SimPlus

A Simulation Toolkit for C++

Scott Harper, Tom Mancine, Ryan Scott

Abstract

Simlib is a simulation library devised by the authors of our textbook for rudimentary simulation tasks. We completed several assignments with simlib over the course of the semester and became intimately acquainted with its structure, strengths, and weaknesses. For our project, we were assigned to produce an improved simulation library based upon simlib. At the minimum, we were to re-implement the code in C++ and improve event-list management and random number generation. In addition to these requirements, we have built into SimPlus a class hierarchy designed for flexibility and extensibility.

SimPlus At A Glance:

- Fully object-oriented design, implemented entirely in C++
- Improved random number generation: the Mersenne Twister for pseudorandom sequences, and a network-based true random number generator for limited use.
- Events now stored in a heap, permitting O(log n) insertions (and extractions), rather than the O(n) provided by simlib's linked-list implementation.
- Simple and extensible API allows users to develop their own classes for use in the simulation environment.
- Advanced memory management techniques such as event-pooling are used to optimize performance of the kernel.
- Callback-based kernel allows users to implement event handling in their own objects.
- Extensible statistics-gathering model provides more robust, flexible solution to reporting needs.
- Kernel implements Singleton design pattern for universality of reference.

Table of Contents

1. Introduction

1.1 Purpose

Simulation toolkits make a programmer's life easier; writing simulation software is difficult enough without having to reinvent the wheel at the beginning of each project. However, during the course of the semester, we tried several different simulation toolkits and found each to be inadequate. We tried NS and found it to be lacking in flexibility. We also tried simlib. Although simlib had the flexibility NS lacked, it was difficult to model large systems because there was no support for abstraction. In creating SimPlus, an enhanced version of the simlib toolkit, we set out to improve ease-of-use by providing support for abstraction. In addition, we found the linked-list implementation of the event list used by simlib burdensome, and the random number generation poor. We also attempted to remedy these shortcomings.

1.2 Goals and Objectives

- Improve random number generation
- Improve event-management data structures
- Convert toolkit to C++
- Impose object-orientation on toolkit to permit abstraction
- Increase degree to which event-processing is automated
- Produce meaningful documentation
- Improve programming experience for toolkit users

1.3 Background

In simlib, events were handled by the kernel and scheduled by use of the schedule() function; a call to the timing() function would return the type of the next event to occur. Statistics could be recorded by using the sampst function to observe values. All event processing was the user's responsibility and in general event handling was tightly coupled. Events were stored in a linked list, which caused performance to suffer greatly as the number of events in the system increased. Random number generation was poor as well; the RNG had a short period and there was no proper support for normally-distributed random variables.

2. Project Phases

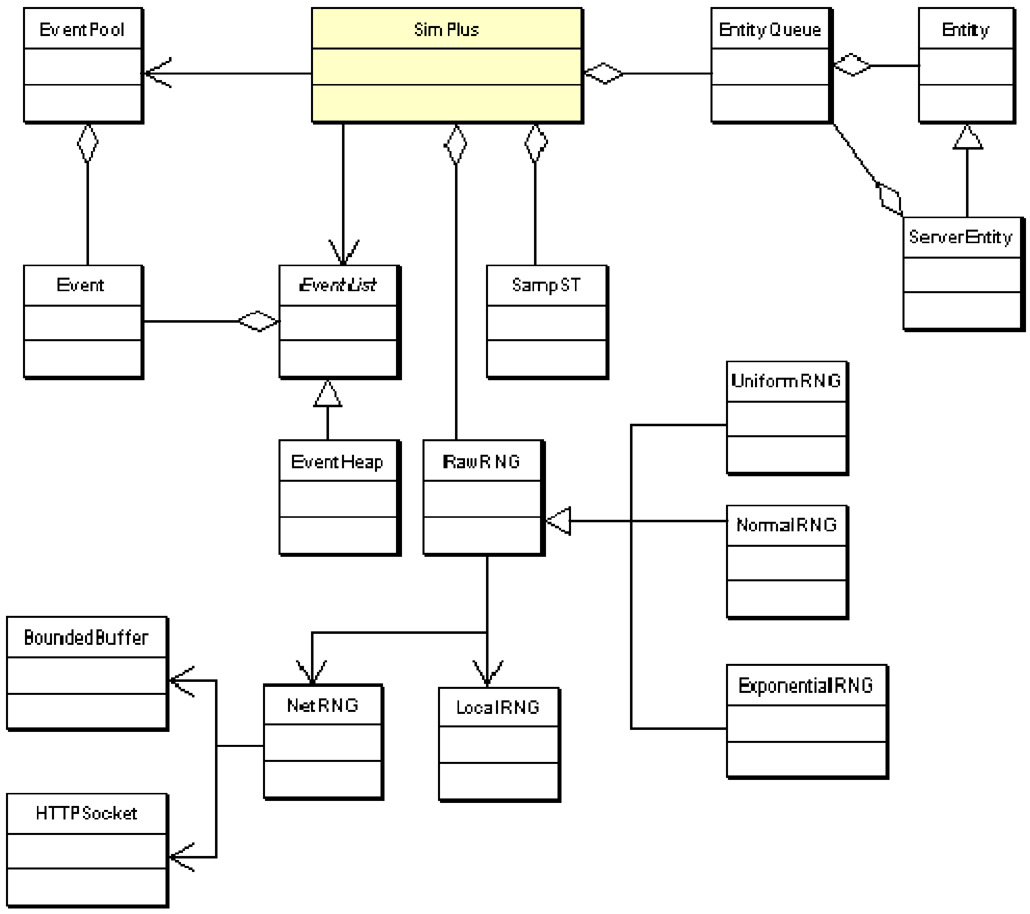
Like any large project, the development of SimPlus was divided into several phases. In the first phase, which lasted most of the semester, we learned what we wanted in a simulation tool by using several inadequate ones and noting their shortcomings. This would roughly correspond to the research phase of the project. We spent several days discussing our goals for the project and formulating a set of objectives; this was the requirements-gathering phase. We then spent about a week on design, laying out the class hierarchy and discussing the interaction between various objects. This was followed by two weeks of implementation and slight modification to the design as needed. The project was completed in four weeks.

3. Project Details

The key details of our project are the class diagram and the toolkit API, both of which follow. We also include the code of a demonstration application written using SimPlus.

