Preface

This manual documents the *Tipc* inter-process communication protocol, which supports the implementation of Virtual Time [Jefferson 85] systems. Tipc can be used to write distributed simulations, as well as, other distributed applications. Tipc was developed in the Jade environment [Jade 85], and, to a large extent, depends on Jade.

This document should be self-contained in that a user shouldn't have to refer to other sources of information to get a Tipc program to run. However, if you are interested in the implementation of Tipc, see [Xiao et al. 86]. The original plan for the implementation of Tipc is presented in [Cleary et al. 85]. A discussion of potential optimisations in the implementations appear in [West et al.87].

For those who know the Jade IPC protocol (Jipc), Tipc was designed to mimic Jipc as closely as possible. Tipc can be viewed as Jipc augmented with facilities for handling time. Specifically, Tipc has a few extra parameters to some subroutine calls and an asynchronous send primitive. Currently, Tipc is accessible only from the C programming language. Preliminary ideas for a prolog version are described in [Li & Unger 87].

In section One, there is an introduction to Tipc. In section Two, we present a simple example system - a multi-process version of Conway's Game of Life. In sections Three to Seven, there is detailed documentation of the Tipc subroutine package. In sections Eight and Nine, we explain how to get a Tipc program to run and describe some utility programs that make the Tipc programmer's life a little easier. Finally, section Ten provides a concise summary of all the Tipc calls and constants.

1. Basic Concepts Of Tipc

The Time Warp Inter-Process Communication protocol (Tipc, pronounced "tipsy") is the foundation on which distributed simulations, or, in general, Virtual Time systems are constructed. Tipc handles the lower-level details of inter-process communication, freeing the application using it from dependencies on particular machines and communication networks. A distributed system that uses Tipc can thus be changed to use a different configuration of machines with minimal effort. Tipc also handles the synchronization of time across a distributed, asynchronous application.

Machines, processes, and messages.

Tipc views the world as consisting of *processes* which send each other *messages*. A process has certain data areas associated with it (global variables, stack, dynamically allocated memory); it is executing a *program* (which other processes may be executing also); it has a program counter that indicates what instruction that program is currently executing. The Tipc notion of a process is quite similar to the notion of a process in Unix.

A message is a packet of data that is transferred between processes. In a pure Tipc system, these message packets are the only way processes communicate. In particular, there is no shared memory between Tipc processes, even those on the same machine.

One may think of a process as having its own *processor* to execute its instructions. Usually this will not be the case; a single processor will be time-shared among several processes. Each Tipc process executes on a particular *machine*. Typically, each of these machines will have only a single processor, but there might be more if it is completely invisible to a program that there are more than one. Which machine a process executes on is of some significance to the Tipc programmer, since it may affect the facilities available to the program and probably affects the cost of communicating with other processes (local communication is almost always cheaper than communication between machines). The way one process talks to another process via Tipc is the same regardless of what machine it is on, however.

Tipc messages are moderate-sized collections of data (up to a few hundred bytes). Each item of data in a message is *typed* - identified as being an integer, a character, or whatever. This allows Tipc to automatically convert data representations when messages are passed between different machines. Typing of message items also allows monitoring tools to print the contents of messages intelligibly and provides a check on programming errors.

Protocol for passing messages.

The basic message interaction in Tipc is: one process *sends* a message to another, the destination process *receives* the message, and later *replies* with another message. The process that sends the message is *blocked* until the reply comes back. A process is said to be blocked when it does not execute any further instructions until an event it is waiting for occurs. A process that tries to receive a message is blocked until a message arrives, unless one was already waiting. A process never blocks when it replies.

Tipc also has an *asynchronous* send. When a process sends a message to another using the asynchronous send, the sender is not blocked awaiting the reply, but continues executing. The receiver of an asynchronous send message does not reply to the sender.

There are several variations on this general pattern. A process may elect to receive a message from a particular process, in which case it is blocked until such a message arrives, or it may instead elect to receive a message from any process. It is also possible for a process to ask to receive a message from some process or any process without being blocked if no such message is pending. Finally, it is possible to wait to receive a message for a certain period of time. If no message has arrived in that time, the process is allowed to continue execution.

Time management.

In a Virtual Time system, processes are synchronized on the basis of time. That is, if a process interacts with another process, their interaction must occur at the same time. For this purpose, every Tipc process has a time value associated with it called its *local virtual time* (LVT). Each process can increment or interrogate its own LVT whenever it chooses. To ensure that when processes interact, they interact at the same moment of time, i.e., their LVTs are equal when they interact, much activity invisible to the programmer occasionally needs to be done. This activity is usually of no concern to the Tipc programmer.

Determining when to advance the LVT of a process is an important design decision when building a Tipc application. Each process must advance its LVT often enough so that all processes proceed more or less together in time. This loose synchronization can be stretched too far if a process goes much further ahead in time, or lags too far behind in time compared to the other processes in the system. The design of the application system should tend to minimize situations where this occurs.

Process creation and destruction.

Tipc contains primitives that allow a process to create another process to run on the same or another machine. It also lets a process kill another process on the same or another machine.

A process may also be "created" as far as Tipc is concerned when a non-Tipc process (e.g., an ordinary Unix process) becomes a process known to Tipc and thereby capable of message interactions with other Tipc processes.

Finding processes.

Every Tipc process has a *process id,* which must be known in order to send messages to it, kill it, request to receive a message from it, etc. Process ids are unique among machines and through time. There are several ways one process can acquire the id of another process.

One method of locating a process is via *process names.* These are assigned when the process is created or becomes known to Tipc. A process can find out about another process by searching for it by this name. The search may be restricted to the same machine as the searcher or may be directed to some other particular machine.

A process that asks to receive a message from any process will be informed of the process id of the sender when a message is received. It may use this id when replying or for any other purpose.

Processes can find out the id of their parent, if they have been created by a Tipc process. Processes that create other processes are informed of the new processes' ids.

