CSC 777- Telecommunication Network Design

Professor: Rudra Dutta

12/4/2013

Individual Study Topic Tutorial:

- **Linear Arrival Rate Approximations**
- **Balking Systems**

By:

Nikhil Khatu

Rushil Chugh

Linear Arrival Rate

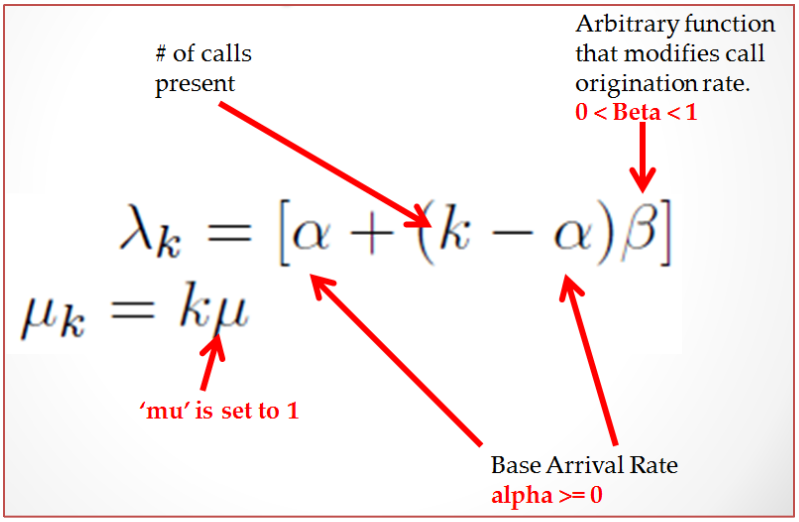
Traffic Characterization was covered in the Overflow Processes topic, in which systems with overflow traffic were modeled. The following measures of unevenness are given in the arrival process of a system:

- $Z = \sigma^2 / R$ ← Peakedness
- $D = V – R$ ← Difference between variance and the mean in the secondary system

For offered traffic: $Z = 1$ and $D = 0$ indicates Poisson arrivals. For overflow traffic: $Z > 1$ and $D > 0$ indicates some form of non-Poisson arrival. Since the secondary system has a non-Poisson arrival process (from overflow traffic), a non-Poisson arrival modeling method is needed for the arrival process.

To find a model for the arrival process in the secondary system Wilkinson[2] asked himself; “What would be the physical description of a cause system with a variance smaller or larger than the Poisson?” His account of the physical description was as follows; “If the variance is smaller, there must be forces at work which retard the call arrival rate as the number of calls recently offered exceeds a normal, or average, figure, and which increase the arrival rate when the number recently arrived falls below the normal level. Conversely, the variance will exceed the Poisson’s should the tendencies of the forces be reversed.”

Linear Arrival Rates as described by Girard[1] is a way to model the traffic offered to a lightly loaded circuit group receiving overflow traffic i.e. overflow traffic from the primary group is now the offered traffic to the secondary group. To model Wilkinson’s approach Girard picks a β value between ‘0’ and ‘1’. The equation for variance will be derived later, however it is important to know that lower β values result in lower variance and higher β values result in higher variance. This β value is the arbitrary function that modifies call origination rate while its counterpart (k-α) amplifies or attenuates the arrival rate depending on the number of calls (busy circuits) present in the system. α is simply the base arrival rate of the system.



# of calls present
Arbitrary function that modifies call origination rate.
0 < Beta < 1
$\lambda_k = [\alpha + (k - \alpha)\beta]$
$\mu_k = k\mu$
‘mu’ is set to 1
Base Arrival Rate
alpha >= 0

To reiterate; the focus is on the arrival process in a secondary system. To model the non-Poisson arrival process, Linear Arrival Rate is used as described above.

Birth/Death Process

The generic Birth Death process is as follows:

$$P_{k-1}(\lambda_{k-1}) + P_{k+1}(\mu_{k+1}) = P_k(\lambda_k) + P_k(\mu_k)$$

$$P_1 = P_0(\lambda_0/\mu_1)$$

$$P_{k+1} = [P_k(\lambda_k + \mu_k)/\mu_{k+1}] - [P_{k-1}(\lambda_{k-1})/\mu_{k+1}] \quad , \qquad k \geq 1$$

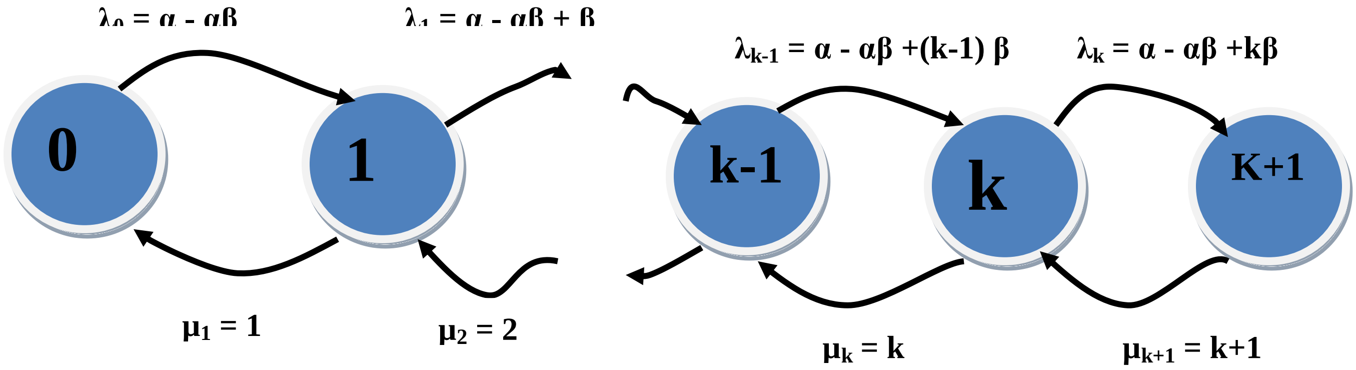
Hence, the distribution of the birth/death system is:

$$p_k = p_0 \prod_{j=0}^{k-1} \frac{\lambda_j}{\mu_j+1}$$

This equation represents the probability that one will find ‘k’ number of objects in the system at any given point in time given that any previous state is unknown.

Linear Arrival Rate Distribution

Each birth in our system is generated at a rate of λ$_k$ and each death is generated at a rate of μ$_k$ as described earlier. We can use the preceding λ$_k$ and μ$_k$ equations in the distribution equations of the birth death process. The state representation of the Linear Arrival Rate Birth/Death system is given in the following depiction.


