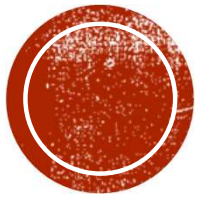
CS559: SYSTEM PERFORMANCE

Dr. Gagan Raj Gupta



SYSTEM PERFORMANCE METRICS

9:00
9:05 > 5 min

METRICS OF SYSTEM PERFORMANCE

- **Response time** is the time from when a job arrives until it completes service, a.k.a. sojourn time.

$T = T_{depart} - T_{arrive}$

We are interested in $E[T]$; $Var[T]$; tail behavior $P\{T > t\}$; 99% of T

Can be broken down into $T_Q + S$ → Service time
(T_Q ← Wait time in Queue)

* Throughput: (X_i): Rate of Completions at device 'i'
* Device Utilization (ρ_i): Fraction of time device 'i' is busy.

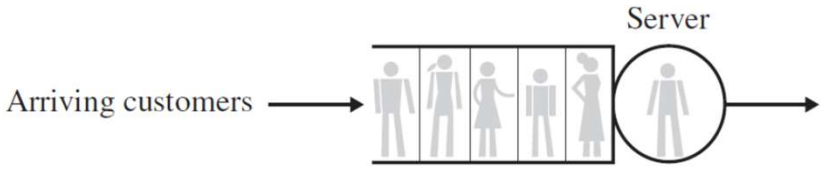
$X_i = \rho_i \mu_i$ Or $\rho_i = X_i E[S]$
{Utilization Law}

T → Period of Observation
B → Total time device is busy
C → # of Job Completions

$\rho_i = \frac{B}{T}$

$X_i = \frac{C}{T}$

QUEUING THEORY 101

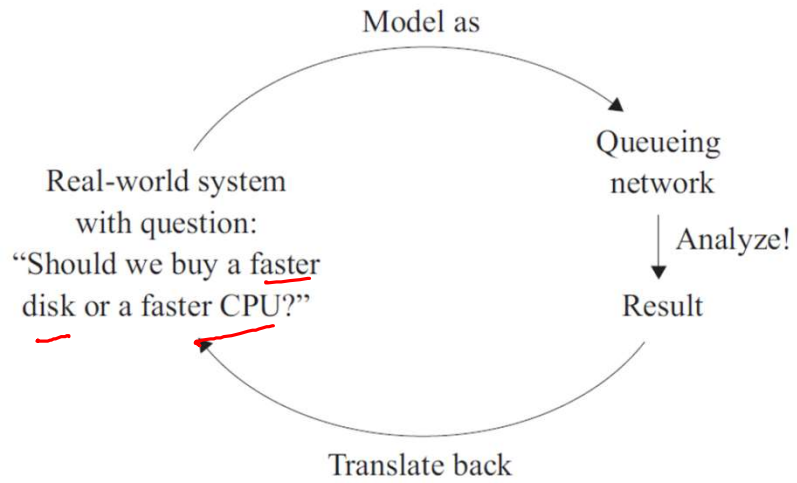


- Bank: Waiting in line at the Bank/ATM, you may have wondered why there are not more tellers/machine
- Super-market: Why the express lane is for 8 items or less rather than 15 items or less, or whether it might be best to actually have *two* express lanes, one for 8 items or less and the other for 15 items or less?

MORE EXAMPLES

- CPU uses a time-sharing scheduler to serve a queue of jobs waiting for CPU time.
- Computer disk serves a queue of jobs waiting to read or write blocks.
- Router in a network serves a queue of packets waiting to be routed. The router queue is a finite capacity queue, in which packets are dropped when demand exceeds the buffer space.
- Memory banks serve queues of threads requesting memory blocks.
- Databases sometimes have lock queues, where transactions wait to acquire the lock on a record.
- Server farms consist of many servers, each with its own queue of jobs.

ANALYTICAL THINKING



- Using mathematical models to answer questions, solve problems

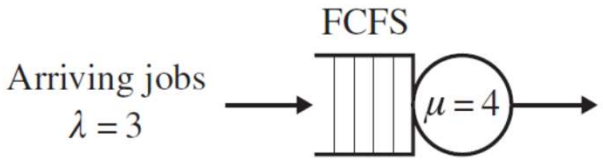
GOALS OF A QUEUING THOERIST

- ***Predicting* the system performance:** Predicting mean delay or delay variability or the probability that delay exceeds some Service Level Agreement (SLA). However, it can also mean predicting the number of jobs that will be queueing or the mean number of servers being utilized (e.g., total power needs), or any other such metric.
- Finding a **superior system *design*** to improve performance. Commonly this takes the form of capacity planning, where one determines which additional resources to buy to meet delay goals (e.g., is it better to buy a faster disk or a faster CPU, or to add a second slow disk). Many times, however, without buying any additional resources at all, one can improve performance just by deploying a smarter scheduling policy or different routing policy to reduce delays.
- Given the importance of smart scheduling in computer systems, all of Week 10 of this course is devoted to understanding scheduling policies.

QUEUEING THEORY AS PREDICTIVE TOOL

- For example, one might be analyzing a network, with certain bandwidths, where different classes of packets arrive at certain rates and follow certain routes throughout the network simultaneously.
- Then queueing theory can be used to compute quantities such as the mean time that packets spend waiting at a particular router *i*, the distribution on the queue buildup at router *i*, or the mean overall time to get from router *i* to router *j* in the network.

SINGLE-SERVER NETWORK



PS

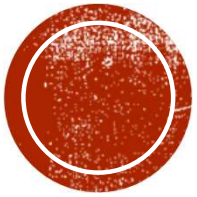
- **Service Order:** Order in which jobs will be served by the server, e.g. First-Come-First-Served (FCFS).
- **Average Arrival Rate:** This is the average rate, λ, at which jobs arrive to the server (e.g., $\lambda = 3$ jobs/sec).
- **Mean Interarrival Time:** This is the average time between successive job arrivals (e.g., $1/\lambda = 1/3$ sec).
- **Service Requirement, Size:** The "size" of a job is typically denoted by the random variable S. This is the time it would take the job to run on this server if there were no other jobs around (no queueing). In a queueing model, the size (a.k.a. service requirement) is typically associated with the server (e.g., this job will take 5 seconds on this server).
- **Mean Service Time:** This is the expected value of S, namely the average time required to service a job on this CPU, where "service" does not include queueing time. In Figure 2.2, $\mathbf{E}[S] = 1/4$ sec.
- **Average Service Rate** This is the average rate, μ, at which jobs are served (e.g., $\mu = 4$ jobs/sec = $1/\mathbf{E}[S]$).

TRANSLATING TO QUEUEING SYSTEM

- The average arrival rate of jobs is 3 jobs per second.
- Jobs have different service requirements,
  - but the average number of cycles required by a job is 5,000 cycles per job.
  - The CPU speed is 20,000 cycles per second.
- That is, an average of 15,000 cycles of work arrive at the CPU each second, and the CPU can process 20,000 cycles of work a second.
- In the queueing-theoretic way of talking, we would never mention the word "cycle." Instead, we would simply say
  - The average arrival rate of jobs is 3 jobs per second.
  - The average rate at which the CPU can service jobs is 4 jobs per second.

STOCHASTIC MODELS AND ASSUMPTIONS

- Stochastic modeling can be used to represent the service demands of jobs and the interarrival times of jobs as random variables
  - The CPU requirements of UNIX processes might be modeled using a Pareto distribution
  - Arrival process of jobs at a busy web server might be well modeled by a Poisson process
  - Stochastic models can also be used to model dependencies between jobs
  - Not always analytically tractable with respect to solving for performance
- *Markovian assumptions*, such as assuming exponentially distributed service demands or a Poisson arrival process, greatly simplify the analysis
  - The arrival process of book orders on Amazon might be approximated by a Poisson
  - Many independent users, each independently submitting requests at a low rate
  - Beaks down when a new Harry Potter book comes out
  - Sometimes far from reality; e.g. when service demands of jobs are highly variable or correlated
  - There are often ways to work around these assumptions
- Workload modeling is essential!



SCALABILITY

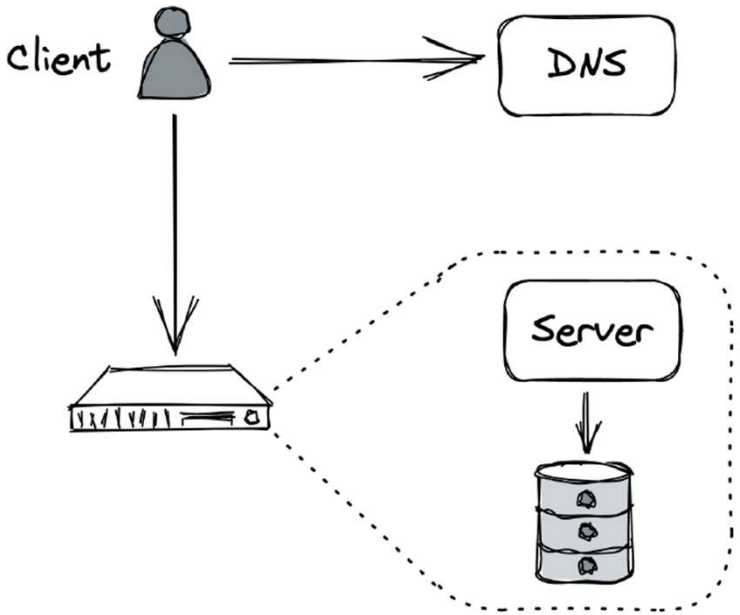
For an application to scale, it must run without performance degradations as load increases.

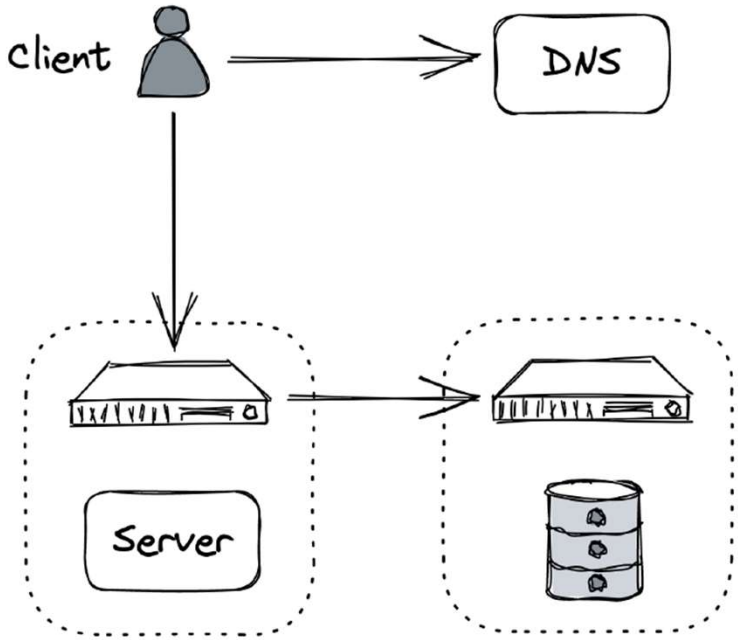
Only long-term solution for increasing the application's capacity is to architect it so that it can **scale horizontally.**

SAMPLE APPLICATION (CRUDER)

- *Cruder* is comprised of a single-page JavaScript application that communicates with an application server through a RESTful HTTP API.
- The server uses the local disk to store large files, like images and videos, and a relational database to persist the application's state.
- Database and the application server are hosted on the same machine, which is managed by a compute platform like AWS EC2.
- Server's public IP address is advertised by a managed DNS service, like AWS Route 53.
- Users interact with *Cruder* through their browsers. Typically, a browser issues a DNS request to resolve the domain name to an IP address (if it doesn't have it cached already), opens a TLS connection with the server, and sends its first HTTP *GET* request to it.