Finally, one process can give a process id to another process as a data item in a message.

In order to allow independent development projects to co-exist, Tipc processes are grouped into *Tipc systems*. Processes in one Tipc system cannot affect processes in another system. In particular, the facilities for finding a process by name restrict the search to processes in one Tipc system.

Handling of failure situations.

Many "failure" conditions can arise in a distributed system. These include "permanent" events such as a machine going down, a network ceasing to transmit, or a process terminating abnormally, as well as "temporary" conditions such as a network transmit collision. Tipc handles some of these conditions; others are left for the application to handle if it wishes.

A Tipc program is expected to assume that messages that it sends are actually delivered to the destination machine. Tipc itself will arrange for the necessary acknowledgements and retransmissions. Of course, if the destination machine or intervening network is very busy, Tipc may eventually give up trying to transmit the message. Currently, this can cause the sender to hang.

If the process to which a Tipc process sends is dead or if it dies before it returns a reply, either normally or because of some error, the process waiting for a reply will be informed of this. However, if the machine on which the process runs crashes, the sender will generally hang.

The support of Virtual Time in a way invisible to the application program requires the use of a limited storage area. This area will overflow if the application program is trying to make too many Tipc calls at the same local virtual time. If overflow is detected, the process will terminate.

To summarize, Tipc currently handles errors in application software and "soft" hardware errors, but software and hardware crashes of entire machines or networks can cause applications to hang, though the Tipc implementation itself will not crash as a result of problems with another machine.

2. The Tipc Version of Conway's Game of Life.

Conway's game of Life is a grid of automata that, starting from an initial configuration, undergo state changes (known as generations) until a stable configuration is attained. A cell in this grid can be in one of two states: alive or dead. In a generation, if a cell is alive and two or three of its neighbours are alive, it continues living in the next generation. If it is alive and less than two of its neighbours are alive, then it dies for lack of company. If it is alive and more than three of its neighbours are alive, then it dies from overcrowding. If it is dead, it requires exactly three alive neighbours to join the living. The initial configuration of alive cells therefore determines the following generations. This example makes each cell a separate Tipc process. The C source code for the Tipc solution is presented in Figures 1 and 2. Figure 1 presents the code for the process that creates all the cells of the game. Figure 2 contains the code that each lifecell executes.

One notable feature of this version of the game of Life is that the grid wraps around: the "outside" neighbours of cells along an edge are cells from the opposite edge of the grid. For example, on a four by four grid, the cell at row 4, column 4 has neighbours, in row major order: <3,3>, <3,4>, <3,1>, <4,3>, <4,1>, <1,3>, <1,4>, <1,1>.

In the version of the game of Life presented here, generations are equated with time quanta, i.e., when a process moves to the next generation, it increases its LVT by one unit. Each cell is informed of the identities of its eight neighbours at start up; they are the only processes that it communicates with. A cell operates as follows: if it is alive, it sends messages to its eight neighbours, using the asynchronous send primitive eight times. If not, it doesn't. Then the cell goes into a maybe-receive loop, receiving messages from its neighbours, if there are any to be received. As soon as a maybe-receive is not successful, i.e., returns NO_PROCESS, the cell assumes that all messages that could have been sent to it in that generation have arrived, and counts the number of messages it received. If it received three messages, or if it was already alive and it received two messages, it decides that it is alive. Otherwise, it decides that it is dead. Then it goes back to the top of its loop.

The only restriction on Tipc processes is that the first thing they must do is either call t_initialize or t_enter_system. This is involved with setting up the invisible storage area described in section 1. If either of these two calls is not made at the very beginning of a Tipc process, inexplicable errors may result.

Parallelism is obtained in two places here. First, if a cell is not alive, it doesn't send any messages, although it does increment its time. Therefore, it can charge ahead in time. If it is still dead in the next generation, then it has saved itself eight messages. Second, it does not have to receive messages from all eight neighbours in a generation - it only receives messages from those processes that are alive. This is a significant saving in message passing costs. A nice thing about Tipc is that it lets processes run in a loosely coupled fashion, so that the programmer has less synchronization to worry about. We have observed that protocols for various problems are simple and even elegant when programmed in Tipc.

```
/* Life - the startup process for the game of life. It creates all the
   lifecells and distributes neighbour ids to each, then goes away.

   To run:  1. Start the underlying system
            2. Start life, which will start the lifecells and then exit.
*/
#include <stdio.h>
#include <tipc.h>

#define        ROWS    4
#define        COLS    4
#define BORN ((i>0 && i<=3 && j==1)?"1":"0")  /* for 4X4 */

static t_process_id cell[ROWS][COLUMNS];
static char cell_name[8];
static int column_left, current_column, column_right;
static int row_above, current_row, row_below;

main()
{ int i,j;

  t_enter_system(637,"life",0);              /* This must be the very first call made */
  for (i=0; i<ROWS; i++)
  for (j=0; j<COLUMNS; j++)
  { sprintf(cell_name,"cell%ld%ld",i,j);
    sprintf(row,"%ld",i);
   sprintf(col,"%ld",j);

    if (i==0)
    { cell[i][j] = t_createm_process(0,"vaxb",cell_name,
                              "/userb/srdg/slind/tw/src/test-suite/game/lifecell",
                              BORN,0);
    }
    else if (i==3 && j==3)
    { cell[i][j] = t_createm_process(0,"vaxd",cell_name,
                                   "/user/srdg/slind/tw/test/lifecell",BORN,0);
    }
    else
    { cell[i][j] = t_createm_process(0,"vaxc",cell_name,
                                     "/user/srdg/slind/tw/test/lifecell",BORN,0);
    }
  }

 for (i=0; i<ROWS; i++)      /* For each cell, tell it who its eight neighbours are */
   for (j=0; j<COLUMNS; j++)
   { row_above = bimod(i-1,ROWS);
     current_row = i;
     row_below = bimod(i+1,ROWS);

     left_column = bimod(j-1, COLUMNS);
     current_column = j;
     right_column = bimod(j+1, COLUMNS);

     t_clear();
     t_putp(cell[row_above][left_column]);
     t_putp(cell[row_above][current_column]);
     t_putp(cell[row_above][right_column]);
     t_putp(cell[current_row][left_column]);
     t_asend(cell[i][j]);    /* The limited size of message buffers requires 2 sends to */
                             /* transmit all the information */
     t_putp(cell[current_row][right_column]);
     t_putp(cell[row_below][left_column]);
     t_putp(cell[row_below][current_column]);
     t_putp(cell[row_below][right_column]);
     t_asend(cell[i][j]);
   }
   t_exit();
}

/* bimod is a two dimensional modulus operator */
static int bimod(no,max)
int no,max;
{ if ((no+1)==0) return(max-1);
  else if (no==max) return 0;
  else return no;
}
```

