus the data message is made and inserted into the queue in increasing send time order. The virtual send time of a message is the current virtual time at which the message was sent plus one (which means increasing the counter byte by 1). As described above, the counter byte of the LVT can be used to record the order in which the messages were sent at the same virtual time. Figure 8 shows that message #1 was sent at virtual time 100H (hexadecimal) with the counter byte equal to 1 (meaning that it is the first event for that time), so the send time, i.e., the LVT of the message, is 101H. Message #2 was sent at LVT 301H and message #3 at 303H. Between these two send events there was probably a receive event at LVT 302H. Altogether three events happened at virtual time 300H.

The "Tipc type" field contains the type of the Tipc message. In this example, two types are encountered, SEND, which corresponds to the process calling the synchronous or asynchronous send primitive, and REPLY, which corresponds to a reply message to a synchronous send message. Other possible types are: the CREATE type which is a message sent to a Creator process, asking for the creation

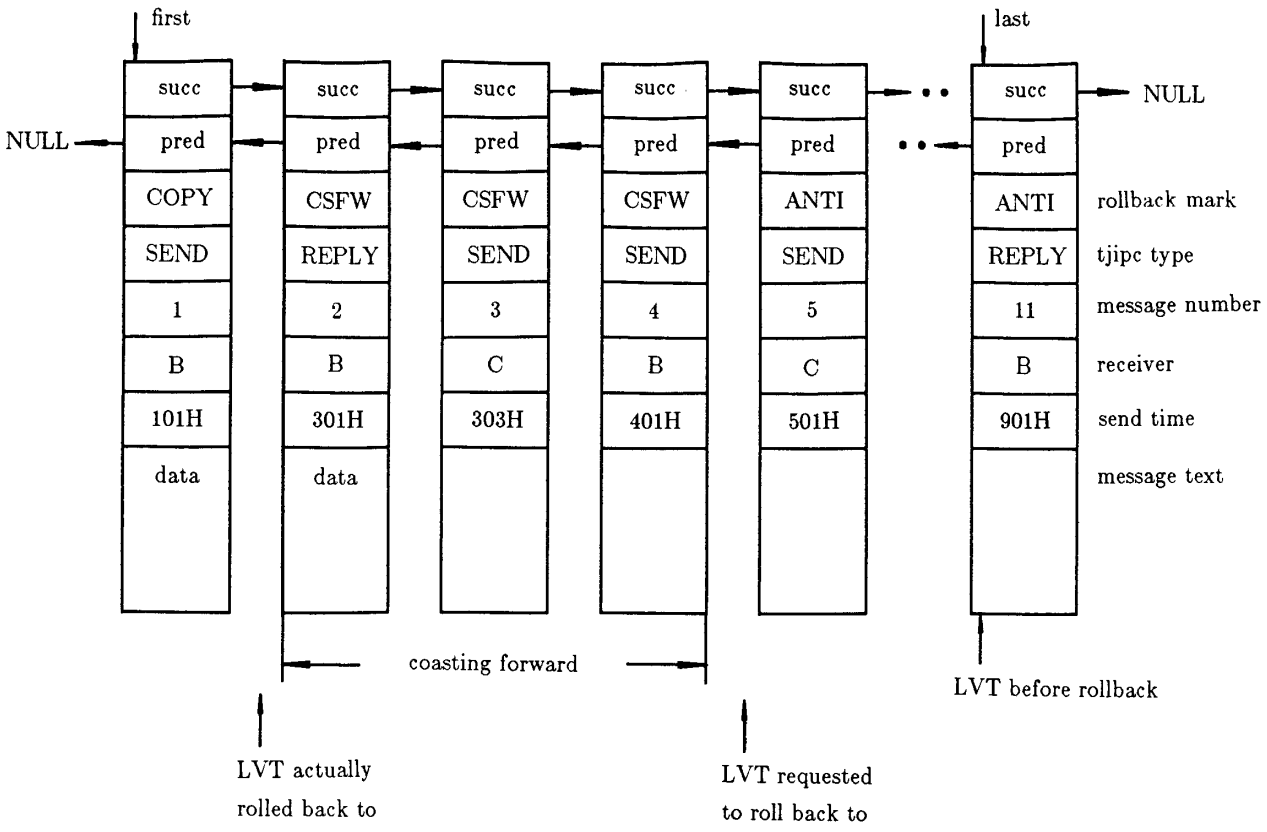


Figure 8. The Output Message Queue.

of a new Tipc process; the SEARCH type which is a message sent to an LCP to search for a named Tipc process; and the SEARCHW type which is similar to SEARCH except that the process that sends the search message will wait until the process being searched for comes to exist. Generally, an event which involves another process is treated as a message event and is inserted into the output queue. This provides a simple and consistent rollback mechanism.

The "rollback_mark" field is used to distinguish messages which might have to be cancelled. When a message is sent out, a copy of it is put in the output queue with "rollback_mark" set to COPY. In the example in Figure 8, suppose the process is at LVT 901H when it finds out it must roll back to time 500H. However, it didn't save a state at 500H; the oldest state saved earlier than 500H is 300H, so the process must roll back to 300H. All the messages in the output queue sent between that time (300H) and the time the process was asked to roll back to (500H) are marked with CSFW meaning that these messages will not be re-sent or cancelled as the process executes forward during the "Coast Forward Phase". All messages in the output queue that were sent out after the time the process was asked to roll back to (500H) are marked with ANTI. These "ANTI" messages might be cancelled if they should not have been sent, otherwise they will be marked with COPY again when the time comes to send them, but they will not actually be sent again because they have already been sent correctly.

Message numbers are local. Each message sent by a process has a unique number that is assigned incremently. This number will help the LCP on the receive end to annihilate pairs of "opposite" messages, for anti-messages have the same number as their corresponding positive messages.

The Process Data

The Process Data (PD) is another data structure maintained by the Tjipc package, in addition to the Output Queue. All the constants and variables associated with a user process are held in the PD. For the Unix version of Tipc, each user process has a private copy of the Tjipc package and the PD are the variables defined in the package as global variables. On the workstation version, a global array of PDs is defined within the Tjipc package and is shared by all user processes; the Tjipc routines operate only on the PD for the process which calls these routines.

The process data include the start time of the process, the virtual time at the moment the process came to exist, the LVT, the current virtual time of the process combined with the counter, the current message sequence number which is to be assigned to the next output message, and the pointers to the first and last message copies in the output queue.

The LVT is updated when the following events occur:

1. *A user process calls the primitive to advance its virtual time.* The LVT is increased by the value passed as the parameter, and the counter of the LVT is cleared. If the parameter is zero, LVT remains unchanged. The effect of advancing a zero time interval is for the LCP to reschedule the process and let other processes with same virtual time have a chance to run. The new LVT is passed to the LCP by a Local Control message of type "ADVANCE" and the LVT in the PCB is updated accordingly.

2. *A user process sends a message, either to another user process or to a system process* (creation or search requests). The LVT increases by one i.e., the counter increases by one and the virtual time remains the same. The LVT of the PCB is updated to the new value of the LVT after a Local Control message of some type is sent to the LCP.

3. *A user process calls a receive or receive-any primitive.* After the message is received, the LVT is

either updated to the send time of the message plus one if the send time is greater than the process's virtual time, or increased by one.

4. *A user process updates its virtual time after rolling back.* Having rolled back, a user process calls the primitive to notify the system of the time that the process actually rolled back to. The LVT is updated with the value passed as parameter, and the LVT in the PCB is also updated.

4.2 The Local Control Process

The LCP acts as the kernel of Tjipc. The functions of the LCP are to:

1. Provide input message buffering and keep copies of recently processed messages for each local Tipc process.

2. Detect rollback requests and inform user processes when they must roll back. Also, to carry out rollback procedures inside itself.

3. Switch the status of local processes when they send Local Control messages and to schedule processes when they are ready to resume execution.

4. Participate in the joint effort of estimating global virtual time and to collect the garbage of obsolete message copies from input queues when GVT advances.

5. Maintain a single special input message queue (the search queue) for search requests and to handle those requests.

6. Handle other messages, like kill requests.

For the workstation version, the LCP also implements memory management and message flow control. The LCP executes in an infinite loop, receiving messages and performing whatever functions necessary. To avoid deadlock, it is never blocked while performing these functions. All messages sent to an LCP have a integer, the "message type", as the first data item. The LCP uses the message type to determine how to process the rest of the message. Three types of message are accepted by the LCP: Tjipc, Local Control, and Global Control messages.

The Types of Tjipc Messages

Tjipc messages include messages sent by Tipc processes and the reply messages from system Creators. Those sent by Tipc processes consist of two separate Jipc messages, the header message and the data message, as described earlier in this chapter. A Tjipc message can be either a Communication message or a Request message.

A Communication message is a user-defined message for communication between Tipc processes. It can be a message sent from one process to another (type SEND) or a reply to a process that sent a message by using the synchronous send primitive (type REPLY). The LCP makes copies of these messages and puts them in the input queues of receiver processes. On receiving a communication message, the LCP carries out the following procedures:

1. If the message is an anti-message, the LCP will try to find its opposite in the input queue and the positive and negative messages will "annihilate" each other (both will be deleted from the input queue as if they had never appeared).

