Georgia Tech Time Warp (GTW Version 3.1) Programmer's Manual for Distributed Network of Workstations

Richard M. Fujimoto, Samir R. Das, Kiran S. Panesar,
Maria Hybinette and Chris Carothers
College of Computing
Georgia Institute of Technology
Atlanta, GA 30332-0280

GIT-CC-97-18
July 3, 1997

Abstract

This manual gives an introduction to writing parallel discrete event simulation programs for the Georgia Tech Time Warp (GTW) system (version 3.1). Time Warp is a synchronization mechanism for parallel discrete event simulation programs. GTW is a Time Warp simulation kernel implemented on distributed network of uniprocessor and shared memory multiprocessor machines.

Contents

GEORGIA TECH TIME WARP (GTW Version 3.1)
PROGRAMMER'S MANUAL

1 Introduction

This manual gives an introduction to writing parallel discrete event simulation programs for the Georgia Tech Time Warp (GTW) system (version 3.1). Time Warp is a synchronization mechanism for parallel discrete event simulation programs. GTW is a Time Warp simulation kernel implemented on shared memory multiprocessor machines, e.g., the SGI Power Challenge machines, the Kendall Square Research (KSR) and Sequent Symmetry.

The GTW application program interface is, by design, "lean and mean." The interface provides a minimal set of simulation primitives in order to allow application programmers great latitude in trying to squeeze as much performance out of the system as possible. The system includes few restrictions to prevent programmers from trying to do "tricky things" to optimize performance. Thus, a relatively simple, low-level interface is provided. The system does not currently support high level simulation constructs commonly found in commercial simulators, e.g., process-oriented simulation. Users looking for a simulator with a rich set of powerful (albeit slow) high level simulation primitives are advised to look elsewhere, or plan on building a significant amount of software on top of the GTW system.

In order to write a simulation program, you must be familiar with the basic concepts of parallel discrete event simulation in general, and the Time Warp mechanism in particular. For an introduction to these topics, see [3]. The material in that reference should provide sufficient background for the purposes of writing application programs for GTW. For additional details on the Time Warp mechanism, read Jefferson's seminal paper "Virtual Time" [4]. An early (albeit dated) description of the direct cancellation mechanism used in GTW is described in [2]. If you are running Time Warp under memory constraints, the memory management protocol used in GTW is described in [5]. Its implementation in GTW (again, somewhat dated) is described in [1].

As an application programmer for the GTW system, you need not know details of the internals of the GTW implementation beyond those which are discussed here. This manual does not assume prior parallel programming experience, but it is assumed the reader is proficient in programming in the C language. At present, only C is supported for application software development.

2 Structure of GTW Programs

A GTW Time Warp program consists of a collection of logical processes (LPs) that communicate by exchanging time-stamped event messages (we use the terms *event* and *message* synonymously). The execution of each LP is entirely message driven, i.e., any execution of application code is a direct result of receiving a message. LPs cannot "spontaneously" begin new computations without first receiving a message.

The Time Warp mechanism ensures that messages within individual processes are processed in timestamp order. In order to accomplish this, Time Warp automatically detects out of order processing of events within an LP, rolls back the incorrectly processed messages, and reexecutes them. Messages sent by rolled back computations must be *canceled*, which may in turn cause additional roll-backs. These roll-backs are implemented automatically within the GTW kernel, and are largely transparent to the application program. Because processes may be rolled back, the state variables (those variables that retain a value from one event to the next) must be checkpointed in order to return to an earlier state.

Application programs must specify the logical processes making up the system. This involves specifying the procedures that are executed to process events, and the state variables that they may reference. Also, the application program must define the messages that are exchanged between LPs.

The remainder of this manual is organized as follows. Section 3 describes the application code that makes up the logical processes, as well as initialization and "wrap up" of LPs, and mapping of LPs to physical processors (PEs). Section 4 discusses memory allocation and checkpointing state variables that are referenced by LPs. Section 5 discusses the primitives for creating and sending messages between LPs. The next two sections describe the random number generation library, and a complete example application illustrating the use of these primitives. Other miscellaneous items, e.g., libraries, configuration files, command line arguments, and termination, are discussed in section 9. Section 10 describes the sequential simulator that is provided for performance comparisons and initial debugging. Finally, a concise summary of the programmer's interface to the GTW system is included as an Appendix.

3 Logical Processes

Each logical process is based on an *event-oriented* (as opposed to process-oriented) simulation paradigm. This means each incoming message (event) causes an *event handler* procedure to be called, and the GTW kernel will not begin executing any other LP on this processor until the event handler procedure returns.

The first step in writing a parallel simulation program for GTW is to define the logical processes. This means specifying the behavior of each LP, and the variables it can access. The behavior of each LP is specified by at three procedures:

1. the *IProc()* procedure is called to initialize the LP at the beginning of the simulation,
2. the *Proc()* procedure is the event handler that is called whenever a new message is received by the LP, and
3. the *FProc()* procedure is optional, and is called at the end of the simulation run, e.g., to output statistics.

The behavior of an LP is specified by these procedures and the procedures that they call. Different LPs may use the same or different IProc(), Proc(), and FProc() procedures, however, each LP contains *at most* one of each type of procedure. If a single LP requires multiple event handlers, the Proc() procedure must determine which event handler is required for each message and call the appropriate procedure.

Each logical process is uniquely identified by an integer ID. If there are N LPs in the simulation, the LPs are assigned ID values of $0, 1, 2, ..., N-1$. At present, all

LPs must be specified at the beginning of the simulation, i.e., no facility for dynamic creation of LPs is provided.

The number of LPs may be larger than the number of physical processors available to the program. In general, several LPs will be mapped to each PE. At present, this mapping is *static*, i.e., each LP remains on the same PE throughout the execution of the program. The mapping of LPs to PEs must be explicitly specified by the application program when the program is initialized.

The execution of the simulation program moves through three phases: initialization, execution of the simulation, and wrap-up. Each of these are described next.

3.1 The Initialization Phase

There are two sub-phases of the initialization phase. The first is a global initialization and the second is the initialization of each logical process.

The global initialization is performed by a procedure called `TWInitAppl()` that must be defined by the application program. `TWInitAppl()` has two parameters, `argc` and `argv` that specify command line parameters that are passed to the application. These parameters have the same meaning as conventional C programs. Command line arguments are discussed later in section 9.3.

`TWInitAppl()` must set several variables defined in the kernel to tell the GTW system things such as the number of logical processes, which procedures are event handlers, etc. Specifically, the following variables must be initialized by the `TWInitAppl()` procedure:

1. `int TWnlp` indicates the number of LPs that will be used in the simulation.
2. `TWLP[]` is an array, indexed by the LP ID. `TWLP[i]` holds information concerning the LP with ID `i`. Each array element is a record (a C struct) with fields (discussed below) that must be initialized by `TWInitLP()`.
3. `int TWLP[i].Map` indicates the physical processor to which LP `i` is mapped. Physical processors are numbered 0, 1, 2, ... `TWnpe` -1, where `TWnpe` is a GTW kernel variable indicating the number of processors available to the application program.
4. `int *TWLP[i].IProc()` is a pointer to the procedure that is called to initialize LP `i`.
5. `int *TWLP[i].Proc()` is a pointer to the event handler for LP `i`. This procedure is called each time the LP has a message (event) to be processed.
6. `int *TWLP[i].FProc()` is a pointer to the "wrap-up" procedure for LP `i` that is called at the end of the simulation.
7. `struct MyLPState *TWLP[i].State` is a pointer to the state variables used by the LP. Unlike the other variables listed above, this need not be set in `TWInitAppl()`; it may be set in the `IProc()` procedure. `MyLPState` is an application defined structure specifying the contents of the LP's state.
8. `int IncrSave` is a boolean indicating if the LP will incrementally save state.

