ROSS: Rensselaer's Optimistic Simulation System User's Guide

Christopher D. Carothers
David Bauer
Shawn Pearce

Department of Computer Science
Rensselaer Polytechnic Institute
Troy, NY 12180, U.S.A.
chrisc@cs.rpi.edu

August 8, 2002

Contents

1 Overview

ROSS is an acronym for Rensselaer's Optimistic Simulation System. It is a parallel discrete-event simulator that executes on shared-memory multiprocessor systems. ROSS is geared for running large-scale simulation models (i.e., 100K to even 1 million object models).

The synchronization mechanism is based on Time Warp. Time Warp is an optimistic synchronization mechanism develop by Jefferson and Sowizral [7, 8] used in the parallelization of discrete-event simulation. The distributed simulator consists of a collection of *logical processes* or LPs, each modeling a distinct component of the system being modeled, e.g., a server in a queuing network. LPs communicate by exchanging timestamped event messages, e.g., denoting the arrival of a new job at that server.

The Time Warp mechanism uses a detection-and-recovery protocol to synchronize the computation. Any time an LP determines that it has processed events out of timestamp order, it "rolls back" those events, and re-executes them. For a detailed discussion of Time Warp as well as other parallel simulation protocols we refer the reader to [5]

ROSS was modeled after a Time Warp simulator called GTW or Georgia Tech Time Warp[4]. ROSS helped to demonstrate that Time Warp simulators can be run efficiently both in terms of speed and memory usage relative to a high-performance sequential simulator.

To achieve high parallel performance, ROSS uses a technique call Reverse Computation. Here, the roll back mechanism in the optimistic simulator is realized not by classic state-saving, but by literally allowing to the greatest possible extent events to be reverse. Thus, as models are developed for parallel execution, both the forward and reverse execution code must be written. Currently, both are done by hand. We are investigating automatic methods that are able to generate the reverse execution code using only the forward execution code as input. For more information on ROSS and Reverse Computation we refer the interested reader to [2, 3]. Both of these text are provided as additional reading in the ROSS distribution.

2 Data Structures

ROSS allows for an application programmer to design an application with the ROSS API which will simulate some real world event. The model that we use for testing and educational purposes is called *pcs*, which is short for "personal communications services". PCS simulates PCS/mobile phones calls being generated at a cell phone tower, and then moved around to other towers at some later time, and then the phone call ends. At the end of a phone call, another call or two may be generated. At the end of the simulation, statistics regarding how often calls were blocked (busy) due to limited tower capacity etc were collected. This details for this model can be found in [1]. This paper is included with this ROSS distribution.

ROSS utilizes three main data structures in C. There is a PE (short for processor) struct, an LP struct (short for Logical Process) and an event struct. It will be necessary to understand these structures if you are going to complete an application of your own design within ROSS.

2.1 Events

This is the most basic part of the ROSS system. An event is simply that, an event that occurs in the system. The ROSS engine operates on events in what we call time-stamp order. So, if we have three events with simulation timestamps of 5, 10 and 15, then the application developer would expect for the ROSS engine to process the event with a timestamp of 5 first, then 10 second and finally 15. Successful applications will first prime the system with a few events, and then ensure that when events are processed and leave the system that new events will be created. So, with this understanding of events, we need to know what data fields within the struct are available and important to us, the application designer.

```
/*
        * tw_event:
        *
        * Holds entire event structure, one is created for each and every
        * event in use.  We preallocate tw_message's into tw_event at
        * startup time so that there is no overhead of trying to locate the
        * tw_message storage area.  The tw_message MUST be the last item
```

```
         * in the tw_event.
        */

DEF(struct, tw_event)
{
```

These variables are strictly for event processing and are not part of the API.

```
        /*
         * next         -- Next pointer for all queues.
         * prev         -- Used by some queues.
         * cancel_next  -- Next event in the cancel queue for the dest_pe.
         * caused_by_me -- Start of event list caused by this event.
         * cause_list   -- List of events caused by the event with its
         *                 caused_by_me pointer set.
         */

        tw_event *volatile next;
        tw_event *volatile prev;
        tw_event *volatile cancel_next;
        tw_event *volatile caused_by_me;
        tw_event *volatile cause_list;
```

This is the simulation time stamp for this event. This would be 5, 10, or 15 in the example above.

```
        /*
         * recv_ts         -- Actual time to be received.
         */

        tw_stime        recv_ts;
```

This bit-field is used by the ROSS engine to determine in which data structure(s) the event is currently located. This is not part of the ROSS API.

```
        /*
         * Bit-field .. the event is currently located in the queue
         * denoted by the bit being set to 1.  The anti-msg bit is a
         * dual-case bit where the event was previously removed from the
         * event_q, but not the cancel_q
         */

        union
        {
                struct
                {
                        unsigned char    cancel_q;
                        unsigned char    abort;
                        unsigned char    anti_msg;
                        unsigned char    pq;
                        unsigned char    event_q;
                        unsigned char    pevent_q;
                        unsigned char    free_q;
                }
                b;
                unsigned int    uint;
        }
        state;
```

The `dest_lp` pointer is the destination LP (logical process) for this event. We also keep track of the source LP, some state-saving information and a bit-field per event. The bit-field is available in the ROSS API for application use and ROSS does not try to impose any limits on the logical functionality of this data type. The lp_state pointer contains lp information prior to the addition of an event.

```
        /*
         * dest_lp   -- Destination LP object.
         * src_lp    -- Sending LP.
         * lp_state  -- dest_lp->state BEFORE this event.
         * cv        -- Used by app during reverse computation.
         * seq_num   -- used by the hash table for remote sends
         */
        tw_lp          *dest_lp;
        tw_lp          *src_lp;
        void           *lp_state;
        tw_bf           cv;
```

The `tw_message` pointer is critical to an event because this is where the application data is kept. ROSS provides in the API for the proper allocation of this pointer.

```
        /*
         * message    -- Actual application data.
         */
        tw_message     *message;

};
```

2.2 Logical Process Data Structure

The LP (logical Process) struct is essential to the application programmer because it provides some logical environment for events to exist in. In PCS, events are cell phones, and LPs are cell phone towers. Cell phones initiate calls which require lines on a towers. Calls move from tower to tower. When a call ends, a line in a tower is freed. If we were to model the Internet using ROSS, events might be TCP/IP packets, and LPs might be routers.

```
        /*
         * tw_lp:
         *
         * Holds our state for the LP, including the lptype and a pointer
         * to the user's current state.  The lptype is copied into the tw_lp
         * in order to save the extra memory load that would otherwise be
         * required (if we stored a pointer).
         *
         * Specific PE's service specific LPs, each PE has a linked list of
         * the LPs it services, this list is made through the pe_next field
         * of the tw_lp structure.
         */

        DEF(struct, tw_lp)
        {
```

LPs can be many different types within ROSS. The application must specify these types and that information is recorded in the *type* variable you see here. LPs also have unique ids, and are assigned to PEs (processors). The application developer must create each lp and assign it to a PE. ROSS does no dynamic load balancing, so it is left to the programmer to determine where sinks and sources should be placed. These simulations typically work best with abstracted LPs which are essentially all the same. For example, a cell phone tower has channels available for placing calls. The cell tower lp is responsible for allocating and freeing its own channels. The ROSS API provides functions for assigning and viewing this information.

```
       /*
        * type     -- COPY of type from type array, holds func ptrs, etc.
        * id       -- ID number, otherwise its not available to the app.
        * pe       -- PE that services this LP.
        * pe_next  -- Next LP in the PE's service list.
        */
       tw_lptype       type;
       tw_lpid         id;
       tw_pe          *pe;
       tw_lp          *pe_next;
```

