Project: Simulation of G/G/c Queues

Queueing Analysis & Simulation

Bart Middag
Master of Science in Computer Science Engineering
Academic year 2015-2016

1. PARAMETER CHOICES

Multiple performance measures were studied for different parameter ranges in order to gain a good understanding of the 4 different scheduling disciplines.

1.1 Parameter range

The gamma distribution of the interarrival times as well as that of the service times accepts two parameters: k and θ. Because the mean of a gamma distribution is simply $k\theta$ and the variance $k\theta^2$, it is not difficult to calculate the parameters k and θ using the mean and variance as follows:

$$k = \frac{mean^2}{variance}$$

$$\theta = \frac{variance}{mean}$$

The queueing systems were thus tested for different means ($0.95 * c$, c and $1.05 * c$) but the same variance for the distribution of service times only. Note that we simply used a mean and variance of 1 for the interarrival times: only the difference between both distributions really matters for our study. We used the case where the mean service time is $1.05 * c *$ the mean interarrival time especially as this clearly shows the strengths and weaknesses of each scheduling discipline.

Additionally, if the amount of servers c is larger than 1 and we want to simulate similar behavior as with fewer servers, both the mean and the variance of the service times are multiplied with c. The mean is multiplied because when picking c samples from the original service time distribution, the mean of the sum is c times the original mean. Likewise, the variance is multiplied because when picking c samples from the distribution used for 1 server, the variance of the sum is c times the original variance.

By default, we measured the behavior for queues with $c = 8$ servers. To summarize, the customers' service times have a mean of mean $0.95 * c$, c and $1.05 * c$, and variance c. Their interarrival times have mean and variance 1.

1.2 Observation times

The observation process is a separate Poisson process, i.e. with exponentially distributed interarrival times (with $\lambda = 1$). This was chosen as to uphold the PASTA property (Poisson arrivals see time averages). Note that we also included code for observing the state of the system at customer arrivals and departures, but decided not to use the performance measures derived from these observations.

2. ANALYSIS

Multiple performance measures were studied in order to gain a good understanding of the 4 different scheduling disciplines.

2.1 Queue content (and system content) vs. simulation time

One of the more simple performance measures is the queue content. Figure 1 shows trajectories of the queue content for the 4 different scheduling disciplines when the mean service time is c times the mean interarrival time. The high variance is due to the distributions being balanced in terms of the mean, though with high variance in both service and interarrival times.

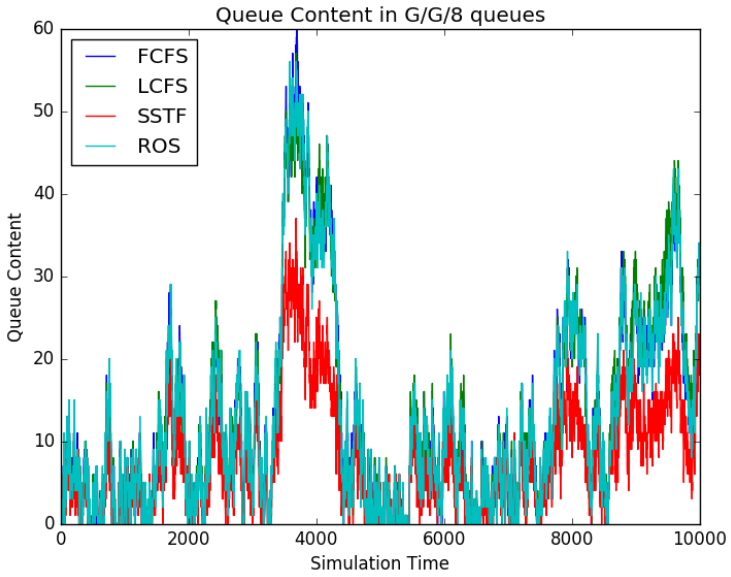

Figure 1: Queue content (mean service time = $c *$ mean interarrival time)

This figure does not tell much about the difference between scheduling disciplines, except that SSTF causes a lower queue content at all times, as it serves more customers since it always serves the customer with the lowest service time first. This is clearer in Figure 2, where the mean service time is slightly higher than $c *$ the mean interarrival time.

