Project of performance evaluation of computer systems and networks

Batch Server

Students:

Damiano Barone

Luigi De Bianchi

Lorenzo Diana

INDEX

Images index

Define the objectives

Our work was to study the response time of a batch server, by varying the maximum number of jobs (R) taken from the queue for a work. We have one queue. The service time and client requests arrive with an exponential distribution. It was also made to vary the mean inter-arrival time and the mean service time. The response time is the time that begins when a job enters in a queue until it comes out from the server, so it is composed of queue time plus work batch time.

Define Performance Indices

a. **batch work size:** is the maximum number of jobs that the server takes by queue for a work;
b. **nidle:** is a scalar that represents the number of times the server ends a work and does not find any job in the queue. So it is the number of times that the server will go to sleep;
c. **npackets:** represents the number of packets in the queue;
d. **queue time:** is the time that a job spends in the queue;
e. **response time:** evaluates the response time;
f. **batch work time:** is the time of a batch work;
g. **nworksArrived:** number of jobs that arrived to the server;
h. **nworksServed:** number of jobs elaborated by the server.

We use all these indexes to have a complete view of the system. Our goal is to study the response time, but we can't study it if we do not see the other statistics. A useful index is nidle, because we prefer a minimized nidle. Npackets allow us to see how the queue varies over time.

All statistics have been collected in *server.cpp*, in the function workbatch the first statistic saved is the queue time of each element taken in the batch. As you can see:

```
job *support = coda.front();
tempicoda[i]=simTime() - ( support->getEnterQueue() );
emit(s_queueTime,tempicoda[i]); //time spend in queue
coda.pop();
```

All other statistics are taken from handlemessage, where at the beginning of the function we have an IF to distinguish whether the message has come from the module itself, or if it comes from the client. In the event the message arrives from the server, it means that batch work has finished and then we save the statistics on the batch; that is: response time, nworksServed, batchWorkTime, batchWorkSize .

```
for(int j=0 ; j<i ; j++)
        emit(s_responseTime, tempicoda[j]+sommaST);
emit(s_nWorksServed, i);
emit(s_batchWorkTime,sommaST); //batch time
emit(s_batchWorkSize,i);
```

In the event the queue doesn't have any element, we save the statistic idle, the server sleep until a message arrive from the client.

Instead when a message arrives from the client, we save nPackets
nWorksArrived.

Model of the system

We build a very simple system with only 2 modules: one client and one server.
Each module is an omnet simple module.

NED DEFINITIONS

CLIENT

```
simple Client
{
  parameters:
     double   meanIAT;     //mean of the Inter Arrival Time distribution
     double   meanST;      //mean of the Service Time distribution

  gates:
        output          outClient;
}
```

SERVER

```
simple Server
{
  parameters:
        int     sizeQueue;          //queue dimension
        int     sizeR;              //batch dimension

     //statistics
     //here we put statistics handler

  gates:
        input   inServer;
}
```

NET

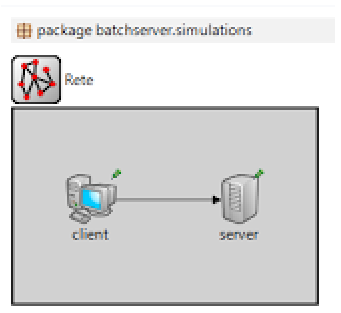


Figura 1 - Net

```
network Rete
{
   submodules:
       client: Client {}
       server: Server {}
   connections:
       client.outClient --> server.inServer;
}
```

JOB

```
message job{
   double ST;                      //time to execute the process
   simtime_t enterQueue;    //timestamp di entrata in coda
}
```

MODULE FUNCTIONING

As we can see the structure is really simple

- The client has only one output gate and its task is to build messages evaluating Interarrival time and Service time.
- Server has also only one input port and should only append and manage the messagges delivered by the client.
- Job is the message we have defined and contains parameters needed to evaluate statistics.

SIMPLE MODULE CODE

CLIENT.CC

```
void Client::initialize()
```

```
{
  scheduleNew();
}


void Client::handleMessage(cMessage *msg)
{
   send(msg, "outClient");
   scheduleNew();
}


void Client::scheduleNew()
{
   //initialize rv
   IATrv=exponential((double)par("meanIAT"));
   STrv=exponential((double)par("meanST"));

   //create job
   job *pkt=new job("pkt");
   pkt->setST( STrv);

   //schedule job
   scheduleAt(simTime()+IATrv, pkt);

}
```

