SIMUL
Discrete Event Simulation Package[1]

Henning Schulzrinne
Department of Electrical and Computer Engineering
Department of Computer and Information Science
University of Massachusetts at Amherst
Amherst, MA 01003
(hgschulz@cs.umass.edu)

July 17, 1991

[1]This work is supported in part by the Office of Naval Research under contracts N00014-87-K-304 and N00014-90-J-1293, the Defense Advanced Research Projects Agency under contract NAG2-578 and an NSF equipment grant, CERDCR 8500332.

Abstract

SIMUL is a collection of subroutines written in C that simplify the design of discrete-event simulations for both event-based and time-based models. SIMUL is transaction-based, that is, objects in the simulation are represented by entities that can be created, scheduled and destroyed. Events are represented by user routines which are passed the entity participating in the event. Entities carry state information in the form of attributes. In addition to maintaining the event calendar and entity list, SIMUL offers routines to administer queues. Besides a general queue manager, a high-level FCFS queue mechanism is provided that hides the details of scheduling and queue maintenance from the programmer. Statistics for counters and tallies can be gathered; both regenerative and spectral methods may be used to estimate confidence intervals and terminate runs on reaching the desired interval widths.

SIMUL is supported under the DEC VMS, DEC Ultrix (Gnu C) and Turbo-C 2.0 environments.

Contents

Chapter 1

User's Guide

1.1 Introduction

SIMUL provides a set of routines which aid in the simulation of discrete-event systems. Both event-based and time-based simulations are supported. It traces its origins to the FORTRAN-callable part of SIMAN[1] and follows it closely in spirit and terminology. The simulation is based on the *entity* paradigm. Entities, representing customers, packets, jobs, and so forth, are created, scheduled together with an action subroutine and eventually deleted. The user-written action subroutine receives the entity as a parameter and can modify its attributes, pass it to another routine, reschedule or delete it. Another set of routines supports low-level queueing operations, allowing the insertion and removal of entities. Higher-level methods are provided for single-server FCFS systems, where the details of scheduling and queue maintenance are hidden from the user.

Each simulation *experiment* consists of one or more *runs*, which may be independent or connected. In the first case, statistics or state are reinitialized after each run, in the second case, a run picks up where the previous one left off. Runs can be used to generate confidence intervals by independent replication for the same parameters and model or to generate simulation estimates for a set of parameters.

1.2 Using SIMUL

1.2.1 Program Structure

Each program to be linked with the SIMUL routines consists of two types of routines. The first kind consists of routines with predefined names that are invoked by SIMUL at specific points in the simulation "life cycle". If no action is required, the routine can be omitted since SIMUL will substitute a dummy routine by default. All of these special routines have no arguments and return no value.

Before the first simulation run, `experiment()` is invoked. In it, tallies and counters are set up, seeds for the random variate generators are established, and the like. A number of SIMUL routines can only be invoked in this routine; they are described in section 1.3.

`prime()` is called prior to each run. For event-based simulation, it has to schedule the first event in the simulation. It may also take care of any other per-run initialization. By default, the simulation is event-driven, i.e., a run ends as soon as there are no more scheduled events. At that point, `wrapup()` executes and performs whatever cleanup action is deemed appropriate. For most simulations, this routine will be omitted.

For time-based simulation (see `replicate()` option), the user-written routine `tick()` is called for every clock tick. (A clock tick lasts one simulation time unit.)

A simulation run stops as soon as one of the following conditions is satisfied:

- the maximum simulation time has been reached, as defined in `replicate()` (see section 1.3.2);
- the relative halfwidth of all confidence intervals falls within a specified margin, as defined by `CI_check()` (section 1.3.4);
- a tally reaches the prescribed number of observations set by `tallies()` (section 1.3.4)
- a counter reaches the maximum count threshold set by `counter()` (section 1.3.4)
- there are no more events to be scheduled (for event-based simulation only)
- an interrupt or *kill* command is issued for the process, in which case `*** SIMUL: Interrupt 2 ***` is displayed and all further runs are cancelled

For each run, the log file shows a heading of the following format:

`*** RUN` *run-number* `of` *runs*`: ***`
`Run terminated:` *reason*
date started `(`*start clock*`) to` *date ended* `(`*end clock*`) [`*entities used*`]`
`CPU seconds:` t_1 `(out of` t_2 `wallclock seconds =` u`% utilization)`

The *date started* and *date ended* refer to the actual wall clock time and calendar date and are shown as day of the week, month, day, time, year. The *start clock* and *end clock* list the values of the simulation clock at the beginning and end of the simulation run. The CPU time is provided with a resolution of one tenth of a second. For the PC version of SIMUL, the CPU time is equal to the elapsed wall clock time. Since the resolution of the wallclock is limited to one second on some systems (Ultrix and MS-DOS), the CPU time shown may exceed the wallclock time.

1.2.2 Linking

VAX/VMS

For VAX/VMS, all relevant object and source libraries are contained in the directory with logical name `SIMUL`, which currently translates to `MERLIN::DCC$DISK:[HGSCHULZ.SIMUL]`. To set up the necessary logical names, execute `@SIMUL:SIMULDEFS`.

The user-written subroutines are compiled as usual and linked with either `SIMUL:SIMUL.OLB` or `SIMUL:SIMUL_CHECK.OLB`. The latter library contains routines that perform additional checking on input parameters, event scheduling and so forth, but may execute slightly more slowly. Since SIMUL requires support services from the extended *Numerical Recipes* library, the complete link command becomes, assuming the user routines are in `user.c`,

```
$ cc/debug user
$ link user,simul:simul/lib/incl=simul,nr:nrd/lib,share/opt
```

Ultrix

All user routines must be compiled with the GNU C compiler. The simulator requires two libraries, one for the simulator itself and the Numerical Recipes run-time library. It is recommended to set up a make file. As an example, the make file for a user simulation program called "mm1.c" should resemble the one shown below, possibly with appropriately modified location for the Numerical Recipes and SIMUL library:

```
# makefile for linking with SIMUL

SIMUL = /usr/users/dcc/hgschulz/simul
NR = /usr/users/dcc/hgschulz/recipes
CC = gcc
CFLAGS = -Wall -I$(SIMUL)

mm1: mm1.o ../libsimul.a
	$(CC) mm1.o -o mm1 -L$(NR) -L$(SIMUL) -lsimul -lnr -lm
```

MS-DOS

User routines must be compiled with Turbo C, version 1.5 or 2.0.

1.2.3 Running

SIMUL programs are executed just as regular user programs. During execution, progress is reported by writing `Run` n `of` m to the console. Also, the result of confidence interval checks are displayed in the format

`CI check` t `@` n`:  spectral =` s `regenerative =` r

where t denotes the tally number, n the current observation count, s and r the relative widths of the intervals for spectral and regenerative method, respectively.

Whenever queues and entities are reinitialized (normally after each run, but possibly only after the last run), a message of the form

l `loose entity found,` t `total`

is displayed. Here, l refers to the number of entities that were not linked to any queue or the scheduler, while t counts the total number of entities reclaimed from queues and the scheduler calendar. It denotes the maximum number of entities that were used concurrently during the simulation run.

1.3 The Experiment Section

Within the user-written `experiment()` routine, basic parameters for the entire simulation are set up. The calls described in this section are valid only within `experiment()`. Some commands described below take options as a character string. Options are separated by white space; option names can be written in either lower, upper or mixed case and may be abbreviated as long as they remain unique. Flags, that is, single words, can be negated by prefixing them with an exclamation mark (!). Options that are omitted take on default values.

1.3.1 Project

A project title and author can be defined with a call to `project(title,author,logfile_name)`, where all arguments are strings. All simulation log output, including summary statistics, are written to the logfile. The user program can write to the file descriptor `logfile` to add to the logfile. Existing log files are not overwritten. If necessary, a version number is appended to the file name.

Example:

```
project("M/M/1 queue","A.K. Erlang","MM1.log");
```

The call to `project()` should be the first SIMUL call in the experiment section.

1.3.2 Replication and Run Length

The call to `replicate(nrun, tbeg, dtrun, options)` establishes the basic parameters of the experiment. **Note:** `replicate()` must be invoked before any calls to `tallies()`, `counter()` or `rankings()`. An experiment will consist of *nrun* runs, starting at simulation clock *tbeg*. A run will last at most *dtrun* simulation clock units. The following options[1] can be defined through `replicate`:

QUEUES=q The maximum number of queues.

ENTITIES=e The maximum number of concurrent entities.