$\lambda_0 = \alpha - \alpha\beta$
$\lambda_1 = \alpha - \alpha\beta + \beta$
$\lambda_{k-1} = \alpha - \alpha\beta + (k-1)\beta$
$\lambda_k = \alpha - \alpha\beta + k\beta$
0
1
k-1
k
K+1
$\mu_1 = 1$
$\mu_2 = 2$
$\mu_k = k$
$\mu_{k+1} = k+1$

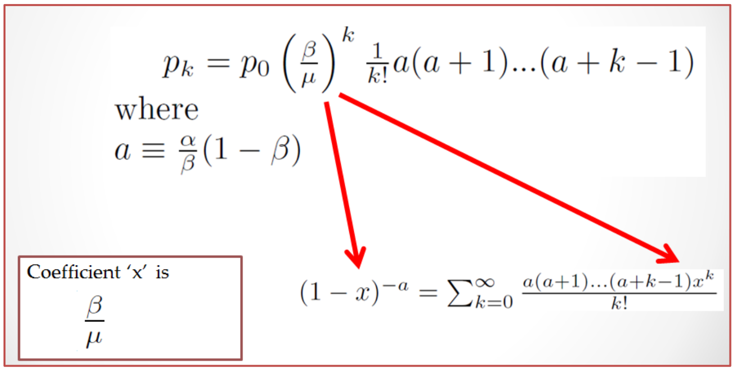
Once the corresponding lambda and mu values are used in the birth/death distribution equation we can derive the following distribution equation.

$$p_k = p_0 \left(\frac{\beta}{\mu}\right)^k \frac{1}{k!} a(a+1)\ldots(a+k-1)$$

where

$$a \equiv \frac{\alpha}{\beta}(1-\beta)$$

The resulting (preceding) equation is normalized by finding the Binomial Coefficient 'x'.



$$p_k = p_0 \left(\frac{\beta}{\mu}\right)^k \frac{1}{k!} a(a+1)\ldots(a+k-1)$$

where

$$a \equiv \frac{\alpha}{\beta}(1-\beta)$$

Coefficient 'x' is $\frac{\beta}{\mu}$

$$(1-x)^{-a} = \sum_{k=0}^{\infty} \frac{a(a+1)\ldots(a+k-1)x^k}{k!}$$

The final Probability Mass Function becomes:

$$p_k = \frac{1}{\left(1-\frac{\beta}{\mu}\right)^{-a}} \left(\frac{\beta}{\mu}\right)^k \frac{a(a+1)\ldots(a+k-1)}{k!}$$

$$a \equiv \frac{\alpha}{\beta}(1-\beta)$$

Simulating with Linear Arrival Rates

The C++ code for simulating the Linear Arrival Rate is posted on GitHub.
Git link: https://github.ncsu.edu/ngkhatu/CSC777.git
Zip file: https://github.ncsu.edu/ngkhatu/CSC777/archive/master.zip

This tutorial on simulating Linear Arrival Rate assumes the reader is familiar with Discrete Event Simulation. With this assumption the tutorial only covers how to create an engine specifically for a Linear Arrival Rate system.

Each simulation runs from 0 to 50,000 seconds with a precision of 1 millisecond. This infers that each simulation will run 50,000 * 1000 = 50,000,000 iterations. Speculating an expected range of objects in our system gives a rough buffer size of 1500.

```
#define initial_time 0.0
#define simulation_time 50000.0
#define hop_time 0.001
#define mu 1
#define Buffer_Size 1500
#define NUM_THREADS 4
```

The program is optimized to run on a quadcore system that it was developed for. Up to four threads will be spawned in main() with each thread running its own instance of a simulation.

```
struct thread_data{
    int alpha;
    double beta;
    int thread_id;
};
struct thread_data thread_data_array[NUM_THREADS];


void *simulate(void *threadarg)
{
//-------Thread Arguments----------------------
    struct thread_data *my_data;
    my_data = (struct thread_data*) threadarg;
    int alpha = my_data->alpha;
    double beta = my_data->beta;
    int thread_id = my_data->thread_id;
//------------------------------------------

int main()
{
    pthread_t threads[NUM_THREADS];

    // Simulate in zero thread
    thread_data_array[0].alpha = 150;
    thread_data_array[0].beta = 0.5;
    thread_data_array[0].thread_id=0;
    pthread_create(&threads[0], NULL, simulate, (void *)&thread_data_array[0]);
```

While the overall arrival process is non-Poisson each inter-arrival time is still exponentially and randomly distributed with lambda representative of the Linear Arrival Rate process. After an arrival event, the next arrival event is the current simulation time plus the Random Variable (inter-arrival time) that is generated.

```
lambda = alpha + ((QueuePackets - alpha)*beta);
Exponential_RV = - ((double)rand()/(RAND_MAX+1)) ;
Exponential_RV = -(log(Exponential_RV)/lambda);
Exponential_RV = static_cast<float>( static_cast<int>(Exponential_RV*1000) ) / 1000;
Arrival_time = simulation_clock + Exponential_RV;
```

Similarly, the time between each service event is also exponential random with mu representative of the Linear Arrival Rate process described earlier.

```
mu_k = QueuePackets * mu;
Exponential_RV = - ((double)rand()/(RAND_MAX+1)) ;
Exponential_RV = -(log(Exponential_RV)/mu_k);
Exponential_RV = static_cast<float>( static_cast<int>(Exponential_RV*1000) ) / 1000;
Service_time = simulation_clock + Exponential_RV;
```

Sampling is done at the end of every one second of simulation time. Each sample increments the corresponding value in the array. E.g. If there are 500 in the system during the time of sampling k_array[500] is incremented by 1.

