Models of Distributed Systems

Message Passing, Queues, Processing and I/O,

Overview

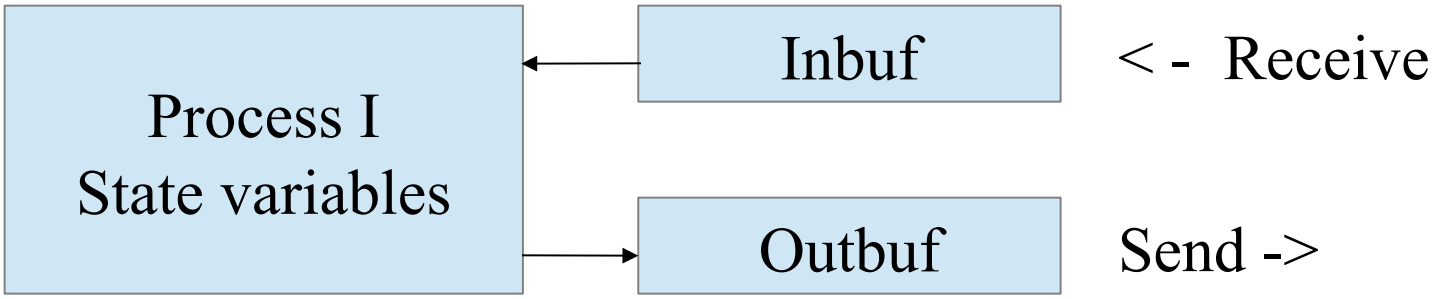
1. Message passing theoretical Model
2. Distributed Computing Topologies
3. Client-Server Systems
   - Critical points, architectures, processing and I/O models

The Message Passing Model

- Modelling and Automata
- Async. vs. Sync Systems
- Protocol Properties: Correctness, Liveness, Fairness,...
- Complexity
- Failure Types

Modeling of Distributed Systems

Ptp primitives:



(After: J.Aspnes): A processing function takes the Inbuf Data from other processes, the internal state variables and computes a new internal state and new Outbuf data. Communication ist point-to-point and deterministic. A configuration is the state vector for all processes. Events change configurations into new ones. An execution is a sequence of configurations and events: C0 e0, C1 e1, C2 e2 ...

Synchronous vs. Asynchronous Systems

Synchronous (lockstep): e== event, t== time
e0t0 ---> delivery at t0+1, e1t1 ---> deliv. t1+1, …..

Asynchronous (delayed):
e0t0 ---> delivery at ?, e1t1 ---> deliv. t1+?, …..
Reqs: infinitely many computing steps possible, events will be **eventually** delivered.

Synchronous systems have simpler distributed algorithms, but are harder to build. The reality is async. Systems with additonal help from failure detectors, randomization etc.

“Eventually”

Does NOT MEAN “perhaps” or “maybe”.

It means “will” happen.

We just don't know WHEN.

http://bravenewgeek.com/from-the-ground-up-reasoning-about-distributed-systems-in-the-real-world/

Message Protocol Properties

- Correctness: invariant properties are shown to hold throughout executions
- Liveness/Termination: the protocol is shown to make progress in the context of certain failures and in a bounded number of rounds
- Fairness: no starvation for anybody
- Agreement: e.g. all processes agree to output the same decision
- Validity: for the same input x, all processes output according to x (Or: there is a possible execution for every possible output value)

(after: Aspnes).

Complexity of Distributed Algorithms

- Time complexity: the time of the last event before all processes finish (Aspens)

- Message complexity: the number of messages sent

Message size and the number of rounds needed for termination are important for the scalability of protocols

Failure Types

- Crash failure: a process stops working and stays down

- Connectivity failures: network failures e.g. causing split brain situations with two separate networks or node isolation. Typically the time for message propagation is affected.

- Message loss: single messages are lost, nodes are up.

- Byzantine Failures: „Evil“ nodes violating protocol assumptions and promises. E.g. breaking a promise due to disk failure, configuration failure etc.

All protocols are validated with respect to certain failure scenarios!!

