Queueing Models and Simulation

Amin Rezaei

Computer Science BSc - Amirkabir University of Technology

January 29, 2021

Abstract

Queueing Systems are widely used in daily life and their applications are much to our life. Call centers, Hospitals, Airport Security , ... are examples of these systems. In these systems, customers arrive at different times and one/many servers are responsible to handle them. There are so many different aspects of analyzing these systems but the most important one is to analyze the performance and optimal number of servers. In first step, we must model the system mathematically and at the second step we will analyze the system properties and performance mesaures. The later step could be solved mathematically for some non-complex systems but for complex systems the suitable way is to use Stochastic Simulation which makes us capable of analyzing complex systems easily. In this project, we will take a look at Queue Modelling and then implement a queue simulation algorithm and at last analyze one real-world system with it.

1 Queueing Systems

Queueing System is a system containing queues and servers that customers come in queue and wait until a server is capable of service and after being served they leave the system.

1.1 Characteristics of Queueing System

(Calling Population) Community of Potential Population that may want to use system, is called Calling Population. This population can be finite or infinite, the main difference between these, is the way arrival rate is modelled.

(System Capacity) Some systems propose a limit that shows maximum number of persons in queue and being processed, but some other has infinite capacity.

(Arrival Process) This process is usually identified by interarrival times (time between successive arrivals) for infinite population systems. One of most important and used processes is Poisson Process , proposing if A_n is interarrival time between customer $n-1$ and n then for a system with poisson process, A_n is exponentially distributed with mean 1λ and arrival rate λ. Arrival rate is number of customers per time unit.

(Queue Discipline) This discipline determines that how and with which ordering queue members to be processed. This is also known as Queue Policy. Most used disciplines/policies are First-In First-Out (FIFO) , Last-In First-Out (LIFO) , Service with prioirty and Shortest process time first.

(Service Mechanism) The number of servers, distriubtion of service times and the communication between queues and servers are the key identifiers of our Service Mechanism.

1.2 Queueing Notation

To display and show main characteristics of a Queueing System, Kendall [1953] proposed a notational system that is widely used. The notation's abridged version is based on format $A/B/c/N/K/R$ and we have:

A	Interarrival times distriubtion
B	Service times distribution
c	Number of servers
N	System capacity
K	Calling population
R	Queue Discipline

For distributions we have some common symbols: M (Markov/Exponential) , D (Constant/Deterministic) , E_k (Erlang order k) , G (General) and GI (General Independent). For example $M/M/1/\infty/\infty/FIFO$ shows a single-server system that has infinite capacity and calling population, interarrival and service times are exponentially distributed and queue discipline is FIFO. Because of wide usage of infinite populations and FIFO policy the last three symbols are omitted usually and our system notation is simplified to $M/M/1$.

1.3 Performance Measures of Queueing Systems

ρ	(Server Utilization) is the rate of time servers are doing work to the system time.
$L(t)$	Number of customers in system at time t.
$L_Q(t)$	Number of customers in queue at time t.
L	Long-run time-average number of customers in system. In long-run stable systems if $T \to \infty$ we would have $\frac{1}{T}\int_0^T L(t)dt \to L$.
w	Long-run average time spent in system per customer
w_Q	Long-run average time spent in queue per customer
$P_n(t)$	Probability of having n customers in system at time t

P_n Steady-state probability of having n customers in system. In steady-state systems after long runtime this probability is independent of time.

2 Queueing System Simulation

With having main characteristics of a queueing system , we can easily generate random observation from interarrival and service distributions and simulate the system for desired number of customers and servers over time, resulting in main performance measures and metrics of the system. This result helps us analyze and optimize our system characteristics to yield optimal state and help handling the realworld system better.

2.1 Algorithm and Code of Simulation

We develop a simple algorithm that receives sorted interarrival times, service times and number of servers and then it starts by calculate each customer departure time by passing it over the server which is less busy (gets idle earlier than others) and all events are calculated for each customer. With having these events it creates list of events like queue length changes and system length changes and at last it calculates performance measures with available data. Algorithms output of simulated system is:

- Arrival, waiting, service and departure times with processor server for each customer
- List of queue length changes over time
- List of system length changes over time
- Performance Measures of System like Mean System Customers, Mean Queue Length, Mean Wait Time, Mean Response Time and Server Utilization

