ECMM424 Coursework

Darren Gichuru Gitagama

21/11/2024

Abstract

Signed: Darren Gitagama

Contents

1 M/M/C/C

The M/M/C/C is a queuing system characterised by exponentially distributed (memoryless) inter-arrival and service times, C servers, with finite service capacity - making it blocking in nature. Therefore event procedures must be designed such that if all servers are busy, the job is forever lost in the system, rather than being stored in a buffer/queue. To simulate behaviour of this system, we outline the high level logic of program using diagram and pseudocode below, where its main characterising feature is the lack any waiting queues.

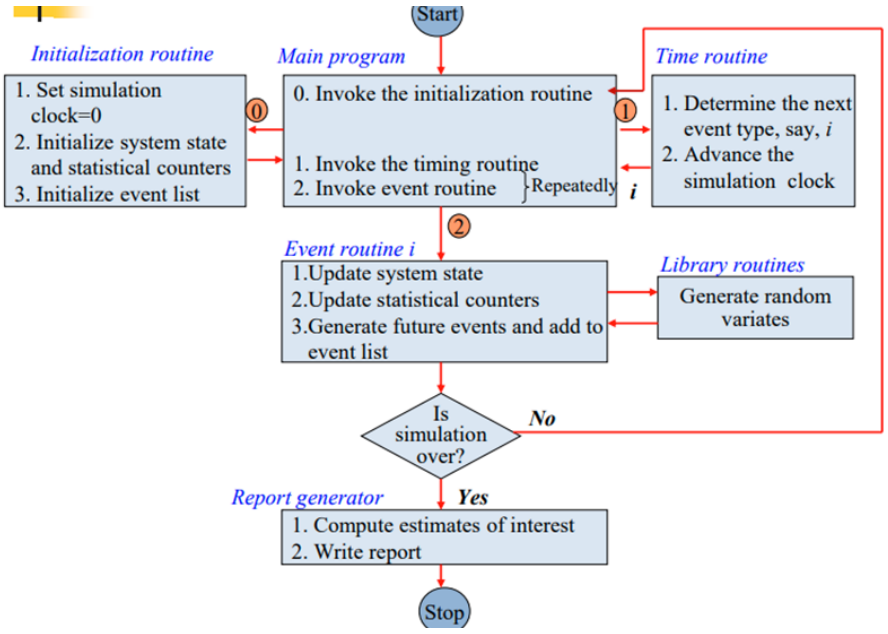

Figure 1: Flow chart of program design

```
Main()                  // executes simulation logic
    Init()              // initialise state, global variables, statistical counters, and event
    while (simulation not complete)
        Timing()       // Determines the next event and updates simulation clock
        Update_stats() // updates statistical counters

        if (event type is arrival)
            Arrive() // Handles arrival events
        else if (event type is departure)
            Depart() // Handles departure events
    Report()         // Generates a report with metrics of interest
```

1.1 Program Code

we use Python Classes to implement the system, being particularly beneficial for simplifying the maintainability and testability of the system as it allows us to bundle related functions together. This makes analysis easier, where objects can be instantiated independently with different parameter configurations enabling more efficient testing. In accordance to the figure 1, we use the following class attributes:

```
class mmcc:
    """

    """
    def __init__(self, arrival_rate, call_duration, channels, iterations):

        # attributes
        self.arrival_rate = arrival_rate
        self.call_duration = call_duration
        self.channels = channels
        self.iterations = iterations
        self.sim_time = 0
        self.total_arrivals = 0
        self.time_of_last_event = float("inf")

        # events
        self.events = [float("inf") for _ in range(self.channels + 1)]
        self.events[0] = self.sim_time + np.random.exponential(1 / self.arrival_rate)
        self.server_status = [ 0 for _ in range (self.channels)]

        # statistical counters
        self.server_usage = [ 0 for _ in range (self.channels)]
        self.blocked_events = 0
        self.all_data = []
```

