Performance Modeling of Computer Systems and Networks

Prof. Vittoria de Nitto Personè

Next Event Simulation
Examples

Università degli studi di Roma Tor Vergata
Department of Civil Engineering and Computer Science Engineering

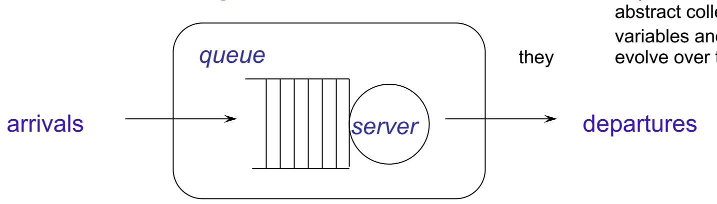
1

1. **Initialize** - set simulation clock and first time of occurrence for each event type
2. **Process current event** - scan event list to determine most imminent event; advance simulation clock; update state
3. **Schedule new events** - new events (if any) are placed in the event list
4. **Terminate** - Continue advancing the clock and handling events until termination condition is satisfied

2

Single Server Queue

- Conceptual model: abstract collection of variables and how they evolve over time



- The *state* is number of jobs in the node at time t: $l(t)$
- Its time-*evolution* is guided by arrival-departure events:
  - An arrival causes $l(t)$ to increase by 1
  - A departure causes $l(t)$ to decrease by 1

3

Single Server Queue

- Specification model: collection of mathematical variables together with logic and equations

The state variable $l(t)$ provides a complete characterization of the state of a ssq

$$l(t) = 0 \Leftrightarrow q(t) = 0 \text{ and } x(t) = 0$$
$$l(t) > 0 \Leftrightarrow q(t) = l(t)\text{-}1 \text{ and } x(t) = 1$$

Da l(t) posso effettivamente definire tutto.
Tutto ciò che è inutile/superfluo non va messo.
Posso scrivere queste formule solo se capisco bene il modello che sto definendo, se non me ne rendo conto allo 'step' prima posso tornare indietro e vedere cosa manca, ma idealmente a questo passo ci arrivo solo dopo aver catturato tutto.

4

l(t) ci dice tutto, non ci serve altro.

- The initial state $l(0)$ can have any non-negative value, typically 0 Ha senso partire da 0, se parto dal sistema vuoto.
  Se partissi da un sistema già stabile, potrei usare altri valori.
- terminal state: any non-negative value
  - Assume at time τ arrival process stopped. Remaining jobs processed before termination
- some mechanism must be used to denote an event impossible
  - Only store possible events in event list
  - Denote impossible events with event time of ∞

se volessi tornare allo stato iniziale, fisso un tempo 'tau' che, superato blocca i nuovi arrivi e fa servire gli ultimi job arrivati in ritardo.

13/04/2023

- The simulation clock (current time) is t
- The terminal ("close the door") time is τ
- The next scheduled arrival time is t_a
- The next scheduled service completion time is t_c
- The number in the node (state variable) is l

Next-Event Simulation

Algorithm

1. **Initialize**: the clock
   the event list (e.g. ssq arrival)
   the system state
2. **Remove** next event from the list
3. **Advance** simulation clock
4. **Process** current event
5. **Schedule** new events (if any) generated from current event
6. Go to 2. until **termination** condition is satisfied

ssq2.c

```
int main(void)
{  long   index     = 0;       /* job index */
   double arrival   = START;   /* arrival time*/
   double delay;               /* delay in queue*/
   double service;              /* service time*/
   double wait;                 /* delay + service*/
   double departure = START;    /* departure time*/
   struct {                     /* sum of ...   */
       double delay;     /*delay times */
       double wait;      /*wait times*/
       double service;   /*service times */
       double interarrival; /*   interarrival times */
   } sum = {0.0, 0.0, 0.0};
PutSeed(123456789);
```

```
while (index < LAST) {
index++;
arrival      = GetArrival();
if (arrival < departure)
    delay  = departure - arrival; /* delay in queue  */
else  delay      = 0.0;            /* no delay  */
service = GetService();
wait  = delay + service;
departure     = arrival + wait;   /* time of departure */
sum.delay    += delay;
sum.wait     += wait;
sum.service += service;  }
```