All LPs are required to specify an `IProc()` and `Proc()` procedure. The `FProc()` procedure need not be specified for LPs that do not require this function.

A sample `TWInitAppl()` procedure is shown in figure 1. The simulation is called ping-pong. This simulation consists of two LPs, "ping" and "pong". When one LP receives a message, it simply sends it back to the other LP after increasing the timestamp on the message by a value drawn from an exponentially distributed random variable.

In Figure 1, the procedure first maps both LPs to processor zero if one processor is available, or to processors zero and one if two processors are used. It then sets the `IProc()` (initialization) and `Proc()` (event handler) procedures for both LPs, and the `FProc()` (wrap-up) procedure for the Ping LP. The Pong LP does not use a wrap-up procedure in this example. In this example, `TWInitAppl()` does not allocate memory for the state vectors of the LPs. It will be seen later that this is performed by the `IProc()` procedures.

The second sub-phase of the initialization calls the `IProc()` procedure of each LP. This is done in parallel; each processor calls the `IProc()` procedure of the LPs that are mapped to it.

The `IProc()` procedure contains a single parameter, a pointer to the state vector for the LP (`TWLP[i].StateVector` for LP `i`). If the state vector has not yet been defined, N̂ULL is passed. The `IProc()` procedure will typically

1. allocate all memory used by the logical process, and indicate to the GTW kernel what portions should be automatically state saved,
2. initialize the state of the LP, and
3. send initial messages to get the simulation started.

The `IProc()` procedure executes at simulated time zero. The structure for a typical `IProc()` procedure is shown below. Note that the second parameter has to be named `MyPE`, otherwise the TWSend() macros will not compile.

```
MyIProc (SV, MyPE)
struct MyLPState *SV;
int MyPE;
{
   struct LPState * CurState;
   CurState = GState[MyPE].CurState;

   /* code for initializing LP defined here */
}
```

We will defer a more detailed explanation of this until later, after the memory allocation and message sending primitives are discussed. The first three lines of code initialize a pointer to the current LP state, and are mandatory.

3.2 The Simulation Phase

The simulation phase consists of calls to event handlers, one for each LP receiving a new message. The event handler may (1) modify state variables, and/or (2) schedule new events (i.e., send new messages) to itself or other LPs in the simulated future.

Timestamps in GTW have the GTW-defined type `TWTime`. Currently, `TWTime` is defined as a `double` (double precision floating point number).

The application-defined event handler procedure has two parameters:

```
TWInitAppl(argc, argv)
int argc;
char **argv;
{
   int IPing(), IPong(), Ping(), Pong(), FPing();
   int i;

   TWnlp = 2;         /* specify number of LP */

   /* LP to PE mapping */
   TWLP[0].Map = 0;
   if (TWnpe == 1) TWLP[1].Map = 0;
   else TWLP[1].Map = 1;

   /* set initialization, event handler, and wrap-up procedures */
   TWLP[0].IProc = IPing;
   TWLP[0].Proc  = Ping;
   TWLP[0].FProc = FPing;
   TWLP[0].IncrSave = TRUE;

   TWLP[1].Proc  = Pong;
   TWLP[1].IProc = IPong;
   TWLP[1].IncrSave = TRUE;

   /* get initial random number generator seeds from file into Seeds array */
   for (i=0; i<TWnlp; i++)
     TWRandInit (&(Seeds[i]), 0);

   TWMsgSize = sizeof(struct MyMsgData);
}
```

Figure 1: `TWInitAppl()` procedure for ping-pong example.

1. a pointer to the state vector for the LP, and
2. a pointer to the data portion of the message that has been received.
3. the PE the LP is executing on. We need this to access processor private variables.

For example, an event handler procedure might be defined as follows:

```
MyProc (SV, M, MyPE)
struct MyLPState *SV;
struct MyMsgData *M;         /* read-only */
int MyPE;
{
   struct LPState * CurState;
   CurState = GState[MyPE].CurState;

   /* code for event handler goes here */
}
```

In this example, `MyLPState` and `MyMsgData` are application defined types indicating the format of the state vector for the LP, and messages that are processed by the event handler. Note that the third parameter has to be named `MyPE`, otherwise it the TWSend() macros will not compile.

It is important to note that the contents of the message data passed to the event handler must not be modified. Modifying the message may lead to unpredictable results because the message data is not checkpointed, so if the event is later rolled back, modifications to the message data cannot be undone. One way to circumvent this restriction is to incrementally checkpoint the modified message. The kernel was designed with this restriction to avoid message copying. The existing GTW kernel does not perform *any* message copying, neither when sending a message nor when receiving one.

The following utility functions may be called by event handler procedures.

1. `TWTime TWNow()` indicates the current simulated time of the LP.
2. `int TWMe()` indicates the ID of the currently executing logical process. This primitive may be invoked by the `IProc()` and `FProc()` procedures as well as the event handler.
3. `TWQuit()` aborts the application program.
4. `TWTime TWEndTime()` Simulated time at which the simulation terminates.

3.3 The Wrap-Up Phase

The wrap-up phase consists of calling the `FProc()` procedure for each LP in the simulation. Currently, this portion of the execution is performed *sequentially* since the main purpose of this procedure is to generate output statistics, and parallel execution might result in interleaved outputs.

This application-defined procedure for wrap-up uses a single parameter, a pointer to the state vector for the LP. Once all of the `FProc()` procedures have been called, the simulation terminates.

An example wrap-up procedure is shown below.

```
MyFProc (SV, MyPE)
struct MyLPState *SV;
int MyPE;
{
   struct LPState * CurState;
   CurState = GState[MyPE].CurState;

   /* code for wrap up procedure defined here */
}
```

Finally, `TWWrapAppl` is called. This procedure is used to print out any application specific global variables computed during the LP wrap up.

4 Memory Allocation and Check-pointing

An important constraint on all GTW programs is that *no variable can be directly accessed by two or more distinct LPs*, except for certain "read-only" variables as noted below. In other words, LPs may *only* interact by exchanging time-stamped event messages. This is essential for the correctness of the Time Warp mechanism. Unfortunately, the kernel has no mechanism to prevent applications from defining and modifying variables that are shared between LPs, so application programs that define such variables will yield unpredictable results.

4.1 Types of Memory

Each LP must specify a state vector that contains all of the variables that persist from one event to the next. These variables should never be accessed by other LPs.

Variables that do *not* persist from one event to the next should be declared as local variables in the event handler procedure, or procedures called by the event handler, as needed. This will maximize performance because no checkpointing is required for these variables.

An LP's state vector may include three different types of state variables that are distinguished by the type of checkpointing that is performed on them:

1. state variables that are never modified after the initialization phase; these *read-only* variables need not be checkpointed.
2. state variables that are *automatically* checkpointed by the GTW kernel; the current implementation will automatically make a copy of all of these variables (*full-copy checkpointing*) prior to processing each event for the LP.
3. state variables that are only checkpointed when the application explicitly requests a variable be checkpointed, i.e., prior to the first modification of the variable within the event handler; the GTW kernel uses an *incremental checkpointing* mechanism to checkpoint these variables.

The cost of checkpointing may be large for some applications, and can easily cripple the performance of the GTW simulator. Thus, it is important to segregate read-only variables from those that are not modified during the simulation, and select an appropriate checkpointing mechanism for those variables that are modified by the event handler. The appropriate checkpointing mechanism to use depends on the size of the LP's state vector, the number of variables modified by each event, and the amount of computation within each event. If the state vector is small, i.e., the amount of time to make a full copy of the state vector is small relative to the amount of computation in an event, then full-copying (automatic checkpointing) is recommended. On the other hand, if the state vector is large, but only a few variables are modified by each event, then incremental checkpointing should be used.

A logical process may elect to have some variables automatically checkpointed, and others incrementally checkpointed. However, the designation for each state variable (automatic vs. incrementally checkpointed) cannot change during the simulation.

