International Journal of Research in Engineering and Science (IJRES)
ISSN (Online): 2320-9364, ISSN (Print): 2320-9356
www.ijres.org Volume 13 Issue 6 ǁ June 2025 ǁ PP. 321-325

Queuing Theory in Cloud Computing: Analyzing M/M/1 and M/M/c Models with AWS

Dr.Pratima singh
Assistant Professor, Mathematics Department, Post Graduate College Ghazipur (U.P),233001

Abstract -:
This paper explores the application of queuing theory in cloud computing, emphasizing its potential to optimize service delivery, resource utilization, and cost efficiency. M/M/1, M/M/care analyzed in the context of cloud services to address dynamic workloads and user demands.
Implementation steps and code examples for each model are provided, highlighting key metrics like queue length, waiting time, server utilization, and blocking probability. A comparative analysis of these models illustrates their suitability for different scenarios, from single-server setups to complex systems with variable service times. The findings underline the importance of selecting appropriate queuing models to meet system-specific requirements and propose future enhancements to tackle challenges like impatient user behavior and resource constraints.
Keywords -: *Queuing Theory , Cloud Computing , M/M/1 Model , M/M/c Model, Service Optimization, Resource Utilization, Performance Metrics, Computational Modeling*

Main Topics -
1. Cloud Computing Overview
Defined as an on-demand resource provisioning platform (e.g., IaaS, PaaS, SaaS).
Discusses its increasing global significance with growing investments.
2. Queuing Theory
Explains queuing theory as the mathematical modeling of waiting lines to optimize service and minimize costs.
Focuses on balancing service capacity and user demand.
3. Queuing Models in Cloud Computing
M/M/1: Single server; efficient but long waiting times under heavy loads.
M/M/c: Multiple servers in parallel; improves throughput and reduces waiting times.
Key Insights
Performance Measures: Models evaluate response time, queue length, throughput, and resource utilization.
Challenges: Impatient users, balking, reneging, and cost-energy trade-offs.
3. Critical Comparisons
Highlights trade-offs among various queuing models based on real-world cloud application demands.
4. Future Scope
a) Recommends developing hybrid models addressing dynamic queues, impatient behavior, and constrained environments.
Steps to Implement M/M/1
1. Define Parameters:
Arrival rate (λ\lambdaλ): Average number of arrivals per unit time.
Service rate (μ\muμ): Average number of services per unit time.
Simulation time (TTT): Total time to run the simulation.
2. Generate Exponential Times:
Use random exponential distribution to simulate inter-arrival and service times.
3. Simulate the Queue:
Track arrival, service start, and departure times.
Use an event-driven approach to process arrivals and departures sequentially.
4. Calculate Metrics:
Average queue length.
Average waiting time.
Server utilization.

```
import java.util.LinkedList; import java.util.Queue; import java.util.Random;
public class MM1Queue {
public static void main(String[] args) {
// Parameters
```

```
double arrivalRate = 5.0;  // λ: arrivals per unit time double serviceRate = 6.0;  // μ: services per unit time double simulationTime = 100.0;  // Total simulation time
// Variables
double currentTime = 0.0; int customersServed = 0;   int totalCustomers = 0; double totalWaitTime = 0.0;
double next Arrival Time = generate Exponential (arrival Rate); double next Departure Time = Double. MAX_VALUE;
// Queue to hold arrival times
Queue<Double> queue = new LinkedList<>();
// Simulation loop
while (currentTime < simulationTime) {
if (nextArrivalTime < nextDepartureTime) {
// Process arrival
currentTime = nextArrivalTime; totalCustomers++; queue.add(currentTime);
// Schedule next arrival
nextArrivalTime = currentTime + generateExponential(arrivalRate);
// If the server is idle, start service immediately if (queue.size() == 1) {
nextDepartureTime = currentTime + generateExponential(serviceRate);
}
} else {
// Process departure
currentTime = nextDepartureTime; double arrivalTime = queue.poll();
totalWaitTime += currentTime - arrivalTime; customersServed++;
// Schedule next departure if the queue is not empty if (!queue.isEmpty()) {
nextDepartureTime = currentTime + generateExponential(serviceRate);
} else {
nextDepartureTime = Double.MAX_VALUE; // No customers in queue
}
}
}
// Output results
System.out.println("Total Customers: " + totalCustomers); System.out.println("Customers Served: " + customersServed); System.out.println("Average Wait Time: " + (totalWaitTime / customersServed));
System.out.println("Server Utilization: " + (customersServed / (simulationTime * serviceRate)));
}
// Method to generate exponential random numbers private static double generateExponential(double rate) {
Random rand = new Random();
return -Math.log(1 - rand.nextDouble()) / rate;
}
}
```