Figure 2: Class initialisation of state, global and statistical variables

In our system, events are represented using an array, `events` of size C + 1, where C is number of servers. Index 0 will always be an arrival event and index 1 – n-1 are departure events. We initialise the contents of this array to infinity, indicating the absence of any scheduled event. This is necessary given that the timing function picks the next event based on the minimum element within an array. The next scheduled arrival is generated from an exponential distribution with a mean-inter-arrival times of 1/lambda, where lambda is arrival rate. A separate array is used to monitor the status of each server (`server_status`), where any server in the array is deemed idle if server[i] == 0, and 1 if busy. In a similar fashion we monitor server utilisation using an array (`server_usage`), where each index in the array will indicate the utilisation of its corresponding server. Utilisation is updated after each iteration by an amount equivalent to the time since last event, given that the server is busy at time of calculation. To ensure adherence to "finite capacity" property of mmcc queue, upon the arrival of an event we first need to check if any servers are free. This involves looping through the server status array and checking for the existence of a 0 (indicating idle). If none are found, we can increment the blocking counters which maintain count of number arrival events blocked. Departure events are performed by resetting server status of server i to 0 and the corresponding event to infinity.

```
"""
handles arrival events
    - schedules next arrival
    - assigns job to server if free otherwise increments block counter
"""
def arrive(self):
    # schedule the next arrival
    self.total_arrivals += 1
    self.events[0] = self.sim_time + np.random.exponential(1 / self.arrival_rate)

    # find if any server is idle, if so set to busy and schedule departure
    any_idle = False

    for i in range(self.channels):
        if self.server_status[i] == 0:
            self.server_status[i] = 1
            self.events[i + 1] = self.sim_time + np.random.exponential(self.call_duration)
            any_idle = True
            break

    self.blocked_events += 1 if any_idle == False else 0
```

Figure 3: Arrival method

1.2 Simulation Testing and Validation

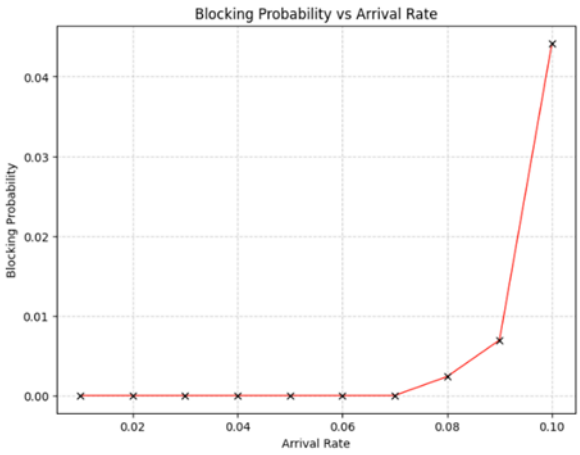

Figure 4: Blocking probabiliy vs arrival rate

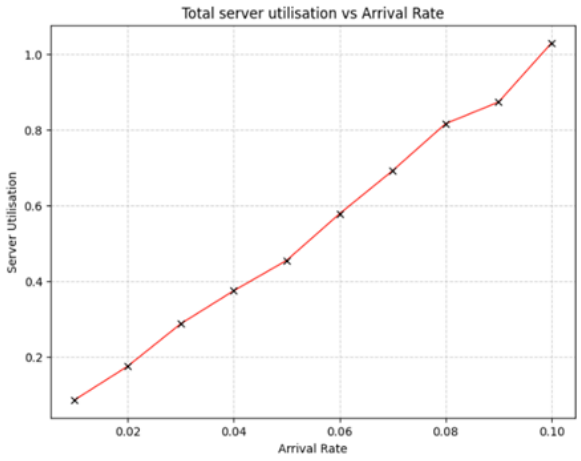

Figure 5: Total server utilisation vs arrival rate

The following plots (fig 4, fig 5) are generated after running a simulation for 10000 seconds, where the left diagram models blocking probability against arrival rate, and the right depicts server utilisation against arrival rate. Looking at the former, we notice what appears to resemble a function of an exponential curve, where for low lambda values, blocking probability remains at 0 until lambda = 0.7, at which point a steep increase is observed.

This behaviour can be validated using an analytical model in which we calculate the blocking probability mathematically using the Erlang B formula (as defined as follows):

$$B = \frac{\frac{A^C}{C!}}{\sum_{k=0}^{C} \frac{A^k}{k!}}$$

Where:

- B = blocking probability.

