Performance Modeling of Computer Systems and Networks

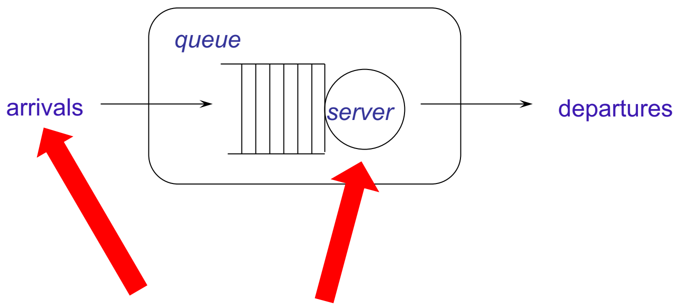
Prof. Vittoria de Nitto Personè

Multi-stream application examples

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

1

Single Server Queue



1. one stream per stochastic component

2. uncoupled processes → no-overlapped streams

2

P. 110 discrete

ssq2 revisited

- Use `rngs` functions for arrivals and services

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

double GetService(void) {
SelectStream(2);
return (Uniform(1.0, 2.0));
}
```

posso mettere da ø a 255

inizializza tutti i 256 flussi

- include "`rngs.h`" and use PlantSeeds(12345) (in place of PutSeed(12345))

arrival and service processes are *uncoupled*

stream 0 for arrivals, stream 1 for services

for 10025 jobs
average interarrival time = 1.99
average wait .... = 3.92
average delay .... = 2.41
average service time .... = 1.50
average # in the node ... = 1.96
average # in the queue .. = 1.21
utilization .... = 0.75

stream 0 for arrivals, stream 2 for services (or e.g. stream 128 to get more separation)

for 10025 jobs
average interarrival time = 1.99
average wait .... = 3.86
average delay .... = 2.36
average service time .... = 1.50
average # in the node ... = 1.93
average # in the queue .. = 1.18
utilization .... = 0.75

Theoretical values

$\bar{r}$	$\bar{w}$	$\bar{d}$	$\bar{s}$	$\bar{l}$	$\bar{q}$	$\bar{x}$
2.00	3.83	2.33	1.50	1.92	1.17	0.75

P63 slide, P111 discrete

Uncoupling Stochastic Processes

Consider changing the service process to
Uniform(0.0, 1.5) + *Uniform*(0.0, 1.5)

· stessa media
· diversa varianza

- Without uncoupling, arrival process sequence would change!
- With uncoupling, the service process "sees" exactly the same arrival sequence
- Important variance reduction technique

Theoretical values

$\bar{r}$	$\bar{w}$	$\bar{d}$	$\bar{s}$	$\bar{l}$	$\bar{q}$	$\bar{x}$
2.00	3.83	2.33	1.50	1.92	1.17	0.75

stream 0 for arrivals, stream 1 for services

for 10025 jobs
average interarrival time = 1.99
average wait .... = 4.29
average delay .... = 2.78
average service time .... = 1.51
average # in the node ... = 2.15
average # in the queue .. = 1.40
utilization .... = 0.76

diverso a causa della variabilità!

for 10025 jobs
average interarrival time = 1.99
average wait .... = 3.92
average delay .... = 2.41
average service time .... = 1.50
average # in the node ... = 1.96
average # in the queue .. = 1.21
utilization .... = 0.75

ssq with Multiple Job Types

- Consider multiple job types, each with its own arrival and service process
- Two job types:

stessa media = 2

  - –Class 0: *Exponential*(4.0) interarrivals, *Uniform*(1.0, 3.0) service
  - –Classe 1: *Exponential*(6.0) interarrivals, *Uniform*(0.0, 4.0) service
- Use `rngs` to allocate a different stream to each stochastic process

Arrival process

