

263-3855-00: Cloud Computing Architecture

Section 05: Performance Modeling

Swiss Federal Institute of Technology Zurich (ETH Zürich)

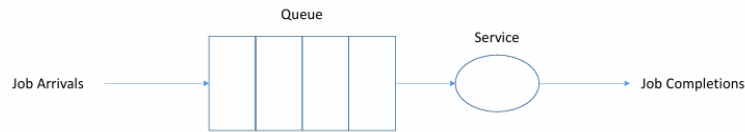
Last Edit Date: 05/31/2025

Disclaimer and Term of Use:

1. We do not guarantee the accuracy and completeness of the summary content. Some of the course material may not be included, and some of the content in the summary may not be correct. You should use this file properly and legally. We are not responsible for any results from using this file.
2. This personal note is adapted (derived) from the slides and codes from *Professor Ana Klimovic and people from ETH Zurich Systems Group*. Please [contact us](#) to delete this file if you think your rights have been violated.
3. This work is licensed under a [Creative Commons Attribution 4.0 International License](#).

I. Basics of querying theory

A queue looks like



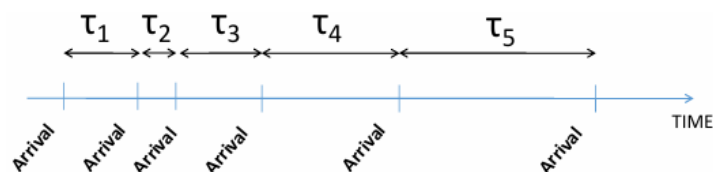
To characterize a queuing system, we have

- Arrival rate
- Service time
- Number of servers
- System capacity
- Population size
- Queue discipline (ordering)

Mean arrival rate

- Mean interarrival time: $E[\tau]$
- Mean arrival rate: $\lambda = 1/E[\tau]$
- λ is not a random variable
- Examples:
 - A single client submits a query every 200 ms, then λ is 5 queries / second
 - 10 clients submit a query each every 500 ms, then λ is 20 queries / second
 - These are the queries submitted to the system

Interarrival time (IAT) distribution



- We assume the interarrival times form a sequence of independent and identically distributed (IID) random variables
- Common assumption is a Poisson distribution, IAT follows an exponential distribution

Assumptions

- Queuing systems assume an arrival rate that is
 - State independent (does not depend on number of previous jobs)
 - Stationary (does not change in time)
- These assumptions do not hold in real systems

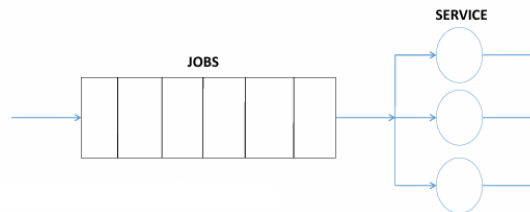
- Burstiness; receive batch of jobs
- Flash crowds (popularity)
- Social effects (time of day variant load)

Service time per job

- The time it takes to process a job (only the time it takes to process it, not including the time it has been waiting in the queue) is s
- Mean service rate $\mu = 1/E[s]$
- If there are m servers, mean service rate is $m\mu$
- μ is not a random variable
- Example:
 - Printer takes on average 20 seconds per job, then $\mu = 0.05$ jobs / second, which is 3 jobs / minutes.

Number of servers

- The service can be provided by one or more servers
- Assume work in parallel and independent
- Server do not interfere with each other
- Total service rate is aggregation of each individual service rate



System capacity

- Buffer capacity is the maximum number of jobs that can be waiting for service
- System capacity includes jobs waiting and jobs receiving service
- In reality, capacity is finite, very important in practice
- Analysis often assumes infinite capacity

Queue discipline

- FCFS: First Come - First Served
 - Ordered queue
- LCFS: Last Come - First Served
 - Stack
- RR: Round Robin
 - CPU allocation to processes

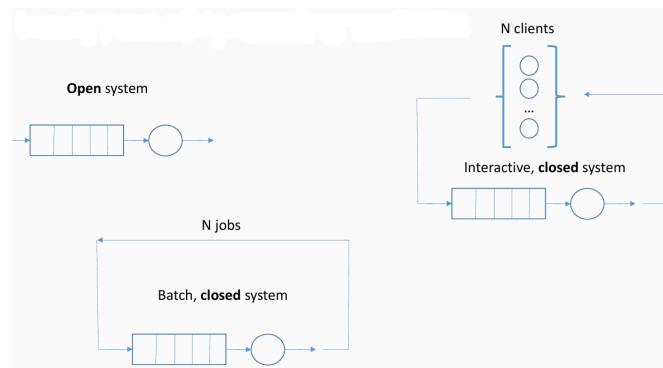
- RSS: random
- Priority based

Kendall's notation for queuing system

A/S/m/C/P/QD

- A: arrival distribution
- S: service distribution
- m: number of servers
- C: buffer capacity
- P: population size (max can serve)
- QD: queue discipline (ordering)