2. If the message has a send time earlier than the receiver's LVT, a check procedure is invoked to determine if the receiver process needs to roll back (for conditions of rollback see "The Detection of Rollback" in Section 4.4).

3. If the receiver is waiting for a message and the incoming message is the right one, the LCP will wake up the receiver (by switching the receiver's status to READY) and put it in the Ready Queue. Also, the receive time and receive status of the message will be set. Otherwise, if the receiver is not yet waiting for a message, the message is buffered in the input queue.

A Request Message is a message sent by a Tipc process to a system process, requesting such things as "search for another process", "create a new process", "kill a process", etc. These messages are not intended to carry information to a particular Tipc process; they only involve the relevant process indirectly. They are actually requests to the T_kernel and are messages in this implementation for the sake of a consistent and clear design. Different requests and the operations performed are:

1. *Creation Request:* The message is sent to the Creator and buffered in its input message queue, just like a communication message to a Tjipc process. The Creator will be woken up and allowed to resume execution, if it is waiting. The scheme of anti-messages is a bit different from that of communication messages, however. The user process that sends a creation request message does not resend it if, in a rollback situation, it turns out that the same creation request should have been sent with a different send time. Instead the user process sends an anti-message with the new send time to the Creator, which will "restart" the previously created process with a new start time rather than kill it and create another one. This kind of anti-message will not annihilate with its positive counterpart as usual; instead, the normal creation message is modified with the new send time and the anti-message disappears. If it turns out that a Tipc process should not have been created at all, the father process would send an anti-message to the Creator with the send time set to infinity. In this case, the normal message in the input queue of the Creator would be annihilated, the process created as the result of the creation message would be killed, and all the effects left by it cleaned up. This variation of anti-messages is intended to avoid unnecessary re-creation of Tipc processes, for we believe that the most common reason to roll back a creation request is to restart a process at a different time.

2. *Kill Request:* Since no process expects to be killed, the kill request is not kept in any queue. After a kill request is sent to the LCP by the Tjipc package, the LCP marks the "kill time" field of the PCB of the process to be killed with the virtual time that the process is supposed to be killed at. When the process reaches that time, it will be put in the DYING state. If the process's LVT already exceeds the time of the kill message, the process will be placed in the DYING state and forced to roll back so that the effects of events that happened after the kill time can be cleaned up. An anti-message to the kill message will cause the process affected to leave the DYING state and to roll back to an event right before the time of the kill message, if the process is already in the DYING state. This enables that process to resume its execution after being "pseudo-dead" for some time. A process being killed (in the DYING state) actually dies after the GVT advances through its kill time.

3. *Search Request:* The search request is put in the LCPs search queue instead of the input queue of the searched-for process because the search message is not a communication message and is therefore never expected by the process. The process may not even exist at the time the search is made. On receiving a search request, the LCP searches the PCB array for the named process,

then replies to the searcher as to whether the process was found, and the id of the process, if found. If the searched-for process comes into existence later, the LCP will let it know that is has been searched for in the LCP's reply to the process's initialization message (a Local Control message). Subsequently, the searched-for process will send a WAKESW message to wake up the searcher process, in the case of a blocking search, or a special anti-message (an AI message, see below), in the case of a non-blocking search. Both of these messages annihilate the search request in the searcher's output queue, causing the searcher process to roll back and search again. In the non-blocking search case, the previous search request will be deleted from the search queue. This scheme sounds complicated, but is necessary since the LCP is not allowed to send a message to someone who is not expecting it. Thus, the searched-for process itself, not the LCP, has to send those messages. An anti-message for a search message only annihilates its opposite in the search queue of the LCP on the destination machine and has no further effects.

An Anti-Input (AI) Message is a rejected input message which is sent back as an anti-message. It is different from an ordinary anti-message, which is directed by a sender process to a receiver process's input queue to cancel a message sent previously. An AI message is directed by a receiver process to a sender process's output queue to cancel a message in the output queue. This is necessary in situations when a received message must be rejected. For example, when a failed nonblocking search request should have been successful, as mentioned above, the previous search message would be rejected by the searched-for process and it would send an AI message to the searcher process, forcing it to roll back and search again.

The LCP does not handle an AI message the way it handles ordinary anti-messages because the LCP has no access to Output Queues. What the LCP does is force rollback in the process that the AI message is directed to. It also notifies it of the message number which has been rejected. It is up to the sender to delete the rejected output message, to roll back to the proper time, and then to resend the message if necessary (this is the usual case, since the message is cancelled not because the sender wants to but because it has been rejected by the receiver).

Figure 9 gives an example of an AI message. AI messages do not take the normal Tjipc message format (header message plus data message), rather a very compact form is adopted for efficiency. The code ANTIINMSG is used so the LCP can recognize this type of message, and the "anti-message" flag is set to -1. Forming the rest of the message are the local number of the user process that sent the positive message, the send time, and the sequence number of the positive message. The PCB in the picture shows the status of the user process (here with local number 7) after the AI message has been processed. The "rollback mark" field is set, meaning that rollback has been requested for this process. The LVT of the PCB is set to the send time of the positive message which is also the time the process has to roll back to. The AI message pointer points to a memory block which contains the sequence numbers of positive messages to be annihilated (it is possible that more than one AI message is received). When the time comes for the process to run, it will be notified of the rollback request and the time to roll back to, as well as the identities of the messages to be annihilated in its output queue.

The Types of Local Control Messages

An LC message is sent by the Tjipc package to notify the LCP that an event of a certain type is happening in a user process, such as waiting for an input message or waiting for GVT to advance. On receiving an LC message, the LCP first checks the "rollback mark" field in the PCB of the process that sent the LC message to see if rollback is requested for that process. If rollback is pending, the LC message will be ignored and the process is scheduled to resume execution (in most situations immediately) and forced to roll back. If no rollback is requested, the LCP either puts the user process in a wait state or reschedules the process in the Ready Queue, depending on the type of the LC message. Finally, the LCP

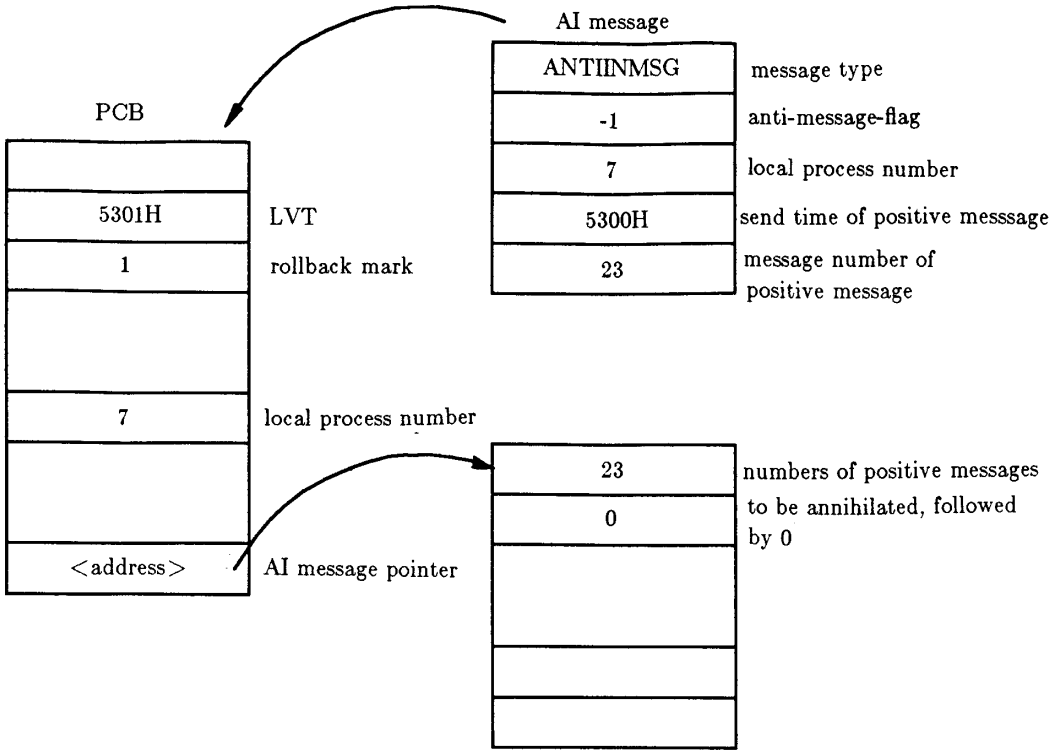


Figure 9. An Example Of An Anti-In-Message.

calls the dispatch routine to run the first process in the Ready Queue, if there is one.

The following are the types of LC messages which are used to notify the LCP of events happening in user processes.

1. ADVANCE: The user process's virtual time has advanced.
2. ASEND: A message has been sent by calling asynchronous send.
3. REPLY: A reply has been made.
4. DYING: The user process called exit.
5. LEAVE: The user process called "leave system".