Distributed Computing Topologies

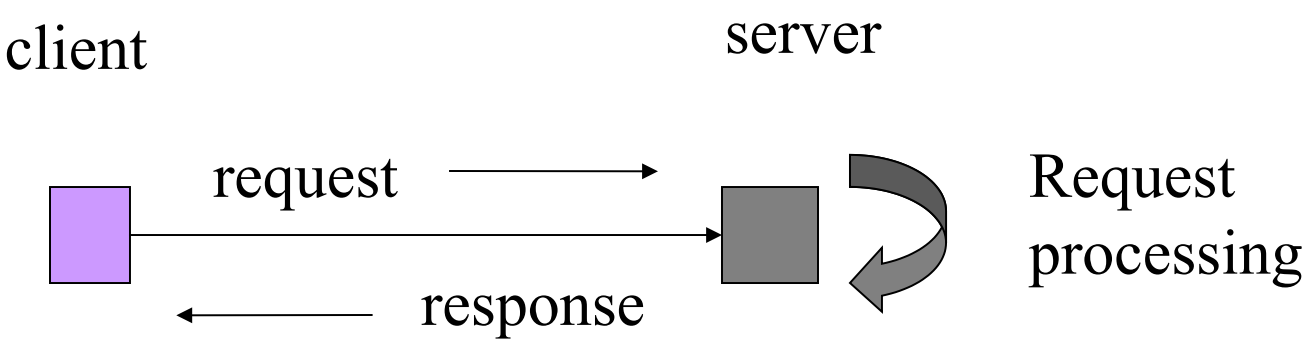
- Client/Server

- Hierarchical

- Totally Distributed

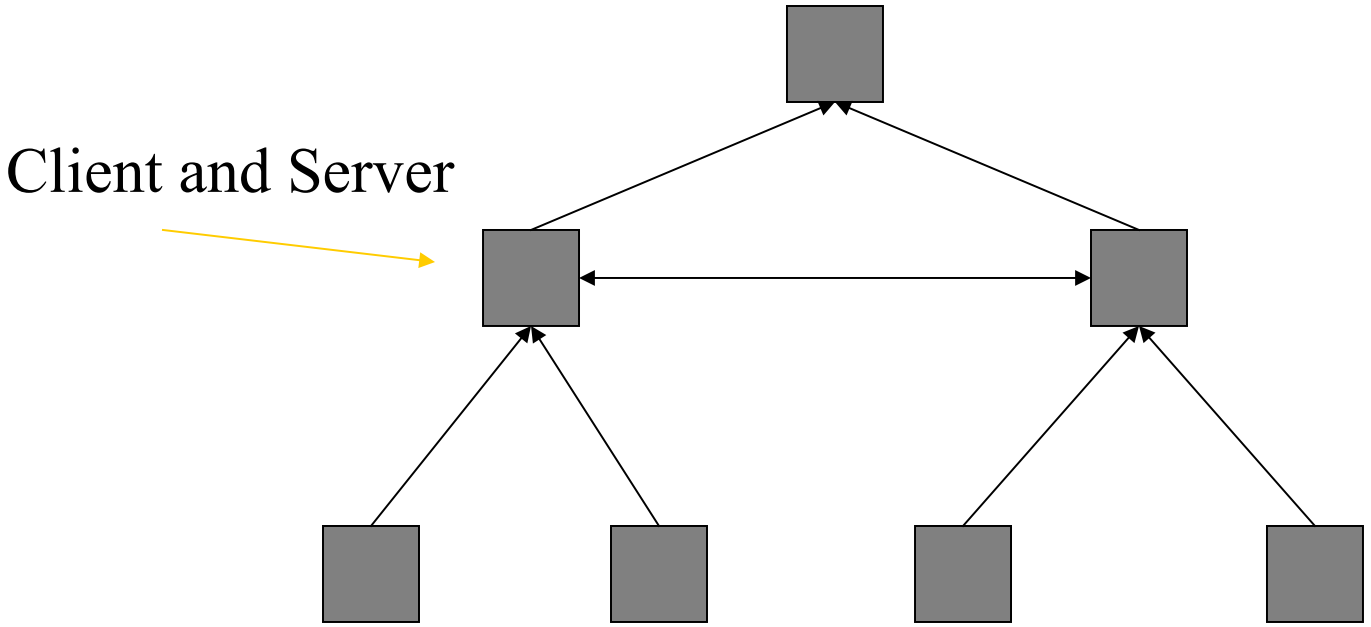
- Bus Topologies

Client/Server Systems



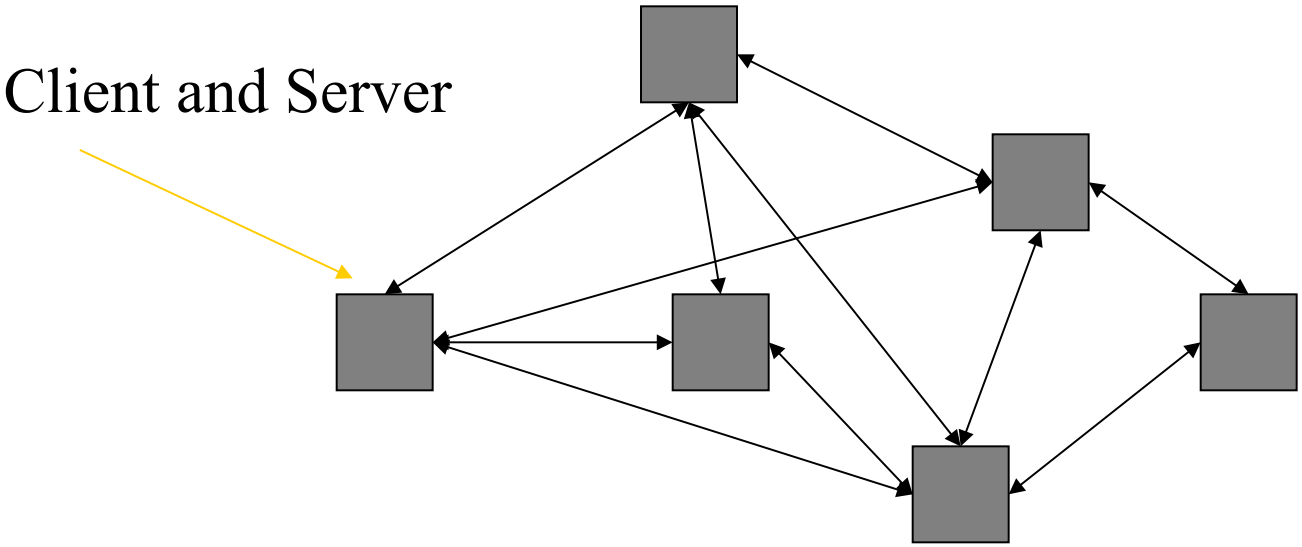
Clients initiate communication and (typically) block while waiting for the server to process the request. Still the most common DS topology.

Hierarchical Systems



Every node can be both client and server but some play a special role, e.g. are Domain Name System (DNS) server. A reduction of communication overhead and central control options are some of the attractive features of this topology.

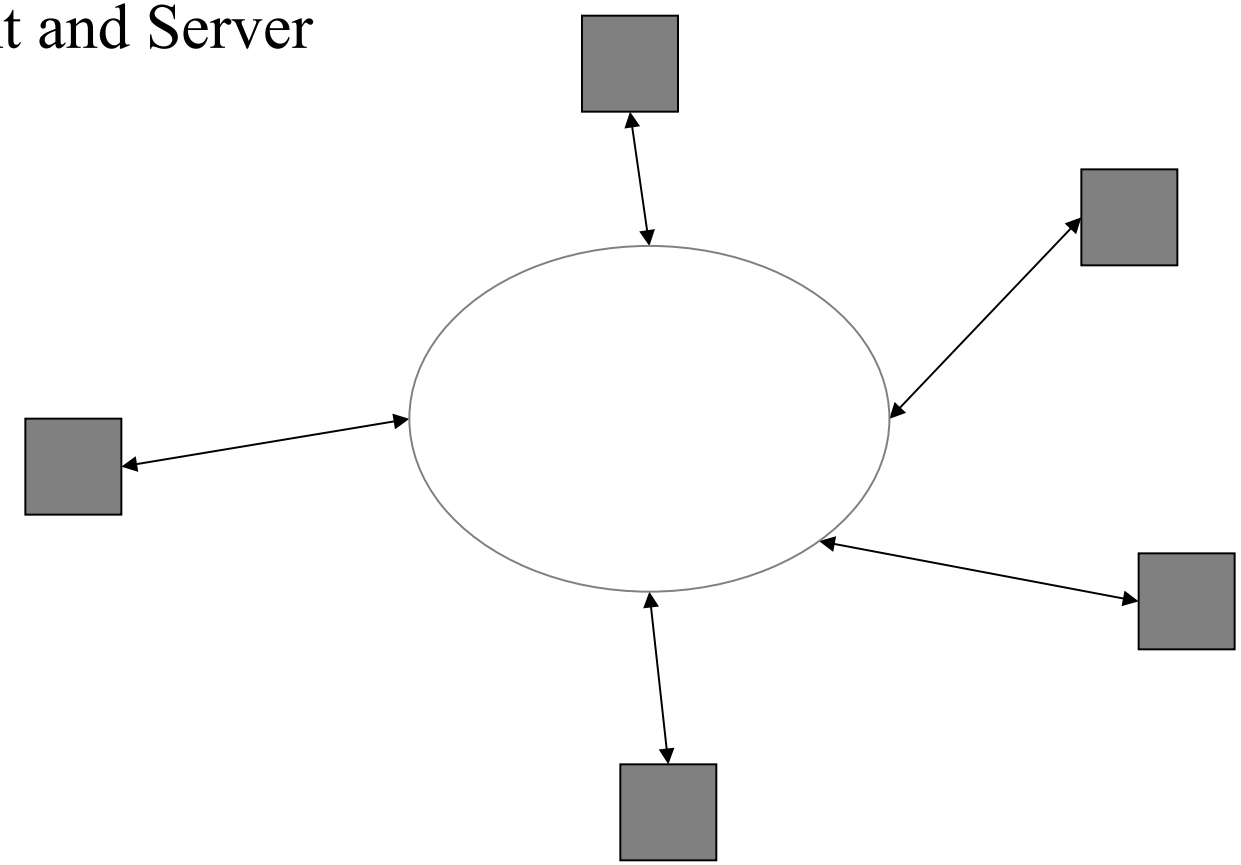
Totally distributed



Every node IS both client and server. Watch out: peer to peer systems need not be totally distributed!

Bus Systems/Pub-Sub

Client and Server



Every node listens for data and posts data in response. This achieves a high degree of separation and indepencence. Event-driven systems follow this topology.

Client-Server Topologies

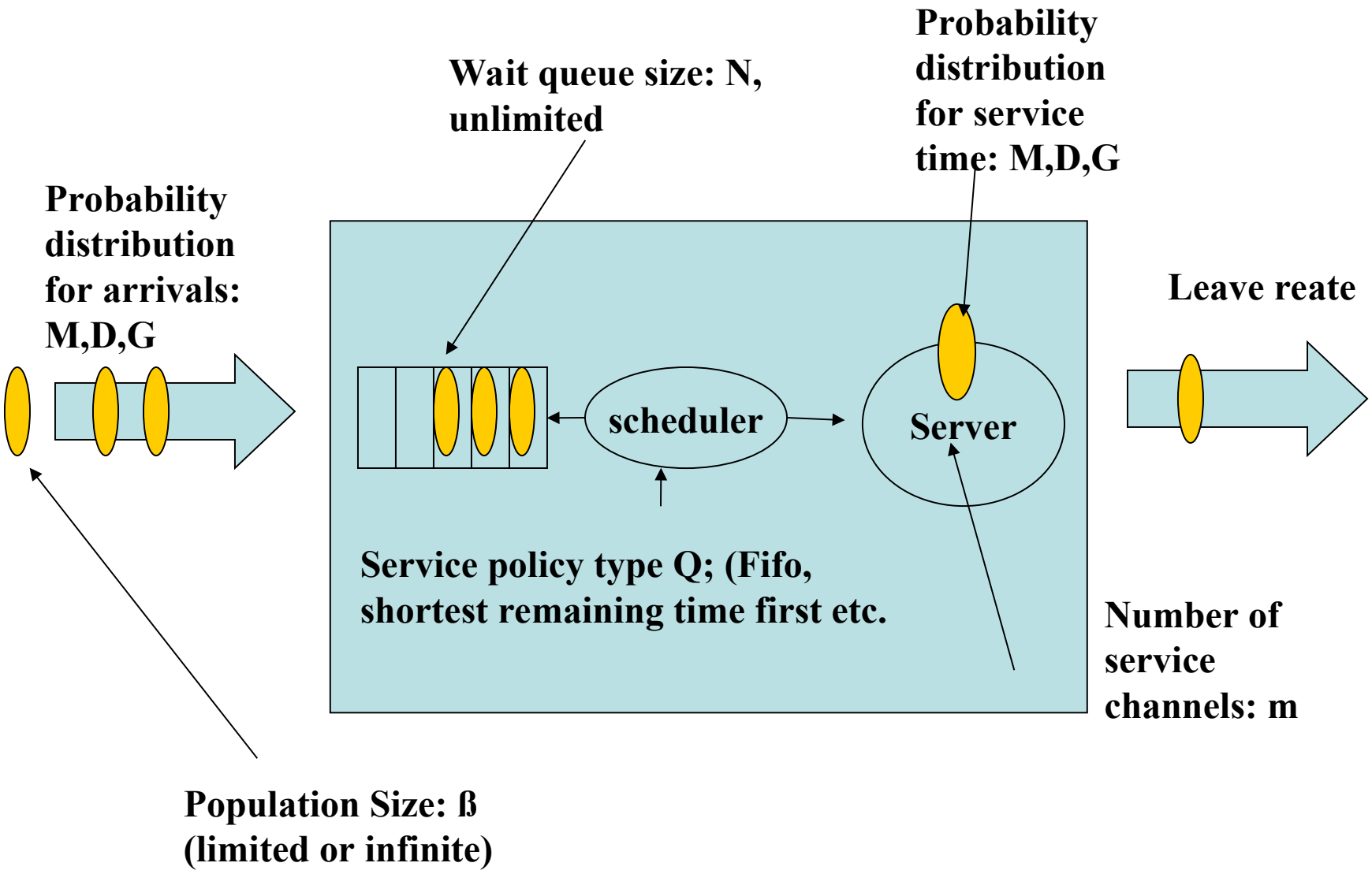
- Theoretical Model
- Terminology
- Critical Points
- Architectures (multi-tier, fan-out, offline)
- Processing Models (cores, threads, processes)
- I/O Models (sync, async, reactive,)

