# Performance of Networked Systems 2

March 21, 2018

## 1 Dimensioning of cellular networks with video-conferencing services

A mobile operator of a cellular GSM network wants to determine how many base stations are needed to satisfy its customers' Quality of Service (QoS) demands. To this end, the operator wants to determine the maximum size of a cell for which the call-blocking probability is still below some given threshold. Voice telephone calls are generated with rate 10 calls per minute per square kilometer (i.e., km2 ), and the call duration has an exponential distribution with mean 1.5 minutes. Assume that each voice call requires a single channel to the nearest base station, and that each cell can support only 6 channels in parallel.

To make a proper decision on the number of base stations to be placed to offer good quality to its customers, the operator wants to understand the impact of the cell size (in km2 ) on the call-blocking probability.

### 1.1 Formulate a simple model description for the problem.

When describing a model we can use Kendall's shorthand notation to characterize queueing models. This notation consists of three different aspects, and one extra:

1. Interarrival time distribution
2. Service time distribution
3. Number of servers
4. Possible queueing capacity (which is 0 because there is blocking instead of queueing)

For the first two aspects we can use three letters:

- G: General distribution
- M: Exponential distribution (also called memoyless or Makov)
- D: Deterministic

Acording to this discription of Kendall's shorthand notation we can describe the model in this assignment to be of the M/M/6/0 variety, because both the interarrival times and the service time have a memoryless property.

### 1.2 Give a formula for the call blocking probability in terms of the model parameters.

#### 1.2.1 The "Erlang-B formula"

The Erlang-B formula is used for calculating the blocking probability that describes the probability of losing calls for a group of identical parallel resources (also known as M/M/c/c queue). The formula is given below:

$$\frac{\frac{(\lambda\beta)^N}{N!}}{1 + \frac{(\lambda\beta)^1}{1!} + \frac{(\lambda\beta)^2}{2!} + ... + \frac{(\lambda\beta)^N}{N!}}$$

The formula does make a few assumptions:

- It assumes a Poisson process for the arival instances, so that they are independent.
- The holding (waiting) times are exponentially distributed.
- The formula assumes an infinite population of sources (e.g. phonecalls).
- The arrival rate (lambda) is constant.
- The arrival rate is independent of the number of active sources.
- Number of resources are assumed to be infinite.
- No denied request will be kept (buffer-less loss system). So there are no queues.

### 1.2.2 Parameters

**Lamda ($\lambda$) is the rate per minute per square kilometer** lamda = 10

**Beta ($\beta$) is the exponential distribution with mean 1.5 minutes** beta = 1.5

**Amount of channels in parallel (N)** N = 6

## 1.3 Write your own 'Erlang-B calculator" and calculate the blocking probability for cell sizes 0.5 km2, 1km2 and 2 km2.

```
In [21]: from ipywidgets import interact
         import numpy as np
         import math

         from bokeh.io import push_notebook, show, output_notebook
         from bokeh.plotting import figure
         from bokeh.models import ColumnDataSource, HoverTool, LassoSelectTool, WheelZoomTool
         from bokeh.palettes import Spectral3

         output_notebook()
```

```
In [22]: # Calculate upper part of Erlang-B formula
         def bp_calc(lamda, beta, N):
             bp = (lamda * beta) ** N / math.factorial(N)
             return bp

         # Calculate Markov chain
         def erlang_B(lamda, beta, N):
             bp1 = bp_calc(lamda, beta, N)
             bp2 = 1

             for i in range(N):
                 bp2 += bp_calc(lamda, beta, i + 1)
```

```
            return bp1 / bp2
```

In [23]:
```
# Initializa data
data_x = ['0.5 km2','1 km2','2 km2']
data_y = []

# Initialize variable
datapoints = [0.5,1,2]
lamda = 10
beta = 1.5

# Fill data
for i in datapoints:
    data_y.append(erlang_B(lamda * i, beta, 6))
```

In [24]:
```
# Setting data in right format
source = ColumnDataSource(data=dict(x_axis=data_x,
                                    y_axis=data_y,
                                    color=Spectral3))

# Creating tooltip
hover = HoverTool(tooltips=[
    ("Cell Size", "@x_axis"),
    ("Blocking Probability", "@y_axis"),
])

# Create figure
p = figure(x_range=data_x, y_range=(0,(max(data_y) + (max(data_y) / 10))),
           plot_height=350, title="Erlang Blocking Calculator",
           tools=[hover,LassoSelectTool(), WheelZoomTool()])

# Create bars
p.vbar(x='x_axis', top='y_axis', width=0.9, color='color',
       legend="x_axis", source=source)

# Colors and legend
p.xgrid.grid_line_color = None
p.legend.orientation = "horizontal"
p.legend.location = "top_center"
```

In [25]:
```
# Show graph
show(p)
```

### 1.4 The call blocking probability is known to be insensitive with respect to the distribution of the call duration. What exactly does that mean?