FIGURE 1 - The Startup Process for the Game of Life.

```
/* t_lifecell - cell for game of life. Gets created by life program and runs
forever.
*/

#include <tipc.h>

#define       ALIVE   1
#define       DEAD    0

t_process_id neighbor[8];
t_process_id pid;

main(argc,argv)
int argc;
char **argv;
{ int life_state, counter, i;

  t_initialize();                      /* Must be very first call made */
  life_state = atoi(argv[3]);                  /* Alive or Dead */
  t_receive(t_parent_process());
  for (i=0; i<4; i++) neighbor[i] = t_getp();  /* Get process ids of neighbours */
  t_receive(t_parent_process());
  for (i =4; i<8; i++) neighbor[i] = t_getp();

  while (1)
  { t_advance_time(1);                         /* Time quanta are generations */
    if (life_state==ALIVE)             /* If alive, tell neighbours */
    { for (i=0; i<8; i++) t_asend_null(neighbor[i]);
    }
    counter = 0;                       /* Find out how many of them are alive */
    for (i=0; i<8; i++)
    { pid = t_maybe_receive_any();
      if (t_same_process(pid,T_NO_PROCESS)) break;
      counter += 1;
    }
    if ((counter==3) || (counter==2 && life_state==ALIVE)) life_state = ALIVE;
    else life_state = DEAD;
  }
}
```

FIGURE 2 - Cell for the Game of Life

3. Message-passing Procedures.

In this section we describe how to send messages to other processes, how to receive messages, and how to reply to those messages that have been received.

To send a message, one must specify to whom it is to be sent and what the content of the message is. A *process id* is used in specifying the destination of a message. The following section discusses process ids and how one can get them. We will assume here that messages can be constructed and stored in a *message buffer* and that the contents of a message residing in a message buffer can be examined. The message-passing procedures operate on such a message buffer. Section 6 describes how to construct messages.

This message buffer is "hidden" - it does not explicitly appear as a parameter of the procedures. Usually, the same buffer is used for all message-passing (and buffer manipulation) procedures. Specifying it explicitly in each call would be tedious. Instead, these routines operate with whatever is the "current" buffer.

Basic message-passing routines.

The basic pattern of message passing in Tipc is the send-receive-reply sequence. A process sends a message to another and is then blocked from executing. The process to which the message is sent eventually receives the message, performs whatever computation is required, and returns the reply, which causes the sender to resume execution. If the second process were to try to receive the message before it was sent by the first process, the second process would be blocked until the message arrived.

```
t_send(to)
  t_process_id to;
```

The message in the current buffer is sent to the process identified by the parameter. The caller is then blocked until a reply arrives. When the caller resumes execution, the current buffer will contain the reply.

```
t_send_null(to)
  t_process_id to;
```

An empty message is sent to the process identified by the parameter. The caller is then blocked until a reply arrives. When the caller resumes execution, the current buffer will contain the reply.

```
t_process_id t_receive_any()
```

This routine causes the caller to be blocked until a message is sent to it by any other process. On return, the current buffer will contain the message received and the id of the process that sent the message will be the return value of the call. If more than one message is pending at the time t_receive_any is called, the message received will be the one with earlest virtual send time.

```
t_process_id t_receive(from)
  t_process_id from;
```

This routine blocks the caller until a message arrives from the process identified by its argument. On return the current buffer will contain the message received and the id of the process that sent the message will be the return value of the call.

```
t_reply(from)
  t_process_id from;
```

This procedure sends the contents of the current buffer as a reply to the process identified by the argument. This call will not block the caller, and unblocks the process to which the reply is directed.

```
t_reply_null(from)
  t_process_id from;
```

This procedure sends an empty message buffer as a reply to the process identified by the argument; the current message buffer is not disturbed. This call will not block the caller, and unblocks the process to which the reply is directed.

Note that one is not required to reply to a message before receiving another one, nor is it required to reply to them in the order they were received. In fact, one is not required to reply to a message at all, but in that case, the sending process will be blocked forever.

Since Tipc was designed to help distributed systems run faster by allowing processes to spend more of their time running in parallel, there is an asynchronous send primitive. Using it, a process can send a message to another process and then continue execution, rather than waiting for a reply before continuing. The receiving process does not reply to a message sent with this primitive.

```
t_asend(to)
  t_process_id to;
```

The message in the current buffer is sent to the process identified by the parameter. The caller is not blocked waiting on a reply. After this call, the current message buffer is empty.

```
t_asend_null(to)
  t_process_id to;
```

An empty message is sent to the process identified by the parameter. The current message buffer remains unchanged after the call. The caller is not blocked awaiting a reply.