Theoretical Model

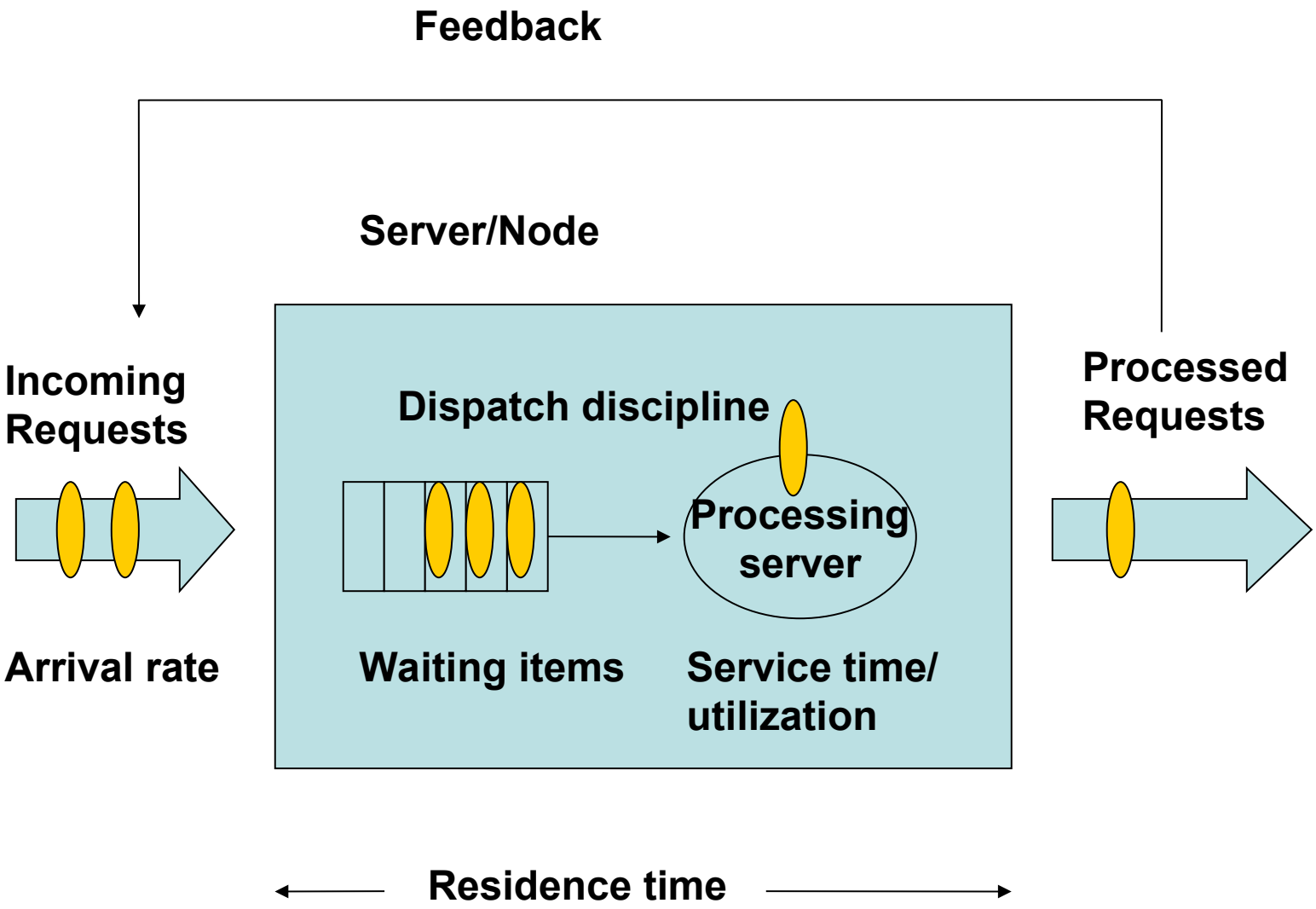
- Queuing Theory
- Little's Law
- Critical Points
- Architectures (multi-tier, fan-out, offline)
- Processing Models (cores, threads, processes)
- I/O Models (sync, async, reactive,)

See: SEDA: An Architecture for Well-Conditioned, Scalable Internet Services, Matt Welsh, David Culler, and Eric Brewer

Queuing Theory: Kendall Notation M/M/m/ß/N/Q



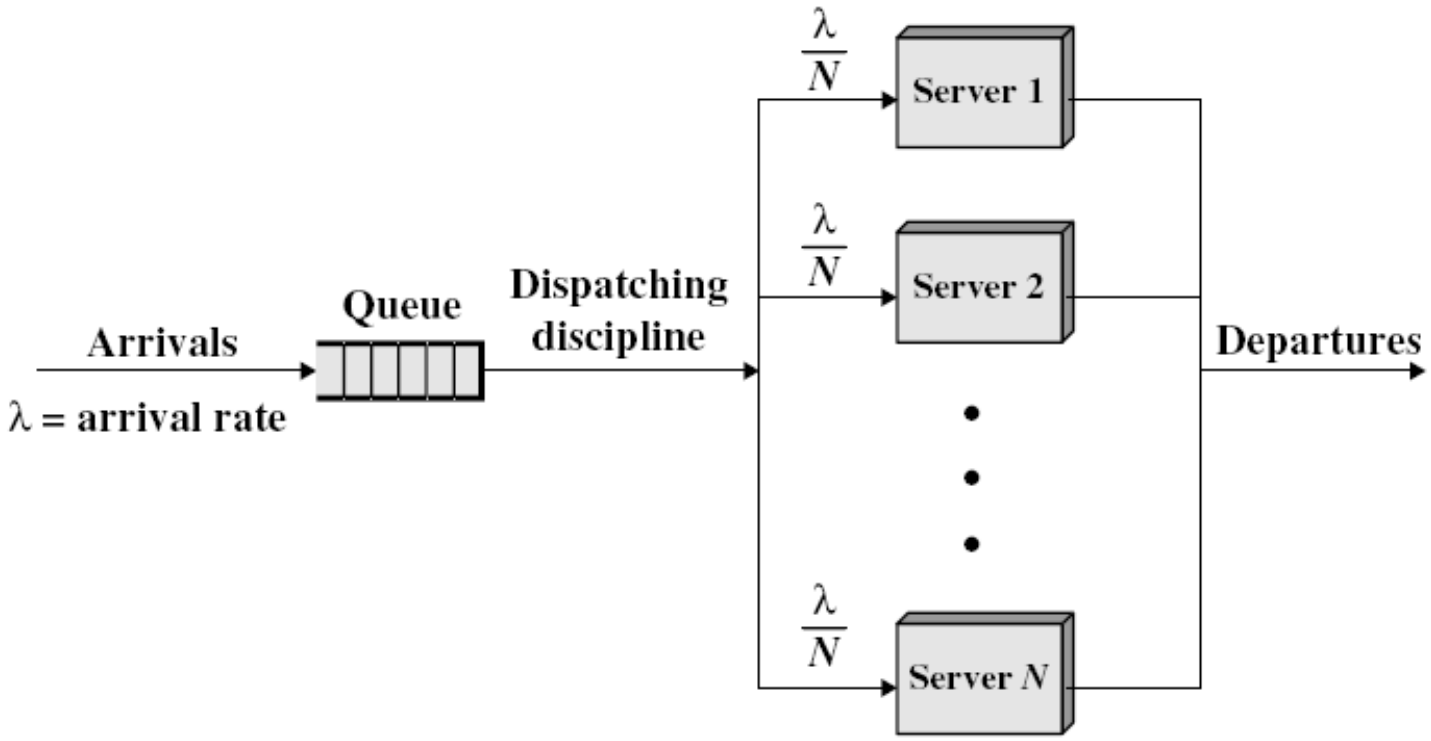
Queuing Theory Terms



• **Server/Node** – combination of wait queue and processing element
• **Initiator** – initiator of service requests
• **Wait time** – time duration a request or initiator has to spend waiting in line
• **Service time** – time duration the processing element has to spend in order to complete the request
• **Arrival rate** – rate at which requests arrive for service
• **Utilization** – portion of a processing element‘s time actually servicing the request rather than idling
• **Queue length** – total number of requests both waiting and being serviced
• **Response time** – the sum of wait time and service time for one visit to the processing element
• **Residence time** – total time if the processing element is visited multiple times for one transaction.
• **Throughput** – rate at which requests are serviced. A server certainly is interested in knowing how fast requests can be serviced without losing them because of long wait time.