In addition to the state vectors for the logical processes, the simulator may define globally shared read-only variables that are accessible to all logical processes. These variable should not be modified after the initialization phase. Such modifications may have unpredictable results.

4.2 Memory Allocation

For each LP, the application program must

1. allocate storage for the state vector of the LP (the collection of variables that can be accessed by the LP) during the initialization phase, and
2. designate the state variables that are to be automatically checkpointed.

The GTW kernel does not need to be explicitly informed of the location of read-only variables, and only needs to know the location of incrementally checkpointed variables when the variable is actually checkpointed.

The application program is responsible for allocating storage for the state variables it requires. For this purpose, the `TWMalloc()` procedure is defined. `TWMalloc(sz)` allocates `sz` bytes of memory, and returns a pointer to this memory to the application. It is functionally identical to the standard `malloc()` procedure defined in the C library.

`TWMalloc()` should only be called during the initialization phase of the simulation. Applications calling `TWMalloc()` during the simulation itself should still execute correctly provided sufficient memory is available in the system. However, calling `TWMalloc()` in this way suffers from the following problems:

1. If an event invoking `TWMalloc()` is rolled back, the memory will be lost; at present, the kernel does not automatically reclaim it.
2. The current implementation of `TWMalloc()` calls `malloc()` to allocate memory. The `malloc()` procedure may yield very poor performance under certain circumstances, e.g., if many small blocks of memory are dynamically allocated.
3. No facility for reclaiming memory (e.g., the `free()` procedure) is provided. If the application calls `free()` directly, unpredictable results may occur because there is no facility to roll back such calls.

Variables created via ordinary declarations should be designated as `TW_SHARED` if the variable is visible between processes (e.g., global constants can be declared as such) or `TW_PRIVATE` if each processor is allocated a private copy. For instance,

```
TW_SHARED char c;
TW_PRIVATE int foo;
```

creates a single, globally shared character variable `c`, and `TWnpe` integer variables called `foo`, one for each processor. However, this feature may not be supported on all architectures.

4.3 Automatic Check-pointing

The GTW kernel provides a single procedure for specifying state variables that are to be automatically checkpointed by the kernel. `TWAutoCheck(addr,sz)` specifies a block of memory containing `sz` bytes, starting at memory location `addr` that is to be automatically checkpointed. In the current implementation, these variables are checkpointed prior to each invocation of the event handler for the LP. The `TWAutoCheck()` procedure may only be called at most *once* for each LP, and it must be called in the `IProc()` procedure for the LP. Note that `TWAutoCheck()` does not allocate any memory. It only informs the kernel of the memory locations that are to be automatically checkpointed prior to processing each event.

4.4 Incremental Check-pointing

Incremental checkpointing (also called incremental state saving) may be used in addition to, or rather than, automatic checkpointing. For incremental checkpointing, each event must explicitly indicate which state variables will be modified during the execution of the event. This could be done at the beginning of the event handler procedure, or just prior to when the modification actually takes place. The incremental checkpointing routines save the old contents of a single variable an a GTW *change-list* buffer. If the event is rolled back, the kernel will go through the change-list buffer in reverse order to restore the previous contents of the variables that were modified by each rolled back event. It is important that the variable be checkpointed *before* it is modified. A variable need only be incrementally checkpointed at most once during each execution of the event handler. Check-pointing the same variable several times within a single invocation of the event handler will not result in an error, but will degrade performance. The `IncrSave` field of `TWLP[]` must be set to `TRUE` in `TWInitAppl()` for incremental state saving to work. It is possible to selectively turn on incremental state saving for different LPs. Each LP has both incrementally and automatically checkpointed states.

The following procedures are defined for incrementally checkpointing variables:

1. `TWCheck(int *addr)`; Checkpoint variable of type `int` at address `addr`.
2. `TWCheckChar(char *addr)`; Checkpoint variable of type `char` at address `addr`.
3. `TWCheckDouble(double *addr)`; Checkpoint variable of type `double` at address `addr`.
4. `TWCheckTWTime(TWTime *addr)`; Checkpoint variable of type `TWTime` at address `addr`.
5. `TWCheckFloat(float *addr)`; Checkpoint variable of type `float` at address `addr`.
6. `TWCheckLong(long *addr)`; Checkpoint variable of type `long` at address `addr`.

4.5 Putting it All Together

A typical application program will include read-only variables, automatically checkpointed variables, and incrementally checkpointed variables. Such an application might include the following data structures for defining these state variables:

```
/* read only state variables for LP */
struct MyReadOnlyVars {
   int FirstReadOnlyVar;
   double SecondReadOnlyVar;
   }

/* automatically checkpointed variables for LP */
struct MyAutoVars {
   float FirstAutoVar;
   float SecondOne;
   }

/* incrementally checkpointed variables for LP */
struct MyIncVars {
   int FirstIncVar;
   char SecondIncVar;
   }

/* total state for LP using all of above */
struct MyLPState {
   struct MyReadOnlyVars ROVars;
   struct MyAutoVars CVars;
   struct MyIncVars ISVars;
   }
```

The `IProc()` procedure would include the following code to allocate storage for the variables, and specify which ones are automatically checkpointed:

```
/* allocate memory for state vector, and set pointer to LP state */
TWLP[TWMe()].State = TWMalloc(sizeof(struct(MyLPState)));

/* specify variables to be automatically checkpointed */
TWAutoCheck(&(TWLP[TWMe()].State->CVars), sizeof(struct(MyAutoVars)));
```

Finally, the `Proc()` procedure might call the following primitives to incrementally checkpoint other variables:

```
MyEventHandler(SV,M,MyPE)
struct MyLPState *SV;
struct MyMsgData *M;    /* message for event handler; discussed later*/
int MyPE;
{
   TWCheck(&(SV->ISVars.FirstIncVar));
   TWCheckChar(&(SV->ISVars.SecondIncVar));
}
```

5 Messages

To send a message, the logical process *must*

1. allocate a message buffer using the `TWGetMsg()` primitive,
2. store the data for the message into the message buffer using the kernel defined `TWMsg` variable, and
3. send the message using the `TWSend()` primitive.

If an event handler wishes to send several messages within a single event, it must perform each of the above steps repeatedly, once for each new message. Note that the message passed to the event handler cannot be "forwarded" to another processor. Rather, a new message must be created, the data copied to the new message buffer and then the *new* message is sent.

The associated primitives for sending messages are defined below:

- `TWGetMsg(TWTime TS, int LP, int sz)` allocates a message buffer where the data portion of the message (excluding timestamps, etc.) is `sz` bytes. `LP` indicates the logical process to which the message will be sent. The message timestamp is set `TS` units of simulated time into the future, i.e., with timestamp `TWNow() + TS`. As a side effect, this primitive also sets the private `TWMsg` variable to point to the application data portion of the message.
- `TWMsg` is a pointer variable within the GTW kernel. The application program should declare this variable as an `extern` of the type pointer to the application structure that specifies the contents of the message. For instance, the application program might declare the variable:

  ```
  extern TW_PRIVATE struct MyMsgData *TWMsg
  ```

  where `MyMsgData` is an application defined structure specifying the message contents. `TW_PRIVATE` indicates the variable is private to the processor (not globally shared). The application writes data into the message by writing into `TWMsg->MyData` where MyData is a field of the `MyMsgData` structure. *The application must never modify* `TWMsg` *itself, only the data buffer that it points to.* Modifying TWMsg so will produce unpredictable results.

  However since the sparc threads programming model does not support cheap thread private variables, TWMsg is implemented as an automatic variable that is read from the proper index into the global array `GState`.
- `TWSend ()` sends the message allocated by `TWGetMsg`. As a side-effect, `TWSend()` also resets the `TWMsg` variable to `NULL`, to discourage the LP from modifying the message after it has been sent.

It is important the logical process not modify the message after it has been sent, e.g., by making a copy of `TWMsg` prior to calling `TWSend()`. This may lead to unpredictable results. `TWSend()` transfers a pointer to the message to the destination processor; no copying of the data field is performed.