1. Three types of queuing system



Open system

- Load comes from a potentially unlimited set of clients
- Load is not limited by clients waiting
- Load is not self-adjusting (load keeps coming even if system stops)
- Tests system's stability
- Example: web server

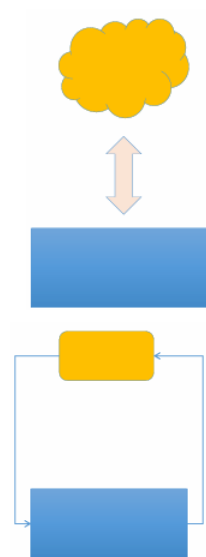
Closed system

- Load comes from a limited set of clients
- Clients wait for response before sending next request
- Load is self-adjusting
- System tends to stability
- Example: database with local clients

What is the system throughput (X)?

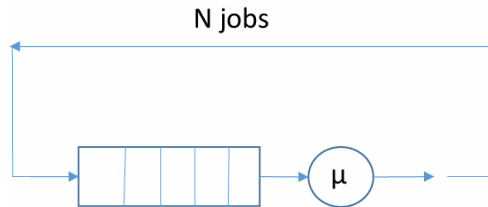
Assume $\lambda < \mu$, then later we will see for otherwise:

- In an open system, the throughput is the mean arrival rate





- In a closed system, the throughput is the service rate



Food for thought

In an **open** system, the throughput is the arrival rate:

- Making the service faster (larger CPU, better disk, etc.) will:
 - Decrease the response time (overall time in the system)
 - Decrease the time waiting in the queue
 - But will not increase the throughput as that depends on the workload (arrival rate)
 - However, assuming $\lambda < \mu$, making the service faster (larger μ) will allow the system to support larger workloads (higher λ)
- In queuing systems we talk about the throughput for a given arrival rate, which is not the same as the max throughput the system can sustain.

In a **closed** system, μ is the throughput

- In steady state closed system, the arrival rate is the service rate ($\lambda = \mu$)
- In real systems, μ is an upper bound for the throughput
 - Assume there are always jobs ready when a job is finished
 - No overhead in switching to new job
 - All jobs complete correctly
 - Service rate is state independent (does not depend on number of jobs in the queue)
 - Service rate is stationary (does not change with time)

Utilization (ρ) and throughput (X)

- We observe a queuing system during the time T
- During this period, the device (i.e., the service) is busy for time B
- The utilization of the device is defined as $\rho = B/T$
- The throughput (X) in the system is the number of jobs completed (C) during the time of observation (T), $X = C/T$
- This can be rewritten as follows:

- $X = C/T = C/B \times B/T$, where $\rho = B/T$ and B/C is the average service time ($S = 1/\mu$) (time the device is busy divided by the number of jobs completed), yielding:

- $X = \rho \times \mu$

- $\rho = X \times S$, which is called the utilization law

Utilization in an open system

- In an open system, the throughput (X) is the arrival rate (λ). Thus, the utilization can be rewritten as $\rho = \lambda/\mu = \lambda \times E[s]$
- The system is stable if $\rho < 1$, then $\lambda < \mu$
- In other words, the system is stable if the mean arrival rate (λ) is less than the mean service rate (μ), otherwise the queue grows without bounds

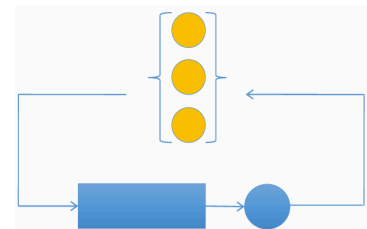
$\rho = 1$

- Unless arrivals and service are deterministic and exactly scheduled, $\rho = 1$ does not lead to a steady system
 - Randomness prevents queuing from emptying
 - Server cannot catch up
 - Queue grows without bound
- One way to avoid this is flow control (drop jobs when load too high)
- In real systems, utilization is kept well below 1 to be on the safe side