When one of these types of LC message is received, the user process's LVT (kept in the PCB) is updated to the value contained in the LC message and the process is put into the READY state or into a temporary terminating state (LEAVE or DYING). The terminating state is temporary because processes in these states might roll back and become active again as long as the GVT remains less than the virtual time that these process are supposed to terminate at.

There are other types of LC messages used to inform the LCP of the event that a user process is waiting for. They cause the process to stay in a wait state with the same code as that of the LC message (discussed in Section 4.1), if the event the process is waiting for hasn't occurred yet. Otherwise, the process will be in the READY state with the new event.

The GVT Message

Only two types of GC message are recognized, the UNLINK message and the GVT message. Both are passed along the logical ring formed by a Tipc system. The UNLINK message originates in the Global Control Process each time a node quits the ring and is used to tell the node concerned of the change. Each UNLINK message is discarded when it reaches the node preceding the quitting node. The GVT message is used to regularly estimate the GVT. It starts circulating when the first user process in the system comes into existence and is discarded after the last user process in the system exits or leaves.

Upon the arrival of a GVT message, the following actions take place in the LCP if GVT has advanced:

1. *Garbage collection is performed.* Outdated messages are collected from the input queues of all user processes. Message copies in input queues that have been processed (have a non-zero receive time) and have a receive time less than the previous GVT are considered obsolete and can be released. All the messages with receive time equal to or greater than the previous GVT and less than the current GVT are preserved in case user processes roll back to a time earlier than the current GVT. This can occur in certain situations, (e.g., when a process has to roll back to GVT, but only has saved a state with an earlier timestamp than GVT) so some messages with an earlier receive time than the current GVT may have to be unreceived. This may or may not occur depending on how the user process states are saved in the Tipc level. It is recommended that careful consideration should be taken when designing the scheme for state saving in Tipc.
2. *Processes may exit or die.* User processes which were supposed to exit, die, or leave the system at

a virtual time less than the current GVT have their PCBs released and input queues cleaned up. Then they are allowed to actually exit, die, or leave the system.

3. *Processes waiting for GVT to advance are unblocked.* User processes which are in the GVT ("waiting for GVT to advance") state are switched to the READY state to resume execution. If a process in the GVT state has an LVT that is less than the current GVT, its LVT will be set equal to GVT because, normally, no user process should have an LVT less than the GVT.

4. *Processes held for flow control are made READY.* User processes which are in the HOLD state are put into the READY state.

5. *GVT calculations are performed.* A procedure is invoked to calculate the current lowest LVT on this node. If the lowest LVT equals the current GVT and the "allow to hold" flag in the GVT message is set, the GVT message will be held at this node until the lowest LVT increases. Otherwise, the GVT message is passed onto the next node in the ring along with the lowest LVT on this node since the last GVT message (called the historical lowest LVT). See Section 4.6 for details of the GVT estimation algorithm.

The GVT message is held at any node whose lowest LVT is equal to the current GVT. When this LVT advances, the GVT message is then sent to the next node in the ring. Otherwise, the GVT message is immediately forwarded, i.e., if the node's lowest LVT is greater than the current GVT.

4.3 Process Scheduling in the T_kernel

A brief outline of process scheduling was presented in Chapters One and Two. A complete description is presented here. The T_kernel schedules user processes on a "voluntary" basis, for there is no other way to control the execution of a Jipc process. This will cause problems because, for example, if a process spent too much time computing without calling any Tipc primitives, or even worse, entered an infinite loop, then other processes on the same machine, and eventually in the entire system, would be prevented from executing and progress would halt. The problem cannot be solved in Jipc, because there is no way to interrupt a process in Jipc other than by killing it.

Figure 10 shows the state transitions of a Tipc process. Usually, a running user process has its PCB first in the Ready Queue. Then the LCP unblocks it. It gets taken off the Ready Queue, and does not get put back into the Ready Queue until it calls another Tipc primitive; meanwhile, it runs under the scheduler of the host machine. In the mean time, no other user process runs on that machine. A running Tipc process switches to a wait state if it calls a primitive to wait for an external event to occur (e.g., an incoming message or for GVT to advance). In Section 4.1 various wait states are listed. In the following situations a process will be in the terminating state (i.e., DYING or LEAVE):

1. It reaches the end of its code and calls the exit primitive;

2. It calls the leave_system primitive;

3. It encounters a runtime error and the Tipc error handler routine (in the Tjipc package) sends a control message to the LCP. It is then suspended in the DYING state by the LCP;

4. Some other process calls the kill primitive to kill it.

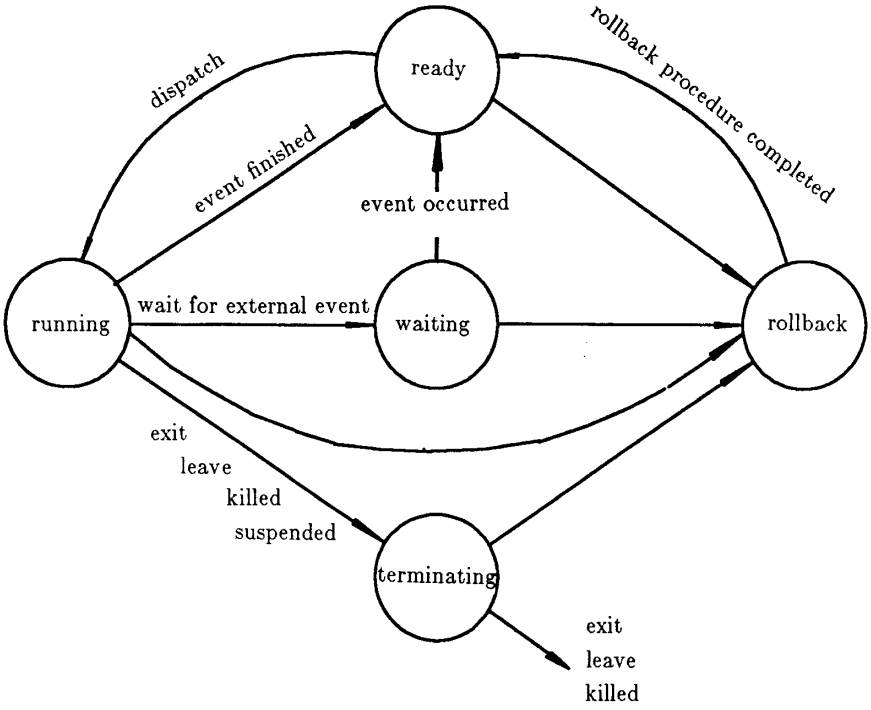


Figure 10. The Tipc Process State Transition.

The last situation occurs asynchronously with the execution of the process. A call to any primitive, whether the call would cause a transition to the READY or one of the wait states, will put the calling process into the DYING state if some other process had called the kill primitive and the process passes through the time it is supposed to be killed. The same is true for the transition to the ROLLBACK state: whenever a rollback request is detected for the running process, it will be forced to enter the ROLLBACK state no matter which Tipc primitive it has called.

A process is in the ROLLBACK state from the moment a rollback request is detected until the process completes rollback by calling a primitive within the Tipc level that informs the LCP of the actual LVT that the process has rolled back to. The transition to the ROLLBACK state could happen to a process in any other state and is non-deterministic. After the completion of rollback procedure, a process enters the READY state.

When the event a process is expecting occurs, the process becomes READY. The scheduling rules adopted in this implementation are:

1. Only one Tipc user process is allowed to run at any time, except for the period starting from the moment a process is created or enters the system until it calls the initialization primitive. This has made a significant contribution to the simplicity of the local control scheme of this implementation.

2. A process is not in the running state only when it completes a event or elects to wait for an external event.

3. The Ready Queue is sorted in nondecreasing virtual time order. The process with the lowest local virtual time is always scheduled to run next. This is natural for a Time Warp system. Notice that the LCP schedules according to virtual time, not according to LVT (the combination of virtual time and event counter).

Processes with the same virtual time have an equal chance to run. Usually, if a process has finished an event and is ready to execute again, it will be scheduled to run after all other processes with the same local virtual time already in the Ready Queue. From experiments with test programs, it has been found that this may not be fair. In some situations, it leads to unexpected rollbacks caused by communication between processes on the same machine. Since most rollbacks are triggered by straggler messages, one alternative is to let processes which call the asynchronous send primitive to continue to run even if there are other processes with the same lowest virtual time in the Ready Queue. This would allow, for example, a process to send messages to several processes at the same virtual time without having to wait for its next turn of execution. This can prevent unnecessary rollback in situations where several processes running on the same machine send and receive messages to and from one another at the same virtual time.

4.4 The Rollback Mechanism

Time Warp relies on rollback as the fundamental mechanism to realize the semantics of Virtual Time. Following the conditions presented in Section 2.2, each user process in the system proceeds forward based on its current situation. The rollback mechanism detects any violation of those conditions caused by late arriving messages and rolls the related user processes back to guarantee that virtual time semantics are maintained.

Not only is the rollback mechanism central to Time Warp systems, but it is also the most difficult part to implement and debug. Since this implementation was built as part of the Jade environment it had to conform to both the semantics of Jipc and the semantics of Virtual Time. The major design goal for the rollback mechanism was a simple and uniform scheme for the different kinds of events (message passing, process creation, searching, etc.).