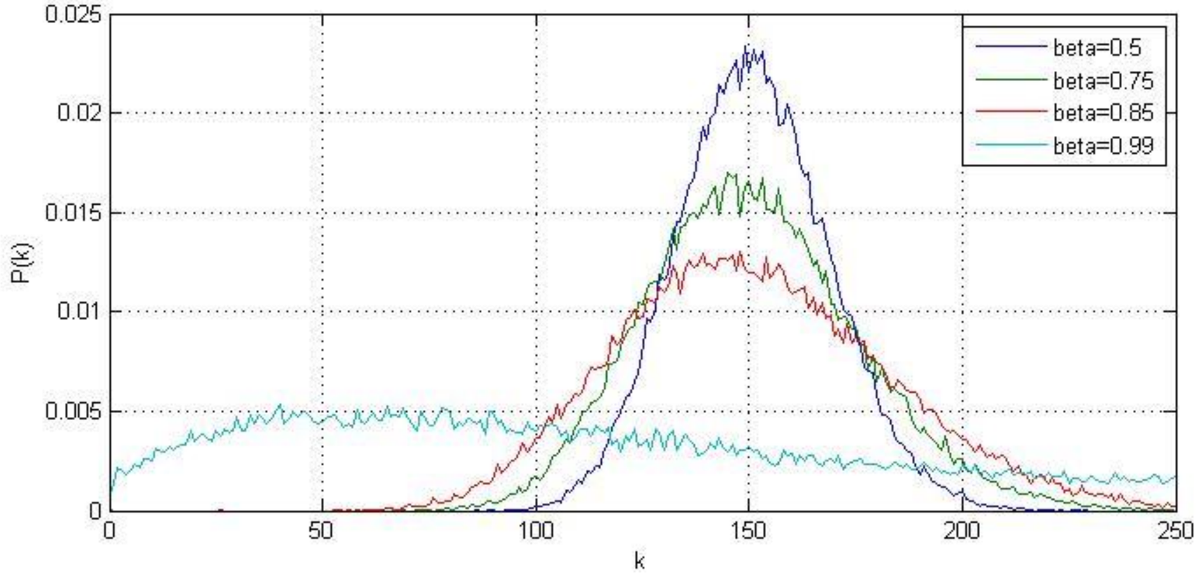
```
if(fmod(simulation_clock,1.0) < hop_time)
{
    if(processing_flag) k_array[QueuePackets + 1] = k_array[QueuePackets + 1] + 1;
    else k_array[QueuePackets] = k_array[QueuePackets] + 1;
}
// Increment Simulation Clock
simulation_clock += hop_time;
```

Check the array after the simulation. Each instance of the array should contain the number of times the sample of the system had ‘k’ objects. Since the simulation is sampled every one second, the probability values can be normalized by dividing by the number of samples = simulation_time = 50,000.

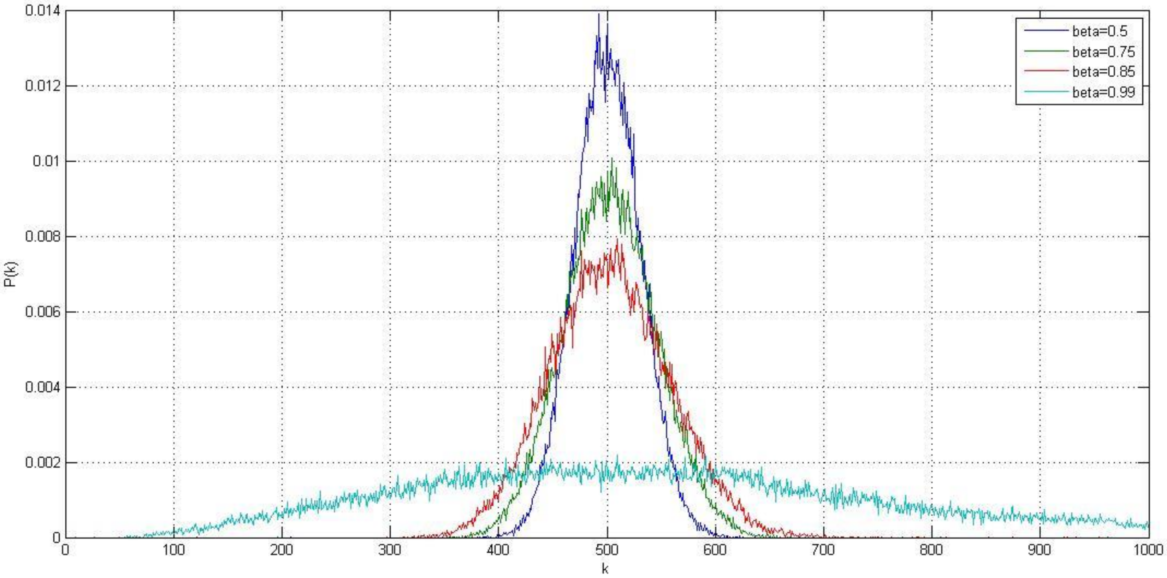
```
for(int j=0; j<Buffer_Size;j++)
{
    if(myfile.is_open()) myfile<<k_array[j]/(double)simulation_time<<endl; //Divide by
simulation time to normalize
    else cout<<"Unable to open file!"<<endl;
}
```

The P(k) values are output to a local file where they may be imported into MatLab.

Four simulation instances are created in one run with four separate threads.
α = 150 and β ranges as shown in the legend below.

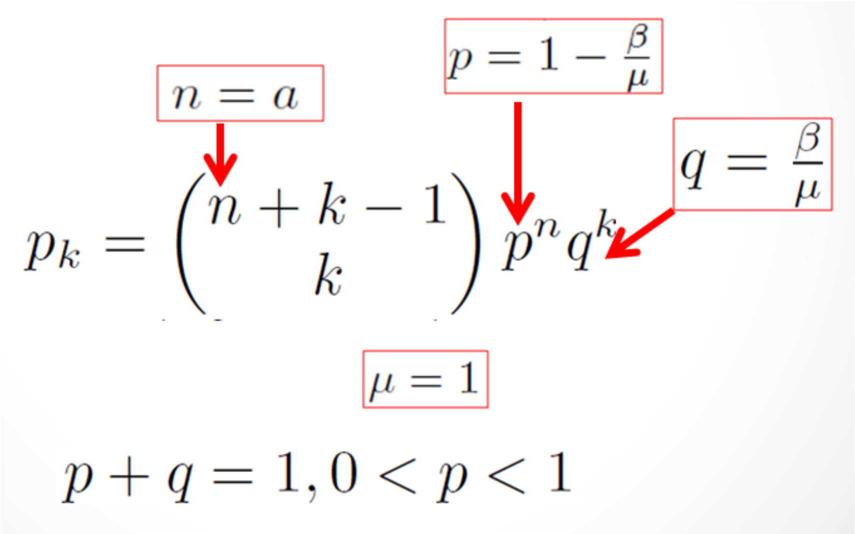


Another four simulation instances are created in a second run with four separate threads.
α = 500 and β ranges as shown in the legend below.