There arise situations where one is willing to wait for a message, but only for a certain length of time, after which other processing should be done. Tipc provides three routines for this. The first two are non-blocking counterparts to the receive and receive_any primitives; the last performs a function mid-way between that of a receive_any and the non-blocking version of receive_any.

```
t_process_id t_maybe_receive_any()
```

Like t_receive_any, except that it never blocks. Instead, it returns T_NO_PROCESS if no message with virtual send time less than the current LVT is pending.

```
t_process_id t_maybe_receive(from)
  t_process_id from;
```

Like t_receive, except that T_NO_PROCESS is returned without blocking if no message with virtual send time less than the current LVT is pending from the argument process.

```
t_process_id t_receive_any_pre(time)
  int time;
```

This routine waits for any message for the amount of time passed as the parameter. If there is no message when that time has passed, the routine quits with the caller's LVT advanced and the message buffer empty; the returned Tipc process id will be that of the caller. This routine returns the sender's process id if there is a message available in that time period, else it returns the caller's process id (this can be used to determine if a message has been received).

4. Getting and Manipulating Process Ids.

Before a Tipc process can communicate with other Tipc processes, it must acquire the *process id* of the process it wishes to talk with. A process id uniquely identifies a Tipc process. No process on any machine at any time in the present, past, or future can have a process id the same as some other process's. This section explains the concept of a process id and how to search for a process to obtain its process id.

Nature of a process id.

A process id contains four components: the process name, the machine name, the system number, and a unique identifier.

Every Tipc process has a 'process name'. Process names are strings of between zero and J_MAX_PROCESS_NAME (currently 14) characters. Unlike process ids, process names are not unique - any number of processes may have the same name. The name of a Tipc process is assigned when it is created (or becomes a Tipc process if it started life as a non-Tipc process). The process name cannot change during the life of the process.

A Tipc process runs on a particular 'machine'. These machines also have names, which are strings of from zero to J_MAX_MACHINE_NAME characters (currently this limit is four). Machine names are unique.

To allow a number of users to develop Tipc systems on the same set of machines simultaneously, Tipc processes are assigned to 'Tipc systems'. Systems are identified by integers. Processes in different Tipc systems cannot talk to one another or affect each other in any other way (via Tipc). An exception to this is system 1, which is used for server processes that must be accessible to any process.

These three attributes of a process - its name, the machine it runs on, and the Tipc system it resides in - together with one additional datum added for uniqueness, combine to form a process id. If you have a process id, you can obtain any of these attributes (except the extra unique datum). However, you may not do so directly by looking at the bits in a process id. Process ids are stored in some format decided by the Tipc implementor. The format is subject to change at any time. You must instead use the procedures provided for enquiring about these attributes. Similarly, you may only obtain process ids by use of the supplied functions.

A special process id value, represented by the symbol J_NO_PROCESS, exists to provide something for functions to return when they can't find a process. It is also occasionally convenient for an application to use this value as a "null" process. It is not really the id of anything; you can't send messages to it.

To declare a variable that can hold a process id in C, use the "t_process_id" type, e.g.:

```
t_process_id printer_server;
```

You may assign to process id variables, pass them as parameters, and return them as function values. You may not try to see what they really are internally.

Process ids are obtained when: one Tipc process creates another; a Tipc process receives a message from another Tipc process; a search for a process is successful; and when a process id is extracted from a message buffer. There is also a routine that allows a Tipc process to find out its own process id.

Searching for a process.

A Tipc process may search for another Tipc process on its machine, or on another machine, provided that the other machine is known to Tipc. If the searched-for process does not yet exist, the caller can either decide to wait until it does exist, or continue execution. Searches are made by name - each search call takes a name and returns a process id, if it returns. Searches are confined to the Tipc system of the caller.

```
t_process_id t_search_locally(process_name)
  char *process_name;
```

This function looks for a process with the given name on the same machine as the caller and with a virtual start time (start_time) no later than the caller's current LVT. If no such process exists at the virtual time this call is made, it returns T_NO_PROCESS. The search is confined to processes in the same Tipc system as the caller.

```
t_process_id t_searchw_locally(process_name)
  char *process_name;
```

This function is the same as t_search_locally except it waits until the named process comes to exist. On return, the caller's LVT will be the searched-for process's start time if the caller's LVT at the time of the call was earlier than the start time.

```
t_process_id t_search_machine(machine_name,process_name)
  char *machine_name;
  char *process_name;
```

This function looks for a process with the given name on the specified machine. If no such process exists at the time (LVT) that this call is made, T_NO_PROCESS will be returned. The search is confined to the caller's system.

```
t_process_id t_searchw_machine(machine_name,process_name)
  char *machine_name;
  char *process_name;
```

The function is the same as t_search_machine, except that it waits until the named process comes to exist on the specified machine. On return, the caller's LVT will be the searched-for process's start time if the caller's LVT at the time of the call was earlier than the start time.

Other routines for process ids.

The following all deal with process ids in some way or another. There are routines for process identification, some for system identification, and some for extracting information from a process id.

```
t_process_id t_this_process()
```

This routine returns the Tipc process id of the caller, if the caller is in a Tipc system. If not, the call will fail.

```
t_process_id t_parent_process()
```

This routine returns the Tipc process id of the caller's parent. Both the caller and the parent must be Tipc processes. The caller must have been created by a Tipc process.

```
int t_same_process(pid1,pid2)
  t_process_id pid1,pid2;
```

This routine tells if two Tipc process ids refer to the same process.

```
int t_this_system()
```

This routine returns the number of the Tipc system that the caller inhabits.

```
int t_system(pid)
  t_process_id pid;
```

This routine returns the number of the Tipc system that the argument inhabits.

```
char *t_machine_name(pid,machine)
  t_process_id pid;
  char machine[T_MAX_MACHINE_NAME+1];
```

This routine returns the name of the machine that the process identified by the first parameter inhabits. This routine requires that a second parameter be passed. This is space for the machine name to be written to. Otherwise, the Tipc kernel would have to allocate space itself and worry about garbage collection.

```
char *t_process_name(pid,name)
  t_process_id pid;
  char name[T_MAX_PROCESS_NAME+1];
```

This routine returns the name of the Tipc process identified by the first parameter. As for t_machine_name, the second parameter must be the address of a block of space to write the name to.