```
double GetArrival(int *j)          /* j corrisponds to job type */
{
const double mean[2] = {4.0, 6.0};
static double arrival[2] = {START, START};
static int init = 1;
double temp;
if (init) {
        SelectStream(0);
        arrival[0] += Exponential(mean[0]);
        SelectStream(1);
        arrival[1] += Exponential(mean[1]);
        init = 0;
        }
if (arrival[0] <= arrival[1])
        *j = 0;
else
        *j = 1;
temp = arrival[*j];
SelectStream(*j);
arrival[*j] += Exponential(mean[*j]);
return (temp);
}
```

The first arrival is of class 0!

ritorna prossimo arrival time e job type con indice Ø o 1.

- streams 0 and 1 are used for interarrival times of class 0 and class 1 jobs respectively

Multi-stream RNG
Case study ssq

example 3.2.9

Service process

```
double GetService(int j)   classe ∅ o 1
{
  const double min[2] = {1.0, 0.0};
  const double max[2] = {3.0, 4.0};
  SelectStream(j + 2);   stream '2' o '3'
  return (Uniform(min[j], max[j]));
}
```

- *j* corrisponds to the job type (0 or 1)
- streams 2 and 3 are used for service times of class 0 and 1 respectively
- All four simulated stochastic processes are uncoupled!
- Any process could be changed without altering the random sequence of others!

Prof. Vittoria de Nitto Personè 9

9

65 slide, 113 discrete

Multi-stream RNG
Case study ssq

Consistency checks

(se amplio un modello, e' importante verificarne la consistenza!)

- The "teorethical" steady-state statistics are

$\bar{r}$	$\bar{w}$	$\bar{d}$	$\bar{s}$	$\bar{l}$	$\bar{q}$	$\bar{x}$
2.40	7.92	5.92	2.00	3.30	2.47	0.83

exact analytical results,
No simulation!

- obvious consistency checks: $\bar{w} = \bar{d} + \bar{s}$ $\quad \bar{l} = \bar{q} + \bar{x}$
- other consistency checks:
  - both job types have avg service time of 2.0 → $\bar{s} = 2.00$
  - arrival rate should be 1/4 + 1/6 = 5/12 → $\bar{r} = 12/5 = 2.40$
  - $\bar{x}$ should be ratio of arrival to service rates

$$\frac{5/12}{1/2} = 5/6 \cong 0.83$$

Prof. Vittoria de Nitto Personè 10

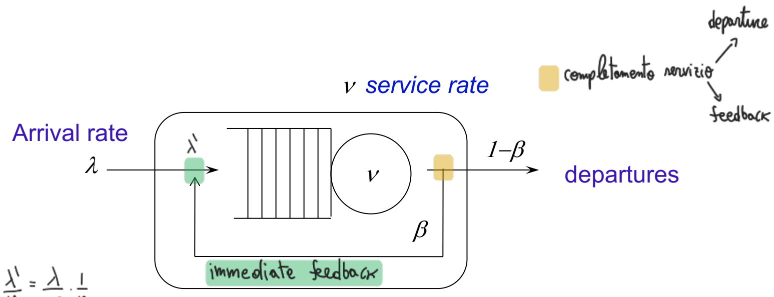
10

Exercises

- Exercises: 3.2.3, 3.2.4, 3.2.7

P 115 discrete

Model Considerations



Bilanciamento flusso:
$\lambda = \lambda'(1-\beta) \rightarrow \rho = \frac{\lambda'}{\nu} = \frac{\lambda}{1-\beta} \cdot \frac{1}{\nu}$

sistema saturo se $\rho \rightarrow 1$, cioè $\frac{\lambda}{(1-\beta)\nu} \rightarrow 1$: $\frac{\lambda}{\nu} \rightarrow 1-\beta$ cioè $\beta \rightarrow 1-\frac{\lambda}{\nu}$

- When feedback occurs the job joins the queue consistent with the queue discipline
- The decision to depart or feed back is random with *feedback probability β*

- Feedback is independent of past history
- In theory, a job may feed back arbitrarily many times