3.1 Class Diagram

The following diagram illustrates the class hierarchy of the SimPlus toolkit. Documentation of the individual classes and their methods, etc. is found in section 3.2**, The SimPlus API**.



3.2 The SimPlus API

BoundedBuffer

A templatized, thread-safe circular buffer structure which implements the Monitor::Notify semantic. Its size is a construction-time parameter. When it is full the thread(s) calling put() is(are) blocked, and when the buffer empties the thread(s) calling get() is(are) blocked.

`BoundedBuffer(unsigned short)`
Constructor; size parameter determines size of circular buffer.

`~BoundedBuffer()`
Destructor

`get(T& ) : bool`
A monitor function which retrieves an element from the buffer. If the buffer is empty, the calling thread is blocked. It will return false without removing an element if the buffer has been torn down.

`put(T )  : bool`
A monitor function which puts an element in the buffer. If the buffer is full, the calling thread is blocked until slots are available. It will return false without adding an element if the buffer has been torn down.

`tearDown()`
A non-monitor function used to asynchronously tear down the bounded buffer. It instructs the bounded buffer to shut down. It does not deallocate system resources or memory.

`empty() : bool`
Non-monitor routine; returns true/false depending on state of queue.

Entity

Objects of type Entity represent any object that could possibly require control of the simulation. Objects that inherit from Entity are automatically assigned an unsigned int as a unique identifier. The methods generateEvent and processEvent are declared virtual to enable polymorphism. Any object that inherits from Entity should override these methods if that Object is capable of generating and/or processing events.

`Entity()`
Sole constructor. Sets the Entity's ID.

`~Entity()`
Destructor

`getID() : unsigned int`
Returns the ID of the Entity.

`virtual operator==(Entity&) : bool`
Compares for equality based on ID.

`virtual operator!=(Entity&) : bool`
Wrapped call to operator==. Tests for equality based on ID.

`virtual operator<(Entity&) : bool`
Tests whether the calling Entity's ID is less than the target Entity's ID.

`virtual generateEvent(const unsigned short&, const double&) : Event*`
Grabs an Event from the kernel's event pool, and sets its attibutes. First argument determines Event type. Second argument sets event time. By default, both owner ID and destination ID are set to the ID of the Entity calling this method. The event is automagically inserted into the system Event list. A pointer to the Event is returned to allow subclasses to modify the default behavior.

`virtual processEvent(Event*)`
Empty function to enable polymorphism. No default behavior.

`reportError( String )`
Used to panic the kernel when an Entity is called back with an event it does not know how to process.

EntityQueue

A class composed of a doubly linked list, representing queues capable of holding Entity objects. Implemented as a doubly-linked list. Care should be taken to avoid inserting a single Entity into multiple queues. Doing so will likely cause unexpected behavior.

`EntityQueue()`
Sole constructor. Sets size, front and back pointers to zero.

`~EntityQueue()`
Destructor.

`addFirst( Entity* )`
Adds the sole argument to the front of the queue.

`addLast( Entity* )`
Adds the sole argument to the end of the queue.

`removeFirst()`
Removes the last Entity from the queue and returns a pointer to it. Returns zero if the queue is empty.

`removeLast()`
Removes the last Entity from the queue and returns a pointer to it. Returns zero if the queue is empty.

`removeEntity( unsigned int )`
Removes from the queue the Entity with ID equal to the sole argument. Returns a pointer to the Entity if it is found, zero if not found. Operates in O(n).

`find( unsigned int )`
Finds within the queue the entity with ID equal to the sole argument. Returns a pointer to the Entity if it is found, zero if not found. Leaves the Entity in the queue. Operates in O(n).

Event

Event objects are used to control the flow of a system.

`Event()`
Default constructor. Initializes all data member to zero, and set the Event's ID.

`Event( double, unsigned int, unsigned int, unsigned short )`
Constructor. The first argument is used to set the timeStamp, the second sets the ownerID, the third the destinationID, and the fourth sets the eventType. The Event's ID is set automatically.

`~Event()`
Destructor.

`operator==( const Event& theEvent ) : bool`
Tests for equality based on Event ID.

`operator!=( const Event& theEvent ) : bool`
Wrapped call to operator==. Tests for inequality based on EventID.

`operator<( const Event& theEvent ) : bool`
Tests whether the calling event has a smaller timeStamp (i.e. is scheduled to occur earlier) than the other object.

`operator>( const Event& theEvent ) : bool`
Tests whether the calling event has a larger timeStamp (i.e. is scheduled to occur later) than the other object.

`reset()`
Reinitializes the data members of the Event to zero.

EventHeap

A heap data structure for storing SimPlus Events. Implemented as an in-place heap; permits insertion, retrieval, cancellation, and resizing. Acts like a priority queue ordered by Event::timeStamp. Implements the EventList interface.

`EventHeap(unsigned int)`
Constructor; parameter is initial size of heap. We recommend powers of two.

`~EventHeap`
Destructor

`get() : Event *`
Returns a pointer to the highest-priority (lowest-timestamped) event in the heap, while maintaining the heap property.

`put(Event *) : bool`
Puts the event passed as a parameter into the event heap; it will be inserted so as to maintain the heap property. Returns true/false depending on success; insertion will succeed unless there is no available space and the heap cannot be reallocated.

`resize(unsigned int) : bool`
Resizes the heap to the specified size. This is not necessary, as the heap is self-resizing, but can be useful if you expect a burst of events (for optimization). We recommend powers of two.

`cancelNext(unsigned short) : bool`
Cancels the next event of the type specified in the argument. In the case of the heap, the event is not actually removed because reheapification would take too long; the event is simply marked cancelled. Returns true if an event was found and cancelled; returns false if no events of that type were found.

EventList

An abstract base class defining the interface for the various event queues provided by SimPlus. Note that there is currently only one event queuing implementation; however, all other implementations are guaranteed to use this interface.

`EventList()`
Constructor; does basically nothing

`~EventList()`
Destructor; does basically nothing, but is virtual.

`get() : Event *`
Get an event from the event list. This is a pure virtual method and is not implemented in this class. Should return the next event to occur, i.e. the event with the lowest timestamp.

`put(Event *) : bool`
Put an event in the event list. A pure virtual method not implemented in EventList. Should return true if insertion was successful.

`resize(unsigned int) : bool`
Send a resize message to the data structure. A pure virtual method not implemented in EventList. Should return true if resize was successful.