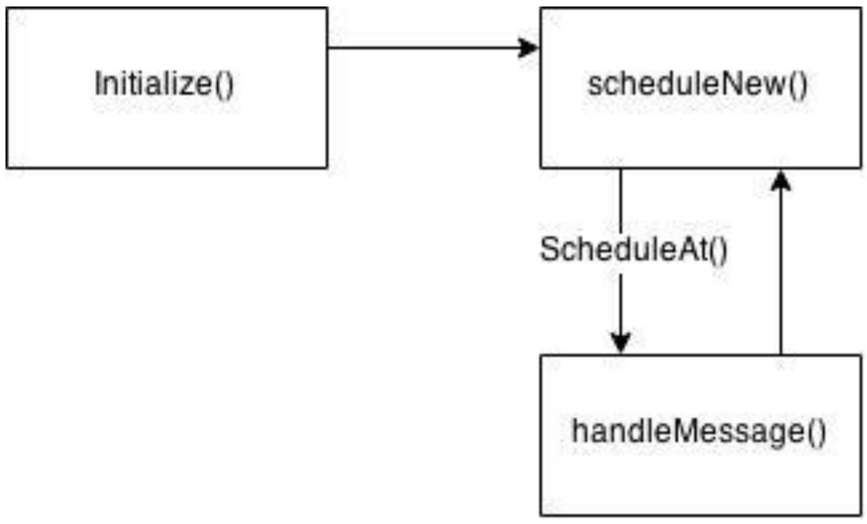


Figura 2 - client

SERVER.CC

```
#include <cmessage.h>
#include <cobjectfactory.h>
#include <cregistrationlist.h>
#include <onstartup.h>
#include <regmacros.h>
#include <server.h>
#include <simutil.h>
#include <iostream>
#include <queue>

using namespace std;

Define_Module(Server);

void Server::initialize()
{
   idle = true;

   tempicoda = new simtime_t[(int)par("sizeR")];

   //statistics
   s_nPackets = registerSignal("nPackets");
   s_nWorksArrived = registerSignal("nWorksArrived");
   s_nWorksServed = registerSignal("nWorksServed");
   //s_nWorksDropped = registerSignal("nWorksDropped");
   s_queueTime = registerSignal("queueTime");
   s_batchWorkTime = registerSignal("batchWorkTime");
   s_batchWorkSize = registerSignal("batchWorkSize");
   s_responseTime = registerSignal("responseTime");
   s_nidle=registerSignal("nidle");
}


void Server::handleMessage(cMessage *msg)
{
   ev <<"handler\n";

   if ( msg->isSelfMessage() )
   {
      ev<<"batch finito\n";
      delete msg;

      /**
       * gestione emit riferiti al batch appena terminato
```

```
     */
    for(int j=0 ; j<i ; j++)
        emit(s_responseTime, tempicoda[j]+sommaST);

    emit(s_nWorksServed, i);            //statistica globale di lavori serviti
    emit(s_batchWorkTime,sommaST);      //tempo di un batch
    emit(s_batchWorkSize,i);            //pachetti per batch



    if( !coda.empty() ){    //coda non vuota
        ev<<"coda non vuota chiamo il successivo batch\n";
        idle = false;
        workBatch();
    } else {
        ev<<"coda vuota vado in idle\n";
        idle = true;
        emit(s_nidle,1);
    }
} else {
    ev<<"messaggio esterno\n";

    //saving number of packets in the queue
     emit(s_nPackets, (int)coda.size());
     emit(s_nWorksArrived, 1); //numero di pacchetti arrivati

     /**
      * This code must be uncommented if we want to try the system in condition of
      * finite queue
      */
     /*
     if ( (int)coda.size()==(int)par("sizeQueue") ) {
        ev<<"coda piena\n";
        emit(s_nWorksDropped,1);//numero pachetti persi
        delete msg;
    } else {
    */
        ev<<"push\n";
        dynamic_cast<job*>(msg)->setEnterQueue(simTime());
        coda.push(  dynamic_cast<job*>(msg) );

        if( idle ){
            idle = false;
            workBatch();
        }
    //}
}
```

```
    ev <<"finehandler\n";
}


void Server::workBatch()
{
    sommaST = 0;

    ev <<"work\n";

    //inizializzazione
    int R=(int)par("sizeR");

    for (i=0; i<R;i++)
    {
        ev<<"ciclo: "<<i<<endl;

        if ( coda.empty() ){
            ev<<"coda svuotata\n";
            break;
        } else {
            ev<<"pop\n";

            sommaST+=coda.front()->getST();

            job *support = coda.front();
            tempicoda[i]=simTime() - ( support->getEnterQueue() );
            emit(s_queueTime,tempicoda[i]); //tempo che va dalla coda al server
            coda.pop();
            delete support;//perche coda.pop non chiama il distruttore
        }
    }
    scheduleAt(simTime()+sommaST, new cMessage);

    ev <<"endwork\n";
}

void Server::finish()
{
    while( !coda.empty() ){
        job *support = coda.front();
        coda.pop();
        delete support;
    }
    delete tempicoda;
}
```