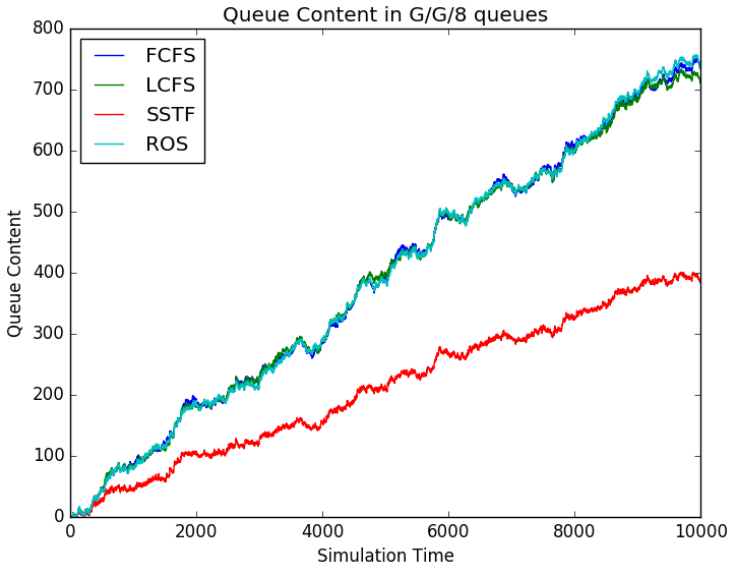

Figure 2: Queue content (mean service time = $1.05 * c *$ mean interarrival time)

Note that the system content, due to it being just the queue content as well as the customers in service, results in nearly identical graphs, though we included Python code for this too.

2.2 Waiting time (and sojourn time) vs. simulation time

Figure 3 shows trajectories for the waiting times of customers *in the queue* for the different scheduling disciplines when the mean service time is c times the mean interarrival time. The spikes in the plot are due to the variance in both distributions, though it is clear that some scheduling disciplines allow the waiting time of customers to climb higher: LCFS of course does this because it only serves the first customers in the queue if there are no other customers – therefore, the first customers have to wait indefinitely. SSTF instead lets the customers with the highest service times wait indefinitely, similarly causing spikes in the trajectory. However, in case of SSTF, the spikes are much lower than for LCFS because more customers are served more quickly than when using LCFS. This behavior is confirmed in Figure 4.

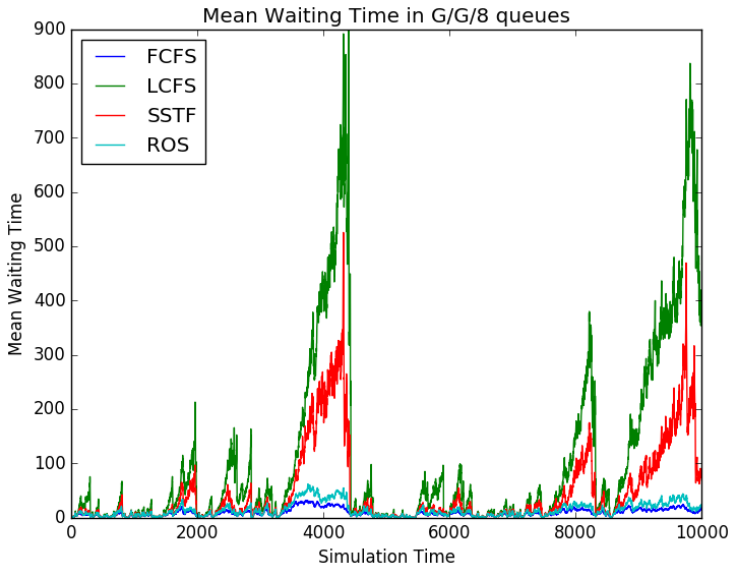

Figure 3: Mean waiting time of customers in queue (mean service time = $c *$ mean interarrival time)

Note that it was specifically mentioned that this is the mean waiting time of customers in the queue. There are different ways to visualize the waiting time of customers, which can tell us more about differences in scheduling disciplines. To demonstrate the differences between the queues as best as possible, we only show plots for simulations where the mean service time is higher than $c *$ the mean interarrival time.