`cancelNext(unsigned short) : bool`
Cancel the next event of the specified type. A pure virtual method not implemented in EventList. Should return true if an event was cancelled; otherwise returns false.

EventListException

An exception which can be thrown by objects implementing the EventList interface to indicate an unrecoverable error (such as during construction).

`EventListException(char *)`
Constructor; invoked by calling `throw(EventListException("my error message"))`.

`~EventListException()`
Destructor; does nothing.

`getMessage() : char *`
Returns a pointer to the error message put into the object at throw time.

EventPool

EventPool is used to speed up the SimPlus kernel by reducing context-switching overhead associated with use of the new operator. By pre-allocating a stack of Event objects, we save the user from using the new operator by allowing Events to be checked in and checked out from the pool when needed. The EventPool will somtimes need to use the new operator itself, but the Pool structure's pre-allocation minimizes that need.

`EventPool()`
Create an empty EventPool.

`EventPool( const unsigned int )`
Create an EventPool holding the specified number of events.

`~EventPool()`
Deletes all Events in the pool.

`release( Event* )`
Returns the first argument to the EventPool.

`get() : Event*`
Returns the top element of the EventPool.

`reserve( const unsigned int )`
Allocates a number of additional Events equal to the sole argument.

`getSize() : unsigned int`
Returns the number of Events currently in the pool.

ExponentialRNG

The ExponentialRNG class extends RawRNG. It will sample a random double from an exponential distribution with the specified mean.

`ExponentialRNG( double, unsigned short )`
Constructor; parameters are mean inter-arrival time for distribution and an unsigned short specifiying the RNG source.

`~ExponentialRNG()`
Destructor

`getRandom() : double`
Returns a random double sampled from the exponential PDF.

HTTPSocket

A C++ class that sends and receives STL strings via TCP/IP--designed as an HTTP client but will probably work for other TCP-based protocols. Uses a ton of C include files to provide networking functionality, and may require linking with -lxnet on the compile line.

`HTTPSocket(unsigned short)`
Constructor; argument is size of char[] buffer used to read from the socket--probably no effect, so it defaults to 1024 and tinkering is not recommended.

`~HTTPSocket()`
Destructor; deallocates buffer & closes socket if it's open.

`open(string, unsigned short) : bool`
Opens a connection to the specified server, at the specified port. The port defaults to 80 since this is HTTPSocket. Server may be a FQDN or an IP address.

`close() : bool`
Closes the open socket; returns false for convenience, not to indicate an error condition.

`operator!() : bool`
Returns false if the socket connection is open, true if closed.

`eof() : bool`
Returns true if socket is in EOF state. Provided for compatibility with iostream interface, but not currently in use.

`operator<<(string) : HTTPSocket&`
Overloaded stream insertion operator; used to send message across socket to server. Should be cascadable.

`operator>>(string) : HTTPSocket&`
Overloaded stream extraction operator; used to get a message from the server over TCP.

LocalRNG

A pseudorandom number generator based on the Mersenne Twister; basically a C++ port of the code by Nishimura and Matsumoto; the workhorse of the SimPlus RNG tree. The period of this RNG is 2^{19937}-1. Various random functions are provided, but the only one we use in SimPlus is genRandReal1().

`LocalRNG()`
Constructor.

`seedRand(unsigned long)`
Seed the random number generator with the specified seed.

`genRandReal1() : double`
Return a random double on the interval [0,1].

NetRNG

A class for random-number "generation" which fetches numbers from the random.org CGI application. It requires a working PThread implementation.

NetRNG uses a static member variable to track active instances. If a NetRNG being constructed is the first, a new thread of execution is created whose only job is to fill a bounded buffer with doubles on the interval [0,1] which are acquired by TCP connection with a random.org CGI app that generates true random numbers based on digitized radio-band noise. When the last active NetRNG has its destructor invoked, the static stop() method is called to tear down the Bounded Buffer and collect the "fetcher" thread.

`NetRNG()`
Constructor; nothing to do unless we're the first object being created, in which case we start up a thread to fetch random numbers into a buffer for us. Each object constructed increments the count of active RNGs.

`~NetRNG()`
Destructor; nothing to do unless we're the last active object, in which case we tear down the buffer and collect the fetcher thread. Each object destroyed decrements the count of active RNGs.

`operator!() : bool`
Checks to see if NetRNG statics are in a state where we can get numbers from it. Returns true if we are ready.

`genRandReal1() : double`
Return a double on the interval [0,1]; this call can block the calling thread if the buffer is empty.

NormalRNG

NormalRNG extends RawRNG. It receives two arguments at construction, mu and sigma (mean and standard deviation respectively). A polar method is used to generate two new random numbers (one is saved) that conform to a normal distribution. The getRandom() function will return a random double from an N(μ,σ) distribution.

`NormalRNG( double, double, unsigned short )`
Constructor; parameters are mean, standard deviation, and an unsigned short specifiying the RNG source.

`~NormalRNG()`
Destructor

`getRandom() : double`
Returns a random double sampled from the normal PDF.

RawRNG

A class for random number generation that aggregates the NetRNG and LocalRNG behaviors. The type of behavior is a construction-time parameter, so it's possible to have instances possessing both behaviors.

Two static constants are defined that let the user select behavior, such as the following:

```
RawRNG lRNG(RawRNG::Local);
RawRNG nRNG(RawRNG::Net);
```

Currently we default to the RawRNG::Local behavior.

`RawRNG(unsigned short)`
Constructor; argument sets the instance behavior to Net or Local RNG. Constructs the appropriate RNG for its behavior.

`~RawRNG()`
Destructor; deallocates our captive RNG object.

`genRandReal1() : double`
Fetches a double on the interval [0,1] from the appropriate RNG object.

`seedRand(unsigned long)`
Seeds the random number generator, if the random number generator is seedable.

SampST

SampST allows the user to specify handles to simple statistics-gathering mechanisms.

`SampST()`
Sole constructor. Initializes data members.

`~SampST()`
Destructor.

`observe(double)`
Adds its sole argument to the observed sum and increments the number of observations. Tracks the smallest and largest observations.

`getSum() : double`
Returns the current sum of all observations.

`getMinimum() : double`
Returns the current minimum of all observations.

`getMaximum() : double`
Returns the current maximum of all observations.

`getSampleSize() : unsigned int`
Returns the current number of observations.