6 I/O Events

The computation for events invoked by the `TWSend` message primitive described above may be rolled back. I/O events address the rollback issue. Two types of I/O events are provided: *conservative* and *optimistic*. Conservative events never roll back, while optimistic events provides an anti-computation that is activated if the event is rolled back. I/O events differ from regular events in that the handler is specified as a parameter in the I/O send primitive. I/O events are not sent between different machines.

In order to compute irrevocable operations of an operations such as I/O while a simulation is running two conservative message primitives are provided: `TWBlockingIOSend` and `TWNonBlockingIOSend`. These primitives are *conservative* events because the processing of the event is delayed until it can be guaranteed that the event can never roll back. An event can be guaranteed to never roll back if its time stamp is equal to the smallest/earliest un-processed message in the entire simulation. This time is often referred to as the global virtual time (GVT).

A *blocking* event prevents the optimistic execution of the LP beyond the time stamp of the I/O event. The LP resumes optimistic execution when the blocking event is processed, the LP is rolled back or the blocking event is canceled. A blocking event may access or change the state of the LP. In contrast to the blocking event, the *nonblocking* event allows optimistic execution of the LP, and thus does not block the LP. A nonblocking event may not access or change the state of the LP, instead all data needed by the nonblocking event must be sent within the message of the event. The primitives for sending conservative I/O messages are summarized below:

- `TWNonBlockingIOSend(IOFuncPtr MyNonBlockingProc)` – sends message allocated by `TWGetMsg`[1] to schedule the user defined procedure: `MyNonBlockingProc`. As a side-effect, `TWNonBlockingIOSend` also resets the `TWMsg` variable to `NULL`, to discourage the modification of the message after it has been sent.
- `TWBlockingIOSend(IOFuncPtr MyBlockingProc)` – sends message allocated by `TWGetMsg`[1] to schedule the user defined procedure: `MyBlockingProc`. As a side-effect,
  `TWBlockingIOSend` also resets the `TWMsg` variable to `NULL`, to discourage the modification of the message after it has been sent.

Code fragment highlighting the conservative I/O primitives is shown in Figure 2.

Optimistic events are scheduled in the same manner as regular events. It is processed as soon as it becomes the smallest unprocessed messages in an LPs local queue. The primitive to schedule an optimistic event is: `TWOptimisticIOSend`. The handler that is evoked upon rollback of the event is specified as a parameter. This represents the anti-computation of the function. The primitive for sending optimistic I/O messages is specified below:

- `TWOptimisticIOSend(IOFuncPtr MyOptProc, IOFuncPtr MyRBProc)` – sends message allocated by `TWGetMsg`[1] to schedule the user defined procedure: `MyOptProc`. The user defined procedure `MyrBProc` is evoked if the event rollback. As a side-effect, `TWOptimisticIOSend` also resets the `TWMsg` variable to `NULL`, to discourage the modification of the message after it has been sent.

[1] `TWGetMsg` is described in section 5.

Code fragment highlighting the optimistic I/O primitive is shown in Figure 3.

7 Random Number Generation

The Time Warp kernel provides several functions for generating random numbers from different distributions. Any application using these primitives must allocate storage for the seeds used by the random number generator. Typically, this storage is allocated in the automatically checkpointed portion of the state vector of the LP. The seeds must be of type `TWSeed`, and should be declared in the LP's state vector as a checkpointed variable. `TWSeed` is a collection of two 32 bit variables, each of type `TWEachSeed_t`. If the seeds are checkpointed and the Time Warp program is otherwise deterministic, then the execution of the program using the random number generators will also be deterministic, and should always produce exactly the same results from one run to the next, independent of the number of processors that are used.

The GTW executive currently uses a random number generator using L'Ecuyer's double congruential algorithm [6] using two 32-bit integer seeds. A file containing over 20,000 different initial seeds is included in the GTW distribution to create multiple streams of random numbers. These seeds represent those produced by a single stream of random numbers sampled at intervals of 10 million calls to the generator.

A procedure called `TWRandInit (TWSeed *S, int skip)` is defined to initialize the random number generator seeds. If the second parameter is zero (or negative), each call will initialize the seeds pointed to by `S` to the next pair of values in the seed file. If skip holds a positive value k, then the next k pairs of seeds in the seed file will be skipped, and the seeds initialized to the $(k+1)th$ value.

Random number seeds *must* be initialized using `TWRandInit` or explicitly through calls to `TWRandSetSeeds` (defined below). The random number generators will produce an error if the seeds are both zero. It is recommended that the seeds be declared with initial value of zero so an error is produced if they are not properly initialized. For example, the seeds might be declared:

```
TWSeed MySeeds = 0, 0;
```

The other random number generator functions provided by the kernel are defined below. The first argument of each generator specifies the seed to be used. Each invocation of the random number generator modifies these seeds.

1. `TWRandSetSeeds (TWSeed *S, TWEachSeed_t s1, TWEachSeed_t s2)` Set the random number generator seeds `S` to `s1` and `s2`.
2. `TWRandGetSeeds (TWSeed *S, TWEachSeed_t *s1, TWEachSeed_t *s2)` Copy the random number generator seeds in `S` into `s1` and `s2`.
3. `double TWRandUnif(TWSeed *S)` Returns a random number between 0.0 and 1.0 that is selected from a uniform distribution.
4. `int TWRandGeometric (TWSeed *S, double p)` Returns a random number selected from a geometric distribution with parameter `p` (probability of success in each trial).
5. `double TWRandExponential (TWSeed *S, double mn)` Returns a random number selected from an exponential distribution with mean `mn`.

```
void MyNBProc (struct MyLPState *SV,  struct MyMsgData *M, int MyPE)
{
  /* no modification or access of state */
  printf ("Nonblocking Data: %d", M->count );
}

void MyBLProc (struct MyLPState *SV,  struct MyMsgData *M, int MyPE)
{
   struct LPState * CurState;
   CurState = GState[MyPE].CurState;

   /* update state, send more messages, etc, goes here */
}

void MyProc (struct MyLPState *SV,  struct MyMsgData *M, int MyPE)
{
   struct LPState * CurState;
   CurState = GState[MyPE].CurState;

   /* send a nonblocking message */
   ts = TWRandExponential (&(SV->CVars.Seeds), 1.0 );
   TWGetMsg ( ts, SV->ROVars.NeighborID, sizeof(struct MyMsgData) );
   TWMsg->count = M->count + 1;
   TWNonBlockingIOSend ( (IOFuncPtr) MyNBProc );

   /* send a blocking message */
   ts = TWRandExponential ( &(SV->CVars.Seeds), 1.0 );
   TWGetMsg ( ts, SV->ROVars.NeighborID, sizeof(struct MyMsgData) );
   TWMsg->count = M->count + 1;
   TWBlockingIOSend ( (IOFuncPtr) MyBLProc );
}
```

Figure 2: Conservative I/O event handling.

```
void MyOptProc (struct MyLPState *SV,  struct MyMsgData *M, int MyPE)
{
  /* no modification or access of state */
  printf ("Optimistic Data: %d", M->count );
}

void MyRBProc (struct MyLPState *SV,  struct MyMsgData *M, int MyPE)
{
  /* no modification or access of state */
  printf ("RB Optimistic Data: %d", M->count );
}

void MyProc (struct MyLPState *SV,  struct MyMsgData *M, int MyPE)
{
   struct LPState * CurState;
   CurState = GState[MyPE].CurState;

   /* send a nonblocking message */
   ts = TWRandExponential (&(SV->CVars.Seeds), 1.0 );
   TWGetMsg ( ts, SV->ROVars.NeighborID, sizeof(struct MyMsgData) );
   TWMsg->count = M->count + 1;

   TWOptimisticIOSend((IOFuncPtr) MyOptProc, (IOFuncPtr) MyRBProc );
}
```

Figure 3: Optimistic I/O event handling.