We implement **queueCompute** function that simulates and returns the system with getting sorted interarrival times, service times and number of servers.

```r
queueCompute <- function(arrivalTimes, serviceTimes, serversCount) {
  departures <- NULL
  waitings <- NULL
  serveds <- NULL
  serverStates <- numeric(serversCount)
  enqueueTimes <- NULL
```

```r
dequeueTimes <- NULL
detachTimes <- NULL

# Calculate Departures
serverSelect <- function (arrivalTime){
  which.max(arrivalTime - serverStates)
}

for(i in seq_along(arrivalTimes)){
  cArrival <- arrivalTimes[i]
  cService <- serviceTimes[i]
  selectedSrv <- serverSelect(cArrival)

  waitingTime <- max(0, serverStates[selectedSrv] - cArrival)
  serverStates[selectedSrv] <- cArrival + waitingTime + cService

  detachTimes <- c(detachTimes, serverStates[selectedSrv])
  dequeueTimes <- c(dequeueTimes, waitingTime + cArrival)
  enqueueTimes <- c(enqueueTimes, cArrival)

  serveds <- c(serveds, selectedSrv)
  waitings <- c(waitings, waitingTime)
  departures <- c(departures, serverStates[selectedSrv])
}
# Calculate QueueLength for Events
queueCount <- function (cT){
  c <- 0
  if(length(enqueueTimes) == 0 || length(dequeueTimes)==0) return(0)
  for(i in seq_along(enqueueTimes)){
    eT <- enqueueTimes[i]
    dT <- dequeueTimes[i]
    if (eT <= cT && dT > cT){
      c <- c+1
    }
  }
  c
}
qStates <- 0
qTimes <- 0
```

```r
for(i in seq_along(arrivalTimes)){
  cArrival <- arrivalTimes[i]
  w <- waitings[i]
  t <- 0
  if (w == 0){
    t <- 1
  }
  qTimes <- c(qTimes, cArrival)
  qStates <- c(qStates, queueCount(cArrival) + t)
  qTimes <- c(qTimes, cArrival + w)
  qStates <- c(qStates, queueCount(cArrival + w))
}
oD <- order(qTimes)
qTimes <- qTimes[oD]
qStates <- qStates[oD]

# Calculate SystemLength for Events
servingCount <- function (cT){
  c <- 0
  if(length(dequeueTimes) == 0 || length(detachTimes)==0) return(0)
  for(i in seq_along(dequeueTimes)){
    eT <- dequeueTimes[i]
    dT <- detachTimes[i]
    if (eT <= cT && dT > cT){
      c <- c+1
    }
  }
  c
}

allEvents <- sort(unique(c(detachTimes, dequeueTimes, enqueueTimes)))
sTimes <- 0
sStates <- 0
for(i in seq_along(allEvents)){
  e <- allEvents[i]
  sTimes <- c(sTimes, e)
  sStates <- c(sStates, queueCount(e) + servingCount(e))
}
```

```r
  area_under <- function(x, y){
    nx <- c(0, x[1:length(x)-1])
    ny <- c(0, y[1: length(y)-1])
    nd <- x - nx
    area <- nd * ny
    return(sum(area))
  }

  df <- data.frame(
    arrival=arrivalTimes,
    service=serviceTimes,
    departure=departures,
    waiting=waitings,
    server=serveds
  )
  ql <- data.frame(
    time=qTimes,
    len=as.integer(qStates)
  )
  sl <- data.frame(
    time=sTimes,
    len=as.integer(sStates)
  )
  return(list(
    departures=df,
    queueLength=ql,
    systemLength=sl,
    meanSystemCustomers=area_under(sl$time, sl$len)/max(departures),
    meanQueueLength=area_under(ql$time, ql$len)/max(ql$time),
    meanResponseTime=mean(serviceTimes + waitings),
    meanWaitTime=mean(waitings),
    serverUtilization=sum(serviceTimes) / (max(departures) * serversCount)
  ))
}
```

To verify algorithms correctness we check it with implemented algorithm by queuecomputer library.

2.2 Testing Steady-state Probability

We mentioned at section 1 that if a queueing system is stable over long runs , with long runtime the probability of having exact number of customers in system will become independent of time.

To check this, it's enough to simulate system over long time and check probability plots over time to see if they converges to special value or not.

Our implementation, due to being pure R and use of non-optimized structures to solve system, doesn't have good time performance. queuecomputer's backend is implemented in C++ thus it's capable of doing simulation very fast. For this reason we will simulate the system using our algorithm for 3000 customers over time and for 10000 customer using queuecomputer's algorithm. This way we can check correctness of our algorithm too.