Figure 5 shows that while LCFS and SSTF perform badly for customers in the queue, the customers that it actually serves do not wait very long at all in comparison to FCFS and ROS. However, it also shows that once customers stop arriving and the customers that have been kept waiting in the queue are finally served, the mean waiting time of served customers increases immensely.
In general, the higher the mean waiting time in the queue (Figure 4), the lower the mean waiting time for served customers (Figure 5): preferring certain customers will always

make others wait longer. The reason why ROS remains close to FCFS in this regard is due to the fact that there are many more newly arrived customers in the queue than customers that have been waiting for a long time (arrival rate > combined service rate).

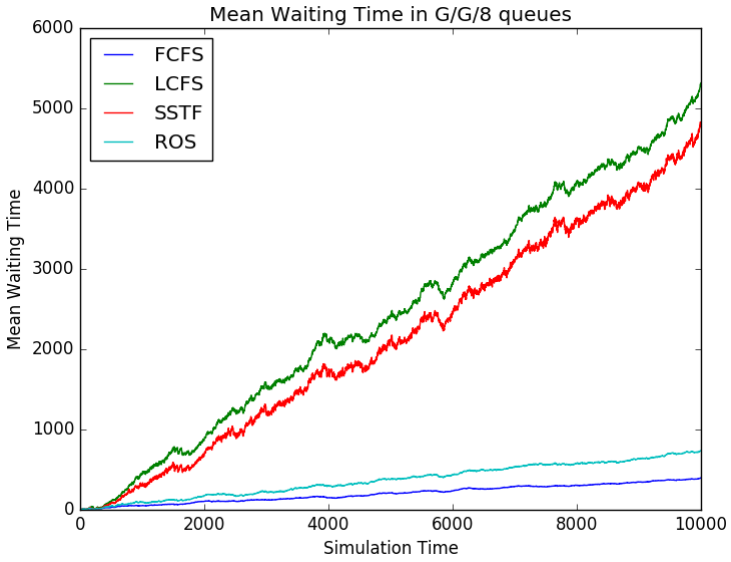

Figure 4: Mean waiting time of customers in queue (mean service time = $1.05 * c *$ mean interarrival time)

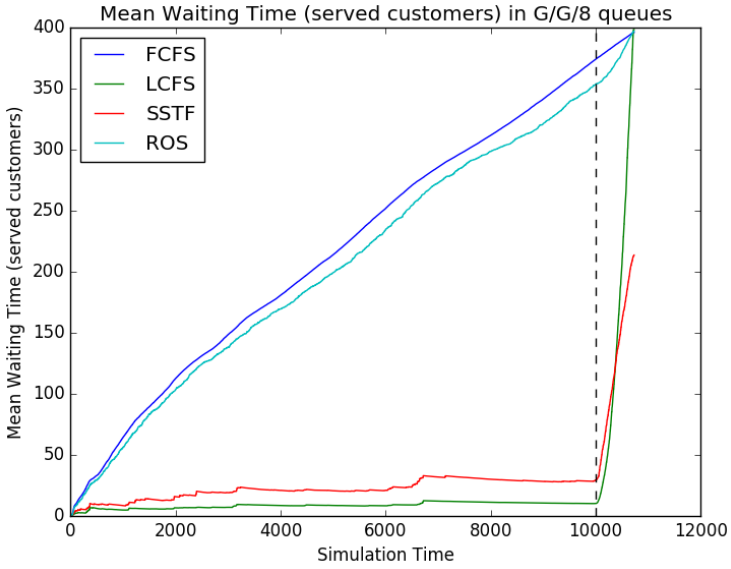

Figure 5: Mean waiting time of all customers that have left the system
(mean service time = $1.05 * c *$ mean interarrival time)
The dashed line shows when customers stop arriving to the system.

Until now, we have observed the mean waiting time, either of customers in the queue or customers that have left the system. However, it is also interesting to study localized values of the customers' waiting time. Figure 6 shows the waiting time of customers as they leave the system. It is here that we can clearly see the difference between FCFS and ROS for the waiting times of customers, which we could not see on any of the previous figures. While the mean waiting time is similar for both scheduling disciplines, the variance of the waiting time using ROS is much larger, due to random customers being served: the customer could be from the beginning of the queue and have a really

large waiting time, or the end of the queue and have a really short waiting time. This is not the case for FCFS since customers are served in order.