The cur_state pointer is used to determine the current application data for this lp. The cur_event pointer is the event currently being processed by this lp. These are used by the ROSS engine to determine which is the next event to process on an lp basis.

```
       /*
        * cur_state    -- Current application LP data.
        * cur_event    -- Current event being processed
        * state_qh     -- Head of [free] state queue (for state saving).
        */
       void           *cur_state;
       tw_event       *cur_event;
       tw_lp_state    *state_qh;
```

The kp pointer is the owning kernel process for this LP. The Kernel Process data structure will be discussed next. However, from the model developers point of view, you should not be directly accessing KP data fields.

```
       tw_kp          *kp;
```

This time element is the time of the last processed event in this lp. The ROSS engine uses this information to determine it's current time in the simulation and is not part of the ROSS API.

```
       /*
        * last_time        -- Time of the last PROCESSED event, NOT the last SENT event
        */
       tw_stime        last_time;
};
```

2.3 Kernel Processes Data Structure

A Kernel Process (KP) holds our state for the Kernel Process (KP), which consists only of processed event list for a collection of LPs.

```
DEF(struct, tw_kp)
{
        /*
         * id       -- ID number, otherwise its not available to the app.
         * pe       -- PE that services this KP.
         * kp_next  -- Next KP in the PE's service list.
         */
        tw_kpid        id;
        tw_pe          *pe;
        tw_kp          *kp_next;

        /*
         * last_time        -- Time of the last PROCESSED event, NOT the last SENT event
         */
        tw_stime        last_time;
```

This next series of variables is used to contain the processed event list. This is the list of events that is keep around for purposes of being able to rollback LPs. Please note, that because we are aggregating the processed event-list of many LPs, a single rollback at the KP level could result in many LPs being rolled back, so of which are not necessary. This is a trade-off in design between the need for efficient fossil collection and “false” rollbacks. Currently, “false” rollbacks do not appear to be an issue in the models we have tested to date. For more information on KPs see [3].

```
        /*
         * pevent_qh  -- Last event processed (for rollback).
         * pevent_qt    -- Oldest uncollected event procesed.
         */
        tw_event       *pevent_qh;
        tw_event       *pevent_qt;
```

And finally, we collect ROSS specific stats on a KP by KP basis.

```
        /*
         * s_nevent_processed       -- Number of events processed.
         * s_e_rbs                  -- Number of events rolled back by this LP.
         * s_rb_total               -- Number of total rollbacks by this LP.
         * s_rb_secondary           -- Number of secondary rollbacks by this LP.
         */
        long    s_nevent_processed;
        long    s_e_rbs;
        long    s_rb_total;
        long    s_rb_secondary;
};
```

2.4 Processing Element Data Structure

Most of what is contained within the PE (processor) struct is internal to the ROSS engine so it is not shown here. There is one field of note and that is the id field. Within a machine, each processor is assigned a unique PE id, starting from zero and going up. This is important to the application designer when mapping LPs to PEs.

3 ROSS Application Programming Interface (API)

The ROSS API is pretty small and gives the application designer a fair amount of flexibility. In this section we will discuss each of the functions available to the application programmer. We will note which functions are required where necessary. The first job of any application is to provide a main function for ROSS to be compiled against. In this main function we will need to call a couple required functions in order to setup the system properly. In this section we will use the PCS application to highlight how these functions are realized. We will start by going through the main function, and then each of the handlers (yes, this is event driven programming).

3.1 Main

The purpose of the `main` function is to map our LPs (logical processes) and KPs (kernel processes) to PEs (processors). This is necessary because we are going to be determine where to implement parallelism in the system. It is possible to place all of the LPs and KPs on a single PE, but then this would be considered the ’sequential case’, where we would only utilize one processor. This would be bad because we like to use multi-processor machines. This main function builds a square mapping of LPs to PEs. It could be considered a matrix, but why make things more complicated. Think of a map where cell phone towers are placed into a grid. Each grid space is really a processor with an lp inside of it. We may have many more cell phone towers than grid spaces, so some grid spaces will have many towers. We would like to achieve a mapping such that all grid spaces have an equal number of towers.

```
int
main(int argc, char **argv)
{
```

These are the local variables we will be using to determine the proper mapping. Because this is a grid, we will need an x and y coordinate. Also, we will have virtual x and y coordinates, xvp and yvp (for x-virtual processor and y-virtual processor). We have a simple counter, i, the number of LPs to map, TWnlp (Time Warp number of LPs), the number of processors in this machine to use, TWnpe (Time Warp number of PEs) and an lp pointer, lp (yet unallocated).

```
        int             x, y, xvp, yvp;
        int             neighbor_x[4], neighbor_y[4];
        int             vp_per_proc;
        int             i;
        int             TWnlp;
        int             TWnkp;
        int             TWnpe;
        tw_lp          *lp;
```

We have some `#define` variables at the top of this file to make changing the number of LPs easy to change..

```
        TWnlp = NUM_CELLS_X * NUM_CELLS_Y;
```

We must have a nice machine because there appears to be four processors in it. It may really only have two processors, in which case our simulation would be hampered by context switching. The TWnpe variable should really be set to the number of processors in a machine.

```
        TWnpe = 4;
```

Here we come across our first global variable, g_tw_ts_end (Global Time Warp Time Stamp Endtime). This must be a fast machine because we are telling the simulation here to run to 100 million time stamps. This is a required step if you like your simulations to end cleanly and to have statistics computed.

```
        g_tw_ts_end = 100000000.0;
```

This variable is the number of virtual processes per processor. The math here is not really important to us unless we are interested in doing these types of mappings. You'll notice however, that we are simply taking the number of grid spaces we desire and dividing that by the number of real processors in the system.

```
        vp_per_proc = ((double)NUM_VP_X * (double)NUM_VP_Y) / (double)TWnpe;
```

We love printf's. We love them more when they are followed by a fflush(stdout);

```
        printf("Running simulation for %f time units on %d processors.\n\n",
                (double) g_tw_ts_end, TWnpe);
```

This is the first function we are required to call in ROSS. The purpose of this function is to setup the ROSS internals for the simulation you desire. We pass through mylps, which is an array of lp types and their associated handlers, the number of processors to build PE structs for, and the number of KPs to build lp structures for as well as the number LPs to build lp structs for. We also send through the size of our application defined Msg_Data (message data) struct. This is where the real work of our application will be done. This is important, because ROSS will also be building a linked list of events equal in size to that specified internally and we would like those events to reflect our application data. The application programmer trusts ROSS internals to be configured properly, and so is only concerned that ROSS knows to allocate space for our message data.

```
    if (tw_init(mylps, TWnpe, TWnkp, TWnlp, sizeof(struct Msg_Data)))
       {
       for(x = 0; x < NUM_CELLS_X; x++)
          {
             for (y = 0; y < NUM_CELLS_Y; y++)
                {
                neighbor_x[0] = ((x - 1) + NUM_CELLS_X) % NUM_CELLS_X;
                neighbor_y[0] = y;
                neighbor_x[1] = (x + 1) % NUM_CELLS_X;
                neighbor_y[1] = y;
                neighbor_x[2] = x;
                neighbor_y[2] = ((y - 1) + NUM_CELLS_Y) % NUM_CELLS_Y;
                neighbor_x[3] = x;
                neighbor_y[3] = (y + 1) % NUM_CELLS_Y;

                for (i = 0; i < 4; i++)
                   Neighbors[x + (y * NUM_CELLS_X)][i] = neighbor_x[i] +
                      (neighbor_y[i] * NUM_CELLS_X);

                xvp = (int)((double)x * ((double)NUM_VP_X / (double)NUM_CELLS_X));
                yvp = (int)((double)y * ((double)NUM_VP_Y / (double)NUM_CELLS_Y));
```