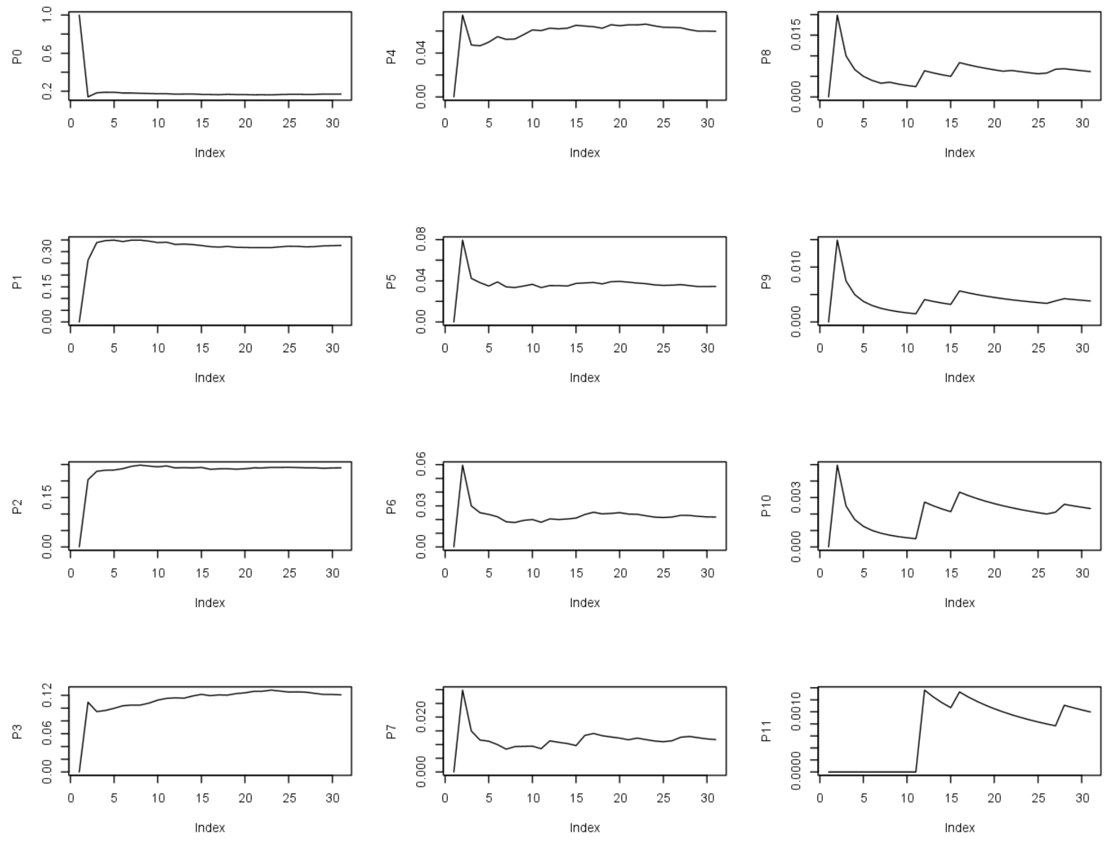


Figure 1: Probability of having exact number of customers over time (3000 customers - Implemented Algorithm)