- N = number of servers (channels).
- A = traffic intensity (in Erlangs)

Blocking probability B defines the likelyhood of an event being blocked due to insufficient capcity in the system, or lack of resources. For instance, $B = 0.01$ suggests 1% of the arrival events will be denied service. When traffic intensity$A$ is smsll, it is indicative of low server utilisation and free capacity in the system. This however changes rapidy as A approaches/exceeds C. In this case the values for the numerator and denomintor increase sharply, reflecting the case where servers are frequently busy/congested therefore far more likely to block an incoming arrival. Although the graph depicts the blocking probability as remaining at zero for values of λ between 0 and 0.7, it is, in fact, increasing rapidly by several orders of magnitude. The graph appears flat because it lacks the sensitivity to capture these minute but dramatic changes. Using the Erlang B Formula we implement an analytical model with the code shown in figure 6, validating the behaviour of our simulation, with results being depicted in fig 7:

```
"""
calculates blocking given number of servers
E: traffic intensity
m: Number of servers (C)

"""
def ErlangB(E, m=16):

    InvB = 1.0
    for j in range(1, m + 1):
        InvB = 1.0 + InvB * (j / E)
    return (1.0 / InvB)

estimated_blocking = []
for lamda in arrivals:
    rho = lamda / channels * call_duration
    b = ErlangB(rho)
    estimated_blocking.append(b)
```

Figure 6: Erlang B formula

Figure 6 shows a recursive implementation of the Erlang B formula which can be explained mathematically by equations 1,2 where 1 is the base case and 2 is the recursive case:

$$\frac{1}{B(E,0)} = 1. \tag{1}$$

$$\frac{1}{B(E,j)} = 1 + \frac{j}{E} \cdot \frac{1}{B(E,j-1)} \quad \forall j = 1, 2, \ldots, m. \tag{2}$$

Furthermore, our testing shows a linear relationship between server utilisation and arrival rate. Similar to the blocking probability case, this behaviour is to be expected as when λ is low, less servers are likely to be in use, therefore utilisation is low (vice versa). We can explain the linear relationship mathematically also as p = $\frac{\lambda}{c\mu}$, where C and μ are constants. Therefore p is directly proportional to λ and will grow linearly according to arrival rate. The diagrams below are plots generated using our analytical model, and further validate our results by exhibiting the same behaviours.

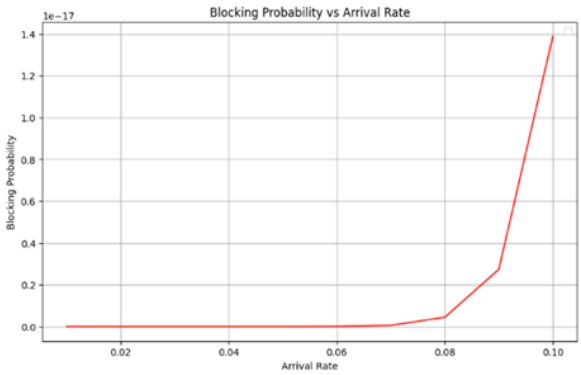

Figure 7: Blocking probabiliy vs arrival rate

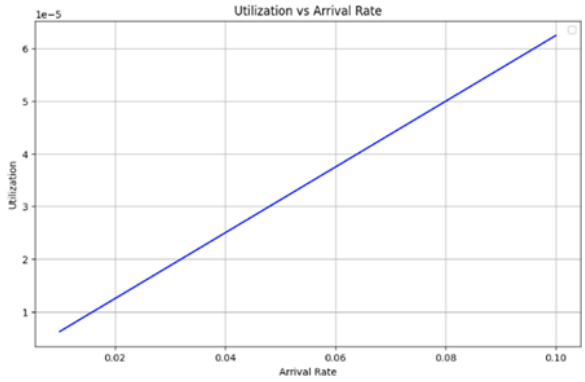

Figure 8: Total server utilisation vs arrival rate

1.3 Find the maximum value for arrival rates so that the Blocking Probability < 0.01.

```
arrivals = np.linspace(0.01, 1, 100)

call_duration = 100 # average service time
total_duration = 10000 # num of iterations
channels = 16
results = []
usage = []
mn = float("-inf")

for i in range(len(arrivals)):

    print(arrivals[i])
    obj = mmcc(arrivals[i], call_duration, channels, total_duration)
    obj.main()
    block_prob = obj.report()
    if block_prob < 0.01:
        mn = arrivals[i]

    usage.append((sum(obj.server_usage)/total_duration)/10)
    results.append(block_prob)

print(f"MAXIMUM λ < 0.01 : {mn}")
```