Types of Rollback

Rollback is required for a user process in the following three situations:

1. *Receipt of a straggler.* Suppose a process that lags behind in terms of virtual time sends a message to another process that has gone ahead. That message may become a straggler for the latter process, in which case the latter must roll back to deal with the straggler in order to keep the correct ordering of events. This type of rollback is caused by the straggler message and may trigger further rollbacks in other processes.

2. *Receipt of an anti-message.* A process that has rolled back may need to cancel a message sent out previously. To eliminate the effect caused by the original message, an anti-message has to be sent. We call this type of anti-message a Normal Anti-Message: it is initiated by the sender and is aimed at a positive message in the destination process's input queue. On the arrival of the anti-message, the positive message will annihilate with the anti-message if the positive message is still in the input queue. However, if the positive message has already been processed, the receiver process will have to roll back.

3. *Receipt of an anti-in-message.* Suppose a message is rejected for some reason by the receiver. The rejected message will then be sent back to its sender as an anti-message. We call this type of anti-message an Anti-In-Message. It is sent by the receiver to cancel the positive message in the sender's output queue and forces the sender to roll back. In this implementation, messages in the search queue may be rejected when: a process that was searched for comes into existence after the search operation finishes (actually fails) and the search should have been successful; a process rolls back to the very beginning of its execution to restart at a different time (some of the search requests made before may have to be remade); a process should not have been created (all searches for that process must be undone). Other occasions when input messages may be rejected involve exiting or dying Tipc processes. For these processes, the unprocessed input messages should be rejected because the receiver no longer exists.

The rollback caused by receipt of a straggler message is the basic form of rollback; all other types of rollback are triggered by it. The LCP treats the three types of rollback differently for simplicity and efficiency.

The Detection of Rollback

In this section a more precise description is given of the criteria for each of the three types of rollback. These criteria are derived from the conditions listed in Section 2.2 under the following assumptions:

1. The send order of messages from one process to another is preserved on the receive end. This is true for Jipc messages.

2. The receive time of a message equals one plus the maximum of the send time of the message and the current LVT of the receiver.

Type 1 Rollback, Receipt of a Straggler

First, some notation must be defined. Let the send time of the new incoming message be M_ST, the receive time of that message be M_RT, the current LVT of the receiver process be RP_LVT, and the receive state of the message be RS. Remember that the receive state defines which call the receiver made to receive the message. Also let the send time and receive time of a message in the input queue be M_ST' and M_RT' respectively.

On the arrival of a positive message, rollback may be required for the receiver only if M_ST $<$ RP_LVT. Otherwise, we do not let the message be received at a time earlier than RP_LVT (i.e., we ensure that M_RT $>$ RP_LVT so that no rollback is needed). Therefore, because of assumption two, M_RT = max(M_ST,RP_LVT) + 1. Furthermore, if there is a received message in the input queue and one of the following conditions is satisfied, then the receiver has to roll back.

(a) (M_ST $<$ M_ST') and (M_RT' $<=$ RP_LVT) and (RS = RCVANY)

or

(b) (M_ST $<=$ M_ST') and (M_RT' $<=$ RP_LVT) and (RS = MAYANY)

or

(c) (M_ST $<=$ M_ST') and (M_RT' $<=$ RP_LVT) and (RS = MAYRCV) and
both messages are from the same sender

or

(d) (M_ST $<$ (RP_LVT (with counter cleared))) and (RS = SEND)

where RCVANY, RCV, MAYRCV, MAYANY, and SEND are the possible receive states of a incoming message (see "The Input Queue" in Section 4.1). If the following condition were true:

(M_ST $<$ M_ST') and (M_RT' $<=$ RP_LVT) and (RS = RCV) and
both messages are from the same sender

the receiver would have to roll back. But this situation never arises because of assumption one. Besides, when a process calls "receive from", it waits for and expects message only from a specific process. It does not care for messages from other processes until it decides to receive them. So there won't be rollback required for the receiver even if the following condition is met:

(M_ST $<$ M_ST') and (M_RT' $<=$ RP_LVT) and (RS=RCV) and
both messages are from different senders.

Type 2 Rollback, Receipt of an Anti-Message

If a processed (received) incoming message in an input queue is annihilated by an anti-message, the receiver process has to roll back to a LVT less than the receive time of the processed message. The input queue of the Creator is an exception. The Creator is not a Tipc process, but is solely responsible for creating new Tipc processes on request from existing processes. It never rolls back. When an anti-message arrives at its input queue it decides whether the process affected by the anti-message should have been created at a different time (earlier or later) or whether it should have been created at all. In either case, the LCP will force the process concerned to roll back to its very beginning and undo what it has done, after which it will either restart at a different time or cease to exist.

Type 3 Rollback, Receipt of an Anti-In-Message

An Anti-In-Message, an anti-message aimed at a message in an output queue, always causes the owner of the output queue to roll back to the send time of the message to be cancelled by the Anti-In-Message. It is complicated to decide under what conditions an Anti-In-Message should be sent. We are not going to list those conditions, for too many implementation details are involved. For this implementation, except for the unprocessed messages in an exiting or dying process's input queue, only search request messages are concerned. Search requests are handled in the context of a virtual time scale, and so are the situations when a rollback request arises.

Rollback Procedures

In the current implementation Tipc splits into two levels, the T_kernel and Save_Restore. The Save_Restore level handles user state saving and restoration, which is a large part of the rollback mechanism. Since the T_kernel level is also concerned with rollback, the rollback mechanism is divided between the two levels. Therefore, the two levels must coordinate their rollback actions. Since this system is implemented solely in terms of Jipc primitives, and since Jipc does not provide any kind of signalling facility, the only way to notify the Save_Restore level from the T_kernel level of a pending rollback request is through messages.

The method we adopt is to check, at the Save_Restore level, the newest virtual time value by calling an interrogation primitive provided at the T_kernel level. If the newest value is less than the last one (maintained in the Tjipc package), then a rollback is pending, and rollback procedures must be followed at the Save_Restore level. The same principle applies when the routines in the Tjipc package, belonging to the user process, try to detect a rollback request from the LCP.

On the detection of a rollback request, the procedures for handling rollback are as follows.

1. The LCP puts the process being requested to roll back in a rollback pending state by setting the "rollback mark" field of its PCB to a nonzero value, no matter what the process's current status is. Usually the rollback requests occur asynchronously. To distinguish rollback requests raised in different situations, the "rollback mark" is set to different values so that different handling procedures can be followed. The latest LVT that the process is requested to roll back to (call it bLVT) is recorded in the LVT field of its PCB. At this stage no further significant action is taken because the LCP does not yet know what LVT the process will actually roll back to. The LCP merely waits for a chance to notify the process of the pending rollback, i.e., when the time comes for that process to resume execution, if the process is not currently executing. If the

process is already executing, then when it finishes its current event and calls the LCP for the next event the LCP will inform the process that it has to roll back, ignoring the request from the process. A process in the rollback pending state will not get a chance to roll back until there is no process with an LVT earlier than its bLVT.

2. When the time comes for a process in the rollback pending state to resume execution, the LCP passes the bLVT (which tells how far to roll back) to the process in its reply message. The bLVT is less than the last LVT that the user process keeps. (If the user process is using Tipc, the Tipc package takes care of all facets of rollback.) If the user process does not check the bLVT and roll back promptly, the LCP will keep sending back the bLVT to remind the user process and will not let it proceed until rollback is completed. Other information besides the bLVT may be passed back to the user process in the reply message from the LCP. If the rollback is caused by an Anti-In-Message, the sequence number of the output message to be annihilated will be passed back so that the user process can cancel the message from its output queue (this is done automatically by the Tjipc package). If the user process is to restart at a different time or is to be undone (with the "rollback mark" field of its PCB set to 2 or 3 respectively), it may cause user processes that searched for it to roll back by sending them Anti-In-Messages. In this case, the user process has to know which Anti-In-Messages for search requests must be sent; it will get this information in the reply given by the LCP. The reason why the LCP does not send anti-messages directly is that all kinds of input messages are buffered at the LCP and LCPs are not allowed to send messages to each other asynchronously because of possible deadlock.

   If the process requested to roll back is to be undone because it shouldn't have been created, or if the process is to be killed by another process, the Tjipc package will clean things up and let the process terminate once GVT has passed through the time of the event.

3. Then the following steps are taken in the user process by the Tjipc package:

   The state queue is searched to find the latest state whose LVT $<=$ bLVT. The state queue keeps all the user states saved at an LVT equal to or greater than the current GVT and at least one state whose LVT is less than GVT,

   That state is restored, i.e., the user process rolls back to that previous state.

   The "set back time" primitive of Tjipc is called to inform the LCP of the actual LVT that the user process has rolled back to (call it oLVT).