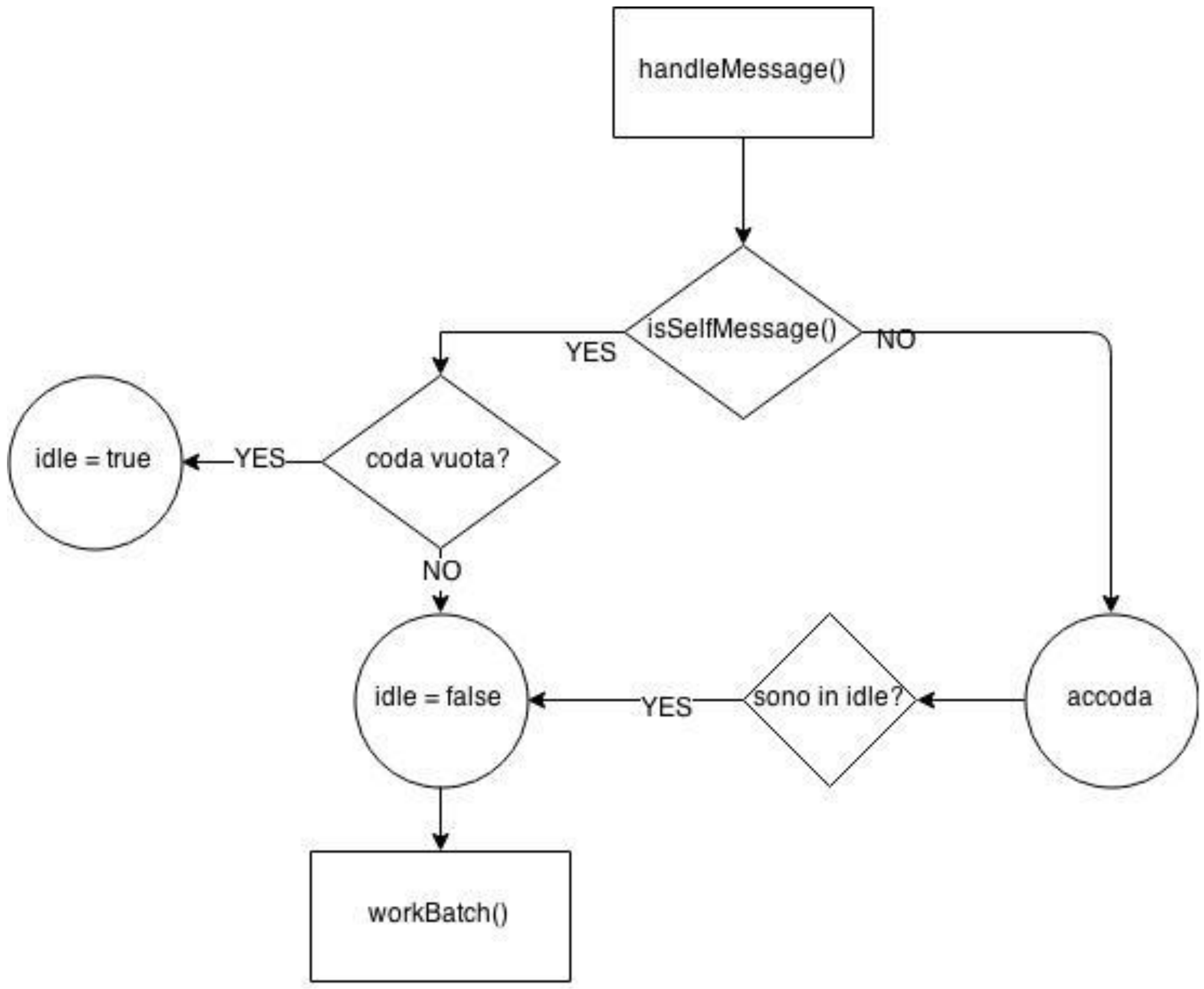


Figura 3 - server

NOTE: workBatch() is called only if there is at least one message.

Validate the model

We develop the system to be realistic so each step of the system is the same as a real working server batch.
In our model we have a queue followed by a server that takes work group (batch).
On its arrival, a new job is placed in the queue, where it remains until the server picks up a batch that contains this job. All the jobs of the same batch are released simultaneously as soon as the server finishes processing all the jobs belonging to the batch.
The queue time is estimated from the time when the job enters the queue to when it is taken in by the server. The service time is calculated from the time when the job is extracted from the queue to the time that is issued by the server.

Also every emit to evaluate statistics is called in the system to be as faithful as possible.

Define the factor

Regarding seeds we used *seed-set=${repetition}* in order to let Omnet++ choose seeds automatically. In the system we use two exponential distributions, one to generate inter-arrival time and one for service time, so we need two different seeds for each repetition.

About the queue size, we choose to release the problem on this parameter, that is why we used an endless queue. Regarding batch sizes, we choose {5, 50, 100} jobs per batch, in order to show different system behaviors.

We choose to set the inter-arrival time to 4, and we use a service time equal, 10% and 50% greater and lesser than inter-arrival time. So we use {2, 3.6, 4, 4.4, 6} for service time.