Generalized Queuing Theory terms after (Henry Liu)

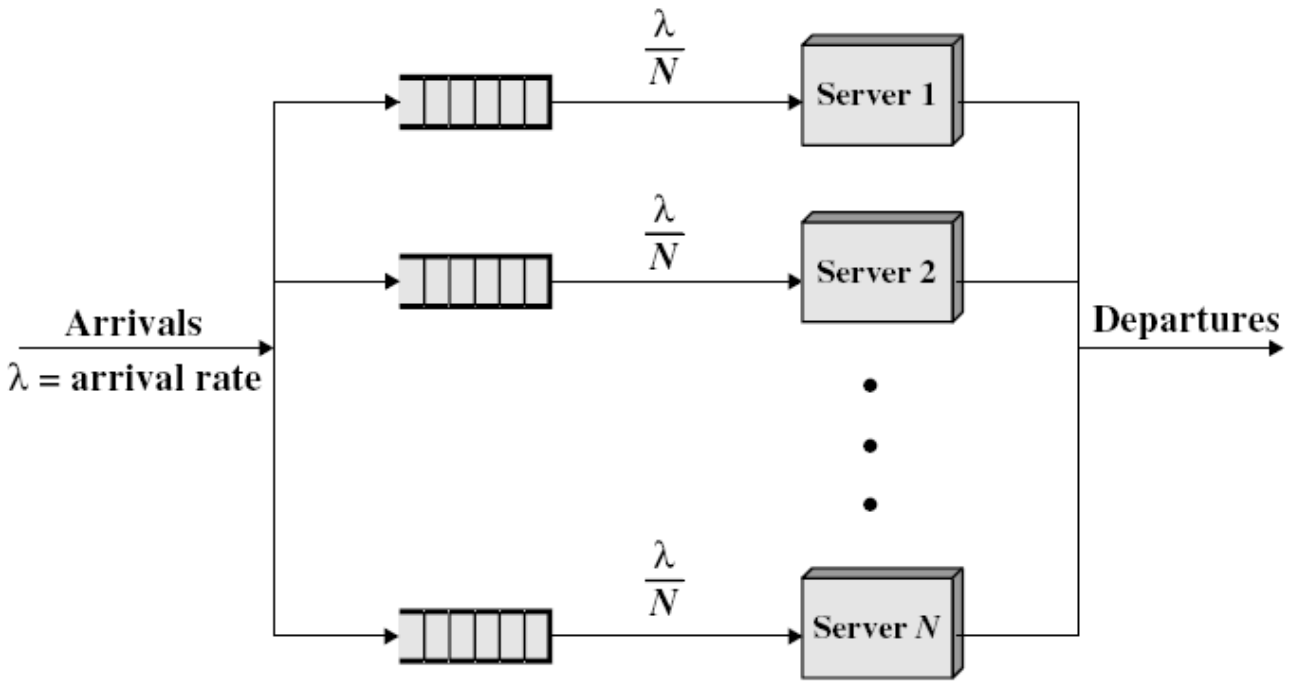
Multiserver Queue



(a) Multiserver queue

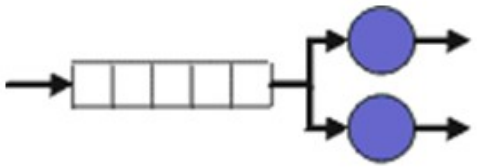
After:
Stallings

Multiple Single-Server Queues



(b) Multiple Single-server queues

After: Stallings



E[N] = 1,8

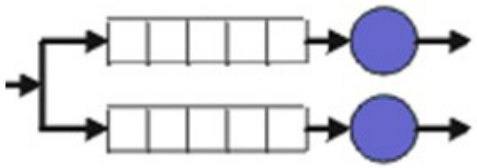
(a) λ=1,8; c=2; μ=1

Mittlere gesamt Ankunftrate = 1.8,

Mittlere einzelne Bedienrate = 1.0,

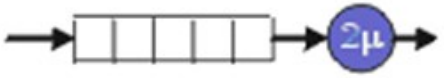
Variationskoeffizient des Ankunftstroms $v_I = 0$

Variationskoeffizient des Bedienprozesses $v_S = 0$



E[N] = 1,8

(b) λ=0,9; c=1; μ=1 -> Erg. ×2



E[N] = 0,9

(c) λ=1,8; c=1; μ=2

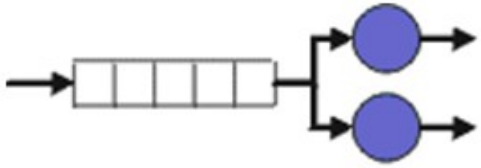
E[N]: Erwartungswert aller Requests im System

C= Prozessoren

M = Bedienrate

From [Hänsch]

$$E[N] \approx \frac{\rho}{(1-\rho)} \cdot \sqrt{\rho^{(c+1)}} \cdot \frac{(v_A^2 + v_S^2)}{2} + \rho c$$



E[N] = 32,5

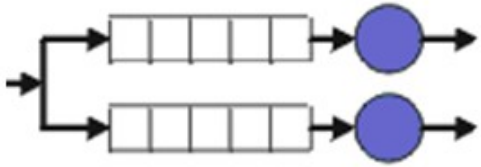
(a) λ=1,8; c=2; μ=1

Mittlere gesamt Ankunftrate = 1.8,

Mittlere einzelne Bedienrate = 1.0,

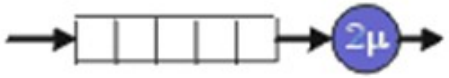
Variationskoeffizient des Ankunftstroms $v_I = 2$

Variationskoeffizient des Bedienprozesses $v_S = 2$



E[N] = 66,6

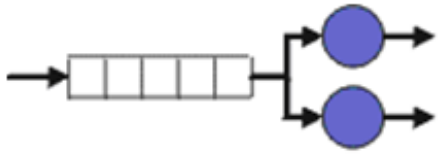
(b) λ=0,9; c=1; μ=1 -> Erg. ×2



E[N] = 33,3

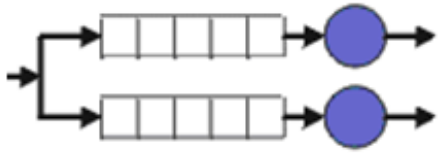
(c) λ=1,8; c=1; μ=2

From [Hänsch]



(a)

Mittlere gesamt Ankunftrate	= 0,9
einzelne Bedienrate high	= 1.5
einzelne Bedienrate low	= 0.5
Variationskoeffizient des Ankunftstroms	$v_I = 1$
Variationskoeffizient des Bedienprozesses	$v_S = 1$

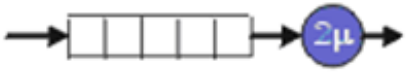


(b)

E[N]Low=9; E[N]High=0,43

λ=0,45; c=1; μLow=0,5 μHigh=1.5

-> Erg. QLow+QHigh



(c)

E[N] = 0,81

λ=0,9; c=1; μ=2

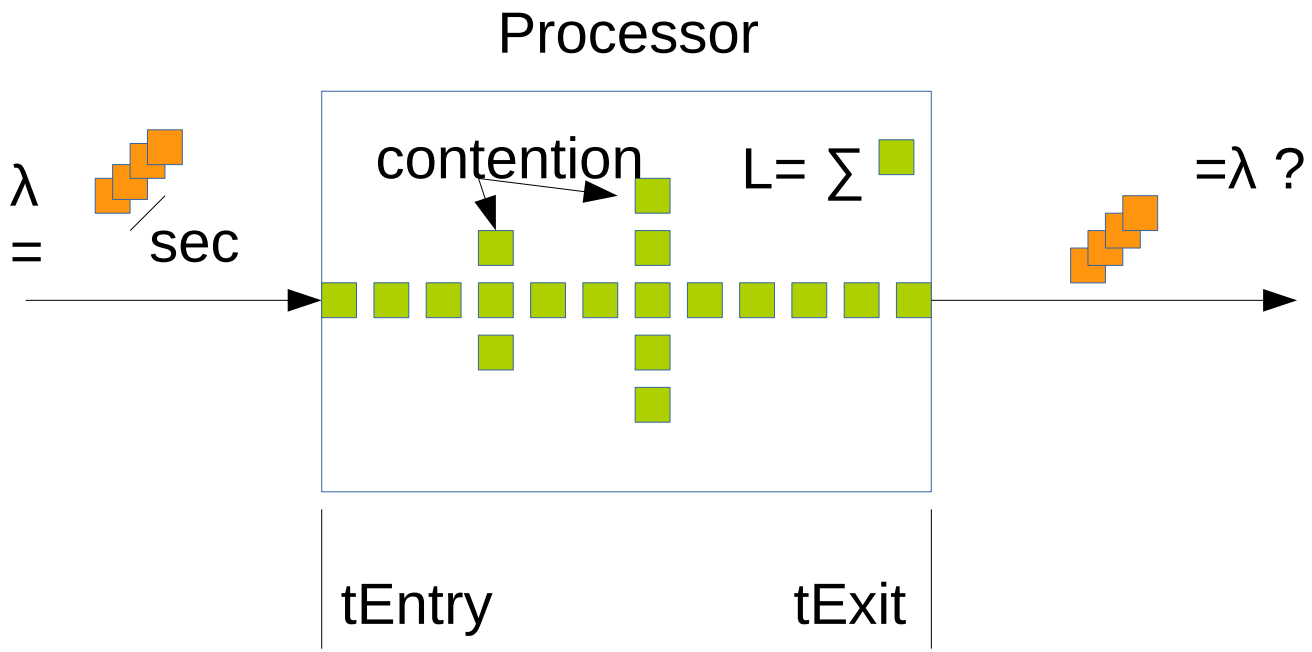
From [Hänsch]