Calculating the Pascal Distribution

Since simulation is computation intensive, the solution can be quickly computed using approximations. The derived distribution has attributes of the Pascal (negative binomial) distribution. The approximation is the use of the Pascal distribution to arrive at the solution. The distribution representation as it applies to the Pascal distribution is:



$$\boxed{n = a} \qquad \boxed{p = 1 - \frac{\beta}{\mu}} \qquad \boxed{q = \frac{\beta}{\mu}}$$

$$p_k = \binom{n+k-1}{k} p^n q^k$$

$$\boxed{\mu = 1}$$

$$p + q = 1, 0 < p < 1$$

The parenthesis is equivalent of C(n,r) = n!/(r!(n-r)!). The distribution equation appears again here for convenience. One can compare values and will notice that they’re the same equation in a different representation.

$$\boxed{p_k = \frac{1}{\left(1-\frac{\beta}{\mu}\right)^{-a}} \left(\frac{\beta}{\mu}\right)^k \frac{a(a+1)\ldots(a+k-1)}{k!}}$$

While using the Pascal distribution reduces the computation effort, calculating the Pascal distribution brings new challenges. The Combination of ‘k’ in a set of (n + k – 1) is:

C(n + k – 1, k) = (a + k – 1)! / (k! * (a – 1)!)

In our simulation we assume up to k = 1500 busy circuits and a= 125. This means solving 1624!/(1500!124!) . The C data type – ‘unsigned long long int’ has a 64 bit precision or up to a value of 18446744073709551616. Trying to solve with factorials quickly becomes unscalable.

21! = 51090942171709440000

124! = 1.5061417415111408797950141619933e+207

1500! = 4.8119977967797748601669900935814e+4114

1624! = 5.0335115205698725070069385133486e+4510

1624!/1500! = 1.0460336295121199552237193196918e+396

One solution to managing factorials past 10^1! is using Stirling's approximation:

$$n! \sim \left(\frac{n}{e}\right)^n \sqrt{2\pi n}.$$

Although easier to compute, the solution requires much more precision than 64 bits. Additionally, we must divide numbers with this additional precision. To address the precision problem a C/C++ multiple precision library is used. The GNU Multiple precision library can downloaded, compiled and installed by following directions in the reference manual. This manual also provides guidelines on initializing and deallocating multiple precision variables as well as performing arithmetic operations on these variables. Snips of code are posted here for reference.

Also, the C++ code for calculating the Pascal Distribution is posted on GitHub.
Git link: https://github.ncsu.edu/ngkhatu/CSC777.git
Zip file: https://github.ncsu.edu/ngkhatu/CSC777/archive/master.zip

For each value of 'k' the right side of the distribution equation is calculated.

```
void calculate_right_side(double a_var, int k)
{

// Upper Values
    mpf_set_d(tmp_upper, 1.0);
    for(int k_iter=0; k_iter<=(k-1); k_iter++)
        {
            mpf_set_d(tmp_upper2, (a_var+k_iter));
            mpf_mul(upper_mult, tmp_upper, tmp_upper2);
            mpf_set(tmp_upper, upper_mult);
        }
// Lower values
        mpz_fac_ui(k_factorial, k); // Factorial of k with gmp int
        mpf_set_z(k_factorial_fp, k_factorial); // To gmp float

// Divide upper final product with with large factorial
        mpf_div(factorials_divided, upper_mult, k_factorial_fp);
```

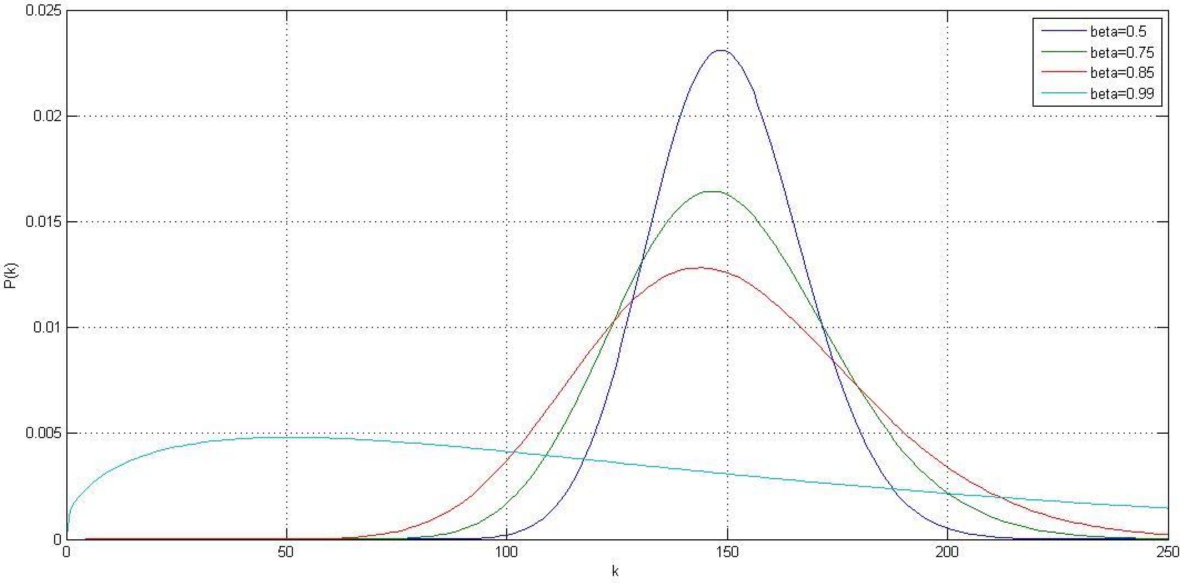
Next, for each value of k the left side of the distribution equation is calculated in this snip of code.

```
void calculate_left_side(long double a_var, double beta, int k)
{
    // (1-Beta)^a_var
    bottom_part=pow((1-beta), a_var);
    mpf_set_d(tmp_gmp_var, bottom_part);

    // Beta ^k
    mpf_set_d(beta_gmp_fp, beta);
    mpf_pow_ui(beta_k_gmp_fp, beta_gmp_fp, k);

    // Left Side
    mpf_mul(left_side_gmp, tmp_gmp_var, beta_k_gmp_fp);
```

