GTW^{++} – An Object-oriented Interface in C^{++} to the Georgia Tech Time Warp System

Kalyan S. Perumalla and Richard M. Fujimoto
College of Computing
Georgia Institute of Technology
Atlanta, GA 30332-0280

GIT-CC-96-09
September 17, 1996

Abstract

This document describes GTW^{++}, an efficient object-oriented interface to the Georgia Tech Time Warp (GTW) parallel simulation system for shared memory multiprocessors. The interface, which is in C^{++}, provides a clean and extensible set of abstractions for model developers wishing to use Time Warp as the parallel simulation paradigm. This interface delivers virtually the same performance as that of the C language interface to GTW. The object-oriented approach facilitates easily building higher-level interfaces, such as process-oriented views, over the basic GTW^{++} interface. GTW^{++} has been carefully designed so that almost identical interfaces can be supported for different parallel computing platforms, such as shared-memory machines and network of workstations, with appropriate underlying implementations for each platform. Furthermore, the GTW^{++} interface can be directly provided by the GTW kernel if and when the kernel itself is redesigned using an object-oriented approach.

Contents

GTW^{++} – An Object-oriented Interface in C^{++} to the Georgia Tech Time Warp System

Kalyan S. Perumalla Richard M. Fujimoto

September 17, 1996

1 Introduction

This document describes GTW^{++} – a C^{++} interface to the Georgia Tech Time Warp (GTW) parallel discrete event simulation system. The reader is assumed to have a basic knowledge of Time Warp and is interested in developing discrete event simulation applications in C^{++} using Time Warp as the underlying parallel simulation mechanism. For effective exploitation of the power provided by this interface, the user is expected to be conversant with the basic concepts in object-oriented (OO) methodology, such as *encapsulation*, *inheritance* and *polymorphism*. However, advanced knowledge of such techniques is not necessary for basic application development.

The C^{++} interface described here is an object-oriented (OO) version of the C language interface to GTW, with a one-to-one mapping of their respective functionalities. Thus, the basic "logical process"-level interface is provided in GTW^{++}. However, it is structured so that OO techniques can be appropriately applied to easily develop extensions and enhancements over the basic LP-level interface. This document assumes that the reader is not necessarily conversant with the C language interface to GTW. However, the interested reader is referred to [1] for a description of the same. The exposition of simulation application structure presented here is not dependent on that of [1]. Also, some material from [1] may be found repeated in this document.

The rest of the document is organized as follows. Section 1.1 provides a brief overview of the basic Time Warp simulation technique. Section 1.2 provides a brief overview of the functionality provided by the Georgia Tech Time Warp simulator. Section 2 presents an outline of the GTW^{++} interface. Section 3 describes the GTW^{++} interface and the associated semantics in detail. Section 4 presents a simple illustrative example application. Section 5 describes the compilation and execution process for GTW^{++} applications. Appendix A is a complete programmer's reference manual for GTW^{++}.

1.1 Overview of Time Warp Simulations

"Time Warp" is a mechanism for performing discrete event simulations in parallel using an optimistic-computation approach. Originally proposed by D. R. Jefferson in [3], this mechanism is based on the concept of *logical processes* (*LP*'s). The simulation is assumed to be modeled as a set of LPs, which interact with each other by exchanging *events* (see figure 1). The LPs are, in effect, units of computation, that can be executed

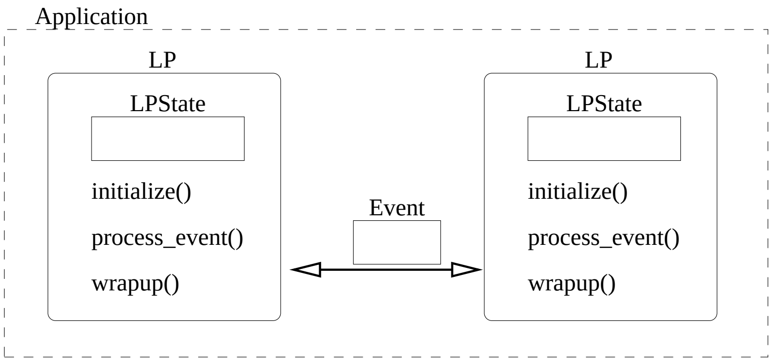

Figure 1: Fundamental objects in Time Warp

in parallel. All the LPs share the same simulated time line, and hence their computations are interdependent. The core of Time Warp lies in optimistically performing the LP computations while correcting any out-of-order computations that violate the order of the simulated time. Correction is achieved by undoing the effect of out-of-order computations and restarting the computations at the correct time. To undo computations, the concept of the *state* of an LP is used, which represents the set of all variables that could be modified by any computation in the LP. Techniques are used to appropriately save the state of the LP and use the saved states to undo incorrect computations by restoring the values of the state variables to their most recent, correct values.