Figure 9: Code snippet used to test for maximum arrival rates < 0.01

In order to find the max λ where blocking probability is < 0.01, we create a function (shown in fig 9) to create different instances of the mmcc queue with different λ values. As we iterate through the instances, we calculate their blocking probability and update the `mn` variable - should its value be less than 0.01. Therefore given t iterations, `mn` is always the largest possible value for λ whilst satisfying the constraint. It is important to note that the mmcc queue is fundamentally stochastic, where scheduled arrival and departure events are memoryless and random. Therefore every run of the simulation will generate a different optimal solution. In response, we conduct multiple trials and calculate an average to determine the average maximum value and the λ values we use are generated from a uniform distribution between 0.01-0.1. This way we ensure our results are more reliable and repeatable. Fig 10 shows a subset of these results after 100 trials, after which the maximum seen was

0.09996402314660618 and the average maximum is 0.09558131503309837.

```
MAXIMUM λ < 0.01 : 0.09747104283495833
MAXIMUM λ < 0.01 : 0.09649007764158091
MAXIMUM λ < 0.01 : 0.09888117377829096
MAXIMUM λ < 0.01 : 0.09408534133781027
MAXIMUM λ < 0.01 : 0.09801182988084857
MAXIMUM λ < 0.01 : 0.09639667479032034
MAXIMUM λ < 0.01 : 0.09861513966173527
MAXIMUM λ < 0.01 : 0.0924215838843335
MAXIMUM λ < 0.01 : 0.09547714790510213
MAXIMUM λ < 0.01 : 0.09444387928718719
MAXIMUM λ < 0.01 : 0.09839202811776593
MAXIMUM λ < 0.01 : 0.09039893084931505
MAXIMUM λ < 0.01 : 0.0986683797418129
MAXIMUM λ < 0.01 : 0.09392290179766442
MAXIMUM λ < 0.01 : 0.09245190216135071
 MAX SEEN λ : 0.09996402314660618, AVG MAX: 0.09558131503309837
```

Figure 10: Maximum and average maximum arrival rate results

2 M1+M2/M/C/C

2.1

The M1+M2/C/C graph is conceptually very similar to the M/M/C/C, being a loss system with C servers and exponentially distributed (memoryless) inter-arrival and service times. The key differentiator comes from the use of multiple arrival stream, thereby requiring modifications to the `events`, `server_status` arrays to accommodate this additional arrival information. We can assign these events as follows:

- `events[0]` = exp(λ) // new call events
- `events[1]` = exp(λ) // handover events
- `events[2, ..., C-1]` = exp(μ) // departure events

```python
def __init__(self, call_rate, handover_rate, call_duration, channels, iterations, threshold):

    self.call_rate = call_rate
    self.handover_rate = handover_rate
    self.call_duration = call_duration

    self.channels = channels
    self.iterations = iterations
    self.sim_time = 0
    self.time_of_last_event = float("inf")

    self.total_arrivals = 0
    self.new_call_arrival = 0
    self.handover_arrival = 0
    self.threshold = threshold


    # idx 0 = new call event, idx1 = handover call event
    self.events = [float("inf") for _ in range(self.channels + 2)]
    self.events[0] = self.sim_time + np.random.exponential(1 / self.call_rate) # exponential of interarrival times
    self.events[1] = self.sim_time + np.random.exponential(1 / self.handover_rate) # exponential of interarrival times
    self.server_status = [ 0 for _ in range (self.channels)]

    # statistical counters
    self.server_usage = [ 0 for _ in range (self.channels)]
    self.new_call_bloked = 0
    self.handover_blocked = 0
    self.all_data = []
```

