DESCRIPTION OF THE SIMLIB LIBRARY FOR COMPUTER ENGINEERING 4DK4

T. D. Todd
Department of Electrical and Computer Engineering
McMaster University

September 22, 2016

The Simlib library consists of a set of C functions which facilitate learning about discrete event simulation of computer networks. In order to use the library, place the files `simlib.h, simlib.c` and `trace.h` in your C code directory. Compile and link them into your code when building your simulation program. To use the C functions, place the following include statement at the top of any file that accesses them.

```
#include "simlib.h"
```

Simlib will define a number of variables and data structures needed for a simulation run including event definitions and an event list. In addition, types are declared for defining servers and FIFO queues.

1 Server Declaration and Functions

Servers are constructed by first declaring a pointer to the server and then by calling a library function to create the server itself. For example, a pointer to a server named `server` is first declared using the statement

```
Server *server; // or Server_Ptr server;
```

Then the server is created and initialized using

```
server = server_new();
```

The following is a list of all other functions that may be used to access a server.

1. `Server_Ptr server_new(void)`
   Memory is allocated and the server is initialized. A pointer to the server is returned. During creation, the server state is set to "FREE".

2. `void server_put(Server *server_ptr, void* content_ptr)`
   This enters an object into the server. To place something into the server you must pass a pointer to the object through `content_ptr`. The server is then marked "BUSY".

3. `void* server_get(Server *server_ptr)`
   This removes an object previously placed into service. A pointer to the object is returned and the server is marked "FREE". The calling program should cast the return pointer to the correct type.

4. `Server_State server_state(Server *server_ptr)`
   This returns the current state of a server. A pointer to the server is passed to this function and it returns an enumerated type being either "FREE" or "BUSY".

The following is an example that uses servers.

```
// Declare something to put into a server.
struct customer {
  float arrive_time;
  float service_time;
} customer_1;

// Make a server.
Server *server_1;
server_1 = server_new();

// Put the customer into it.
server_put(server_1, (void*) &customer_1);

// Sometime later take it out.
struct customer *customer_ptr;
if(server_state(server_1) == BUSY)
   customer_ptr = (struct customer*) server_get(server_1);

// Print the arriveTime.
printf("Customer arrive time = %f\n", customer_ptr->arriveTime);
```

2 FIFO Queue Declaration and Functions

FIFO queues are created in a manner similar to servers. A pointer to the queue is first declared, then a library function is called to create the queue. As in the server case, arbitrary objects may be stored in the queue and are passed via pointers. Any number of objects may be entered and are removed in first-in-first-out order. In the following example, a FIFO queue named `queue_2` is declared and created.

```
Fifoqueue *queue_2; // or Fifoqueue_Ptr queue_2;
.
.
.
queue_2 = fifoqueue_new();
```

The following is a list of functions which may be used to manipulate FIFO queues.

1. `Fifoqueue* fifoqueue_new(void)`
   This function creates a queue and returns a pointer to it. The queue is initialized with a size of zero.

2. `void fifoqueue_put(Fifoqueue *queue_ptr, void* content_ptr)`
   This function enters an object into the queue. Objects entered are added to the back of the queue. This is done by passing a pointer to the object through `content_ptr`. `queue_ptr` is a pointer to the queue which is being accessed.

3. `void* fifoqueue_get(Fifoqueue *queue_ptr)`
   This returns the object currently at the front of the queue pointed to by QueuePtr.

4. `int fifoqueue_size(Fifoqueue *queue_ptr)`
   This function returns the number of objects currently stored in the FIFO queue pointed to by `queue_ptr`.

5. `void *fifoqueue_see_front(Fifoqueue *queue_ptr)`
   This function returns a void pointer to the contents of the front of the specified FIFO queue. This permits one to inspect the contents of the front object in the queue without removing it.

The following is an example illustrating the use of these functions.

```
// Create some things to put in a queue.
struct customer {
  float arrive_time;
  float service_time;
} customer_1, customer_2;
```

```
// Create a queue.
Fifoqueue_Ptr this_queue;
this_queue = fifoqueue_new();

// Put the two customers in it.
fifoqueue_put(this_queue, (void*) &customer_1);
fifoqueue_put(this_queue, (void*) &customer_2);

// Check the size of the queue. Returns a value of 2.
queue_size = fifoqueue_size(this_queue);

// Record the arrive time of the customer at the front of the queue.
// Leave the customer in the queue.
customer_ptr = (struct customer*) fifoqueue_see_front(this_queue);
arrive_time = customer_ptr->arrive_time;

// Take a customer out of the queue.
customer_ptr = (struct customer*) fifoqueue_get(this_queue);
```

3 Simulation Run Functions

To do a simulation run you must declare and initialize it as follows. This will typically be done in `main()`.

```
Simulation_Run *this_run; // or Simulation_Run_Ptr this_run;
.
.
.
this_run = simulation_run_new();
```

This creates a new simulation run object that contains a new event list and clock. The code is written in an object oriented fashion. For this reason, a pointer to the state of the simulation (i.e., `this_run`) is passed between various functions. For this to happen you need to declare the data objects that you want to pass, then attach them to the created simulation run. This is done as follows.

```
Simulation_Run_Data sim_data;
.
.
.
```

where `Simulation_Run_Data` is a struct typedef (typically defined in `main.h`) that defines the data to be shared. The data object is then attached to the simulation run using the following statement.

```
simulation_run_attach_data(this_run, (void*) &sim_data);
```

i.e., `this_run` is a pointer to a struct with event list, clock and data (i.e., `data`) members. Now when `this_run` is passed between various functions, they have access to the complete simulation run and user defined objects. The data can be accessed via

```
Simulation_Run_Data_Ptr sim_data;
.
.
.
sim_data = (Simulation_Run_Data_Ptr) simulation_get_data(this_run);
```

where `Simulation_Run_Data_Ptr` is a typedef pointer to the data struct that is defined (usually in `main.h`).