CRUDER





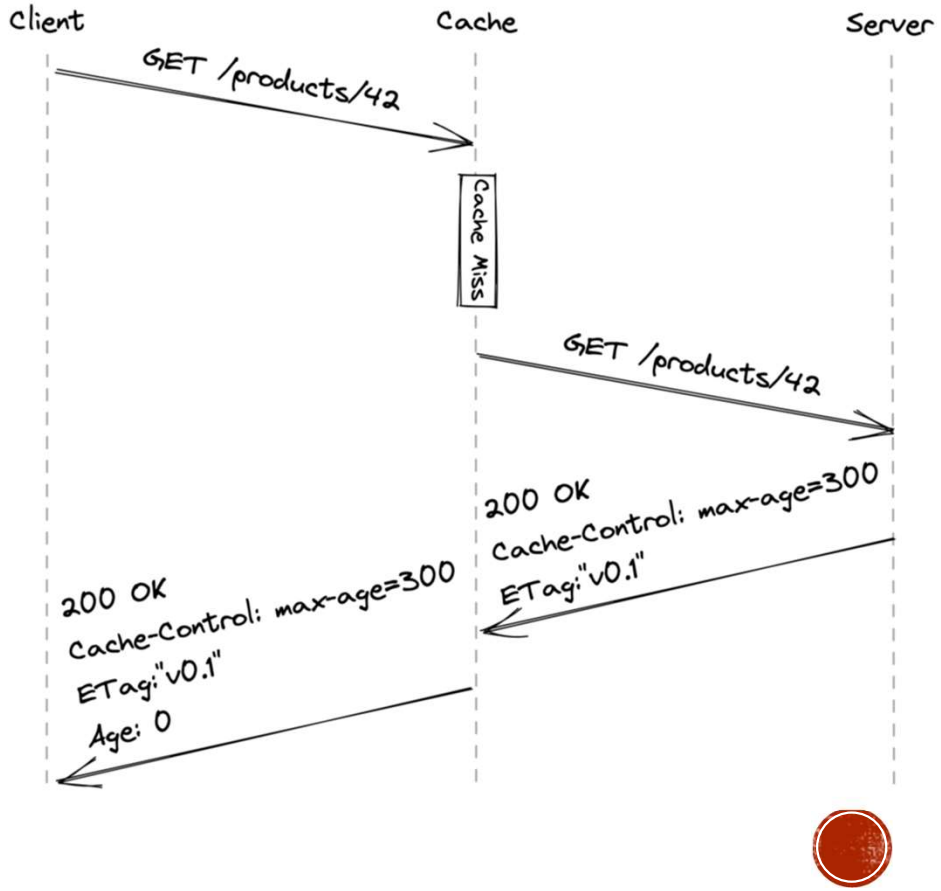
TECHNIQUES

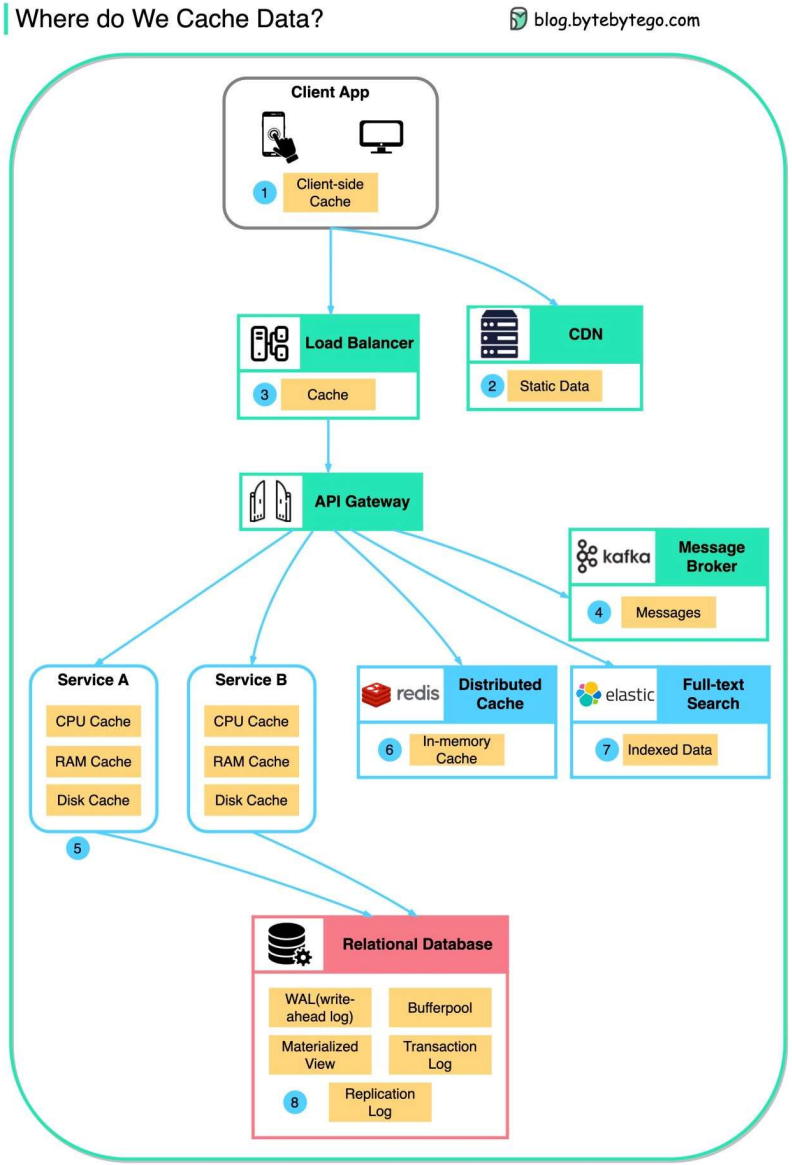
- **Functional decomposition:** breaking down an application into separate components, each with its own well-defined responsibility
- **Partitioning:** splitting data into partitions and distributing them among nodes
- **Replication:** Replicating functionality or data across nodes, also known as horizontal scaling
- Client side caching: HTTP-Caching
- Caching at Reverse-proxy
- CDN (geographically distributed network of reverse proxies)
- Managed File Storage
- Load Balancing

CACHING

we treat the read path (*GET*) differently from the write path (*POST, PUT, DELETE*) because we expect the number of reads to be several orders of magnitude higher than the number of writes.

This is a common pattern referred to as the *Command Query Responsibility Segregation*[3] (CQRS) pattern





There are **multiple layers** along the flow.

1.Client apps: HTTP responses can be cached by the browser. We request data over HTTP for the first time, and it is returned with an expiry policy in the HTTP header; we request data again, and the client app tries to retrieve the data from the browser cache first.

2.CDN: CDN caches static web resources. The clients can retrieve data from a CDN node nearby.

3.Load Balancer: The load Balancer can cache resources as well.

4.Messaging infra: Message brokers store messages on disk first, and then consumers retrieve them at their own pace. Depending on the retention policy, the data is cached in Kafka clusters for a period of time.

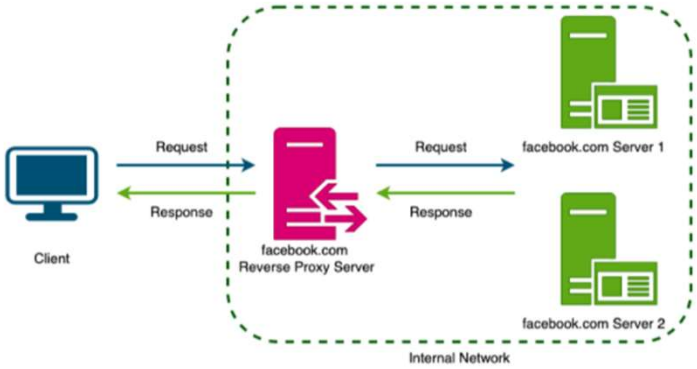
1. **Services**: There are multiple layers of cache in a service. If the data is not cached in the CPU cache, the service will try to retrieve the data from memory. Sometimes the service has a second-level cache to store data on disk.
2. **Distributed Cache**: Distributed cache like Redis hold key-value pairs for multiple services in memory. It provides much better read/write performance than the database.
3. **Full-text Search**: we sometimes need to use full-text searches like Elastic Search for document search or log search. A copy of data is indexed in the search engine as well.
4. **Database**: Even in the database, we have different levels of caches:

- WAL(Write-ahead Log): data is written to WAL first before building the B tree index
- Bufferpool: A memory area allocated to cache query results
- Materialized View: Pre-compute query results and store them in the database tables for better query performance
- Transaction log: record all the transactions and database updates
- Replication Log: used to record the replication state in a database cluster

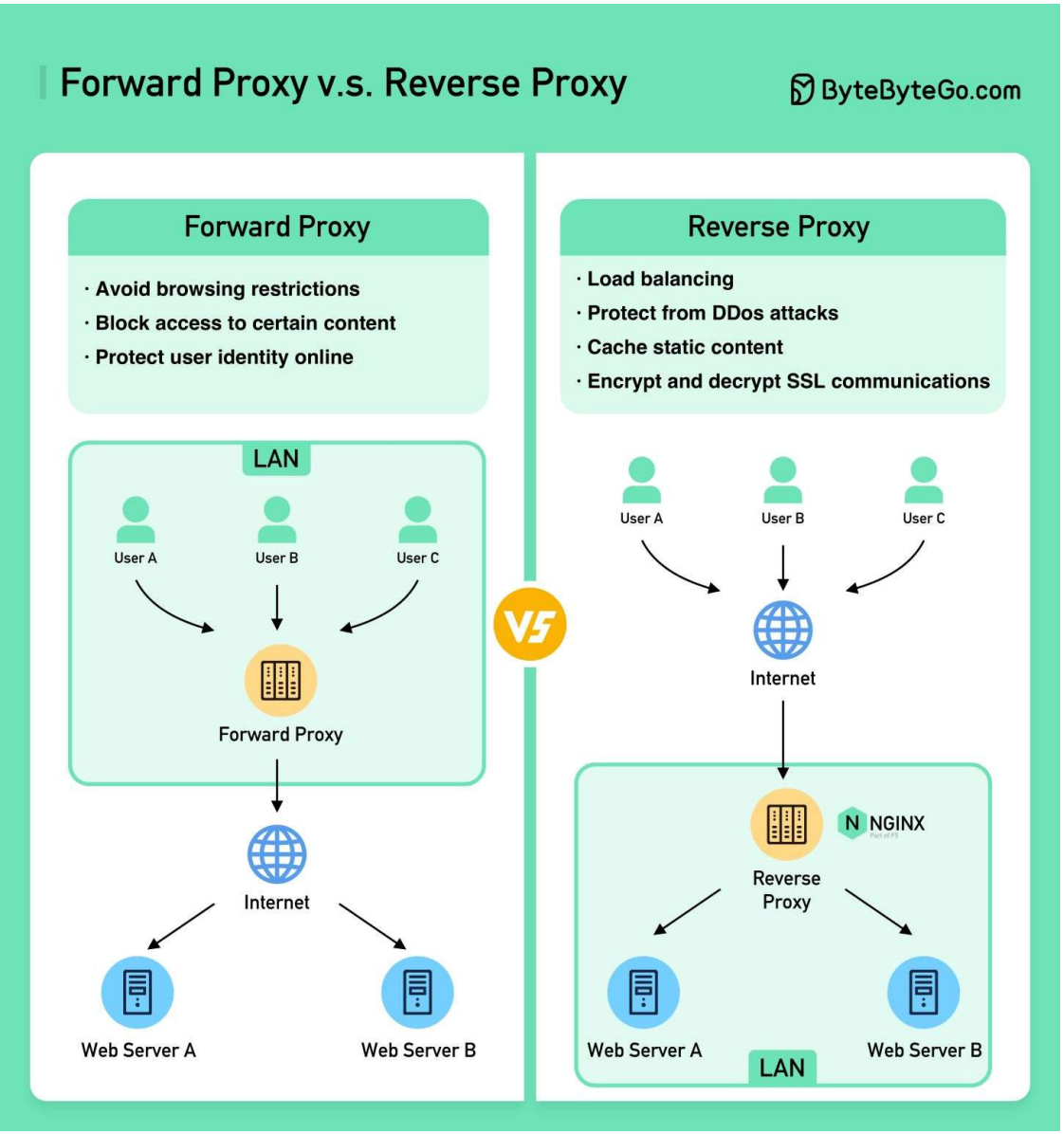
LOAD BALANCERS

- ELB comes in three variants:
  - Classic: Legacy option for very old AWS accounts
  - ALBs: Proper reverse proxies that sit between your application and the internet
  - NLB: Works by routing packets from clients to servers (transparently)
- Application Load Balancers (ALB)
  - Every request to your application gets handled by the load balancer first
  - LB makes another request to your application and finally forwards the response from your application to the caller.
  - Features: support sophisticated routing rules, redirects, responses from Lambda functions, authentication, sticky sessions, and many other things
- Both ALBs and NLBs support TLS/HTTPS, using for example AWS Certificate Manager.
  - AWS becomes the man-in-the-middle is able to see all your data! (decrypts and encrypts)
  - Alternative is to use TCP passthrough and manage certificates on applications hosts, renewals etc.