Key Points of the Code

1. Parameters:

- arrivalRate (λ\lambdaλ): Set as 5.0 arrivals per time unit.

- service Rate (μ\muμ): Set as 6.0 services per time unit.

simulationTime: Duration for which the simulation runs.

Queue Handling**:**

A Queue tracks the arrival times of customers.

Events**:**

Arrival: A new customer arrives based on exponential inter-arrival times.

Departure: A customer is served and leaves after an exponential service time.

Metrics**:**

Average Wait Time: Total wait time divided by the number of served customers.

Server Utilization: Ratio of time the server is busy to the total simulation time.

Output Example

For parameters:

Arrival rate: 5.0

Service rate: 6.0

Simulation time: 100.0

Server Utilization: 0.859

Steps to Implement the M/M/c Queuing Model

In the **M/M/c** model:
M: Markovian (exponential inter-arrival times).
M: Markovian (exponential service times).
c: Multiple servers available to handle requests.
The model adds complexity with multiple servers, requiring tracking of:
Server Status: Which servers are busy or idle.
Queue Length: Customers waiting when all servers are busy.
Steps to Implement
Define Parameters:
Arrival rate (λ\lambdaλ).
Service rate per server (μ\muμ).
Number of servers (ccc).
Total simulation time.
Generate Events**:**
Arrival: Generate based on exponential inter-arrival times.
Departure: Generate based on exponential service times for each server.
Simulation**:**
Assign customers to idle servers if available.
If all servers are busy, add customers to the queue.
Process departures when a server finishes servicing.
Metrics to Calculate**:**
Average queue length.
Average waiting time.
Server utilization.
import java.util.LinkedList; import java.util.Queue; import java.util.Random;
public class MMcQueue {
public static void main(String[] args) {
// Parameters
double arrivalRate = 5.0; // λ: arrivals per unit time
double serviceRate = 6.0; // μ: services per server per unit time int numServers = 3; // c: number of servers
double simulationTime = 100.0; // Total simulation time
// Variables
double currentTime = 0.0; int totalCustomers = 0; int customersServed = 0;
double totalWaitTime = 0.0;
double[] serverAvailableTime = new double[numServers]; // Tracks when each server will be free
Queue<Double> queue = new LinkedList<>(); // Tracks arrival times of waiting customers double next ArrivalTime = generate Exponential (arrivalRate);
// Simulation loop
while (currentTime < simulationTime) {
double nextDepartureTime = getNextDepartureTime(serverAvailableTime); if (nextArrivalTime < nextDepartureTime) {
// Process arrival
currentTime = nextArrivalTime; totalCustomers++;
// Find an idle server
int idleServer = findIdleServer(serverAvailableTime, currentTime); if (idleServer != -1) {
// Assign the customer to the idle server
serverAvailableTime[idleServer] = currentTime + generateExponential(serviceRate);
} else {
// Add customer to the queue queue.add(currentTime);
}
// Schedule next arrival
nextArrivalTime = currentTime + generateExponential(arrivalRate);
} else {
// Process departure
currentTime = nextDepartureTime; customersServed++;
if (!queue.isEmpty()) {
// Serve the next customer in the queue double arrivalTime = queue.poll(); totalWaitTime += currentTime - arrivalTime;
// Assign to the server that just finished

```
int departingServer = findDepartingServer(serverAvailableTime, currentTime);
serverAvailableTime[departingServer] = currentTime +
generateExponential(serviceRate);
}
}
}

// Output results
System.out.println("Total Customers: " + totalCustomers); System.out.println("Customers Served: " +
customersServed); System.out.println("Average Wait Time: " + (totalWaitTime / customersServed));
System.out.println("Server Utilization: " + calculateUtilization(serverAvailableTime,
simulationTime, numServers));
}
// Generate exponential random number
private static double generateExponential(double rate) { Random rand = new Random();
return -Math.log(1 - rand.nextDouble()) / rate;
}
// Find the next departure time
private static double getNextDepartureTime(double[] serverTimes) { double nextTime = Double.MAX_VALUE;
for (double time : serverTimes) { if (time < nextTime) {
nextTime = time;
}
}
return nextTime;
}
// Find an idle server
private static int findIdleServer(double[] serverTimes, double currentTime) { for (int i = 0; i <
serverTimes.length; i++) {
if (serverTimes[i] <= currentTime) { return i;
}
}
return -1;  // No idle server
}
// Find the server responsible for the next departure
private static int findDepartingServer(double[] serverTimes, double currentTime) { for (int i = 0; i <
serverTimes.length; i++) {
if (serverTimes[i] == currentTime) { return i;
}
}
return -1;  // Should never happen
}
// Calculate server utilization
private static double calculateUtilization(double[] serverTimes, double simulationTime, int numServers) {
double busyTime = 0.0;
for (double time : serverTimes) {
busyTime += Math.min(time, simulationTime);
}
return busyTime / (simulationTime * numServers);
}
}
```