…


```
l = 0; t = 0.0;
tc = ∞; ta = GetArrival();     /* initialize the event list */
while ((ta < τ ) or (l > 0)) {
        t = min(ta, tc);       /* scan the event list */
        if (t == ta) {         /* process an arrival */
                l++;
                ta = GetArrival();
                if (ta > τ )
                        ta = ∞;
                if (l == 1)
                tc = t + GetService();
        }
        else {                 /* process a completion */
                l − −;
                if (l > 0)
                        tc = t + GetService();
                else
                        tc = ∞;
        }
}
```

arrival

depature

2 event types: arrival, depature

Algorithm 1

Program ssq3

- `number` represents $l(t)$ (system state)
  `struct t` represents time
  - `t.arrival, t.completion` event list (t_a, t_c from algorithm 1)
  - `t.current` simulation clock (t from algorithm 1)
  - `t.next` next event time (min(t_a, t_c) from algorithm 1)
  - `t.last` last arrival time
- `struct area` (time-averaged) statistics-gathering structure
  - $\int_0^t l(s)ds$ evaluated as `area.node`
  - $\int_0^t q(s)ds$ evaluated as `area.queue`
  - $\int_0^t x(s)ds$ evaluated as `area.service`

ssq3.c

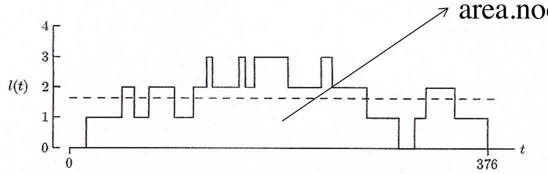
```
#include <stdio.h>
#include <math.h>
#include "rngs.h"    /* the multi-stream generator */
#define START   0.0
#define STOP    20000.0  /* terminal (close the door) time*/
#define INFINITY (100.0 * STOP) /* must be much larger than STOP */

double Min(double a, double c)
{  if (a < c)     return (a);
    else     return (c);}

double Exponential(double m) …
double Uniform(double a, double b) …
double GetArrival() …
double GetService() …
```

```
 int main(void)
{  struct {
        double arrival;     /* next arrival time */
        double completion; /* next completion time */
        double current; the clock! urrent time */
        double next;        /* next (most imminent) event time */
        double last;        /* last arrival time */
   } t;
   struct {
        double node;    /* time integrated number in the node */
        double queue;    /* time integrated number in the queue */
        double service; /* time integrated number in service */
   } area      = {0.0, 0.0, 0.0};
   long index  = 0;     /* used to count departed job */
   long number = 0;     /* number in the node */ system state
```



```
PlantSeeds(123456789);
t.current    = START;          /* set the clock */
t.arrival    = GetArrival();  /* schedule the first arrival */
t.completion = INFINITY; /* the first event can't be a completion */
while ((t.arrival < STOP) || (number > 0)) {
   t.next= Min(t.arrival, t.completion); /* next event time   */
   if (number > 0)  {          /* update integrals  */
        area.node+= (t.next - t.current) * number;
        area.queue+= (t.next - t.current) * (number - 1);
        area.service += (t.next - t.current);    }
   t.current = t.next;  advance the clock!
```

$$\text{area.node} = \int_0^t l(s)\,ds$$



```
if (t.current == t.arrival)  {
   number++;
   t.arrival= GetArrival();
   if (t.arrival > STOP)  {
     t.last= t.current;
     t.arrival    = INFINITY;
   }
   if (number == 1)
        t.completion = t.current + GetService();
}
else {
  index++;
   number--;
   if (number > 0)
        t.completion = t.current + GetService();
   else
        t.completion = INFINITY;
   }
}
```

process an arrival

process a completion


```
printf(" … jobs", index);
printf(" average interarrival time ..", t.last / index);
printf(" average wait …", area.node / index);
printf(" average delay ...", area.queue / index);
printf(" average service time ...", area.service / index);
printf(" average # in the node ... ", area.node / t.current);
printf(" average # in the queue .. ", area.queue / t.current);
printf(" utilization ....", area.service / t.current);
```