Little's Law

The long-term average number of customers in a stable system L is equal to the long-term average effective arrival rate, λ, multiplied by the (Palm) average time a customer spends in the system, W; or expressed algebraically: $L = \lambda W$.

https://en.wikipedia.org/wiki/Little's_law

L = λ * W



W = tExit - tEntry

Uses of Little's Law

Say that elements in the system equals service units needed. Now we can calculate, whether our service units are over- or under-provisioned or just about right.

$$\text{Maximum throughput} = \frac{\text{App instances}}{\text{Average reponse time}}$$

Twitter (old)
600 requests/second
180 application instances (mongrel)
About 300ms average server response time

600*0,3 = ca.
180 instances

Shopify receives 833 requests/second.
They average a 72ms response time
They run 53 application servers with a total of 1172 application instances (!!!) with Nginx and Unicorn.

833*0.072 = ca.
60 instances

http://www.nateberkopec.com/2015/07/29/scaling-ruby-apps-to-1000-rpm.html. “blocking of application instances (anything that stops all 1172 application instances from operating at the same time) can cause major deviations from Little’s Law. Realize you have three levers - increasing application instances, decreasing response times, and decreasing response time variability. A scalable application that requires fewer instances will have fast response times and low response time variability.”

Estimating Queuing Delay

the Pollaczek-Kinchin

formula [26] tells us that, for an M/G/1 queue:

$$\mathbb{E}[\text{queueing delay}] = \frac{\rho}{1-\rho} \cdot \frac{C^2+1}{2},$$

where ρ is the load and C 2 is the squared coefficient of variation of job sizes.

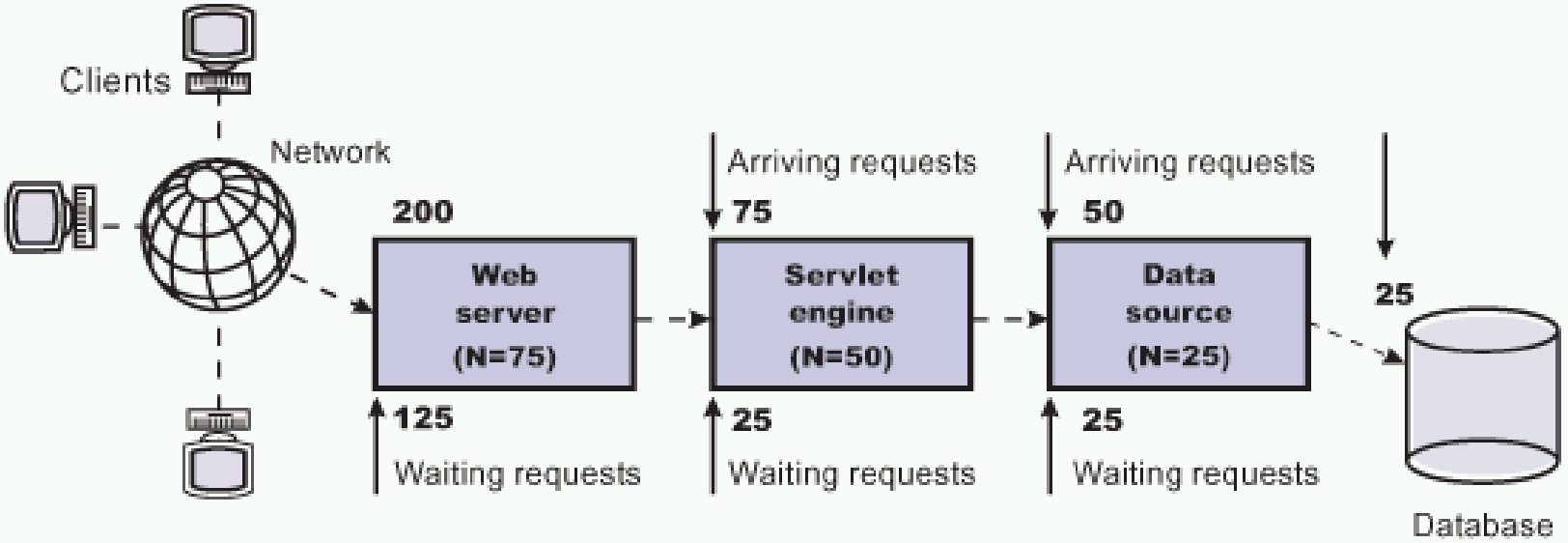
Note: Even with low load the system can experience large delays due to the variation coefficient! Very heterogeneous workloads kill throughput!

Taken from: Borg, the next generation. https://dl.acm.org/doi/pdf/10.1145/3342195.3387517

Lessons Learned from Queuing Theory

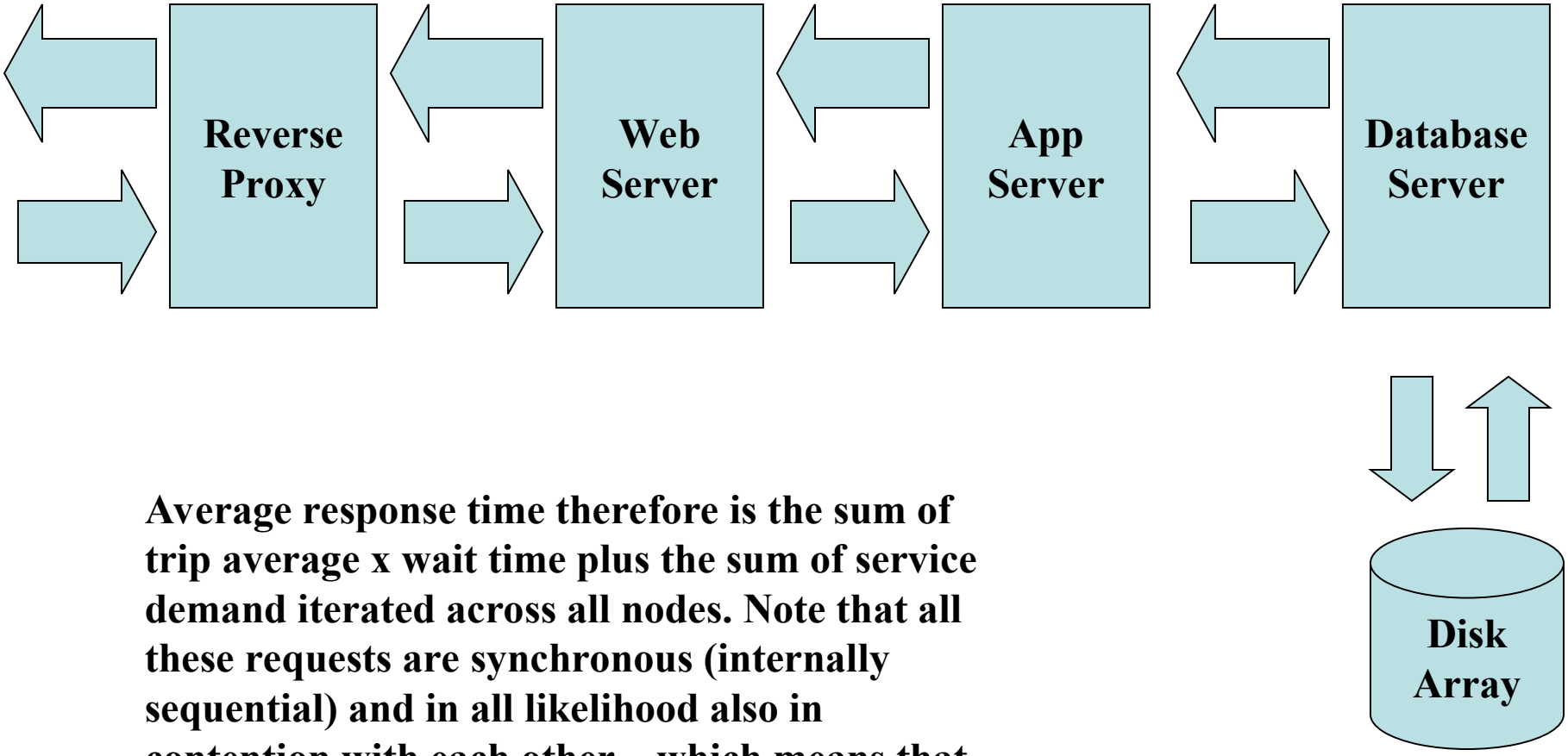
- Request Numbers: Caching
- Batching: Multi-Get API
- Task Sizes and Variability: SLAs, Hejunka