Thus, the concepts of *LP*, *LP state* and *event* are fundamental in Time Warp simulations.

1.2 Overview of GTW

The *Georgia Tech Time Warp* (GTW) is an implementation of the basic Time Warp parallel simulation mechanism. GTW version 2.3 is the implementation for cache-coherent shared-memory multiprocessors. This version is currently available on several platforms, including SUN SparcStation and SGI PowerChallenge.

The C language interface supports the concepts of LP, LP state, and event (see Figure 1). An LP is identified by three C functions – one for initialization of the LP, one for processing the events destined for the LP, and the third routine for wrapping up the LP computation. The state of an LP is identified by a pointer to a C `struct` of appropriate type that contains the set of modifiable variables. A single global C function is used for initially setting up the data structures identifying all the LPs in the application. Routines are provided for the LPs to query the system status at runtime. In addition, the GTW release provides utilities for random number generation of various types, considering the fact that most simulation applications require the use of random numbers.

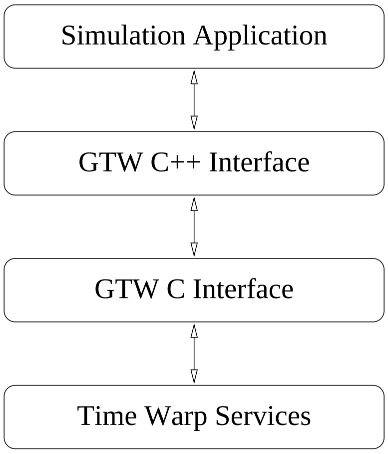

Figure 2: GTW^{++} Interface layers

2 Overview of GTW^{++}

GTW^{++} is an object-oriented interface to the Georgia Tech Time Warp System (see figure 2). The GTW^{++}interface is provided as a class library, with their associated definitions and implementations. The concepts of *LP*, *LP state* and *event* are directly supported as abstract C^{++} objects (classes). Applications are written as a set of objects that are defined as derived classes[1] of these abstract classes. Thus, the derived classes inherit all the functionality of the abstract base classes, while the base classes encapsulate the underlying implementation and provide a clean interface to the derived classes.

Currently, the internal implementation of the GTW^{++} classes makes use of the C interface layer of GTW. However, this may change in the future, so the applications using GTW^{++} should rely solely on the interface provided by the GTW^{++} classes and make no assumptions about their underlying implementations.

2.1 Fundamental Classes

The GTW^{++} interface consists of a set of abstract-class definitions that directly correspond to the basic objects in Time Warp simulations – namely, application, LP, LP state and event. The class definitions are designed to hide as much of the implementation details as possible, while providing a "clean service" interface to their derived classes. Application objects – LPs, states and events – are implemented by the user as specializations of these abstract classes.

A GTW^{++} application is defined as a derived class of `class CGTWApp`. Exactly one instance of this class will be instantiated per simulation run. This application class is responsible for the creation of the LPs. Each of the LPs must be defined as a derived

[1]The terms *derived class* and *base class* are used instead of *subclass* and *superclass* respectively, in line with the terminology of [4].

class of `class CLP`. These LPs exchange events, each of the events being a derived class of `class CEvent`. Each LP is free to make use of all the functionality available to any C^{++} `class`, with one exception – the set of all modifiable variables of that LP must be included in a class which is a derived class of `class CLPState`. Also, no two LPs should directly access any common variable unless it is for read-only purposes. *All* interaction between LPs must be performed using events.

2.2 Simulation Phases

Every GTW^{++} simulation consists of three distinct phases – the initialization phase, the simulation phase and the wrapup phase. In the initialization phase, the LPs are created and initialized. In the simulation phase, the LPs react to events and send events to other LPs. In the wrapup phase, summaries of the simulation are computed and the results are printed out. The bulk of the computation is performed as part of the simulation phase.