World Views and Synchronization

- ssq2 produces :

```
while (index < LAST) {
index++;
arrival      = GetArrival();
if (arrival < departure)
     delay  = departure - arrival;
else  delay      = 0.0;
service = GetService();
wait  = delay + service;
departure     = arrival + wait;
sum.delay    += delay;
sum.wait     += wait;
sum.service += service;  }
```

$$\sum_{i=1}^{n} d_i \qquad \sum_{i=1}^{n} w_i \qquad \sum_{i=1}^{n} s_i$$

World Views and Synchronization

- ssq2 produces :

$$\sum_{i=1}^{n} w_i$$

```
printf("… jobs", index);
printf("average interarrival time = ", sum.interarrival / index);
printf("average wait ........... = ", sum.wait / index);
printf("average delay .......... = ", sum.delay / index);
printf("average service time .... = ", sum.service / index);
printf("average # in the node ... = ", sum.wait / departure);
printf("average # in the queue .. = ", sum.delay / departure);
printf("utilization ............ = ", sum.service / departure);
```

$$\bar{w} = \frac{1}{n}\sum_{i=1}^{n} w_i$$

$$\bar{l} = \frac{n}{c_n}\bar{w}$$

$$\bar{q} = \frac{n}{c_n}\bar{d}$$

$$\bar{x} = \frac{n}{c_n}\bar{s}$$

$$\frac{\sum_{i=1}^{n} w_i}{c_n} = \frac{n\bar{w}}{c_n}$$

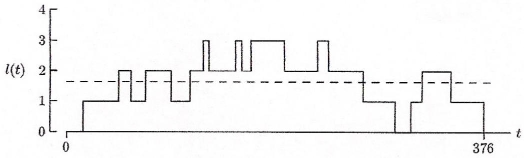
- ssq3 produces :

```
PlantSeeds(123456789);
t.current    = START;
t.arrival    = GetArrival();
t.completion = INFINITY;
while ((t.arrival < STOP) || (number > 0)) {
    t.next= Min(t.arrival, t.completion);
    if (number > 0)  {
        area.node+= (t.next - t.current) * number;
        area.queue+= (t.next - t.current) * (number - 1);
        area.service += (t.next - t.current);    }
    t.current        = t.next;
```

$\int_0^\tau l(t)dt$ ← area.node

$\int_0^\tau q(t)dt$ ← area.queue

$\int_0^\tau x(t)dt$ ← area.service



$$\tau \bar{l} = \int_0^\tau l(t)dt$$

```
printf(" … jobs", index);
printf(" average interarrival time ..", t.last / index);
printf(" average wait …", area.node / index);
printf(" average delay ...", area.queue / index);
printf(" average service time ...", area.service / index);
printf(" average # in the node ... ", area.node / t.current);
```

$$\bar{w} = \frac{\tau}{n}\bar{l}$$

$$\bar{l} = \frac{1}{\tau}\int_0^\tau l(t)dt$$

```
printf(" average # in the queue .. ", area.queue / t.current);
printf(" utilization ....", area.service / t.current);
```

World Views and Synchronization

- programs ssq2 and ssq3 simulate exactly the same system
- The two have different *world views*
  - ssq2 naturally produces job-averaged statistics (based upon *process-interaction*)
  - ssq3 naturally produces time-averaged statistics (based upon *event-scheduling*)

World Views and Synchronization

The programs should produce exactly the same statistics

- in ssq2 random variates are always generated in the alternating order:

$$a_1, s_1, a_2, s_2, \ldots$$

```
while (index < LAST) {
index++;
arrival      = GetArrival();
if (arrival < departure)
     delay  = departure - arrival;
else  delay      = 0.0;
service = GetService();
wait  = delay + service;
departure     = arrival + wait;
sum.delay    += delay;
sum.wait     += wait;
sum.service += service;  }
```

World Views and Synchronization

• in ssq3 the order cannot be known a priori

```
while ((t_a < τ ) or (l > 0)) {
        t = min(t_a, t_c);     /* scan the event list */
        if (t == t_a) {        /* process an arrival */
                l++;
                t_a = GetArrival();
                if (t_a > τ )
                        t_a = ∞;
                if (l == 1)
                t_c = t + GetService();
        }
        else {                 /* process a completion */
                l – –;
                if (l > 0)
                        t_c = t + GetService();
    .....
```


World Views and Synchronization

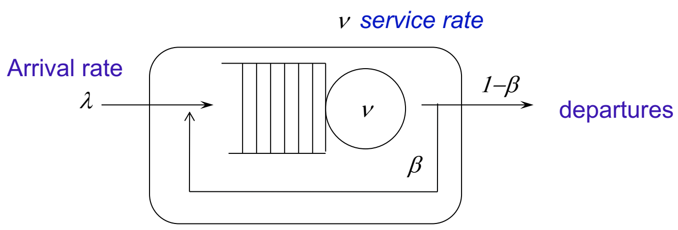
The programs should produce exactly the same statistics