After building the variables, we will allocate our lp pointer by calling the second ROSS functions we have seen so far, tw_getlp. All tw_getlp does is return to us a pointer to a specified lp. The lp id is a simple number (0, 1, 2 .. TWnlp). All of that fancy mapping from above does is give us a number which is an lp id. Similarly, we do the same to find the right KP to which we wish to map this LP to.

```
lp = tw_getlp(x + (y * NUM_CELLS_X));
kp = tw_getkp((x + (y * NUM_CELLS_X))/num_cells_per_kp);
```

Now that we have the appropriate pointers to LPs and KPs, we can set it's type. Since all PCS LP types are cell phone towers, all LP types will be TW_CELL, another #define.. in this case, TW_CELL is defined to be 1, or, the first element in the mylps array. PLEASE NOTE, YOUR LP TYPE NUMBERS CANNOT START WITH ZERO. THE ZERO TYPE IS USED TO DENOTE THE END OF THE ARRAY.

```
tw_lp_settype(lp, TW_CELL);
```

We now bind the LP to a particular KP using the previously computed KP and LP pointers. The `tw_lp_onkp` function accomplishes this mapping task.

Next, lets assign this lp to a processor. More fancy mapping here.. but you should notice two new ROSS API functions, `tw_lp_onpe` and `tw_getpe`. The purpose of `tw_getpe` is similar to `tw_getlp` and `tw_getkp`. It will return to us the PE pointer specified by a number. If we have four PEs, then we have PE ids 0, 1, 2 and 3. In this case, we had better be sure that the expression ((xvp + (yvp * NUM_VP_X)) / vp_per_proc) is going to reduce to one of these ids. If it does not, then `tw_getpe` will fail. All this mapping does is tell us which grid to place the lp into. We then use `tw_lp_onpe` to assign this lp to the PE we chose. PLEASE NOTE, YOU MUST MAKE SURE YOUR KP TO PE and LP TO PE MAPPINGS ARE CONSISTENT OR ROSS WILL GENERATE AN ERROR DURING STARTUP AND INITIALIZATION.

```
                tw_lp_onkp(lp, kp);
                tw_lp_onpe(lp, tw_getpe((xvp + (yvp * NUM_VP_X)) /
                                        vp_per_proc));
                tw_kp_onpe(kp, tw_getpe((xvp + (yvp * NUM_VP_X)) /
                                        vp_per_proc));
             }
       }
    }
```

Other than calling `tw_run()`, as you see below, that's really about it to the main function. Some initializing of the simulator, and some assignment of LPs to PEs and we're ready to go. Before we start running, we will be sure to zero out our application statistics variables.

```
        /*
         * Initialize App Stats Structure
         */
        TWAppStats.Call_Attempts = 0;
        TWAppStats.Call_Attempts = 0;
        TWAppStats.Channel_Blocks = 0;
        TWAppStats.Busy_Lines = 0;
        TWAppStats.Handoff_Blocks = 0;
        TWAppStats.Portables_In = 0;
        TWAppStats.Portables_Out = 0;
        TWAppStats.Blocking_Probability = 0.0;

        tw_run();
```

If we were lucky and our simulation completed without error, we can then compute and print our statistics. ROSS knows nothing about our application, so these are application defined functions. ROSS has it's own statistics that it keeps.

```
        CellStatistics_Compute(&TWAppStats);
        CellStatistics_Print(&TWAppStats);

        return 0;
}
```

A final note on the main function before we move on to some of the user defined functions and data types. It is not necessary to try to complete such difficult mappings. If we knew we had only four LPs, four KPs and four PEs, we could have reduced that mapping down to 12, nearly identical lines:

```
         tw_lp_onpe (tw_getlp(0), tw_getpe(0);
         tw_lp_onpe (tw_getlp(1), tw_getpe(1);
         tw_lp_onpe (tw_getlp(2), tw_getpe(2);
         tw_lp_onpe (tw_getlp(3), tw_getpe(3);

         tw_lp_onkp (tw_getlp(0), tw_getkp(0);
         tw_lp_onkp (tw_getlp(1), tw_getkp(1);
         tw_lp_onkp (tw_getlp(2), tw_getkp(2);
         tw_lp_onkp (tw_getlp(3), tw_getkp(3);

         tw_kp_onpe (tw_getkp(0), tw_getpe(0);
         tw_kp_onpe (tw_getkp(1), tw_getpe(1);
         tw_kp_onpe (tw_getkp(2), tw_getpe(2);
         tw_kp_onpe (tw_getkp(3), tw_getpe(3);
```

One observation about this LP/KP/PE mapping interface is that it is *overkill.* An alternative interface would be to only specify the LP-to-KP mapping and then map KPs to PEs. However, this precludes any sort of consistency checking, since the developer will typically want to determine which LPs are bound to a particular PE. The KPs are merely a logically grouping of LPs. Thus, the system can check both LP to PE mapping and KP to PE mappings and provide the developer with proper feedback.

3.2 Event Handler Data Structure

Next we'd like to view some of this application and try to clarify how we make something meaningful out of this simulator. We talked about LPs having types. In PCS we had one type, TW_CELL. Here we see where that comes from.

TW_CELL is simply a `#define` which allows us to access the first element in the array `mylps`. In `mylps`, we have declared our event handlers. There are five major events handlers in ROSS, initialization, processing, rollback, and a final state. Remember that ROSS executes speculatively across processors, and that from time to time processors may process events out of order, in which case the system will need to be able to rollback the offending events, process the out of order event, and then begin executing again. This is why we require a rollback event handler in ROSS. We also have a state-saving handler which is not used in ROSS because state-saving is not as efficient a technique and more conservative an approach versus reverse computation.

```
#define       TW_CELL       1

tw_lptype       mylps[] =
{
       {TW_CELL, sizeof(struct State),
        (init_f) Cell_StartUp,
        (event_f) Cell_EventHandler,
        (revent_f) RC_Cell_EventHandler,
        (final_f) CellStatistics_CollectStats,
        (statecp_f) NULL},
       {0},
};
```

From `ross-types.h` header file, the `tw_lptype` structure looks as follows.

```
        /*
         * User implements virtual functions by giving us function pointers
         * for setting up an LP, handling an event on that LP, reversing the
         * event on the LP and cleaning up the LP for stats computation/collecting
         * results.
         */
typedef void    (*init_f) (void *sv, tw_lp * me);
typedef int     (*event_f) (void *sv, tw_bf * cv, void *msg, tw_lp * me);
typedef void    (*revent_f) (void *sv, tw_bf * cv, void *msg, tw_lp * me);
typedef void    (*final_f) (void *sv, tw_lp * me);
typedef void    (*statecp_f) (void *sv_dest, void *sv_src);

        /*
         * Each LP must have an lptype, consisting of the LP name and function
         * pointers.  Each LP will contain a COPY of this data after startup,
         * allowing the LP to quickly jump into each function.
         *
         *  name        -- Must be any non-zero value.
         *  state_sz    -- Number of bytes that SV is for the LP.
         *  init        -- LP setup routine.
         *  event       -- LP event handler routine.
         *  revent      -- LP RC event handler routine.
         *  final       -- LP final handler routine.
         *  statecp     -- LP SV copy routine.
         */
DEF(struct, tw_lptype)
{
        int             name;
        size_t          state_sz;
        init_f          init;
        event_f         event;
        revent_f        revent;
        final_f         final;
        statecp_f       statecp;
};
```