Queuing Theory for Multi-tier Process Networks



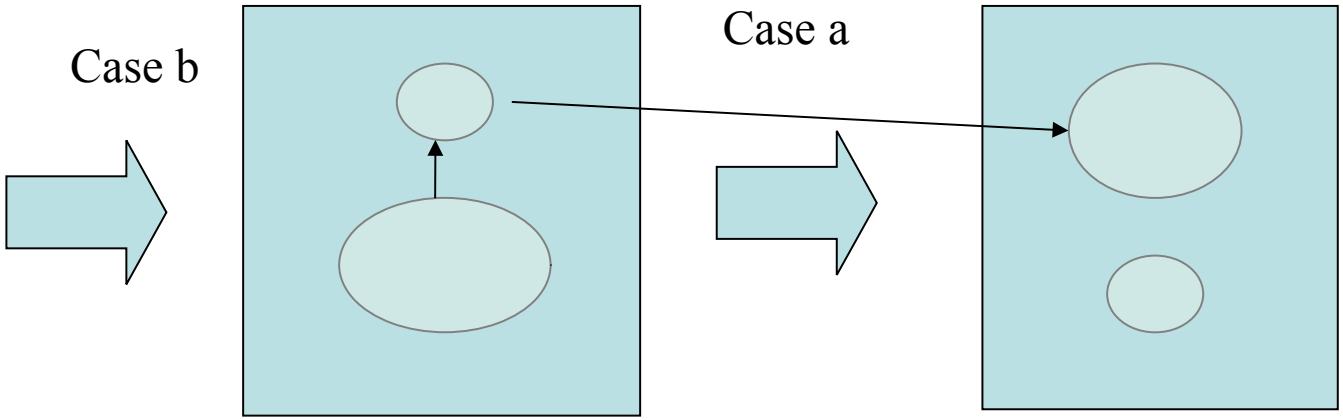
A modern application servers performance largely relies on the proper configuration of several queues from network listening to threadpools etc. Queuing theory lets us determine the proper configurations (see resources). In general, architectures like above are very sensitive for saturated queues. Good architectures create a funnel from left to right and limit resources like max. threads. Caching and batching are directly derived from queuing theory. Picture from: http://publib.boulder.ibm.com/infocenter/wasinfo/v5r1//index.jsp?topic=/com.ibm.websphere.base.doc/info/aes/ae/rprf_queue.html

Request Problem in Multi-Tier Networks



Average response time therefore is the sum of trip average x wait time plus the sum of service demand iterated across all nodes. Note that all these requests are synchronous (internally sequential) and in all likelihood also in contention with each other – which means that wait times occur due to contention

Task Size Problem in Multi-Tier Networks

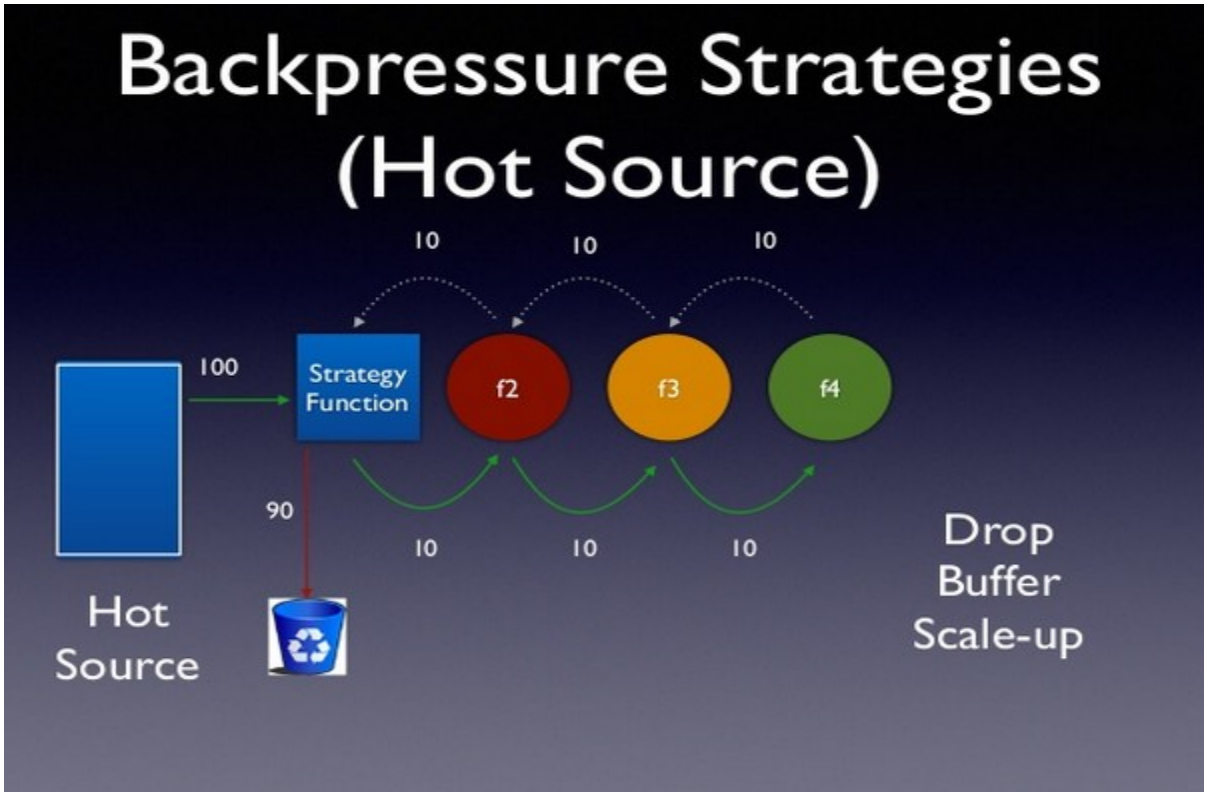


Large differences in task size cause pipeline stalls between nodes (case a) and lead to resource starvation within nodes causing contention and coherence effects (case b)

From Model to Reality

- Latency
- Blocking/locking/Serialization in Service Units
- Non-Random distributions, feedback effect
- Dead Requests
- Backpressure
- Missing Variables, Coherence Losses

Backpressure Strategies



Drop, Buffer or Scale-up, but do NOT crash your infrastructure. Look at the response time delays due to more requests on time-sharing systems. Diagram from: Mantis in Action, Neeraj Joshi, Justin Becker Qcon, 6/12/2015

Critical Points in C/S Systems 1

client

server

Many Clients?

Session state?

Authentication?

authorization?

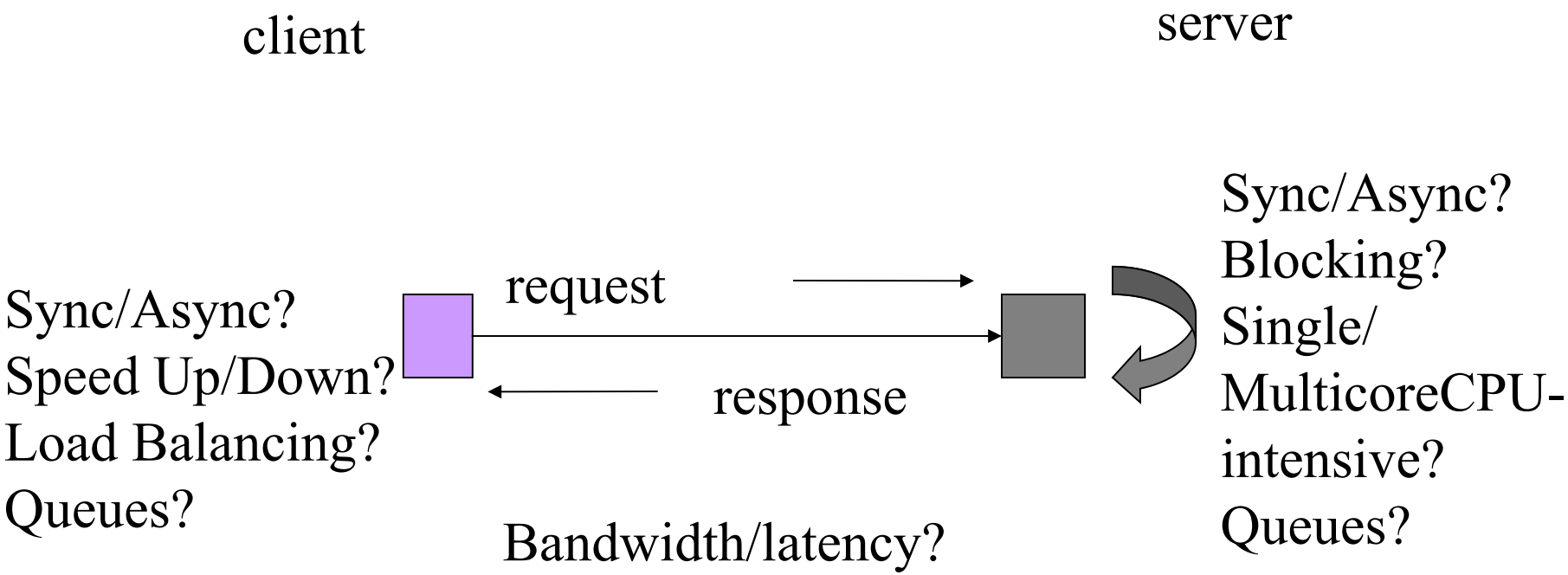
Privacy?