6. `long TWRandInteger(TWSeed *S, long low, long high)` Returns an integer random number between `low` and `high` inclusive that is selected from a uniform distribution.
7. `double TWRandNormal01(TWSeed *S)` Returns a Normal(0,1) random variable using Box-Muller method.
8. `double TWRandNormalSD(TWSeed *S, double mu, double sd)` Returns a random number selected from a normal distribution with mean `mu` and standard deviation `sd` using Box-Muller Method.
9. `long TWRandBinomial(TWSeed *S, long N, double P)` returns the number of trial until N successes occur, with probability P of success.
10. `double TWRandGamma(TWSeed *S, double shape, double scale)` Returns Gamma(shape, scale) random number.
11. `long TWRandPoisson(TWSeed *S, double Lambda)` Returns a Poisson random variable with mean Lambda.

8 An Example

We now describe a complete application using the primitives defined above. Specifically, we will complete the ping-pong example program discussed earlier. The `TWInitAppl()` procedure for this program is shown in Figure 1. The remainder of the program follows.

The first part of the program including the necessary `#include` files and data structures for the state vector are shown in Figure 4. The file `gtw.h` specifies various data structures and should be included in each application program file (using `#include`). The structure of the state vector is patterned after the example described in section 4.5, and includes a read only variable indicating the ID of the LP to which messages are sent, automatically checkpointed variables to hold the random number generator seeds and a count variable indicating the number of messages received by the LP, and an incrementally checkpointed array that records the timestamps of the last 1000 messages that were received. In this example, both the Ping and Pong LPs use the same data type to define their state vector, but in general, this need not always be the case.

In addition to the state vector, one additional data structure is defined to specify the format of messages. This is shown in figure 5. The message in this example is a single integer that counts the number of times the "ball" has bounced back and forth between the two LPs. More complex applications may utilize many different message formats. It is up to the application program to distinguish one format from another (e.g., using C variant records, i.e., unions) if this is the case.

Figure 7 shows the code for the initialization procedures for each logical process. Each procedure allocates storage for the state vector (using `TWMalloc()`) since `TWInitAppl()` did not already do so, and indicates the location of automatically checkpointed state variables to the kernel (using `TWAutoCheck()`). Note that this procedure must set the `State` field of the `TWLP` variable since `TWInitAppl()` did not already do so. Also, the procedure uses the `TWMe()` primitive to determine its own ID. The procedure then initializes the state variables defined by the LP. Finally, the Ping process sends an initial message to the Pong process using the `TWSend()` procedure. In any simulation, at least one of the LP initialization procedures must generate initial messages or

```
#include <gtw.h>

#define NLPs 2

/* random number generator seeds */
   TWSeed Seeds[NLPs];

/* read only variables in state vector */
struct MyReadOnlyVars {
   int NeighborID; /* ID of neighboring LP */
   };

/* automatically checkpointed variables for LP */
struct MyAutoVars {
   TWSeed Seeds;         /* seeds for random number generator */
   int Count;            /* number of messages received */
   };

/* incrementally checkpointed variables for LP */
struct MyIncVars {
   TWTime LastTS[1000]; /* remember last 1000 timestamps */
   };

struct MyLPState { /* total state for LP using all of above */
   struct MyReadOnlyVars ROVars;
   struct MyAutoVars CVars;
   struct MyIncVars ISVars;
   };
```

Figure 4: Header and state vector declarations for LP.

```
struct MyMsgData {              /* Message data */
   int count;
};
```

Figure 5: Data structure for messages.

else the simulator will immediately terminate. The Pong process does not send such a message.

The event handlers for the two LPs are shown in Figure 8 and 9. Each procedure records the timestamp of the message that was just received (available via the `TWNow()` macro) and increments the counter indicating the number of messages received. The former requires a call to the incremental state save routine *before* the variable is modified. A message is then returned to the other LP with an increased count field, and a timestamp increment selected from an exponential distribution with mean 1.0.

Finally, the wrap-up procedure for the Ping process is shown in Figure 10. In this example, no wrap-up procedure is defined for the Pong LP, although in general, any or all LPs may define wrap up procedures. The Ping wrap-up procedure used here prints the timestamps of the last 1000 messages received by the LP.

9 Compiling, Starting, and Stopping Execution

This section describes the runtime libraries that must be linked with the application program to produce an executable, the configuration file used to initialize the GTW system, command line arguments, and termination of the simulation.

9.1 Libraries

The application program needs to be linked with the kernel library code using the compiler (linker) flag `-lTW`. The Math library is also required, and is linked using the `-lm` flag. On the KSR one other library `-para` must also be linked. On the Sequent the libraries `-lseq` and `-lpps` should be linked. On the sparc, thread libraries should be linked with `-lthread`. A sample makefile is included with the GTW distribution.

9.2 The Configuration File

The kernel may be customized according to the simulation and the available hardware for better performance. The customization parameters include GVT frequency, amounts of event and state memory, and various debugging options. They are specified in a configuration file `.tw-config` that should be present in the directory in which the simulation is executed. The file consists of a list of parameter names followed by the values. Comments are preceded by a # sign. The parameter names are case independent. Some of the parameters are

GVT_Period
: Period of GVT calculation, in milliseconds. A smaller period will result in bad performance, whereas bigger periods cause a lot of aborted events due to non availability of message buffers. Values range from 10-1000, with 50 being typical.

PE_Event_Memory
: Bytes of memory per PE for events. Small values will cause TW to starve for memory, where as large amounts will cause the memory system to thrash. A value of 2 Megabytes is typical.

```
IPing(SV, MyPE)              /* Ping is LP 0 */
   struct MyLPState *SV;
   int MyPE;
{   struct MyMsgData *TWMsg;
   int i;
   TWTime ts;
   TWEachSeed_t s1, s2;

   struct LPState * CurState;
   CurState = GState[MyPE].CurState;


   /* SV is NULL since TWInit() did not allocate State */
   /* allocate memory for state vector, and set pointer to LP state */
   TWLP[TWMe()].State = TWMalloc(sizeof(struct MyLPState));
   SV = (struct MyLPState *) TWLP[TWMe()].State;

   /* specify variables to be automatically checkpointed */
   TWAutoCheck(&(SV->CVars), sizeof(struct MyAutoVars));

   /* initialize state variables */
   SV->ROVars.NeighborID = 1;   /* Pong is LP 1 */
   SV->CVars.Count = 0;
   for (i=0; i<1000; i++) SV->ISVars.LastTS[i] = 0.0;


   /* initialize random number generator seeds from seeds array */
   /* then initialize SV->CVars.Seeds */

   TWRandGetSeeds (&(Seeds[TWMe()]), &s1, &s2);
   TWRandSetSeeds (&(SV->CVars.Seeds), s1, s2);


   /* send initial message */
   ts = TWRandExponential(&(SV->CVars.Seeds), 1.0);
   TWGetMsg(ts, SV->ROVars.NeighborID, sizeof (struct MyMsgData));
   TWMsg->count = 0;
   TWSend();
}
```

Figure 6: Procedures to initialize each LP.

```
IPong(SV, MyPE)              /* Pong is LP 1 */
   struct MyLPState *SV;
   int MyPE;
{   struct MyMsgData *TWMsg;
   int i;
   TWEachSeed_t s1, s2;

   struct LPState * CurState;
   CurState = GState[MyPE].CurState;

   /* allocate memory for state vector, and set pointer to LP state */
   TWLP[TWMe()].State = TWMalloc(sizeof(struct MyLPState));
   SV = (struct MyLPState *) TWLP[TWMe()].State;

   /* specify variables to be automatically checkpointed */
   TWAutoCheck(&(SV->CVars), sizeof(struct MyAutoVars));

   /* initialize state variables */
   SV->ROVars.NeighborID = 0;   /* Ping is LP 0 */
   SV->CVars.Count = 0;
   for (i=0; i<1000; i++) SV->ISVars.LastTS[i] = 0.0;

   /* initialize random number generator seeds from seeds array */
   /* then initialize SV->CVars.Seeds */

   TWRandGetSeeds (&(Seeds[TWMe()]), &s1, &s2);
   TWRandSetSeeds (&(SV->CVars.Seeds), s1, s2);
}
```