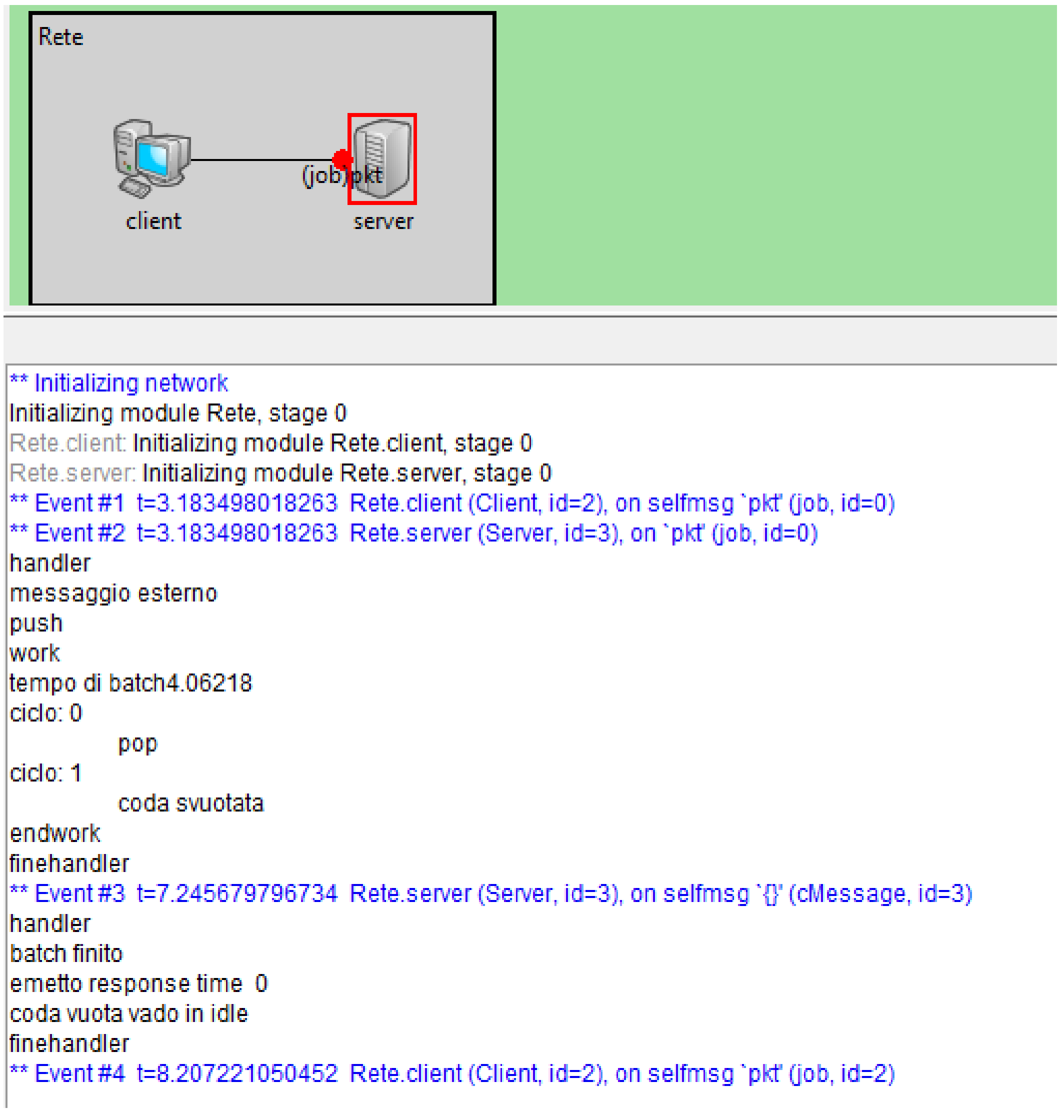
Implementation

In order to verify the correctness of the code we have added some debug messages to keep track of the events occurred during the execution of the simulation.
To name a few:

- The reception of a message;
- Start and stop of a batch
- Push and pop of a job from the queue.

Debug messages show the behavior we expect. Below you can see a screen with the debug messages obtained from an execution.
We can observe:

- The push of a new job;
- Start of batch, that pop messages from the queue until it empties the queue or reaches the maximum number of jobs per batch;
- Computation of the batch time for this batch;
- The end of the batch elaboration;
- In the end the attempt to start a new batch, if possible.



```
** Initializing network
Initializing module Rete, stage 0
Rete.client: Initializing module Rete.client, stage 0
Rete.server: Initializing module Rete.server, stage 0
** Event #1  t=3.183498018263  Rete.client (Client, id=2), on selfmsg `pkt' (job, id=0)
** Event #2  t=3.183498018263  Rete.server (Server, id=3), on `pkt' (job, id=0)
handler
messaggio esterno
push
work
tempo di batch4.06218
ciclo: 0
                pop
ciclo: 1
                coda svuotata
endwork
finehandler
** Event #3  t=7.245679796734  Rete.server (Server, id=3), on selfmsg `{}' (cMessage, id=3)
handler
batch finito
emetto response time  0
coda vuota vado in idle
finehandler
** Event #4  t=8.207221050452  Rete.client (Client, id=2), on selfmsg `pkt' (job, id=2)
```

Figura 4 - debug example

Calibrate the simulator

The simulation is obviously calibrated to be as realistic as possible, but it must also be considered how the two random variables contribute to the definition of the response time.

We therefore decide to keep one of the two sets and evaluate the response of the system to the variation of the second.
The evidence to the contrary is not necessary as we would go to evaluate the same space of solutions which in our case is represented by the two random variables that can be the same or one greater than the other.

Typical values of traffic are:

- OpenStreetMap seems to have 10-20 per second;
- Wikipedia seems to be 30000 to 70000 per second spread over 300 servers (100 to 200 requests per second per machine, most of which is cache);
- Geograph is getting 7000 images per week (1 upload per 95 seconds).

So we have extremely heterogeneous data according to the service.

We choose a low enough inter-arrival time assuming our server is not particularly required. So we will have a mean IAT fixed at 1 request every 4 second.

Therefore the mean service time must be comparable to inter-arrival time; in this case we choose to have a service time greater and smaller than 10% and 50%.

So meanST can be 2, 3.6, 4, 4.4 and 6.

Regarding the queue this will be infinite.

About the size of the batch, we are not able to make assumptions so we choose arbitrary values like 5, 40 and 100 and we later evaluate the behavior.

Run simulation

The first step in making the simulation is to configure the file *omnetpp.ini*, to find the warmup for each configuration of meanIat, meanSt and R a simtime 100,000 and see where the mean value stabilized for each statistic. After we have done all the simulation adopting various warmups and saving its mean in excel, we calculate the averages of repetitions of each statistic and the confidence interval. Finally we plot graphs of each statistic, and add error bars.

Data analysis

So after we have defined our range of interest and parameters we can evaluate our statistics with various configurations of our *.ini* file.

This is a general ini file with all the combinations of parameters that we want to test.

```
[General]
network= Rete
sim-time-limit=100000s
#warmup-period=
repeat=10
seed-set=${repetition}
Rete.client.meanIAT = ${4}
Rete.client.meanST = ${2, 3.6, 4, 4.4, 6}
Rete.server.sizeR = ${1, 5, 50}
```

WHY 10 REPETITIONS

First of all we must evaluate how many repetitions are needed.
We launched a simulation with 15 repetitions with meanIAT=4 and meanST=3.6 and we exported this data of the response time with the mean evaluated in sliding window.

Run number	Mean	Media in sliding window
1	24,31467935	24,31467935
2	65,73165203	45,02316569
3	29,48077771	39,8423697
4	35,67378983	38,80022473
5	32,56731547	37,55364288
6	37,08067473	37,47481485
7	37,47691784	37,47511528
8	36,61142494	37,36715399

9	39,33851351	37,58619393
10	41,47877402	37,97545194
11	29,71780231	37,22475652
12	35,62694004	37,09160515
13	36,89705851	37,07664002
14	27,54516781	36,39582058
15	51,06235586	37,3735896

We can clearly see that the mean of the seed is stable using the mean from 7 seeds but we choose to select 10 seeds to have more reliable data.

We decided to evaluate the response of the system with different batch sizes.

Here we put our assessments over the collected data; for numeric value and graph see attached excel files.

BATCH SIZE 1

This is an extreme example since it’s the basic server with FIFO service.