`getMean() : double`
Calculates and returns the current mean of the observations.

ServerEntity

ServerEntity is a subclass of Entity designed to represent Entities that are capable of performing service on/for other Entities moving through the system.

`ServerEntity()`
Assigns this ServerEntity its numeric prefix and registers it with the kernel for Event callbacks.

`virtual ~ServerEntity()`
Virtual destructor.

`bind( EntityQueue*, string ) : bool`
Binds the specified EntityQueue to the current ServerEntity. Pointers to EntityQueues are stored in associative arrays (STL maps) with key equal to the second argument. A ServerEntity may be bound to more than one EntityQueue and an EntityQueue may be bound by more than one ServerEntity. Binding only implies that a ServerEntity may insert into/extract from an EntityQueue. Returns true if the insertion is successfule, false otherwise.

`unbind( string ) : bool`
Unbinds the queue with key equal to sole argument. Returns true if the queue is unbound, false if it is not bound to begin with.

`getBoundQueue( string ) : EntityQueue*`
Returns the queue associated with the sole argument.

`virtual generateEvent( const unsigned short&, const double& ) : Event*`
Wraps call to Entity::generateEvent. Adds this ServerEntity's prefix to the Event's eventType. Care should be taken to account for prefixes in subclass Event processing.

`getCheckDelay() : double`
Returns a normally distributed random number with mean 1.0 and standard deviation 0.1. The preferred method for performing service is to generate and process BEGIN_SERVICE events at offset equal to checkDelay->getRandom() until an Entity is available for service.

`getPrefix() : unsigned short`
Returns this ServerEntity's prefix.

SimPlus

Also known as "The Kernel," SimPlus is a C++ class that encapsulates most of the functionality required for small to medium scale simulation projects. SimPlus manages the creation and distribution of most simulation-related objects, such as events, RNGs, and several kinds of queues and keeps track of these objects so that it can clean up after itself.

SimPlus is loosely based on the simlib.c code. The kernel should be explicitly **`delete`**d; since no automatic objects of it may be created, the destructor is not implicitly invoked at program exit.

SimPlus is implemented using the "Singleton" design pattern. This means that within a given execution, only one instance of the Kernel object may be created. This is ensured by declaring the constructor, copy constructor, and overloaded assignment operator to be protected. References to the current kernel are obtained by SimPlus::getInstance (see below).

`static getInstance() : SimPlus*`
Returns a pointer to the current instance of SimPlus. Creates a new instance and returns a pointer to it if this is the first call.

`~SimPlus()`
Explcitly deletes all of the object references handed out by calls to the various getXXX methods.

`static reportError( string )`
Panics the kernel, prints the error message contained in the sole argument and exits with an error code of 1. To be used in the case of non-recoverable errors only.

`timing() : Event*`
Processes the next event contained in the event list. If the Entity where the event is scheduled to take place has registered itself with the system (default behavior for objects of type ServerEntity), timing calls the processEvent method for that Entity and returns zero. If the target Entity is not registered with the kernel, a pointer to the Event object is returned so that the Event may be processed manually.

`registerServer( ServerEntity* )`
Registers the sole argument with the kernel for event callbacks.

`scheduleEvent( Event* )`
Inserts the sole argument into the system's event list. Calls reportError if the event list cannot or will not accept more Events.

`cancelEventType( const unsigned short& ) : bool`
Cancels the next event with type equal to its sole argument. Operates in O(n) time. Returns true if the event is cancelled, false otherwise.

`cancelEventID( const unsigned int& ) : bool`
Cancels a specific event with eventID equals to its sole argument. Operates in O(n) time. Returns true if the event is cancelled, false otherwise.

`getEvent() : Event*`
To save time once a simulation has begun executing, an initial pool of Event objects is allocated. The getEvent method returns a handle to the top Event in the pool if the pool has any events in it, and returns a new Event if the pool is empty.

`releaseEvent( Event* )`
Calls the reset method of its sole parameter and then inserts it into the Event pool for reuse.

`expandEventPool( const unsigned short& )`
Causes the EventPool to allocate a number of new events equal to its sole argument (usually in anticipation of a large number of new events.)

`availableEvents() : unsigned int`
Returns the number of Event objects currently accounted for in the EventPool.

`getEntityQueue() : EntityQueue*`
Allocates a new EntityQueue for the user, saves a reference to it for cleanup, and returns a pointer to it.

`getSampST() : SampST*`
Allocates a new SampST for the user, saves a reference to it for cleanup, and returns a pointer to it.

`getExponentialRNG( const double&, unsigned short ) : ExponentialRNG*`
Allocates a new ExponentialRNG for the user, saves a reference to it for cleanup, and returns a pointer to it. The second argument indicates the type of RawRNG to be used in the ExponentialRNG being created. The default is RawRNG::Local, which is seeded. Another possibility is RawRNG::Net, which retrieves truly random numbers via an HTTPSocket from random.org. The first argument specifies the mean of the numbers generated.

`getNormalRNG( const double&, const double&, unsigned short ) : NormalRNG*`
Allocates a new NormalRNG for the user, saves a reference to it for cleanup, and returns a pointer to it. The last argument indicates the type of RawRNG to be used in the NormalRNG being created. The default is RawRNG::Local, which is seeded. Another possibility is RawRNG::Net, which retrieves truly random numbers via an HTTPSocket from random.org. The first argument specifies the mean of the numbers generated. The second specifies the standard deviation.

`getUniformRNG( const double&, unsigned short ) : UniformRNG*`
Allocates a new UniformRNG for the user, saves a reference to it for cleanup, and returns a pointer to it. The second argument indicates the type of RawRNG to be used in the UniformRNG being created. The default is RawRNG::Local, which is seeded. Another possibility is RawRNG::Net, which retrieves truly random numbers via an HTTPSocket from random.org. The first argument specifies the lower bound of the numbers generated. The second specifies the upper bound.

`getSimTime() : double`
Returns the current simulation time.

UniformRNG

The UniformRNG class inherits from RawRNG. Two parameters are received at construction, the upper and lower bound of the new uniformly distributed random number. The getRandom() function will return one random double.

`UniformRNG( double, double, unsigned short )`
Constructor; parameters are lower and upper bounds and an unsigned short specifiying the RNG source.

`~UniformRNG()`
Destructor

`getRandom() : double`
Returns a random double sampled from the uniform range.