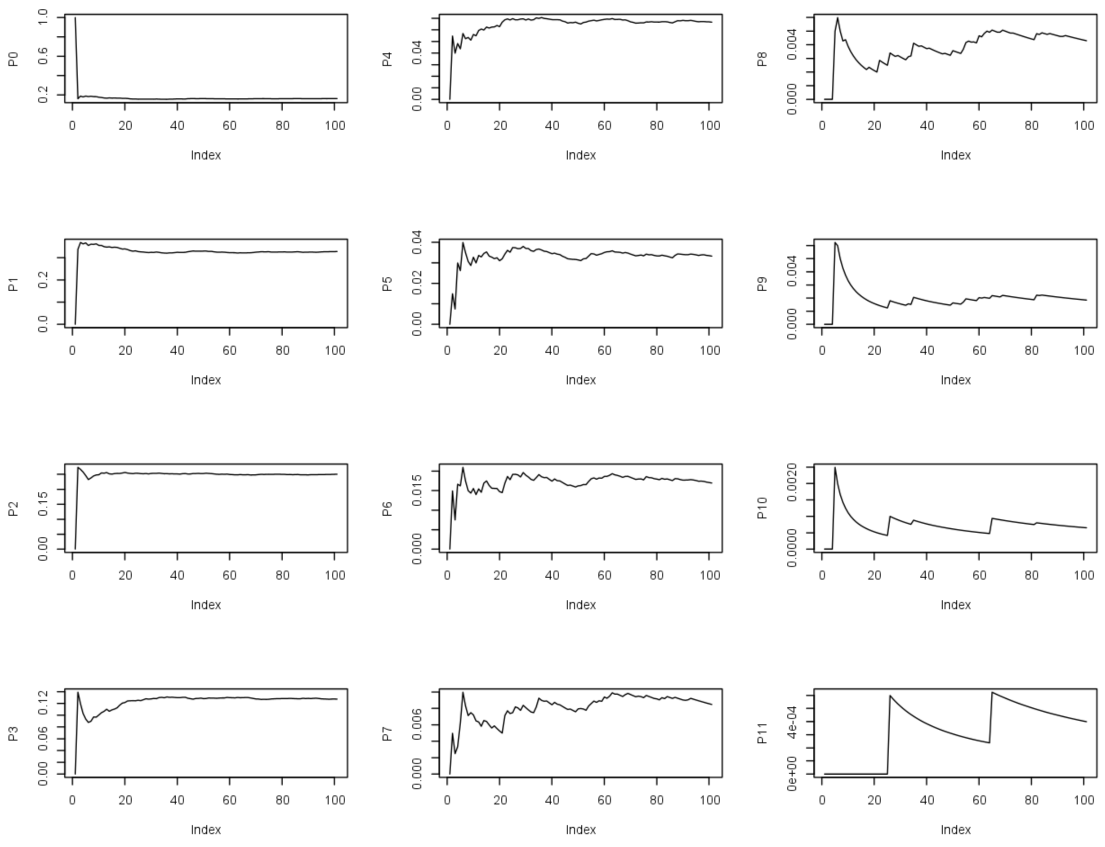


Figure 2: Probability of having exact number of customers over time (10000 customers - queuecomputer implementation)

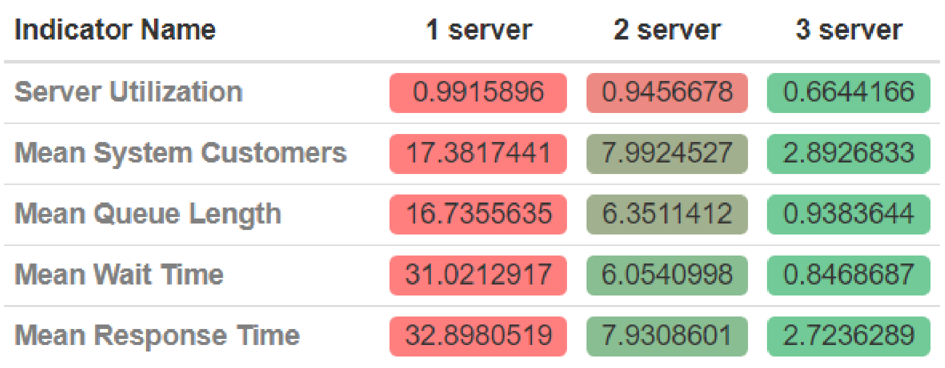
By looking at the plots we notice that almost all of them have converged to a special value/area (noting that the ones with high oscillation are oscillating in a very small range). Here the test codes are proposed, **calculateSteadyState** function calculates the discussed probability in different times by 200 steps for different number of customers and returns a list of arrays, **testSteadyState** and **testSteadyStateLibrary** functions simulate the system with defined functions and calculate probabilities then plot them.

```r
library(queuecomputer)
calculateSteadyState <- function (systemLengthsState){
  uLen <- unique(systemLengthsState)
  prob <- list()
  for(i in seq(1, length(systemLengthsState), 200)){
    for(u in uLen){
      tg <- NULL
      if(length(prob) >= u+1){
        tg <- prob[[u+1]]
      }
      prob[[u + 1]] <- c(tg,sum(systemLengthsState[1:i] == u) / i)
    }
  }
  return(prob)
}

testSteadyState <- function (){
  servers <- 2
  n <- 3000
  set.seed(9712017)
  arrivals <- rexp(n)
  arrivalTimes <- cumsum(arrivals)
  serviceTime <- rexp(n)
  od <- queueCompute(arrivalTimes, serviceTime, servers)
  prob <- calculateSteadyState(od$systemLength)
  par(mfcol = c(2, 3))
  for(i in seq_along(prob)){
    plot(prob[[i]], type="l", ylab = sprintf("P%d", i-1))
  }
}
testSteadyStateLibrary <- function (){
  servers <- 2
  n <- 10000
  set.seed(9712017)
  arrivals <- rexp(n)
  arrivalTimes <- cumsum(arrivals)
  serviceTime <- rexp(n)
  od <- queue_step(arrivalTimes, serviceTime, servers)
  prob <- calculateSteadyState(od$systemlength_df)
  par(mfcol = c(2, 3))
  for(i in seq_along(prob)){
    plot(prob[[i]], type="l", ylab = sprintf("P%d", i-1))
  }
}
```