Figure 11: M1+M2/M/C/C initialisation method

At a high level, the methods used in both M/M/C/C and M1+M2/M/C/C systems are the same, following a process as defined in fig 1, with key methods: `arrive()`, `depart()`, `Timing()`, `update_stats()`, `main()`, and `report()`. However, since we use 2 streams we can enforce a priority policy that places preference on one arrival over another. This is controlled using a threshold variable `self.thersholds` which outlines a base capacity required in order for new calls to use resources. This is implemented in the code snippet shown in fig 12, where to assign a new call to a server, we first check the number of servers free. If the amount found is above the threshold, then a new call can be assigned to a server, else it is blocked. These restrictions are not placed on handover calls, where as long as at least one server is free, it is guaranteed to be serviced.

```
def arrive(self, event_idx):
    # schedule the next arrival
    self.total_arrivals += 1
    self.new_call_arrival += 1 if event_idx == 0 else 0
    self.handover_arrival += 1 if event_idx == 1 else 0

    lamda = self.call_rate if event_idx == 0 else self.handover_rate
    self.events[event_idx] = self.sim_time + np.random.exponential(1 / lamda)

    # find if any server is idle, if so set to busy and schedule departure
    def assign_server():
        for i in range(self.channels):
            if self.server_status[i] == 0:
                self.server_status[i] = 1
                self.events[i + 2] = self.sim_time + np.random.exponential(self.call_duration)
                break

    # check how many servers are free
    num_of_active_servers = self.server_status.count(1)
    free_servers = self.channels - num_of_active_servers

    if free_servers == 0:

        self.new_call_bloked += 1 if event_idx == 0 else 0
        self.handover_blocked += 1 if event_idx == 1 else 0

    # else there are some free servers
    else:
        if event_idx == 0:
            if free_servers > self.threshold:
                assign_server()
            else:
                self.new_call_bloked += 1
        else:
            assign_server()
```

Figure 12: Arrive method to handle new call or handover arrivals

2.2 Find the maximum value for the handover rate so that the Aggregated Blocking Probability (ABP) < 0.02

To find the maximum value for the handover rate so that the Aggregated Blocking Probability (ABP) < 0.02 we define the function handover_rate_trial() (as seen in fig 13), which iterates through different handover rates and calculates their respective ABP, CBP and HFP. The handover rate values selected from a uniform distribution ranging from 0.01 - 0.1, thereby providing broad coverage without excessive number of runs. We conduct multiple runs of this function and calculate averages in response to the inherent randomness in the system.

```python
"""
this function finds the maximum value for handover rate where ABP < 0.02
Returns: max handover rate
"""
def handover_rate_trial():

    call_rate = 0.1
    handover_rate = np.sort(np.random.uniform(0.01, 1, 100))
    call_duration = 100
    servers = 16
    iterations = 10000
    threshold = 2
    minimum = float("-inf")

    for i in range(len(handover_rate)):

        obj = m1m2cc(call_rate , handover_rate[i] , call_duration, servers, iterations, threshold)
        obj.main()
        cbp, hfp = obj.report()
        abp = cbp + (10 * hfp)
        if abp < 0.02:
            minimum = handover_rate[i]


    print(f"MAXIMUM HANDOVER RATE WHERE ABP< 0.02 : {minimum}")
    return minimum

num_trials = 10
avg = 0
mx = 0
for i in range(num_trials):
    t = handover_rate_trial()
    avg += t
    mx = max(mx, t)

avg /= num_trials
print(f" MAX SEEN λ : {mx}, AVG MAX: {avg}")
```

Figure 13: functions defined to find the maximum value for the handover rate so that the Aggregated Blocking Probability (ABP) < 0.02

```
MAXIMUM HANDOVER RATE WHERE ABP< 0.02 : -inf
MAXIMUM HANDOVER RATE WHERE ABP< 0.02 : -inf
MAXIMUM HANDOVER RATE WHERE ABP< 0.02 : -inf
MAXIMUM HANDOVER RATE WHERE ABP< 0.02 : -inf
MAXIMUM HANDOVER RATE WHERE ABP< 0.02 : -inf
MAXIMUM HANDOVER RATE WHERE ABP< 0.02 : -inf
MAXIMUM HANDOVER RATE WHERE ABP< 0.02 : -inf
MAXIMUM HANDOVER RATE WHERE ABP< 0.02 : -inf
MAXIMUM HANDOVER RATE WHERE ABP< 0.02 : -inf
MAXIMUM HANDOVER RATE WHERE ABP< 0.02 : -inf
```

Figure 14: `handover_rate_trail()` results

When observing the results (fig 14) we notice that such a value for handover rate does not exist. This behaviour can be explained mathematically where the equations used to calculate aggregated blocking probability are defined as follows:

$$\text{Call Blocking Probability (CBP)} = \frac{\text{total\_new\_call\_loss}}{\text{total\_new\_call\_arrivals}} \tag{3}$$

$$\text{Handover Failure Probability (HFP)} = \frac{\text{total\_handover\_loss}}{\text{total\_handover\_arrivals}} \tag{4}$$

$$\text{Aggregated Blocking Probability (ABP)} = \text{CBP} + 10 \times \text{HFP} \tag{5}$$