II. Interactive law for closed systems

1. Interactive response time law

- Users submit a request, when they get a response, they think for a time Z , and submit the next request
- Applies to measurements in any system under a generic set of conditions



2. Interactive response time law (closed system)

- Response time R
- Total cycle time $R + Z$
- Each user generates $T/(R + Z)$ requests in time T
- There are N users (sometimes called multiprogramming level, MPL)

$$X = \frac{\text{Jobs}}{\text{Time}} = \frac{N \cdot \frac{T}{R+Z}}{T} = \frac{N}{R+Z} \Rightarrow R = \frac{N}{X} - Z$$

Real systems are not ideal, in reality:

- Not all jobs are equal

- Not all processing is identical
- Network effects
- Processing overhead (not only wait)
- Exceptions and not “normal” return values

Simplifications

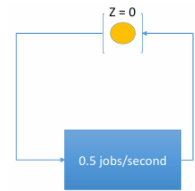
To better understand these systems, we will now make them deterministic and simplify them by assuming:

- Jobs go automatically to the right service
- No overhead in routing jobs
- Perfect load balancing
- The jobs have a size distribution, instead we will work different service rates and multiple services

3. Interactive response time law (in practice)

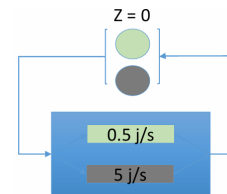
For 1 client:

- Throughput: 0.5 job / second
- Response time: $1 / 0.5 = 2$ seconds (inverse of throughput)



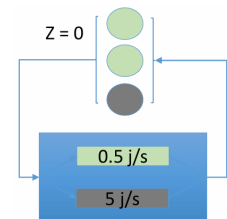
For 2 clients:

- Throughput: 5.5 jobs / second
- Average response time: $2 / 5.5 = 0.36$ second
- Actual response time for each job
 - $1 * (1 / 0.5) = 2$ seconds for green jobs
 - $1 * (1 / 5) = 0.2$ second for gray jobs

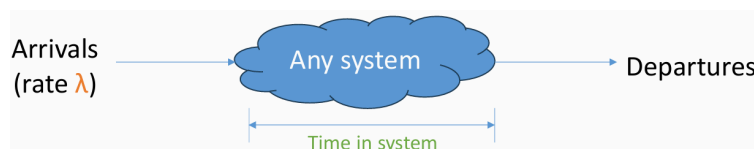


For 3 clients:

- Throughput: 5.5 jobs / second
- Average response time: $3 / 5.5 = 0.54$ second
- Actual response time for each job
 - $2 * (1 / 0.5) = 4$ seconds for green jobs
 - $1 * (1 / 5) = 0.2$ second for gray jobs



III. Little's law



- The average number of jobs in the system = average arrival rate * average time a job spends in the system
- Holds regardless of arrival distribution, service time distribution, service order, topology of queuing network, etc.

Notation (the following all are random variables)

- $n = n_s + n_q$, where
 - n is the number of jobs in the whole system
 - n_s is the number of jobs in the service
 - n_q is the number of jobs in waiting for service
- $w = w_q + s$, where
 - w is the total time in the whole system
 - w_q is the time waiting in the queue
 - s is the time in the service

1. Little's law for open systems

- For queuing system: $E[n] = \lambda \cdot E[w]$
- For the queue: $E[n_q] = \lambda \cdot E[w_q]$
- With
 - $E[n] = E[n_q] + E[n_s]$
 - $E[w] = E[w_q] + E[s] = E[w_q] + 1/\mu$

2. Little's law for closed systems

Similar to that of open systems, but with different parameters $N = X \cdot (R + Z)$.

Example:

You measure that the average time to respond to a request was 100 ms and the server received ~100 requests / second. If each active request requires 5 KB memory, how much memory needs to be reserved for the average number of requests in the system?

Jobs in the system = $100 \cdot 0.1 = 10$ jobs (Little's law)

Memory needed = $10 \cdot 5 \text{ KB} = 50 \text{ KB}$

IV. Queuing theory: basic models

1. Birth-death processes

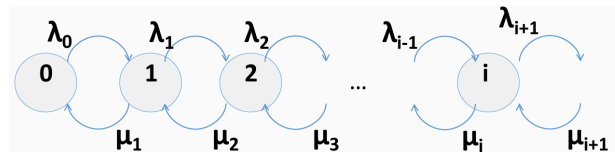
Stochastic processes

- Many of the values in a queuing system are random variables that are a function of time, e.g., the waiting time at a queue
- A random function (i.e., collection of random variables) is called a stochastic process
- If the values that a process can take are finite or countable, it is a discrete process or a stochastic chain