4. When the "set back time" routine is called, all the messages in the output queue of the user process with a send time greater than oLVT will be set to the "anti" state, meaning that anti-messages may need to be sent for them. The anti-messages will not be sent unless it proves to be necessary (this is called "lazy cancellation"). Then a local control message containing the oLVT will be sent to the LCP which compares the oLVT to the bLVT to judge whether the process has rolled back or not (it has rolled back if oLVT $<$ bLVT). The LCP resets the process's input queue by setting the receive state to pending and the receive time to zero for all messages with receive time greater than oLVT.

In summary, rollback situations are detected by the LCP, which starts rollback. The procedures for handling rollback are initiated in the LCP, continued in the Tjipc package and at the Tipc level in the user process, and finally, end up back in the LCP. Then the process has finished the rollback process, and

transits to the READY state.

Batch Cancellation

There are two alternative cancellation mechanisms for performing rollback in a Time Warp system, Lazy and Aggressive. While aggressive cancellation immediately cancels all messages whose send times are greater than the LVT a process has rolled back to, lazy cancellation cancels only those messages which are not to be sent in the new situation; cancellation is delayed until the process executes forward again after rolling back.

Generally speaking, lazy cancellation rolls back only those processes which are affected by the arrival of a straggler message, while aggressive cancellation rolls back all processes that have received messages from the rolled-back process. Among these rollbacks, some are not necessary if they receive the same message with the same send time as they did before. Especially in the case of a straggler message which is side-effect free, lazy cancellation has an obvious advantage over aggressive cancellation.

A problem arises in some situations with lazy cancellation, due to "partial cancellation" . That is, some messages may be cancelled while others remain in receivers' input queues as future messages which may or may not be cancelled later on. There is a possibility that a process might run into a message it never expects, resulting in an unpredictable situation.

Figure 11 gives a simple example. Suppose process B sends two messages of different makeup to process A, and that process A expects to receive message two after message one. Further suppose that, after sending the two messages, process B rolls back previous to the time it sent message one and that it will now send message one at a different time (call the new message one'). An anti-message has to be sent to cancel message one and causes process A to roll back. Process A will then proceed forward again and try to receive message one'. But process A may catch message two instead of message one' if it executes fast enough. Sending message one' before sending the anti-message won't be of much help either.

To resolve this problem, a method called "Batch Cancellation" is introduced in this implementation. When a process rolls back, it does not send any anti-messages immediately (similar to lazy cancellation). When the process wants to send a message to another process in the new computation, it will check the output queue to see if there is a similar message which was sent at the same send time. If there is no such message, the process will send anti-messages for all messages marked with "anti", directed to that same process. It seems very likely that if a message to a process is different in the new computation, the following messages directed to it will be different too. If this assumption is true for most cases, the advantage of lazy cancellation can still be preserved.

4.5 Provision for Irrevocable Operations

Generally speaking, an operation which involves the outside world is irrevocable, either because it is physically irrevocable or because some part of the operation is carried out by a process that is not involved in the Tipc system and therefore cannot be rolled back. Since the GVT is the earliest virtual time that any Tipc processes could possibly roll back to, an irrevocable operation must not take place until the GVT advances through the time the operation is supposed to happen. In other words, processes which perform irrevocable operations must wait for GVT to advance before they can perform their function. A "wait for GVT" primitive is provided by Tjipc as the basic facility for irrevocable operations.

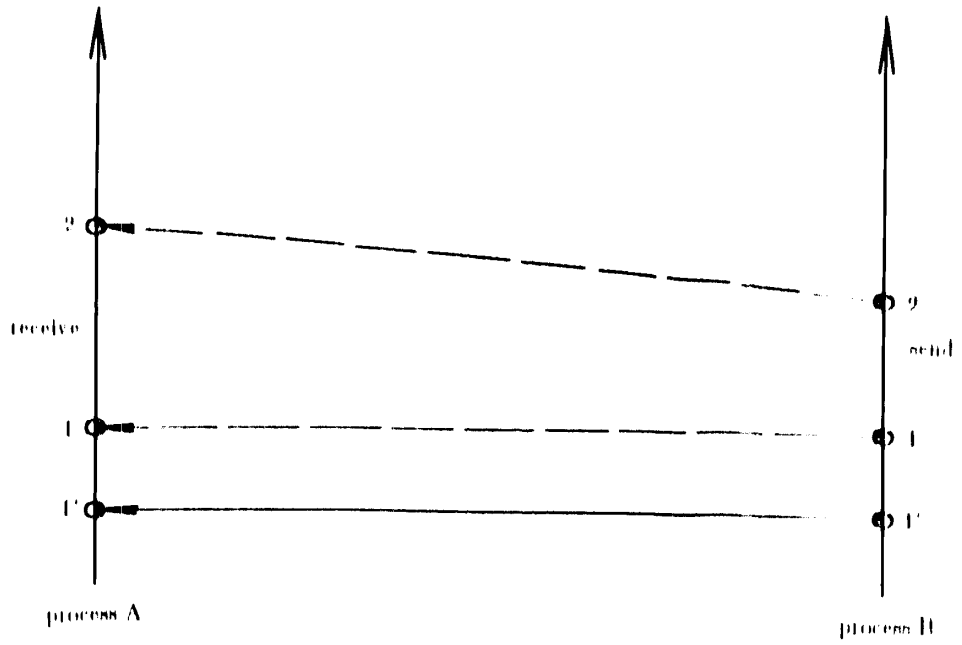


Figure 11. An Example Of The Problem With Lazy Cancellation

The pseudocode for an irreversible process looks like:

```
while GVT < infinity
begin
  wait for GVT to advance;
  if there exist any incoming messages with send time < GVT,
  process them;
end
```

Higher level facilities, implemented as Tipc processes, are created by system and use the basic primitive just mentioned to provide different kinds of services for user programs. These system processes serve as intermediates between Tjipc user processes and the outside world. For example, assume that "print" is an output routine provided by Tipc. When a user process calls "print", the request will be submitted to an output process as a message containing the text the user wants to print out. The output process will not actually send the text to whatever physical device is being used for output until GVT advances through the virtual time of the output request. Note that a caller of a higher level facility will not be blocked waiting for GVT to advance.

4.6 The Calculation of Global Virtual Time

Global Virtual Time (GVT) plays a central role in the global control of a Time Warp system. Theoretically, GVT is defined to be the lower bound of the current and future LVTs of all user processes in the same Tjipc system. At a given time, the GVT is the minimum of

1. All current LVTs of user processes in the system,

2. All virtual send times of all unreceived messages in input queues,

3. All virtual send times of messages in transit (or messages that have been sent but have not yet arrived at their destinations).

Point three can be ignored, because, when a message is in transit, the sender is blocked waiting for a reply from the receiver; the sender's current local virtual time remains equal to the send time of the message. For this implementation, those unreceived messages which may cause a process's LVT to decline are straggler messages; the current LVT of a process affected by a straggler message is set to the time of the straggler when the straggler message enters the input queue. So point one includes point two; point one alone can be considered to be the computational definition of GVT for this implementation.

It is impractical and unnecessary to maintain an accurate GVT because GVT changes constantly. There is actually no way to calculate the exact value of the GVT at a given time unless we can stop the entire system instantaneously. Therefore, an "estimate" of GVT is maintained in this implementation. The estimation may take place regularly on a certain time interval, or it may be triggered by some event. The algorithm we adopted gives a very close estimate of the real GVT (sometimes even the accurate value) without excess overhead. The estimation is launched only when it is necessary and stops when it reaches the real GVT. In the mean time, the most recent estimates, which are slightly out of date, are sent to each machine in the system. Following is a description of the algorithm.

Assume that the LCPs are numbered A1, A2, ... An and that there is a separate Global Controller, A0, that is a Jipc process. A0, A1, A2, ... An form a logical ring around which the GVT message circulates. Ai knows the process id of A(i+1), the next member in the ring. An knows the process id of

A0, the head of the ring.

An LCP joins in the ring by sending a LINK message to A0 after the first user process on its machine comes into existence, and it leaves the ring by sending an UNLINK message to A0 after the last user process on its machine dies (or leaves the system). On startup, the ring is empty, i.e., the successor process to A0 is itself. During the life time of the system the ring may expand and shrink dynamically. In the end, the ring becomes empty again, which indicates the termination of the entire system.

The GVT message is initiated when the first LCP joins the ring, and it is discarded on system termination. At any time, only a single GVT message exists in the ring. It starts and stops circulating under certain conditions described in the algorithm below. The GVT message contains three fields other than those used to distinguish it from other T_kernel messages and LINK/UNLINK messages:

TT: a working value that will be GVT at the end of each cycle (i.e., when the GVT messages comes back to A0),

GVT: last value estimated for GVT,

HF: a flag, telling if the GVT message is allowed to be held (delayed) in an LCP.

The following pseudo code outlines how the Global Controller and the LCP estimate GVT.