5. Process Creation and Termination.

Tipc systems are dynamic, that is, processes can begin executing or die as the rest of the system runs. There are two ways for a process to begin running - it can be created by another process, or it may be started up from the Unix shell. To maintain Virtual Time semantics, the created process is required to start at a local virtual time at least as large as that of the creating process. Similarly, if a process started by the user tries to enter the system at a time less than that of all currently executing Tipc processes, it will not be allowed to run.

A Tipc process can also stop being a member of a Tipc system, either by voluntarily leaving the system, or by being killed by another Tipc process.

```
t_process_id t_create_process(time,name,program,arg,...,0)
  int time;
  char *name;
  char *program;
  char *arg,...;
```

This call creates a new Tipc process on the same machine and in the same system as the caller, with the name given as the second parameter. The new process will run the program whose object code is contained in the file named by the third parameter. The program will be passed the arguments given as fifth and following parameters (at most nine arguments). The creation time (or start time of the new process) will be the LVT at which the call is made plus the time passed as the first parameter. The id of the new process is returned as the return value.

The first parameter is the time interval between the current LVT when this procedure is called and the LVT at which the new process will start. If time is zero, the new process will start at the LVT equal to the caller's current LVT.

```
t_process_id t_create_process_shared(time,name,proc,arg,...0)
  int time;
  char *name;
  void (*proc)();
  char *arg,...;
```

This call creates a new Tipc process on the same machine and in the same system as the caller. The new process executes the procedure passed as the second parameter rather than loading and executing a new program.

```
t_process_id t_createm_process(time,machine,name,program,arg,...,0)
  int time;
  char *machine;
  char *name;
  char *program;
  char *arg,...;
```

> This routine is the same as t_create_process, except the new process is created on the machine given as the second parameter.

There are two ways to become a Tipc process - by entering into the system, or by being created by another Tipc process and then initializing. In either case, if a process is going to become a Tipc process, it must call the appropriate routine as the *very first* thing that it does.

```
void t_enter_system(no,name,time)
  int no,time;
  char *name;
```

> This routine is called from a non-Tipc process. If it succeeds, the calling process will be a process in the Tipc system whose number is given as the first parameter, with the name specified as the second parameter. The time that the calling process starts is given by the third parameter. If the time given is less than the current GVT or there are already too many processes, this call will fail and the caller dies immediately. It is very important that this call be the first thing done in the code for a Tipc process.

```
void t_initialize()
```

> This routine is called from a process that been created by another Tipc process. If there are too many processes on the same machine (maximum 30 including some system processes), the calling process will die. It is very important that this call be the first thing done in the code for a Tipc process.

There is an idiosyncracy shared by all Tipc processes which are created by another process, i.e., those that must do a t_initialize: they do not have *stdin, stderr,* or *stdout.* Therefore, input and output in these processes is not possible through the predefined Unix files. It is up to the ingenuity of the Tipc programmer to find a way around this limitation.

Once a Tipc process has finished its processing, it must inform the system that it is going to stop. There are two routines provided for this.

```
void t_exit()
```

> This procedure should be called at the end of a Tipc process. By calling this procedure, the Tipc package can be informed of the termination of the caller and apply proper procedures for cleaning things up.

```
void t_leave_system()
```

> With this routine, the caller signifies that it no longer wants to be a Tipc process. After the call, the caller will no longer be in the Tipc system, although it will be able to continue executing.

A Tipc process may kill another Tipc process in the same system and on the same machine with the call

```
void t_kill_process(pid)
  t_process_id pid;
```

> This routine kills the Tipc process identified by the parameter, provided that the target of the kill is in the same Tipc system as the caller, and on the same machine. Under Unix, it is not possible to kill a process in this way unless the caller has the same user id as the process to be killed, or is the super user.

6. Manipulating Message Buffers.

A Tipc *message buffer* is used to hold a message received from another process while the program is examining it or to contain a message destined for some other process while it is under construction.

Declaring and using message buffers.

Message buffers in Tipc are an "abstract data type," like process ids. You should access a message buffer only with the routines supplied in the Tipc library, not by trying to look into it directly. The C type "t_message_buffer" is defined in <tipc.h> for use in Tipc programs. One can declare (and allocate space for) a message buffer by a statement such as

```
t_message_buffer pending_req;
```

Before such a message buffer can be used, it must be initialized with the t_clear procedure described below. Failure to do so may result in random results.

Often, however, it is not necessary for a Tipc program to declare any message buffers, because a message buffer is automatically allocated by Tipc at initialization and used unless the program switches usage to a message buffer it has declared itself.

Many Tipc procedures interact with a message buffer, but few of them take a buffer as a parameter. Instead, they use the *current buffer*. Initially, the current buffer is the one automatically allocated by Tipc. A program may use the following routines to change this:

```
void t_use_buff(buffer)
  t_message_buffer *buffer;
```

This call changes the current buffer to the one whose address is given as the parameter. This does not change the contents of either the new or the old current buffer.

```
t_message_buffer *t_current_buff()
```

This function returns a pointer to the current buffer.

Buffers may be copied with the following procedure:

```
void t_copy_buff(from,to)
  t_message_buffer *from;
  t_message_buffer *to;
```

Copies the contents of the buffer whose address is given first to the buffer whose address is given second. This procedure copies only relevant portions of the buffer, and so may be faster than simply assigning one message buffer variable to another.

Message items.

A message consists of a number of *items*. Each item has a type, such as "integer," and a particular value of that type. Types are represented by symbols of C type "t_item_type," as follows:

T_INT	Integer in the range -2^31 to +2^31-1.
T_CHAR	Character (7-bit ASCII).
T_STRING	String of ASCII characters other than NUL.
T_BLOCK	Block of 8-bit bytes.
T_FLOAT	Floating-point number (precision and exponent range undefined).
T_ATOM	Small integer in the range 0 to 255.
T_PROCESS	Tipc process id (including T_NO_PROCESS).

An additional symbol, T_END, of type t_item_type is defined to represent the end of a message buffer.