RESPONSE TIME
We expect a low response time with ST<<IAT and on the contrary a high response time because the server receives more jobs compared to the ones that is capable to serve.

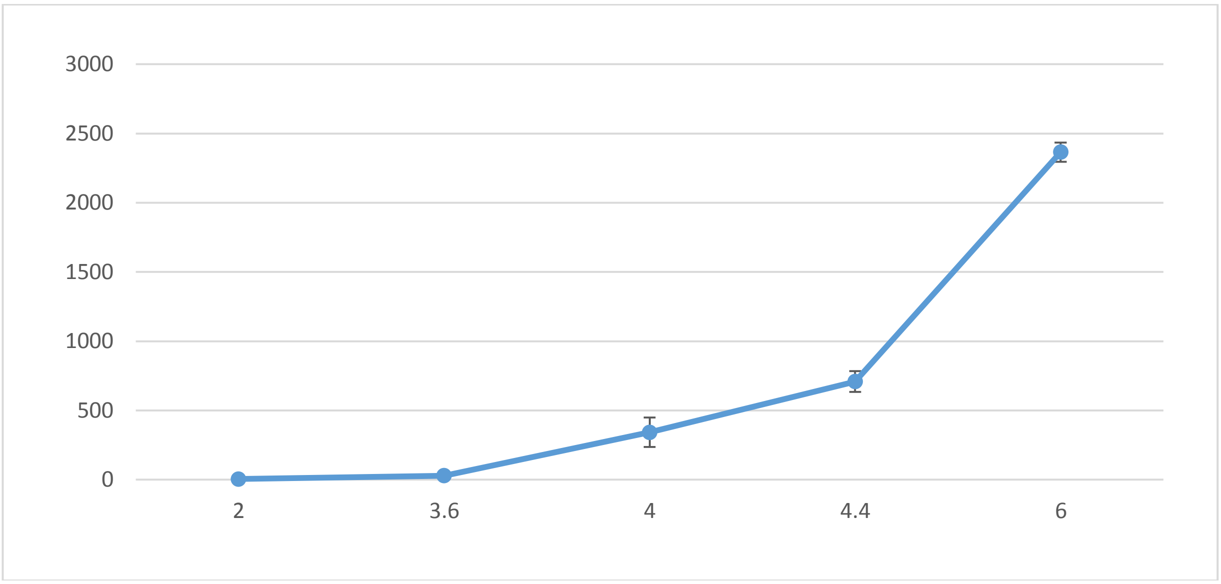


Figura 5 - Response time - batch size 1

IDLE

As expected with big ST the number of idle decreas up to the case that have no idle, except in the warm-up period.

N PACKETS IN QUEUE & QUEUE TIME

Like in the other analysis the number of packets in queue grows with the increase of the Service Time. We have an almost empty queue when the server is fast (service time =2) and a fairly big one on the opposite case because the server is not able to serve all the requests.

BATCH SIZE 5

RESPONSE TIME

We have a convergent behavior of the response time for low values of ST, but when ST becomes higher compared to the fixed IAT we obtain a divergent behavior.

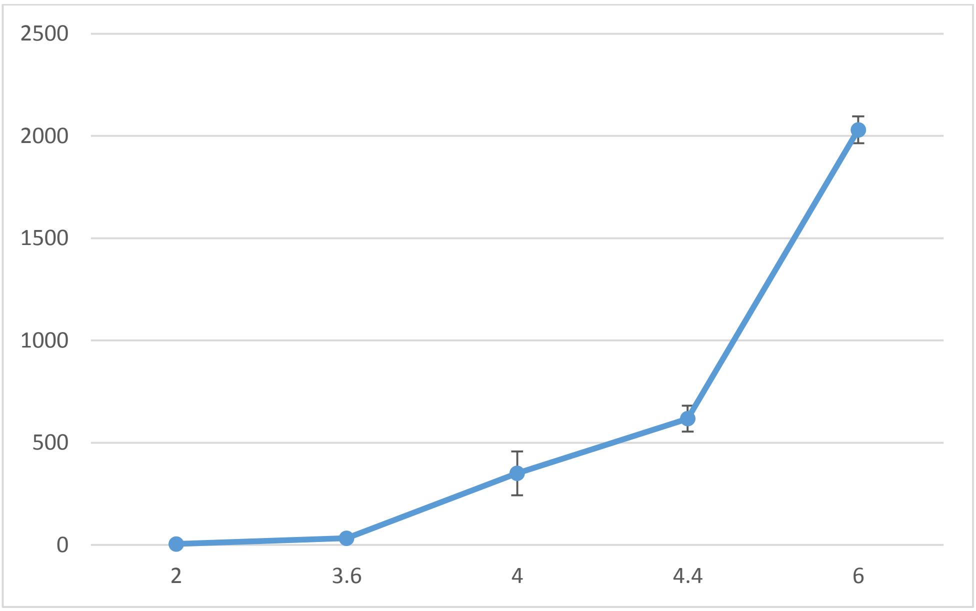


Figura 6 - Response time - batch size 5

N PACKETS IN QUEUE & QUEUE TIME
Like in the previous case study for batch size equal to 1 we can see that the two statistics grow with a slower server, that is big value of ST.

BATCH STATISTICS:
BATCHTIME & BATCH SIZE
This statistics are related to each other, because when the batch size grows the batch time grows consequently. In order to increase the batch size we need a big value for ST (respect to the IAT fixed), so we get a big value of batch time.
As expected the correctness of the simulation is verified observing that batch time = batch size * service time

BATCH SIZE 50

In this case all the considerations made for the previous batch size are confirmed.