Process A0 — Global Controller:

```
Local variables:
   next_node          pointer to next member in the ring,
                      initialized to point to A0.
   gvt                local copy of GVT.

initialize:
first_node -> A0;
gvt := 0;

loop:
   case receipt of LINK message:
            next_node -> new member of LCP;
            if first new member then  /* Start GVT message circulating */
            begin
               gvt := GVT := 0;
               TT := inf;   /* inf is plus infinity */
               HF := 0;
               send TT, GVT, HF to A1;
            end;
            goto loop;

   case receipt of UNLINK message:
            get rid of the leaving member from ring;
            if ring not empty then
               goto loop;
            else
               terminate;

   case receipt of GVT message with TT, GVT:
```

```
if GVT = gvt then
   HF := 1;     /* Allow to hold */
else
   HF := 0;     /* Hold not allowed */
gvt := GVT:= TT;
TT := inf;
send TT, GVT, HF to A1;
goto loop;
```

Process Ai — Computed within the LCP:

Local variables:

Cmin_LT	Current local time (minimum of LVTs of all processes on this machine)
Lmin_LT	maintained as minimum value of local time since last visit of the GVT message
Hold	a flag, set if waiting for local time to advance before sending on GVT message, clear otherwise
Lgvt	Local copy of GVT

Procedure Send_gvt_msg:

```
TT := minimum(TT,Lmin_LT);
Lmin_LT := Cmin_LT;
Hold := 0;
send TT,GVT,HF to A(i+1 mod n+1);
```

Function Calc_LT:

```
return minimum of LVTs of all alive local processes;
```

At system startup:

```
initialize variables:
        Cmin_LT := Lgvt := Hold := 0;
        Lmin_LT := inf;
```

When GVT message received:

```
receive TT, GVT, HF from A(0 or i-1 if i>0);
Lgvt := GVT;
Cmin_LT := Calc_LT;
if Cmin_LT = GVT then
        Hold := 1;
else
begin
        Hold := HF;
        Send_gvt_msg;
end;
```

```
When Cmin_LT changes:
   Cmin_LT := Calc_LT;
   Lmin_LT := minimum(Lmin_LT,Cmin_LT)
   if Hold = 1 and Cmin_LT > GVT then
   begin
            HF := 0;
            Send_gvt_msg;
   end;
```

There are some points in this algorithm worth mentioning. Since it is impossible for any practical algorithms to calculate Cmin_LT for each node at exactly the same time, Cmin_LT is calculated for each node at the time of arrival of the GVT message. It is obvious that the GVT would not be the minimum of the Cmin_LTs calculated at different times. To estimate GVT, a lower bound of local time since the last visit of the GVT message must be maintained at each node. That is what Lmin_LT is. The estimate of GVT is the minimum of all Lmin_LTs. Another point concerns the use of the "Hold" flag which can prevent the following situation from arising: when a GVT message is held by Ai, the Ai+1, Ai+2, ... An would not get the latest estimate of GVT until the local time on Ai advances and the GVT message is passed onto them. A0 sets this flag if the new GVT is equal to the last estimate, i.e., all the nodes already have the lastest GVT.

4.7 Error Handling

The types of error dealt with in Tipc are those run-time errors which can be detected by the T_kernel. Some errors are specific to the implementation, arising in complex rollback situations, and are not expected by the programmer. The problem with Lazy Cancellation raised in Section 4.4 is such an example. However, this type of error is not a real error since it can be corrected if the process rolls back to a virtual time earlier than that of the error and executes forward, handling the situation properly. Therefore, we don't want a process or even the entire system to crash because of a "false" error. We see the error handling procedure as a last resort for preventing all false errors. A "real" Tipc error is that caused by a logical error in user-written code, for example, trying to send a message to a dead process or reading a wrong type of data item from a message buffer. This type of error is impossible to correct by rollback.

A uniform procedure is followed for handling both real and false errors. When a user process commits a run-time error, the T_kernel suspends that process's execution by calling an exit routine on behalf of that process. The process will stay in the DYING state until some straggler message comes in to roll it back or GVT advances through the time when the error is committed. For a false error, a straggler will arrive and cause the process to roll back, and the new message will "make things right" so that the process can execute forward properly. If an error turns out to be real, i.e., not remediable by rollback, then it will be reported to the user after GVT advances through the time of the error.

4.8 Memory Management and Flow Control

The flow control mechanism for this implementation attempts to cope with the limited primary storage space available on Jade Workstations. What we were seeking was a simple and practical solution rather than a general approach. Under the assumption that processes and processors are confined to a limited primary storage space, the overflow may arise in one of the following situations:

1. A message arrives and the processor or the receiver process runs out of space.

2. A output message is created and there is no space for a copy of that message in the output queue.

3. The processor or the process is out of space when a state is to be saved.

4. There is no space on the processor on which a new process is to be created.

The traditional flow control problem is the overflow of communication buffers, which usually are of fixed size and are held separate from the storage space of the processor. In this implementation, the overflow caused by incoming messages resembles the traditional problem, while the rest are problems of overflow of internal work space. In this implementation, all these overflow problems are closely related and are handled with a single mechanism.

The main issues involved in resolving the overflow problem are:

1. When a processor or a process runs out of storage, the only way to truly (not temporarily) recover currently allocated space is with "garbage collection". Storage is consumed by any event which demands it, whether it is an input message, an output message, state saving, or creating a new process. If the rate of consumption of storage is greater than the rate of reclamation, one of the above mentioned overflow conditions will occur. A flow control mechanism resolves the overflow problem either by temporarily removing some still useful information, or by holding the sources which produce storage-demanding events.

2. There exists a feedback effect between the rate of space consumption and the rate of its reclamation when the overflow condition is raised; i.e., the higher the consumption rate, the lower the recovery rate. This is because the system blocks processes contending for space until space is available. In the worst case, there is a possibility of deadlock if some process blocks on an overflow condition forever.

3. The usefulness of information is determined by GVT. Storage can be re-used only when GVT exceeds the virtual time associated with it. If a process gets blocked because of some overflow condition, it will be unblocked only after the overflow condition disappears. Since the recovery of storage must eventually be made by garbage collection, the disappearance of the overflow condition relies on overall system progress, indicated by the GVT. If the blocked process happens to prevent the advancement of the the message used in estimating GVT, the whole system will come to a halt. The flow control mechanism should be able to prevent this kind of starvation.

4. Allowing for the differences in the relative rates of progress of processes is essential for the speedup achieved by parallelism in Time Warp systems. This contributes to the flow control problem, for there is a relationship between space and the degree of such asynchrony. The higher the asynchrony, the more likely overflow is to occur. For example, an overflow is likely to occur when a process is (in terms of its LVT) far behind the processes from whom it receives messages because of the piling up of input messages. Those processes which go relatively far ahead in time may increase the size of their output and state queues, eventually causing overflow. A good flow control mechanism will restrict excessive asynchrony in the system. Yet it does not overly restrict the rate of system progress unless the user tries to run a large program in a small space when space is at a premium.

5. There is always a limit on the size of a program in a finite storage space. A tradeoff exists between the abundance of space in a system and the rate of progress of the system. If a Tipc

program runs at the limit of available space, a significant slowdown in global progress of the system is expected because of frequent blocking on overflow conditions. A reasonable abundance of space is required.

Considering the above issues and the limitations on this implementation, we decided to adopt a simple and practical flow control mechanism. To address the first problem, a traditional method suffices. The sender of a Tipc message gets acknowledged if the message is received and queued by the receiver's LCP. If not acknowledged, the sender knows that the receiver is running out of space. When a process encounters an overflow condition, it will be held (not allowed to run) until space is available, unless its LVT is equal to GVT. Since space reclamation depends on the advance of GVT, the process remains held until GVT increases.

To resolve the second problem, a spare memory pool is introduced. As long as there is no overflow, this pool is never used. It is used only in case of emergency. For example, if a process runs into an overflow condition and happens to have an LVT equal to GVT, then that process cannot be held; otherwise, GVT would never advance and the process would not have any chance to be released. In such a case, the spare pool is used. The spare pool has privilege in the recovery of space, i.e., a piece of space recovered by the garbage collection procedure does not go into the common memory pool before the spare pool is filled. In fact, this technique is nothing new other than that it controls overflow before it is too late to use any space for buffering at all. The overflow of the spare pool is much like the overflow of a stack of fixed size in that the size of the pool depends on the application and should be adjustable. An error message is given if the user program runs out of space in the spare pool. In that case, the user should either increase the size of the spare pool or reduce the number of processes running on the machine. This simple approach is ideal for preventing those processes which go far ahead from sending messages at a high rate to other processes which lag behind.

The following is the pseudo code for the overflow control mechanism adopted in this implementation. Process creation is treated as a request message, but it is considered as creation failure if there is no space for a process to be created.

```
Function hold:
   send local HOLD message to LCP;  /* suspended */
   if GVT advanced then
            return RELEASED;
   else return ROLLBACK;

Procedure malloc:
   ml:      call memory allocator;
      if no space available then
               if LVT > GVT then
               begin
                        rt:= hold();
                        if rt = ROLLBACK then
                           call rollback procedure;
                        else if rt = RELEASED then
                           call garbage collect procedure;
                           goto ml;
               end;
               else use spare pool;
```

When a Tjipc message arrives:

```
call memory allocator;
if memory allocated then
        send acknowledgement to sender;
else no space available
begin
        if send time = GVT then
           use spare pool;
           send acknowledge to sender;
        else message rejected;
end;


When sending a message:
        malloc();
        if memory allocated then
        begin
          rs: send message out;
           if message rejected then
           begin
                   rt:= hold();
                   if rt = ROLLBACK then
                      call rollback procedure;
                   if rt = RELEASED;
                      goto rs;
           end;
           else message accepted;
        end;
        else invoke rollback procedure;
```

4.9 State Saving and Restoration

State restoration is an essential part of the rollback mechanism in Tipc. When a process gets a rollback request, an earlier state of that process must be restored and all the operations performed after that earlier state should be undone (variable assignments are undone, input messages are unreceived, and output messages are cancelled). Only the state of a Tipc process is restored, not the state of the entire system.