```
int GetFeedback(double beta)        /* 0.0 <= β < 1.0 */
{
    SelectStream(2);
    if (Random() < beta)
        return (1);                  /* feedback*/
    else
        return (0);                  /* no feedback */
}
```


Statistical considerations

- Index i=1, 2, 3, … counts jobs that enter the service node
  - fed-back jobs are not recounted
- Using this indexing, all job-averaged statistics remain valid
- We must update delay times, wait times and service times for each feedback
- Jobs from outside the system are merged with jobs from the feedback process
- The steady-state request-for-service rate is larger than λ by the positive additive factor $\beta \bar{x} \nu$ (Prob. feedback · job in servizio · rate servizio)
- Note that $\bar{s}$ increases with feedback but $1/\nu$ is the average service time per request

esempio 3.3.1

Algorithm and Data Structure Considerations

job index	1	2
Arrival/feedback	1	3
service	9	3
departure	10	**13**

feedback

single server node FIFO con immediate feedback

DE simulation
ssq with feedback

Algorithm and Data Structure Considerations

job index	1	2	3	4	5	.	6
Arrival/feedback	1	3	4	7	10		14
service	9	3	2	4	7		6
departure	10	**13**	**15**	19	**26**		37

Algorithm and Data Structure Considerations

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** …

At the computational level, some algorithm and data structure is necessary

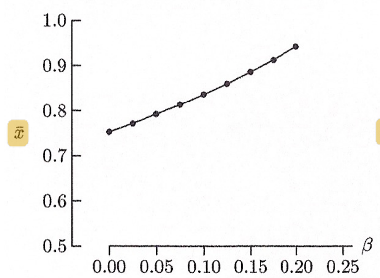
Le departure per i feedback coincidono con il loro arrival time nella coda (job "i" departure a t=15, ma deve rientrare nella coda, vi entra a t=15)

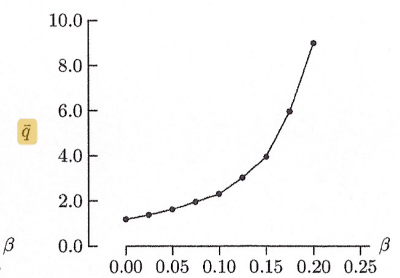
Theoretical values

$\bar{r}$	$\bar{w}$	$\bar{d}$	$\bar{s}$	$\bar{l}$	$\bar{q}$	$\bar{x}$
2.00	3.83	2.33	1.50	1.92	1.17	0.75

Program ssq2 was modified to incorporate immediate feedback

- interarrivals = *Exponential*(2.0)
- service times = *Uniform*(1.0, 2.0)



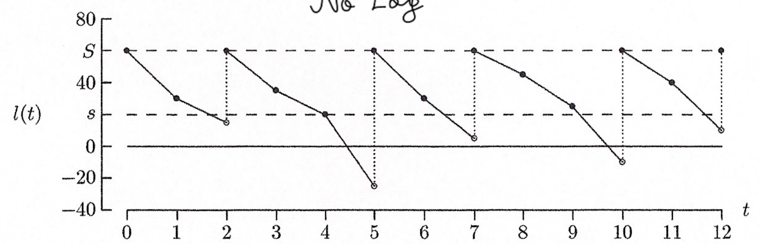


- It appears saturation is achieved as $\beta \to 0.25$

Inventory system with delivery lag

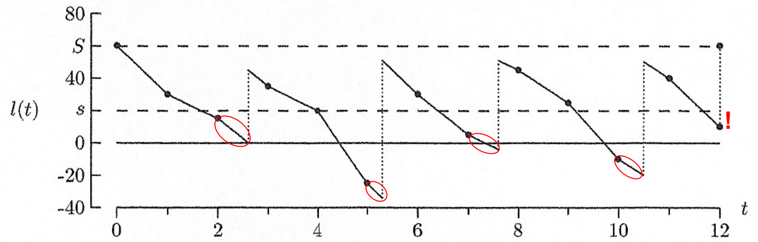
- *delivery lag* (dl) occurs when orders are not delivered immediately
- Lag is assumed to be random and independent of order size
- Without lag, inventory jumps occur only at inventory review times

No Lag



Inventory system with delivery lag

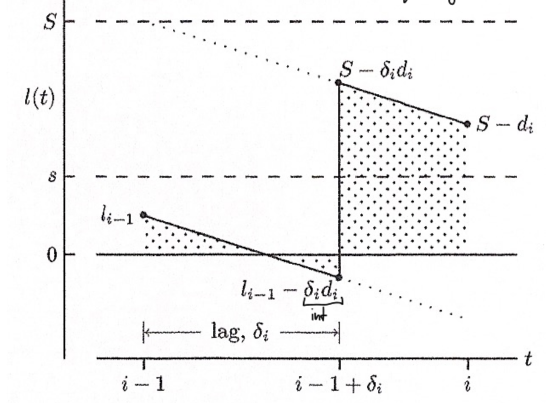
- With delivery lag, inventory jumps occur at arbitrary times



- The last order is assumed to have no lag (per bilanciamento flusso)
- We assume that orders are delivered before the next inventory review
- With this assumption, there is no change to the specification model