Unlike the previous figure, Figure 6 does not show what happens when customers stop arriving into the system (this is to show the difference between ROS and FCFS in waiting times more clearly). When we do plot this in Figure 7, it is shown that another difference between two scheduling disciplines can be explained using this localized view of waiting times. The difference between SSTF and LCFS is similar to the difference between ROS and FCFS: the variance in waiting times using SSTF is also much higher than when using LCFS, while the mean value is similar (higher for LCFS).

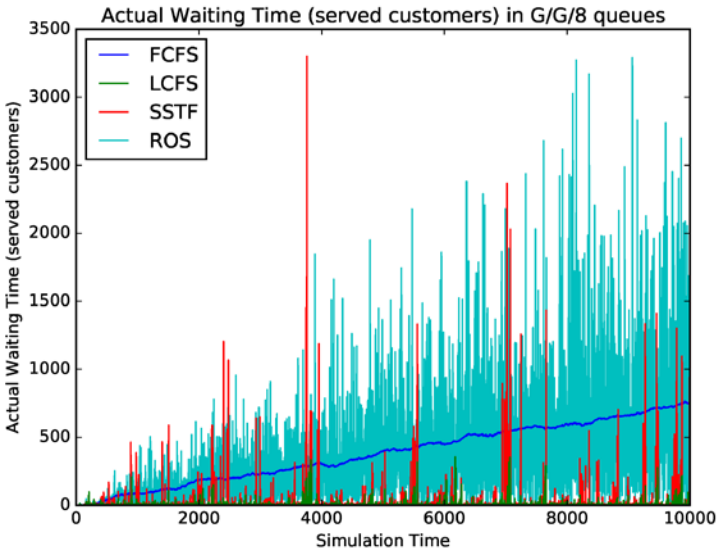

Figure 6: Localized waiting time of customers that have left the system (mean service time = $1.05 * c *$ mean interarrival time). More specifically, this is the mean waiting time of customers that have left the system since the last observation.

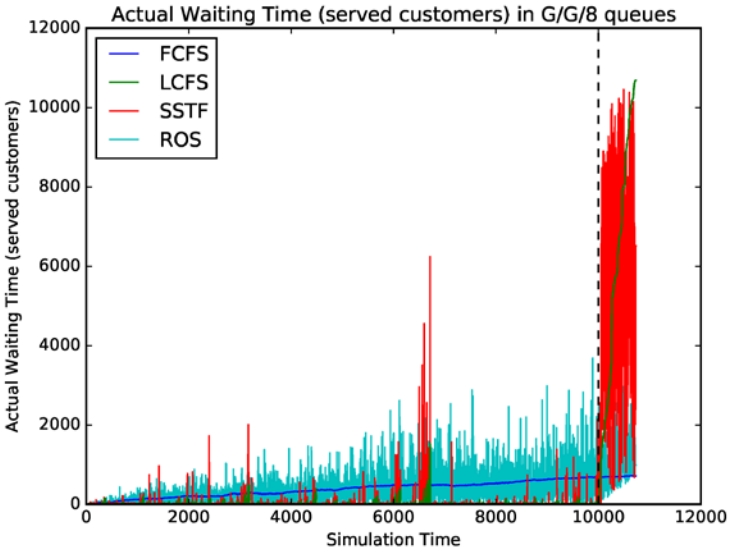

Figure 7: Localized waiting time of customers that have left the system, without forced stop of the simulation (mean service time = $1.05 * c *$ mean interarrival time)

Next to waiting times, the same observations can be made for sojourn time. Although we are not using these plots in this document as waiting times and sojourn times are identical apart from the inclusion of service times, we have included this in the Python code.

2.3 Queue content and waiting time vs. mean service time

While the analysis using simulation time already shows much of the differences between the behaviors of the different scheduling disciplines, it is also interesting to analyze them from a different perspective: using system parameters, such as the mean service time.

The following plots were achieved by running different simulations using a simulation period of 4000 (in simulation time) and a burn-in period of 1000. The data points on the plots are the mean of all of the observations of that performance measure in the simulation corresponding to the mean service time (from $0.75 * c$ to $1.25 * c$).