4 Event List Functions

When a simulation run is created, an event list is created and provides functions for scheduling and executing events. A simulation clock is also declared and is initialized to 0.

To define and process an event, a function which schedules the event, and an event function are defined. In the following example, a transmission end event example is given. The event function associated with the event is `end_packet_transmission_event` and the function used to schedule the arrival event is `schedule_end_packet_transmission_event`.

```
/*
 * This function will schedule the end of a packet transmission at
 * a time given by event_time.
 */

long
schedule_end_packet_transmission_event(Simulation_Run_Ptr this_run,
       double event_time,
       Server_Ptr link)
{
  Event event;
```

```
  event.description = "Packet Xmt End";
  event.function = end_packet_transmission_event;
  event.attachment = (void *) link;

  return simulation_run_schedule_event(this_run, event, event_time);
}

/*
 * This is the event function which is executed when the end of a
 * packet transmission event occurs.
 */

void
end_packet_transmission_event(Simulation_Run_Ptr this_run, void * link)
{
.
.
.
}
```

This function creates an instance of the proper event type, then calls the Simlib library function `simulation_run_schedule_event` which schedules the event by placing it on the event list. `event` is initialized with three values. The first is a description string that is used for output when the simulation trace is enabled. The second must be the name of the event function defined for this event. The third is a pointer to any object that should be attached to (i.e., carried along with) this event. The event function will be called whenever a transmission end event occurs. In the simulation run, when one wants to schedule a transmission end event, a call is made as follows.

```
schedule_end_packet_transmission_event(this_run, 10.0, (void*) link);
```

This will schedule the event for time 10.0. The first argument is a pointer to the simulation run. The second argument must be the time that the event is to occur. The last argument is a pointer which is stored with the event and may be used to pass object references along with the event. When the event occurs, this pointer will be passed as the argument of the called event function. Note that when passing a pointer to an object, the pointer should be cast to a void pointer. If nothing is to be passed, then pass a NULL pointer using the syntax shown.

The following are the functions which may be used to handle events.

1. `long int simulation_run_schedule_event(Simulation_Run_Ptr sim, Event event, double evTime)`
   This function schedules an event of type `Event` to occur at time `evTime`. When the event occurs, the appropriate event function is called. A unique event identifier is returned when an event is scheduled.

2. `void simulation_run_execute_event(void)`
   This function removes the next event from the event list, updates the simulation clock to the new event time, then calls the appropriate event function. At that time, a pointer to any data sent with the event will be passed through the argument of the event function. See the example below.

3. `double simulation_run_get_time(Simulation_Run_Ptr this_run)`
   This function returns the current value of the simulation run clock time.

The following is an example illustrating the use of these functions.

```
long int
schedule_packet_arrival_event(Simulation_Run_Ptr simulation_run,
      double event_time, void * packet)
{
  Event event;

  event.description = "Packet Arrival";
  event.function = packet_arrival_event;
  event.attachment = (void *) packet;

  return simulation_run_schedule_event(simulation_run, event, event_time);
}

// Define a packet.
struct packet {
  int destination_address;
  int source_address;
} packet_1;

// Sometime later, schedule a packet arrival for clock = 3.0.
// Attach the packet to the event.
schedule_packet_arrival_event(this_run, 3.0, (void*) &packet_1);

// At clock = 3.0 we would find ourselves in the packet_arrval function.
//Here is the declaration of this event function.

void packet_arrival_event(Simulation_Run_Ptr this_run, void * data_ptr)
{
.
.
.

// Look at the destination address of the arriving packet.
// The passed data must be type cast to the correct type.
```

```
struct packet *packet_ptr;

packet_ptr = (struct packet*) data_ptr;
destination_address = packet_ptr->destination_address;
.
.
.
```

5 Random Number Generators

A couple simple random number generators are provided and are described as follows.

1. `double uniform_generator(void)`
   This function returns a pseudo-random number uniformly distributed over the interval $[0, 1)$. It uses the standard library function `rand()` and the constant `RAND_MAX` for this purpose. It is important that these are available.

2. `double exponential_generator(double mean)`
   This returns pseudo-random variates exponentially distributed with a mean of `mean`. Since a Poisson process has exponentially distributed interarrival times, one may be generated by scheduling events at times defined by calls to `exponential_generator`.

```
// Define a Poisson process arrival event.
void schedule_poisson_arrival(Simulation_Run_Ptr this_run,
                              double event_time, void* object);
{
  EVENT_TYPE this_event = {"Poisson Arrival", poisson_arrival};
  long int event_id;

  event_id =
    simulation_run_schedule_event(this_run, this_event, event_time,
                                  (void*) object);
  return event_id;
}

// In main(), schedule the first Poisson arrival for clock = 0.0.
schedule_poisson_arrival(this_run, 0.0, (void*)NULL);

// In the poisson_arrival function, schedule the next Poisson arrival.
// The time for the next arrival will be the current time plus the next
   exponential interarrival time.
```

```
void poisson_arrival(Simulation_Run_Ptr this_run, void *ptr) {
.
.
.
schedule_poisson_arrival(this_run,
     simulation_run_get_time(this_run),
     Clock + exponential_generator((double) MEAN_INTERARRIVAL_TIME),
     (void*) NULL);
.
.
.
```