The "atom" type is intended for use in building representations of higher-level data structures. It should not be used in place of the integer type, nor is there any efficiency reason to wish to do so.

The ability to send Tipc process ids in messages is an important mechanism by which processes may find out about other processes.

When messages are sent between unlike machines, Tipc automatically converts the items to the destination machine's internal representation.

A message may contain up to T_MAX_DATA bytes of data (this will be a few hundred). Items take a variable amount of space, from one byte for the integers zero and one to an indefinite number for strings and byte blocks. The exact amount of space taken by an item depends on details of the Tipc implementation not known to the application program (and subject to change).

Putting items in a message buffer.

To begin putting items in the current message buffer, a program can call the following routine:

```
void t_clear()
```

This routine sets the current message buffer to contain no items.
This routine must be called to initialize a message buffer allocated
by the application program before its first use.

After calling t_clear, the program may put items in the buffer using the following set of routines. Each call of one of these routines appends an item to the end of the buffer. If insufficient space exists in the message buffer for the item, an error results, and the caller is suspended until either it gets rolled back and reprocesses the buffer properly, or GVT passes through the LVT of the caller.

```
void t_puti(i)
  int i;
```

Puts an integer item in the current buffer.

```
void t_putc(c)
  char c;
```

Puts a character in the current buffer. Note that only ASCII characters are allowed (i.e., characters with codes 0 to 127).

```
void t_putf(f)
  double f;
```

Puts a floating-point item in the current buffer.

```
void t_puta(a)
  unsigned char a;
```

Puts a small integer into the buffer (range 0 to 255).

```
void t_putp(p)
  t_process_id p;
```

Puts a Tipc process id in the buffer. It is legal to pass T_NO_PROCESS.

```
t_puts(s)
  char *s;
```

Puts a string in the current buffer. The string is terminated by a null character. Hence strings cannot contain null. The zero-length string is legal.

```
t_putb(b,n)
  char *b;
  int n;
```

Puts a block of bytes in the current buffer. The number of bytes in the block is given as the second parameter; the start address of the block as the first parameter. Zero-length byte blocks are allowed.

Getting items from a message buffer.

To start looking at a message buffer, a program may call the routine

```
void t_reset()
```

This procedure sets the "read pointer" for the message buffer to the beginning, causing the next "get" from the buffer to apply to the first item.

Following a call of t_reset, the program may extract the values of items in the buffer with the following procedures. Successive calls look at successive items in the buffer. An error occurs if the next item in the buffer is not of the type corresponding to the particular "get" call made. An error also occurs if one of these routines is called when the read pointer is at the end of the buffer.

int t_geti()

> Returns the value of the next item in the buffer, which must be an integer.

char t_getc()

> Returns the value of the next item in the current buffer, which must be a character.

double t_getf()

> Returns the floating-point value next in the buffer. Theoretically, the value returned may have to be reduced in precision from that held in the buffer, or an exponent overflow or underflow condition might arise. At present, however, the floating-point formats on all the machines used by Jade have the same precision and exponent range, so this doesn't happen.

unsigned char t_geta()

> Returns the next item in the current buffer, which must be an "atom" item.

t_process_id t_getp()

> Returns the process id, which must be next in the buffer. May return T_NO_PROCESS. No guarantees are made by Tipc that this process id makes sense in any way; the process that sent the message being examined could have buggered things up.

```
void t_gets(s,l)
  char *s;
  int l;
```

> The next item in the current buffer must be a string. This call extracts that string and stores it at the address passed as the first parameter, with a null at the end. The second parameter is the amount of space that the caller has allocated at that address. If the string in the buffer is too long to store in this space, an error is generated. Note that since a null is used to terminate the string, the maximum length string that may be returned is one less than the value of the second parameter.

```
int t_getb(b,l)
  char *b;
  int l;
```

> The next item must be a byte block. This block will be stored starting at the address passed as the first parameter. The length of the block stored will be returned as the value of this procedure. The second parameter should be the amount of space allocated by the caller for the block. If the actual size of the block exceeds this,

the block will not be stored, but instead an error will be raised.

The next two procedures are useful when the program examining the message does not know the types or sizes of the items:

t_item_type t_next_item_type()

> Returns T_INT, T_CHAR, T_FLOAT, T_ATOM, T_PROCESS, T_STRING, or T_BLOCK to indicate the type of the next item in the current buffer, or it returns T_END if there is no next item. The read pointer is not advanced by this call.

int t_next_item_size()

> The next item in the buffer must be a string or byte block. This procedure returns the size of the string or block. The size returned for a string includes the null-terminating byte.

Implicit resetting and clearing of the buffer.

A message buffer has associated with it a *mode*, which is either "get" or "put". Calling t_clear sets a buffer to "put" mode; calling t_reset sets it to "get" mode. If a t_getX (or t_next_item_X) procedure is called when the buffer is in "put" mode, an automatic reset is performed, just as if t_reset had been called. If a t_putX procedure is called when the buffer is in "get" mode, an automatic clear is done, just as if t_clear has been called.

This means that one may put some items in a buffer and then look at them without calling t_reset between these two operations. One may also start putting new items in the buffer after having looked at the old items without explicitly calling t_clear.

The message-passing routines that put a received message in the current buffer also reset the buffer to allow immediate examination of it with t_getX calls.

This all means that, if one wishes, one may avoid calling t_reset and t_clear explicitly in almost all circumstances.

Mark and restore operations.

A program may save its current position in a message buffer with a procedure to "mark" the current position and restore to that point at a later time.

void t_mark()

> This call saves the current read or write position of the buffer.

void t_restore()

> This call sets the read or write pointer to the marked position. If the buffer is in "put" mode, any items after the marked position are discarded.

A clear or reset, either explicit or implicit, sets the "marked" position to the beginning of the buffer. Note that each message buffer has its own independent mark position. This facility is most useful in eliminating items from a buffer when it is found that the sequence of which they were a part would not completely fit in the buffer.