Every GTW^{++} application must define an *application object* that represents the specific application. It has to be defined as a derived class of the abstract GTW application object, `class CGTWApp`. Exactly one instance of this class will be created per simulation run. The `initialize()` method of this object is invoked as the very first method in the simulation. All the LPs of the simulation should be created and enrolled into the system as part of this application initialization[2]. At the end of this application initialization, the `initialize()` method of each of the LPs is invoked one after the other. At the end of the `initialize()` of the last LP, the initialization phase of the application is said to be completed, and the simulation phase is said to have started. In the simulation phase, the LPs react to the events that they receive. The `process_event()` method of an LP is invoked whenever an event arrives for it. Several types of events can be defined in the application, and the LPs can perform different computations based on the event type, as part of their `process_event()` method. LPs can send events to other LPs (or to themselves) as part of their `initialize()` or `process_event()` methods. The simulated time is at zero when the `initialize()` methods of the LPs are executed. No events can be sent as part of the `wrapup()` methods of the LPs.

Each LP is identified by a unique integer, called its LP ID. LP ID's range from 0 to N-1, where N is the total number of LPs in the system. The number of LPs can be more than the number of processors available to the system. Currently, LPs are mapped to processors in the initialization phase only, and the mapping is static. Future versions of GTW^{++} may support dynamic mapping and movement of LPs across processors.

3 The GTW^{++} Interface

The following are the abstract classes that constitute the basic GTW^{++} interface:

1. `class CGTWAPP` – object defining the specific application
2. `class CLP` – Logical Process object
3. `class CLPState` – object used to specify the state of an LP
4. `class CEvent` – object exchanged among the LPs

[2]Dynamic creation of LPs is not yet supported in the current version. Hence, LPs cannot be created after the end of the initialization phase.

These abstract classes are discussed in detail next.

3.1 CGTWApp – Application Object

This is the abstraction of any application that uses GTW^{++}. Every application should be defined as a derived class of CGTWApp. Part of the definition of CGTWApp is as follows:

```
class CGTWApp
{
    SUBCLASS_INTERFACE_REQD:
        virtual CBool initialize( int ac, char *av[] ) = 0;
        virtual CBool wrapup( void ) = 0;
};
```

Consider a "Pingpong" game simulation, where two players play against each other. Then the Pingpong application can be defined as a derived class of CGTWApp as follows:

```
class CPingpongGame : public CGTWApp
{
    CBool initialize( int ac, char *av[] );
    CBool wrapup( void );
};
```

In the CPingpongGame::initialize() method, the players should be created and enroled into the simulation (LPs are discussed in section 3.2). In the CPingpongGame ::wrapup() method, any statistics collected during the execution phase can be printed out.

Variables can be defined in this class that are used as read-only parameters by the LPs. Also, variables can be declared in this class so that the LPs can use them during their wrapup phase to compute summaries of the results.

Application Creator

In addition to defining the application class, exactly one function new_app() should be defined per application.

```
CGTWApp *new_app( void );
```

This function is invoked at runtime by GTW^{++} in order to identify the application object specific to the simulation. In the Pingpong example, the function can be defined as:

```
CGTWApp *new_app( void )
{
    return new CPingpongGame;
}
```

3.2 CLP – Logical Process Object

The class CLP is an abstraction of a Logical Process (LP). Every LP of the application should be defined as a derived class of this abstraction. This is a pure virtual class, and hence it cannot be instantiated as it stands. Only concrete derived classes of this class can be instantiated for use in the GTW^{++} system.

The definition of this class includes several services to its derived classes, along with abstraction of their interface to clients. The services provided by this object include:

- enrolling self into the GTW simulation; this is usually done immediately after the instantiation of the LP
- sending an event to another LP; this is usually performed during initialization of the LP and/or event processing by the LP
- signaling completion of the simulation at a given simulated time instant; this can be used by an LP when the actual length of simulation time is not known to the user in advance
- incrementally checkpointing some state variables; this is used by an LP that may have a large but sparingly-modified state
- convenience routines for querying simulation system variables at runtime.

The following abstract methods have to be defined by the derived classes:

- initialization before the start of the simulation
- processing an event received as part of the simulation
- final computation (wrapping up) at the end of the simulation
- specification of the *state* of the LP.