ATTRIBUTES=a The maximum number of attributes for each entity.

SERVERS=s The maximum number of servers.

KEEP_STATS With this flag, statistics (tallies and counters) are maintained across runs. By default, they are reinitialized before each run.

KEEP_STATE If this flag is specified, the system state at the beginning of a run is that at the end of the previous run. By default, the system state is reinitialized before each run; that is, the calendar is cleared, all entities are returned to the free pool and the simulation clock is reset to *tbeg*.

CI A confidence interval for all tallies based on the method of independent replications is computed (default).

SHOW Show individual statistics for each run (default). Individual tallies can override the global value.

[1] For a general discussion of option syntax, see the beginning of this section.

TIME_BASED This flag calls for a time-based rather than an event-based simulation. At each clock tick, which lasts one simulation time unit, the user-written routine `tick()` is called. The scheduler is disabled, i.e., calls to `sched` have no effect.

SILENT This flag suppresses the termination and CPU usage message printed to the log file for each run. This is useful if an experiment consists of a very large number of runs.

Example for a simulation consisting of two runs, each starting at 0. and lasting at most 10^6 time units. All tallies and counters are displayed after each run, but no confidence intervals are computed.

```
replicate(2,0.0,1e6,"SHOW !CI");
```

1.3.3 Queue Discipline

Queue disciplines for queue q can be established by a call to

```
rankings(q, r, k)
```

The rule r can be one of the following (quoting the corresponding text from the SIMAN manual).

FIFO	The entities are ranked based on their time of insertion in the file, with early insertion given priority. This is a first-in, first-out rule. The value of the argument k is ignored.
LIFO	The entities are ranked based on their time of insertion in the file, with late insertion given priority. This is a last-in, first-out rule. The value of the argument k is ignored.
LVF	The entities are ranked low-value-first, based on the value of the attribute k of the entity. Ties are resolved using the FIFO rule.
HVF	The entities are ranked high-value-first, based on the value of the attribute k of the entity. Ties are resolved using the FIFO rule.

1.3.4 Counters

A counter n ($0 \leq n < 20$) with the name string *id* is defined by

```
counter(n, id, options)
```

The following options can be specified as a character string:

KEEP This flag causes counters to be maintained across runs. By default, counters are reset at the start of each run.

SHOW Counters are displayed. By default, counters are not shown.

STOP=s The simulation stops as soon as the counter reaches the value s.

1.3.5 Tallies

Similarly, a tally or observation variable is created by calling

```
tallies(n,id,options).
```

n and *id* have the same meaning as for counters. The following options can be specified as a character string:

KEEP This flag causes tallies to be maintained across runs. By default, tallies are reset at the start of each run.

SHOW Tallies are displayed. This is the default.

START=*s* The first *s* observations are ignored.

STOP=*s* The simulation stops as soon as the tally has been invoked *s* times, regardless of how many of these observations have been ignored.

BETWEEN This flag causes `tally()` to record not the actual observation but rather the difference between the previous and current observation. This is useful for tallying interarrival times.

TIME Tallies the average of the last value and the current value, divided by the simulation time elapsed since the last call to `tally()`. Experimental.

TRANSIENT With this flag, an initial set of observations is recorded. By default, the value of every tenth call to `tally()` is recorded, up to the 1000th call to `tally()`. The interval *i* (default 10) and duration *d* (default 1000) for all tallies can be changed by calling `transient(i,d)`.

BUCKET_WIDTH=*w* Enables the collection of histogram information with buckets of width *w*. If this option is absent, no histograms will be produced.

LO_BUCKET=*l* The first bin for histograms will have a lower edge of *l*.

HI_BUCKET=*h* The last bin for histograms will contain *h*. It may reach beyond *h* if *h* is not separated by an integer multiple of *w* from *l*.

SPECTRAL Enables computation of confidence intervals using the spectral method.

REGENERATIVE Enables computation of confidence intervals using the regenerative method.

DIGITS=*d* Display *d* decimal digits for tallies. By default, 3 digits are shown.

VARIANCE Display estimate of sample variance. This estimate is meaningful only for uncorrelated samples. Extreme caution in interpreting this value is advised.

Rather than stopping a simulation after accumulating a fixed number of observations, a sequential stopping rule is usually preferable. A call to

```
CI_check(eps,first_check)
```

enables sequential stopping. As implemented here, a simulation run is terminated if all confidence intervals have a relative halfwidth less than *eps*. A tally is first tested after *first_check* observations. Then, the testing interval increases by a factor of 1.5 each time. For example, `CI_check(0.10,10000)` ensures that all confidence intervals have a halfwidth of no more than 10% of the point estimate, with a first check performed after 10,000 samples and subsequent checks at 15,000, 22,500, etc.

The confidence level for all confidence intervals computed is set by

```
confidence(level).
```

For example, `confidence(0.90)` establishes 90% confidence intervals. (This is the default.)

Observations are also be written to a text file with name *fn* if set up through

```
tally_log(n,fn)
```

in the experiment section. Note that no higher moments are computed as the traditional estimates lead to grossly misleading results for the (usual) case of correlated observations.

For histogram, samples falling outside the range specified are tabulated in two bins for outliers in either direction. The log file displays the absolute and relative counts, both for each bin and cumulative.

See section 1.9 for functions to gather statistics and obtain current point estimates.

1.3.6 Defining Parameters

Parameter values for set *ns* with name *name* are defined with

```
parameter(ns, name, n, v1, ...).
```

or

```
parameterv(ns, name, n, v).
```

The number of values is given by n, the values themselves form a variable-length list of parameters (`parameter`) or an array (`parameterv`). Note that the values have to be of type double or floating-point constants; using integers will give incorrect results without warning. The function returns the value of the first element in the list. The list of parameters is shown in the log file. Example, defining four parameters in set 2:

```
parameter(2,"Set 2",4,1.2,2.3,3.4,4.5);
```

or, equivalently,

```
double p[] = {1.2,2.3,3.4,4.5};
parameter(2,"Set 2",4,p);
```

Parameters are used mainly by the random-variate generators; their values may be changed and retrieved with `setp` and `getp`, respectively, described in section 1.10.2.

1.4 Entity Creation, Modification and Destruction

Entities are the basic objects of the simulation model, representing customers, packets, jobs, transactions, and so forth. Variables to hold entities are declared with the type `EntityType`, as in

```
EntityType packet, customer, job;
```

Within function and procedure descriptions, the letter `e` will be used to denote entities. A maximum of 500 entities can be active at one time.

1.4.1 Creation

The low-level function `create_()` returns a pointer to a fresh entity, with all attributes undefined. The index number (returned by `id()` or `bid()`) is a strictly positive serial number incremented with each call.

If entities are to be created at regular intervals,

```
create(double (*iat)(ip,is), ip, is,
       double (*bsz)(bp,bs), bp, bs,
       n, void (*first) EntityType))
```

is more convenient. Here, entities are created with a simulation time separation given by the return value of the function *iap* passed as a parameter to `create()`. *iap* will be passed the two integer values *ip* and *is*. At each creation instant, a batch of the size returned by the function *bsz* is created. *bsz* will be passed the integer values *bp* and *bs*. If only single-entity batches are desired, the *bsz* parameter can be set to zero. Typically, the functions *iap* and *bsg* are drawn from section 1.5 and produce a random variate, where *ip/bp* and *is/bs* convey parameter set and seed index. The simulation run is terminated after n batches have been created. If n is negative, `create()` will produce entities until some other mechanism stops the run.

Each entity is "sent", that is, passed as an argument to, function *first*. As an example, suppose 1,000 entities with exponentially distributed time between creation (parameter set 2, seed 1) are to be generated and passed to routine *cpu*:

```
create(ex,1,2,1000,cpu);
...
void cpu(EntityType e)
{
  /* process entity */
}
```

Entities are labeled with their creation time and an index number (see section 1.4.2).

1.4.2 Attributes

Each entity e carries with it a list of twenty double-precision numeric attributes. On entity creation, all attributes contain garbage. The nth ($n \in 0 \ldots 9$) attribute is set by `seta(e, n, val)` to the new value *val* or incremented by one with `inca(e, n)`. `assign(e,a)` copies the values of the double-precision array a to the entity e. Note that `seta()` returns the assigned value. The respective retrieval operations are `A(e,n)` and `copy(e,a)`. If the entity is undefined, a very large positive number (approximately 10^{37}) will be returned.