Markov processes

- If the future states of a process depend only on the current state and not on past states (i.e., memoryless), the process is a Markov process
- Discrete Markov processes are called Markov chains
- A Markov chain in which the transition between states is limited to neighboring states is called a birth-death process

Discrete Markov chain, steady state, and equilibrium probability

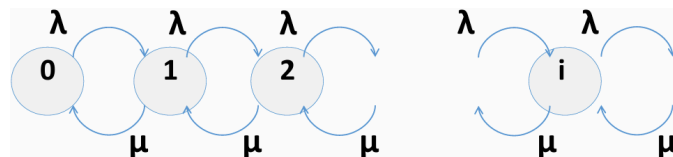


- λ_i and μ_i are the state transition rates
- At equilibrium, $\lambda_n P_n = \mu_{n+1} P_{n+1}$ for all n , where P_n is the probability of being in state n , so

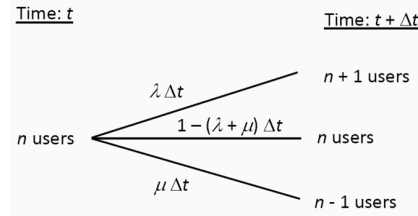
$$P_{n+1} = (\lambda_n / \mu_{n+1}) P_n$$
- The probability of being in state n is $P_n = \frac{\lambda_0 \lambda_1 \dots \lambda_{n-1}}{\mu_1 \mu_2 \dots \mu_n} P_0$

2. M/M/1

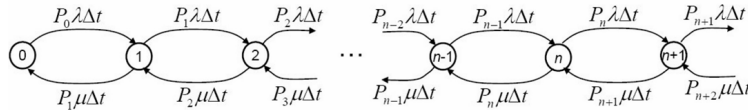
- “M” for memoryless distribution of arrival and service
 - Poisson arrival process -> exponential arrival time distribution with mean rate λ
 - Poisson service process -> exponential service time distribution with mean rate μ
- “1” for single server
- Infinite buffers and first come first serve (FCFS)



We can also use the state transition diagram to model transition probabilities:



The probability of observing a transition from state i to state j during the next Δt (for the system in steady-state):



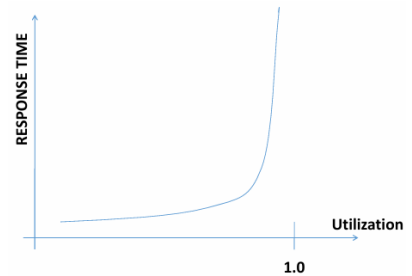
How to solve for P_n in M/M/1?

Recall at equilibrium, $\lambda P_n = \mu P_{n+1}$ for all n , where P_n is the probability of being in state n .

- $P_n = (\lambda/\mu)^n P_0, n = 1, 2, \dots, \infty$
- $P_n = \rho^n P_0$

We also know that all probabilities must add to 1:

- $\sum_{n=0}^{\infty} \rho^n P_0 = 1 \Rightarrow P_0 = 1 - \rho$
- $P_n = \rho^n (1 - \rho)$



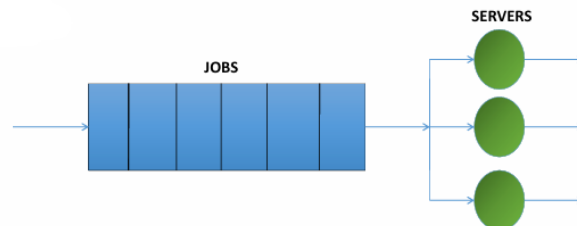
Utilization

Probability that there is one or more jobs in the system, $U = 1 - p_0 = \rho = \lambda/\mu$, for the M/M/1 queue.

M/M/1 behavior

- The mean number of jobs is $E[n] = \sum_{n=1}^{\infty} n \cdot P_n = \sum_{n=1}^{\infty} n(1 - \rho)\rho^n = \frac{\rho}{1-\rho}$
- Applying Little's law ($E[n] = \lambda \cdot E[w]$), we get the response time $E[w] = \frac{1/\mu}{1-\rho}$

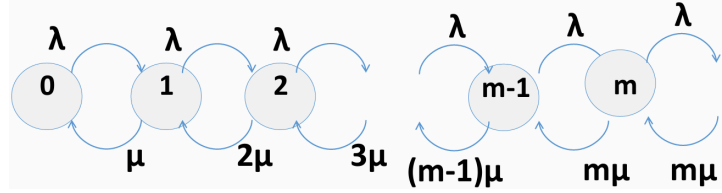
3. M/M/m



- Each server serves μ jobs per unit of time

- Jobs get service right away if less than m jobs in system, otherwise they wait in queue