REVERSE PROXIES



- A reverse proxy is a web server that centralizes internal services and provides unified interfaces to the public.
- Requests from clients are forwarded to a server that can fulfill it before the reverse proxy returns the server's response to the client.
- The identity of the actual server that served the request is hidden by the reverse proxy
- Additional benefits include:
  - **Increased security** - Hide information about backend servers, blacklist IPs, limit number of connections per client
  - **Increased scalability and flexibility** - Clients only see the reverse proxy's IP, allowing you to scale servers or change their configuration
  - **SSL termination** - Decrypt incoming requests and encrypt server responses so backend servers do not have to perform these potentially expensive operations
    - Removes the need to install X.509 certificates on each server
  - **Compression** - Compress server responses
  - **Caching** - Return the response for cached requests
  - **Static content** - Serve static content directly
    - HTML/CSS/JS; Photos; Videos

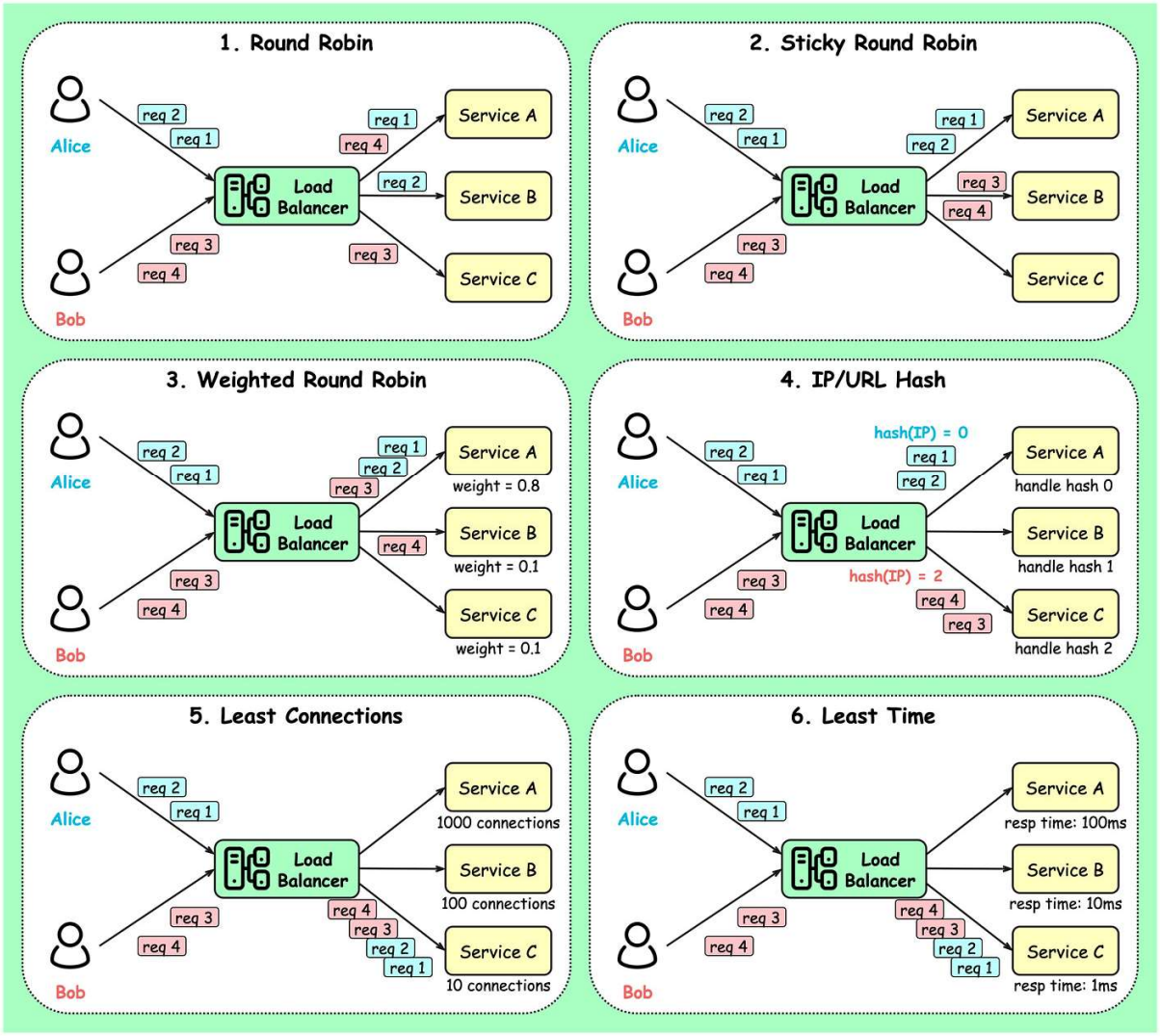


A **forward proxy** is a server that sits between user devices and the internet.

A **reverse proxy** is a server that accepts a request from the client, forwards the request to web servers, and returns the results to the client as if the proxy server had processed the request.

LOAD BALANCING ALGORITHMS

- Load balancers choose the backend servers based on some criterion
  - They must be "healthy", as determined by regular "health checks"
- Among the healthy servers choose one based on:
  - **Least Connection Method** — Server with the fewest active connections
    - Useful when there are a large number of persistent client connections which are unevenly distributed
  - **Least Response Time Method** — fewest active connections and the lowest average response time
  - **Least Bandwidth Method** - currently serving the least amount of traffic measured in Mbps
  - **Round Robin Method** — Cycles through a list of servers and sends each new request to the next server. When it reaches the end of the list, it starts over at the beginning.
    - It is most useful when the servers are of equal specification and there are not many persistent connections.
  - **Weighted Round Robin Method** —Each server is assigned a weight (an integer value that indicates the processing capacity).
    - Servers with higher weights receive new connections before those with less weights and servers with higher weights get more connections than those with less weights.
  - **IP Hash** — hash of the IP address of the client is calculated to redirect the request to a server



Static Algorithms

- Round robin: The client requests are sent to different service instances in sequential order. The services are usually required to be stateless.
- Sticky round-robin: This is an improvement of the round-robin algorithm. If Alice's first request goes to service A, the following requests go to service A as well.
- Weighted round-robin: The admin can specify the weight for each service. The ones with a higher weight handle more requests than others.
- Hash: This algorithm applies a hash function on the incoming requests' IP or URL. The requests are routed to relevant instances based on the hash function result.

Dynamic Algorithms

- Least connections: A new request is sent to the service instance with the least concurrent connections.
- Least response time: A new request is sent to the service instance with the fastest response time.

POWER OF 2 LOAD BALANCING

- **Random with Two Choices load balancing** –
  - When clustering load balancers, it is important to use a load-balancing algorithm that does not inadvertently overload individual backend servers.
  - Random with Two Choices load balancing is extremely efficient for scaled clusters and is the default method in NGINX Ingress Controller for Kubernetes.
- Instead of making the absolute best choice using incomplete data, with "power of two choices" you pick two queues at random and chose the better option of the two, *avoiding the worse choice*.
- "Power of two choices" is efficient to implement.
- And, perhaps unintuitively, it works better at scale than the best-choice algorithms.
- It avoids the undesired herd behavior by the simple approach of avoiding the worst queue and distributing traffic with a degree of randomness.

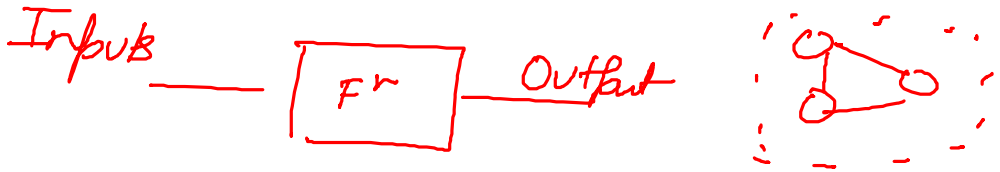
https://www.eecs.harvard.edu/~michaelm/postscripts/mythesis.pdf
https://www.nginx.com/blog/nginx-power-of-two-choices-load-balancing-algorithm/
https://cpb-us-w2.wpmucdn.com/sites.coecis.cornell.edu/dist/9/287/files/2019/08/Srikant-1-YinSriKan14tech.pdf

ADVANCED TECHNIQUES

- Scaling out relational DBs
- Caching (subtle issues)
- Microservices, API Gateway
- Separation of Control Plane and Data Plane
- Asynchronous Messaging channels to decouple communication between services

OPEN AND CLOSED SYSTEMS

MOTIVATION



- *Approach 1: First* build the system and *then* make changes to the policies to improve system performance. For an existing system, *simulate* the system and spend many days running their simulation under different workloads waiting to see what happens.
- Approach 2: Mathematically model the system, stochastically characterize the workloads and performance goals, and then analytically derive the performance of the system as a function of workload and input parameters.
- The fields of *analytical modeling* and *stochastic processes* have existed for close to a century, and they can be used to save systems designers huge numbers of hours in trial and error while improving performance.
- Analytical modeling can also be used in conjunction with simulation to help guide the simulation, reducing the number of cases that need to be explored.

Table 3.2. *Discrete and continuous distributions*

Distribution	**p.m.f. $p_X(x)$**	**Mean**	**Variance**
Bernoulli(p)	$p_X(0) = 1-p\,;\, p_X(1) = p$	p	$p(1-p)$
Binomial(n,p)	$p_X(x) = \binom{n}{x} p^x (1-p)^{n-x}$, $x = 0, 1, \ldots, n$	np	$np(1-p)$
Geometric(p)	$p_X(x) = (1-p)^{x-1} p, \quad x = 1, 2, \ldots$	$\frac{1}{p}$	$\frac{1-p}{p^2}$
Poisson(λ)	$p_X(x) = e^{-\lambda} \cdot \frac{\lambda^x}{x!}, \quad x = 0, 1, 2, \ldots$	λ	λ
Distribution	**p.d.f. $f_X(x)$**	**Mean**	**Variance**
Exp(λ)	$f_X(x) = \lambda e^{-\lambda x}$	$\frac{1}{\lambda}$	$\frac{1}{\lambda^2}$
Uniform(a,b)	$f_X(x) = \frac{1}{b-a}, \quad \text{if } a \le x \le b$	$\frac{b+a}{2}$	$\frac{(b-a)^2}{12}$
Pareto(α), $0 < \alpha < 2$	$f_X(x) = \alpha x^{-\alpha-1}, \quad \text{if } x > 1$	$\begin{cases} \infty & \text{if } \alpha \le 1 \\ \frac{\alpha}{\alpha-1} & \text{if } \alpha > 1 \end{cases}$	∞
Normal(μ, σ^2)	$f_X(x) = \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{1}{2}\left(\frac{x-\mu}{\sigma}\right)^2}$, $-\infty < x < \infty$	μ	σ^2

OPEN SYSTEMS

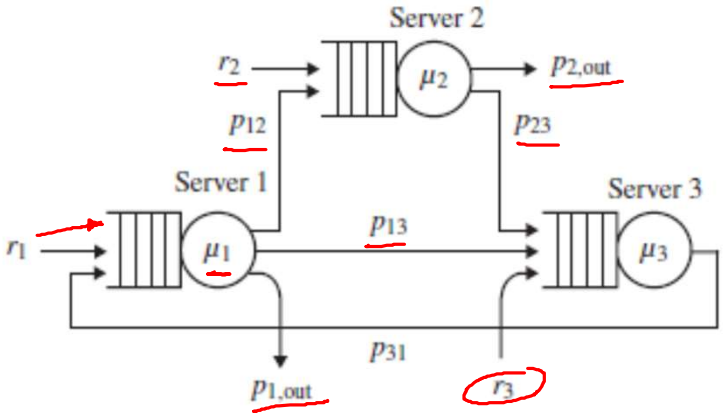
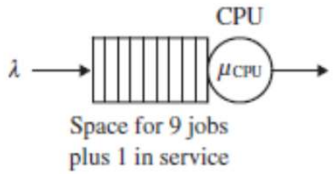
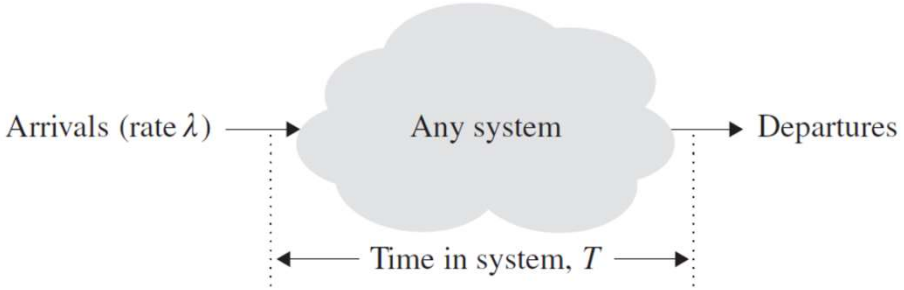


Figure 2.3. Network of queues with probabilistic routing.