This means that the formula is also valid for non-exponential call holding times.

### 1.5 Is the call blocking probability also insensitive with respect to the inter-arrival time distribution of the calls? If so, why, if not so, give a counter-example.

The call blocking probability is not insensitive with respect to the inter-arrival time distribution of the calls. This is because Erlang Blocking Formula requires Poisson arrival times. These Poisson arrival times have to satisfy two requirements.

1. Mutually independent
2. Exponentially distributed

So by this definition the inter-arrival time distribution has to be, by definition, exponentially distributed.

#### 1.5.1 Now suppose the service provider wants to offer a new additional service to its customers: video conferencing. Each conference call requires 3 parallel channels for each connection. Video conferencing calls arrive according to a Poisson process with rate 0.25 calls per hour per km2, and the conference call duration is exponentially distributed with mean 15 minutes. Recall that each cell has 6 channels. Call attempts are blocked when there are not enough lines available. Assume throughout that the cell size is 1 km2.

### 1.6 Formulate the evolution of the system as a two-dimensional continuous-time Markov chain, describing the numbers of calls at both call types (i.e., voice calls and video conferencing calls, respectively).

The two dimensional continuous-time Markov chain looks like figure (ref). This models follows these parameters:

- Number of types N: 2
- Number of channels C: 6
- b1, Cost of Type 1 (voice calls): 1
- b2, Cost of Type 2 (video conferencing): 3
- Lambda1: 10
- Lambda2: 0.25/60 = 0.00416
- mu1 = 1/d1 = 1/1.5 = 0.666
- mu2 = 1/d2 = 1/15 = 0.0666

In figure (ref) the probability that a video conferencing call is added to the system is depicted by lambda2 (INSERT PROPER CHAR) and the probability of a voice call being added to the system is depicted by lambda1 (INSERT). The probability that a call is finished is depicted by mu1 of 2 depending on the type, times the amount of calls of the same type.

### 1.7 Formulate and solve the balance equations for the Markov chain and calculate the equilibrium state probability for each state.

State space = {(0,0), (0,1), (0,2), (1,1), (2,1), (3,1), (1,0), (2,0), (3,0), (4,0), (5,0), (6,0)} (12 states)
Balance equations:

1. For state (0,0): *(lambda1 + lambda2) * pi(0,0) = mu1 * pi(1,0) + mu2 * pi(0,1)*

2. For state (0,1): *(lambda1 + mu2 + lambda2) * pi(0,1) = mu1 * pi(1,1) + (2 * mu2) * pi(0,2) + lambda2 * pi(0, 0)*
3. For state (0,2): *(2 * mu2) * pi(0,2) = lambda2 * pi(0,1)*
4. For state (1,1): *(lambda1 + mu2 + mu1) * pi(1,1) = (2 * mu1) * pi(2,1) + lambda1 * pi(0,1) + lambda2 * pi(1,0)*
5. For state (2,1): *(lambda1 + mu2 + 2 * mu1) * pi(2,1) = (3 * mu1) * pi(3,1) + lambda1 * pi(1,1) + lambda2 * pi(2,0)*
6. For state (3,1): *(mu2 + 3 * mu1) * pi(3,1) = lambda1 * pi(2,1) + lambda2 * pi(3.0)*
7. For state (1,0): *(lambda1 + lambda2 + mu1) * pi(1,0) = lambda1 * p(0,0) + mu2 * p(1,1) + 2* mu1 * p(2,0)*
8. For state (2,0): *(lambda1 + lambda2 + 2 * mu1) * p(2,0) = lambda1 * pi(1,0) + mu2 * pi(2,1) + 3 * mu1 * p(3,0)*
9. For state (3,0): *(lambda1 + 3 * mu1 + lambda2) * pi(3,0) = lambda1 * pi(2,0) + (4 * mu1) * pi(4,0) + mu2 * pi(3,1)*
10. For state (4,0): *(lambda1 + 4 * mu1) * pi(4,0) = lambda1 * pi(3,0) + (5 * mu1) * pi(5,0)*
11. For state (5,0): *(lambda1 + 5 * mu1) * pi(5,0) = lambda1 * pi(4,0) + (6 * mu1) * pi(6,0)*
12. For state (6,0): *(6 * mu1) * pi(6,0) = lambda1 * pi(5,0)*