→ to do so requires rngs

```
 double GetArrival()
{ static double arrival = START;
  SelectStream(0);
  arrival += Exponential(2.0);
  return (arrival);}

double GetService()
{ SelectStream(1);
  return (Uniform(0.0, 1.5)+Uniform(0.0, 1.5));}
```

Model Extensions

SSQ with immediate feedback



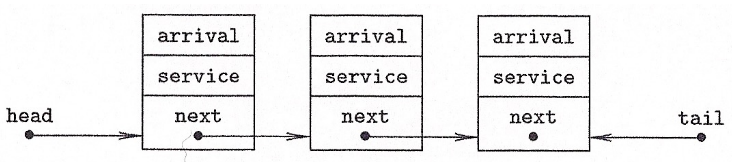
job index	1	2	3	4	5	.	6	.	7	8	.	9	…
Arrival/feedback	1	3	4	7	10	**13**	14	**15**	19	24	**26**	30	…
service	9	3	2	4	7	5	6	3	4	6	3	7	…
departure	10	**13**	**15**	19	**26**	31	37	**40**	44	50	53	**60**	…

Model Extensions

SSQ with immediate feedback

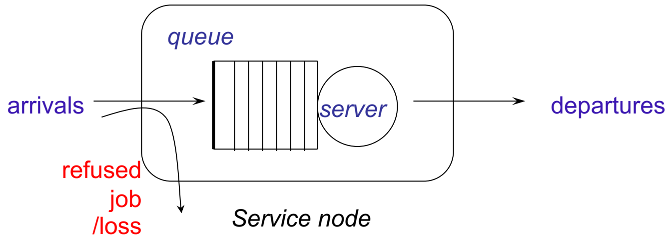
```
else {                    /* process a completion */
   if (GetFeedback() == 0) {   /* this statement is new */
         index++;
         number--;}
   if (number > 0)
         t.completion = t.current + GetService();
   else    t.completion = INFINITY;
```

- alternate queue disciplines
  - it is necessary to add a dynamic-queue data structure



- +2 supporting queue functions: Enqueue, Dequeue

Model Extensions



Assumption:

for the finite *capacity case*,
only the accepted jobs are considered

BUT
the *loss probability* could be of interest!

Model Extensions

SSQ with finite capacity

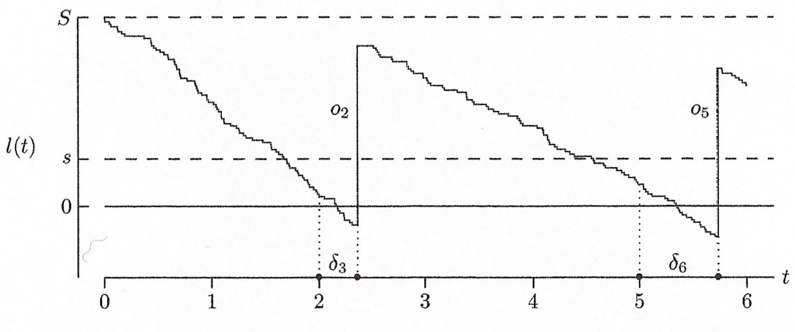
```
if (t.current == t.arrival) {          /* process an arrival */

   if (number < CAPACITY) {
      number++;
      if (number == 1)
         t.completion = t.current + GetService();
   }
   else
      reject++;
   t.arrival = GetArrival();
   if (t.arrival > STOP) {
      t.last = t.current;
      t.arrival = INFINITY;
   }
}
```

Random Sampling

- The structure of ssq3 facilitates adding sampling
- Add a sampling event to the event list
  - Sample deterministically, every δ time units
  - Sample Randomly, every *Exponential*(δ) time units

A Simple Inventory System with Delivery Lag

Two changes relative to sis2
- *Uniform*(0,1) lag between inventory review and order delivery
- More realistic demand model
  - Demand instances for a single item occur *at random*
  - Average rate is λ demand instances per time interval
  - Time between demand instances is *Exponential*($1/\lambda$)



Comparison of Demand Models

sis2: used an *aggregate* demand for each time interval, generated as an *Equilikely*(10,50) random variate