3.3 Demonstration Application (Source)

A few pieces of source code from the demonstration application are presented below; hopefully, they will help illuminate the use of the API for simulation tasks. The code samples provided are from a simulation of the infamous "healthcare-clinic" model, involving an administrative station, two nurses, and two doctors.

Doctor Class (Header)

```
/****************************************************************************

Scott Harper, Tom Mancine, Ryan Scott

Doctor.h

The Doctor node processes incoming Patients and either removes them from the
system or sends them to another Doctor's queue (5% probability).

METHODS:
--------

Doctor( double, double, EntityQueue*, EntityQueue*)
Sole constructor.  First argument is mean service time.  Second argument is
standard deviation of service time.  Third argument is the queue from which
the Doctor will pull.  Fourth argument is the queue of the other Doc
to which the Doctor can send Patients.  Generates a BEGIN_SERVICE event
to force the node to begin checking for Patients.

~Doctor()
Destructor.

virtual generateEvent( const unsigned short&, const double& ) : Event*
Calls generateEvent in the ServerEntity class.

virtual processEvent( Event* )
Handles event callbacks from the kernel.  For a BEGIN_SERVICE event, the
Doctor grabs the first Patient in its queue.  If the Patient grabbed is
null (zero) a BEGIN_SERVICE event is generated at a random time offset
(a call to getCheckDelay in ServerEntity.)  If the patient is not null, the
Doctor goes to work (schedules an END_SERVICE event at time offset equal
to the next random available from the mean interarrival time.)
For an END_SERVICE event, the Patient is either removed from the system or is
added to the end of the other Doctor's queue (with 5% probability).

friend operator<<( ostream&, Doctor& ) : ostream&
Allows the Doctor to output its statistics to any ostream, such as
STDOUT or a file.  Returns a reference to the ostream.

****************************************************************************/

#include <iostream>
using std::iostream;
using std::endl;

#include <string>
```

```
using std::string;

#include "SimPlus.h"
#include "Patient.h"

#ifndef DOCTOR_H
#define DOCTOR_H

class Doctor : public ServerEntity
{
	public:
		Doctor( double, double, EntityQueue*, EntityQueue* );
		~Doctor();

		virtual Event* generateEvent( const unsigned short&, const double& );
		virtual void processEvent( Event* );
		static unsigned short patientsProcessed() { return numPatients; };

		friend ostream& operator<<( ostream&, Doctor& );

	protected:
		static unsigned short numPatients;
		Patient* currentPatient;
		SampST* idleTimeStat;
		SampST* waitTimeStat;
		SampST* serviceTimeStat;
		NormalRNG* myServiceTime;
		UniformRNG* myRedirectProb;
		double lastStart;
		double lastStop;

	private:
};

#endif
```

Doctor Class (Implementation)

```
/*****************************************************************************

Scott Harper, Tom Mancine, Ryan Scott

Doctor.cpp

The documentation within this file is sparse, and is only intended to provide
an overview of coding practices.  For a more detailed description of Doctor,
see Doctor.h.

*****************************************************************************/

#include "Doctor.h"

unsigned short Doctor::numPatients = 0;

Doctor::Doctor( double mu, double sigma, EntityQueue* myQ,
	EntityQueue* otherQ ) : ServerEntity()
{
	SimPlus* handle = SimPlus::getInstance();
	idleTimeStat = handle->getSampST();
	waitTimeStat = handle->getSampST();
	serviceTimeStat = handle->getSampST();

	// this doctor's service time is a normal distribution with a mean of
```

```
        // fifteen minutes and a standard deviation of five minutes
        myServiceTime = handle->getNormalRNG( mu, sigma );

        // for redirection we will use a uniform between zero and one
        myRedirectProb = handle->getUniformRNG( 0.0, 1.0 );

        lastStart = 0;
        lastStop = 0;

        bind( myQ, "MY_QUEUE" );
        bind( otherQ, "OTHER_DOC_QUEUE" );

        generateEvent( BEGIN_SERVICE, getCheckDelay() );
}

Doctor::~Doctor()
{
}

Event* Doctor::generateEvent( const unsigned short& eventType,
        const double& eventTime)
{
        if( eventType != BEGIN_SERVICE && eventType != END_SERVICE )
                return 0;
        return ServerEntity::generateEvent( eventType, eventTime );
}

void Doctor::processEvent( Event* anEvent )
{
        double simTime = SimPlus::getInstance()->getSimTime();

        if( anEvent->getEventType() == (getPrefix() + BEGIN_SERVICE) )
        {
                // see if there are any patients waiting
                currentPatient = (Patient*)getBoundQueue("MY_QUEUE")->removeFirst();

                // if no patients are waiting, check again in 1 minute
                if( currentPatient == 0 )
                        generateEvent( BEGIN_SERVICE, simTime + getCheckDelay() );
                // otherwise go to work
                else
                {
                        generateEvent( END_SERVICE, simTime +
                                myServiceTime->getRandom() );
                        lastStart = simTime;
                        idleTimeStat->observe( simTime - lastStop );
                        waitTimeStat->observe( currentPatient->endWait( simTime ) );
                }

        }
        else if( anEvent->getEventType() == (getPrefix() + END_SERVICE) )
        {
                // we redirect 5% of the time
                if( myRedirectProb->getRandom() < 0.05 )
                {
                        getBoundQueue( "OTHER_DOC_QUEUE" )->addLast( currentPatient );
                        currentPatient->beginWait( simTime );
                }
                // otherwise we kick the patient out of the system
                else
                {
                        ++numPatients;
                        currentPatient->exitSystem( simTime );
                        //delete currentPatient;
                }

                // send my patient to the corresponding doc
                currentPatient = 0;

                lastStop = simTime;
                serviceTimeStat->observe( simTime - lastStart );
                //check for patients again in 1 minute
                generateEvent( BEGIN_SERVICE, simTime + getCheckDelay() );
        }
        else
        {
                reportError( "Doctor" );
        }
        SimPlus::getInstance()->releaseEvent( anEvent );
}

ostream& operator<<( ostream& out, Doctor& theDoctor )
{
        out << "Doctor: Average idle time between patients for this doctor: "
            << theDoctor.idleTimeStat->getMean() << endl;
        out << "Doctor: Average service time for patients for this doctor:  "
            << theDoctor.serviceTimeStat->getMean() << endl;
        out << "Doctor: Average patient wait time in queue for this doctor: "
            << theDoctor.waitTimeStat->getMean() << endl << endl;
        return out;
}
```