In our case, call rate remains constant at 0.1, so we can expect the lowest value for CBP to occur at the beginning where there is the most free capacity. As handover rate increases, however, the share of higher priority tasks entering system will also increases, meaning that new call events are for more likely to be blocked with free server capacity becoming increasingly scarce and the threshold for accepting new calls becomes harder to attain. HFP behaves differently, where given its priority status, they will always be access to a server (provided its free). Therefore, HFP increases at a much slower rate as handover rate increase HFP . This happens because HFP the denominator (total_handover_arrivals) increase much faster than numerator the numerator, leading to slower convergence. In other words, the ratio of blocked handover calls to total handover arrivals continues to decreases, however this trend occurs only until the system becomes congested (shown in fig 13, where the HFP curve increase at a much slower rate than CBP), where at this point, if majority of arrivals are priority the "priority" status becomes irrelevant and essentially assumes normal queue behaviour. Looking at the some results from the simulation, when handover_rate is small = 0.0008559346307064342, HFP = 0 but CBP = 0.05 and ABP = 0.05, showing that whenever handover rate is very low, ABP approximately equals to CBP. As a result we can not accheive a result better than minimum value of CBP, demonstrating its infeasible to for ABP to fall below 0.02 for any handover rate.

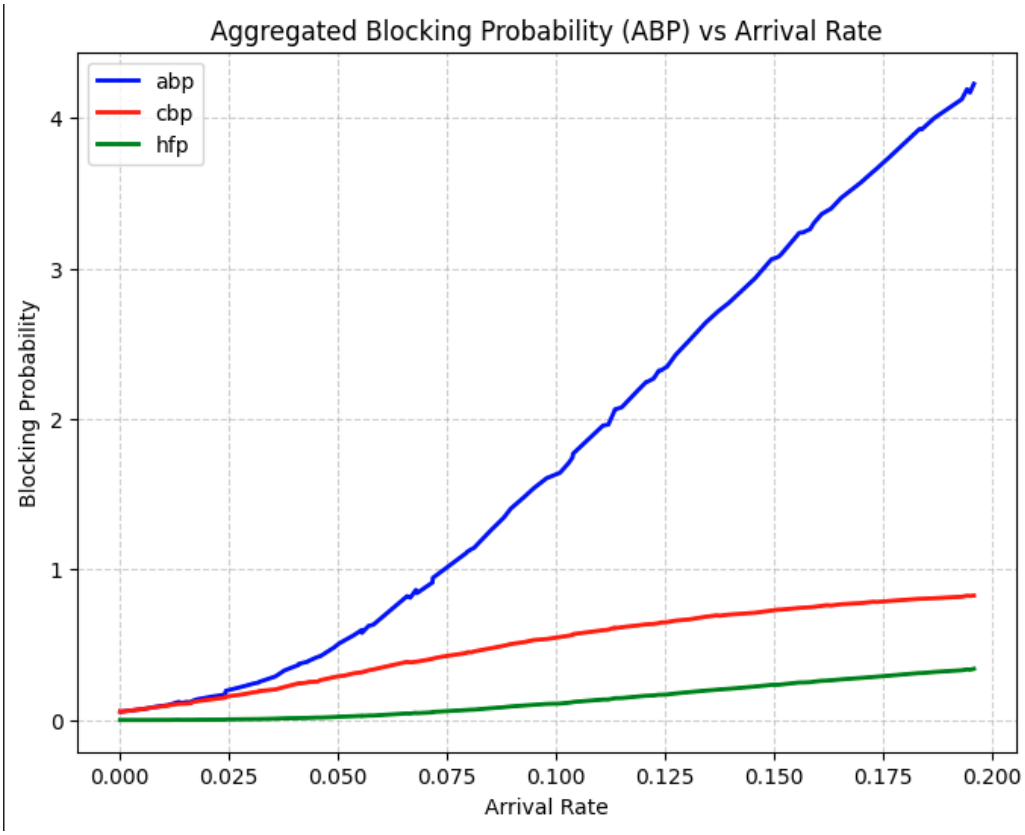

Figure 15: HFP, CBP, and ABP vs handover arrival rate

2.3 Find the maximum value for the new call rate so that the ABP < 0.02

To find maximum value for the new call rate so that the ABP < 0.02, we will use an approach very similar to that discussed in 2.2, where instead handover rate is constant at 0.03 and we iterate through different call rate values. After running 10 iterations of the simulation (10000 seconds) we found the the maximum value for the new call rate to be 0.066, however on average the max was found was approximately 0.56 (as seen in the figure 16)