Part of the definition of CLP is as follows:

```
class CLP
{
    INTERFACE_TO_CLIENTS_OR_SUBCLASS:
        LP_ID enroll( PE_ID pe );

    INTERFACE_TO_SUBCLASS_ONLY:
        CBool send( CEvent *event );
        void shout_eureka( int flag );

    SUBCLASS_IMPLEMENTATION_REQD:
        virtual void initialize( CGTWApp *app ) = 0;
        virtual void process_event( const CEvent *event ) = 0;
        virtual void wrapup( CGTWApp *app ) = 0;
        virtual CLPState *get_state( void ) = 0;

    INTERFACE_TO_SUBCLASS_ONLY:
        // Other methods for incremental state saving
        // and query routines ...
};
```

In the Pingpong game example, each player can be defined as a derived class of the class CLP:

```
class CPlayer : public CLP
{
    ...
};
```

and the abstract methods can be defined. The send() method can be used by the player's initialize() and process_event() methods to send events (say, ball event) to the other player (events are described in section 3.4). The state of the player can be defined to hold information such as number of balls sent and number of balls received (states are described in section 3.3).

3.3 CLPState – State Object

The class CLPState is an abstraction of the state of an LP. Every LP should aggregate into a class the set of all variables that the LP may modify across any two event computations. This class should be defined as a derived class of CLPState. Part of the definition of CLPState is as follows:

```
class CLPState
{
    SUBCLASS_INTERFACE_REQD:
        virtual int size( void ) = 0;
};
```

Suppose we would like to keep track of the number of balls received and sent by the player LP in the Pingpong game simulation. Two variables can be used to count the same. Since these variables will be modified during the simulation, they should be included in the state of the LP. This is achieved as follows:

```
class CPlayerState : public CLPState
{
    int nsent;
    int nrecd;
    int size( void ) { return sizeof( CPlayerState ); }
};
```

And, the state can be used in the definition of the player LP as follows:

```
class CPlayer : public CLP
{
    ...
    CPlayerState state;
    CLPState *get_state( void ) { return &state; }
    ...
};
```

3.4 CEvent – Event Object

The class CEvent is an abstraction of any event that is exchanged among the LPs in the simulation. Several different types of events can be defined in a single application, and the LPs can exchange these events; LPs distinguish among the events using their *event types/tags*. Hence, globally unique event tags should be used to identify different types of events. Part of the definition of the class CEvent is as follows:

```
class CEvent
{
    INTERFACE_TO_SUBCLASS_ONLY:
        CEvent( EVENT_TYPE t );

    INTERFACE_TO_CLIENT_OR_SUBCLASS:
        void *operator new( size_t sz, LP_ID to,
            CSimTime ts, PE_ID pe );
};
```

In the Pingpong example, a Ball event can be defined using the following set of definitions:

```
const EVENT_TYPE BallEventTag = 1234; // Some globally unique#
class CBall : public CEvent
{
    CBall( void ) : CEvent( BallEventTag ) {}
    // Any event-specific variables can be defined here.
    // For example, ball_weight, or ball_size.
};
```

When a player in the Pingpong application needs to send a ball event to the other player, say, as a part of its initialize() method, the following can be used:

```
void CPlayer::initialize( CGTWApp *app )
{
    ...
    // Assume that the other player's LP ID is 1, and
    // the time for ball travel is, say, 2.0.
    CBall *ball = new( 1, 2.0, my_pe() ) CBall;
    send( ball );
    ...
}
```

Note that the new() operator of the event class has been overloaded to require as additional arguments the ID of the destination LP, and the time step into the future at which the event is being scheduled.

The process_event() method of an LP is invoked whenever an event arrives for that LP. The LP should perform the appropriate computations as part of the processing. It may itself send more events as part of the processing. If more than one type of event is used in the simulation, then the LP can distinguish among the events by using the type() method of CEvent to determine the type of the received event.

For correct operation of the Time Warp simulation, the LP is not allowed to modify the received event. For this reason, the `event` argument to the `process_event()` method is defined as a `const`, thus prohibiting its modification.

3.5 Random Number Generation

Several useful random number generator classes are provided along with the GTW^{++} class library. If a random number generator variable is used by an LP, then that variable has to be included as part of the state of the LP in order to obtain reproducible results across simulations. If the variable is not included in the LP state (and instead, defined simply as a class variable in that LP), then variations due to the Time Warp rollback mechanisms introduce variations in the generated random number sequences across different runs.