Eg. Modeling packet flows in the internet



Eg. Finite buffers in routers and switches



External arrivals and departures

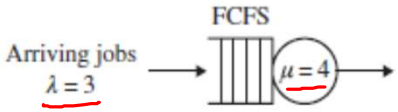
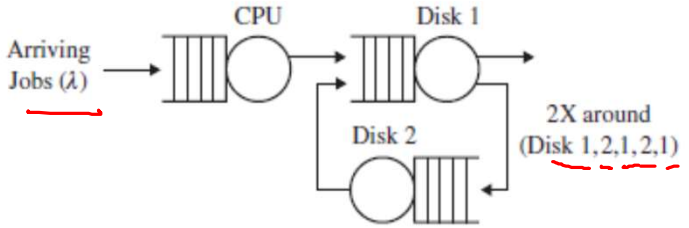


Figure 2.2. Single-server network.



Eg. Fixed service order

THROUGHPUT AND UTILIZATION

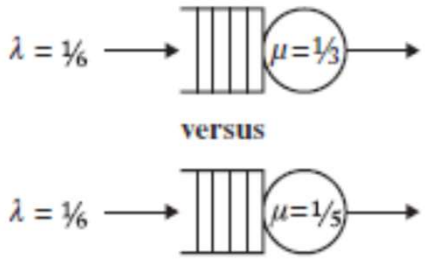


Figure 2.6. Comparing throughput of two systems.

Throughput doesn't depend on the service rate whatsoever!

- Which of the above systems have higher throughput?
- Device throughput (X_i) is the rate of completions at a device i (e.g. jobs/sec)
- System throughput X is the rate of job completions in the system
- We saw Utilization law in last class: $\rho_i = X_i E[S] = \frac{X_i}{\mu}$
- It can be shown that in a stable single server queue: $\rho = \frac{\lambda}{\mu}$

PROBABILISTIC NETWORK OF QUEUES

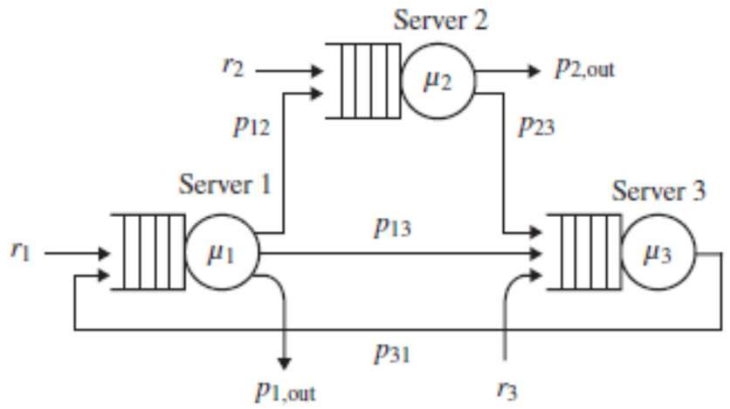
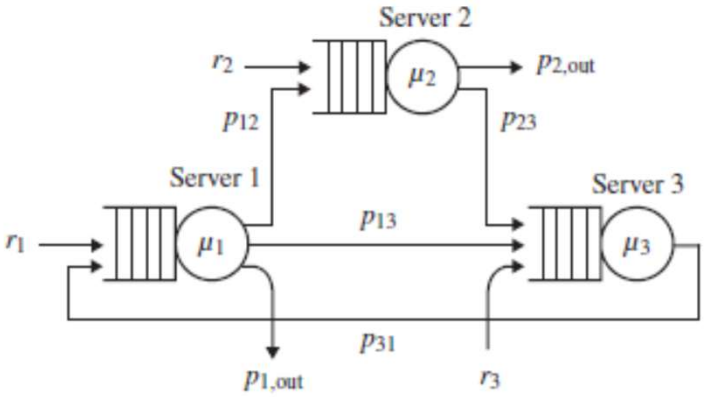


Figure 2.3. Network of queues with probabilistic routing.

- Here throughput, X is given by $\Sigma_i r_i$
- Throughput at server i is given by the solution of simultaneous equations
  - $\lambda_i = r_i + \Sigma_j \lambda_j P_j$
- Stability Condition: $\lambda_i < \mu_i$, for all i

BQ 5.1: STABILITY CONDITIONS

- For the network-of-queues with probabilistic routing each server serves at an average rate of 10 jobs/sec; that is, $\mu i = 10$, $\forall i$.



- Suppose that
  - $r2 = r3 = 1$
  - $p12 = p2,out = 0.8$,
  - $p23 = p13 = 0.2$,
  - $p1,out = 0$, and $p31 = 1$.
- What is the maximum allowable value of $r1$ to keep this system stable?

CLOSED SYSTEMS: INTERACTIVE

Terminals represent users who each send a job to the "central subsystem" and then wait for a response. The central subsystem is a network of queues. A user cannot submit her next job before her previous job returns. Thus, the number of jobs in the system is fixed (equal to the number of terminals). This number is sometimes called the **load** or **MPL** (multiprogramming level).

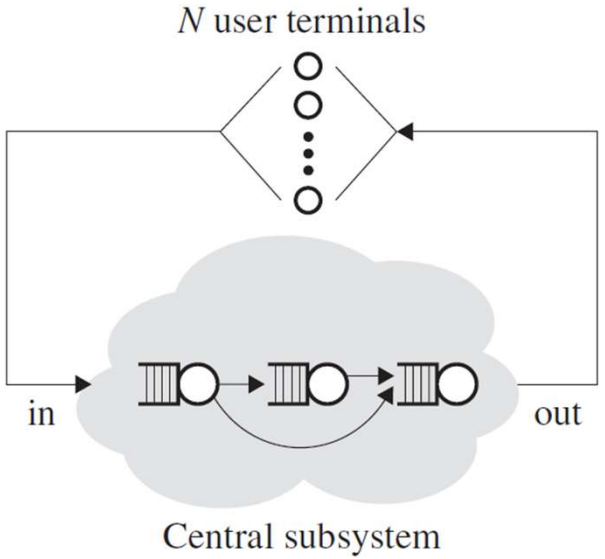


Figure 2.10. Interactive system.

$$\mathbf{E}[T] = \mathbf{E}[R] + \mathbf{E}[Z]$$

- Response time (R) is the time it takes a job to go between "in" and "out"
- T is the *system time* (or "time in system")
- **Goal:** Find a way to allow as many users as possible to get onto the system at once, so they can all get their work done, while keeping $\mathbf{E}[R]$ low enough.

CLOSED SYSTEMS: BATCH

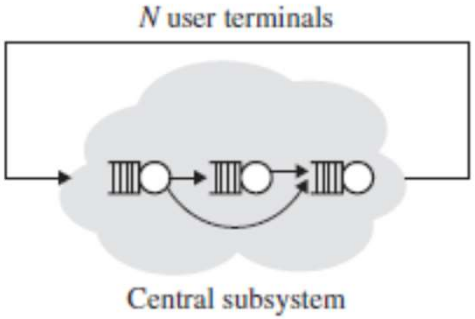


Figure 2.12. Batch system.

- Interactive system with zero think time
- Typical example is that of running many jobs overnight
- As soon as one job completes, another one starts
- Number of jobs may be limited by the system memory and are kept constant at N
- **Goal:** Maximize throughput!

THROUGHPUT IN A CLOSED SYSTEM

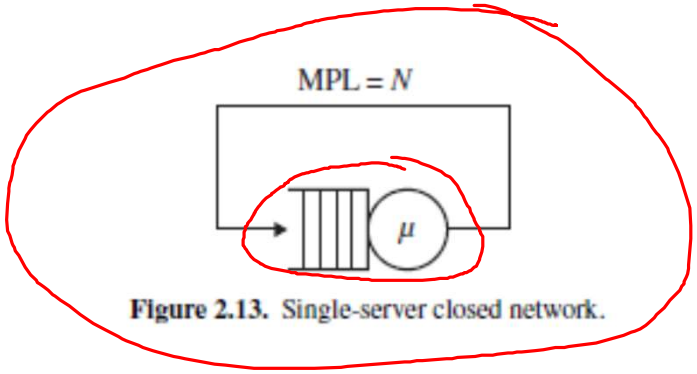


Figure 2.13. Single-server closed network.

Avg. # of completions per sec.

- What is the throughput?
- Answer: $X = \mu$
- Observe that this is *very different* from the case of the open network where throughput was independent of service rate!
- E[R]=$\frac{N}{\mu}$ because every "arrival" waits behind N-1 jobs and then runs
- Note that X and E[R] are inversely related!

HOW DOES IT MATTER?

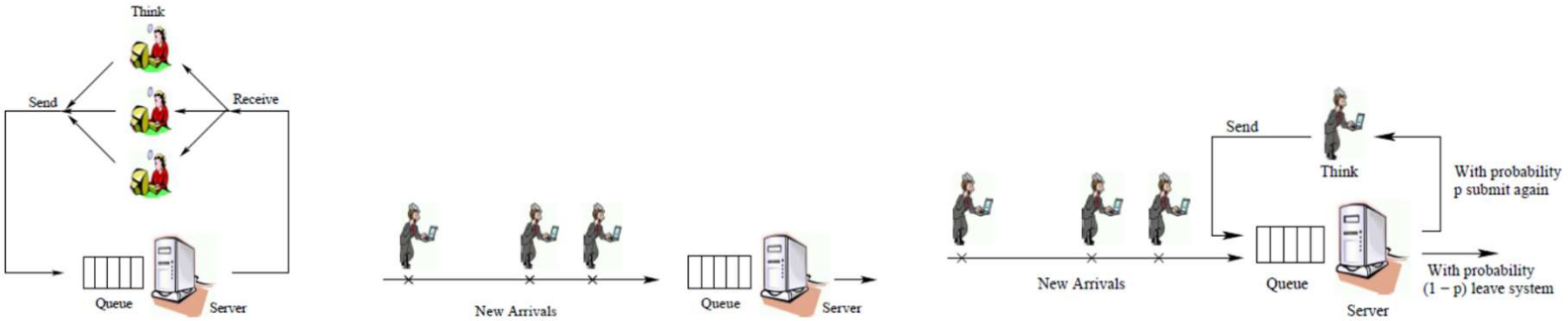


Figure 1: *Illustrations of the closed, open, and partly-open system models.*

- In this paper, we show that closed and open system models yield significantly different results, *even when both models are run with the same load and service demands*.
- Not only is the measured response time different under the two system models, but the two systems respond fundamentally differently to varying parameters and to resource allocation (scheduling) policies.

http://www.cs.cmu.edu/~harchol/Papers/nsdi06.pdf

SUMMARY

Open Systems

- Throughput, X, independent of the μ_i
- X is not affected by doubling the μ_i
- Throughput and response time are *not* related.