Figure 7: Procedures to initialize each LP.

```
/* event handler for Ping */
Ping(SV, M, MyPE)
   struct MyLPState *SV;
   struct MyMsgData *M;
   int MyPE;
{   struct MyMsgData *TWMsg;

   TWTime ts;

   struct LPState * CurState;
   CurState = GState[MyPE].CurState;

   /* update state */
   TWCheckTWTime(&(SV->ISVars.LastTS[SV->CVars.Count % 1000]));
   SV->ISVars.LastTS[SV->CVars.Count % 1000] = TWNow();

   SV->CVars.Count++;

   /* bounce message back to Pong */
   ts = TWRandExponential(&(SV->CVars.Seeds), 1.0);
   TWGetMsg(ts, SV->ROVars.NeighborID, sizeof(struct MyMsgData));
   TWMsg->count = M->count + 1;
   TWSend();
}
```

Figure 8: Event handler procedures for ping-pong example.

```
/* event handler for Pong; actually, it is identical to Ping! */
Pong(SV, M, MyPE)
   struct MyLPState *SV;
   struct MyMsgData *M;
   int MyPE;
{  struct MyMsgData *TWMsg;
   TWTime ts;

   struct LPState * CurState;
   CurState = GState[MyPE].CurState;

   /* update state */
   TWCheckTWTime(&(SV->ISVars.LastTS[SV->CVars.Count % 1000]));
   SV->ISVars.LastTS[SV->CVars.Count % 1000] = TWNow();

   SV->CVars.Count++;
   ts = TWRandExponential(&(SV->CVars.Seeds), 1.0);
   /* bounce message back to Ping */
   TWGetMsg(ts, SV->ROVars.NeighborID, sizeof (struct MyMsgData));
   TWMsg->count = M->count + 1;
   TWSend();
}
```

Figure 9: Event handler procedures for ping-pong example.

```
FPing (SV, MyPE)
struct MyLPState *SV;
int MyPE;
{  int i;

   if (SV->CVars.Count < 1000) /* less than 1000 messages received */
      for (i=SV->CVars.Count; i>=0; i--)
         printf ("%f\n", SV->ISVars.LastTS[i]);
   else /* 1000 or more messages received */
      for (i=0; i<1000; i++)
         printf ("%f\n", SV->ISVars.LastTS[(SV->CVars.Count-i + 1000) % 1000]);
}
```

Figure 10: Wrap-up procedure for Ping LP.

PE_State_Memory
: Bytes per PE for state. A value of 2 megabytes is typical.

Rand_Mixer
: the random number generator "mixer;" different values of this parameter will cause different initial seeds for the random number generator to be used.

Debug
: If set to "message" a warning is printed whenever a message is sent or received. If set to "feedback", the .tw-config parameters are printed out. If set to "silent" only the essential statistics are printed out at the end. Other debugging flags were used for debugging different parts of GTW during development (e.g. GVT, Buffer management etc.)

  More than one value may be specified. The specified values are OR'ed to form a debugging bit vector.

Statistics

RetThreshold
: Determines when redistribution should happen. A higher value causes more frequent redistribution. It can be set from 1 to 100, a typical value is 4. It is the fraction of initial pool allocation at which redistribution is triggered.

Batch
: The number of events executed before the message queue is checked. This amortizes access to shared queues over several messages. Values depend on simulation, ranging from 2 for frequently communicating simulations (*e.g.* PHOLD) to 50 for PCS. If in doubt, set it to 1.

If the parameter file `.tw-config` is absent, a warning message is printed. Default values for the parameters are assumed, and a suboptimal performance may result in that case. There are other fields in the config file, *e.g.* Time Barrier, Block and Unblock. They relate to variations of TW that we have experimented with, and a user need not consider them.

9.3 Command Line Arguments

Command line arguments can be specified if the simulation run on a single machine. If the simulation is distributed parameters must be specified in the parameter file `.tw-config` described above. Usually two parameters are specified as arguments. They are respectively (i) an integer specifying the number of processors that will be used, and (ii) a floating point number indicating the length of the simulation in simulated time units. Thus if the executable is `sim`, the command `sim -p 8 -t 1000` executes the GTW simulation on 8 processors for 1000 virtual time units. Memory allocated can be specified via the flag `-m`. These arguments override values in `.tw-config`.

Application specific command line arguments may also be passed to the `TWInitAppl()` procedure using the `-A` flag. All arguments following this flag are passed to `TWInitAppl()` via the `argc` and `argv` parameters.

At the end of the simulation run, a set of performance related statistics is displayed. The statistics indicate the number of events processed by each individual LP, the number of primary and secondary roll-backs, total execution time, etc. The statistics should be self-explanatory.

9.4 Termination

The application specifies the length of the simulation run in simulated time units via a command line argument. The kernel terminates the simulation when there are no unprocessed message or anti-message in the system, or when simulation time advances up to the application specified end time, whichever occurs first. Operationally, `TWSend()` discards all messages with timestamp larger than the end time of the simulation. The *local virtual time (LVT)* of an LP is defined as *infinity* if the process has no unprocessed events. The simulation is terminated when *global virtual time (GVT)* advances past the end time of the simulation.

9.5 Output statistics

The kernel outputs various statistics. Here is how the various numbers add up:

Total number of Events processed = useful or net events processed + aborted events + number rolled-back + number cancelled

Events rolled-back = processed events rolled-back due to stragglers + processed events rolled-back due to annihilate

Number of roll-backs = roll-backs due to stragglers + roll-backs due to annihilation

Event rate = Net events/execution time

For a given run length, and model parameters, net events should be the same across runs on different number of processors i.e. the kernel is deterministic (modulo events scheduled on the same processor at the same time stamp).

The execution time reported is the time spent in the parallel portion of Time Warp. The initialization time is not counted.

10 The Sequential Simulator

A sequential simulator is provided that executes GTW programs on a single processor. GTW programs may be executed, without modification, on the sequential simulator. It is recommended that initial application software first be debugged on the sequential simulator before attempting execution on a parallel machine. In addition to the KSR and Sequent versions, a sequential simulator is also provided that runs on Sun Sparc and SGI workstations, enabling initial program development without using a parallel computer. For compatibility, the `.tw-config` configuration file must be provided for the sequential simulator. The command line arguments are as described in 9.3, except that the first argument is always 1, corresponding to a single processor.

The sequential simulator should be used for all speedup measurements rather than executing the Time Warp kernel on a single processor. The sequential simulator does not incur many of the overheads associated with Time Warp (e.g., state saving). The current implementation of the sequential simulator uses a calendar queue structure to implement the event list.

11 Unpacking and Compiling Notes

After untaring the distribution file, go to the various source and application directories and type make.

For deterministic execution, initial random number seeds are read from a seeds file, `gtw-seeds` included in the SEQ and GTW directories. Before compilation edit the `gtw-rand.h` file to point `TWSEEDFILE` to where ever the GTW and SEQ directories are.

References

[1] S. R. Das and R. M. Fujimoto. A performance study of the cancelback protocol for Time Warp. 7^{th} *Workshop on Parallel and Distributed Simulation*, 23(1):135–142, May 1993.

[2] R. M. Fujimoto. Time Warp on a shared memory multiprocessor. *Transactions of the Society for Computer Simulation*, 6(3):211–239, July 1989.

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

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

[5] D. R. Jefferson. Virtual time II: The cancelback protocol for storage management in distributed simulation. In *Proc. 9th Annual ACM Symposium on Principles of Distributed Computing*, pages 75–90, August 1990.