Alternatively, you could declare and initialize your `tw_lptype` array as follows as well in the `main` routine:

```
        tw_lptype mylps[4];

        mylps[0].name = 1;
        mylps[0].state_sz = sizeof(struct State);
        mylps[0].init = Cell_StartUp;
        mylps[0].event =  Cell_EventHandler;
        mylps[0].revent = RC_Cell_EventHandler;
        mylps[0].final = CellStatistics_CollectStats;
        mylps[0].statecp = NULL;
        mylps[1].name = 0;
```

3.3 Global Variable and Function Declarations

Before we look at one of these handlers, lets first see what the application variables are so that we will be familiar with them when looking through the code. Below we will list every variable in *pcs* noting the intent of the application programmer.

All ROSS applications must include ross.h if they would like to have their application work.

```
#include <ross.h>
```

These `#defines` are application specific and determine the number of LPs in both the x and y direction. They are used, as we saw, in the mapping of LPs and KPs to PEs. In this case we see a mapping of 4 LPs (2 X 2) to 4 PEs (2 X 2).

```
#define TW_MAX_NAME_LEN 31

#define NUM_CELLS_X 2
#define NUM_CELLS_Y 2

#define NUM_VP_X 2
#define NUM_VP_Y 2
```

Channels are phone lines in a cell tower. Here we see that each cell tower will be able to support 10 simultaneous phone calls with no reserve channels.

```
#define MAX_NORMAL_CHANNELS 10
#define MAX_RESERVE_CHANNELS 0
```

We must define how large a time step to make in the simulation for certain events. Here, we define a cell phone moving from one tower to another to take 4500 time stamp units. The next call average to be 360 time stamp units, and the average call length to be 180 time step units, which in this case is equated to seconds in the real system (i.e., the average phone last 180 seconds or 3 minutes).

```
#define MOVE_CALL_MEAN 4500.0
#define NEXT_CALL_MEAN 360.0
#define CALL_TIME_MEAN 180.0
```

This is the number of initial calls to start with per cell phone tower. Since this application came from Georgia Tech, and we wanted to preserve it in ROSS, we did not try to rename variables and functions to something more meaningful.

```
#define BIG_N (double)16.0
```

Here we have more application logic. These values were picked to have logical meanings, but you can see that they are all basically integer values.

```
/*
 * When Normal_Channels == 0, then all have been used
 */
#define NORM_CH_BUSY ( !( SV->Normal_Channels & 0xffffffff ) )

/*
 * When Reserve_Channels == 0, then all have been used
 */
#define RESERVE_CH_BUSY ( !( SV->Reserve_Channels & 0xffffffff ) )

typedef int     Channel_t;
typedef int     Min_t;
typedef int     MethodName_t;

#define NONE 0
#define NORMAL_CH 1
#define RESERVE 2

#define COMPLETECALL 3
#define NEXTCALL 4
#define MOVECALL 5

#define NEXTCALL_METHOD 6
#define COMPLETIONCALL_METHOD 7
#define MOVECALLIN_METHOD 8
#define MOVECALLOUT_METHOD 9
```

This is the state of a cell tower (lp). We said earlier that our application would have to be responsible for maintaining it's own logic and we do that with this struct. This is not to be confused with state-saving. We can see here the developer is tracking the number of portables that have come through a tower, the number of attempts which are made to the tower, the number of blocks (busy signals) given, and of course the cell tower location on the grid.

```
struct State
{
       double          Const_State_1;
       int             Const_State_2;
       int             Normal_Channels;
       int             Reserve_Channels;
       int             Portables_In;
       int             Portables_Out;
       int             Call_Attempts;
       int             Channel_Blocks;
       int             Busy_Lines;
       int             Handoff_Blocks;
       int             CellLocationX;
```

```
        int             CellLocationY;
};
```

This is a reverse computation, while-one loop, struct.

```
struct RC
{
        int             wl1;
};
```

Here is something we recognize from the initialization steps. We allocated space in every event we allocate for this struct, but what does it do' Here, it keeps track of the method name to run, the time stamp at which the call moved, ended, started a new call. We also log the channel type, or phone line type and the RC struct value.

```
struct Msg_Data
{
        MethodName_t    MethodName;
        double          CompletionCallTS;
        double          NextCallTS;
        double          MoveCallTS;
        Channel_t       ChannelType;
        struct RC       RC;
};
```

Here we find a struct filled with variables for keeping tracking of statistics. When the simulation ends, we can call the statistics compute function and fill in these values for the entire system.

```
struct CellStatistics
{
        int             Call_Attempts;
        int             Channel_Blocks;
        int             Busy_Lines;
        int             Handoff_Blocks;
        int             Portables_In;
        int             Portables_Out;
        double          Blocking_Probability;
};
```

Function declarations are included to show how each function might be defined. In this document we are not going to cover each function, but only those which illustrate useful ROSS API characteristics.

```
void Cell_EventHandler(struct State *SV, tw_bf * CV, struct Msg_Data *M, tw_lp * lp);
Min_t Cell_MinTS(struct Msg_Data *M);
void Cell_StartUp(struct State *SV, tw_lp * lp);
void Cell_NextCall(struct State *SV, tw_bf * CV, struct Msg_Data *M, tw_lp * lp);
void Cell_CompletionCall(struct State *SV, tw_bf * CV, struct Msg_Data *M, tw_lp * lp);
void Cell_MoveCallIn(struct State *SV, tw_bf * CV, struct Msg_Data *M, tw_lp * lp);
void Cell_MoveCallOut(struct State *SV, tw_bf * CV, struct Msg_Data *M, tw_lp * lp);
void RC_Cell_EventHandler(struct State *SV, tw_bf * CV, struct Msg_Data *M, tw_lp * lp);
void RC_Cell_NextCall(struct State *SV, tw_bf * CV, struct Msg_Data *M, tw_lp * lp);
void RC_Cell_CompletionCall(struct State *SV, tw_bf * CV, struct Msg_Data *M, tw_lp * lp);
void RC_Cell_MoveCallIn(struct State *SV, tw_bf * CV, struct Msg_Data *M, tw_lp * lp);
void RC_Cell_MoveCallOut(struct State *SV, tw_bf * CV, struct Msg_Data *M, tw_lp * lp);
void CellStatistics_CollectStats(struct State *, tw_lp *lp);
void CellStatistics_Compute();
void CellStatistics_Print();
int Neighbors[NUM_CELLS_X * NUM_CELLS_Y][4];
```

This final variable is a global struct by which we will collect statistics at the end of the simulation.

```
struct CellStatistics TWAppStats;
```