```
MAXIMUM HANDOVER RATE WHERE ABP< 0.02 : 0.0665374437260139
MAXIMUM HANDOVER RATE WHERE ABP< 0.02 : 0.04909449902951585
MAXIMUM HANDOVER RATE WHERE ABP< 0.02 : 0.05533067658154661
MAXIMUM HANDOVER RATE WHERE ABP< 0.02 : 0.05379344020263699
MAXIMUM HANDOVER RATE WHERE ABP< 0.02 : 0.05877616702118861
MAXIMUM HANDOVER RATE WHERE ABP< 0.02 : 0.062085109539583766
MAXIMUM HANDOVER RATE WHERE ABP< 0.02 : 0.05343509023635294
MAXIMUM HANDOVER RATE WHERE ABP< 0.02 : 0.05147205690558643
MAXIMUM HANDOVER RATE WHERE ABP< 0.02 : 0.05275073096133385
MAXIMUM HANDOVER RATE WHERE ABP< 0.02 : 0.061272049682511334
 MAX SEEN λ : 0.0665374437260139, AVG MAX: 0.056454718267355264
```

Figure 16: `call_rate_trail()` results for 10 iterations

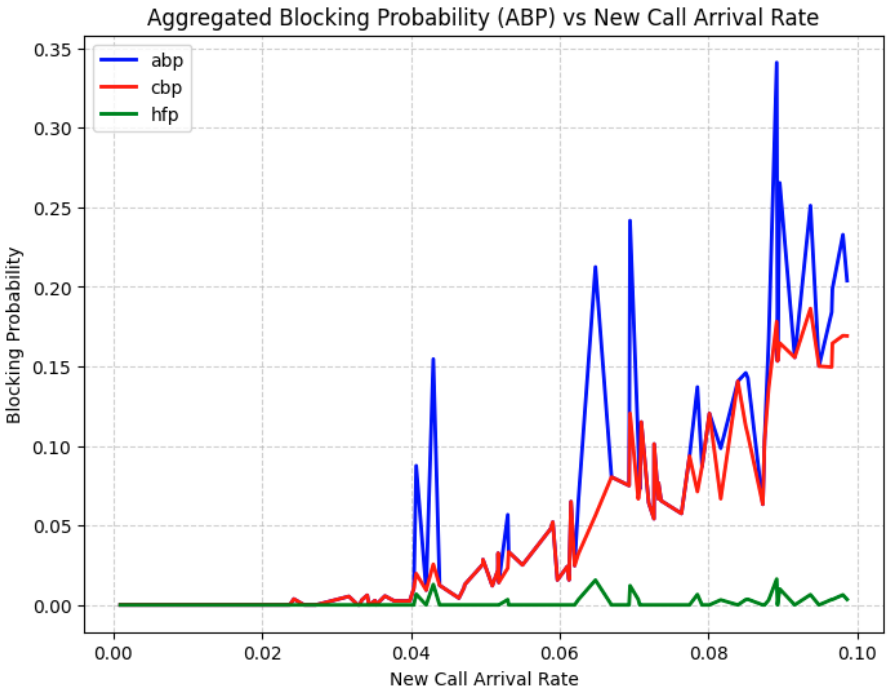

Figure 17: HFP, CBP, and ABP vs handover arrival rate

Fig 17 shows a plot of ABP against new call arrival rate, where (similar to 2.2 and 1.2), we use 100 rates selected from uniform distribution between 0 - 0.1. We notice several behaviours from this plot:

- firstly, HFP remains relatively constant - fluctuating mildly around 0 - 0.025. This behaviour is similar, in nature, to what we have seen in 2.2, whereby placing priority on handover calls means they always will have access to any free server.
- CBP however, increases accordingly with rate, though with some volatility. This means that as its rate increases the share of arrivals being new calls will also increase. This makes the system congested with new call events, where server capacity is reduced and blocking is far more likely to occur.
- The fluctuations start to occur when the system hits thresholds where more calls will get blocked thus free up more servers, reducing competition and creating a dip in ABP, CBP and or HFP. Secondly we observe the impact of the weightings placed on HFP, whereby in regions such as $\lambda = 0.4$ or $\lambda = 0.65$, we see massive spikes in ABP which mirror the patterns of HFP.