non sto sotto soglia minima, non sto in contesto di 'delivery lag'

- If $l_{i-1} \geq s$ the equations for $\bar{l}_i^+$ and $\bar{l}_i^-$ remain correct
- If $l_{i-1} < s$, the time-averaged holding and shortage intervals must be modified
  - the dl for interval i is $0 < \delta_i < 1$ delivery lag case



if $l_{i-1} < s$ then order $S - l_{i-1}$ placed at $t = i-1$, con $0 <$ lag < 1

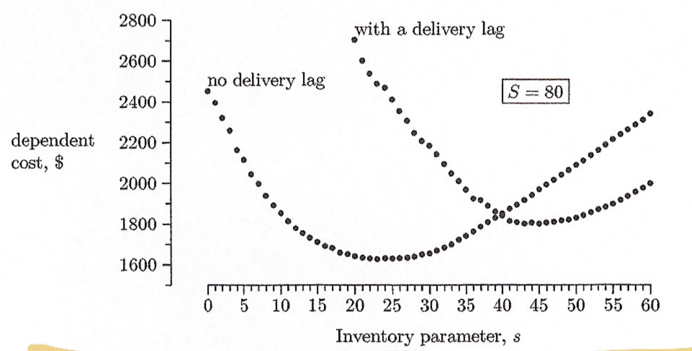
- senza lag:
  $S = 10$, $s = 3$
  appena $l_i = 3$, metterò $S - l_i = 7$: oggetto aveva $l_i = 10$
- con lag
  $S = 10$, $s = 3$
  $l_i = 3$, ordino oggetti $S - l_i = 7$, ma mi arrivano quando magari $l_i = 0 \to l_i = 7$

Consistency Checks

- It is fundamentally important to verify extended models with the parent model (before the extension)
  - Set system parameters to special values
- Set $\beta = 0$ for the ssq with feedback
  - Verify that all statistics agree with parent
- Using the library `rngs` facilitates this kind of comparison
- It is a good practice to check for intuitive "small-perturbation" consistency
  - Use a small, but non-zero β and check that appropriate statistics are slightly larger

- For the InvSys with delivery lag, $\delta_i = 0.0$ iff no order during i^{th} interval, $0 < \delta_i < 1.0$ otherwise
- The InvSys is *lag-free* iff $\delta_i = 0.0$ for all i
- If (S, s) are fixed then, even with small dl:
  - $\bar{o}, \bar{d}, \bar{u}$ are the same regardless of delivery lag
  - Compared to the lag-free system, $\bar{l}_i^+$ will decrease
- Compared to the lag-free system, $\bar{l}_i^-$ will increase or remain unchanged

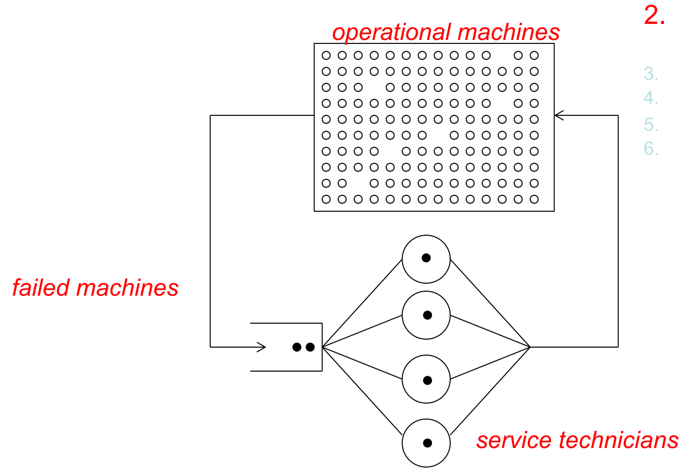
- Delivery lags are indipendent *Uniform*(0.0, 1.0) random variates



- delivery lag causes $\bar{l}_i^+$ to decrease and $\bar{l}_i^-$ to increase or remain the same
- with C_{hold}= \$25 and C_{short}= \$700, cause shift up and to the left

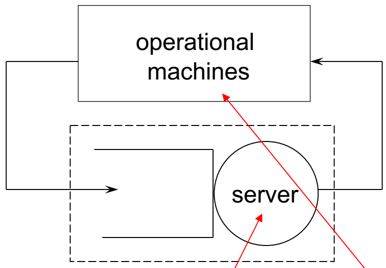
Machine Shop Model

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



Machine-shop, ssq model

The machine shop model is closed because there are a finite number of machines in the system



- Assume repair times are *Uniform*(1.0, 2.0) random variates
- There are *M* machines that fail after an *Exponential*(100.0) random variate

Program `ssms`

- program `ssms` simulates a Single Server Machine-Shop
- the library `rngs` is used to uncouple the random process
- the failure process is defined by the array failures
  - a O(M) search is used to find the next failure
  - alternate data structures can be used to increase computational efficiency