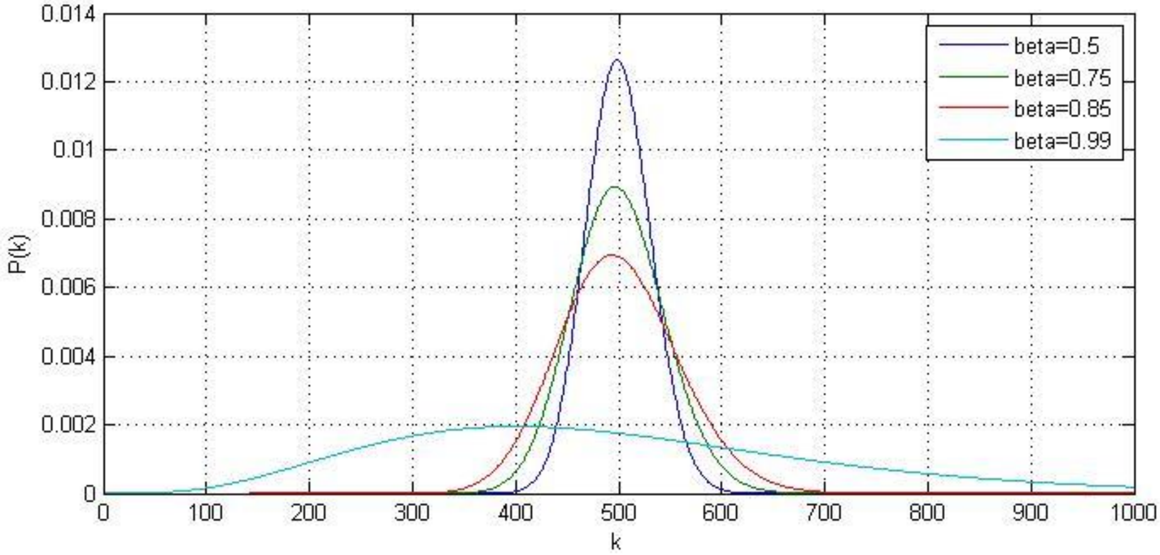
Once the left and right sides of the distribution are solved, they're multiplied to arrive at the solution. The multiple precision values are then converted to C- long double precision. The computation is repeated for each value of k.

```
void run_pascal_calc(unsigned long int alpha, double beta)
{
sum = 0;
a_var = (alpha / beta)*(1 - beta);

for(int k=1; k<1500;k++)
    {
        mpf_t final_prob_k_gmp;

        // Begin Calculating
        calculate_right_side(a_var, k);
        calculate_left_side(a_var, beta, k);
        mpf_mul(final_prob_k_gmp, left_side_gmp, factorials_divided);
        // Return GMP variable with double precision
        p_k = mpf_get_d(final_prob_k_gmp);

        // Final value
        sum += p_k;
```

Each run outputs probability values for k = 0:1:1499 for a given alpha and beta value. The data from the file is imported to MatLab to produce these graphs. Here, each of these distributions has alpha values of 150 and beta values according to the legend.



Each of the following distributions has an alpha value of 500 with beta values according to the legend.



Conclusion

Initially the model for Linear Arrival Rates was presented as a solution to modeling overflow arrivals in a secondary system. The Linear Arrival Rate distribution was then derived using Birth/Death processes. The simulation process was described along with code provided on NCSU GitHub. The distribution results from the simulation were presented here. After simulation, the Pascal Distribution was calculated and presented using the GNU multiple precision library. The distribution results from the simulation can be compared with the Pascal distribution. The results obtained from simulation runs of 50,000 seconds with a sampling rate at one per simulation second are in line with calculations from the Pascal Distribution.

We can further examine the distributions by finding the 1st and 2nd moments. The 1st moment of the distribution is the mean and the 2nd moment is the variance. These are easily calculated from the Mass Function to be:

$$E(x) = \frac{nq}{p} \Rightarrow \boxed{M = \alpha}$$
$$var(x) = \frac{nq}{p^2} \Rightarrow \boxed{V = \frac{\alpha}{1-\beta}}$$

In all instances the mean is equal to the given alpha. The variances of the results are in accordance with calculations.

	β = 0.5	**β = 0.75**	**β = 0.85**	**β = 0.99**
α = 150	**M = 150, V = 300**	**M = 150, V = 600**	**M = 150, V = 1000**	**M = 150,V =15000**
α = 500	**M = 500, V = 1000**	**M = 500, V = 2000**	**M = 500,V=3333.33**	**M = 500,V=50000**

As mentioned earlier Z is the peakedness of the distribution; it is the measure of unevenness in the arrival process. Peakedness is found by dividing the variance by the mean. Since beta is always between 0 and 1, the resulting Z values will always be greater than 1. This results in a peaked process (as opposed to a smooth process, where arrivals are poisson).

$$\boxed{Z = \frac{V}{M} = \frac{1}{1-\beta}}$$

	β = 0.5	**β = 0.75**	**β = 0.85**	**β = 0.99**
Z	**2**	**4**	**6.67**	**100**

References

[1] Andre Girard. 1990. *Routing and Dimensioning in Circuit-Switched Networks* (1st ed.). Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

[2] Wilkinson, R. I. (1956), Theories for Toll Traffic Engineering in the U. S. A.. Bell System Technical Journal, 35: 421–514. doi: 10.1002/j.1538-7305.1956.tb02388.x

1. Introduction to Balking Systems

A significant aspect in modeling call centers is customer impatience. Motivated by analyzing the call center operations, an M/M/s queuing system with impatient customers was considered. The customers arrive according to a Poisson process with rate λ, and request IID (independent and identically distributed) service times with an exponential distribution. There are $s \geq 1$ servers in the system available to serve the customers. All servers are identical and unit-rate, i.e., each server is capable of processing one unit of service requirement per unit time.

Two common modes in which customers display their impatience are balking and reneging. A call-in customer who cannot be helped immediately by a human server might be told how long a wait he/she faces before an operator is available. Then the customer might hang up (i.e. balk) or decide to hold. This is an example of the balking behavior: a customer refuses to enter the queue if the wait is too long. On the other hand, a customer who is waiting for an operator might hang up (i.e. renege) before getting served if the wait in line becomes too long. This is the reneging behavior. Additionally, a combination of the two is possible.

Queuing models with balking incorporate the characteristics of the customers' impatience or a specific admission control policy in force at a service system. In a typical queuing model with balking the service requirement of an arriving customer may not be (completely) accepted if the system is "too congested" at the time of its arrival. From the perspective of an arriving customer one natural measurement of system congestion is the queuing time he/she faces to get service started. A no-join decision based on this congestion measurement is the aforementioned wait-based balking. The model considered in this thesis uses a wait-based balking rule. Before stating the balking rule, we define the virtual queuing time (vqt) in the system. The vqt at time t in the system, denoted by W (t), is the queuing time (i.e., time spent in the system before commencing service) that would be experienced if a customer joins the system at time t. The process $\{W(t), t \geq 0\}$ is referred as vqt process. We call a queuing system with balking based on the vqt a wait-based balking queue. It works as follows. We assume that each customer knows his/her exact queuing time at the time of arrival. A customer arriving at time t joins the system if and only if W (t−) is no more than a pre-specified threshold (possibly random). The balking customers (i.e., customers who do not join) are lost forever. The entering customers wait in an infinite capacity FCFS (first-come, first-served) queue until a server is available and leave when the service completes.