For the queue content, we can make the same observation as before: SSTF attempts to keep the queue as small as possible. However, we now also notice that the queue size expands at a lower rate than with the 3 other scheduling disciplines when increasing the mean service time.

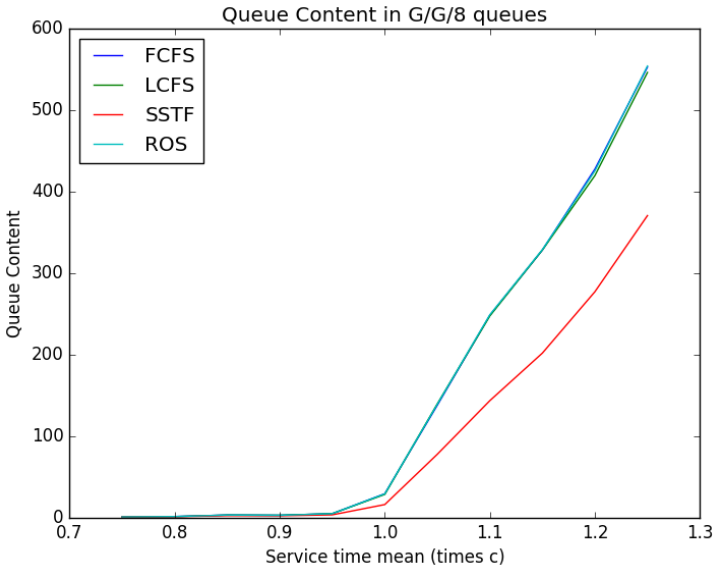

Figure 8: Mean queue content for different mean service times

For the mean waiting time for customers in the queue, as illustrated in Figure 9, we see a very large difference for LCFS and SSTF between the mean service times 0.95 and 1.05. This is due to the fact that the system can serve its customers faster than they arrive below that point, but they arrive faster than the system can serve beyond that point. Since these two scheduling disciplines effectively prefer some customers over others (either based on how late they arrived or how long they will need to be serviced), the other customers will be left waiting indefinitely, greatly increasing the mean waiting time.

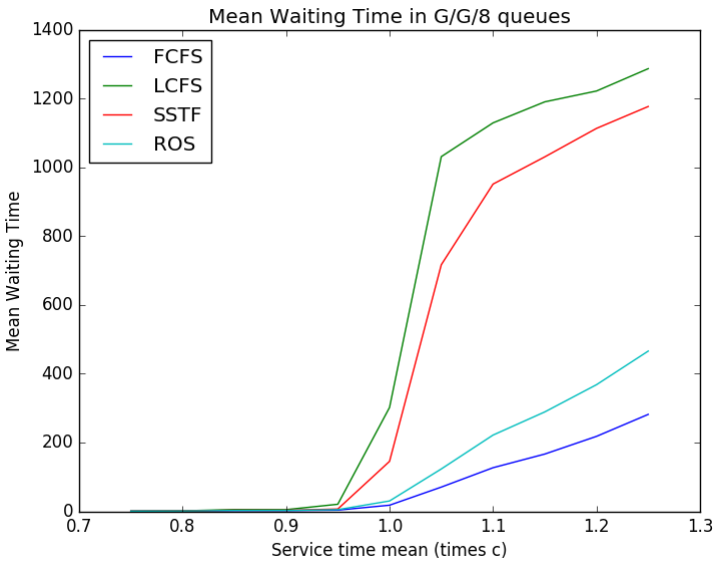

Figure 9: Mean waiting time of customers in the queue for different mean service times

It is also noteworthy that the curve for ROS rises at a slightly faster rate than that of FCFS. With increasing load of the system, more customers will wait in the queue for a long time and while FCFS will always serve those who have waited the longest, ROS will not, causing it to be possible that a customer will wait indefinitely, but also that a customer will be served upon arrival. Outliers, or customers who wait for an especially long time, will pull the mean waiting time up in this case, causing the mean waiting time for ROS to rise faster with increasing mean service time. The median waiting time would not increase as strongly.