```
 double GetFailure(void)
{   SelectStream(0);
    return (Exponential(100.0));}

double NextFailure(double failure[], int *m)
{   int     i = 0;
    double t = failure[0];

    *m = i;
    for (i = 1; i < M; i++)
        if (failure[i] < t) {
           t = failure[i];
          *m = i;
        }
    return (t);
}

double GetService(void)
{   SelectStream(1);
    return (Uniform(1.0, 2.0));
}
```

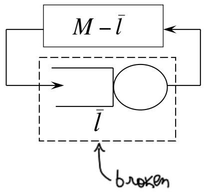
```c
int main(void)
{
  long   index     = 0;      /* job (machine failure) index */
  double arrival   = START;  /* time of arrival (failure)   */
  double delay;              /* delay in repair  queue      */
  double service;            /* service (repair) time       */
  double wait;               /* delay + service             */
  double departure = START;  /* time of service completion  */
  int    m;                  /* machine index 0,1,...(M-1)  */
  double failure[M];         /* list of next failure times  */
  struct {                   /* sum of ...            */
    double wait;             /*   wait times          */
    double delay;            /*   delay times         */
    double service;          /*   service times       */
    double interarrival;     /*   interarrival times  */
  } sum = {0.0, 0.0, 0.0};
```



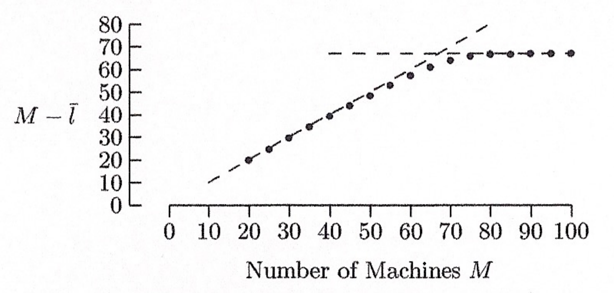
```c
PlantSeeds(123456789);

for (m = 0; m < M; m++)           /* initial failures */
failure[m] = START + GetFailure();

while (index < LAST) {
   index++;
   arrival      = NextFailure(failure, &m);
   if (arrival < departure)
       delay      = departure - arrival;
   else
       delay      = 0.0;
   service      = GetService();
   wait         = delay + service;
   departure    = arrival + wait;             /* completion of service  */
   failure[m]   = departure + GetFailure(); /* next failure, machine m */
   sum.wait    += wait;
   sum.delay   += delay;
   sum.service += service;
}
sum.interarrival = arrival - START;
```



- The time-averaged number of working machines is $M - \bar{l}$



- For small values of M the time-averaged number of operational machines is essentially M
- For large values of M this value is essentially constant at approximately 67

Exercises

- Exercises: 3.3.2, 3.3.3, 3.3.4, 3.3.7