The justification of such a model proceeds as follows. Although the queuing time information in call centers is not precise, this model incorporates the right characteristics of the customer behavior. Balking although depreciates response time and hence is not detrimental to the system. In the long run, however, if multiple customers balk, they can reduce the waiting queue and hence mitigate loss.

1.1 Flow of a Typical Balking System

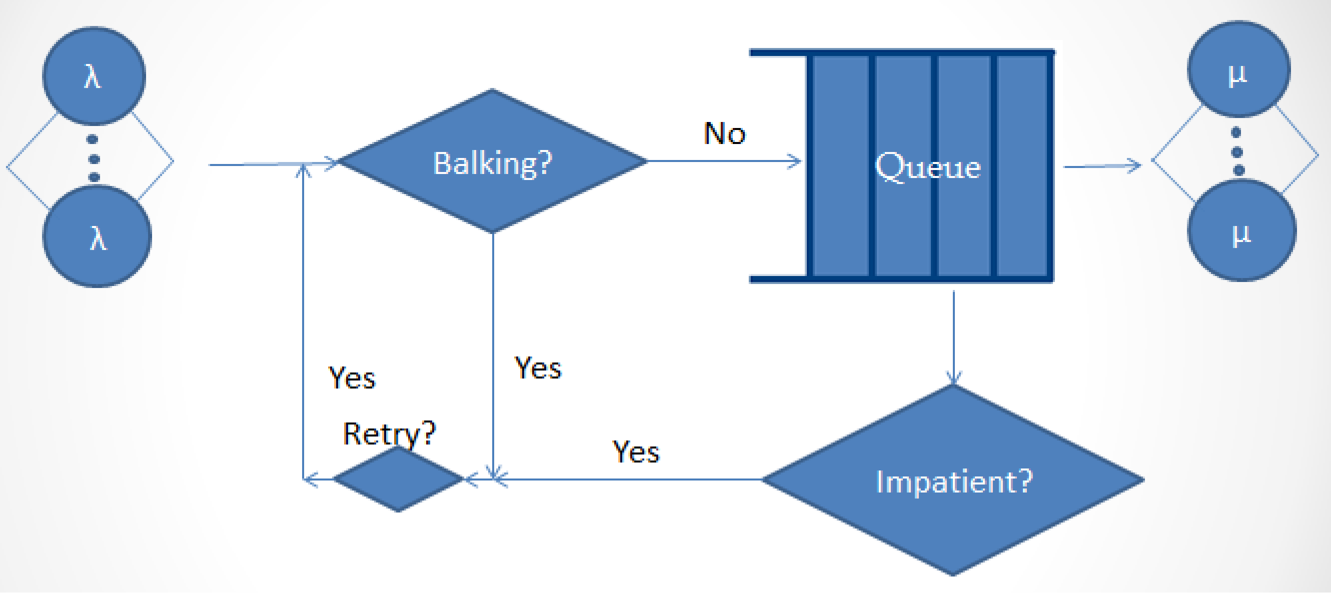


Figure 1: Flow of a typical balking system

The system assumes that calls arrive with an inter arrival rate of λ. The first flow decision is initially when the customer decides if he has to balk or not. If he decides to balk, the customer will enter the retrial queue and then retries with a retrial rate of α.The second scenario is if the customer doesn't balk and instead joins the queue, the customer might get impatient after some time and decides to renege. If the customer reneges, the customer will enter the retrial queue and then retries with a retrial rate of α. The third scenario is when the customer doesn't decide to balk or renege, but instead joins the queue and gets served with a service rate of μ. An actual balking system can be one of the aforementioned scenarios or can be a combination of the three.

2. Single Sever Queues

Single server wait-based balking queues are studied under a variety of names in the literature: "finite workload capacity", "dams", "workload-dependent arrival rates", or "queues with limited accessibility". For single server systems with a FCFS service discipline, the vqt coincides with the workload (work content) of the system, i.e., sum of the service times of all customs in queue and the remaining service time of the customer in service.

But the simulation and the presentation were done with respect to multi server queues, hence only an introduction to single server was provided for the single server queues. The mathematical analysis and simulation results have been plotted for multi-server queues as shown below.

3. Multi-Server Queues

The reneging version of the model we consider has been studied by under the name “systems with limited waiting time”. They consider exponential service times to obtain a multidimensional Markov process for the number of busy servers and workload in each server. They derive a system of integro-differential equations for the limiting joint distribution and give explicit solution.

They give formulas for the loss probability and average queueing time. They also give the limiting distribution of the vqt process. Instead, there is an alternative formula for the loss probability as a function of the tail probability of the stationary vqt process in a corresponding queue with no impatience which is provided.

The formula is exact in the M/M/s case and can be used as a heuristic for the M/G/s case. A severe restriction is that the formula is valid only when the traffic intensity is less than 1, which is not required for the reneging queue to be stable. Although we are unable to give the joint distribution for the workload and busy servers, we don’t lose much since many common performance measures can be derived directly from the limiting distribution of the vqt process.

Even in the absence of the balking behavior the M/G/s queueing system is notorious for its complexity which forbids analytical solutions. Analytical results are available for only a few special cases, while a handful of approximations for the limiting analysis have been proposed in the past decades. The system is linked with system approximations, i.e. approximations that take the results from an exact analysis of a simpler system as approximations of the true operating characteristics of the original system. Although the approximate methods vary by motivations and the techniques used, it turns out that all results can be viewed as the so-called “systems interpolation”, i.e., some mixture of the known analytical results for a few special cases, such as M/M/s, M/Ek /s, M/D/s, and M/G/∞.

To develop a system approximation for the multi-server system with impatient customers, the idea is to treat the s-server system as an M/G/∞ system (or M/G/s − 1 loss system) when some servers are idle and an M/G/1 system when all servers are busy. Using such a system decomposition we construct a single server system whose operating characteristics approximate those of the M/G/s queuing system with wait-based balking. The approximation is exact when G = M , the balking threshold is zero or s = 1. The approximation is evaluated by comparing performance measures against simulations.