3.4 Initialization

Going back to the `mylp` array, recall that the initialization function was called `Cell_StartUp`. When we call `tw_run()`, ROSS will go through each LP invoking it's assigned initialization function. We are passed through a state vector and a pointer to the corresponding LP.

```
void
Cell_StartUp(struct State *SV, tw_lp * lp)
{
   int             currentcell = 0;
   int              newcell = 0;
   int             i;
   int              dest_index = 0;
   tw_stime          ts;
   struct Msg_Data TMsg;
   struct Msg_Data * TWMsg;
   tw_event *CurEvent;

   SV->Normal_Channels = MAX_NORMAL_CHANNELS;
   SV->Reserve_Channels = MAX_RESERVE_CHANNELS;
   SV->Portables_In = 0;
   SV->Portables_Out = 0;
   SV->Call_Attempts = 0;
   SV->Channel_Blocks = 0;
   SV->Handoff_Blocks = 0;
   SV->Busy_Lines = 0;
   SV->Handoff_Blocks = 0;
   SV->CellLocationX = lp->id % NUM_CELLS_X;
   SV->CellLocationY = lp->id / NUM_CELLS_X;
   if (SV->CellLocationX >= NUM_CELLS_X ||
       SV->CellLocationY >= NUM_CELLS_Y)
      {
       tw_error(TW_LOC, "Cell_StartUp: Bad CellLocations %d %d \n",
                SV->CellLocationX, SV->CellLocationY);
      }
```

This is where we prime the system with events or phone calls. `GenInitPortables` simply returns that `BigN` variable we've already seen, so we know that each LP will start out with 16 mobile subscribers/portables which are realized as events.

```
   SV->Portables_In = GenInitPortables(lp);
   for (i = 0; i < SV->Portables_In; i++)
      {
      TMsg.CompletionCallTS = HUGE_VAL;
```

ROSS includes a reversible random number generator, and here we see the ROSS API call to generate an exponential value. We will use as a seed the lp id and time stamp step for moving a call. We will be returned a time stamp value which is the actual time stamp step for moving this call. A complete listing of all random number distributions and their respective interfaces will be included at the end.

```
      TMsg.MoveCallTS = tw_rand_exponential(lp->id, MOVE_CALL_MEAN);
      TMsg.NextCallTS = tw_rand_exponential(lp->id, NEXT_CALL_MEAN);
```

We need to determine the status of this call. It may already be completed, which would be in error because we are just beginning the simulation. It may generate a new call, or move to another tower.

```
    switch (Cell_MinTS(&TMsg))
       {
       case COMPLETECALL:
            tw_error(TW_LOC, "APP_ERROR(StartUp): CompletionCallTS Is Min");
            break;
```

If we are generating a new call, which we need to do in order for the simulation to continue advancing, we want to create a new event and place it into the system. So we have four new, and most widely used ROSS API functions, `tw_event_new`, `tw_event_send`, `tw_event_data` and `tw_now`.

The function `tw_now` will return to us the current time (in the simulation) for an LP. Remember that this may be out of sync with other LPs because we are speculatively executing.

We create a new event by calling `tw_event_new(tw_lp * destination, tw_stime ts, tw_lp * sender)` where `sender` is the current LP and `destination` is the destination LP. `ts` is the time stamp at which this event was sent into the *future* and is the time at which the destination LP will receive it.

We now need this new events message data pointer so that we can fill it in. We call `tw_event_data`, passing to it the event we want the Msg_Data pointer for, and are returned that pointer. We begin filling it in with whatever application data we like.

We then schedule the event into the future by calling `tw_event_send`. In this case, we are always sending the event to ourselves because the source and destination LPs are the same.

```
       case NEXTCALL:
            ts = max(0.0, TMsg.NextCallTS - tw_now(lp));
            CurEvent = tw_event_new(lp, ts, lp);
            TWMsg = (struct Msg_Data *) tw_event_data(CurEvent);
            TWMsg->CompletionCallTS = TMsg.CompletionCallTS;
            TWMsg->MoveCallTS = TMsg.MoveCallTS;
            TWMsg->NextCallTS = TMsg.NextCallTS;
            TWMsg->MethodName = NEXTCALL_METHOD;
            tw_event_send(CurEvent);
            break;
```

Here we would like to move a call to another tower, or LP. First we want to figure out where to send this event to, so we call `tw_rand_integer` (remember all LP identifiers are integers) and set that as our destination index. We then use our neighbor array to find a neighbor to this LP. Once we have done all this, we will have an integer value for `newcell`. We see `tw_get_lp` being used again to give us the pointer to an lp from the value of newcell. We will again create a new event with this information as the destination LP, the time the event was received into the system, and of course our source LP. It is important to note here that when determining the time stamp offset at which the new event will enter the system must be at least zero. We would not want this called to be moved backwards in time. Again we fill in the message data fields with application data and send the event to the destination LP. Because we mapped LPs to PEs, ROSS will handle the mechanism of actually moving events around in the system and processing events in time stamp order.

```
       case MOVECALL:
            newcell = lp->id;
            while (TMsg.MoveCallTS < TMsg.NextCallTS)
               {
               double result;
               currentcell = newcell;
               dest_index = tw_rand_integer(lp->id, 0, 3);
               newcell = Neighbors[currentcell][dest_index];
               result = tw_rand_exponential(lp->id, MOVE_CALL_MEAN);
               TMsg.MoveCallTS += result;
```

```
                }

            ts = max(0.0, TMsg.NextCallTS - tw_now(lp));
            CurEvent = tw_event_new(tw_getlp(newcell), ts, lp);
            TWMsg = (struct Msg_Data *)tw_event_data(CurEvent);
            TWMsg->CompletionCallTS = TMsg.CompletionCallTS;
            TWMsg->MoveCallTS = TMsg.MoveCallTS;
            TWMsg->NextCallTS = TMsg.NextCallTS;
            TWMsg->MethodName = NEXTCALL_METHOD;
            tw_event_send(CurEvent);
            break;
        }
    }
}
```

And that's all there is to initializing the system! At this point ROSS has 4 PEs, each with a single lp. Each lp has been primed with 16 events or more. Now it is a simple matter for ROSS to being emptying lp queues and processing events. An important attribute not to be overlooked here is that if your events to not generate at least one other event before leaving the system, the simulation may be starved and run indefinitely, having not enough events to reach the end time that you specified in main. So it is important to continually generate events. Equally important is to realize that if each event generates too many events before leaving the system, then ROSS may not be able to allocate enough buffers and never progress, and the system will be considered 'choked'. So it is important for the application programmer to realize a 1:1 or 1:2 mapping of events leaving to events generated.

3.5 Event Handlers

The next function which is important to look at is the event handler routine. ROSS will dequeue an event from an lp struct and try to process it, or call your handler. So these handlers must work properly.