In [26]:
```
import numpy

lambda1 = 10
lambda2 =  0.25/60
mu1 =  1/1.5
mu2 = 1/15

# State space = {(0,0), (0,1), (0,2), (1,1), (2,1), (3,1), (1,0), (2,0), (3,0), (4,0),
# [0, 0 ,0, 0 , 0 ,0 , 0 , 0 , 0 , 0, 0, 0 ]
e00 = [lambda1 + lambda2, mu2 ,0, 0 , 0 , 0 , -mu1 , 0 , 0 , 0, 0, 0 ]
e01 = [-lambda2, lambda1 + mu2 + lambda2 , -2 * mu2, -mu1 , 0 , 0 , 0 , 0 , 0 , 0, 0, 0
e02 = [0, -lambda2 ,2 * mu2, 0 , 0 , 0 , 0 , 0 , 0 , 0, 0, 0 ]
e10 = [-lambda1, lambda1 + lambda2 + mu1 ,0, -mu2 , 0 , 0 , 0 , -2* mu1 , 0 , 0, 0, 0 ]
e11 = [0, -lambda1 ,0, lambda1 + mu2 + mu1 , -2 * mu1 , 0 , -lambda2 , 0 , 0 , 0, 0, 0
e20 = [0, 0 ,0, 0 , -mu2 , 0 , -lambda1 , lambda1 + lambda2 + 2 * mu1 , -3 * mu1 , 0, 0
e21 = [0, 0 ,0, -lambda1 , lambda1 + mu2 + 2 * mu1 , -3 * mu1 , 0 , -lambda2 , 0 , 0, 0
e30 = [0, 0 ,0, 0 , 0 , -mu2 , 0 , -lambda1 , lambda1 + 3 * mu1 + lambda2 , -4 * mu1, 0
e31=  [0, 0 ,0, 0 , -lambda1 , mu2 + 3 * mu1 , 0 , 0 , -lambda2 , 0, 0, 0 ]
e40 = [0, 0 ,0, 0 , 0 ,0 , 0 , 0 , -lambda1 , lambda1 + 4 * mu1, -5 * mu1, 0 ]
#e50 = [0, 0 ,0, 0 , 0 ,0 , 0 , 0 , 0 , -lambda1 , lambda1 + 5 * mu1, -6 * mu1]

# Sum of all probabilities
e_normalization = [1, 1 ,1, 1 , 1 ,1 , 1 , 1 , 1 , 1, 1, 1 ]
e60 = [0, 0 ,0, 0 , 0 ,0 , 0 , 0 , 0 , 0, - lambda1, 6 * mu1 ]

matrix_param = numpy.matrix([e00, e01, e02, e10, e11, e20, e21, e30, e31, e40, e60, e_n
matrix_param_inverse = numpy.linalg.inv(matrix_param)


matrix_multiple = numpy.matrix([[0], [0], [0], [0] , [0], [0], [0], [0] , [0] , [0], [0
```

```
        matrix_probability = matrix_param_inverse * matrix_multiple
        # v00 = (lambda1 + lambda2) * pi(0,0) = mu1 * pi(1,0) +  mu2 * pi(0,1)
        # v01 = (lambda1 + mu2 + lambda2) * pi(0,1) = mu1 * pi(1,1) + (2 * mu2) * pi(0,2) + lam
        # v02 = (2 * mu2) * pi(0,2) = lambda2 * pi(0,1)
        # v10 = (lambda1 + lambda2 + mu1) * pi(1,0) = lambda1 * p(0,0) + mu2 * p(1,1) + 2* mu1
        # v11 = (lambda1 + mu2 + mu1) * pi(1,1) = (2 * mu1) * pi(2,1) + lambda1 * pi(0,1) + lam
        # v20 = (lambda1 + lambda2 + 2 * mu1) * p(2,0) = lambda1 * pi(1,0) + mu2 * pi(2,1) + 3
        # v21 = (lambda1 + mu2 + 2 * mu1) * pi(2,1) = (3 * mu1) * pi(3,1) + lambda1 * pi(1,1) +
        # v30 = (lambda1 + 3 * mu1 + lambda2) * pi(3,0) = lambda1 * pi(2,0) + (4 * mu1) * pi(4,
        # v31 = (mu2 + 3 * mu1) * pi(3,1) = lambda1 * pi(2,1) + lambda2 * pi(3.0)
        # v40 = (lambda1 + 4 * mu1) * pi(4,0) = lambda1 * pi(3,0) + (5 * mu1) * pi(5,0)
        # v50 = (lambda1 + 5 * mu1) * pi(5,0) = lambda1 * pi(4,0) + (6 * mu1) * pi(6,0)
        # v60 = (6 * mu1) * pi(6,0) = lambda1 * pi(5,0)

        prob_list = numpy.array(matrix_probability).reshape(-1,).tolist()
        print ("Equilibrium state probability for each state: \n \
               {(0,0), (0,1), (0,2), (1,1), (2,1), (3,1), (1,0), (2,0), (3,0), (4,0), (5,0), (6

        print (matrix_probability, "\n")
        # 1 only accept
                        #{(0,0), (0,1), (0,2), (1,1), (2,1), (3,1), (1,0), (2,0), (3,0), (4,0),
                        #{(0,0), (0,1), (1,0), (2,0), (3,0)}
        b1 = 1 - (prob_list[0] + prob_list[1] + prob_list[3] + prob_list[4] + prob_list[6] + pr
        b2 = 1 - (prob_list[0] + prob_list[1] + prob_list[6] + prob_list[7] + prob_list[8])

        print (b1)
        print (b2)
```

```
Equilibrium state probability for each state:
       {(0,0), (0,1), (0,2), (1,1), (2,1), (3,1), (1,0), (2,0), (3,0), (4,0), (5,0), (6,0)}

[[-1.65059210e-04]
 [ 1.45875334e-06]
 [ 4.55860421e-08]
 [ 2.30587955e-05]
 [ 1.82422573e-04]
 [ 9.21913715e-04]
 [-2.47677389e-03]
 [ 1.24846572e-03]
 [ 1.94550288e-02]
 [ 8.28732337e-02]
 [ 2.56553202e-01]
 [ 6.41383004e-01]]

0.6423049632545225
0.9819368798599208
```