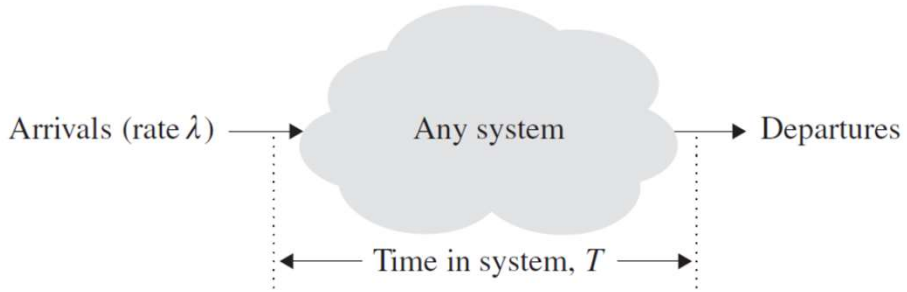
Closed Systems

- X depends on μ_i's.
- If we double all the μ_i's while holding N constant, then X changes.
- In fact for closed systems,
- Higher throughput $\Leftarrow\Rightarrow$ Lower avg. response time.

LITTLE'S LAW

- Single most famous queuing theory result
- It states that the average number of jobs in the system is equal to the product of the average arrival rate into the system and the average time a job spends in the system.
- It also holds when the system consists of just the "queues" in the system.
- Little's Law applies to both open and closed systems, and we explain it in both cases

LITTLE'S LAW FOR OPEN SYSTEMS



For any ergodic open system, $E[N] = \lambda E[T]$,

- where $E[N]$ is the expected number of jobs in the system, λ is the average arrival rate into the system and $E[T]$ is the mean time the jobs spend in the system

Little's Law makes no assumptions about the arrival process, the service time distributions at the servers, the network topology, the service order, or anything!

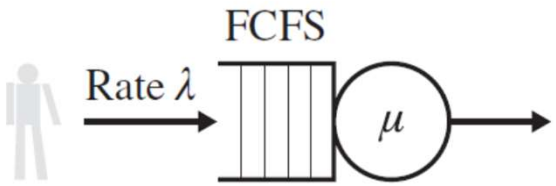
It should seem intuitive that $\mathbf{E}[T]$ and $\mathbf{E}[N]$ are proportional.

Ergodicity: Average behavior of the system can be deduced from the trajectory of a "typical" point.

A FAST-FOOD RESTAURANT

- A fast-food restaurant gets people out fast (low $\mathbf{E}[T]$) and also does not require much waiting room (low $\mathbf{E}[N]$).
- By contrast, a slow-service restaurant gets people out slowly (high $\mathbf{E}[T]$) and therefore needs a lot more seating room ($\mathbf{E}[N]$).
- Thus $\mathbf{E}[T]$ should be directly proportional to $\mathbf{E}[N]$.

SERVICE RATE OR ARRIVAL RATE?



- Think about a single FCFS queue, as shown above.
- A customer arrives and sees $\mathbf{E}[N]$ jobs in the system.
- The expected time for each customer to complete is $1/\lambda$ (not $1/\mu$), because the average rate of completions is λ.
- Hence the expected time until the customer leaves is $E[T] = \frac{1}{\lambda} E[N]$

LITTLE'S LAW (CLOSED SYSTEMS)

Theorem 6.2 (Little's Law for Closed Systems) *Given any ergodic closed system,*

$$N = X \cdot \mathbf{E}[T],$$

where N is a constant equal to the multiprogramming level, X is the throughput (i.e., the rate of completions for the system), and $\mathbf{E}[T]$ is the mean time jobs spend in the system.

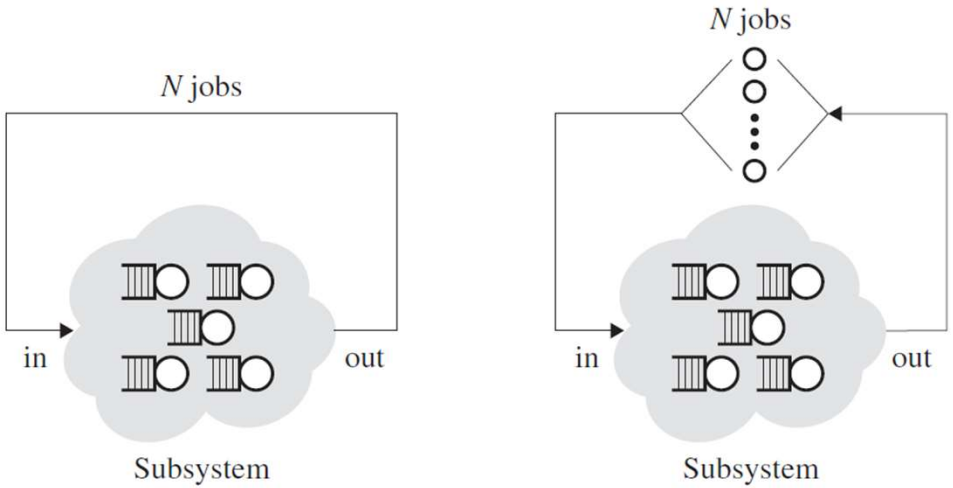


Figure 6.3 shows a batch system and an interactive (terminal-driven) system. Note that for the interactive system (right), the time in system, T, is the time to go from "out" to "out," whereas response time, R, is the time from "in" to "out." Specifically, for a *closed interactive system*, we define $\mathbf{E}[T] = \mathbf{E}[R] + \mathbf{E}[Z]$, where $\mathbf{E}[Z]$ is the average think time, $\mathbf{E}[T]$ is the average time in system, and $\mathbf{E}[R]$ is the average response time. The notation is a little overloaded, in that for open systems and closed batch systems, we refer to $\mathbf{E}[T]$ as mean response time, whereas for closed interactive systems $\mathbf{E}[T]$ represents the mean time in system and $\mathbf{E}[R]$ is the mean response time, since response time does not include thinking.

DETERMINISTIC ROUTING

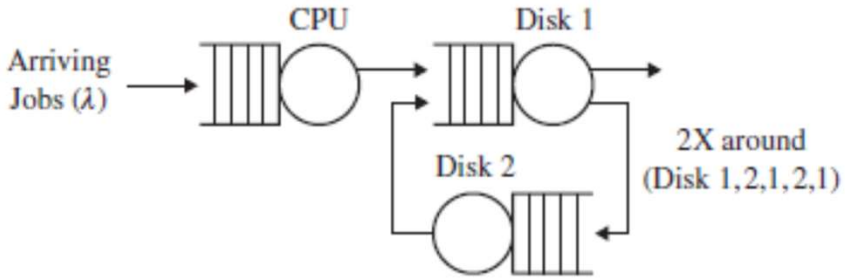
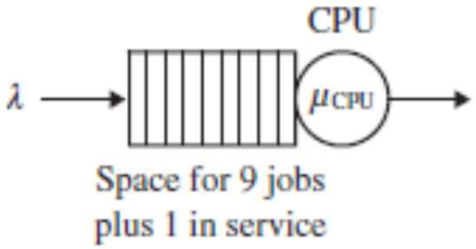


Figure 2.4. Network of queues with non-probabilistic routing.

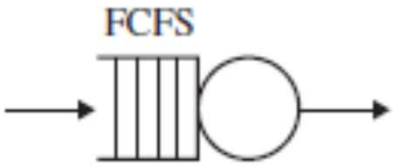
- Here $X = \lambda$
- $X_{Disk1} = 3\lambda$
- $X_{Disk2} = 2\lambda$

FINITE BUFFER



- $X = \rho\mu$
- In general, some arrivals will get dropped, $X < \lambda$
- Exact analysis is need to determine ρ

BQ 5.2: SLOWDOWN IN A SYSTEM



- Jobs arrive in a system at rate $\lambda = \frac{1}{2}\ job/sec$ and it serves them in FCFS order
- The job sizes (service times) are independently and identically distributed according to random variable S where S =1 with probability ¾ and 2 otherwise .
- You have measured the mean response time: $\mathbf{E}[T] = \frac{29}{12}$
- Based on this information, compute the mean slowdown, **E**[Slowdown],
- Slowdown(j) = $\frac{T(j)}{S(j)}$, where $T(j)$ is the response time of job j and $S(j)$ is the size of job j.
- (b) If the service order in part (a) had been Shortest-Job-First (SJF), would the same technique have worked for computing mean slowdown?

BQ 5.3: SLOWDOWN WITH SRPT

- For a single-server CPU, where jobs arrive according to some process, let SRPT denote the *preemptive* scheduling policy that always serves the job with the currently Shortest-Remaining-Processing-Time (assume one knows this information). It is claimed that for any *arrival sequence*, consisting of the arrival time and size of every job, SRPT scheduling minimizes mean response time over that arrival sequence. Prove or disprove this claim.
- The slowdown of a job is defined as the job's response time divided by its service requirement. (i) Mean slowdown is thought by many to be a more important performance metric than mean response time. Why do you think this is? (ii) It seems intuitive that the SRPT scheduling policy should minimize mean slowdown. Prove or disprove this hypothesis.

BQ 5.4: PARAM SUPERCOMPUTER

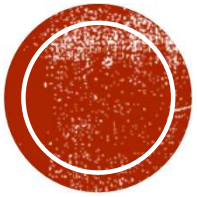
- The PARAM supercomputer runs large parallel jobs for scientists from all over the country. To charge users appropriately, jobs are grouped into different bins based on the number of CPU hours they require, each with a different price.
- Suppose that job durations are Exponentially distributed with *mean* 1,000 processor hours. Further suppose that all jobs requiring less than 500 processor-hours are sent to bin 1, and all remaining jobs are sent to bin 2.
- **Question:** Consider the following questions:

  (a) What is **P**{Job is sent to bin 1}?

  (b) What is **P**{Job duration < 200 | job is sent to bin 1}?

  (c) What is the conditional density of the duration X, $f_{X|Y}(t)$, where Y is the event that the job is sent to bin 1?

  (d) What is **E**[Job duration | job is in bin 2]?



MODIFICATION ANALYSIS

MOTIVATION AND OBJECTIVE

- Many times, we are asked "what-if" questions about design changes and their impacts to system performance (response time, reliability, throughput, …)
- We also need to account for cost and decide about the best investment
- In this lecture, we study:
  - Examples of modification analysis on open systems
  - Asymptotic bounds for closed systems
  - Examples of modification analysis for closed systems
- Finally, we discuss the topic of difference between open and closed systems again

REVIEW OF OPERATIONAL LAWS

- Little's Law for Open Systems: $E[N] = \lambda E[T]$
- Queueing in Open Systems: $E[N_Q] = \lambda E[T_Q]$
- Specific jobs in Open Systems: $E[N_{red}] = \lambda_{red} E[T_{red}]$
- Little's Law for Closed Batch System (Zero think time): $\mathrm{N} = X.E[T]$
- Response Time Law for Closed Interactive Systems: $E[R] = \frac{N}{X} - E[Z]$
- Utilization Law (Single server i): $\rho_i = \frac{\lambda_i}{\mu_i} = \lambda_i E[S_i]$
- Forced Flow Law: $X_i = E[V_i].X$
- Bottleneck Law: $\rho_i = X.E[D_i]$

Operational laws are laws that hold independently of any assumptions about the distribution of arrivals or the distribution of service times (job sizes). They are extremely useful and simple to apply.