- Aggregate demand per time interval is random
- Within an interval, time between demand instances is constant
- Example: if aggregate demand is 25, inter-demand time is 1/25=0.04

- Now using *Exponential*($1/\lambda$) inter-demand times
  - Demand is modeled as an arrival process
  - Average demand per time interval is λ

Specification Level: States and Notation

- The simulation clock is t (real-valued)
- The terminal time is τ (integer-valued)
- Current inventory level is $l(t)$ (integer-valued)
- Amount of inventory on order, if any, is $o(t)$ (integer-valued)
  - Necessary due to delivery lag
- $l(t)$ and $o(t)$ provide complete state description
- Initial state is assumed to be $l(0)=\mathrm{S}$ and $o(0)=0$
- Terminal state is assumed to be $l(\tau)=\mathrm{S}$ and $o(\tau)=0$
- Cost to bring $l(\mathrm{t})$ to S at simulation end (with no lag) must be included in accumulated statistics

Specification Level: Events

Three types of events can change the system state

- A *demand* for an item at time t
  - $l(t)$ decreases by 1
- An inventory *review* at integer-valued time t
  - If $l(t) \geq s \rightarrow o(t)=0$
  - If $l(t) < s \rightarrow o(t)=S-l(t)$
- An *arrival* of an inventory replenishment order at time t
  - $l(t)$ increases by $o(t)$
  - $o(t)$ becomes 0

Algorithm 2: initialization

Time variables used for event list:

- t_d: next scheduled inventory *demand*
- t_r: next scheduled inventory *review*
- t_a: next scheduled inventory *arrival*

∞ denotes impossible events

```
l = S;                  /* initialize inventory level */
o = 0;                  /* initialize amount on order */
t = 0.0;                /* initialize simulation clock */
td = GetDemand();       /* initialize event list */
tr = t + 1.0;           /* initialize event list */
ta = ∞;                 /* initialize event list */
```

Algorithm 2: main loop

3 types of events:
demand, review, arrival

```
while (t < τ) {
        t = min(t_d , t_r , t_a);      /* scan the event list */
        if (t == t_d) {                /* process an inventory demand */
            l--;                       demand
            t_d = GetDemand();
        }
        else if (t == t_r ) {          /* process an inventory review */
            if (l < s) {               review
                o = S - l;
                δ= GetLag();
                t_a = t + δ ;
            }
            t_r += 1.0;
        }
        else {                         /* process an inventory arrival */
            l += o;                    arrival
            o = 0;
            t_a = ∞;
        }
}
```


Program sis3

- implements algorithm 2

correspond to t_d, t_r, t_a

```
while (t.current < STOP) {
  t.next = Min(t.demand, t.review, t.arrive);
  if (inventory > 0)
   sum.holding  += (t.next - t.current) * inventory;
  else
    sum.shortage -= (t.next - t.current) * inventory;
  t.current = t.next;
  if (t.current == t.demand) {
    sum.demand++; /* process an inventory demand */
    inventory--;
    t.demand = GetDemand();    }
  else    ………
```

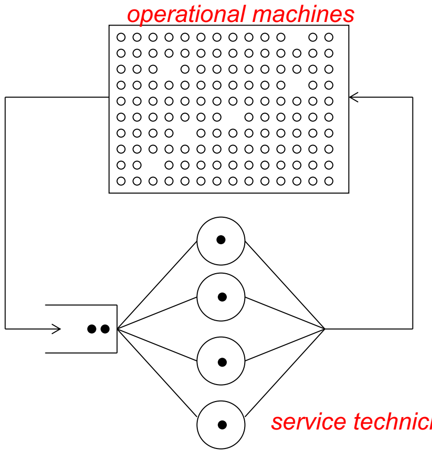
Program sis3

- State variables `inventory` and `order` correspond to $l(t)$ and $o(t)$
    - `t.next` next event instant (min(t_d , t_r , t_a) in algorithm 2)
    - `t.last` last arrival instant

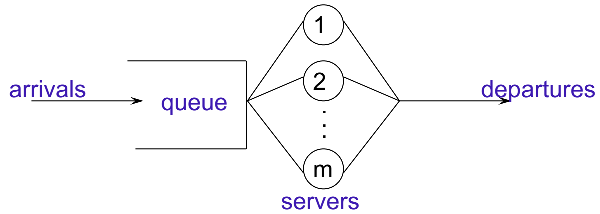
`sum.hold` and `sum.short` accumulate the time-integrated holding and shortage integrals