The set of generators supported includes Uniform, Geometric, Exponential, Normal, Binomial, Poisson, Gamma and Pareto distributions. For each distribution `X`, a C^{++} class named `CXRNG` is predefined. Each such class has two interface methods defined – `init()` and `next()`. The `init()` method can be used to initialize the generator with seed values and other generator-dependent parameters. After initializing the generator, the `next()` method can be successively called to generate successive sample values of the distribution.

4 Illustrative Example

A simple contrived application is presented here in order to illustrate the use of the various concepts in a complete application program. In this example, called "Pingpong", two players pass one ball back and forth. When a player receives a ball, he responds by passing back another ball to the sender. Let the player that serves the first ball be called "Ping," and let the other player be called "Pong." Each player keeps track of the number of balls received and the number of balls sent by that player. At the end of the simulation, each of the players prints out its statistics.

Each player is modeled as a logical process (LP). Except for the distinction of which player is the one that serves the first ball, both the players, Ping and Pong, are indeed identical, and hence, the common functionality among the two will be implemented as a class `CPlayer`, and then, two different classes, `CPing` and `CPong` will be defined as derived classes of `CPlayer` (see figure 3). Also, since the statistics about the number of balls has to be computed during the simulation, the state of each player will be defined to include two variables, `nsent` and `nrecd`, which will be used to keep track of the number of balls sent and received respectively.

First, the player, `class CPlayer`, is defined as a derived class of `class CLP` (see figure 6). Since each player should keep track of the number of balls received and sent by itself, the player's state is defined to hold two variables – `nsent` and `nrecd` (see figure 5). Note that the class variables `i_serve_first`, `travel_time` and `other_id` are *not* modified by the player during the simulation, and hence these variables should not be included as part of the player's state.

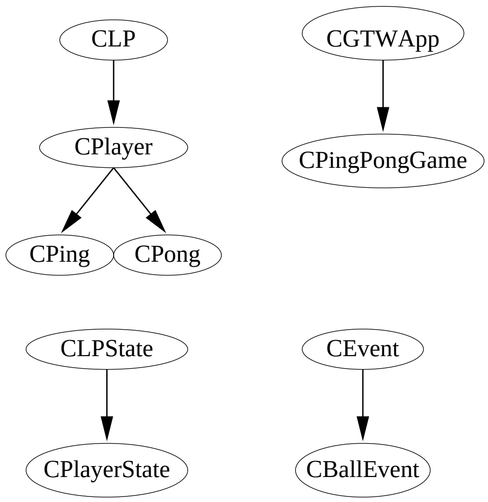

Figure 3: Class hierarchies for the Pingpong example application

```
const EVENT_TYPE BallEventTag = 1234; // Some globally unique#

class CBall : public CEvent
{
    INTERFACE_TO_CLIENT_OR_SUBCLASS:
        CBall( void ) : CEvent( BallEventTag ) {}

        // Also, any event-specific data
        // variables can be declared here.
        // For ex: ball_color, ball_weight, etc
};
```

Figure 4: Example — Ball event definition

```
class CPlayerState : public CLPState
{
    INTERFACE_TO_CLIENT_OR_SUBCLASS:
        int nsent;  //#balls sent so far
        int nrecd;  //#balls received so far

        CPlayerState( void ) { nsent = nrecd = 0; }

        int size( void ) { return sizeof( CPlayerState ); }
};
```

Figure 5: Example — Player state definition

```
class CPlayer : public CLP
{
    INTERFACE_TO_SUBCLASS_ONLY:
        CPlayer( CBool flag, CGTWApp *a ) : CLP( a )
        {
            i_serve_first = flag;
            travel_time = 2.0;  //arbitrary value used
            other_id = i_serve_first ? 1 : 0;
        }

    INTERNAL_IMPLEMENTATION:
        void initialize( CGTWApp *app );
        void process_event( const CEvent *event );
        void wrapup( CGTWApp *app );
        CLPState *get_state( void ) { return &state; }

    INTERNAL_IMPLEMENTATION:
        CPlayerState state;
        CBool i_serve_first;  //do I serve the first ball?
        CSimTime travel_time; //...of ball to reach other player
        LP_ID other_id;       //other player's ID
};
```