ASYMPTOTIC BOUNDS FOR CLOSED SYSTEMS

- By knowing the Device service demands, D_i's alone, we are able to:
  - Estimate X and E[R] as a function of the multi-programming level, N
  - Determine which changes to the systems will be worthwhile and which won't
- Let m be the number of devices in the system.
  - $D = \sum_{i=1}^{m} E[D_i]$ and
  - $D_{max} = \max_i\{E[D_i]\}$
- Theorem: For any closed interactive system with N terminals,

$$X \leq \min\left(\frac{N}{D + E[Z]}, \frac{1}{D_{max}}\right)$$

$$E[R] \geq \max(D, N.D_{max} - E[Z])$$

Important: The first term on r.h.s. is an asymptote for small N and second term for large N

PROOF:

$$X \leq \min\left(\frac{N}{D + E[Z]}, \frac{1}{D_{max}}\right)$$

$$E[R] \geq \max(D, N.D_{max} - E[Z])$$

- Large N asymptotes
  - $\forall_i\ X.E[D_i] = \rho_i \leq 1 \Rightarrow X \leq \frac{1}{E[D_i]} \Rightarrow X \leq \frac{1}{D_{max}}$
  - $E[R] = \frac{N}{X} - E[Z] \geq N.D_{max} - E[Z]$

Insight: Improve your bottleneck device first!

- Small N asymptotes
  - Let E[R(N)] denote response time with MPL, N
  - $E[R(N)] \geq E[R(1)] = D = \sum_{i=1}^{m} D_i$
  - $X = \frac{N}{E[R]+E[Z]} \leq \frac{N}{D+E[Z]}$

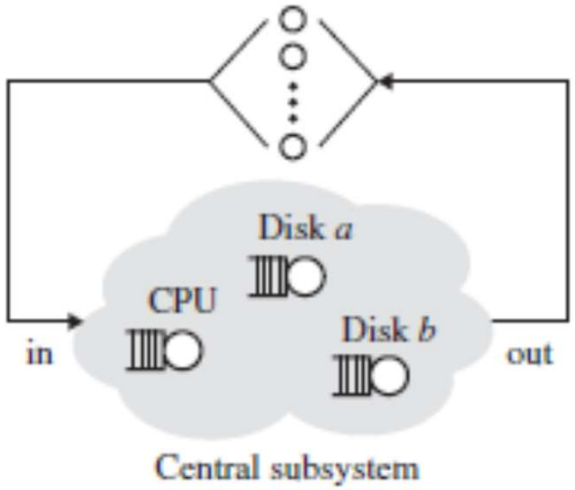
Insight: All devices need improvement!

Important: Power of this theorem is that these are tight for large or small N.
Even tighter estimate are possible using balanced bounds! (Sec 5.4 of LZCS book)

EXAMPLE OF BOUNDS

$$X \leq \min\left(\frac{N}{D + E[Z]}, \frac{1}{D_{max}}\right)$$

$$E[R] \geq \max(D, N.D_{max} - E[Z])$$



E[Z]=18

$E[D_{CPU}] = 5\ sec$

$E[D_{disk\ a}] = 4\ sec$

$E[D_{disk\ b}] = 3\ sec$

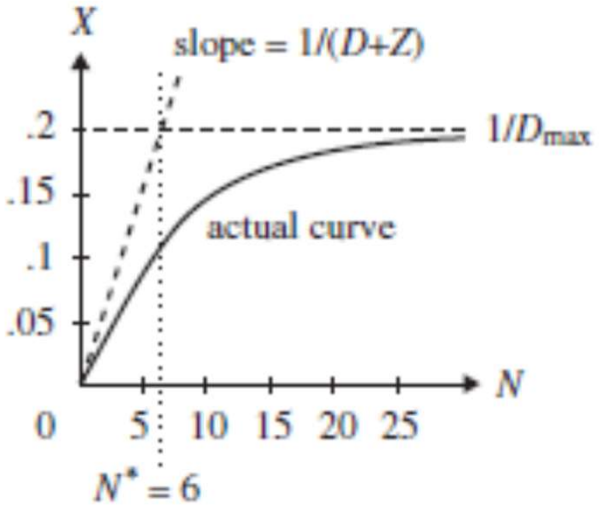
Goal: Determine X and E[R]?

- D = 5+4+3 = 12 sec
- $D_{max} = 5$ sec (CPU is the bottleneck device)
- $X \leq \min\left\{\frac{N}{30}, \frac{1}{5}\right\}$ and $E[R] \geq \max\{12, 5N - 18\}$

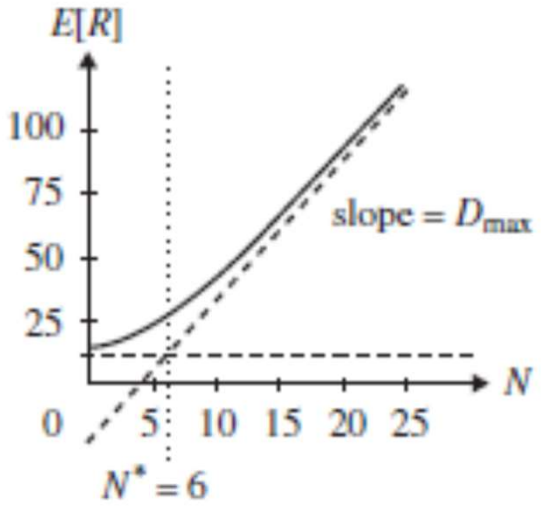
ASYMPTOTES

$$X \leq \min\left(\frac{N}{D + E[Z]}, \frac{1}{D_{max}}\right)$$

$$E[R] \geq \max(D, N.D_{max} - E[Z])$$



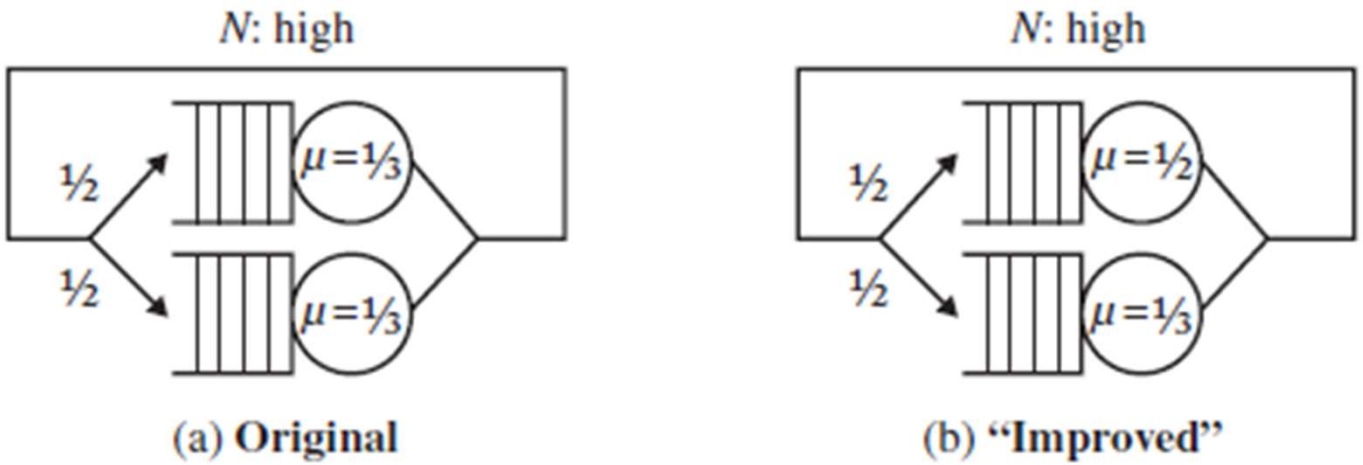
(a) X versus N



(b) $E[R]$ versus N

- The point where the asymptotes cross is denoted by N*= $\frac{D+E[Z]}{D_{\max}}$
- Beyond this point there must be some queuing in the system, (E[R]>D).
- For fixed N>N*, to get more throughput, one must decrease $D_{\max}$. Other changes will be ineffective.

DOES THIS IMPROVEMENT HELP?



- No effect because in the high N regime, $D_{\{max\}}$ is the main factor
- $D_{\{max\}}$ has not changed

WHICH IS BETTER?

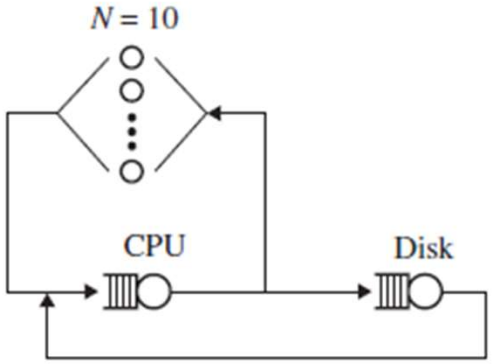


Figure 7.4. Simple closed system.

- **System A** looks like Figure 7.4 with *D*cpu = 4.6 and *D*disk = 4.0.
- **System B** looks like Figure 7.4 with *D*cpu = 4.9 and *D*disk = 1.9 (a slightly slower CPU and a much faster disk)
- Question: Which of the systems has a higher throughput?

MEASURING AN INTERACTIVE SYSTEM

- T = 650 seconds (the length of the observation interval)
- B_cpu = 400 seconds
- $B\_slowdisk$ = 100 seconds
- $B\_fastdisk$ = 600 seconds
- $C = C\_cpu$ = 200 jobs
- $C\_slowdisk$ = 2,000 jobs
- $C\_fastdisk$ = 20,000 jobs
- $\mathbf{E}[Z]$ = 15 seconds
- N = 20 users

- $D_{cpu} = B_{cpu}/C = 400 \text{ sec}/200 \text{ jobs} = 2.0 \text{ sec/job}$
- $D_{slowdisk} = B_{slowdisk}/C = 100 \text{ sec}/200 \text{ jobs} = 0.5 \text{ sec/job}$
- $D_{fastdisk} = B_{fastdisk}/C = 600 \text{ sec}/200 \text{ jobs} = 3.0 \text{ sec/job}$
- $\mathbf{E}[V_{cpu}] = C_{cpu}/C = 200 \text{ visits}/200 \text{ jobs} = 1 \text{ visit/job}$
- $\mathbf{E}[V_{slowdisk}] = C_{slowdisk}/C = 2{,}000 \text{ visits}/200 \text{ job} = 10 \text{ visits/job}$
- $\mathbf{E}[V_{fastdisk}] = C_{fastdisk}/C = 20{,}000 \text{ visits}/200 \text{ job} = 100 \text{ visits/job}$
- $\mathbf{E}[S_{cpu}] = B_{cpu}/C_{cpu} = 400 \text{ sec}/200 \text{ visits} = 2.0 \text{ sec/visit}$
- $\mathbf{E}[S_{slowdisk}] = B_{slowdisk}/C_{slowdisk} = 100 \text{ sec}/2{,}000 \text{ visits} = .05 \text{ sec/visit}$
- $\mathbf{E}[S_{fastdisk}] = B_{fastdisk}/C_{fastdisk} = 600 \text{ sec}/20{,}000 \text{ visits} = .03 \text{ sec/visit}$

- The above are typically easy-to-measure quantities. In this example, we examine four possible improvements (modifications) – hence the name “modification analysis.”

- **1. Faster CPU:** Replace the CPU with one that is twice as fast.
- **2. Balancing slow and fast disks:** Shift some files from the fast disk to the slow disk, balancing their demand.
- **3. Second fast disk:** Buy a second fast disk to handle half the load of the busier existing fast disk.
- **4. Balancing among three disks plus faster CPU:** Make all three improvements together: Buy a second fast disk, balance the load across all three disks, and also replace the CPU with a faster one.

1. **Faster CPU:** Originally, $D_{\text{max}} = 3$ sec/job, $D = 5.5$, $N^* = \frac{20.5}{3} \approx 7 \ll N$. $D_{\text{cpu}} \to 1$ sec/job does not change $D_{\text{max}} = 3$ sec/job. Notice that N^* hardly changes at all. The fast disk is the bottleneck. We can never get more than 1 job done every 3 seconds on average.
2. **Balancing slow and fast disks:** Shift some files from the fast disk to the slow disk, balancing their demand. To do this we need that

$$V_{\text{slow}} + V_{\text{fast}} = 110 \text{ as originally}$$

$$\text{but } S_{\text{slow}} \cdot V_{\text{slow}} = S_{\text{fast}} \cdot V_{\text{fast}} \text{ because we are balancing the demand.}$$

Solving this system of linear equations yields the new demands $D_{\text{slow}} = D_{\text{fast}} = 2.06$. Now, $D_{\text{max}} = 2.06$ sec/job, although D increases slightly because some files have been moved from the fast disk to the slow disk.