Response time

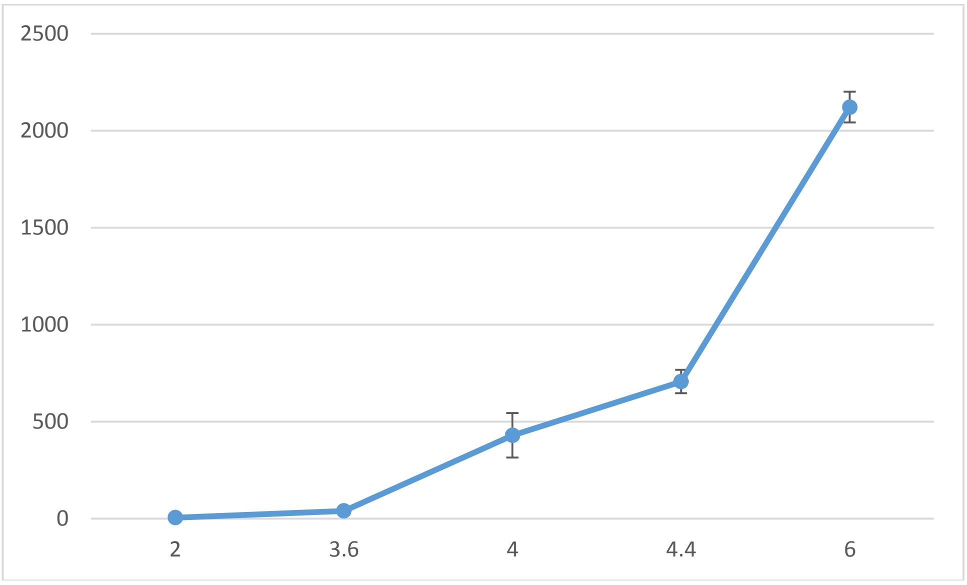


Figura 7 - Response time - batch size 50

Analysis with exponential batch work time

Here we evaluate a system with a batch work time independent to the number of packets.

WORK CYCLE

The work loop is the same as the previous analysis but this time the batch time is generated by the server using an exponential random variable and it’s independent from the number of packets.
So we consider a server capable of parallel execution of each job.

WHAT WE EXPECT

We can think of two main cases; where the batch time is fairly bigger or smaller than the inter-arrival time.

We can predict that if

batchWorkTime << interArrivalTime

The server is very fast and searches for new batches fairly frequently, we can think that the queue time for each job is small since the consumption rate is high and consequently the batch size is nearly one because the queue can’t be filled.

batchWorkTime >> interArrivalTime

Server is fairly slow and this means that the number of jobs in the queue increases during the simulation, so the queue time is bigger and bigger over time.
Probably the mean batch work size is fairly near to the maximum batch size, so in this case we have a very good utilization of the batch server ability.

REVISION OF THE CODE

Now the batch work time is evaluated by the server once for batch.

```
void Server::workBatch()
{
    sommaST = 0;
    ev <<"work\n";

    //inizializzazione
    int R=(int)par("sizeR");
    sommaST=exponential( (double)par("batchST") );

    ev<<"tempo di batch"<<sommaST <<"\n";

    for (i=0; i<R;i++)
    {
        ev<<"ciclo: "<<i<<endl;
        if ( coda.empty() ){
            ev<<"\tcoda svuotata\n";
            break;
        } else {
            ev<<"\tpop\n";
            job *support = coda.front();
            tempicoda[i] = simTime() - ( support->getEnterQueue() );
            emit(s_queueTime,tempicoda[i]); //tempo che va dalla coda al server
            coda.pop();
            delete support;//perche coda.pop non chiama il distruttore
        }
    }
    scheduleAt(simTime()+sommaST, new cMessage);
    ev <<"endwork\n";
}
```

DATA COLLECTION

Here we have data collected from the simulation.
In the table we have the mean value of the parameters over 10 repetitions like we have in the previous analysis.

This time we evaluated different scenarios with 2 values of the batch size and different service time for the batch with fixed values for the inter-arrival time (4 like the previous case study).

Response time	ST=200	ST=40	ST=8	ST=4,4	ST=2,7	ST = 0,4
R = 5	44.971,1522956577	25.005,4820	16,4130314	7,5628015689640	4,16306099	0,439941932

R = 50	2.240,6895634217	80,4037120	14,7654744	7,4124155092561	4,14148742	0,439941932

Queue time	ST=200	ST=40	ST=8	ST=4,4	ST=2,7	ST = 0,4
R = 5	44.862,0826229269	24.975,5691	8,45657334	3,1932739536439	1,45785997	0,040542096
R = 50	2.044,0162109708	40,3541618	6,79442113	3,0369419284691	1,43505916	0,040542096

# work per batch	ST=200	ST=40	ST=8	ST=4,4	ST=2,7	ST = 0,4
R = 5	4,99120303047	4,99791786	2,28724727	1,5590635373261	1,27257792	1,009713955
R = 50	48,3718394640	10,1026627	2,32732382	1,5707170003590	1,27484174	1,009713955

As we can see previsions are confirmed and the system behaves as we expected.

RESPONSE TIME & QUEUE TIME

With higher service time we have a slower server and consequently we can see higher response time and queue time.