Nurse Class (Header)

```
/*****************************************************************************

Nurse.h

The Nurse node processes incoming Patients and directs them to one of two
Doctor queues.  Patients are directed to the primary queue with 85%
probability.

METHODS:
--------

Nurse( double, double, EntityQueue*, EntityQueue*, EntityQueue*)
Sole constructor.  First argument is mean service time.  Second argument is
standard deviation of service time.  Third argument is the queue from which
the Nurse will pull.  Fourth and fifth arguments are the two queues
to which the Nurse can send Patients.  Generates a BEGIN_SERVICE event
to force the node to begin checking for Patients.

~Nurse()
Destructor.

virtual generateEvent( const unsigned short&, const double& ) : Event*
Calls generateEvent in the ServerEntity class.

virtual processEvent( Event* )
Handles event callbacks from the kernel.  For a BEGIN_SERVICE event, the
Nurse grabs the first Patient in its queue.  If the Patient grabbed is
null (zero) a BEGIN_SERVICE event is generated at a random time offset
(a call to getCheckDelay in ServerEntity.)  If the patient is not null, the
Nurse goes to work (schedules an END_SERVICE event at time offset equal
to the next random available from the mean interarrival time.)
For an END_SERVICE event, the Patient is added to the end of one of the two
queues with which the Nurse is associated.

friend operator<<( ostream&, Nurse& ) : ostream&
Allows the Nurse to output its statistics to any ostream, such as STDOUT or a
file.  Returns a reference to the ostream.

*****************************************************************************/
```

```
#include <iostream>
using std::ostream;
using std::endl;

#include "SimPlus.h"
#include "Patient.h"

#ifndef NURSE_H
#define NURSE_H

#include <string>
using std::string;


class Nurse : public ServerEntity
{
       public:
              Nurse( double, double, EntityQueue*, EntityQueue*, EntityQueue* );
              ~Nurse();

              virtual Event* generateEvent( const unsigned short&, const double& );
              virtual void processEvent( Event* );

              friend ostream& operator<<( ostream&, Nurse& );

       protected:
              Patient* currentPatient;
              SampST* waitTimeStat;
              SampST* idleTimeStat;
              SampST* serviceTimeStat;
              NormalRNG* myServiceTime;
              UniformRNG* myRedirectProb;
              double lastStart;
              double lastStop;

       private:
};

#endif
```

Driver Program

```
/****************************************************************************

office.cpp

The driver program for the SimPlus simulation library best practices app.

METHODS:
--------

main() : int
Creates a queueing network representing a healthcare clinic.  A subclass of
ServerEntity, called EntryNode, generates traffic composed of Patient Entities,
arriving according to an Exponential mean interarrival time.  From the
EntryNode, Patients are directed to an Administrator who handles preliminary
paperwork.  The Administrator then distributes the Patients randomly between
two Nurses on approximately a 50/50 basis.  The Nurses then perform
preliminary checks on the Patient.  Each Nurse is associated with a Doctor
and by default sends Patients to that Doctor upon completion of preliminary
exams.  A small portion of the time (15%), a Nurse will redirect its Patients
to the other Doctor.  Each Doctor, upon receiving a Patient, will perform a
more thorough examination of the Patient.  Upon completion of the examination,
a Doctor will sometimes (5% of the time) find it neccessary to send the Patient
to the other doctor Doctor for a second opinion.  If no second opinion is
needed, the Patient is allowed to exit the system.

****************************************************************************/
```

```
#include <iostream>
#include <fstream>

#include "SimPlus.h"
#include "EntryNode.h"
#include "Administrator.h"
#include "Nurse.h"
#include "Doctor.h"

using namespace std;

int main()
{
	// take input parameters from a file
	ifstream INFILE;
	INFILE.open( "office.in" );

	double meanIntTime, numPatients, adminMST, adminSTSD;
	double nurseMST, nurseSTSD, doctorMST, doctorSTSD;

	INFILE >> meanIntTime >> numPatients >> adminMST >> adminSTSD;
	INFILE >> nurseMST >> nurseSTSD >> doctorMST >> doctorSTSD;

	INFILE.close();

	// get a handle to the kernel
	SimPlus* handle = SimPlus::getInstance();

	// get some queues from the kernel
	EntityQueue* adminQ = handle->getEntityQueue();
	EntityQueue* nurse1Q = handle->getEntityQueue();
	EntityQueue* nurse2Q = handle->getEntityQueue();
	EntityQueue* doctor1Q = handle->getEntityQueue();
	EntityQueue* doctor2Q = handle->getEntityQueue();

	// instantiate the nodes
	// Because all of the following entities inherit from
	// ServerEntity, it is not neccessary to explicitly
	// register them with the kernel, as the ServerEntity
	// constructor does so automagically.
	EntryNode theFrontDoor( meanIntTime, numPatients, adminQ );
	Administrator admin( adminMST, adminSTSD, adminQ, nurse1Q, nurse2Q );
	Nurse nurse1( nurseMST, nurseSTSD, nurse1Q, doctor1Q, doctor2Q );
	Nurse nurse2( nurseMST, nurseSTSD, nurse2Q, doctor2Q, doctor1Q );
	Doctor doctor1( doctorMST, doctorSTSD, doctor1Q, doctor2Q );
	Doctor doctor2( doctorMST, doctorSTSD, doctor2Q, doctor1Q );

	// 1000 patient simulation
	while( Doctor::patientsProcessed() < 1000 )
	{
		// We don't need to process the events manually, because
		// the kernel handles callbacks.  If we get anything but
		// a zero back from timing, it means that there was no
		// Entity a associated with the current Event.  While this
		// may be perfectly acceptable under some circumstances,
		// this demonstration is written such that all Events should
		// be handled via callbacks.  Thus, if the pointer returned
		// from timing is not zero, we have run into an error.
		if( handle->timing() != 0 )
			handle->reportError( "Unhandled event." );
	}

	// Output some statistics.  This is not a strict best practice,
	// as writing getters for the statistics associated with each
	// Entity is more robust.  The way
	Patient p;
	cout << endl << endl << p;
	cout << "Administrator:" << endl << admin;
	cout << "Nurse 1:" << endl << nurse1;
	cout << "Nurse 2:" << endl << nurse2;
	cout << "Doctor 1:" << endl << doctor1;
```