locate server?

Authenticate?

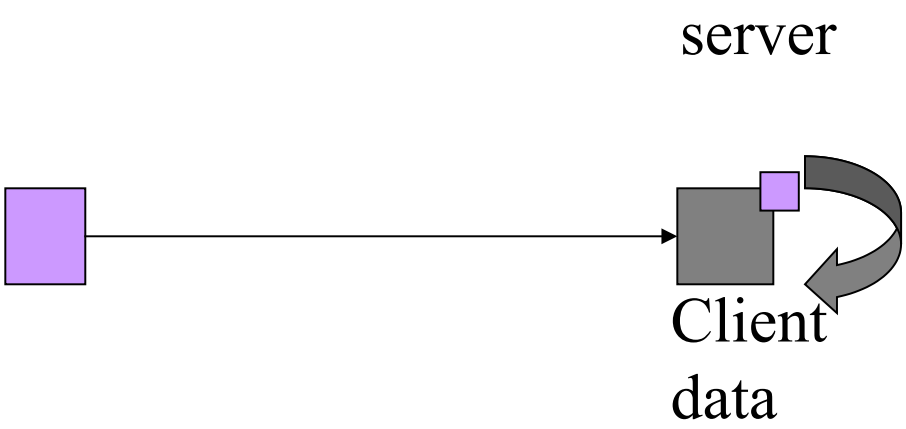
Load management, delays and bottlenecks, failures in backend systems, capacity planning problems, network throughput, security, deployment (global)

Critical Points in C/S Systems 2



Wrong decisions here can make the difference between 10 req/sec and 80.000 req/sec! Think about the upload of a large image. Is it going to hurt your architecture?

Stateful Server Problem



+ data locality

+ consistency

- availability

- load balancing

Stateless design puts all data in DB's, caches etc. In case of failures, this makes programming hard. Stateful services bring the function to the data – at a price...

Terminology 1

Host: A physical machine with n CPU's

Server: A process running on a host, receiving messages , performing computations and sending messages (not necessarily responses)

Thread: Independent computation context within a process, pre-empted by kernel (kernel-thread) or yielding voluntarily (application level scheduling)

Multi-Threading: Several threads running within a process context. Either executed by one kernel-thread switching between threads, or by several kernel threads running in parallel (multi-core). Always non-deterministic.

Multi-Channel: A thread is able to watch several channels with one system call. This is typically done by some variant of a select() call and needs good OS support.

Synchronous processing: A caller calls some function and waits for its results, doing nothing while waiting.

Terminology 2

Asynchronous processing: A caller calls a function and immediately continues executing its own code. The called function gets executed eventually and a callback function is called to inform the caller about the completion. Nothing is said about who executes the called function.

Parallel processing: Deterministic execution of independent code paths.

Blocking: A thread calls a function that needs time to e.g. get a resource from disk. The thread can't continue and would block an execution core waiting for the result. The thread gets “context switched” and a new code path is loaded and executed by the core. Context switching is expensive.

Non-blocking calls: A caller calls the non-blocking version of a function. If the function can perform immediately without delaying the caller, it will do so. If the function would need time to perform its job, it will let the caller return immediately and tell it, that it would be blocked. The caller can then decide to do something else and try later again (poll again).

Synchronization: Needed, when threads share data and need to control the order of access to prevent data inconsistencies

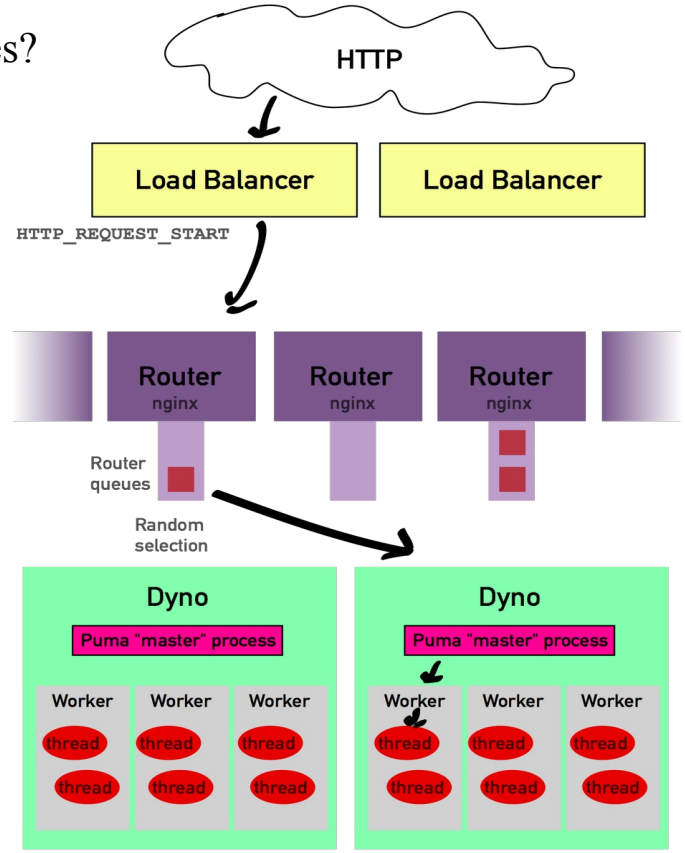
Architectures of C/S Systems

- multi-tier
- fan-out,
- pipeline (SEDA)
- offline

Multi-Tier System

Request Distribution?

Latency and Response Times?



Request Routing?

Queue Sizes and Behavior?

I/O Model?

Process Model?

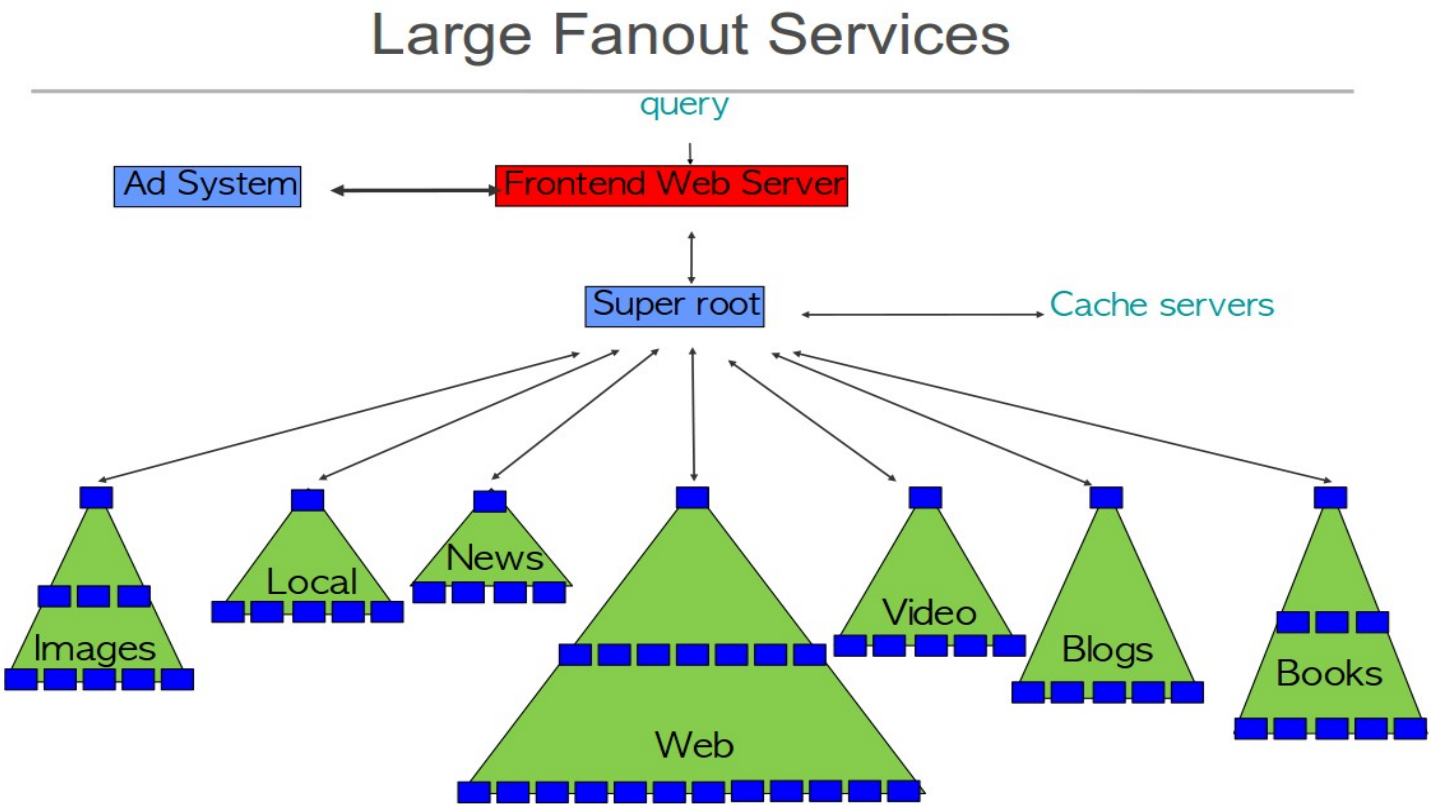
Performance Data?