Each entity also maintains a current station counter, which is used by the server functions. It is set with `setm(e,ns)` to the integer value *ns* and retrieved with `M(e)`.

Each entity *e* is stamped with its creation time (in simulation time units), which can be inspected by `creation_time(e)`. For brevity, the age (`tnow() - creation_time(e)`) is directly available as `age(e)`

Entities produced by `create()` also bear a "serial number" identifying the batch number it belongs to, starting at one. It is returned by the function `bid(e)`. The sequence number of an entity within its batch is returned by the function `id(e)`. However, if the *bsz* parameter of `create()` is zero, that is, all batches contain exactly one entity, `id(e)` returns the same value as `bid(e)`.

1.4.3 Destruction

Entity *e* is returned to the system pool by calling `dispose(e)`.

1.5 Random Variates

1.5.1 Seeds

`seed(s,d)` makes the seed with index d ($d \geq 0$) the new seed for random variate stream s. The seeds are listed in Tables 2.5 and 2.6. By default, seed i is set to the $i+1$th seed from Table 2.5, reading across rows.

1.5.2 Random Variates

The following routines generate a single variate from the indicated distribution, using stream s (valued zero and up) and parameter set p ($0 \ldots$). The parameter set p has components p_1, p_2, ...

`ra(p,s)`	uniform distribution over $(0,1)$, p ignored
`er(p,s)`	Erlang distribution with $r = p_2$ stages and mean $1/(rp_1)$
`ex(p,s)`	exponential distribution with mean p_1 (parameter $1/p_1$)
`dp(p,s)`	discrete probability distribution (see below)
`di(p,s)`	discrete probability distribution (see below)
`co(p,s)`	constant p_1
`np(p,s)`	Poisson distribution with mean p_1
`ge(p,s)`	Geometric distribution on $[0,\infty]$ with parameter $P[X=0]=p_1$ (thus, with mean $\frac{1-p_1}{p_1}$ or $p_1 = 1/(1+\text{mean})$)
`bi(p,s)`	Bernoulli (0,1)-distribution with parameter $P[X=1]=p_1$
`bn(p,s)`	Binomial distribution with parameter p_1 over range $[0,p_2]$

For the discrete distribution `dp`, the parameter vector contains pairs $P(x \leq x_k), x_k$, i.e., the odd parameters denote values of the cumulative distribution function while the even parameters denote the associated values of the random variable. For example, the parameter list 0.1,1.0, 0.2,2.0, 0.4,3.0, 1.0,4.0 describes a random variable that assumes the value 1 with probability 0.1, 2 with probability 0.1, 3 with probability 0.2 and 4 with probability 0.6. The algorithm for `dp` is based on linear search and requires $O(n)$ time for each sample, where n is the number of values in the discrete distribution.

For the discrete distribution `di`, the parameter vector contains the points of the probability distribution. The first point is associated with value zero. `di` is significantly faster than `dp` and executes in $O(1)$, plus a set-up time after each change of the probability distribution of $O(n)$. `di()` uses the alias method [8, p. 158f]. `di()` is significantly faster than `dp()` for long-tailed distributions, but slightly less flexible in that the random values are limited to the range $0 \ldots n$.

Note that p is not used by `ra`, while `co` ignores s. The dummy parameters were included to simplify using these functions with `create`.

1.6 Scheduling

`sched(e,ev,dt)` schedules event *ev* for entity *e* in *dt* simulation time units. Events are simply subroutines that take are passed an entity. Note that an entity can only be scheduled for a single event at a time; attempt to re-schedule an entity will lead to a run-time error.

The event handler is passed the entity. Example:

```
sched(job,cpu_dept,dt);
```

schedules entity *job* for the event handler routine `cpu_dept` *dt* time units from the time of invocation. The routine `job` is declared as

```
void cpu_dept(entity e)
{
   ...
}
```

`nevent(e)` returns the pointer to the event handler scheduled for entity *e*, while `tevent(e)` returns the time that event will occur.

1.7 Low-level Queueing

SIMUL maintains 100 queues numbered from zero on up that can hold entities. Entity e is inserted at the end of queue q with `insert(e,q)` and removed with `fremove(e)`. (There is no need to specify the queue since the entity record contains that information.) The current number of entities queued in q is returned by `NQ(q)`. The first, nth ($n \geq 1$) and last entry of queue q are returned by `lfr(q)`, `lrank(n,q)` and `llr(q)`, respectively. The predecessor and successor of entity e are given by `lpred(e)` and `lsucc(e)`. In all cases, the function have a return value of type `EntityType` and return the value zero if the request cannot be fulfilled (e.g., the desired entity does not exist).

1.8 FIFO Queues

While all queues of interest can be built from the elements described in section 1.7, this can be rather tedious, as the user program has to insert each entity into the queue, obtain the next entity on service completion and remove it from the queue. For FIFO queues, a set of routines is provided that hides all these details. The queue mechanism discussed in this section is completely independent of that in the previous section, i.e., the same queue index can be used for a both.

A FIFO queue with the non-negative integer index q and departure handler *dh* is established by calling

```
setup_server(q,dh)},
```

typically within `experiment()`. *dh* is passed the queue identifier q and the entity departing from the queue. Rather than maintaining an ordered list of entities to be scheduled for service, these routines compute the end of service for an arriving customer, as this is known on arrival for a FIFO-queue assuming that the service time is provided on queue arrival.[2]

An entity e with service demand s enters queue q by calling `enter_fifo(q,e,s)`. The amount of work, consisting of the work remaining for the customer in service and the service demands of the queued customers, is returned by `current_work(q)`, while `in_system(q)` provides the number of customers in the system, waiting and in service.

1.9 Gathering Statistics: Counters and Tallies

Counter n is incremented by the integer *inc* with a call to `count(n, inc)`. A negative value of *inc* can be used to decrement the counter. The current value of counter n is returned by `NC(int n)`.

An observation with value *val* is recorded for tally n by calling `tally(n,val)`. Note: Do not use `tally()` from within the `experiment()` section; it will crash the program. If the tally was defined with the option `BETWEEN`, the difference between this observation value and the previous one will be recorded instead. For regenerative confidence interval estimation, a call to `cycle(n)` indicates that a regeneration cycle has just been completed. The current estimates for some summary statistics of tally n are also available by calling appropriate functions:

`tavg(n)`	expected value, average
`tavg_regen(n)`	expected value, regenerative estimate
`tmin(n)`	minimum observation
`tmax(n)`	maximum observation
`tnum(n)`	number of observations
`tvar(n)`	sample variance

Warning: The sample variance estimate is valid only for i.i.d. random samples. Most performance metrics of interest in queueing theory are strongly correlated, yielding variance estimates that are too small.

1.10 Miscellaneous Routines and Global Variables

1.10.1 Page Size

The routine `page(h,w)` establishes the physical page size as h lines and w columns. Currently, this information is not used, but might be in the future.

[2]The work in the queue is updated on arrival, the number of customers on arrival and departure.

1.10.2 Parameters

`setp(ns,np,val)` sets parameter *np* of set *ns* to the new floating-point value *val*; `getp(ns,np)` returns the current value of the *np*th parameter in set *ns*.

1.10.3 Current Run Number

`run()` returns the index of the current run, counted from one. The run number is also written to the standard output device in the format "`Run x of y`".

1.10.4 Current Simulation Time

`tnow()` returns the current simulation clock value.

1.10.5 End of Simulation

`tfin()` returns the clock value at which the simulation will end, if available. If the simulation is not terminated by reaching a value of the simulation clock, a very large real number is returned instead.

1.10.6 Command-line Arguments

The global double-precision array `argv[]` contains the numerical values of the command-line arguments, with the first found in `argv[0]`. The number of arguments is given in `argc` and limited to twenty. Non-numeric command line arguments are translated to the value zero.

1.10.7 Interactive Parameter Entry

The library routines `prompt_int(msg)` and `prompt_double(msg)` from the extended NUMERICAL RECIPES library may be used to conveniently enter run-time parameters interactively.