Here we see a more complex function. Its parameters include the state vector, a bit field (which is provided to your application), the message data and of course the current lp this event is scheduled onto. We simply look at the message data to determine which function we would like to call. In Cell_StartUp we set the MethodName field in all of our events, and here we exercise that information. Relatively simple. You will also note that there are a few other things going on here. First, we set the reverse-computation, while-one loop to zero. Also, we have two new API calls, tw_error and tw_exit. ROSS tries to provide some convenient way of handling errors. Every tw_error call's first parameter is TW_LOC, which is the file location of this call. Second is the string parameter, and third are the fields for the string. The tw_exit function is provided and should be used to ensure proper thread exiting on shutdown.

```
void
Cell_EventHandler(struct State *SV, tw_bf * CV, struct Msg_Data *M, tw_lp * lp)
{
        *(int *)CV = (int)0;
        M->RC.wl1 = 0;

        switch (M->MethodName)
        {
        case NEXTCALL_METHOD:
               Cell_NextCall(SV, CV, M, lp);
               break;
        case COMPLETIONCALL_METHOD:
               Cell_CompletionCall(SV, CV, M, lp);
               break;
        case MOVECALLIN_METHOD:
               Cell_MoveCallIn(SV, CV, M, lp);
               break;
        case MOVECALLOUT_METHOD:
                Cell_MoveCallOut(SV, CV, M, lp);
                break;
        default:
                tw_error(TW_LOC, "APP_ERROR(8)(%d): Invalid MethodName(%d)\n",
                             lp->id, M->MethodName);
                tw_exit(1);
        }
 }
```

You probably thought that completing a telephone call was a simple matter, but for simulation designers it is where the work really begins. It is the point at which all the information is known.

```
void
Cell_CompletionCall(struct State *SV, tw_bf * CV, struct Msg_Data *M, tw_lp * lp)
 {
        int             dest_index = 0;
        int             currentcell = 0, newcell = 0;
        struct Msg_Data TMsg;
        double          result;
        tw_stime          ts;
        struct Msg_Data *TWMsg;
        tw_event       *CurEvent;
```

First we collect all our application message data.

```
        TMsg.MethodName = M->MethodName;
        TMsg.ChannelType = M->ChannelType;
        TMsg.CompletionCallTS = HUGE_VAL;
        TMsg.NextCallTS = M->NextCallTS;
        TMsg.MoveCallTS = M->MoveCallTS;

        if ((CV->c1 = (NORMAL_CH == M->ChannelType)))
                SV->Normal_Channels++;
        else if ((CV->c2 = RESERVE == M->ChannelType))
                SV->Reserve_Channels++;
        else
        {
                tw_error(TW_LOC, "APP_ERROR(2): CompletionCall: Bad ChannelType(%d) \n",
                             M->ChannelType);
                tw_exit(1);
        }
        if (SV->Normal_Channels > MAX_NORMAL_CHANNELS ||
                SV->Reserve_Channels > MAX_RESERVE_CHANNELS)
        {
                tw_error(TW_LOC, "APP_ERROR(3): Normal_Channels(%d) > MAX %d OR
                                 Reserve_Channels(%d) > MAX %d \n",
                                 SV->Normal_Channels, MAX_NORMAL_CHANNELS,
                                 SV->Reserve_Channels,
                                 MAX_RESERVE_CHANNELS);
 tw_exit(1);
}
TMsg.ChannelType = NONE;
```

Then we must generate new event(s) into the system.

```
        switch (Cell_MinTS(&TMsg))
        {
        case COMPLETECALL:
                tw_error(TW_LOC, "APP_ERROR(NextCall): CompletionCallTS(%lf) Is Min \n",
                              TMsg.CompletionCallTS);
                tw_exit(1);
                break;

        case NEXTCALL:
                ts = max(0.0, TMsg.NextCallTS - tw_now(lp));
                CurEvent = tw_event_new(lp, ts, lp);
                TWMsg = (struct Msg_Data *)tw_event_data(CurEvent);

                TWMsg->MethodName = TMsg.MethodName;
                TWMsg->ChannelType = TMsg.ChannelType;
                TWMsg->CompletionCallTS = TMsg.CompletionCallTS;
                TWMsg->NextCallTS = TMsg.NextCallTS;
                TWMsg->MoveCallTS = TMsg.MoveCallTS;

                TWMsg->MethodName = NEXTCALL_METHOD;
                tw_event_send(CurEvent);
                break;

        case MOVECALL:
                newcell = lp->id;
                while (TMsg.MoveCallTS < TMsg.NextCallTS)
                {
```

You'll notice we are incrementing the RC.wl1 variable for a move.

```
                        M->RC.wl1++;
                        currentcell = newcell;
                        dest_index = tw_rand_integer(lp->id, 0, 3);
                        newcell = Neighbors[currentcell][dest_index];

                        result = tw_rand_exponential(lp->id, MOVE_CALL_MEAN);
                        TMsg.MoveCallTS += result;
                }
                ts = max(0.0, TMsg.NextCallTS - tw_now(lp));
                CurEvent = tw_event_new(tw_getlp(currentcell), ts, lp);
                TWMsg = (struct Msg_Data *)tw_event_data(CurEvent);
                TWMsg->MethodName = TMsg.MethodName;
                TWMsg->ChannelType = TMsg.ChannelType;
                TWMsg->CompletionCallTS = TMsg.CompletionCallTS;
                TWMsg->NextCallTS = TMsg.NextCallTS;
                TWMsg->MoveCallTS = TMsg.MoveCallTS;
                TWMsg->MethodName = NEXTCALL_METHOD;
                tw_event_send(CurEvent);
                break;
        }
}
```

3.6 Reverse Computation

So what's left? Well, because ROSS is a speculative execution simulator, this means that we may get into trouble from time to time when LPs get ahead or fall behind one another. LPs may not always be of the same time. Some may be sinks and some may be sources. Some may never have events scheduled to them at all. ROSS guarantees the application designer that the system will run through all these inconsistencies, and without thrashing, but how does it take advantage of the rollback technology? Put simply, when ROSS realizes an out-of-order event, it runs back every event for that lp until it reaches the correct point in time when that event was to be executed. Because this out-of-order event is going to possibly change events in the future, those future events that we rolled back must be restored and rescheduled. Rescheduled is simple, but how can you restore an event without saving it?s original state? Simply run the formulas backwards. This is called reverse computation and is as difficult a task as it sounds, but ROSS simplifies it for the application programmer. The application programmer is required to provide ROSS with a proper rollback event handler. When ROSS rolls back an event, this handler is invoked and (hopefully) the event is restored to its original state. In order to accomplish this, ROSS provides a random number generator that is capable of being run backwards. This makes life easy for the developer, as you will see.

All we are doing here is determining which function was used when processing this event, and then calling that function?s appropriate reverse computation handler.

```
void
RC_Cell_EventHandler(struct State *SV, tw_bf * CV, struct Msg_Data *M, tw_lp * lp)
{
        switch (M->MethodName)
        {
        case NEXTCALL_METHOD:
                RC_Cell_NextCall(SV, CV, M, lp);
                break;
        case COMPLETIONCALL_METHOD:
                RC_Cell_CompletionCall(SV, CV, M, lp);
                break;
        case MOVECALLIN_METHOD:
                RC_Cell_MoveCallIn(SV, CV, M, lp);
                break;
        case MOVECALLOUT_METHOD:
                RC_Cell_MoveCallOut(SV, CV, M, lp);
                break;
        }
}
```

Let's look at one of those functions, specifically, the reverse computation of the function we have already seen, `Cell_CompletionCall`.

```
void
RC_Cell_CompletionCall(struct State *SV, tw_bf * CV, struct Msg_Data *M, tw_lp * lp)
{
        int             i;

        if (CV->c1)
                SV->Normal_Channels--;
        else if (CV->c2)
                SV->Reserve_Channels--;
```

Notice here that we finally use the reverse-computation, while one counter to determine how many times the random number generator was invoked. Every move call caused the RNG to be invoked twice, and so for each move call the RNG must be reversed twice. It would have been nice if this variable had been named something more meaningful, but again, we tried to maintain code integrity in order to prove results against GTW. The two applications were, for all intensive purposes, identical.

```
        for (i = 0; i < M->RC.wl1; i++)
        {
               tw_rand_reverse_unif(lp->id);
               tw_rand_reverse_unif(lp->id);
        }
 }
```

This is all there is to it. We simply need to reverse compute the application variables we set in Cell_CompletionCall. Which means decrement that which was incremented. We use the lp id to rewind the random number generator so that in the future, when this event is re-executed, it will come up with the same value as the first time it was executed. Why do we care about this random number generator? It doesn't seem to be doing much for us anyway.. but it is, because you'll remember that we used the value from the random number generator to compute the destination lp of the next event.