Figure 6: Example — Player definition

```
void CPlayer::initialize( CGTWApp *app )
{
    if( i_serve_first )
    {
        CEvent *ball =
            new( other_id, travel_time, my_pe() ) CBall;
        send( ball );
        state.nsent++;
    }
}

void CPlayer::process_event( const CEvent *event )
{
    state.nrecd++;
    CEvent *ball =
        new( other_id, travel_time, my_pe() ) CBall;
    send( ball );
    state.nsent++;
}

void CPlayer::wrapup( CGTWApp *app )
{
    cout << "Player #" << me() << ":";
    cout << " #sent = " << state.nsent;
    cout << " #recd = " << state.nrecd << endl;
}
```

Figure 7: Example — Player definition (continued)

```
class CPing : public CPlayer
{
    INTERFACE_TO_CLIENTS_ONLY:
        CPing( CPingpongGame *g ) : CPlayer( TRUE, g ) {}
};

class CPong : public CPlayer
{
    INTERFACE_TO_CLIENTS_ONLY:
        CPong( CPingpongGame *g ) : CPlayer( FALSE, g ) {}
};
```

Figure 8: Example — Definition of Ping and Pong

```
class CPingpongGame : public CGTWApp
{
    INTERFACE_TO_CLIENTS_ONLY:
        CBool initialize( int ac, char *av[] );
        CBool wrapup( void );
};

CBool CPingpongGame::initialize( int ac, char *av[] )
{
    int pe = 0;
    CPing *ping = new CPing( this );
    CPong *pong = new CPong( this );
    ping->enroll( pe );
    pong->enroll( pe );
    cout << "Pingpong game initialized." << endl;
    return TRUE;
}

CBool CPingpongGame::wrapup( void )
{
    cout << "Pingpong game done." << endl;
    return TRUE;
}
```

Figure 9: Example — Pingpong application definition

```
CGTWApp *new_app( void )
{
    return new CPingpongGame;
}
```

Figure 10: Example — Pingpong application creation

5 Compilation and Execution

For compiling applications, the sample `Makefile` for example applications provided as part of the GTW^{++} software should be used, and appropriately customized if necessary. The `Makefile` includes references to all the libraries and include directories required for compiling and building GTW^{++} applications.

The application requires two mandatory command line arguments, and other optional arguments. The first required argument should be a small integer that specifies the number of processors to be used for the simulation. Values greater than the total number of physically available processors are also legal. The next argument should be a floating point number, and it defines the length of the simulation in simulated time units. Application-specific command line arguments can be given by listing them after a `-A` flag after the mandatory arguments. These arguments will be passed to the initialization method of the application object of the simulation (see section 3.1).

For example, a command of the form

```
myapp 6 1e6 -A 25 Hello
```

specifies that the GTW^{++} simulation executable `myapp` should be executed on `6` processors and the arguments `25` and `Hello` be passed to the application object, and the application should be simulated until the virtual time reaches `1e6` units,

At the beginning and at the end of the simulation, the GTW^{++} runtime system outputs status and statistical information, such as the total number of LPs, memory allocations and event statistics.

A Reference Manual

The interface methods and the associated semantics of all the classes in GTW^{++} are presented in this section. For additional information on these classes, the header files `gtw++.h` and `gtw++util.h` can be perused.

A.1 Class CGTWApp

This class is an abstraction of all GTW^{++} applications. Every GTW^{++} application should be defined as a derived class of this abstract application object.

The following are the methods defined for this class:

♠ `CBool initialize( int ac, char *av[] )`
The application should perform the instantiation of its LPs, and their addition to the GTW system, as part of its `initialize()` method. This method is invoked by GTW^{++} runtime system as the first application method. The command-line arguments provided by the user are passed as parameters to this method.

♠ `CBool wrapup( void )`
The LPs of the application are assumed to summarize their computation results into this application object at the end of simulation (as part of their wrapup phase). The `wrapup()` method of this application object is invoked after its last LP has wrapped up.

A.2 Class CLP

This class is an abstraction of the Logical Process (LP) in a Time Warp simulation. Every LP of the application should be defined as a derived class of this abstraction. This is a pure virtual class; hence it cannot be instantiated as it is. Only concrete derived classes of this abstract LP class can be instantiated for use in the GTW^{++} system. Any derived class of this class can be made concrete by defining the pure virtual methods that are declared in this class.

LPs are often instantiated as part of the initialization method of the GTW application object (see section A.1).
The services provided by this object to its derived classes include the following:

♠ `virtual LP_ID enroll( PE_ID pe )`
Adds this LP to the GTW simulation. This method is usually invoked immediately after this object's instantiation. The LP is bound to the processor numbered `pe`, and the identifier assigned to this new LP is returned.