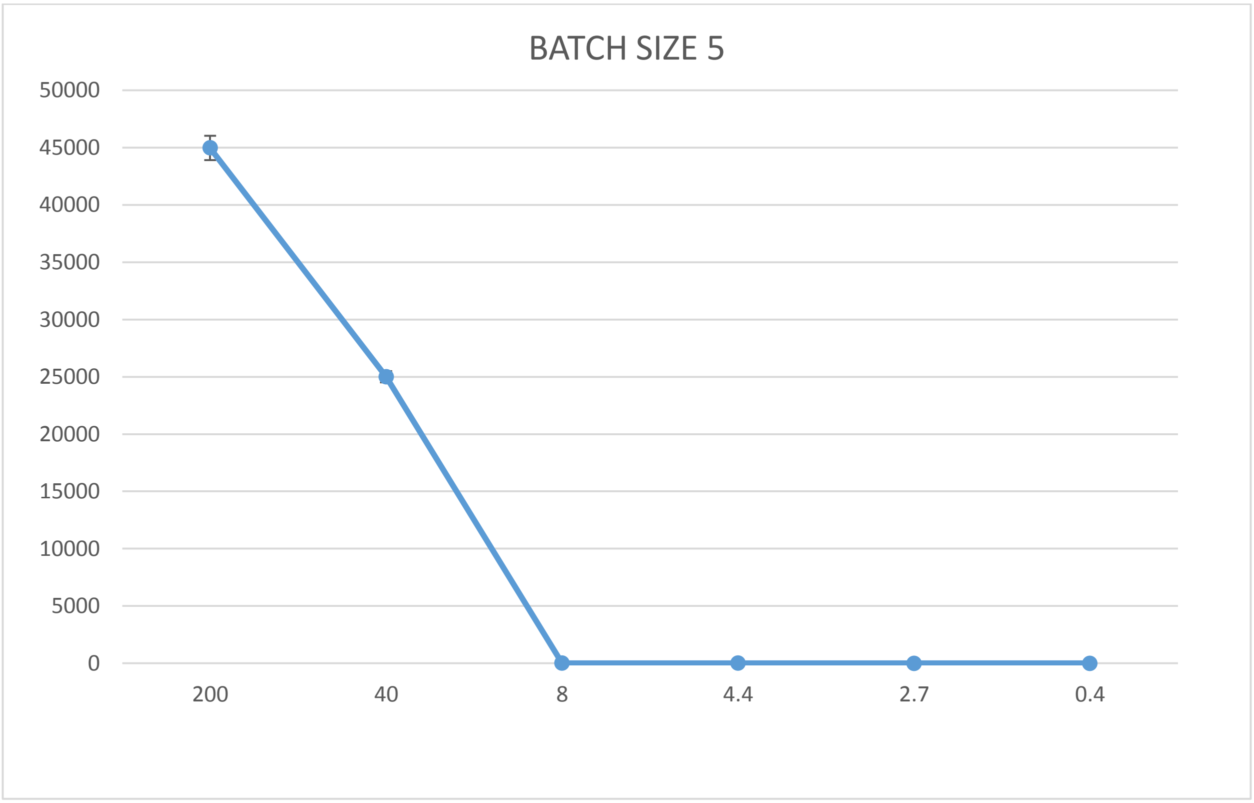


Figura 8 - Response time - batch size 5

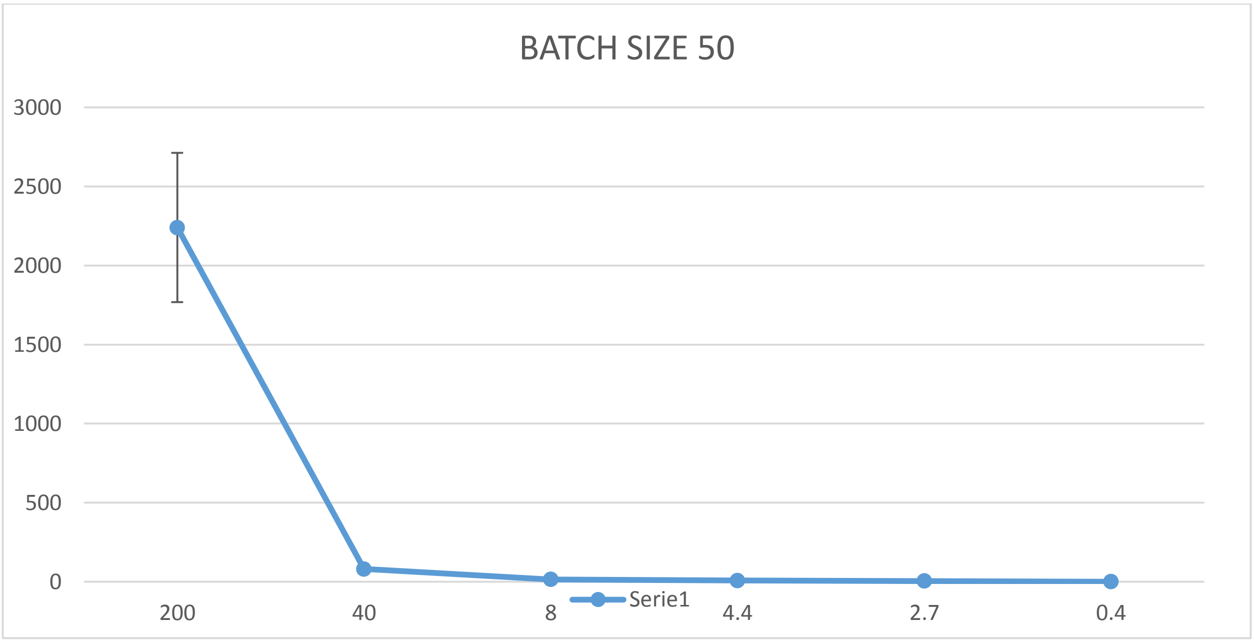


Figura 9 - Response time - batch size 50

MEAN BATCH SIZE

Also in this case the previsions are correct and we can notice two interesting facts:

1. In case we have small service time the batch size in not relevant;
2. In case we have high service time we can take advantage of the batch characteristics of the system.

Final note

In the end we can highlight the different behaviors between the two systems.