1.10.8 Log File Descriptor

The global file descriptor `log_file` can be used by the user program to write messages to the simulation log declared in `project()`. Example:

```
fprintf(log_file,"Queue %d overflow\n",q);
```

1.10.9 Model Creation Date

A string can be assigned to the global character variable `model_created` which is then printed in the log file. Example:

```
char *model_created = __FILE__;
```

1.10.10 Trace Flag and Tracing Routine

If the global variable `trace` or the DCL symbol `TRACE` is assigned a non-zero value[3], routines creating, queueing, dequeueing and destroying entities will print a trace message of the format:

clock: facility # batch_id/entity_id other_messages

as in

```
1.234e-01: CREATE     #   348/  7;
1.345e-01: INSERT     #   348/  1; into   2 (  3) =9.023452 1 7 9
1.350e-01: REMOVE     #   348/  1; from   2 (  2) 7 9
1.356e-01: DISPOSE    #   200/  9; age 5.393927
1.500e-01: TALLY                 ; 4.68736 ->  3
```

For queues, the value in parentheses shows the queue length after insertion or deletion. For queues with HVF and LVF ranking, the value after the equal sign shows the value used for insertion. If queue tracing has been enabled (see below), the following numbers are the entity ids, in queue order from left to right. The tally trace provides the value assigned (here, 4.68736) and the tally number (3).

The extent and content of the trace report depends on the value of the trace variable. Values from the table below can be added together.

Constant	value	effect
`TRACE`	1	enable tracing for entities and queues
`TRACE_TALLY`	2	enable tracing for `TALLY`
`TRACE_QUEUE`	3	show queue content

For example, a trace value of 3 traces all entity-related operations and tallies, but does not show the queue state.

The routine that prints the clock, facility and entity identifier is available to the user as `trace_report(s,e)`, where the string *s* identifies the caller and *e* is the entity acted upon. The routine does not print a newline.

1.11 Paginating output

Summaries for runs are separated by form feeds. Since a summary rarely fills a whole page, listing these summaries wastes a lot of paper. The program `PAGE` collects as many summaries as will fit completely without splitting a summary across pages for each page. It is invoked with the input file as the first argument and the output file as the second. If the second argument is missing, the output file is the next higher-numbered version of the input file. The page length is currently fixed at 86 lines, corresponding to the dense format on the LN03.

[3] Assignments to the global variable override the value of the DCL symbol.

1.12 Extracting columns from output listing

The utility `colcp` may be used to extract certain columns from the simulation log, e.g., for statistiscal analysis or plotting. The first argument, p_1, to `colcp` specifies the starting column. If the second argument, p_2, is positive, `colcp` extracts the number of columns specified by p_2, beginning at column p_1. If p_2 is missing, it is taken to be a very large positive number, causing the remainder of the line beginning with the starting column to be copied. If p_2 is negative, all but the $-p_2$ columns starting at column p_1 are copied. `Colcp` prompts for the input and output file. If they have the same name, the output file will have a version number higher than the input file.

Chapter 2

Implementation Details

2.1 Confidence Intervals Using the Spectral Method

Confidence intervals for the means of tallies may be computed using the spectral method [2, 3]. The method is invoked by the option `SPECTRAL` in the call to `tallies()`. Internally, `batch(`n`,`L`)` sets up the necessary parameters. n names the tally for which confidence intervals are desired; L determines the minimum number of batches to be used. (The maximum number of batches is $2L$.) L is typically chosen as 100. The interval halfwidth and the confidence interval limits are shown in the tally listing.

2.1.1 Computation of constants C_1 and C_2

According to [2], $\hat{p}(0) = C_1(K,d)e^{\hat{a}_0}$ is an approximately unbiased estimate of $p(0)$. The constant $C_1(K,d)$ is given as

$$C_1(K,d) = e^{-\sigma^2/2} \tag{2.1}$$

where

$$\sigma^2 = 0.645 s_{11}(K,d) = 0.645[(\mathbf{X}'\mathbf{X})^{-1}]_{1,1}, \tag{2.2}$$

i.e. the upper leftmost element of the inverse. (Compare for example [4, p. 512].) $\mathbf{X}$ is the K by $d+1$ design matrix for the least squares fit $\mathbf{Y} = \mathbf{X}\alpha + \epsilon$ with elements of $\mathbf{X}$ being $x_{i,j} = f_i^{j-1}$. Throughout, f_n is taken to be $(4n-1)/2N$, letting it range from slightly above 0 to just below 0.5 as n varies from 1 to $N/4$.

While the f_n^k values depend on the actual value of N, the s_{11}, the variance of the constant term in the polynomial $\sum_{i=0}^{d} a_i f_n^i$, element does not. "To see this, suppose you have $d = 2$, frequencies $f_1, f_2, \ldots, f_K$ and data points $Y_1, Y_2, \ldots, Y_K$. Let the optimal (least squares) coefficients be $\hat{a}_0, \hat{a}_1$ and $\hat{a}_2$. If we now scale the frequencies by c, the new model is equivalent to $Y = a_0 + a_1(cf_c) + a_2(cf_c)^2 + \epsilon$ (with the original frequencies) and therefore the least squares solution is $\hat{a}_0, \hat{a}_1/c, \hat{a}_2/(c^2)$." (Philip Heidelberger, private communications)

C_2 is the number of degrees of freedom for the t-distribution and given as

$$C_2(K,d) = 2/(e^{\sigma^2} - 1) \tag{2.3}$$

The routines `sigma()`, `C1()` and `C2()` compute the required quantities.

2.1.2 Confidence Interval Computation

The algorithm presented in [2, p. 236, 239] proceeds as follows:

1. Agree on a minimum number of batches to be collected and call that number L. The actual number of batches, M, will fall between L and $2L$.

2. Initialize the batch size B to 1 and set the current bin to 0.

3. Collect data samples with each call to `tally()`, adding the value to the current batch bin. If the number of samples has reached the current batch size B, proceed to the next bin. If the last bin (or $2L$th bin) has been filled, re-batch the data by assigning the sum of the $2j$th and $2j+1$ to the jth bin, where $j = 0, 1, \ldots$. After re-batching, double the current batch size B and re-start at bin L. Sampling is performed by the routine `sample()` invoked by `tally()`.

4. At the end of the run, obtain batch means $x(j)$ by dividing the contents of all bins by B. Call the number of batch means M. It will lie between L and $2L$.

5. Compute the periodogram

$$I(n/M) = \left| \sum_{k=0}^{M-1} e^{-j2\pi k/M} \right|^2 /M = |X(n)|^2/M \qquad (2.4)$$

   where $X(\cdot)$ is the sequence of $2K+1$ DFT coefficients computed by using a "naive" (direct) implementation of the discrete Fourier transform. This approach was chosen to avoid the potential problems of having to zero-pad the sequence to an integral power of 2 samples, necessary for the application of (straightforward) FFT algorithms. It is not known whether the zero padding will significantly affect the computed confidence intervals (Philip Heidelberger, private communication). The performance penalty incurred by forsaking the FFT is not very severe in this case, however, since the number of data points is typically less than 200 and the number of desired frequency point on the order of 50. Thus, about 10,000 complex multiplications and additions are required.

6. Compute the logarithmic averaged periodogram

$$J(f_n) = \log_e(I((2n-1)/M) + I(2n/M))/2), \text{ for } n = 1, 2, \ldots, K \qquad (2.5)$$

   with f_n defined as above, i.e., $f_n = (4n-1)/2N$. That is, each sample of $J(f_n)$ is the logarithm of the average of two consecutive samples from the power spectral density. This operation reduces the number of points to $M/4$. With the frequency definition shown, the frequencies now range from $f_1 = 3/2N$ to $f_{M/4} = 0.5 - 1/M$. Note that the zero frequency is not included in the averaging. Also, we only need to consider the first K points since these are the points that will fitted a polynomial. [5, p. 132] describes the smoothing operation and its properties in detail.

7. Fit a polynomial in f_n of degree d to the first K points of the smoothed spectrum. In our case, the quantities d and K are fixed at 2 and 25, respectively. In general, we have the

relation $K \leq M/4$ or $M \geq 4K$. Therefore, L is constrained to be at least $4K$. The value 100 chosen here satisfies the constraint. The polynomial has the form

$$g(f_n) = \sum_{k=0}^{d} a_k f_n^k \text{ for } n = 1, 2, \ldots, K. \tag{2.6}$$

The coefficients are estimated in a least-squares sense using singular value decomposition to solve the normal equations (routine `svdfit()` with `fpoly()`), resulting in the estimates $\hat{a}_k$ for the coefficients.

8. The zero-frequency point of the power spectral density is estimated as

$$\hat{p}(0) = C_1(K, d)e^{\hat{a}_0} \tag{2.7}$$

9. The confidence interval of level $1 - \alpha$ (with 0.9 or 0.95 being a typical value for $1 - \alpha$) has a width

$$CI = t_m(1 - \alpha/2)\sqrt{\hat{p}(0)/M} \tag{2.8}$$

where $t_m(x)$ is the percentile value of Student's t-distribution for $m = C_2(K, d)$ degrees of freedom at point x.