[6] P. L'Ecuyer. Efficient and portable combined random number generators. *Communications of the ACM*, 31(6):742–774, June 1988.

A Appendix: GTW Reference Manual

The following is a concise description of the the GTW interface intended for programmers that are familiar with the GTW software. In the following, all procedure, data structure, and variable names beginning with `My` (e.g., `MyIProc()`) are names defined by the application program. These names must not conflict with GTW primitive names, most of which begin with `TW`. All other names defined below are those defined by the GTW kernel and must be used "as is."

A.1 Application Program Data Structures

Application programs must define data structures for each LP's state and messages passed between LPs:

struct MyLPState
: A more or less arbitrary data structure defining the state vector of the LP. Different LPs may have different structures defining their state vectors. While it is not essential that the state vector be encapsulated into a C struct, it is usually convenient to do so. One common approach is to divide `MyLPState` further into C `struct`s for read-only, automatically checkpointed, and incrementally checkpointed variables.

struct MyMsgData
: A more or less arbitrary data structure defining the contents of messages that are sent. Different message formats may be used within a single application. Arbitrary data structures (even complex, dynamic data structures) may be passed in messages since message sends do not copy messages. Messages should not include pointers to modifiable portions of the LP's state vector.

A.2 Application Program Procedures

Application programs must define the following procedures:

TWInitAppl(int argc, char **argv)
: This procedure does a global initialization of the GTW program, and must set the `TWnlp` variable and the `TWLP` array. Memory may be allocated for the LP's state vectors here, or in the individual `MyIProc()` procedures. This procedure is the first application procedure called by the GTW kernel during the execution of a simulation. The `argc` parameter indicates the number of *application* command line parameters (specified after the `-A` flag) that were given, and `argv` is an array of character strings, one per application parameter.

MyIProc(struct MyLPState *SV, int MyPE)
: Function that is called to initialize an LP. Each LP must be assigned one such procedure (by setting the `TWLP[i].IProc`) variable. `SV` is a pointer to the state vector for the LP if one was designated in `TWInitAppl()` (by setting the `TWLP[i].State` variable). This procedure should allocate (using `TWMalloc()`) and initialize storage for the LP's state vector if this has not already been done, specify any variables that are to be automatically checkpointed (using `TWAutoCheck()`), and generate any initial messages that are used by the simulator (using `TWGetMsg`, `TWMsg`, and

TWSend). The IProc() procedure executes at simulated time 0. **Restriction:** The second parameter has to be named MyPE.

MyProc(struct MyLPState *SV, struct MyMsgData *M, int MyPE)

Event handler function that is called each time a message is received by the LP. Each LP must be assigned one such procedure (by setting the TWLP[i].Proc variable). SV is a pointer to the state vector for the LP. M is a pointer to the data portion of the message that was received.

Restriction: Because no copying of the message is performed by the GTW kernel, and message contents must be preserved in case the event is rolled back, it is recommended that application programs not modify the received message pointed to by M. If modifications are made, they must be incrementally checkpointed or unpredictable results may occur.

Restriction: The third parameter has to be named MyPE.

MyFProc(struct MyLPState *SV, int MyPE)

Optional function that is called at the end of the simulation for the LP, e.g., for generating output statistics. SV is a pointer to the final state vector for the LP. If used, a pointer to this function must be set in the TWLP[i].FProc variable for LP i during initialization.

Restriction: The second parameter has to be named MyPE.

MyBlockProc(struct MyLPState *SV, struct MyMsgData *M, int MyPE)

Optional function that is called in response to a blocking I/O message. MyBlockProc is not called until the Global Virtual Time (GVT) is later than or equal to the time stamp of message. A blocking event prevents the optimistic execution of the LP beyond the time stamp of the I/O event. The LP resumes optimistic execution after the blocking event is processed, the LP is rolled back or the blocking event is cancelled. A blocking event may access or change the state of the LP. SV is a pointer to the state vector for the LP. M is a pointer to the data portion of the message that was received.

Restriction: The third parameter has to be named MyPE.

MyNonBlockProc(struct MyLPState *SV, struct MyMsgData *M, int MyPE)

Optional function that is called in response to a nonblocking I/O message. MyNonBlockProc is not called until the global virtual time (GVT) is later than or equal to the time stamp of message. A nonblocking event allows optimistic execution of the LP, and thus does not block the LP. The nonblocking event may not access or change the state of the LP, instead all data needed by the nonblocking event must be sent within the message of the event. SV is a pointer to the state vector for the LP (cannot be accessed). M is a pointer to the data portion of the message that was received.

Restriction: The third parameter has to be named MyPE.

MyOptProc(struct MyLPState *SV, struct MyMsgData *M, int MyPE)

Optional function that is called in response to an optimistic I/O message. The optimistic event may not access or change the state of the LP, instead all data needed by the optimistic event must be sent within the message of the event. SV is a pointer to the state vector for the LP (cannot be accessed). M is a pointer

```
extern TW_SHARED int TWMemMap[MAXPROC][MAXPROC];

TWInitAppl(argc, argv)
int argc;
char **argv;
{

   for(i=0;i<TWnpe;i++)
     for(j=0;j<TWnpe;j++)
       if(i==j) TWMemMap[i][j] = 10;
       else TWMemMap[i][j] = 2;
}
```

Figure 11: Example use of TWMemMap

to the data portion of the message that was received.
Restriction: The third parameter has to be named `MyPE`.

MyRBProc(struct MyLPState *SV, struct MyMsgData *M, int MyPE)
Optional function that is called when the corresponding optimistic I/O message is rolled back (see `MyOptProc`). The roll-back event may not access or change the state of the LP, instead all data needed by the roll back event must be sent within the message of the event. `SV` is a pointer to the state vector for the LP (cannot be accessed). `M` is a pointer to the data portion of the message that was received.
Restriction: The third parameter has to be named `MyPE`.

A.3 The TW Memory Map

Each PE is allocated some amount of memory (the amount is specified via the `TW_Event_Memory` parameter in the `.tw-config` file). The buffer memory allocated for each processor, in turn, is partitioned into separate pools, one for each other processor in the system. Thus, there are N^2 pools, since each PE has N local pools. The default is to make each pool the same size. The way to override the default is to modify the `TW_MemMap`. `TW_MemMap[i,j]` gives a weight indicating how large to make the pool for messages sent from processor `i` to processor `j`. Thus, the sizes of the buffer pools for processor `i` are determined by the relative values in the i-th row, i.e., `TW_MemMap[i,*]`.

The absolute value of the weight doesn't matter, only its size relative to other weights in that row of `TW_MemMap` array. One way to assign the weights is to set `TW_MemMap[i][j]` to an estimate of the number of messages sent from processor `i` to processor `j` in the entire simulation. If no messages are sent, set the weight equal to zero.

The sizes of the buffer pools cannot be changed once execution has started.

Figure 11 contains an example of `TWMemMap`. In this particular application the ratio of messages to the same PE to messages off PE is 10:2. i.e. a PE sends ten messages to itself for every two that it sends to another PE. `TWMemMap[][]` is specified in `TWInitAppl()`.