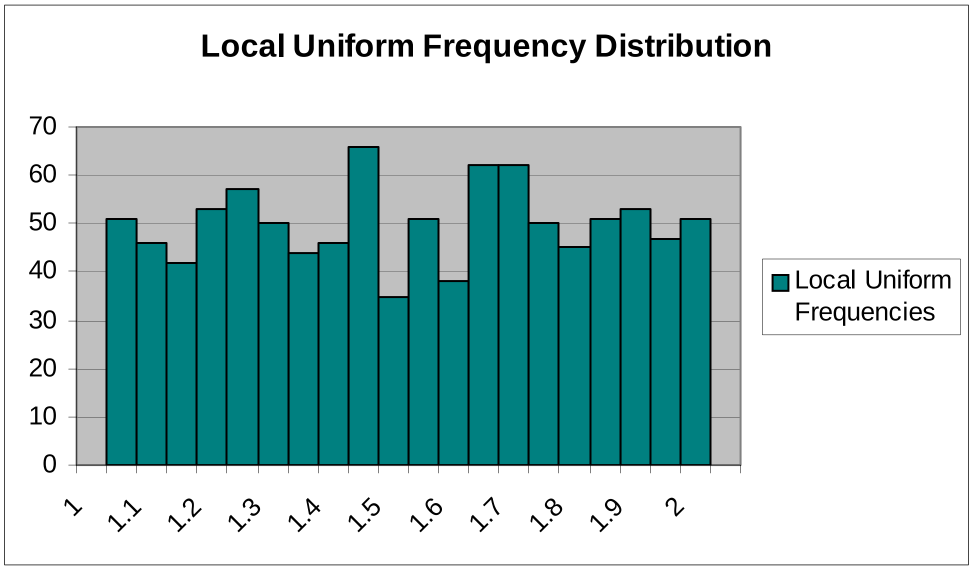
```
        cout << "Doctor 2:" << endl << doctor2;
        cout << "Simulation ended at time: " << handle->getSimTime() << endl;

        // It is neccessary to explicitly delete the kernel handle to
        // avoid leaking memory.
        delete handle;
        return 0;
}
```