### 1.8 Use the product-form theorem discussed during the lecture of March 5 to give an explicit formula for the equilibrium state probabilities of the Markov chain.

$K = 2 $ call classes
load $ ro_k = lambda_j * d_j $ (k = 1,...,K)
$n_k$ := number of class-k calls in progress (k = 1,...,K)
$\pi_n = 1/G \prod_{j=1}^{K} \frac{\rho_j^n j}{n_j}$
Where:
$G := \sum$

```
In [27]: import math
         import numpy
         K = 2
         states = [(0,0), (0,1), (0,2), (1,1), (2,1), (3,1), (1,0), (2,0), (3,0), (4,0), (5,0),
         lamda = [10, 0.25/60]
         d = [1/1.5, 1/15]

         G_calc = []
         pf = [[],[]]
         equilibrium = []

         for state in states:
             for j in range(K):
                 ro = lamda[j] * d[j]
                 G_calc.append(ro ** state[j] / math.factorial(state[j]))

         G = numpy.product(G_calc)

         for state in states:
             for j in range(K):
                 ro = lamda[j] * d[j]
                 pf[j].append(ro ** state[j] / math.factorial(state[j]))

         for p in pf:
             equilibrium.append((1/G) * numpy.product(p))

         print(equilibrium, numpy.sum(equilibrium))
```

```
[4.3535646720000016e+21, 1.6964906514146546e-14] 4.3535646720000016e+21
```