2.4 Batch means for mean waiting time (in queue)

Many of the figures shown have been for a faster interarrival rate than the server rate. However, for cases where the interarrival rate and service rate are equal or where the interarrival rate is slower than the service rate (i.e. the Markov chain is ergodic), we can use the batch means method to estimate the variance of these performance measures and obtain a confidence interval. Python code to determine the confidence interval of any estimator is included, but we have opted to show only that of the mean waiting time of customers in the queue, for mean service time $= 0.95 * c *$ mean interarrival time.

For the following results, we have simulated 1000 batches of size 20, which is actually not sufficient, but this is necessary to keep the computation time acceptable.

Discipline	Batch mean	Variance	99% Confidence Interval
FCFS	5.59279	25.12675	[-7.33987, 18.52545]
LCFS	43.88086	5233.22346	[-142.75898, 230.52070]
SSTF	19.67214	835.41761	[-54.89913, 94.24340]
ROS	8.21327	69.61387	[-13.31294, 29.73948]

These results can be improved using more and bigger batches and the autocorrelation of the batches is high with a batch size this small, but these values clearly show the differences in variance between waiting times using different scheduling disciplines.

3. PYTHON CODE

This section contains our Python code. Some of the code is omitted/compacted.

```
from random import gammavariate, expovariate, uniform
from math import sqrt
from collections import deque

# -------- CONFIG --------
amountOfServers = 8

meanS, varS, meanA, varA = amountOfServers*0.95, amountOfServers, 1.0, 1.0
kS, tS, kA, tA = meanS**2/varS, varS/meanS, meanA**2/varA, varA/meanA

obsL = 1.0 # random observation points: exp distribution lambda
observeArrivals = False
observeDepartures = False
verbose = False
maxSimulationTime = 10000
forceStop = True # halt simulation immediately or just stop customers from arriving?

mode = "simulate" # set to systemParams or batchMeans for other behavior
sys_simulationTime, sys_burnin = 4000, 1000
sys_meanS = [0.75, 0.8, 0.85, 0.9, 0.95, 1.0, 1.05, 1.1, 1.15, 1.2, 1.25]

M = 20 # batch size - rather small but huge computation time otherwise
L = 1000 # batch amount - see above
k_conf = 2.58 # for 99% confidence interval

# -------- INIT CODE (GLOBAL) --------
def init():
    global firstSimulation, arrivals, obsTimes, observations, batchDict
    firstSimulation = True
    arrivals = []
    obsTimes = []
    observations = {}
    observations["random"] = {}
    << same for arrival and departure observations >>
    if mode == "batchMeans":
        batchDict = {}

init()
schedulingDisciplines = ["FCFS", "LCFS", "SSTF", "ROS"]
obsVariables = [["queueContent", "Queue Content"], ["systemContent", "System Content"],
    ["meanWaitingTime", "Mean Waiting Time"], ["meanSojournTime", "Mean Sojourn Time"],
    ["unfinishedWork", "Unfinished Work"], ["newWaitingTime", "Actual Waiting Time (served customers)"],
    ["newSojournTime", "Actual Sojourn Time (served customers)"],
    ["meanCompleteWaitingTime", "Mean Waiting Time (served customers)"],
    ["meanCompleteSojournTime", "Mean Sojourn Time (served customers)"]]

# -------- INIT CODE (SIMULATION) --------
def reset():
    global agenda, continueDES, currentTime, queue, servers, done
    global doneSinceLastObsRandom, doneSinceLastObsArrival, doneSinceLastObsDeparture
    global observationsRandom, observationsArrival, observationsDeparture
    global currentL, currentM, batches, batchMeans
    agenda = {}
    continueDES = True
    currentTime = 0
    queue = deque([])
    servers = []
    done = []

    doneSinceLastObsRandom, doneSinceLastObsArrival, doneSinceLastObsDeparture = [], [], []
    observationsRandom, observationsArrival, observationsDeparture = [], [], []
    currentL, currentM, batches, batchMeans = 0, 0, [], []

# -------- HELPER CLASSES --------
class Server:
    def __init__(self, discipline):
        self.discipline = discipline
        self.content = None

    # Should only be called when the server is done serving a customer
    def depart(self):
        if(self.content is not None):
            if observeDepartures == True:
                observe("departure")
            done.append(self.content)
            doneSinceLastObsRandom.append(self.content)