2.1.3 Example

The example in [2, p. 239] was used to test the performance of the implemented algorithm. The example consists of a simple, six-node closed queueing system with exponential service times and first-come, first-served disciplines, as specified in Table 2.1. Two sets of parameters are considered.

node	type	service time	
		A	B
1	M/M/∞	100	100
2	M/M/1	1.00	1.00
3	M/M/1	1.39	5.56
4	M/M/1	1.39	5.56
5	M/M/1	12.5	25.0
6	M/M/1	12.5	25.0

Table 2.1: Example System

The routing probability matrix for both model A and B is given by

$$\begin{bmatrix} 0 & 1 & 0 & 0 & 0 & 0 \\ 0.2 & 0 & 0.36 & 0.36 & 0.04 & 0.04 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \end{bmatrix}$$

Using mean-value analysis, the theoretical performance measures desired can easily be computed. The system was simulated using SIMUL, with results collected in Table 2.2. All simulation results are the average of 50 runs with 14,000 samples each of the variable indicated. In each run, the first 500 samples were discarded to control the effect of the initial conditions.

The response time R is measured from the departure of a customer from the service center 1 to its next arrival at the same service center. (Service center 1 represents the "thinking" state of interactive users.)

		A		B	
		theory	simul.	theory	simul.
Waiting times	W_2	3.77	3.77	0.84	
	W_3	1.02		34.2	34.4
	W_4	1.02		34.2	
	W_5	9.2		20.9	
	W_6	9.2		20.9	
Utilizations	U_2	0.89		0.46	
	U_3	0.44		0.92	
	U_4	0.44		0.92	
	U_5	0.44		0.46	
	U_6	0.44		0.46	
response time	R	41.2	41.1	171	171

Table 2.2: Theoretical and Simulated Performance Results

For the four cases (W_2 and R for model A, W_3 and R for model B), confidence interval half widths CI were computed using the batching method described. The parameters were $d = 2, K = 25, B = 128, M = 105, 1 - \alpha = 0.90$. The results in Table 2.3 are the average of 50 independent runs. σ^2_{CI} labels the variance of the estimate. Coverage denotes the fraction of confidence intervals generated that actually contain the theoretical mean. Except for the coverages, which tend to be higher in the present implementation, the values agree closely with those in the reference, Table III.

For $n = 50$ experiments, the proportion of covering confidence intervals is approximately normally distributed with standard deviation

$$\sigma = \sqrt{\frac{pq}{n}}$$

where $p = 1 - q$ is the theoretical coverage probability, 0.90 in our case. For $n = 50$, $\sigma = 0.04243$, for $n = 100$, $\sigma = 0.03$. The 90-percentile of the standardized normal distribution is 1.645. Thus the method is rejected at the 0.10 level of significance for 50 experiments if the fraction of covering intervals is lower than 0.83 or higher than 0.97. For 100 experiments, the cut-off points are at 0.851 and 0.949.

As an additional check, a simple M/M/1 model was simulated ($\lambda = 0.8, \mu = 1.0$). Each run consisted of 100,000 customers served, with the first 2,000 dropped to control transient effects. SIMUL was modified to use the same multiplicative random number generator as found in RESQ[6]

variable	CI	n	σ^2_{CI}	coverage
Model A, W_2	50	0.474	0.018	0.92
Model A, R	50	1.59	0.104	1.00
Model B, W_3	50	3.66	1.00	0.92
Model B, R	50	5.69	1.61	1.00

Table 2.3: Confidence Intervals for CPU model

(see below). The first SIMUL run was compared with an actual RESQ simulation and found to be identical in the sample path. Confidence intervals were computed for the mean of the system time (nominal value: 5.00) and the 0.90 point of the system time distribution (11.5, with exact probability 0.899741). The results are collected in Table 2.4. It may be of interest to note that the RESQ run consumed 363.28 seconds of IBM mainframe CPU time, compared to 223 seconds of real MicroVAX II time for the equivalent SIMUL run.

For the first run, the spectral estimates from SIMUL and RESQ were compared. For the mean, RESQ estimated the width of the interval at 0.01135, while SIMUL pegged it at 0.01147. For the point on the distribution, the respective values came in at 0.19617 and 0.19637. In either case, the RESQ interval is smaller by a factor of about 0.9989. This points to a potential difference in the computation of the point on the t-distribution or an off-by-one divisor somewhere. Since the two intervals agree to three significant digits, it does not seem worthwhile to track down the source of the disagreement.

variable	r.v. generator	runs	average	avg. CI	coverage	below	above
mean delay	RESQ	50	5.0028	0.2635	0.96	1	1
	Num. Recipes	50	4.9845	0.2868	0.98		
	Num. Recipes	100	4.9219	0.2720	0.93	7	0
distribution	RESQ	50	0.8995	0.01388	0.92	2	2
	Num. Recipes	50	0.8989	0.01423	1.00	0	0
	Num. Recipes	100	0.9039	0.01518	0.88	0	12

Table 2.4: Confidence Intervals for M/M/1 Queue

In a third experiment, a sequence of ten M/M/1 queues was modeled. Again, the mean end-to-end delay and a point on the end-to-end delay distribution were measured and confidence intervals computed. The system had an arrival rate $\lambda = 0.8$, each server had unity service time. The theoretical end-to-end delay is 50.00; the fraction of customers that exceed an end-to-end delay of 78.5 time units is 0.05. After 100 runs with 100,000 customers (first 2,000 discarded), the fraction was measured at 0.04873, with an average spectral confidence interval estimate of 0.01770 and a coverage of 0.89. One interval fell above, ten below the true percentage. The 0.90 confidence interval computed through independent replications, based on 100 runs with a total of 10^7 customers, came out to (0.04718,0.05027), a half-width of 0.001545. Note that this is fairly close to $0.01770/\sqrt{1000}$, according to the notion that confidence intervals widths decrease with the square root of the number of observations.

The mean end-to-end delay was estimated at 49.895. The spectral confidence intervals averaged to 1.4442, covering the true mean in 92% of the runs. Five confidence intervals fell below the true mean, three above. For the mean end-to-end delay, the 0.90 confidence interval was also estimated through independent replications, spanning the range (49.7721,50.0188). The half-width of 0.1234 again confirms the square-root behavior.

2.2 Random Numbers

All random variates are derived from a source generating uniformly distributed pseudo random numbers in the interval $(0,1]$. Exponentially distributed variates with mean $1/\mu$ are generated by the equation $-\log_e(u)/\mu$, where u is a $(0,1]$ uniform variate. It is clear that this method makes it necessary to exclude zero from the domain of the uniform random number generator.

By default, the random numbers are generated by a linear congruential generator of the form

$$\begin{aligned} x_n &= (4096x_{n-1} + 150889) \bmod 714025 \\ u_n &= (x_n + 1)/714025 \end{aligned}$$

The constants were chosen such that the parenthesized term in the expression for x_n does not exceed $2^{32} - 1$, allowing computation using only unsigned integers. The shift by 1 in the computation of u_n is necessary to avoid occurrence of the value 0. The generator is described in [4, p. 195] and referred to as the "Numerical Recipes" generator.

For some applications, it may be desirable to replicate results obtained with RESQ[6]. RESQuses a multiplicative generator with prime modulus, as described in [7, p. 227, 251] and [6, p. 75f]:

$$\begin{aligned} x_n &= 7^5 x_{n-1} \bmod (2^{31} - 1) = 16807x_{n-1} \bmod 2147483647 \\ u_n &= x_n/2147483647 \end{aligned}$$

The random number generator can be implemented entirely in 32-bit signed integer arithmetic by suitably modifying above algorithm [8, p. 214].

$$\begin{aligned} k &= \lfloor x_{n-1}/b \rfloor \\ x_n &= a(x_{n-1} \bmod b) - kc \\ u_n &= x_n \cdot 1/P \end{aligned}$$

If x_n is less than zero in the second step, add P. In the current implementation, $a = 16807$, $b = 127773$, $c = 2836$, $P = 2^{31} - 1$. The set of parameters $a = 40692$, $b = 52774$, $c = 3791$, $P = 2147483399$ is also possible. The integer implementation executes about five times faster than an output-equivalent floating point implementation, consuming on the order of 10 μs of CPU time on a VAX 6310. Performance improvements on machines without floating point hardware should be even more pronounced.