Machine Shop Model

1. Goals
2. Conceptual model
3. Specification model
4. Computational model
5. Verify
6. Validate



Conceptual model: MSQ



Servers in a multi-server service node are called *service channels*

- m is the number of servers
- The *server index* is $s = 1, 2, \ldots, \mathrm{m}$

The *state* includes:

the number of jobs in the node at time t: $l(t)$
the state of each server at at time t: $x_s(\mathrm{t})$

Specification Model: States and Notation

$l(t)$ denotes the number of jobs in the service node at time t

- If $l(t) \geq m$, all servers are busy and $q(t) = l(t) - m$
- If $l(t) < m$, some servers are idle
- If servers are distinct, need to know which servers are idle

For *s* = 1, 2, . . . , *m* define

$x_s(\mathrm{t})$: the number of jobs in service (0 or 1) at server *s* at time t

The complete state description is $l(t), x_1(t), x_2(t), \ldots, x_m(t)$

$$q(t) = l(t) - \sum_{s=1}^{m} x_s(t)$$

Specification Model: Events

What types of events can change state variables
$l(t), x_1(t), x_2(t), \ldots, x_m(t)$?

- *arrival* at time t
  - $l(t)$ increases by 1
  - If $l(t) < m$, an idle server s is selected, and $x_s(t)$ becomes 1
    else all servers are busy

- A *completion of service* by server s at time t
  - $l(t)$ decreases by 1
  - if $l(t) \geq m$, a job is selected from the queue to enter service
    else $x_s(t)$ becomes 0

→ m+1 event types

Specification Model: Additional Assumptions

- The <u>initial</u> <u>state</u> is an empty node
  - $l(0) = 0$
  - $x_1(0)=x_2(0)= \ldots = x_m(0)=0$
  - The <u>first</u> <u>event</u> must be an arrival

- The arrival process is turned off at time τ
  - The node continues operation after time τ until empty
  - The <u>terminal state</u> is an empty node
  - The <u>last event</u> is a completion of service

For simplicity, all servers are independent and *statistically identical*

- *Equity* selection is the server selection rule (lowest-utilized)

All of these assumptions can be relaxed

Event list

1 on, 0 off
1 busy, 0 idle

	0	t	x	arrival
	1	t	x	completion by 1
	2	t	x	completion by 2
m=4	3	t	x	completion by 3
	4	t	x	completion by 4

- can be organized as an array of m + 1 event types
- field t: scheduled time of next occurrence for the event
- field x: current *activity status* of the event

- for large m $\longrightarrow$ alternate event-list structures

Program msq

Implements this next-event multi-server service node simulation model

- `number` state variable $l(t)$
- state variables $x_1(t), x_2(t), \ldots, x_m(t)$ are part of the event list
- `area` time-integrated statistic $\int_0^t l(\theta)d\theta$
- `sum` array, records for each server
  - the sum of service times
  - the number served
- function `NextEvent` searches the event list to find the next event
- function `FindOne` searches the event list to find the longest-idle server (because equity selection is used)

program msq.c

```
typedef struct {
  double t;
  int    x;
} event_list[SERVERS + 1];
…
    int NextEvent(event_list event)
    { int e;
      int i = 0;
      while (event[i].x == 0)
        i++;
      e = i;
      while (i < SERVERS) {
        i++;
        if ((event[i].x == 1) && (event[i].t < event[e].t))
           e = i;  }
      return (e);}
```

is the first active event, assume it is the next

look for the next active event

if it is previous, update e

programma msq.c

```
 int FindOne(event_list event)
{ int s;
  int i = 1;
  while (event[i].x == 1)
    i++;
  s = i;
  while (i < SERVERS) {
     i++;
     if ((event[i].x == 0) && (event[i].t < event[s].t))
     s = i;  }
  return (s);}
```

first server idle

look for the next idle

if its completion is previous,
it is idle since more time

Exercises

- 5.1.1, 5.1.2, 5.1.3
- 5.2.1, 5.2.2,
- 5.2.8: modify program msq to allow for a finite capacity (max r jobs); a. draw a histogram of the time between lost jobs at the node; b. comment on the shape of this histogram.