            if observeArrivals == True:
                doneSinceLastObsArrival.append(self.content)
            if observeDepartures == True:
                doneSinceLastObsDeparture.append(self.content)
            self.content = None

    def serve(self, customer):
        self.content = customer
        customer.timeEnteredService = currentTime
        customer.waitingTime = currentTime - customer.arrivalTime
        customer.sojournTime = customer.waitingTime + customer.serviceTime
        schedule(currentTime + customer.serviceTime, departure, [self])

class Customer:
    def __init__(self, arrivalTime, serviceTime):
```

```
        self.arrivalTime = arrivalTime
        self.serviceTime = serviceTime

class Observation:
    def __init__(self, time):
        self.time = time

# -------- HELPER METHODS --------
def getFreeServer():
    for i in range(0, len(servers)):
        if servers[i].content is None:
            return servers[i]
    return None

def serviceTime():
    return gammavariate(kS, tS) # python's gammavariate pdf uses (k, theta) parameters instead of (alpha, beta)

def arrivalTime():
    return gammavariate(kA, tA)

# -------- SCHEDULING --------
<< Literal copy of page 8.29 in course notes >>

# -------- EVENTS --------
def arrival(cust):
    if firstSimulation == True:
        arrivals.append((currentTime, cust.serviceTime))
    queueSize = len(queue)
    if verbose == True:
        print("Arrival at " + str(currentTime))
    if observeArrivals == True:
        observe("arrival")
    if queueSize == 0 and getFreeServer() is not None:
        # No other customer in queue, this one is picked by default, whatever the scheduling discipline
        server = getFreeServer()
        server.serve(cust) # also schedules departure and sets customer waiting time, etc
    else:
        queue.append(cust)
    # Schedule arrivals only if this is the first simulation. Otherwise, we use the same arrivals as previous simulations.
    if firstSimulation == True:
        newCust = Customer(currentTime + arrivalTime(), serviceTime())
        if newCust.arrivalTime < maxSimulationTime or mode == "batchMeans":
            schedule(newCust.arrivalTime, arrival, [newCust])

def departure(server):
    queueSize = len(queue)
    if verbose == True:
        print("Departure at " + str(currentTime))
    server.depart()
    if queueSize > 0:
        discipline = server.discipline
        nextCust = None
        if discipline == "FCFS":
            nextCust = queue.popleft()
        elif discipline == "LCFS":
            nextCust = queue.pop()
        elif discipline == "SSTF":
            nextCust = queue[-1]
            for queueCust in queue:
                if queueCust.serviceTime < nextCust.serviceTime:
                    nextCust = queueCust
            queue.remove(nextCust)
        elif discipline == "ROS":
            # generate random
            randInd = int(round(uniform(0,queueSize-1)))
            nextCust = queue[randInd]
            del queue[randInd]
        server.serve(nextCust) # also schedules departure and sets customer waiting time, etc

def stop():
    global continueDES
    print("Stopped at " + str(currentTime))
    continueDES = False

def calcBatchMean(ind):
    batch = batches[ind]
    bobs = Observation(currentTime)
    bobs.queueContent = sum(obs.queueContent for obs in batch) / max(len(batch),1)
    << Same for other performance measures >>
    batchMeans.append(bobs)

def observe(type):
    global currentL, currentM
    customersInService = [server.content for server in servers if server.content is not None]

    obs = Observation(currentTime)
    obs.queueContent = len(queue)
    obs.systemContent = obs.queueContent + len(customersInService)
    obs.meanCompleteWaitingTime = sum(cust.waitingTime for cust in done) / max(len(done),1) # max to prevent division by zero
    obs.meanWaitingTime = (sum(cust.waitingTime for cust in customersInService) + sum((currentTime - cust.arrivalTime) for cust in
queue)) / max(len(customersInService) + len(queue),1)
    obs.meanCompleteSojournTime = sum(cust.sojournTime for cust in done) / max(len(done),1)
    obs.meanSojournTime = (sum(cust.sojournTime for cust in customersInService) + sum((currentTime - cust.arrivalTime +