♠ `CBool send( CEvent *event )`
Sends the given event. Once the event is sent using this method, the GTW^{++} runtime system is assumed to have taken control over the event, and it should *not* be modified after this time. Deletion of the memory space occupied by the event is also automatically performed, and the user LPs should *not* perform any memory freeing operations on this event.

♠ `void shout_eureka( int flag = 0 )`
This method can be used by an LP to halt the GTW simulation at a

given simulated time instant (before the user-specified simulation end time). The calling LP may wish to use the `flag` argument to convey application-dependent information about the premature halt. This feature can be used in the application to stop the simulation at an instant in simulated time when, say, the required result has been found (as, for example, when the required number of ATM cells have been serviced by an ATM multiplexer model). It is usually the case that such a termination time instant is not known in advance to the user, and hence the user may supply an over-estimate of the required length of the simulation, and the LPs can signal the end of simulation at the appropriate instant during the simulation.

♠ `virtual void hear_eureka( int flag )`
This method is invoked on *every* LP if and when an LP "shouts eureka." Using this method, the LP can perform any actions that it may be necessary before halting. This method gets invoked before the wrapup phase begins. The value of `flag` broadcast by the "shouting" LP is passed to this method, and it can be used by the"hearing" LPs in an application-dependent way.

♠ `void incr_check( X *px )`
Incrementally checkpoints the variable pointed to by `px`. `X` can be any of the types `char`, `int`, `long`, `float` or `double`. Checkpointing of a variable must be done before the modification of the variable.

The following are convenience routines for querying system parameters:

♠ `LP_ID me( void )`
Returns the ID of this LP.

♠ `PE_ID my_pe( void )`
Returns the ordinal number of the processor to which this LP is bound.

♠ `CGTWApp *my_app( void )`
Returns the application to which this LP belongs.

♠ `CSimTime now( void )`
Returns the current value of the simulated time for this LP. Note that this may be different for different LPs even at the same physical time instant, due to the nature of computation using Time Warp.

♠ `int nLP( void )`
Returns the count of LPs currently enrolled into the simulation.

♠ `int nPE( void )`
Returns the number of available physical processors.

The following methods are left for the derived classes of this class to define:

♠ `virtual void initialize( CGTWApp *app )`
This is the LPs initialization before the start of the simulation (at simulated time 0).

♠ `virtual void process_event( const CEvent *event )`
This method is invoked by the system when there is an event destined to

this LP. The event is passed to this method, and the LP is assumed to "react" to this event in an application-dependent fashion.

Note: The LP is not supposed to modify the event in any way (not even deletion/destruction).

- ♠ `virtual void wrapup( CGTWApp *app )`
  This method is the final computation (wrapping up) of the LP at the end of simulation.

- ♠ `virtual CLPState *get_state( void )`
  This method specifies the state of the LP that has to be automatically checkpointed by the system.

A.3 Class CLPState

This is an abstraction of the automatically-checkpointable state of any LP. All states of LPs should be defined as derived classes of this abstract class.

- ♠ `virtual int size( void )`
  This virtual method has to be defined by the derived class and should return the total memory size (in number of bytes) of the derived class.

  Note: If pointer variables exist in the derived class, those pointers are *not* chased while saving the state, and so the contents that those pointers point to are *not* check-pointed. It is upto the programmer to use incremental check-pointing facilities to save such "indirect" state.

A.4 Class CEvent

This class is an abstraction of any GTW event exchanged by LPs. Every event in the application should be defined as a subclass of this event. Application-wide unique tags should be used to distinguish among various types of events used in the application.

The `new()` and `delete()` operators on this class are overloaded and redefined for achieving higher performance on shared-memory multiprocessors. This overloading also ensures that events cannot be allocated on the stack, i.e., the `new()` operator *must* be used in order to instantiate any event.

- ♠ `EVENT_TYPE type( void )`
  Returns the event tag associated with this event.

- ♠ `LP_ID to_lp( void )`
  Returns the ID of the destination LP of this event.

- ♠ `void *operator new( size_t sz, LP_ID to, CSimTime ts, PE_ID pe )`
  Overloaded instantiation operator for all event types. This is used by an LP (sender) wishing to send this event to another LP (destination). `to` is the identifier of the destination LP, `ts` is the amount of time from now into the future (also called the receive time) when the destination LP should receive the event, and `pe` is the ordinal number of the processor on which this (sender) LP resides.