3. **Second fast disk:** We keep $D_{\text{slow}} = 0.5$, the same as before. However, we buy a second fast disk to handle half the load of the original fast disk. So now

$$D_{\text{fast1}} = D_{\text{fast2}} = 1.5 \text{ sec/job}.$$

Thus our new $D_{\max}$ is 2.0 sec/job (the CPU becomes the bottleneck).

4. **Balancing among three disks plus faster CPU:** We now make the CPU faster *and* balance load across all three disks, so

$$V_{\text{slow}} + V_{\text{fast1}} + V_{\text{fast2}} = 110.$$
$$S_{\text{slow}} \cdot V_{\text{slow}} = S_{\text{fast1}} \cdot V_{\text{fast1}} = S_{\text{fast2}} \cdot V_{\text{fast2}}.$$

Solving these simultaneous equations yields: $D_{\text{disk1}} = D_{\text{disk2}} = D_{\text{disk3}} = 1.27$. So $D_{\max} = 1.27$, since we cut D_{cpu} to 1 already.

A graph of the results is shown in Figure 7.5. Assuming N is not too small, we conclude the following:

- Change 1 is insignificant.
- Changes 2 and 3 are about the same, which is interesting because change 2 was achieved without any hardware expense.
- Change 4 yields the most dramatic improvement.

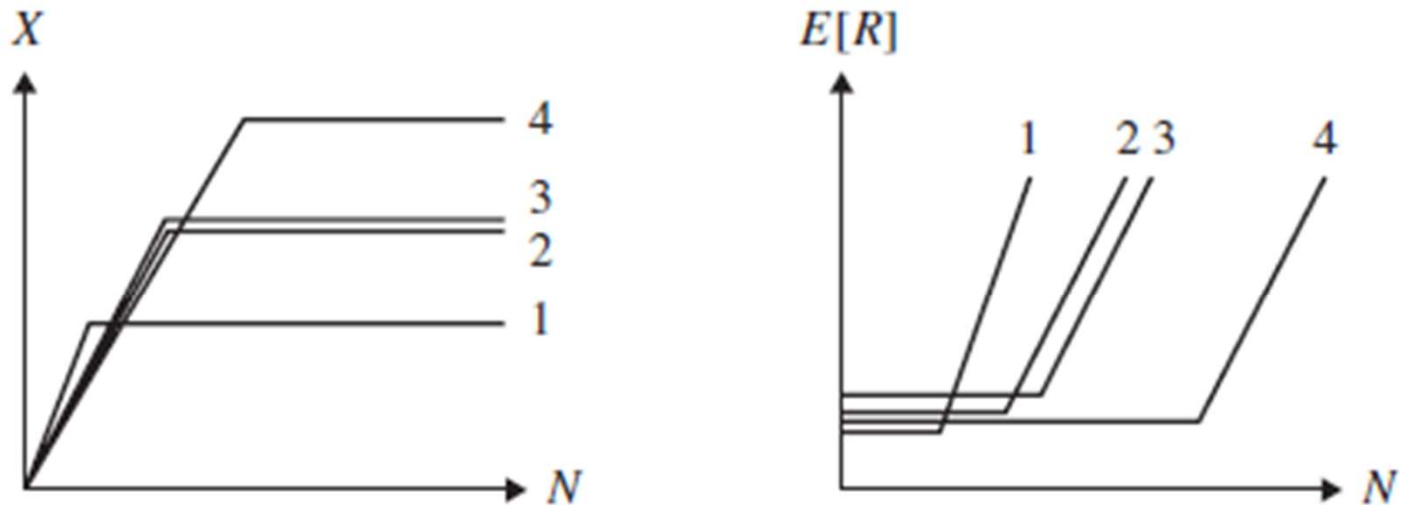


Figure 7.5. Throughput and response time versus N, showing the effects of four possible improvements from the harder example, where the improvements are labeled 1, 2, 3, and 4.

SOME PUZZLES FOR SYSTEM DESIGNERS

1. Given a choice between a single machine with speed s, or n machines each with speed s/n, which should we choose?
2. If both the arrival rate and service rate double, will the mean response time stay the same?
3. Should systems really aim to balance load, or is this a convenient myth?
4. If a scheduling policy favors one set of jobs, does it necessarily hurt some other jobs, or are these "conservation laws" being misinterpreted?
5. Do greedy, shortest-delay, routing strategies make sense in a server farm, or is what is good for the individual disastrous for the system as a whole?
6. How do high job size variability and heavy-tailed workloads affect choice of scheduling policy?
7. How should one trade off energy and delay in designing a computer system?
8. If 12 servers are needed to meet delay guarantees when the arrival rate is 9 jobs/sec, will we need 12,000 servers when the arrival rate is 9,000 jobs/sec?

DP1 : DOUBLING ARRIVAL RATE

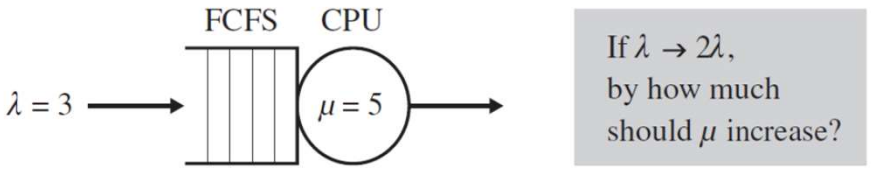


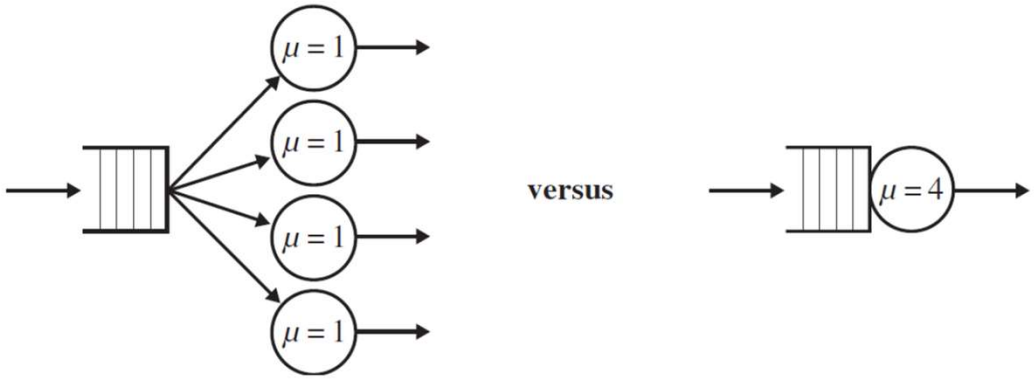
Figure 1.2. A system with a single CPU that serves jobs in FCFS order.

- System consists of a single CPU that serves a queue of jobs in First-Come-First-Served (FCFS)
- Jobs arrive according to some random process with average arrival rate, say $\lambda = 3$ jobs per sec
- Each job has some CPU service requirement, drawn independently from some distribution of job service requirements (we can assume any distribution on the job service requirements)
  - Let's say that the average service rate is $\mu = 5$ jobs per second
  - Note that the system is not in overload ($3 < 5$).
  - Let $\mathbf{E}[T]$ denote the mean response time of this system
- **Q:** Arrival rate is going to double tomorrow. How much should you increase the CPU speed?
  - (a) Double the CPU speed; (b) More than double the CPU speed; (c) Less than double the CPU speed.

SOLUTION TO DP1

- Doubling CPU speed together with doubling the arrival rate will generally result in cutting the mean response time in half!
  - Suppose we have a clock B which runs twice as fast as the normal clock {Frame of Reference}
  - The arrival rate will appear to be the same in this reference B.
  - For the system with double CPU speed, the service time distribution will also remain the same in reference B.
  - The mean response time will also remain the same in the reference B.
  - Now, in the real world, mean response time will actually be half of that seen in reference B
- Thus, the correct solution is that we need to increase the CPU speed, but less than double!
- Suppose the CPU employs time-sharing service order (known as Processor-Sharing, or PS for short), instead of FCFS. Does the answer change?
- No, the basic argument remains the same!

DP2: ONE FAST OR MANY SLOW?



- You are given a choice between one fast CPU of speed *s*, or *n* slow CPUs each of speed *s/n.* Your goal is to minimize mean response time. Assume that jobs are *non-preemptible* (i.e., each job must be run to completion).
- **Question:** Which is the better choice: one fast machine or many slow ones?
- **Hint:** Suppose that I tell you that the answer is, "It depends on the workload." What aspects of the workload do you think the answer depends on?
- **Answer:** It turns out that the answer depends on the variability of the job size distribution, as well as on the system load.

ONE FAST OR MANY SLOW?

- **Question:** Which system do you prefer when job size variability is high?
- **Answer:** When job size variability is high, we prefer many slow servers because we do not want short jobs getting stuck behind long ones.
- **Question:** Which system do you prefer when load is low?
- **Answer:** When load is low, not all servers will be utilized, so it seems better to go with one fast server.
- **Question:** Now suppose we ask the same question, but jobs are *preemptible*; that is, they can be stopped and restarted where they left off. When do we prefer many slow machines as compared to a single fast machine?
- **Answer:** If your jobs are preemptible, you could always use a single fast machine to simulate the effect of n slow machines. Hence a single fast machine is at least as good.

APPLICATIONS OF DP2

- Power Management in data centers: you have a fixed power budget P and a server farm consisting of n servers. You have to decide how much power to allocate to each server, so as to minimize overall mean response time for jobs arriving at the server farm.
  - CPU Frequency is proportional to power allocated with some max frequency and min power level
- Bandwidth allocation: Bandwidth is the resource, we can ask when it pays to partition bandwidth into smaller chunks and when it is better not to.
- Price vs. Performance Trade-off:
  - It is often cheaper (financially) to purchase many slow servers than a few fast servers.
  - Yet in some cases, many slow servers can consume more total power than a few fast ones.
  - All of these factors can further influence the choice of architecture.

DP3

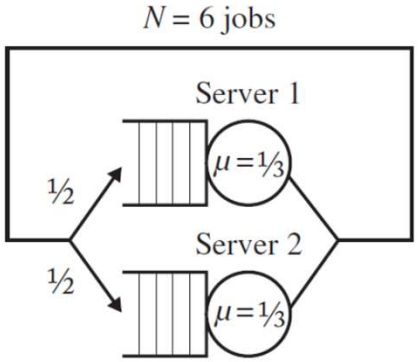


Figure 1.3. A closed batch system.

- There are always $N = 6$ jobs in this system (this is called the multiprogramming level).
- As soon as a job completes service, a new job is started (this is called a "closed" system).
- Each job must go through the "service facility." At the service facility, with probability 1/2 the job goes to server 1, and with probability 1/2 it goes to server 2.
- Server 1,2: Services jobs at an average rate of 1 job every 3 seconds.
- **Que:** You replace server 1 with a server that is twice as fast (the new server services jobs at an average rate of 2 jobs every 3 seconds). Does this "improvement" affect the average response time in the system? Does it affect the throughput?

SOLUTION TO DP3

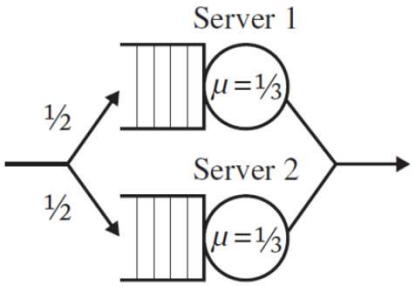


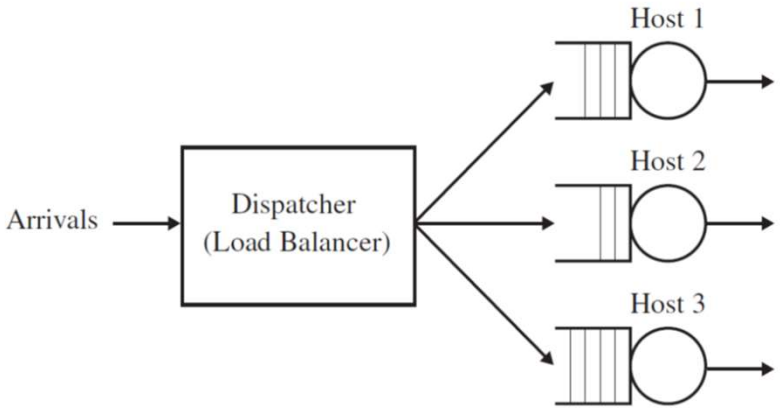
Figure 1.4. An open system.