$2^{31} - 1$ is a prime number, and therefore it is guaranteed that $x_n \neq 0$, as required. The same generator is also employed by IMSL, the APL System and SIMPL/I [8, p. 212]. Another seed, 131,072, is also suggested (see [8, p. 226]).

The initial seed is determined by both the stream number (modulo 5) and the seed index specified through the call to `seed()`. The values are listed in Tables 2.5 and 2.6, which were obtained by collecting every one hundred thousandth sample from the random number generator, starting with the seed listed in the upper left hand corner. The values are to be read newspaper-column style, that is, adjacent entries in a row are six million samples apart. To provide for more replications, an additional 200 samples, spaced and ordered as before, were gathered and collected as sets 21 through 60, listed in Table 2.6. Streams 0 through 4 (modulo 5) use entries from the respective columns, e.g., stream 2, set 3 would start with the seed 400884188. By default, all streams use set 1.

For independent replications of an experiment, it is recommended to choose sets according to the run number. However, in this case, a single run may not exceed 100,000 random samples, as otherwise the next run would use see the end of the sample path of that run as its initial sample path.

The seeds used by RESQ can be readily duplicated. If set s_R from RESQ is desired, use stream and set from SIMUL according to the relationships:

Type	SIMUL stream	SIMUL set
Sources	0	s_R
Active queues	0	$s_R + 20$
Routing decisions	0	$s_R + 40$
Passive queues	1	s_R
Set nodes	1	$s_R + 20$

Two other generators of exponentially distributed random variates were investigated and found to be equivalent in terms of matching the moments of the theoretical distribution. The first, `GGEXN`, is part of the IMSL library of mathematical functions. The second, `SEXPO`, is described as Algorithm 599 in the *Collected Algorithms of the ACM* [9]. In the test, it was driven by the IMSL uniform random number generator, `GGUBFS`.

2.3 Computation of Sample Variance

$$\hat{\sigma}^2 = \frac{1}{n-1}\left[\sum_{i=1}^{N} x_i^2 - \frac{\left(\sum_{i=1}^{N} x_i\right)^2}{n}\right]$$

2.4 Numerical Computations

All standard numerical computations (SVD, least-squares fitting, ...) are performed with routines from [4]. Routines operating on vectors were modified to use vector and matrix descriptors rather than passing the array address and dimensions separately. This allows for better checking and more concise function invocations. Matrix manipulation routines (matrix multiplication, inversion, transpose) were added to the *Numerical Recipes* library.

	Stream (mod 5)				
Set	0	1	2	3	4
1	377003613	150006407	1607319776	880444232	1394235503
2	648473574	1633650593	1309449718	1664858565	502188969
3	1396717869	751601611	1467083827	1411092801	1840218403
4	2027350275	1410990605	286945040	532965288	1311275561
5	1356162430	1262214427	721343149	1722068622	283895623
6	1752629996	645360044	2116874573	1746569219	1167529898
7	645806097	1504645702	611952202	1400531143	1807656809
8	201331468	1063375004	586955910	1515855558	1957748143
9	1393552473	941885586	994660726	1609521801	1579477183
10	1966641861	1753135176	1293671801	1558217481	1077143577
11	711072531	253642018	1059442189	324388764	952106381
12	769795447	1701685042	1635715578	204169991	1971964782
13	1074543187	1448665492	2013644497	701383841	69303298
14	1933483444	1034856864	89569975	529261923	25167950
15	625102656	428280431	1640847845	1939917326	595933115
16	1116874679	259758456	1959091614	1082615488	1543655512
17	1442211901	600732272	1626357251	590057105	2145019186
18	989455196	704726097	612359260	890888446	1625181081
19	1996695068	398944698	590317743	286118126	1534372830
20	1850124212	114386769	712899670	510537184	2126692676

Table 2.5: Seeds for Random Number Streams

	Stream (mod 5)				
Set	0	1	2	3	4
21	1267310126	288727775	150940551	607706381	1283649514
22	1741371275	1499601820	642753845	1402948160	849817272
23	886499692	2136214308	1023651841	653358986	16270566
24	1014119573	1197972807	733927801	608814417	1578039323
25	933913228	1888007825	833901440	810995036	154844069
26	2082204497	686553263	446361094	134680510	162835168
27	920168983	747119178	1977130419	194538709	1333215125
28	1079618777	154337000	1567767826	325213425	1533246242
29	1888797415	136758808	592378857	129868077	1480452100
30	1002901030	9182540	1078081430	2066156210	1102295945
31	1582733583	303111010	1861270919	191003277	1961023568
32	254293472	154232008	1523794436	1044815317	523891923
33	1095895189	921093990	1018399670	480787176	1110586555
34	219529399	1684263351	1150780213	1777053489	228239308
35	1706847402	1166344707	1331847403	1456328099	2013541127
36	1951007719	1167753617	1789804243	793774834	1672985180
37	1169002398	1374693082	825622320	2027890820	1063656330
38	1482199345	1812641667	1256914017	1456964824	1102857485
39	1976077334	502455872	1100922234	1893698389	1745731166
40	775245191	857532898	915440313	1012863273	1370011933
41	1976418161	253676790	114314084	886217887	1543209256
42	35067978	193260737	88629880	2043086365	383950479
43	400884188	1100810747	961724062	1439625194	1116142313
44	1895732964	323117192	1045424616	671290808	963130134
45	1904749580	54898333	1369891993	434454241	1488002623
46	1301700180	1625075820	219567801	1632195104	227656887
47	63685808	458400561	546264204	1014572672	1441290500
48	936615625	190143079	1426893033	1041716927	1756174720
49	110322717	1161033978	165322447	950958659	1294608159
50	1029730003	1550885996	1014255177	1574664477	674700901
51	251900732	590236475	1389802225	901126851	1097448981
52	725094089	135814162	1302467997	577358624	1595118099
53	828842333	947384063	768292438	1423377107	29529110
54	1471230052	2128837091	688473660	1703397528	379322926
55	1703522097	797986569	1724128983	144715914	429798183
56	1356420548	487126607	1207223749	515170782	1743241938
57	1670372925	603065500	1527050499	923799337	608313449
58	437765009	1461527675	1593137863	426925463	889758719
59	39279049	427719601	1304425021	807287867	811561727
60	2123613511	1513298893	1210324428	445907939	631275438

Table 2.6: Seeds for Random Number Streams, continued

2.5 Portability

SIMUL is written in ANSI-C and intended to be portable across a wide range of operating systems and architectures. It assumes that `long` variables are at least 32 bits, `int` variables at least 16 bits long. Machine-dependent sections are enclosed within `#ifdef`/`#endif` pairs. Currently, only the routines `getsymbol`, `getcputype`, `opentext` are heavily operating system dependent, while `rv` and `simul` contain some trivial machine-dependent code.

2.6 Significant program modifications

09-Mar-1989 The bounds of the confidence interval is displayed, rather than its half-width. (`show_tallies()`)

10-Mar-1989 `parameter()` warns if values greater than 10^{10} or less than 10^{-6} are assigned, as these are most likely due to passing integer rather than floating-point arguments. (`parameter()`)

10-Mar-1989 Use unbiased estimator for sample variance. (`show_tallies()`)

19-Aug-1989 Change spectral method computation of confidence intervals to double precision, using `nrd` *Numerical Recipes* library.

20-Aug-1989 `Create` primitive renamed to `create_`.

20-Aug-1989 `fifo_server` and related procedures.

15-Sep-1989 `EntityType` redefined as pointer.

11-May-1990 transient recording added.

18-Aug-1990 regenerative method fix: now uses unbiased estimate for sample variance

20-Aug-1990 ported to GNU C on VAX DECstations; bit field options changed to text arguments; consistent sequential numbering of seeds

13-Sep-1990 added handler for SIGTERM (kill) and SIGINT (terminal interrupt)

2.7 Known Bugs and Restrictions

If a simulation is interrupted while executing `tally`, the last observation may not have been completely processed. After premature termination, no experiment tallies are written to the log file.