### 1.9 Use this result to calculate the blocking probabilities for both call classes.

### 1.10 Formulate the Kaufman-Roberts recursion for the model.

```
In [28]: def getBlockProbabilityForKClass():
             b = (1, 3) # Tuple of required channels for type of connection
             d = (1.5, 15) # Tuple of call duration
```

```
    lamda = (10, 0.25/60) # arrival rate
    p = (lamda[0] * d[0], lamda[1] * d[1]) # load

    C = 6 # Number of channel
    K = 2 # Number of class

    g = [None] * (C+1)
    g[0] = 1

    # Calculate g array
    for c in range(1, C+1):
      gc_temp = 0
      for j in range(K):
        g_cb_index = c-b[j]
        if (g_cb_index >=0):
          gc_temp += b[j] * p[j] * g[g_cb_index]

      gc_temp = gc_temp * 1/c
      g[c] = gc_temp


    # Calculate G_Constant
    G = sum(g)


    # Equilibrium distribution for the number of occupied resources
    q = [None] * len(g)
    for c in range(C+1):
      q[c] = g[c]/G

    # Blocking probability result
    B = [None] * K
    for j in range(K):
      B_temp = 0
      for c in range(C-b[j]+1, C+1):
        B_temp += q[c]
      B[j] = format(B_temp, '.3f')

    return B

  blockProbabilityArr = getBlockProbabilityForKClass()

  print('====== Block probability for class=======')
  for index in range(len(blockProbabilityArr)):
    print ('Class ', index, ': ', blockProbabilityArr[index])
```

```
====== Block probability for class=======
Class  0 :  0.634
```

```
Class  1 :  0.972
```

## 2 Three cell analysis

Consider a cellular voice network with three neighboring cells, as depicted below. The mean number of call attempts per minute for cells 1, 2 and 3 are 5, 10 and 15, respectively. We assume that the mean call duration is 2 minutes. There are a total of 32 channels, and there is a fixed number of channels per cell. However, to avoid interference neighboring cells cannot use the same channel.

**2.1 Determine the distribution of the 32 channels over the three cells that minimizes the call blocking probability of an arbitrary call (regardless of the cell in which takes place. To this end, extend your Erlang-B calculator to the three-cell case, and sure it calculates the call blocking probabilities per cell and the overall call blocking probability.**

**2.2 Suppose we want to have an overall call blocking probability less than 5%. Do we need any additional channels? If so, how many additional channels are needed, and what would the optimal allocation of these channels then be?**

In [29]:
```
import math

def try_backtrack(i, triple, result_subsets, space, total):
    for index in range(len(space)):
        triple[i] = space[index]

        if (i == (len(triple)-1) and sum(triple) == total):
            copy_tripe = list(triple)
            if copy_tripe != []:
                result_subsets.append(copy_tripe)

        if (i != len(triple) - 1):
            try_backtrack(i + 1, triple, result_subsets, space, total)


def generateAllSubsetSumToN(space, n , total):
    subsets = [[]]
    backtrack_triple = [None] * n
    try_backtrack(0, backtrack_triple, subsets, space, total)
    return subsets

####### Erlang model
lamda1 = 5
lamda2 = 10
lamda3 = 15
lamda_sum = lamda1 + lamda2 + lamda3
```

```
beta = 2
lamda1_average = (lamda1)/lamda_sum
lamda2_average = (lamda2)/lamda_sum
lamda3_average = (lamda3)/lamda_sum

# Calculate upper part of Erlang-B formula
def bp_calc(lamda, beta, N):
    bp = (lamda * beta) ** N / math.factorial(N)
    return bp

# Calculate Markov chain
def erlang_B(lamda, beta, N):
    bp1 = bp_calc(lamda, beta, N)
    bp2 = 1

    for i in range(N):
        bp2 += bp_calc(lamda, beta, i + 1)

    return bp1 / bp2

######## Collect all probability of arbitrary in 3 cells
space = list(range(1, 33))
subsets = generateAllSubsetSumToN(space, 3, 32)
subsets = [x for x in subsets if x != []]

nsubset = len(subsets)
prob = [None] * nsubset
for index in range(nsubset):
    subset = subsets[index]
    prob[index] = lamda1_average * erlang_B(lamda1, beta, subset[0]) + lamda2_average *

min_prob = min(prob)
print("Min probability of blocking: ", min_prob)
set_index_for_min_prob = prob.index(min_prob)
print("Number of channels to minimize blocking prob of three cells: ", subsets[set_inde
```

```
Min probability of blocking:  0.5057064834151516
Number of channels to minimize blocking prob of three cells:  [3, 10, 19]
```