As we have discussed, the rollback procedure is performed at two levels, the T_kernel level mainly takes care of the undoing of messages and the Save_Restore level deals with process state restoration. Process state refers to the state of execution of a user process and may include: the contents of registers, the value of the stack pointer, the value of the program counter, the value of the program status register, the contents of the user stack area, the values of global variables, and dynamically allocated memory. In this implementation we use the "checkpoint restoration" method for state saving and restoration. Checkpoint restoration normally requires that state be saved at regular intervals. If a rollback is requested, process state is restored to one of the saved checkpoints, and execution resumes from that point.

Since the rollback mechanism is the basic method for realizing virtual time semantics, state saving is performed on a regular basis. A useful method will be efficient and transparent to the Tipc programmer. This is required because is it very difficult for the programmer to decide where and how often the state should be saved and where to embed state saving and restoration commands in the program. Here we

present the basic components for performing state saving and restoration in one of our implementations, the Jade workstation version.

Two components are provided at the Tipc level for state restoration: SAVE (a function) and RESTORE (a procedure). Suppose that a user process is in state Si. When SAVE() is called, Si is established as a restoration point of the process. A reference to Si is returned by the SAVE function (for this implementation it returns a pointer to the memory block where the state of the process is stored). Suppose that, further on in the program's execution, a rollback request is detected requiring that the process roll back to state Si. A call to the procedure RESTORE(Si) will cause state Si to be restored, and the process will resume its execution from state Si.

Figure 12 illustrates the use of these two components. Suppose procedure one was called at some point previously, so that we have a state Xi to restore. State Xi will be restored after RESTORE is called, perhaps as a result of another call to procedure one, or a call to procedure two. The arrow shows the point at which execution will resume.

```
Procedure 1:                        Procedure 2:

   :                                   :
   :                                   :

  Xi:=SAVE();                         prepare message;
->                                    send;
  receive;                            if rollback pending
  if rollback pending                 begin
  begin                                Xj := (find Xj);
   Xj:= (find Xj);                     RESTORE(Xj);
   RESTORE(Xj);                       end;
  end;
  process message;                     :
                                       :
   :
   :
```

Figure 12. The Use of Save/Restore.

Another major issue in Tipc is when and how often state should be saved. Not only is the frequency of state saving a tradeoff between the overhead of saving and the overhead caused by rollback (in terms of both space and time), but there is also a requirement of consistency between the states saved and the input/output messages kept by the T_Kernel for a process. If the interval is too long between state saves, after a state restoration a process may find that some messages which should be processed have already been garbage collected. To avoid this problem, a criterion for state saving must be established. The following is based on these assumptions:

1. At the T_kernel level, all the input and output messages with receive time equal to or greater than the current GVT and all the unprocessed messages are kept in input/output queues.

2. At the Save_Restore level, one state with a time earlier than the GVT is always kept in the state queue.

With these assumptions, we can establish the largest interval between saved states without risk of missing messages in a rollback situation. Figure 13 shows this interval: state must be saved at the very beginning of a process, and every time a process's LVT advances. In other words, if more than one event occurs at the same virtual time, the current state just after the last Tipc event at that time must be saved.

Figure 13 shows that 3 is the latest state saved before GVT. Theoretically, GVT is the farthest a process can be requested to roll back. However, suppose a process finds out from the LCP that it has to roll back to GVT. If the process does not happen to have a saved state with a timestamp equal to GVT, it may have to restore a state with a timestamp less than GVT. Therefore, it is possible to roll back to state 3 in the situation shown in Figure 13. In that case, all the events (messages in the input and output queues) that happened after state 3 have time stamps equal to or greater than GVT and are preserved in input and output queues so there can be no missing messages. However, if state were saved at 3' instead of 3, or if no state were saved for virtual time VT3, then in the extreme case the process might have to roll back to state 3' or even state 2. If it rolled back to state 3', the events happening between state 3' and 3 might be missing because their time stamps are less than GVT and could have been garbage collected. If it rolled back to state 2, all the events of time VT3 might be missing. This would lead to fatal error.

In this implementation, state is saved every time a process's virtual time increases. That is, before a process calls "advance_time" to increase its LVT, or before calling any "receive" primitive. Note that state is saved before calling a receive primitive, not after. Otherwise, the received message (also a part of the user state) would be saved each time the state is saved, meaning that two copies for each input message are required. Also, if SAVE were called after "receive", then the message buffer would have to be restored in a state restoration.

In the version for Jade workstations, SAVE and RESTORE are two routines written in Assembler. They are similar to routines in operating systems used to save and restore the execution state of a process in process scheduling. But the SAVE described in this document saves not only the state of process execution (see above for the definition) but also the entire data area of a process, likewise for the RESTORE. This major difference results from the fact that when a process is swapped out in an operating system, its execution is suspended; the entire data area of the process remain untouched until it is scheduled to resume execution. In a Time Warp system the user state is taken as a snapshot at a checkpoint during its execution, so its data area is no longer the same after the snapshot. To restore a state of a Tipc process, every aspect of the process must be put back to that previous state.

For the workstation version only, the user stack area of a process is saved. This limitation restricts the user to define, as local, data which changes from state to state and which is relevant to the user state for the particular application. It is possible to save the global data as well if the underlying download facility can pass the pointer to the data upon the creation or entry of a new process. The problem is that large chunks of data which are irrelevant to the state would then be saved unnecessarily. By defining only those data relevant to the state as local, the space needed for state saving can be reduced to the extent that only state variables are saved.

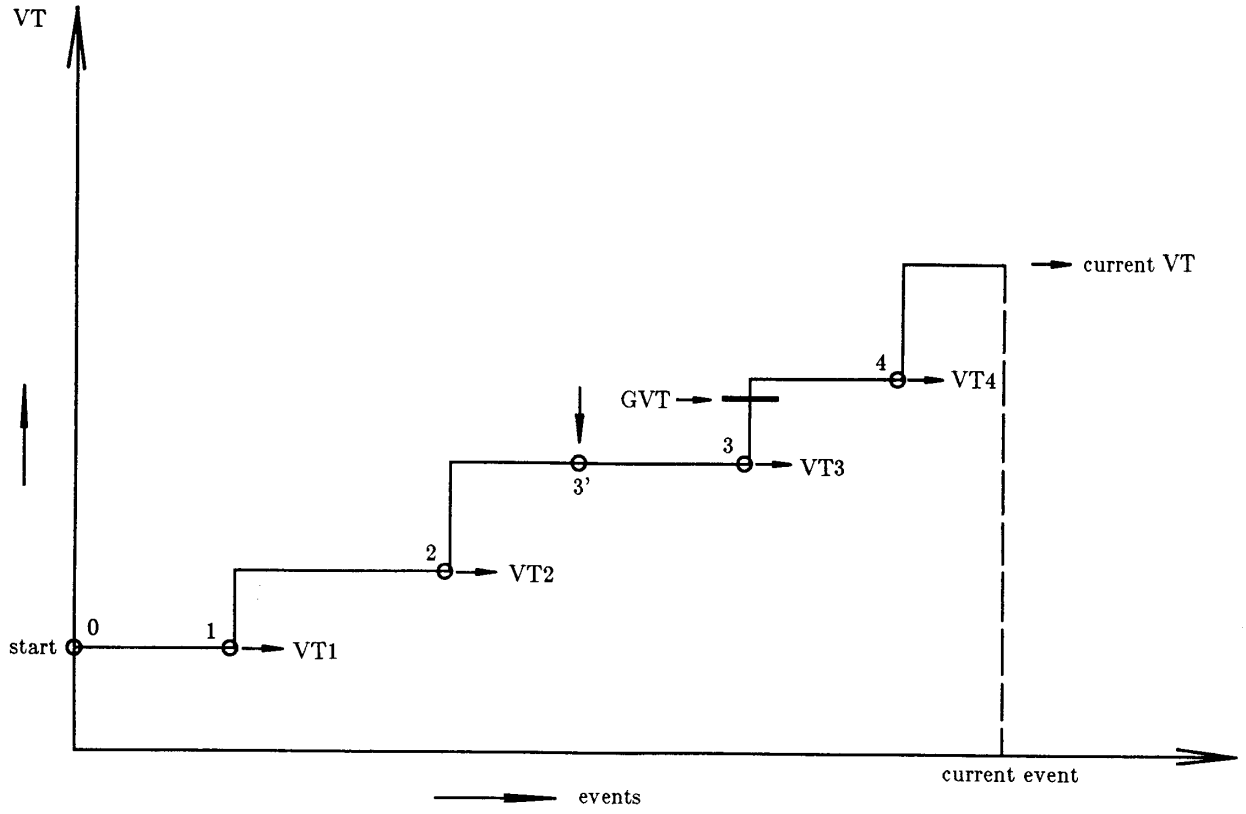


Figure 13. The Minimum Interval Of State Saving.