Chapter 3

Examples

3.1 M/M/1 Queue: Source

```
/*
MM1
===

Test example for SIMUL: a simple M/M/1 queue.

Copyright (c) 1989-1990 by H. Schulzrinne, University of Massachusetts
Last modified: 20-Aug-1990
*/

#include <simul.h>
#undef TRACE

enum {Lambda=1,Mu,D,Transient};
enum {Systime,Insys,St,Iat,Late};

/*****************
***            ***
***   QUEUE1   ***
***            ***
******************
First stop after customer creation: enter server 1.
Check if queue is empty to start another regeneration interval.
*/
void queue1(EntityType e)
{
  double service_time;

  /* regeneration point */
  if (in_system(1) == 0) {
    cycle(Systime); cycle(Insys); cycle(St); cycle(Iat); cycle(Late);
  }
  tally(Iat,tnow());
```

```
  enter_fifo(1,e,service_time = tally(St,ex(2,1)));
#ifdef TRACE
  fprintf(logfile,"+ %6d %12.4f (%10.4f)\n",id(e),tnow(),service_time);
#endif
}


/***************
***          ***
***   SINK   ***
***          ***
****************
Departure handler for server.
*/
void sink(queue_id n,EntityType e)
{
  double delay = tnow() - creation_time(e);

  tally(Insys,(double)in_system(1));
  tally(Late,(double)(delay > getp(D,1)));
  tally(Systime,delay);
#ifdef TRACE
  fprintf(logfile,"- %6d %12.4f\n",id(e),tnow());
#endif
  dispose(e);
}

/********************
***              ***
***   EXPERIMENT   ***
***              ***
********************/
void experiment(void)
{
  project("MM1: M/M/1 queue","H. Schulzrinne","mm1.log");

  parameter(Lambda,   "Mean interarrival time",1,1/0.8);
  parameter(Mu,       "Mean service time",     1,1.00);
  parameter(D,        "End-to-end deadline",   1,11.5); /* 0.10 loss */

  seed(0,1);  /* 0: interarrival (set 1) */
  seed(1,1);  /* 1: service time (set 1) */

  replicate(1,0.0,1e10,"");

/*                              initial limit   CI method */
#define TO "start=100 spectral regen "
  tallies(Systime,"System time", TO "stop=100000");
  tallies(Insys,  "In System",   TO);
  tallies(St,     "Service time",TO);
```

```
  tallies(Iat,    "Interarrival",TO "between");
  tallies(Late,   "Late",        TO);
  CI_check(0.2,10000);

  setup_server(1,sink);
}


/****************
***           ***
***   PRIME   ***
***           ***
*****************
create -> queue1 -> [server1] -> sink
*/
void prime(void)
{
  /* exponential interarrival with parameter set (1,1), no limit */
  create(ex,1,0, /* interarrival time (par 1, stream 0) */
         0, 1,1, /* fixed batch size of one */
         -1, /* no batch generation limit */
         queue1); /* first destination */
}
```

Table 3.1 traces the first thirty packets in a simulation of an M/M/1 queue with $\lambda = \rho = 0.8$. The first stream of random variates is used for interarrival times, the second for service times.

3.2 M/M/1 Example: Run-length control

Below, the screen output and simulation summary for the M/M/1 queue example, with run-length control, is reproduced. The nominal values are 5 for expected system time, 4 for expected number in system, 1 for expected service time, 1.25 for expected interarrival time and 0.10 for the fraction of late packets.

```
Run   1 of   1
CI check  0 @    9999: spectral= 0.11 regenerative= 0.10
CI check  1 @    9999: spectral= 0.12 regenerative= 0.12
CI check  2 @    9999: spectral= 0.02 regenerative= 0.02
CI check  3 @   10000: spectral= 0.02 regenerative= 0.02
CI check  4 @    9999: spectral= 0.40 regenerative= 0.32
CI check  0 @   14998: spectral= 0.09 regenerative= 0.08
CI check  1 @   14998: spectral= 0.09 regenerative= 0.09
CI check  2 @   14999: spectral= 0.02 regenerative= 0.01
CI check  3 @   15000: spectral= 0.02 regenerative= 0.01
CI check  4 @   14998: spectral= 0.27 regenerative= 0.25
CI check  0 @   22496: spectral= 0.07 regenerative= 0.07
CI check  1 @   22496: spectral= 0.07 regenerative= 0.08
CI check  2 @   22499: spectral= 0.01 regenerative= 0.01
CI check  3 @   22500: spectral= 0.02 regenerative= 0.01
CI check  4 @   22496: spectral= 0.20 regenerative= 0.21
```

event	id	clock	service time
+	1	0.0000	5.1971
+	2	0.7038	0.0160
+	3	1.9265	0.7082
+	4	2.8714	1.0219
+	5	3.0923	2.9659
+	6	4.2615	0.1976
−	1	5.1971	
−	2	5.2132	
+	7	5.5910	0.1660
−	3	5.9213	
−	4	6.9432	
+	8	7.1598	4.1051
+	9	7.1628	2.0591
+	10	8.6748	0.0320
+	11	8.9303	0.3267
+	12	9.1989	0.3412
+	13	9.3018	0.0923
+	14	9.6413	2.4934
−	5	9.9091	
−	6	10.1067	
−	7	10.2727	
+	15	10.3267	0.2416
+	16	11.0323	0.2365
+	17	11.4886	0.4069
+	18	12.0083	1.3706
−	8	14.3778	
−	9	16.4369	
−	10	16.4689	
−	11	16.7956	
−	12	17.1368	
−	13	17.2291	
+	19	17.2456	1.8495
+	20	17.8525	0.1210
−	14	19.7225	
−	15	19.9641	
−	16	20.2006	
−	17	20.6075	
+	21	21.7150	6.6382
−	18	21.9781	
+	22	22.0790	4.9846
+	23	22.2534	6.2791
+	24	23.6407	0.6620
−	19	23.8277	
−	20	23.9487	
+	25	25.7880	0.0330
+	26	25.8886	0.3996
+	27	25.9611	0.1831
+	28	29.0461	5.6214
+	29	30.2044	0.1797
−	21	30.5869	
+	30	30.9781	0.3641

Table 3.1: Trace for first packets in M/M/1 queue with $\lambda = \rho = 0.8$

```
CI check  0 @   33739: spectral= 0.04 regenerative= 0.06
CI check  1 @   33739: spectral= 0.05 regenerative= 0.06
CI check  2 @   33749: spectral= 0.00 regenerative= 0.00
CI check  3 @   33750: spectral= 0.01 regenerative= 0.01
CI check  4 @   33739: spectral= 0.16 regenerative= 0.18
1 loose entity found, 17 total.

SIMUL SUMMARY REPORT
====================

Project:  MM1: M/M/1 queue
Author:   H. Schulzrinne
Date:     Mon Aug 20 16:02:54 1990
CPU type: gaia.cs.umass.edu
Runs:     1

Stream  0: seed  1 =  377003613
Stream  1: seed  1 =  150006407
Confidence interval halfwidth <= 0.20
Random number generator: RESQ.


PARAMETERS
----------

 #  Description              Values
----------------------------------------------------------------------
 1  Mean interarrival time    1.25
 2  Mean service time         1
 3  End-to-end deadline       11.5
*** RUN   1 of   1: ***
Run terminated: all confidence intervals within tolerance.
Mon Aug 20 16:02:54 1990 (0.0000e+00) to Mon Aug 20 16:03:01 1990 (4.2477e+04)
CPU seconds:        6.9 (out of       7.0 wallclock seconds =  98% utilization)


TALLIES
-------

           |           0           1           2           3           4
           | System time   In System Service tim Interarriva        Late
------------------------------------------------------------------------
Samples    |       33639       33639       33649       33650       33639
Transient  |         100         100         100         100         100
Minimum    |   1.665e-04   0.000e+00   6.231e-05   6.065e-05   0.000e+00
Maximum    |   3.223e+01   2.600e+01   1.061e+01   1.382e+01   1.000e+00
Average    |   4.695e+00   3.682e+00   1.001e+00   1.259e+00   8.151e-02
0.90 SCI +-|   1.926e-01   1.667e-01   8.658e-03   1.591e-02   1.317e-02
0.90 SCI % |   4.102e+00   4.529e+00   8.650e-01   1.264e+00   1.616e+01
0.90 SCI > |   4.503e+00   3.515e+00   9.922e-01   1.243e+00   6.834e-02
0.90 SCI < |   4.888e+00   3.849e+00   1.009e+00   1.275e+00   9.468e-02
```

```
Reg. cycl. |        7010        7010        7010        7010        7010
Reg. est.  |   4.705e+00   3.689e+00   1.001e+00   1.259e+00   8.213e-02
0.90 RCI +-|   2.644e-01   2.287e-01   9.203e-03   1.203e-02   1.505e-02
0.90 RCI % |   5.620e+00   6.198e+00   9.195e-01   9.560e-01   1.832e+01
0.90 RCI > |   4.440e+00   3.460e+00   9.917e-01   1.247e+00   6.708e-02
0.90 RCI < |   4.969e+00   3.918e+00   1.010e+00   1.271e+00   9.718e-02
```