cust.serviceTime) for cust in queue)) / max(len(customersInService) + len(queue),1)
    obs.unfinishedWork = (sum(cust.serviceTime for cust in queue) + sum((cust.timeEnteredService + cust.serviceTime - currentTime) for
cust in customersInService)) / amountOfServers
    if verbose == True:
        print("Observation at " + str(currentTime))
```

```
    if(type == "random"):
        if firstSimulation == True:
            obsTimes.append(currentTime)
            if len(agenda) > 0:
                if mode != "batchMeans" or currentL != L-1 or currentM != M-1:
                    schedule(currentTime + expovariate(obsL), observe, ["random"])
        # Additional measures
        if(len(doneSinceLastObsRandom) > 0):
            obs.newWaitingTime = sum(cust.waitingTime for cust in doneSinceLastObsRandom) / len(doneSinceLastObsRandom)
            obs.newSojournTime = sum(cust.sojournTime for cust in doneSinceLastObsRandom) / len(doneSinceLastObsRandom)
            del doneSinceLastObsRandom[:]
        else:
            if(len(observationsRandom) > 0):
                obs.newWaitingTime = observationsRandom[-1].newWaitingTime
                obs.newSojournTime = observationsRandom[-1].newSojournTime
            else:
                obs.newWaitingTime = 0
                obs.newSojournTime = 0
        observationsRandom.append(obs)
        if mode == "batchMeans":
            if currentM == 0:
                batches.append([])
            batches[currentL].append(obs)
            currentM = currentM + 1
            if currentM == M:
                calcBatchMean(currentL)
                currentM = 0
                currentL = currentL + 1
                if currentL == L:
                    schedule(currentTime + expovariate(obsL), stop, [])
    elif(type == "arrival"):
        << Same as random, different lists >>
    elif(type == "departure"):
        << Same as random, different lists >>

# -------- MAIN --------
def simulate():
    global firstSimulation
    for schedulingDiscipline in schedulingDisciplines:
        print("Beginning simulation for " + schedulingDiscipline)
        reset()
        for i in range(0,amountOfServers):
            server = Server(schedulingDiscipline)
            servers.append(server)
        if forceStop == True and mode != "batchMeans":
            schedule(maxSimulationTime, stop, [])
        if firstSimulation == True:
            firstCust = Customer(0, serviceTime())
            schedule(firstCust.arrivalTime, arrival, [firstCust])
            schedule(0, observe, ["random"])
        else:
            for arrivalTuple in arrivals:
                cust = Customer(arrivalTuple[0], arrivalTuple[1])
                schedule(cust.arrivalTime, arrival, [cust])
            for obsTime in obsTimes:
                schedule(obsTime, observe, ["random"])
        run()
        firstSimulation = False
        observations["random"][schedulingDiscipline] = observationsRandom;
        if observeArrivals == True:
            observations["arrival"][schedulingDiscipline] = observationsArrival;
        if observeDepartures == True:
            observations["departure"][schedulingDiscipline] = observationsDeparture;
        if mode == "batchMeans":
            batchDict[schedulingDiscipline] = batchMeans;

def calculateBatchMeansVariance():
    for schedulingDiscipline in schedulingDisciplines:
        batchList = batchDict[schedulingDiscipline]
        for obsVariable in obsVariables:
            batchSum = sum(getattr(batch, obsVariable[0]) for batch in batchList)
            batchSum2 = sum(getattr(batch, obsVariable[0])**2 for batch in batchList)
            batchMean = batchSum/L
            batchVar = batchSum2/L - (batchSum/L)**2
            batchSD = sqrt(batchVar)

            print(schedulingDiscipline + " - " + obsVariable[1])
            print("99% CONFIDENCE INTERVAL: [" + str(batchMean-batchSD*k_conf) + ", " + str(batchMean+batchSD*k_conf) + "]")

if mode == "systemParams":
    maxSimulationTime = sys_simulationTime
    sys_obs = []
    for sys_mean in sys_meanS:
        init()
        meanS, varS= amountOfServers*sys_mean, amountOfServers
        kS, tS = meanS**2/varS, varS/meanS
        simulate()
        sys_obs.append(observations)
    plot_sys(sys_obs)
else:
    simulate()
    plot()
    calculateBatchMeansVariance() # only if mode == batchMeans
```