5 DEBUGGING AND PRELIMINARY EXPERIMENTS

It is well known that debugging a distributed program is a very difficult job, mainly because of the nondeterministic nature of distributed systems. The rollback mechanism makes debugging even more difficult because processes are often in incorrect states before a rollback occurs to set them on the correct execution path. Many efforts have been made to create development environments that facilitate distributed programming. Jade is such an effort; it not only served as the base for this implementation, but Jade tools also greatly helped in the the development and debugging of Tipc.

The Jade Window System has been used intensively throughout the development process, allowing us to control processes running on different machines. It also provides a program interface, so that the user can develop new tools using Window System facilities. In fact, a monitor for debugging has been written using this program interface and has been heavily used in the implementation of our design. This monitor displays in its own window the Tipc data structures and queues of any processes you wish to examine in the system, and is driven by menu selections.

All the test programs for Tipc are animated using Jaggies, the Jade graphics tool. The progress of program execution, each process's LVT, the GVT, and the rollback of processes are all all displayed with animated pictures which makes many bugs visible at a glance. Of course, finding and fixing a bug is not as easy as noticing that you have one in your program!

Several test programs have been written, mainly for the purpose of debugging, but they did demonstrate the Time Warp approach to synchronization in distributed systems. Testing shows that the extra overhead demanded by a Time Warp system may not be as high as expected. For this implementation, about two to three times of the underlying Jipc communication cost are needed for a corresponding Tipc communication. The cost of a rollback depends to a large extent on the cost of state restoration. More extensive tests and evaluation of this implementation are under way. Here we give the algorithms for two test programs in comparison with those suggested by ordinary synchronization mechanisms.

5.1 The Readers and Writers Problem

This is a mutual exclusion problem. The system consists of a fixed number of processes which share a single resource, say a file. The scheme of a central scheduling process which grants read/write requests in the order they are received will not work for an ordinary distributed system because the requests may not arrive in the order they are made. To solve the problem, an algorithm is suggested in [Lamport78]:

> "A system of clocks which satisfies the clock condition (given in [Lamport78]), is implemented and used to define a total ordering of all events. An event is either a request or a release operation. With this ordering, the major concern of the solution is to make sure that each process learns about all other processes' operations. The algorithm can be summarized as the following: to request the resource, a process sends a timestamped message to every other process and all other processes send back an acknowledgment message; to release the resource, a process sends timestamped messages to every other process; each process maintains a request queue and decides locally if the resource is available to him according to the timestamps of the messages it receives from all other processes."

The central process scheme works because Tipc guarantees that the requests will be accepted in the order they are made. The test program is a simulation of the reader/ writer problem. We assume that a system

of clocks similar to that in [Lamport78] is implemented on top of Tipc. A process sends a request message (either "read" or "write" is chosen randomly), stamped with the time from its clock, to a file process (which simulates the shared resource or the central scheduling process) and it waits for a reply from the file process indicating that the request is granted. Upon completion of the read or write operation, it sends a release message also stamped with its time to the file process. Then it does some local computing for a random amount of time and sends another request. Tipc guarantees that the file process will process the requests in their send time order. For a request event, only two messages are involved (only one message for a release event) against 2(n-1) messages (n is the number of processes which share the file) in the algorithm suggested in [Lamport78]. A process does not have to learn what and when the other processes make their requests. This algorithm is much simpler than Lamport's and requires less communication.

5.2 Conway's Game of Life

Suppose that there are n by n cells on a board. The rules of the game of Life are that: a cell will come alive if it has exactly three neighbour cells alive; and an alive cell will die if the number of its alive neighbours is less than two or greater than three. The game starts with some arrangement of alive cells, then the life board evolves from generation to generation. For each generation, some cells die, some stay alive or dead, and some others become alive. In our test program, each cell is represented by a Tipc process. A cell process sends a message, with its current generation as the timestamp, to each of its eight neighbours if it is alive for the current generation, then it counts the number of alive neighbours that it has by counting the number of messages received in that generation. Then it applies the rules of the game to see if it is alive and repeats. Here the system of clocks ticks with the generations, i.e., the virtual time of a cell process is the number of its generation. This algorithm allows that some groups of cells may be many generations ahead if they are kept away from other active groups by a zone of dead cells so that there is little or no influence between the different groups.

For a non-Time Warp system this algorithm does not work. To make it work, the major change to the algorithm would be that a cell process must send a message to each of its neighbours to notify then of its state, then wait for messages from all neighbours in the same generation before it can decide on its state in the next generation. With this algorithm, all the cell processes are synchronized and proceed at the same rate. Again, we can see that, for a non-Time Warp system, more communication costs are involved.

Generally, the system is similar to that suggested in [Cheriton79] - a system of clocks of some type must be implemented. The difference resides in the fact that, for a Time Warp system, the ordering of events is "automatic" (guaranteed by the system) as long as correct clock values are assigned to messages. So, it is not necessary to consider how to keep the events occurring in the proper order, whereas in other systems, algorithms must be designed in such a way that events occur in a rigid order. In fact, the design of a clock system is an essential phase in the design of an algorithm for a Tipc system. Once a suitable clock system has been chosen for an application, the rest of the algorithm is often straightforward. In the example programs discussed above, a synchronized real time clock system is required for the distributed readers/writers problem and a logical clock system which counts the number of generations is chosen for the Game of Life. However, an algorithm which counts excessively on rollback as method of synchronization may degrade the performance.

6 CONCLUSION

We view Tipc as a distributed software system supporting distributed concurrent processes. It provides an interface similar to conventional inter-process communication protocols, while obeying virtual time semantics. It enforces a user-defined temporal coordinate system to partially order the events happening in the system. The virtual time can be thought of as a resource provided by Tipc as a means of coordinating distributed systems. Therefore, a major concern in designing algorithms at the application level is the design of proper clock systems, suitable to the synchronization requirements of the application. The algorithms themselves may be much simpler compared to those for conventional systems.

The advantages of Time Warp systems include higher concurrency for distributed systems by eliminating unnecessary waiting, and less communication involved in synchronization as a result of much simpler algorithms. The costs are the extra communication overhead, the large amounts of memory required to save message copies and user states, and the cost of rollback. Further investigation is necessary for evaluating the performance of Tipc and for determining the types of application which are suitable to Time Warp.

As a new paradigm for distributed systems, Time Warp is feasible and worth experimenting with. The efficiency of Tipc is much higher than might be expected for such a complicated system. In fact, the major portion of the system overhead still lies in Jipc, which was primarily designed for reliability and program development, not speed.

Acknowledgements

This project has involved many people. The specifications for Tjipc and Tipc are the result of a series of discussions among those who were involved or interested in Time Warp. I would like to express my sincere thanks to Dr. Brian Unger, who was my supervisor and the director of this project during my visit to the University of Calgary for the opportunity, the support, and the advice he provided. I would also like to thank Dr. John Cleary and Greg Lomow for their numerous ideas and suggestions. Thanks to Konrad Slind who worked on the Unix version of Tipc and will continue to work on this project, for his help, especially his careful editing of this paper. I am also grateful to Li Xining who first explored and experimented with the Save_Restore level.

There are so many people in the Computer Science department of the University of Calgary, and in the Jade project, that I would like to thank. My special thanks go to Murray Peterson for the consultation he gave whenever I asked. Forgive me for not being able to mention everyone.

REFERENCES

Chandy, K.M., and Misra, J.
"Asynchronous Distributed Simulation via a Sequence of Parallel Computations."
Communications of the ACM, 24(4), 198-206, April 1981.

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

Jade User's Manual
"Part I: Developing Distributed Systems in Jade, Part II: The Jade Workstation, Part III: The Jade Graphics System, Part IV: An Example System."
Research Report, Department of Computer Science, University of Calgary, October 1985.

Jefferson, D.R., and Sowizral, H.A.
"Fast Concurrent Simulation Using the Time Warp Mechanism, Part I: Local Control"
Technical Report, The Rand Corporation, Santa Monica, California, December 1982.

Jefferson, D.,
"Virtual Time"
ACM Transactions on Programming Languages and Systems, 7(3), 404-425, July 1985.

Joyce, J., Lomow, G., Slind, K., and Unger, B.
"Monitoring Distributed Systems." to appear in ACM Transactions on Computer Systems.

Lamport, Leslie,
"Time, Clock, and the Ordering of Events in a Distributed System"
Communications of the ACM, 21(7), 558-565, July 1978.

Lomow, G.A., and Unger, B.W.
"Distributed Software Prototyping and Simulation in Jade"
Canadian Journal of Operational Research and Information Processing, 23(1), 69-89, February 1985.

Peacock, J.K., Wong, J.W., and Manning, E.G.
"Distributing Simulation Using a Network of Processors"
Computer Networks, 3(1), 44-56, February 1979.