7. Manipulating Virtual Times.

LVT and GVT are maintained or computed inside the Tipc package. The user process can never directly alter them, only refer to them by value or pass them as parameters to the Tipc level (which is able to change them).

```
int t_get_time()
```

This call returns the current LVT of the caller.

```
void t_advance_time(time)
  int time;
```

This is the only function that the user process can use to advance the LVT. The time parameter is taken to be the interval that the user process wants to advance its LVT by.

```
void t_wait_for_gvt()
```

The caller is blocked until GVT advances. This routine is designed specifically for those processes that cannot be allowed to rollback, therefore the use of this routine must be restricted. A process that calls this routine may only receive messages. Usually it is not allowed to send messages. In other words, such a process acts like a sink.

Note: The restriction mentioned above has been relaxed as a result of recent changes. User processes using this routine may now send messages out, but we recommend following the restriction.

8. Running Tipc Programs.

Tipc is implemented as a library of routines linked with your program, which either directly perform the Tipc functions or interface to the underlying support system. We describe in this section how you get access to the Tipc routines and how your program uses them. Later sections describe the individual routines in detail.

Compiling and running your programs.

You should be able to compile a C program that uses Tipc as follows:

```
cc prog.c -ltipc -o prog
```

Of course, more complex commands would be used if the program had more than one source file. Just make sure to include the "-ltipc" library in the final link stage. The file prog.c would contain the line

```
#include <tipc.h>
```

at the beginning. In general, all source files that refer to Tipc entities must include this file.

Before you can run your Tipc program, you must start up the underlying Tipc system. There are some system utilities, described in the next section, that enable you to start up the underlying system over a collection of machines, find out what processes are running over those machines, and clean up the underlying system when you are done. Once the underlying system is started, you can run your Tipc programs.

Usually, you will have several programs that interact via Tipc. These should all be compiled independently. If the programs are all set up to enter a Tipc system themselves, they may then be run under Unix by a command such as

```
prog1 a b c & prog1 d e f & prog2 & prog3 x & prog3 y &
```

In the above example there are three programs and five processes. Two of the programs are run twice with different arguments. Alternatively, you could write the system so that all but one of the processes are created from other Tipc processes. In that case the command to start the system would be simply

```
start_prog &
```

The object files for the other processes would still have to exist of course, so that start_prog could create them with a j_create_process call.

Naming conventions.

The Tipc include file, <tipc.h>, contains declarations of the constants, types, and routines that make up the Tipc interface. The symbols for these all begin with either "t_" or "T_". This (we hope) ensures that they do not conflict with symbols in your program, with Unix library routines, or with other library routines.

The include file also contains some symbols beginning with "_t_" or "_T_" and the names of the fields within the structures representing Tipc types. You are *not* supposed to know about this information. If you write a program that refers to these "secret" symbols or directly references the fields of Tipc structures, it will likely cease working some fine morning when the Tipc implementation changes. You are also discouraged from using the actual values of Tipc constants rather than their symbols.

9. System support utilities.

In its current implementation, Tipc interacts with a support system of processes. These processes must be running before the Tipc program will run. There are utility programs for starting this system of processes, for obtaining a list of all processes in a Tipc system over all machines of interest, and for terminating all processes in a Tipc system, over all machines of interest.

Starting the underlying system.

The startup program for the underlying system is called *str.* It attempts to create the gvt_control process and the local_control processes. These, taken together, form the underlying support system for Tipc. There is one gvt_control process per Tipc system, and one local_control process per machine. The str program looks for the object code for the gvt_control and local control processes in the private jades directory in the user's home directory. For more details see the Jade manual on remote creation of Jipc processes (Volume 1, Chapter 1, section C.10, page 41). More specifically, if you had a program that ran on machines a, b, and c, you would need to inter the object code for the local_control process in the private jades directory on all three machines, and inter the gvt_control program in the private jades directory on one of the three machines. The object code for the local_control and gvt_control processes can be copied from /usr/local/jade/bin/local_control and /usr/local/jade/bin/gvt_control.

The str program requires the Tipc system number in which the program will run, the user id, the user's password, and a list of machines that the Tipc program will run on. The str program assumes that the user_id and password are consistent on all machines. Also, it creates the gvt_control process on the first machine in the argument list. Str is called as follows (the plus sign postfix means "one or more"):

```
str <sys> <user_id> <passwd> <machine>+
```

The preferred method of using the str program is to enclose it in a shell script and limit access to the shell script to yourself. This will prevent people from ascertaining your password.

Finding out about the Tipc system.

The information program for Tipc is called *jsr.* It prints out the process names of all Tipc processes in the specified Tipc system on all machines supplied on the command line. It is not as informative as the Jipc utility *js,* in that it doesn't print out what call is being made by each Tipc process, but it is still helpful in determining if processes have actually been created on other machines. Jsr is called as follows:

```
jsr <sys> <machine>+
```

Terminating the Tipc system.

Once you have finished running your Tipc system, or if a previous run has gotten hung up, the Tipc cleanup program, *krill,* should be run. The krill (it stands for "kill remote") program kills all Tipc processes in the specified system on all machines supplied on the command line. The invocation of this command is the same as that of the str command and the same remarks apply - it assumes that the password and user_id are consistent over all machines supplied in the argument list.

```
krill <sys> <user_id> <passwd> <machine>+
```

10. Tipc Summary Sheet.

Tipc data types and constants.

t_message_buffer	
t_process_id	
t_item_type	
T_NO_PROCESS	A t_process_id signifying no Tipc process
T_MAX_PROCESS_NAME	
T_MAX_MACHINE_NAME	

Message item types.