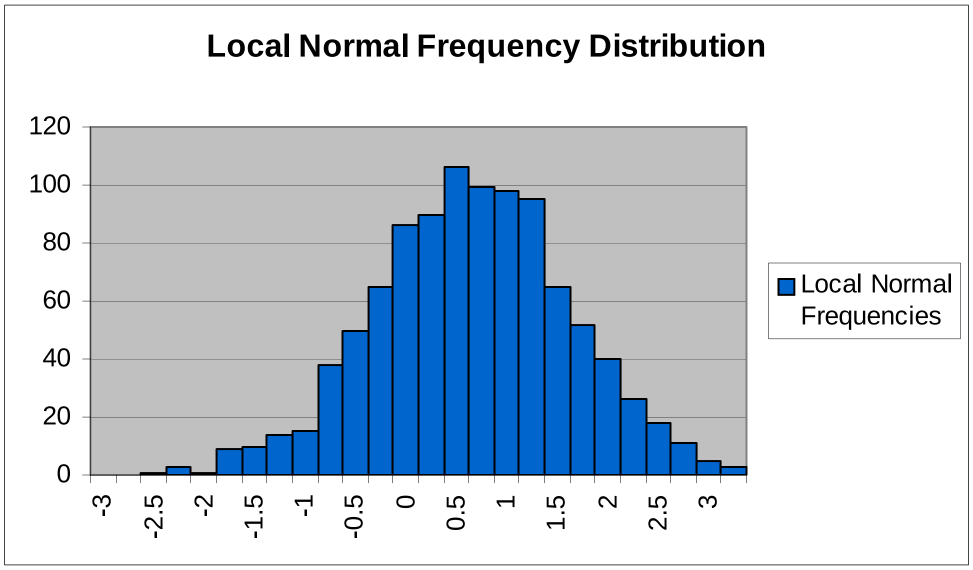
4. Results and Analysis

4.1 Random Number Generation

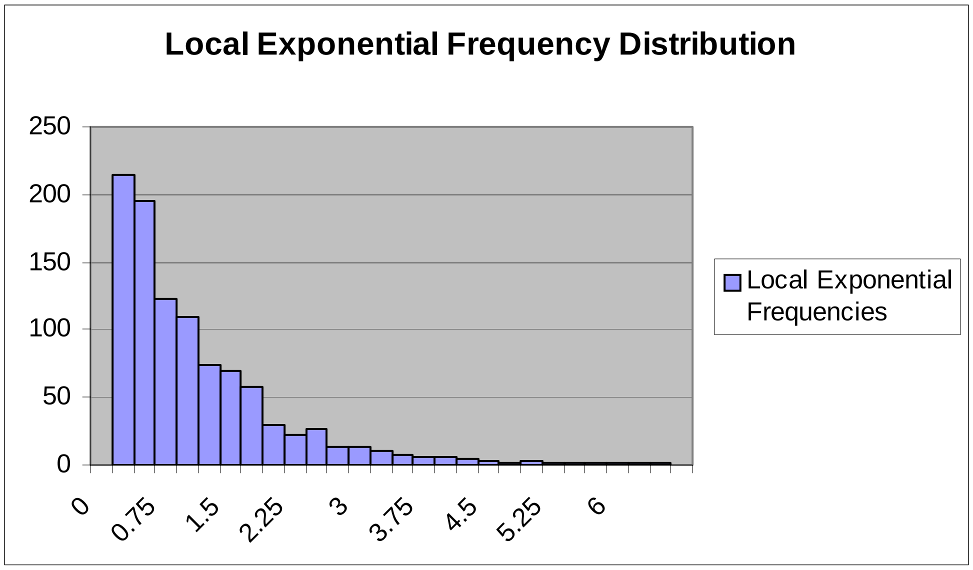
UniformRNG



NormalRNG

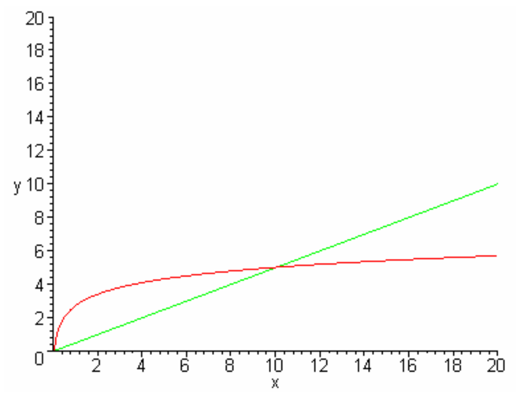


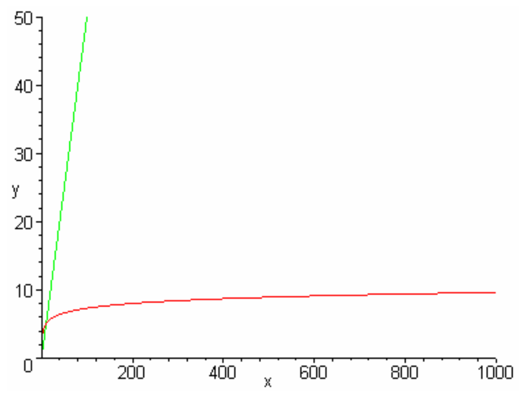
ExponentialRNG



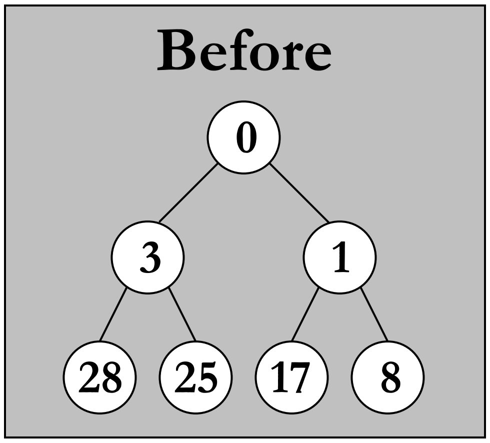
4.2 Event List Management

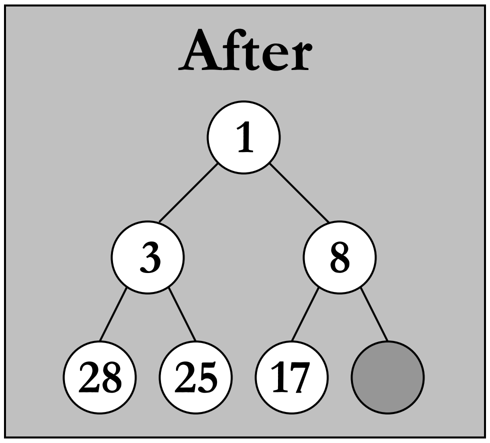
One of the key improvements made to simlib to produce SimPlus was the replacement of a linked event list with an event heap. In general, ordered insertions in a linked list require O(n/2) ~ O(n) from a time complexity standpoint. By comparison, insertions in a heap require merely O(log n) time. Of course, the flip side of this coin is that extraction from the linked list requires O(1), where the heap (being a self-organizing data structure) requires O(log n). However, for the system to be in a steady state, net insertions must be balanced by net extractions; therefore, we consider OVERALL OPERATIONS on the linked list to be O(n) and on the heap to be O(log n). Below we see the familiar time graphs for linear vs. logarithmic complexity.





The heap has two key properties: (1) the heap is a **full binary tree**. (2) each parent node is smaller than either of its child nodes. This means that the heap is a semi-ordered structure; the smallest element is always at the top of the heap, but elements in the same row have no fixed relationship to one another. Below are illustrations of a heap before and after removal of the smallest node—this corresponds to processing of the next event by the kernel.





5. Future Directions

Due to time constraints, we were unable to completely realize our vision for this project. Here's an idea of what we could have done with SimPlus if we'd had more time; for each task, we estimate how much more time would probably have been required.

Debug NetRNG

Although NetRNG performed admirably in unit testing, it proved both too slow and somewhat unreliable in integration testing. Given more time, we would like to have fixed these issues with NetRNG, which basically reduce it to niche tasks such as seeding the other RNGs. Estimated time to complete (including unit & integration testing): 6-8 hrs.

Calendar Queue

Our original design documents called for the implementation of a calendar queue for the event list in addition to a heap; this was the reason for the inclusion of EventList, an abstract class which specified a common interface for EventHeap and the postulated class EventCalendarQueue. The user may still design their own calendar queue, implementing the EventList interface. We would have needed time for design, unit testing, and integration testing. Estimated time to complete: 10-12 hrs.

Enhanced Statistics Gathering

We originally intended to add a subclass of SampST which would track the data points collected as well as their number and sum. This would allow us to calculate standard deviation, perform ANOVAs, etc. The ability for users to extend SampST is still there; however, we did not have time for the required design & testing. Estimated time to complete: 4-5hrs.

Additional RNG Adapters

We would have liked to implement additional adapter classes which, like NormalRNG and ExponentialRNG, would transform the Uniform[0,1] into non-uniform distributions. We simply didn't have time to research other transformations, and we felt that Normal, Uniform, and Exponential were sufficient to most needs; further, the user is free to subclass RawRNG to create their own adapter. Estimated time to complete: 1-2hrs per adapter.

Dynamic Modules

As things currently are, subclassing EventList, SampST, or RawRNG require minor modifications to the kernel source. Given time, we would have liked to use C/C++ dynamic module loading features to obviate this need. Using a factory design pattern and requiring modules to register themselves with the appropriate factory, subclasses implementing a common interface could have been added to the system without recompiling, let alone modifying, the kernel. This is an advanced task and would possibly be "over-the-horizon" even for a 2.0 release; however, the hierarchy was specifically designed to make this behavior possible down the road. Estimated time to complete: 25-30hrs.

6. Concluding Remarks

Implementing SimPlus has been both challenging and rewarding. We feel that SimPlus has been taken as close to completion as was possible given our short time frame. Because the delivered product is not as polished as it could have become over an entire semester (rather than a four week span), it should be considered a beta release. It is, however, suitable for experimental use.

Care should be exercised when implementing projects using SimPlus. SimPlus contains several known bugs and the API should be consulted immediately upon the observation of any unexpected behavior. Please report any previously undocumented bugs to harpers@bgnet.bgsu.edu.

Despite its beta status, SimPlus should be considered when deciding on a platform for a small to medium scale simulation project. The intuitive interface, coupled with the extensive API, should result in shortened development times for experienced C++ programmers. The extensible architecture of the SimPlus object hierarchy additionally allows for highly customized behaviors to be used in specialized simulations.