Key Features of the Code

Server Handling:

Each server's availability is tracked in the serverAvailableTime array.

Customers are assigned to the next available server.

Queue Management**:**

If all servers are busy, the customer waits in a queue (queue).

Metrics**:**

Average Wait Time: Calculated as total wait time divided by customers served.

Server Utilization: Fraction of time servers are busy during the simulation.

Output Example
For parameters:
Arrival rate: 5.0
Service rate: 6.0
Number of servers**: 3**
Simulation time: 100.0
Summary/Conclusion
The best queuing model among M/M/1, M/M/c depends on the specific requirements and constraints of the system you are modeling. Each model has its strengths and weaknesses suited to particular scenarios. Here's a comparative analysis to help you decide:
M/M/1 (Single Server, Exponential Service Time)
When to Use:
Simple systems with low traffic and a single server.
Best for scenarios where queueing complexity is minimal (e.g., basic customer service lines).
Advantages:
Simple to implement and analyze.
Requires fewer resources.
Disadvantages:
Performance degrades with high traffic.
Long queues and waiting times when traffic is high.
M/M/c (Multiple Servers, Exponential Service Time)
When to Use:
Systems with multiple servers handling traffic concurrently (e.g., bank counters, call centers).
Moderate to high traffic scenarios with sufficient resources.
Advantages:
Reduces waiting time compared to M/M/1.
Increases throughput by parallelizing service.
Disadvantages:
Higher implementation complexity than M/M/1.
Increased cost for maintaining multiple servers.

Summary Table

Model	Strengths	Weaknesses	Best For
M/M/1	Simple, low resource needs.	Long queues under high traffic.	Small, low-traffic systems.
M/M/c	Handles moderate to high traffic.	Higher server costs.	Multi-server systems.

Which is Best?
For simplicity: M/M/1.
For scalability and high throughput: M/M/c.
If you need a balance between accuracy and complexity, M/M/c is often the most practical choice for many real-world systems.

References

[1]. Kleinrock, L. (1975). *Queueing Systems Volume 1: Theory*. Wiley-Interscience.
[2]. Gross, D., Shortle, J. F., Thompson, J. M., & Harris, C. M. (2008). *Fundamentals of Queueing Theory*. Wiley.
[3]. Tanenbaum, A. S., & Van Steen, M. (2007). *Distributed Systems: Principles and Paradigms*. Pearson.
[4]. Harchol-Balter, M. (2013). *Performance Modeling and Design of Computer Systems: Queueing Theory in Action*. Cambridge University Press.
[5]. AWS Documentation. (n.d.). *Introduction to Cloud Computing*. Retrieved from AWS Official Documentation.
[6]. Kendall, D. G. (1953). "Stochastic Processes Occurring in the Theory of Queues and their Analysis by the Method of the Imbedded Markov Chain". *The Annals of Mathematical Statistics*, 24(3), 338–354.