- ♠ `void operator delete( void *, size_t )`
  Overloaded destruction operator for all event types. This has been predefined to perform the required actions, and it will be invoked by the

GTW^{++} runtime system. The application should not invoke this method at any time.

There is a limit on the total byte size of any event. This has been fixed at compile time, and remains constant at runtime. The limit size is large enough for most applications. If any event size exceeds this limit, a runtime error message is printed and the simulation is aborted.

A.5 Random Number Generators

All the supported Random Number Generators (RNG's) possess very similar interfaces. For each RNG the following methods are defined:

- A constructor with no arguments. This can be used if the values of the various parameters to this RNG are not known in advance, but will be assigned later.
- A constructor with arguments to initialize the seeds and RNG-dependent parameters. This can be used if all the values of the parameters to this RNG are known at the time of instantiating this RNG.
- An initialization method to (re)intialize the seeds and RNG-dependent parameters. The arguments to this method are exactly same as those for the constructor mentioned above. This method can be used to either initialize or reset the seeds and parameter values.
- A "next-number-of-sequence" generation method. This method takes no arguments. The return type varies with each RNG, mostly depending on whether the distribution is continuous or discrete.

In the following, the class `CRNGSeed` consists of a pair of positive integers; the constructor for `CRNGSeed` requires two integers as arguments for initializing its seed integer variables. `C0To1Interval` is a `typedef`, for readability, to represent the range 0 to 1 inclusive.

Class CUniformRNG

This class generates random numbers uniformly selected between 0.0 and 1.0.

♠ `void init( CRNGSeed s )`

♠ `C0To1Interval next( void )`

Class CParetoRNG

The following distribution is used in the implementation of Pareto distributed random number generator: if X is a *Pareto* distributed random variable, then,

$$f(X) = \begin{cases} \frac{\alpha\beta^{\alpha}}{(x+\beta)^{\alpha+1}} & \text{if } x \geq 0 \\ 0 & \text{otherwise} \end{cases}$$

with mean $\mu = \frac{\beta}{\alpha-1}$. This class generates random numbers from a pareto distribution with $\alpha =$ `alpha` and $\mu =$ `mean`. Note that `alpha` must be greater than unity.

♠ `void init( CRNGSeed s, double alpha, double mean )`

♠ `long next( void )`

Class CGeometricRNG

This class generates random numbers from a geometric distribution with probability of success in each trial = `p`.

♠ `void init( CRNGSeed s, C0To1Interval p )`

♠ `long next( void )`

Class CExponentialRNG

This class generates random numbers from an exponential distribution with mean = `mean`.

♠ `void init( CRNGSeed s, double mean )`

♠ `double next( void )`

Class CIntegerRNG

This class generates random numbers uniformly between `low` and `high` inclusive.

♠ `void init( CRNGSeed s, long low, long high )`

♠ `long next( void )`

Class CPoissonRNG

This class generates random numbers from the Poisson distribution with mean = `lambda`.

♠ `void init( CRNGSeed s, double lambda )`

♠ `long next( void )`

Class CBinomialRNG

This class generates random number X where X = the number of trials until `N` successes occur, with each trial succeeding with probability `P`.

♠ `void init( CRNGSeed s, long N, double P )`

♠ `long next( void )`

Class CGammaRNG

This class generates random numbers from a Gamma distribution of given `shape` and `scale`.

♠ `void init( CRNGSeed s, double shape, double scale )`

♠ `double next( void )`

Class CNormal01RNG

This class generates Normal(0,1) random values using Box-Muller method.

♠ `void init( CRNGSeed s )`

♠ `double next( void )`

Class CNormalSDRNG

This class generates random numbers selected from a normal distribution with mean `mu` and standard deviation `sd`, using Box-Muller method.

♠ `void init( CRNGSeed s, double mu, double sd )`

♠ `double next( void )`

References

[1] R. M. Fujimoto, *et al*, "Georgia Tech Time Warp Programmer's Manual," College of Computing, Georgia Institute of Technology, May 1994.

[2] S. Das, *et al*, "GTW: A Time Warp System for Shared Memory Multiprocessors," College of Computing, Georgia Institute of Technology.

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

[4] M. A. Ellis, B. Stroustrup, "The Annotated C++ Reference Manual," Addison-Wesley Publishing Company, 1992.