In this way we can guarantee that not only is the simulation processing events in order through reverse computation, but also that the simulation will be deterministic. If your simulation is not deterministic, then it is worthless because it will not produce accurate, consistent results.

3.7 Collecting Application Statistics/Results

So at this point there is only one remaining step to complete, and it is the most important step, statistic collection. In the application PCS we define a final state handler to complete this task, then call our own functions to compute the overall statistics for the simulation run. Here we simply fill in our global struct called `TwAppStats` that we created with each LP's state information.

```
void
CellStatistics_CollectStats(struct State *SV, tw_lp * lp)
{
       TWAppStats.Call_Attempts += SV->Call_Attempts;
       TWAppStats.Channel_Blocks += SV->Channel_Blocks;
       TWAppStats.Busy_Lines += SV->Busy_Lines;
       TWAppStats.Handoff_Blocks += SV->Handoff_Blocks;
       TWAppStats.Portables_In += SV->Portables_In;
       TWAppStats.Portables_Out += SV->Portables_Out;
}
```

Then we call our application-specific statistic computation function(s) and print out the statistics as you saw at the end of the main function, after `tw_run()`. Then we `tw_exit(0)` because we are done.

4 Glossary of Functions & Variables

4.1 User Set Variables

The following set of variables are set by the application at start-up.

- **g_tw_ts_end** is a end time of the simulation specified as a floating point number (64 bit precision). This must be set for all (parallel or sequential) simulations. *The default value is 1.0.*

- **g_tw_mblock** is the number of events processed without going to top of the scheduler loop. This variable only has meaning when the optimistic scheduler is used. *The default value is 16..*

- **g_tw_gvt_interval** is the number of times though the main scheduler loop before computing Global Virtual Time (GVT). It combined with the previous `g_tw_mblock` control the amount of time or number of events processed between successive GVT calculations. This variable is only used when the optimistic/parallel event scheduler is used. *The default value is 16.*

- **g_tw_events_per_pe** is the number events allocated to each processor. Each processor has it own pool of event memory is manages. For sequential execution, the application must allocate enough events such that the maximum event population of the simulation model can be supported. Typically this is just equal to the number of events scheduled at start-up multiplied by the number of LPs. For parallel execution, extra memory is required over what the sequential requires, but only slightly more. We recommend 2`g_tw_mblock`*`g_tw_mblock` as the amount extra. See [3] for a more complete discussion as to how we derived this rule of thumb. Note, this paper is included with the documentation in the ROSS distribution. *The default value is 2048.*

- **g_tw_nRNG_per_lp:** User set variable that contains the number of RNGs per LP. This is used to support multiple RNGs per LP. *The default value is 1.* See section below on random number generation.

4.2 Setup Functions

- **main()**: the application model must write its own main function, inside of which it initializes the simulation model.

- **tw_lp *tw_getlp(int lpid)**: This function provides the pointer to a LP given its number identifier.

- **tw_kp *tw_getkp(int kpid)**: This function provides the pointer to a KP given its number identifier.

- **tw_pe *tw_getpe(int peid)**: This function provides the pointer to a PE given its number identifier.

- **tw_init(tw_lptype * types, tw_peid npe, tw_kpid nkp, tw_lpid nlp, size_t msgsz)**: This is the primary initialize function of the system. It requires the application to provide, (i) an array of `tw_lptype` of which each data element is properly initialized (see previous section Data Variables), (ii) number of processors, (iii) number of kernel processes (KPs), (iv) number of logical processes (LPs), (v) and the largest message size needed by any event in bytes.

- **tw_lp_onpe(tw_lp * lp, tw_pe * pe)**: map a particular LP to a particular processor. Note, this mapping must be made consistent with the LP to KP mapping. That is you cannot have an LP mapped to a KP which located on a different processor than specified.

- **tw_lp_onkp(tw_lp * lp, tw_kp * kp)**: map a particular LP to a particular kernel process (KP). Note, this mapping must be made consistent with the LP to KP mapping. That is you cannot have an LP mapped to a KP which located on a different processor than specified.

- **tw_kp_onpe(tw_kp * kp, tw_pe * pe)**: map a particular KP to a particular processor. Note, this mapping must be made consistent with the LP to KP mapping. That is you cannot have an LP mapped to a KP which located on a different processor than specified.

- **tw_run(void)**: Once all all system initialization and LP/KP/PE mappings are established, this functions transfers control to the simulation engine and completes the initialization process and then executes the simulation. The simulation run is complete when this function returns.

4.3 Event Scheduling and Utility Functions

The following set of functions are used to schedule events into the future and are the core set of routines used to construct a simulation model.

- **extern tw_event *tw_event_new(tw_lp * dest, tw_stime offset_ts, tw_lp * sender)**: allocates a new event buffer and fills in the destination LP, amount of time into the future and source LP information. PLEASE NOTE, THE TIME IS AMOUNT OF TIME INTO THE FUTURE FROM TIME NOW. THUS, IT IS A RELATIVE TIME OR OFFSET TIME AND NOT AN ABSOLUTE TIME.

- **extern void *tw_event_data(tw_event * event)**: given an pointer to an event from `tw_new_event`, return the pointer to the data portion of that event buffer.

- **extern void tw_event_send(tw_event * event)**: given an pointer to an event from `tw_new_event` and its data portion is been appropriately filled in, schedule or send that event into the future at the specified time to the specified destination LP.

- **extern tw_stime tw_now(tw_lp * lp)**: given a pointer to an LP, return the current time of that LP.

- **extern void tw_error(TW_LOC, char *fmt,...)**: Generate an error and KILL the ROSS systems. The `TW_LOC` is a macro that will automatically show which SOURCE CODE file and LINE # within that file as to where the error occurred. The string denotes what the error is. PLEASE NOTE, THIS FUNCTION SHOULD NOT BE USED WHEN OPTIMISTIC/PARALLEL EXECUTION IS TURNED ON, AS ERRORS CAN OCCUR IN OPTIMISTIC PROCESSING WHICH ARE LATER ROLLED BACK. For sequential models only, it is fine to directly use this function in the simulation model code or in cases in parallel execution where you know there is an error which could not have been caused by optimistic event processing.

- **extern void tw_exit(int res)**: Terminates the simulation.

4.4 Random Number Generation Functions

The base uniform RNG is based on L'Ecuyer's Combined Linear Congruential Generator [9]. For all RNG functions, the LP identifier (lpid) is used to determine which set of seeds is to be used. In this simulator engine, each LP is given its own set of seeds to prevent correlations among seeds. The seed are approximately 2^{70} calls apart. The overall period of this generator is 2^{121}. The distributions where developed based on [6].