A.4 Initializing the Simulation

The following variables are defined within the GTW kernel, and are used to initialize the simulator:

int TWnpe
: Indicates the number of processors used in the simulation. Processors are numbered $0, 1, 2, \ldots$`TWnpe` $-$ `1`. This variable must *not* be modified by the application program.

int TWnlp
: Indicates the number of logical processes used in the simulation. LPs are numbered $0, 1, 2, \ldots$`TWnlp` $-$ `1`. This variable must be set by the `TWInitAppl()` procedure.

int TWLP[i].Map
: Indicates the processor to which logical process `i` is mapped. The process will remain mapped to this processor throughout the entire simulation. The LP will never be executed by another processor, except possibly during the wrap-up phase of the simulation. This variable must be set by the `TWInitAppl()` procedure.

int *TWLP[i].IProc()
: Holds a pointer to the procedure that is called to initialize LP `i` (`MyIProc` defined in section A.2). This variable must be set by the `TWInitAppl()` procedure.

int *TWLP[i].Proc()
: Holds a pointer to the event handler procedure that is called each time LP `i` receives a message (`MyProc` defined in section A.2). This variable must be set by the `TWInitAppl()` procedure.

int *TWLP[i].FProc()
: Holds a pointer to the procedure that is called during the wrap-up phase of the simulation to perform any final computations and output statistics for LP `i` (`MyFProc` defined in section A.2). This procedure is optional. If used, this variable must be set by the `TWInitAppl()` procedure.

struct MyLPState *TWLP[i].State
: Holds a pointer to the state vector for LP `i`. This variable must be set by either the `TWInitAppl()` procedure or the `IProc()` procedure for LP `i`.

int TWMsgSize
: The size of message. At present all messages have a fixed size, specified at initialization.

A.5 Memory Allocation and Check-pointing

The following procedures are provided for allocating memory for the application program and for checkpointing variables.

(char *) TWMalloc (int sz)
: Dynamic memory allocator, functionally equivalent to `malloc()`. Allocate memory of `sz` bytes and return a pointer to the allocated memory to the caller. If no such memory is available, return `NULL`.

Caution: Dynamic memory allocation during the simulation phase should be avoided because no facility is provided for returning such memory back to the system; memory allocated by rolled back events will not be reclaimed by the system. But, if you must use dynamic memory allocation during the simulation, this memory must be incrementally checkpointed because it cannot be automatically checkpointed.

TWAutoCheck(char *addr, int sz)

Automatically checkpoint the block of memory beginning at memory address `addr` and extending `sz` bytes.

Restriction: At present, each LP may have at most *one* block of memory that is automatically checkpointed. `TWAutoCheck()` may only be called in the initialization procedure for an LP (`MyIProc()`).

TWCheck(int *addr)

Incrementally checkpoint the integer variable at memory location `addr`. All variables must me checkpointed *before* modification.

Restriction: Can only be called by an event handler procedure (`MyProc()`).

TWCheckLong(long *addr)

Incrementally checkpoint the double precision integer variable at memory location `addr`.

Restriction: Can only be called by an event handler procedure (`MyProc()`).

TWCheckChar(char *addr)

Incrementally checkpoint the character variable at memory location `addr`.

Restriction: Can only be called by an event handler procedure (`MyProc()`).

TWCheckDouble(double *addr)

Incrementally checkpoint the double precision floating point variable at memory location `addr`.

Restriction: Can only be called by an event handler procedure (`MyProc()`).

TWCheckFloat(float *addr)

Incrementally checkpoint the single precision floating point variable at memory location `addr`.

Restriction: Can only be called by an event handler procedure (`MyProc()`).

TWCheckTWTime(TWTime *addr)

Incrementally checkpoint the timestamp variable at memory location `addr`.

Restriction: Can only be called by an event handler procedure (`MyProc()`).

A.6 Messages

The following primitives are defined for creating and sending messages. They can be called in either the `IProc()` or the `Proc()` (i.e., the event handler) procedures of the LP.

struct MyMsgData *TWMsg

Variable defined within the GTW kernel that contains a pointer to the data portion of the last unsent message buffer allocated using `TWGetMsg`. Automatically reset to `NULL` by `TWSend()`. Must be declared as a `extern TW_PRIVATE` in the application program.

TWGetMsg(TWTime TS, int LP, int sz)

Allocate a message buffer with `sz` bytes for the data portion of the message. `TWMsg` is set to point to this buffer. `LP` indicates the ID of the logical process to which the message will be sent. The timestamp of the message is set to the LP's current simulated time (`TWNow()`) plus `TS`. Applications should not call `TWGetMsg()` again without first calling `TWSend()`, as this will cause the first message buffer to be permanently discarded.

Restriction: The current GTW software uses fixed sized message buffers, so applications requiring both very large and very small messages should partition the large messages into smaller pieces and transmit several messages, or pass a pointer to large messages. The latter approach should be used with caution, as messages should not be modified by the send after it is sent, or by the receiver.

TWSend ()

Send the message last allocated by `TWGetMsg()`.

Restriction: Because `TWSend` and the associated message receive procedures perform no message copying, application programs must not modify the contents of messages after they are sent. Doing so may lead to unpredictable results. Similarly, as discussed in `MyProc()`, event handlers must not modify received messages or unpredictable results may occur.

TWNonBlockingIOSend (IOFuncPtr MyNonBlockProc)

A nonblocking send of the message last allocated by `TWGetMsg()`. Same restriction as `TWSend`. The event handler `MyNonBlockProc` is invoked when the time stamp of the message is equal to or beyond GVT. `MyNonBlockProc` is a procedure of type void defined by the user.

TWBlockingIOSend (IOFuncPtr MyBlockProc)

A blocking send of the message last allocated by `TWGetMsg()`. Same restriction as `TWSend`. The event handler `MyBlockProc` is invoked when the time stamp of the message is equal to or beyond GVT. `MyBlockingProc` is a procedure of type void defined by the user.

TWOptimisticIOSend (IOFuncPtr MyOptProc, IOFuncPtr MyRBProc)

An optimistic send of the message last allocated by `TWGetMsg()`. Same restriction as `TWSend`. The event handler `MyOptProc` is invoked when the event is processed. The The event handler `MyRBProc` is invoked when the event is rolled back. `MyOptProc` and `MyRBProc` are procedures of type void defined by the user.

A.7 Random Number Generation

The following primitives are provided for generating random numbers.

TWSeed MySeed=0,0

Type for seeds used by the random number generator. Each LP using the generators should define seeds as checkpointed variables in the LP's state vector. Seeds should be set to zeros to detect failure to initialize them vis `TWRandInit` or `TWRandSetSeeds`.

TWRandInit (TWSeed *S, int skip)

Initialize random number generator seeds S, skipping over `skip` seeds in the seed file.

TWRandSetSeeds (TWSeed *S, long s1, long s2)
Initialize the random number generator seeds to s1 and s2.

TWRandGetSeeds (TWSeed *S, long *s1, long *s2)
Copy the random number generator seeds in `S` into `s1` and `s2`.

double TWRandUnif(TWSeed *S)
Returns a random number between 0.0 and 1.0 that is selected from a uniform distribution.

int TWRandGeometric (TWSeed *S, double p)
Returns a random number selected from a geometric distribution with parameter `p` (probability of success in each trial).

double TWRandExponential (TWSeed *S, double mn)
Returns a random number selected from an exponential distribution with mean `mn`.

long TWRandInteger(TWSeed *S, long low, long high)
Returns an integer random number between `low` and `high` inclusive that is selected from a uniform distribution.

double TWRandNormal01(TWSeed *S)
Returns a Normal(0,1) random variable using Box-Muller method.

double TWRandNormalSD(TWSeed *S, double mu, double sd)
Returns a random number selected from a normal distribution with mean `mu` and standard deviation `sd` using Box-Muller Method.

long TWRandBinomial(TWSeed *S, long N, double P)
returns the number of trial until N successes occur, with probability P of success.

double TWRandGamma(TWSeed *S, double shape, double scale)
Returns Gamma(shape, scale) random number.

long TWRandPoisson(TWSeed *S, double Lambda)
Returns a Poisson random variable with mean Lambda.

A.8 Other Procedures and Macros Defined by GTW

TWTime
Type for variables holding values of simulated time. Currently, this is defined as type `double`.

TWTime TWNow()
Returns the current simulated time of a process, i.e., the timestamp of the message it is processing.

int TWMe()
Returns the ID of the currently executing logical process.

TWTime TWEndTime()
Returns the simulated time at which the simulation terminates.