T_END	End of message
T_INT	Integer
T_CHAR	Character
T_STRING	Character string
T_BLOCK	Block of 8-bit bytes
T_FLOAT	Floating-point value
T_ATOM	Byte with conventional meaning
T_PROCESS	Process identification

Communication.

void t_send(to)	Send message in current buffer, wait for reply
t_process_id to;	Process to send to
void t_send_null(to)	Send empty message, wait for reply
t_process_id to;	Process to send to
void t_asend(to)	Send message in current buffer, do not wait for reply
t_process_id to;	Process to send to
void t_asend_null(to)	Send empty message, do not wait for reply
t_process_id to;	Process to send to
t_process_id t_receive_any()	Wait for message from any process
t_process_id t_maybe_receive_any()	Accept message from any process if there is one
t_process_id t_receive_any_pre(time)	Hold until time expires or message arrives
int time;	Amount of time the call spans
t_process_id t_receive(from)	Wait for message from process
t_process_id from;	Process to wait for message frm
t_process_id t_maybe_receive(from)	Accept message from process, if there is one
t_process_id from;	Process to check for message from
void t_reply(from)	Reply to a received message

t_process_id from;	Sending process
void t_reply_null(from)	Reply with an empty buffer to a received message
t_process_id from;	Sending process

Searching.

t_process_id t_search_locally(name)	Search for local process
char *name;	Name of process
t_process_id t_searchw_locally(name)	Search for local process, wait for it to exist
char *name;	Name of process
t_process_id t_search_machine(m,n)	Search for process on a machine
char *m;	Name of machine
char *n;	Name of process
t_process_id t_searchw_machine(m,n)	Search for process on machine, wait for it to exist
char *m;	Name of machine
char *n;	Name of process

Buffers.

void t_use_buff(b)	Change current message buffer
t_message_buffer *b;	The new current message buffer
t_message_buffer *t_current_buff()	Return pointer to current buffer
void t_copy_buff(b1,b2)	Copy buffer contents
t_message_buffer *b1;	Buffer to copy data from
t_message_buffer *b2;	Buffer to copy data to
void t_clear()	Make the current buffer empty
void t_reset()	Set current buffer's read pointer to beginning
void t_mark()	Mark present spot in current buffer
void t_restore()	Restore current buffer to marked spot
void t_puti(i)	Put INT item into current buffer
int i;	
void t_putc(c)	Put CHAR item into current buffer
char c;	
void t_putf(f)	Put FLOAT item into current buffer
double f;	
void t_puta(a)	Put ATOM item into current buffer
unsigned char a;	

void t_puts(s) char *s;	Put STRING item into current buffer

void t_putb(b,l)
char *b;
int l;
	Put BLOCK item into current buffer
	Beginning address of block
	Length of block

void t_putp(p) t_process_id p;	Put PROCESS item into current buffer
int t_geti()	Get INT item from current buffer
char t_getc()	Get CHAR item from current buffer
double t_getf()	Get FLOAT item from current buffer
unsigned char t_geta()	Get ATOM item from current buffer

void t_gets(s,l)
char *s;
int l;
	Get STRING item from current buffer
	Address to store string
	Number of bytes allocated at s

int t_getb(b,l)
char *b;
int l;
	Get BLOCK item from current buffer, return length
	Address to store block
	Number of bytes allocated at b

t_process_id t_getp()	Get PROCESS item from current buffer
t_item_type t_next_item_type()	Type of next item in current buffer
int t_next_item_size()	Size of next item in current buffer (STRING or BLOCK)

Obtaining information about processes.

int t_same_process(p1,p2) t_process_id p1,p2;	Tells if two process ids refer to the same process

char *t_machine_name(pid,mach)
t_process_id pid;
char mach[T_MAX_MACHINE_NAME+1];
	Return machine process is on
	Process id
	Place to store machine name

char *t_process_name(pid,name)
t_process_id pid;
char name[T_MAX_PROCESS_NAME+1];
	Return name of process
	Process id
	Place to store process name

int t_system(pid) t_process_id pid;	Return system number of process Process id
t_process_id t_this_process()	Return process id of caller
int t_this_system()	Return system number of caller

t_process_id t_parent_process()	Return parent process of caller

System entry, exit, creation, etc.

void t_initialize()	Initialize process created from Tipc
void t_enter_system(system,name,time)	Enter a Tipc system
int system;	System to enter
char *name;	Process name
int time;	Time of entry
t_process_id t_create_process(time,name,file,arg, ... , 0)	
	Create a process, return child's process id
int time;	Time that child comes to exist
char *name;	Name of child
char *file;	Absolute pathname of program of child
char *arg;	Unix-style program arguments, zero at end
t_process_id t_createm_process(time,mach,name,file,arg, ... , 0)	
	Create a process on specified machine, return child's pid
int time;	Time that child comes to exist
char *mach;	Machine to create child on
cahr *name;	Name of child
char *file;	Absolute pathname of program of child
char *arg;	Unix-style program arguments, zero at end
void t_exit()	Leave a Tipc system and terminate the caller
void t_leave_system()	Leave a Tipc system
void t_kill_process(pid)	Kill another Tipc process
t_process_id pid;	Id of process to kill

Time.

int t_get_time()	Return local virtual time of caller
void t_advance_time(time)	Advance local virtual time of caller
int time;	Amount to advance lvt
void t_wait_for_gvt()	Caller waits until global virtual time advances

REFERENCES

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

Xiao, Z., Unger, B., Cleary, J., Lomow, G., Xining, L., Slind, K.
"A Virtual Time System Built On Jade"
Research Report, Department of Computer Science, University of Calgary, September 1986.

Li, X., Unger, B. W.
"Languages for Distributed Simulation"
Proc. of the Conference on Simulation and AI, Simulation Series, 18 (3) 35-40, January 1987.

West, D., Lomow, G., Unger, B.W.
"Optimising Time Warp using the Semantics of Abstract Data Types"
Proc. of the Conference on Simulation and AI, Simulation Series, 18 (3) 3-8, January 1987.

Cleary, J. G., Lomow, G. A., Unger. B. W., Xiao, Z.
"Jade's IPC kernel for Distributed Simulation"
Proc. of the Association of Simula Users Conference, Calgary, Alberta, August 1985.