The RNG seeds are initialized based on a seed jumping technique that is possible with this generator. The technique is extremely fast. It is much faster than reading a file of pre-generated seeds. Additionally, when you want to have 100K or even 1 million LP simulations, such seed files are not very practical.

- **g_tw_nRNG_per_lp:** User set variable that contains the number of RNGs per LP. This is used to support multiple RNGs per LP. *The default value is 1.*

- **extern long tw_rand_integer(Generator lpid, long low, long high)**: returns a uniform random number between the ranges of `low` to `high` inclusive. In the case of multiple RNGs per LP, you index the RNG with the LP id times the g_tw_nRNG_per_lp plus the RNG offset.

- **extern long tw_rand_binomial(Generator lpid, long N, double P)**: returns the number of successes (i.e., less than probability P) of the N Bernoulli trials.

- **extern double tw_rand_exponential(Generator lpid, double Lambda)**: returns a exponentially distributed random number with mean, `Lambda`.

- **extern double tw_rand_gamma(Generator lpid, double shape, double scale)**: returns a Gamma distributed random number of a particular shape and scale.

- **extern long tw_rand_geometric(Generator lpid, double P)**: returns the number of trials up to and including the first success (i.e., less than probability P).

- **extern double tw_rand_normal01(Generator lpid)**: returns a *unit* or standard normal distributed random number, where the mean and standard deviation of the distribution are 0 and 1 respectively.

- **extern double tw_rand_normal_sd(Generator lpid, double Mu, double Sd)**: returns a normal distributed random number where the mean is Mu and the standard deviation is Sd.

- **extern double tw_rand_pareto(Generator lpid, double scale, double shape)**: returns a Pareto distributed random number of a particular shape. The `scale` parameter is used to adjust it to fit a particular mean. The mean of the distribution is $shape/shape - 1$.

- **extern long tw_rand_poisson(Generator lpid, double Lambda)**: returns the number of arrivals in a given time period t assuming some average arrival rate, $Lambda$.

- **void tw_rand_reverse_unif(Generator lpid)**: this routine is used to reverse or undo the uniform part of the RNG exactly one call. Thus, for distributions that make multiple calls to the RNG, you will need to save that information and call this function that many times in order to restore the RNG seeds to the state prior to the execution of this event. This routine is to only be used for reverse execution. It should not be used in the forward execution of the simulation.

5 Priority Queue

ROSS supports two priority queues: Calendar Queue and Heap. In a nutshell, you should used the Calendar Queue if the simulation model exhibits the following characteristics:

- Exponential, Gamma, Lognormal, or Pareto time stamp distribution.
- Few if any "ties" or simultaneous events.
- Event population remains relatively static throughout simulation run.

The simulation model fails to meet any of these characteristics, there is a strong chance the Calendar Queue will shift into running its "worse case" performance range, which is $O(n)$ for enqueue and dequeue operations. It's normal "best case" performance complexity is $O(1)$. The push-down heap has a rock solid $O(log(n))$ performance complexity for enqueue and dequeue operations and thus should be used in cases where the Calendar Queue exhibits worse-case performance.

To configure which queue is used, modify the Makefile in the *src* directory of the ROSS distribution. Here, set the *QUEUE* variable to either: `queue-calendar.o` for the Calendar Queue or `queue-heap.o` for the Push-Down Heap.

For a detailed performance study on different queue algorithms, we refer the interested reader to [10]

6 ROSS Errors

The following are typical errors or problems that could be generated from inside of ROSS.

- **Core Dumps**: The most common error is that your application will yield a segmentation fault/core dump. The cause of such an error usually has do with something in your application. Either initialization is not correct or insufficient event memory has been allocated. In any case, the best course of action is to put your application under a debugger (i.e., gdb) and see where the fault lies and then back track it to the source of the error. See below for more hints on model development.
- **GVT Same for 1000 times**. This error only occurs when optimistic/ parallel execution is turned on. In this case, the simulation model has reached a state where it can no longer make progress and thus the Global Virtual Time (GVT) computation is no longer advancing. The solution here is to increase either or both the `g_tw_gvt_interval` and `g_tw_mblock` parameters to allow more events to be processed per GVT epoch. Additionally, you may need to provide more optimistic memory event buffers by increasing `g_tw_events_per_pe`. If both of these fail, start looking at your simulation model.
- **Simultaneous Events**. In many discrete-event system, simultaneous events are possible even if they do not correlate to the physical system being modeled. The proper solution to handling simultaneous events is to simulate ALL possible permutations of event orderings and take an average of the outcome. As such, ROSS does not do anything special to handle tie events. That must be handled within the application. Please note, that having many tie events could result in the previously mentioned GVT advance problem.

7 Model Development Hints

- **Start Small**: In developing your model, keep the event population and overall size of the systems small so you can trace the event flow by hand. When you have even 1000s of events or 100s of LPs/simulation objects, it become to much and bugs are buried to deep within the overall system.

- **Start with Sequential Execution First**: Build your model for sequential execution only first. Make sure it yields correct results across a wide variety of configurations.
- **Become Proficient with a Debugger**: The GNU debugger (GDB) or the Data Display Debugger (front end to GDB) is recommended. As with any complex software system, a debugger is an invaluable tool. However, if you are trying to find an event-flow logic error, the debugger is probably not the best method. Log files are probably better in this case, see below.
- **Construct Event Log Files**: An excellent method to debug logical errors with in a simulation model is to create LP-specific log files. This will allow complete LP-event flows to be captured. It is advisable for the developer to append the LP number to the filename.

References

[1] C. D. Carothers, R. M. Fujimoto and Y-B. Lin, A case study in simulating PCS networks using time warp. In *Proceedings of the 9th Workshop on Parallel and Distributed Simulation*, June 1995, pages 87–94.

[2] C. D. Carothers, K. Perumalla and R. M. Fujimoto. Efficient Parallel Simulation Using Reverse Computation *ACM Transactions on Modeling and Computer Simulation*, volume 9, number 3, July 1999.

[3] C. D. Carothers, D. Bauer and S. Pearce. High-Performance, Low Memory, Modular Time Warp System, In *Proceedings of the 14th Workshop of Parallel on Distributed Simulation (PADS 2000)*, pages 53–60, May 2000.

[4] S. Das, R. Fujimoto, K. Panesar, D. Allison, and M. Hybinette. GTW: A Time Warp system for shared memory multiprocessors. In *1994 Winter Simulation Conference Proceedings*, pages 1332–1339, December 1994.

[5] R. M. Fujimoto. Parallel discrete-event simulation. *Communications of the ACM*, 33(10):30–53, October 1990.

[6] R. Jain. *The Art of Computer Systems Performance Analysis*, John Wiley and Sons, Inc. New York, 1991.

[7] D. R. Jefferson and H. Sowizral. Fast concurrent simulation using the Time Warp mechanism, part I: Local control. Technical Report N-1906-AF, RAND Corporation, December 1982.

[8] D. R. Jefferson. Virtual time. *ACM Transactions on Programming Languages and Systems*, 7(3):404–425, July 1985.

[9] P. L'Ecuyer and T. H. Andres. "A Random Number Generator Based on the Combination of Four LCGs." *Mathematics and Computers in Simulation*, volume 44, pages 99–107, 1997.

[10] R. Ronngren and Rassul Ayani. "A Comparative Study of Parallel and Sequential Priority Queue Algorithms", *ACM Transactions on Modeling and Computer Simulation*, volume 7, number 2, pages 157–209, April 1997.