3.1 Derivation for Multi-Server Queues (M-M-s)

Let P_n (t) denote the transient-state probability that there are n in the system. The differential difference equations of the system are as follows:
P_0` (t) = - λ P_0 (t) + μ P_1 (t)
P_n` (t) = - (λ+nμ) P_n (t) + λ P_{n-1} (t) + (n+1) μ P_{n+1} (t)
1<=n<=c-1
P_c` (t) = - (λp+cμ) P_c (t) + λ P_{c-1} (t) + (cμ+α) μ P_{c+1} (t) n=c
P_n` (t) = - (λp+cμ+ (n-c) α) P_n (t) + λp P_{n-1} (t) + (cμ + (n+1-c) α) P_{n+1} (t) n>c

Assuming steady state distribution exists:
0 = - λ P_0 (t) + μ P_1 (t)
0 = - (λ+nμ) P_n (t) + λ P_{n-1} (t) + (n+1) μ P_{n+1} (t) **----------- (1)**
1<=n<=c-1
0 = - (λp+cμ) P_c (t) + λ P_{c-1} (t) + (cμ+α) μ P_{c+1} (t) **-------- (2)** n=c
0 = - (λp+cμ+ (n-c) α) P_n (t) + λp P_{n-1} (t) + (cμ + (n+1-c) α) P_{n+1} (t) n>c

From (1) and (2) (in bold above), we get:
P_n= [(1/n!)(λ/μ) ^{n}] P_0 n<=c

Putting P_{c-1} and P_c and using BD process derivation equations:
P_{c+1}= [(1/c!)(λ/μ)^c (λp/cμ+α)]P_0 n=c+1
P_{c+2}= [(1/c!)(λ/μ)^c ((λp)^2/(cμ+α)(cμ+2α)]P_0
n=c+2

Generalizing for n<c, we get:

$$P_n = \frac{1}{c!}\left(\frac{\lambda}{\mu}\right)^c \frac{(\lambda p)^{n-c}}{\prod_{r=1}^{n-c}\,(c\mu + r\alpha)}\, P_0.$$

P_0 can be obtained by applying the normalizing condition:

$$P_0 = \left[\sum_{j=0}^{c} \frac{1}{j!}\left(\frac{\lambda}{\mu}\right)^j + \frac{1}{c!}\left(\frac{\lambda}{\mu}\right)^c \sum_{j=1}^{\infty} \frac{(\lambda\rho)^j}{\prod_{r=1}^{j}\,(c\mu + r\alpha)}\right]^{-1}$$

Other results obtained in the paper “On a multiserver markovian queueing system with balking and reneging” [2]:

$$N = \frac{1}{c\mu}\left[\alpha + \lambda pq \sum_{n=c}^{\infty} P_n\right],$$

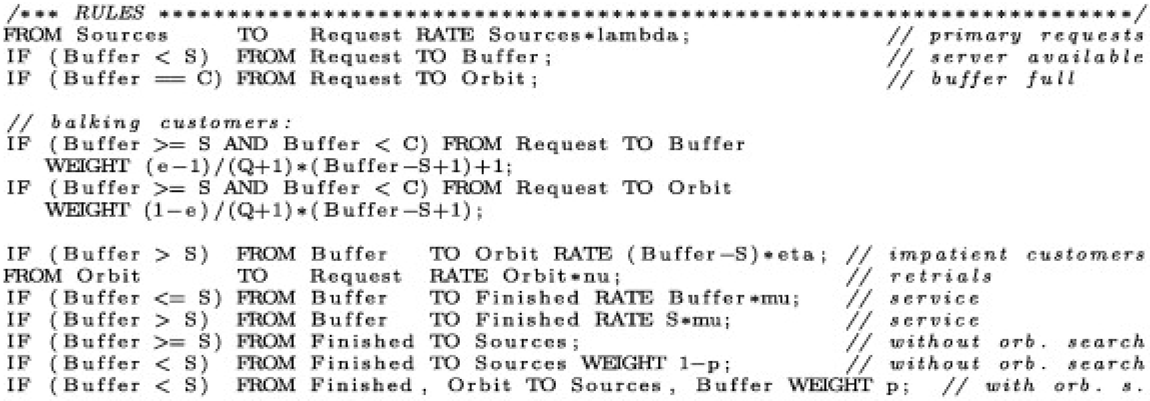
$$A = \frac{1}{c\mu}\left[\sum_{n=0}^{c} \lambda P_n + \sum_{n=c+1}^{\infty} \lambda p P_n\right].$$

Where N represents the average number of customers lost
And A represents the average number of customers attended
C is the number of servers; p is the probability of not balking

4. Simulation Results

4.1 Algorithm for simulation

The following was the algorithm followed according to “Finite-source $M/M/S$ retrial queue with search for balking and impatient customers from the orbit”[1].



```
/*** RULES ***********************************************************************/
FROM Sources      TO   Request RATE Sources*lambda;              // primary requests
IF (Buffer < S)   FROM Request TO Buffer;                        // server available
IF (Buffer == C)  FROM Request TO Orbit;                         // buffer full

// balking customers:
IF (Buffer >= S AND Buffer < C) FROM Request TO Buffer
   WEIGHT (e-1)/(Q+1)*(Buffer-S+1)+1;
IF (Buffer >= S AND Buffer < C) FROM Request TO Orbit
   WEIGHT (1-e)/(Q+1)*(Buffer-S+1);

IF (Buffer > S)   FROM Buffer   TO Orbit RATE (Buffer-S)*eta; // impatient customers
FROM Orbit        TO   Request  RATE Orbit*nu;                // retrials
IF (Buffer <= S)  FROM Buffer   TO Finished RATE Buffer*mu;   // service
IF (Buffer > S)   FROM Buffer   TO Finished RATE S*mu;        // service
IF (Buffer >= S)  FROM Finished TO Sources;                   // without orb. search
IF (Buffer < S)   FROM Finished TO Sources WEIGHT 1-p;        // without orb. search
IF (Buffer < S)   FROM Finished , Orbit TO Sources , Buffer WEIGHT p;  // with orb. s.
```

<u>In the algorithm above:</u>

- S is the number of servers
- C is the capacity of the system
- Buffer denotes the server + queue size
- Q denotes queue size
- e denotes balking probability
- mu is the service rate
- nu is the retrial rate
- eta is the impatience rate

4.2 Considerations for Algorithm

- Number of buffers were fixed at 10
- Number of servers and arrival rate was varied to obtain respective response times.
- The customer balks with a probability of 0.2
- Response time was calculated by taking inter arrival time, service time, impatient time and retrial times for all individual calls into consideration.
- All the aforementioned times are mutually independent.
- Customer reneges if the waiting time is above a particular threshold value
- Retrial inter arrival times assumed to be exponential