- Both the average response time and throughput are hardly affected.
- **Question:** Suppose that the system had a higher multiprogramming level, *N*. Does the answer change?
- **Answer:** No. The already negligible effect on response time and throughput goes to zero as *N* increases.
- **Question:** Suppose the system had a lower value of *N*. Does the answer change?
- **Answer:** Yes. If *N* is sufficiently low, then the "improvement" helps. Consider, for example, the case $N = 1$.
- **Question:** Suppose the system is changed into an open system, rather than a closed system, as shown in Figure 1.4, where arrival times are independent of service completions. Now does the "improvement" reduce mean response time?
- **Answer:** Absolutely!

APPLICATIONS OF DP3

- Power Management in data centers: you have a fixed power budget P and a server farm consisting of n servers. You have to decide how much power to allocate to each server, so as to minimize overall mean response time for jobs arriving at the server farm.
  - CPU Frequency is proportional to power allocated with some max frequency and min power level
- Bandwidth allocation: Bandwidth is the resource, we can ask when it pays to partition bandwidth into smaller chunks and when it is better not to.
- Price vs. Performance Trade-off:
  - It is often cheaper (financially) to purchase many slow servers than a few fast servers.
  - Yet in some cases, many slow servers can consume more total power than a few fast ones.
  - All of these factors can further influence the choice of architecture.

DP4: TASK ASSIGNMENT IN SERVER FARM



- Consider a server farm with a central dispatcher and several hosts.
- Each arriving job is immediately dispatched to one of the hosts for processing.
- Assume that all the hosts are identical (homogeneous) and that all jobs only use a single resource.
- Once jobs are assigned to a host, they are processed in FCFS order and are non-preemptible.
- There are many possible *task assignment policies* that can be used for dispatching jobs to hosts.

POLICIES FOR TASK ASSIGNMENT

- **Random:** Each job flips a fair coin to determine where it is routed.
- **Round-Robin:** The ith job goes to host i mod n, where n is the number of hosts, and hosts are numbered $0, 1, \ldots, n-1$.
- **Shortest-Queue:** Each job goes to the host with the fewest number of jobs.
- **Size-Interval-Task-Assignment (SITA):** "Short" jobs go to the first host, "medium" jobs go to the second host, "long" jobs go to the third host, etc., for some definition of "short," "medium," and "long."
- **Least-Work-Left (LWL):** Each job goes to the host with the least total remaining work, where the "work" at a host is the sum of the sizes of jobs there.
- **Central-Queue:** Rather than have a queue at each host, jobs are pooled at one central queue. When a host is done working on a job, it grabs the first job in the central queue to work on.

DP4

- **Question:** Which of these task assignment policies yields the lowest mean response time?
- **Answer:** Given the ubiquity of server farms, it is surprising how little is known about this.
- If job size **variability is low**, then the **LWL policy** is best.
- If job size **variability is high**, then it is important to keep short jobs from getting stuck behind long ones, so a **SITA-like policy**, which affords short jobs **isolation** from long ones, can be far better.
- It was believed previously that SITA is always better than LWL when job size variability is high
  - It was recently discovered (see [90]) that SITA can be far worse than LWL even under job size variability tending to infinity.
  - It turns out that other properties of the workload, including load and fractional moments of the job size distribution, matter as well.

KNOWING THE SIZE OF JOBS?

- **Question:** For the previous question, how important was it to know the size of jobs? For example, how does LWL, which requires knowing job size, compare with Central- Queue, which does not?
- **Answer:** Actually, most task assignment policies do not require knowing the size of jobs.
- For example, it can be proven by induction that LWL is equivalent to Central-Queue. Even policies like SITA, which by definition are based on knowing the job size, can be well approximated by other policies that do not require knowing the job size; see [82].

PRE-EMPTIBLE JOBS

- **Question:** Now consider a different model, in which jobs are preemptible. Specifically, suppose that the servers are Processor-Sharing (PS) servers, which time-share among all the jobs at the server, rather than serving them in FCFS order. Which task assignment policy is preferable now? Is the answer the same as that for FCFS servers?
- **Answer:** The task assignment policies that are best for FCFS servers are often a disaster under PS servers. For PS servers, the Shortest-Queue policy is near optimal, whereas that policy is pretty bad for FCFS servers if job size variability is high.

OPEN QUESTIONS

- The case of server farms with PS servers, for example, has received almost no attention, and even the case of FCFS servers is still only partly understood.
- There are also many other task assignment policies that have not been mentioned.
- For example, *cycle stealing* (taking advantage of a free host to process jobs in some other queue) can be combined with many existing task assignment policies to create improved policies.
- There are also other metrics to consider, like minimizing the variance of response time, rather than
- mean response time, or maximizing fairness.
- Finally, task assignment can become even more complex, and more important, when the workload changes over time.

DP5: SCHEDULING POLICIES

- Suppose you have a *single* server. Jobs arrive according to a Poisson process. Assume anything you like about the distribution of job sizes. The following are some possible service orders (scheduling orders) for serving jobs:
- **First-Come-First-Served (FCFS):** When the server completes a job, it starts working on the job that arrived earliest.
- **Non-Preemptive Last-Come-First-Served (LCFS):** When the server completes a job, it starts working on the job that arrived last.
- **Random:** When the server completes a job, it starts working on a random job.
- **Question:** Which of these non-preemptive service orders will result in the lowest mean
- response time?
- **Answer:** Believe it or not, they all have the same mean response time.

DP5:

- **Question:** Suppose we change the non-preemptive LCFS policy to a Preemptive-LCFS policy (PLCFS), which works as follows: Whenever a new arrival enters the system, it immediately preempts the job in service. How does the mean response time of this policy compare with the others?
- **Answer:** It depends on the variability of the job size distribution. If the job size distribution is at least moderately variable, then PLCFS will be a huge improvement. If the job size distribution is hardly variable (basically constant), then PLCFS policy will be up to a factor of 2 worse.

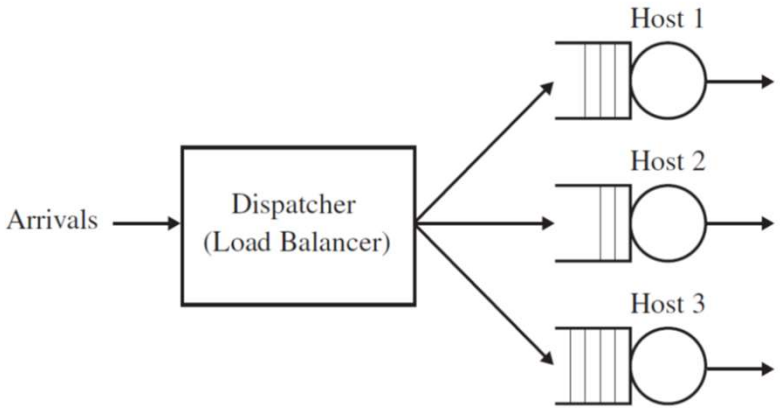
SET-UP COST

- It turns out that it can take both significant time and power to turn *on* a server that is *off*. In designing an efficient power management policy, we often want to leave servers *off* (to save power), but then we have to pay the setup cost to get them back on when jobs arrive.
- Given performance goals, both with respect to response time and power usage, an important question is whether it pays to turn a server off. If so, one can then ask exactly how many servers should be left on.

PRIORITIES

- There are also questions involving optimal scheduling when jobs have priorities (e.g., certain users have paid more for their jobs to have priority over other users' jobs, or some jobs are inherently more vital than others). Again, queueing theory is very useful in designing the right priority scheme to maximize the value of the work completed.

DP: TASK ASSIGNMENT IN SERVER FARM



- Consider a server farm with a central dispatcher and several hosts.
- Each arriving job is immediately dispatched to one of the hosts for processing.
- Assume that all the hosts are identical (homogeneous) and that all jobs only use a single resource.
- Once jobs are assigned to a host, they are processed in FCFS order and are non-preemptible.
- There are many possible *task assignment policies* that can be used for dispatching jobs to hosts.

POLICIES FOR TASK ASSIGNMENT

- **Random:** Each job flips a fair coin to determine where it is routed.
- **Round-Robin:** The ith job goes to host i mod n, where n is the number of hosts, and hosts are numbered $0, 1, \ldots, n-1$.
- **Shortest-Queue:** Each job goes to the host with the fewest number of jobs.
- **Size-Interval-Task-Assignment (SITA):** "Short" jobs go to the first host, "medium" jobs go to the second host, "long" jobs go to the third host, etc., for some definition of "short," "medium," and "long."
- **Least-Work-Left (LWL):** Each job goes to the host with the least total remaining work, where the "work" at a host is the sum of the sizes of jobs there.
- **Central-Queue:** Rather than have a queue at each host, jobs are pooled at one central queue. When a host is done working on a job, it grabs the first job in the central queue to work on.

DP

- **Q:** Which of these task assignment policies yields the lowest mean response time?
- **A:** Given the ubiquity of server farms, it is surprising how little is known about this.
- If job size **variability is low**, then the **LWL policy** is best.
- If job size **variability is high**, then it is important to keep short jobs from getting stuck behind long ones, so a **SITA-like policy**, which affords short jobs **isolation** from long ones, can be far better.
- It was believed previously that SITA is always better than LWL when job size variability is high
  - It was recently discovered that SITA can be far worse than LWL even under job size variability tending to infinity.
  - It turns out that other properties of the workload, including load and fractional moments of the job size distribution, matter as well.

https://www.cs.cmu.edu/~harchol/Papers/Sigmetrics09sita.pdf

KNOWING THE SIZE OF JOBS?

- **Question:** For the previous question, how important was it to know the size of jobs? For example, how does LWL, which requires knowing job size, compare with Central- Queue, which does not?
- **Answer:** Actually, most task assignment policies do not require knowing the size of jobs.
- For example, it can be proven by induction that LWL is equivalent to Central- Queue. Even policies like SITA, which by definition are based on knowing the job size, can be well approximated by other policies that do not require knowing the job size

http://www.cs.cmu.edu/~harchol/Papers/tags.pdf

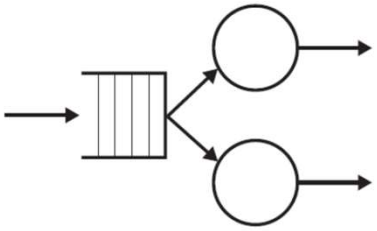
PRE-EMPTIBLE JOBS

- **Question:** Now consider a different model, in which jobs are preemptible. Specifically, suppose that the servers are Processor-Sharing (PS) servers, which time-share among all the jobs at the server, rather than serving them in FCFS order. Which task assignment policy is preferable now? Is the answer the same as that for FCFS servers?
- **Answer:** The task assignment policies that are best for FCFS servers are often a disaster under PS servers.
- For PS servers, the Shortest-Queue policy is near optimal, whereas that policy is pretty bad for FCFS servers if job size variability is high.

OPEN QUESTIONS

- The case of server farms with PS servers, for example, has received almost no attention, and even the case of FCFS servers is still only partly understood.
- There are also many other task assignment policies that have not been mentioned.
- For example, *cycle stealing* (taking advantage of a free host to process jobs in some other queue) can be combined with many existing task assignment policies to create improved policies.
- There are also other metrics to consider, like minimizing the variance of response time, rather than mean response time, or maximizing fairness.
- Finally, task assignment can become even more complex, and more important, when the workload changes over time.

OPEN QUESTIONS ABOUT M/G/2



- There are lots of very simple problems that we can at best only analyze approximately.
- As an example, consider the simple two-server network shown in the above Figure, where job sizes come from a general distribution.
- No one knows how to derive mean response time for this network.
- Approximations exist, but they are quite poor, particularly when job size variability gets high!