3 Simulating a System

In this section, a call center is simulated that is described like $M/M/k$ that for $k \in \{1, 2, 3\}$ system will be analyzed. Interarrival times are exponentially distributed with mean 1 and service times are exponentially distributed with mean 2. System is simulated for 50 customers over time. Results are provided below:



Indicator Name	1 server	2 server	3 server
Server Utilization	0.9915896	0.9456678	0.6644166
Mean System Customers	17.3817441	7.9924527	2.8926833
Mean Queue Length	16.7355635	6.3511412	0.9383644
Mean Wait Time	31.0212917	6.0540998	0.8468687
Mean Response Time	32.8980519	7.9308601	2.7236289

Figure 3: Call center simulation for 50 customers with different number of servers.

It's obvious that 1 server is not the suitable case and it's not stable through long runs because queue length gets increased. 2 servers is a good ooption but 3 servers is the most suitable in which requests are fastly processed an it can be stable through long runs.

Code Simulation and Report generations are included here, **callCenterSimulate** function simulates this system with number of servers as input and provides the simulation results, **testCallCenter** function runs the discussed test for different number of servers and generates the last table. To create and view tables libraries data.table, dplyr, formattable, tidyr are used.

```r
# Libraries for Table Data and Drawing
library(data.table)
library(dplyr)
library(formattable)
library(tidyr)
callCenterSimulate <- function (s){
  # Considering a call center with s servers
  set.seed(9712017)
  arrivals <- rexp(50, 1)
  servers <- s
  arrivalTimes <- cumsum(arrivals)
  serviceTime <- rexp(50, 0.5)
  od <- queueCompute(arrivalTimes, serviceTime, servers)
  r <- c(od$serverUtilization, od$meanSystemCustomers, od$meanQueueLength, od$me
  return(r)
}
testCallCenter <- function (){
  customGreen <- "#71CA97"
  customRed <- "#ff7f7f"
  i1 <- data.table(
    `Indicator Name` = c("Server Utilization",
     "Mean System Customers", "Mean Queue Length",
      "Mean Wait Time", "Mean Response Time"),
    `1 server` = callCenterSimulate(1),
    `2 server` = callCenterSimulate(2),
    `3 server` = callCenterSimulate(3)
  )
  formattable(i1, align =c("l","c","c","c"), list(
    `Indicator Name` = formatter("span", style =
     ~ style(color = "grey",font.weight = "bold")),
    area(row=1, col = 2:4) ~ color_tile(customGreen, customRed),
    area(row=2, col = 2:4) ~ color_tile(customGreen, customRed),
    area(row=3, col = 2:4) ~ color_tile(customGreen, customRed),
    area(row=4, col = 2:4) ~ color_tile(customGreen, customRed),
    area(row=5, col = 2:4) ~ color_tile(customGreen, customRed)
  ))
}
```

References

[1] A. MohammadPour. Simulation lecture and slides.

[2] Anthony Ebert, Paul Wu, Kerrie Mengersen, and Fabrizio Ruggeri. Computationally efficient simulation of queues: The r package queuecomputer, 2019.

[3] Jerry Banks, John S. Carson, Barry L. Nelson, and David M. Nicol. Chapter 6 - queueing models. In *Discrete-Event System Simulation*, page 227–274. Pearson/Prentice Hall, fifth edition, 2010.

A Appendix 1

Complete code for simulation algorithm, steady-state and call center tests in R is provided in the following repository: https://github.com/AminRezaei0x443/QueueSimulation