4.3 Simulation Results

λ- Customers/ms	3 servers Response Time (ms)	4 servers Response Time (ms)	10 servers Response Time (ms)
0.2	0.275615	0.24307	0.093043
0.3	0.517	0.46532	0.365216
0.4	1.30433	1.03064	0.930796
0.5	1.51234	1.50169	1.351521
0.6	1.75072	1.55284	1.617698
0.7	2.16763	2.07193	1.864737
0.8	2.37624	2.18843	2.0790085
0.9	2.45622	2.33869	2.2217555
1	2.54721	2.40513	2.2848735
1.2	2.64682	2.52783	2.4014385
1.4	2.75313	2.64623	2.5139185
1.6	2.83562	2.75441	2.6166895
1.8	2.94453	2.84671	2.7043745
2	3.008	2.89864	2.753708

Number of Servers =3

The results indicate that for number of servers =3, as the arrival rate is increased, it increases dramatically initially, but after a point it dips down and saturates to a value, as will be depicted by the graphs in 4.4

Number of Servers =4

Like in the case of 3 servers, for number of servers =4, the graph similarly shows an initial descent but dips and saturates. The difference in this scenario is that the response time is lower as the number of servers has been increased.

Number of Servers =10

This scenario again shows similar values to 3 and 4 servers, but the values have further dipped down as will be evident from 4.4.

4.4 Expected vs Calculated Simulation Results

The results obtained in the paper were done using a modeling language called MOSEL 2. The plot shown below shows mean response time characteristics versus request generation rate. As described in 4.3 in all the three scenarios, the response time shows an initial ascent, but saturates at a particular value.

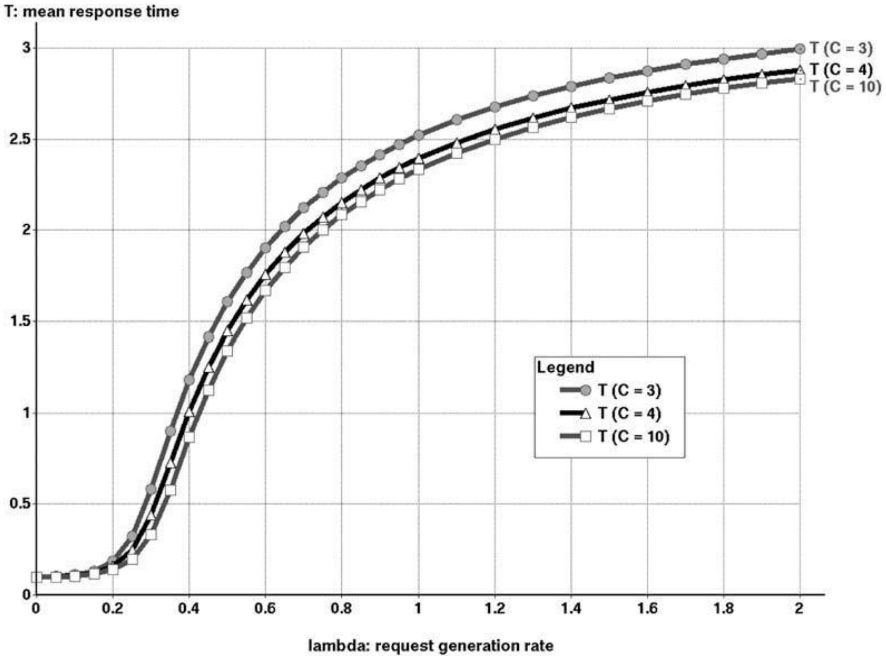


Figure 2: Mean response time versus Request Generation rate obtained in [1]

The results obtained below were obtained by simulating the algorithm given in 4.1 using C++. The marking in red are those that were obtained using simulation versus the original graph traced out in the paper above.

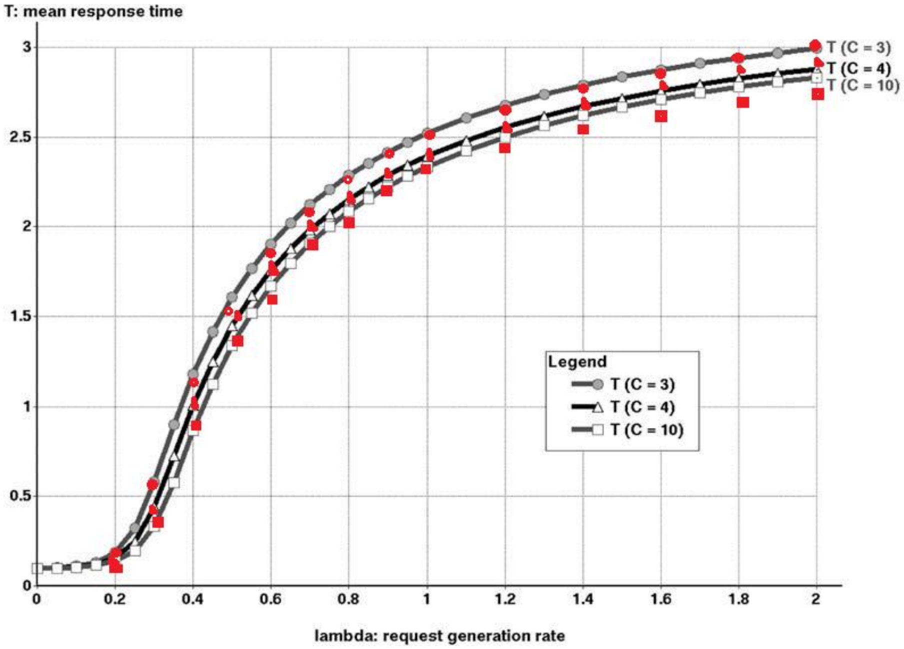


Figure 3: Mean response time versus Request Generation rate obtained in the simulation

5. Conclusion

This tutorial considers the effect of request generation rate on a *M/M/c* queuing process where customers balk with probability *n/N* (*n*=0, 1, 2, …,*N*) and renege after having waited for a random length of time. The busy period process in which transitions from any state (m, n) to any other state are permitted without an intermediate passage to the empty state, was investigated. The tutorial then contrasts the effect for a variety of scenarios varying the value of the servers between 3, 4 and 10.

6. References

[1] Finite-source *M*/*M*/*S* retrial queue with search for balking and impatient customers from the orbit (Patrick Wüchner , János Sztrik , Hermann de Meer)

[2] On a multiserver markovian queueing system with balking and reneging, (A. Montazer-Haghighi, J. Medhi, S.G. Mohanty)

[3] On the impact of customer balking, impatience and retrials in telecommunication systems, (J.R. Artalejo , V. Pla)

[4] CSC 777 Slides by Dr. Rudra Dutta