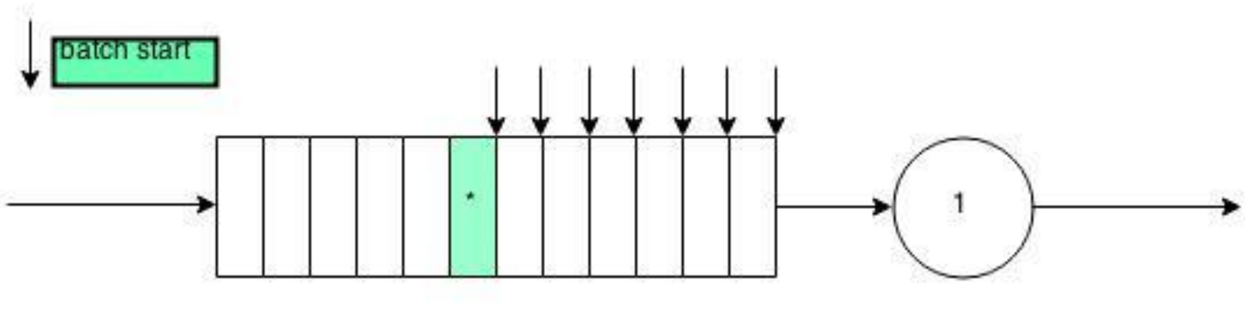


Figura 10 - FIFO server

In the first we have a server that takes many jobs and processes them sequentially, then releases them all together, so when we increase the batch size we don't have any relevant improvement.

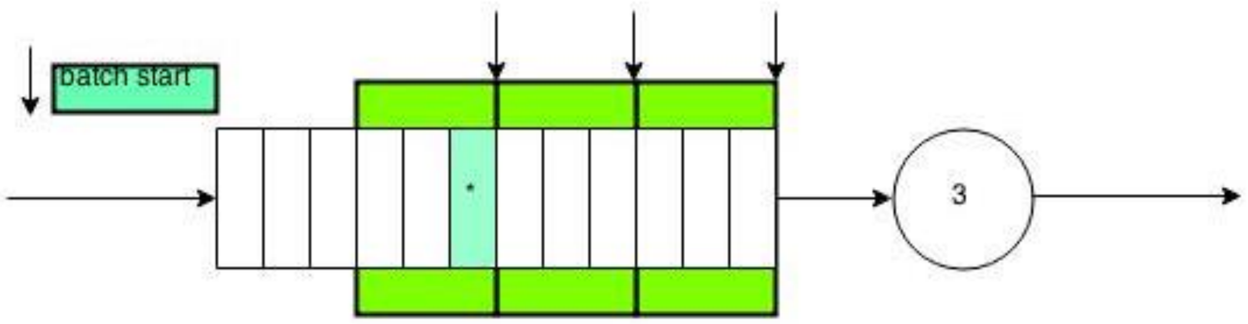


Figura 11 - Serial Batch server

As we can see from the above image, the selected job is in the worst possible position, the first of its batch, because it must waiting the end of all other jobs in the batch. On the other hand a job at the end of the batch wait for the same time experienced in a fifo server.

Resposnse time in worst case scenario (blue job):

In a fifo server:

$6\ st + st + T_{queue}$

In a serial batch:

$2\ st_{bach} + st_{batch} + T_{queue} =$
$6\ st + 3\ st + T_{queue}$

From experimental data we can see that the little variation in response time are given from the fact that the jobs are mostly in a position worst then the best case.
This variation are only given by the batch nature of the server; because we used the same seeds in every simulation so the time spend in queue is the same.
It is important to highlight that this results are only clearly visible in a system always busy.

In a real case the improvement that come from this type of system is measurable in terms of overhead reduction where taking jobs from the queue is a heavy load for the system.

In the second system the server processes the jobs at the same time, so in this case increasing the batch size we get improvements in the response time.

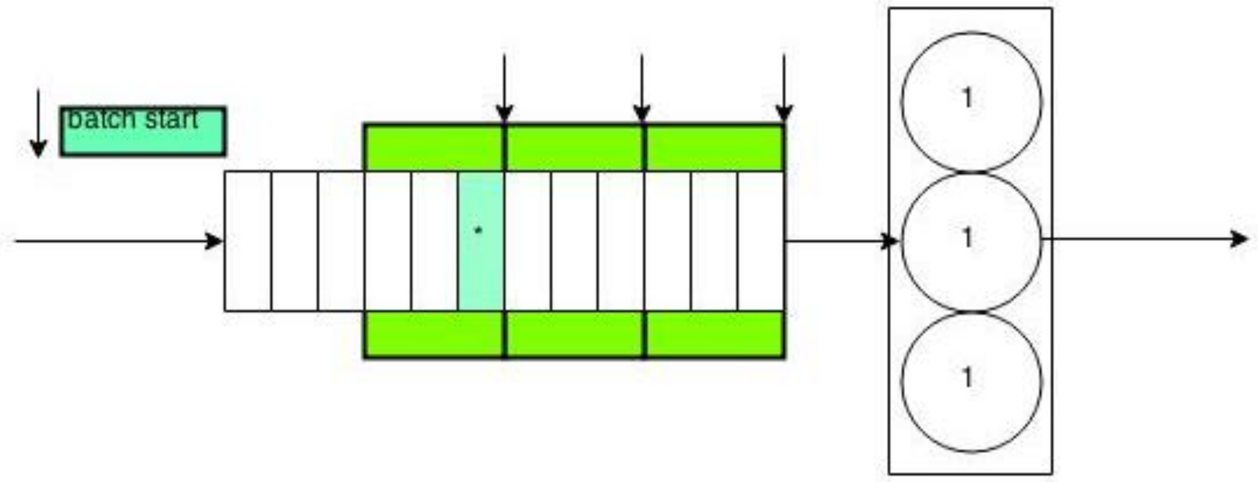


Figura 12 - Parallel Batch Server

Here we have a parallel consumption of the jobs; the improvements of performances is noticeable. On the other hand we need a fairly expensive server because we need N elaboration unit.