3.3 M/M/1 Example: 1000 Customers

Simulation summary for M/M/1 queue example, 1000 customers:

```
SIMUL SUMMARY REPORT
====================

Project:  MM1: M/M/1 queue
Author:   H. Schulzrinne
Date:     Mon Aug 20 16:52:34 1990
CPU type: gaia.cs.umass.edu
Runs:     1

Stream  0: seed  1 =  377003613
Stream  1: seed  1 =  150006407
Confidence interval halfwidth <= 0.20
Random number generator: RESQ.


PARAMETERS
----------

 # Description              Values
----------------------------------------------------------------------
 1 Mean interarrival time    1.25
 2 Mean service time         1
 3 End-to-end deadline       11.5
*** RUN   1 of   1: ***
Run terminated: Tally 0 reached limit 1e+03.
Mon Aug 20 16:52:34 1990 (0.0000e+00) to Mon Aug 20 16:52:34 1990 (1.2657e+03)
CPU seconds:        0.2 (out of       0.0 wallclock seconds = 100% utilization)


TALLIES
-------

           |           0           1           2           3           4
           | System time   In System Service tim Interarriva        Late
------------------------------------------------------------------------
Samples    |         900         900         915         915         900
Transient  |         100         100         100         100         100
Minimum    |   4.287e-03   0.000e+00   1.131e-03   1.012e-03   0.000e+00
```

```
Maximum     |  1.854e+01   1.600e+01   8.000e+00   1.047e+01   1.000e+00
Average     |  4.603e+00   3.596e+00   1.033e+00   1.263e+00   5.889e-02
0.90 SCI +-|   9.530e-01   9.669e-01   4.384e-02   7.912e-02   7.147e-02
0.90 SCI % |   2.070e+01   2.689e+01   4.244e+00   6.267e+00   1.214e+02
0.90 SCI > |   3.650e+00   2.629e+00   9.893e-01   1.183e+00  -1.258e-02
0.90 SCI < |   5.556e+00   4.562e+00   1.077e+00   1.342e+00   1.304e-01
Reg. cycl. |         172         172         172         172         172
Reg. est.  |   4.947e+00   3.805e+00   1.035e+00   1.266e+00   8.951e-02
0.90 RCI +-|   1.061e+00   9.427e-01   4.596e-02   6.851e-02   5.512e-02
0.90 RCI % |   2.145e+01   2.478e+01   4.441e+00   5.412e+00   6.158e+01
0.90 RCI > |   3.886e+00   2.862e+00   9.891e-01   1.197e+00   3.439e-02
0.90 RCI < |   6.008e+00   4.747e+00   1.081e+00   1.334e+00   1.446e-01
```

3.4 M/M/1 Example: 100,000 Customers

Simulation summary for M/M/1 queue example, 100,000 customers:

```
SIMUL SUMMARY REPORT
====================

Project:  MM1: M/M/1 queue
Author:   H. Schulzrinne
Date:     Mon Aug 20 16:56:21 1990
CPU type: gaia.cs.umass.edu
Runs:     1

Stream  0: seed  1 =  377003613
Stream  1: seed  1 =  150006407
Random number generator: RESQ.


PARAMETERS
----------

 #  Description              Values
----------------------------------------------------------------------
 1  Mean interarrival time    1.25
 2  Mean service time         1
 3  End-to-end deadline       11.5
*** RUN   1 of   1: ***
Run terminated: Tally 0 reached limit 1e+05.
Mon Aug 20 16:56:21 1990 (0.0000e+00) to Mon Aug 20 16:56:37 1990 (1.2516e+05)
CPU seconds:       16.1 (out of       16.0 wallclock seconds = 101% utilization)


TALLIES
-------

            |          0           1           2           3           4
```

	System time	In System	Service tim	Interarriva	Late
Samples	99900	99900	99900	99900	99900
Transient	100	100	100	100	100
Minimum	9.728e-05	0.000e+00	3.096e-05	4.807e-05	0.000e+00
Maximum	4.025e+01	3.700e+01	1.061e+01	1.382e+01	1.000e+00
Average	4.880e+00	3.880e+00	1.000e+00	1.252e+00	9.174e-02
0.90 SCI +-	2.352e-01	2.132e-01	3.761e-03	8.316e-03	1.273e-02
0.90 SCI %	4.820e+00	5.495e+00	3.760e-01	6.644e-01	1.387e+01
0.90 SCI >	4.645e+00	3.667e+00	9.966e-01	1.243e+00	7.901e-02
0.90 SCI <	5.115e+00	4.093e+00	1.004e+00	1.260e+00	1.045e-01
Reg. cycl.	19992	19992	19992	19992	19992
Reg. est.	4.884e+00	3.883e+00	1.000e+00	1.252e+00	9.199e-02
0.90 RCI +-	1.954e-01	1.714e-01	5.140e-03	7.078e-03	1.060e-02
0.90 RCI %	4.001e+00	4.414e+00	5.138e-01	5.655e-01	1.152e+01
0.90 RCI >	4.688e+00	3.712e+00	9.953e-01	1.244e+00	8.139e-02
0.90 RCI <	5.079e+00	4.054e+00	1.006e+00	1.259e+00	1.026e-01

3.5 Execution Time Comparison

In Tab. 3.2, we present a comparison of the execution times for the last example, namely an $M/M/1$ queue with 100,000 customers. In all cases, the simulation commanded upwards of 60% of the CPU.

CPU type	node tested	Op. System	CPU seconds
DECstation 5000	gaia.cs.umass.edu	Ultrix 4.0	16.1
DECstation 3100	juno.cs.umass.edu	Ultrix 2.2 (15)	28.6
DEC VAX 8600	alpha.ecs.umass.edu	VMS 5.3	64.6
DEC VAX 6310	sigma.ecs.umass.edu	VMS 5.3	73.8
DEC VAX 6310	kappa.ecs.umass.edu	VMS 5.3	74.1
DEC VAXserver 3500	elsrv1.cs.umass.edu	VMS 5.3	112.7
DEC VAX 780	beta.ecs.umass.edu	VMS 5.3	195.9
DEC VAX 780	gamma.ecs.umass.edu	VMS 5.3	274.9
DEC Microvax II	merlin.cs.umass.edu	VMS 5.1	287.7
DEC VAXstation 2000	el01.cs.umass.edu	VMS 5.3	292.3
DEC VAX 780 (no FPA)	delta.ecs.umass.edu	VMS 5.3	369.1
Compaq 386/20		MS-DOS 3.0	
AT-clone (286, 12 MHz)		MS-DOS 3.3	

Table 3.2: Comparison of CPU time needed to simulate an $M/M/1$ queue with 100,000 customers

Bibliography

[1] C. D. Pegden, *Introduction to SIMAN.* State College, Pennsylvania: Systems Modeling Corporation, 1987.

[2] P. Heidelberger and P. D. Welch, "A spectral method for confidence interval generation and run length control in simulations," *Communications of the ACM*, vol. 24, pp. 233–245, Apr. 1981.

[3] K. Pawlikowski, "Steady-state simulation of queueing processes: a survey of problems and solutions," *ACM Computing Surveys*, vol. 22, pp. 123–170, June 1990.

[4] W. H. Press, B. P. Flannery, S. A. Teukolsky, and W. T. Vetterling, *Numerical Recipes - The Art of Scientific Computing.* Cambridge, England: Cambridge University Press, 1986.

[5] D. R. Brillinger, *Time Series: Data Analysis and Theory.* New York: Holt, Rinehart and Winston, 1975.

[6] C. H. Sauer, A. M. Bloom, P. G. Loewner, E. A. MacNair, and J. F. Kurose, "The research queueing package version 2: CMS reference manual," Tech. Rep. RA139, International Business Machines Corporation, Yorktown Heights, New York, Nov. 1986.

[7] S. S. Lavenberg, ed., *Computer Performance Modeling Handbook.* Notes and Reports in Computer Science and Applied Mathematics, New York: Academic Press, 1983.

[8] P. Bratley, B. L. Fox, and L. E. Schrage, *A Guide to Simulation.* New York: Springer-Verlag, 2nd ed., 1987.

[9] J. Ahrens and U. Dieter, "Computer methods for sampling from the exponential and normal distribution," *Communications of the ACM*, vol. 15, pp. 873–882, Oct. 1972.