Probabilities in M/M/m



The number of jobs in a M/M/m system is a birth death process. Hence $P_n = \frac{\lambda_0 \lambda_1 \dots \lambda_{n-1}}{\mu_1 \mu_2 \dots \mu_n} P_0$

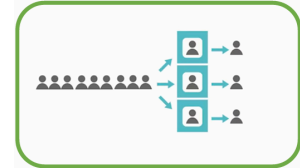
Steady state probabilities for M/M/m

- $P_n = \frac{\lambda^n}{n! \mu^n} P_0$, for $n = 0, 1, 2, \dots, m - 1$
- $P_n = \frac{\lambda^n}{m! m^{n-m} \mu^n} P_0$, for $n = m, m + 1, \dots, \infty$



m times M/M/1 vs. M/M/m

- What is better?
 - m queues of the form M/M/1 with arrival rate λ/m
 - A single system of the form M/M/m with arrival rate λ
- In general, M/M/m will be better because it leads to less waiting time
 - In m times M/M/1, jobs waiting in a queue do not benefit if a server in another queue is free



4. M/M/m/B

- The queuing system has limited buffer capacity. After B buffers are full, jobs are no longer admitted, i.e., drop requests.
- State transition diagram is similar to that of M/M/m but it finishes with B (as opposed to having ∞ states)
- As before $P_n = \frac{\lambda_0 \lambda_1 \dots \lambda_{n-1}}{\mu_1 \mu_2 \dots \mu_n} P_0$
- All the other parameters can be computed from the following probabilities
- Effective arrival rate
 - Arrival rate λ
 - After B jobs, no more jobs enter the system
 - $\lambda' = \lambda(1 - P_B)$ effective arrival rate, i.e., rate of arrivals that actually enter the system
 - $\lambda - \lambda' = \lambda P_B$ packet loss rate

- Apply effective arrival rate to Little's law

State probabilities

- $P_n = \frac{\lambda^n}{n! \mu^n} P_0 = \frac{(m\rho)^n}{n!} P_0$, for $n < m$
- $P_n = \frac{\lambda^n}{m! m^{n-m} \mu^n} P_0 = \frac{\rho^n m^m}{m!} P_0$, for $n = m, m + 1, \dots, B$

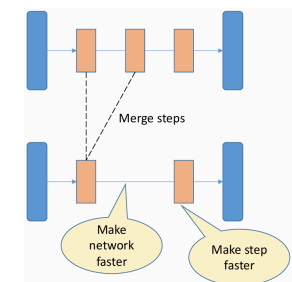
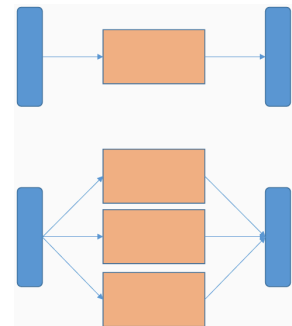
V. Implications for cloud systems

Designing systems for performance & reliability

- When designing cloud systems, we typically optimize for:
 - Throughput
 - Latency
 - Reliability / availability
- Most systems operate under a Service Level Agreement (SLA)
 - Maximize throughput under a latency upper bound
 - At least 99.999...% availability
- The constraints are
 - Cost
 - Complexity

How to improve throughput: scale-out

- Improve throughput via parallelism
 - Increase the number of elements that can process a request, e.g., by adding machines
 - Works well for embarrassingly parallel workload, less well when there are dependencies
- Elastic scaling: add machines when needed, remove them when not needed
- Reduce latency by reducing the length of the critical path. In the cloud:
 - Reduce network communication and hops across machines (less steps)
 - Making every step faster (more powerful machines, accelerators)
 - Make network faster (send less data, remove network overhead, RDMA)
- Applying consolidation often goes against increasing throughput



The consolidation trade-off

- Latency improves from consolidating components
- That puts more load on the processing node, so either
 - We accept less throughput, or
 - We use larger nodes, or
 - We use hardware accelerators

How to improve reliability

- Improve reliability through replication
 - If one component fails, simply move the workload to other components that are also running
 - Goes with techniques that increase throughput (but makes the system more expensive)
 - Needs to be done to entire system (all layers)
 - Creates consistency problems