Availability Model?

Scale Model?

Picture: Nate Berkopec

Large fan-out Architectures at Google



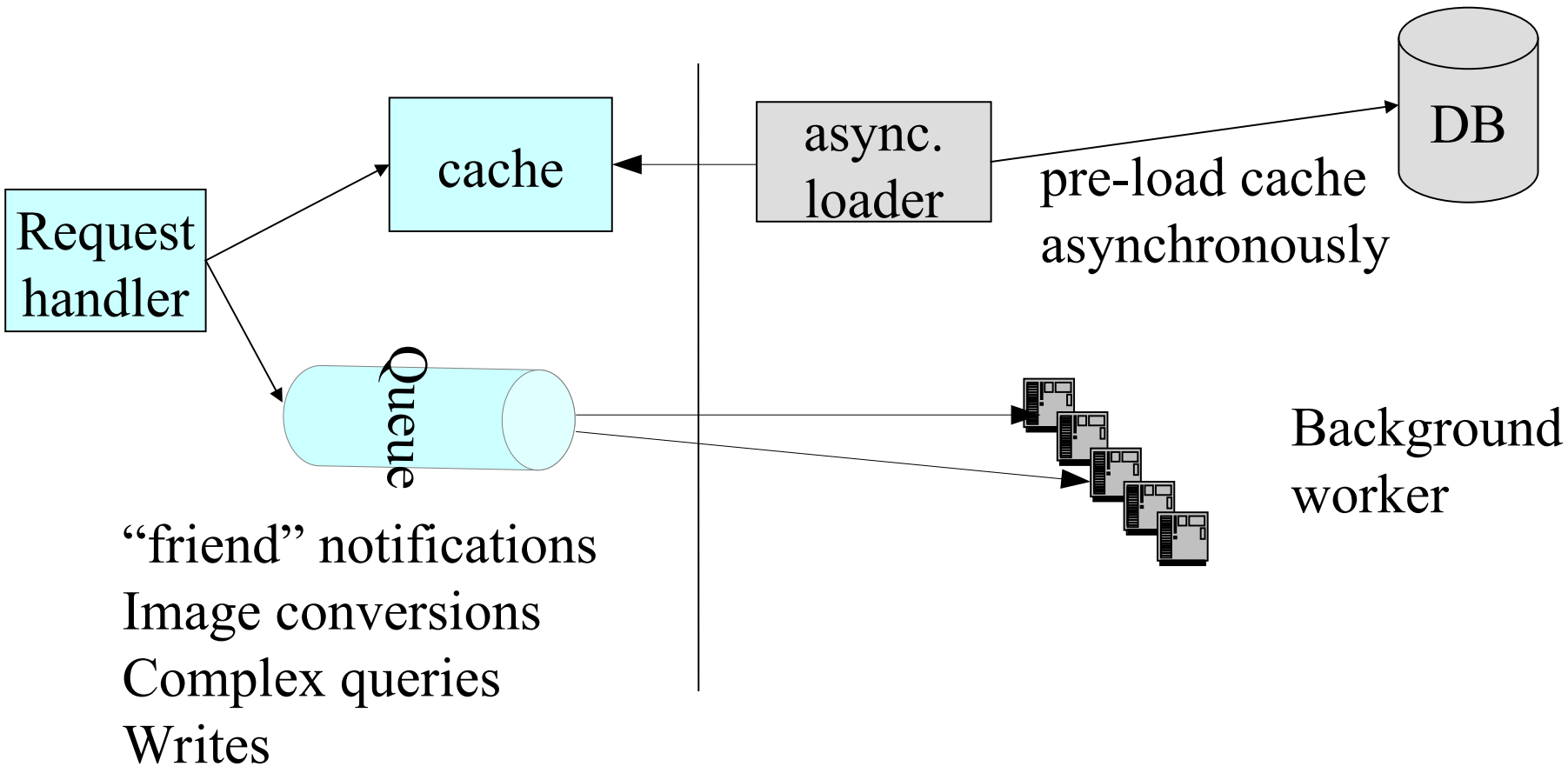
A portal is a typcial “large-fan-out architecture” with long-tail problems. See how google handles this: Talk by Jeff Dean, http://static.googleusercontent.com/media/research.google.com/en//people/jeff/Berkeley-Latency-Mar2012.pdf

The Costs of Delays

100 sub-calls, 1% delayed, how many calls will experience a delay?

“Stalls” of any kind are critical in this architecture. What can we do against hiccups and stalls? Watch out for requests beyond the 99%ile! (Gile Tene, Azul: How NOT to measure latency)

Offline Processing



Do not process things at request time that can be delayed. Pre-calculate and pre-process as much as possible. Fail fast (Netflix..)

Process Models

Single Thread / Single Core

Multi-Thread / Single Core

Multi-Thread /Multi-Core

Single Thread/ Multi-Process

Thread-Level Parallelism

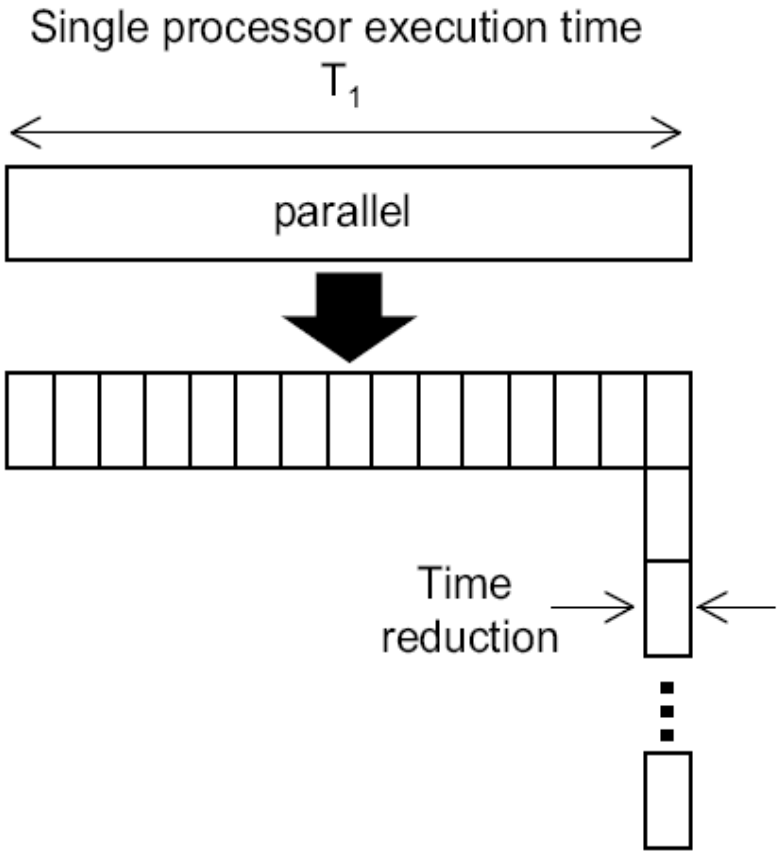


Fig. 4.4. Ideal parallelism. The uniprocessor execution time T_1 is reduced to T_1/p by equipartitioning the workload across p physical processors

From: Gunter, Guerillia Capacity Planning

Serial Fraction limits Speed-up

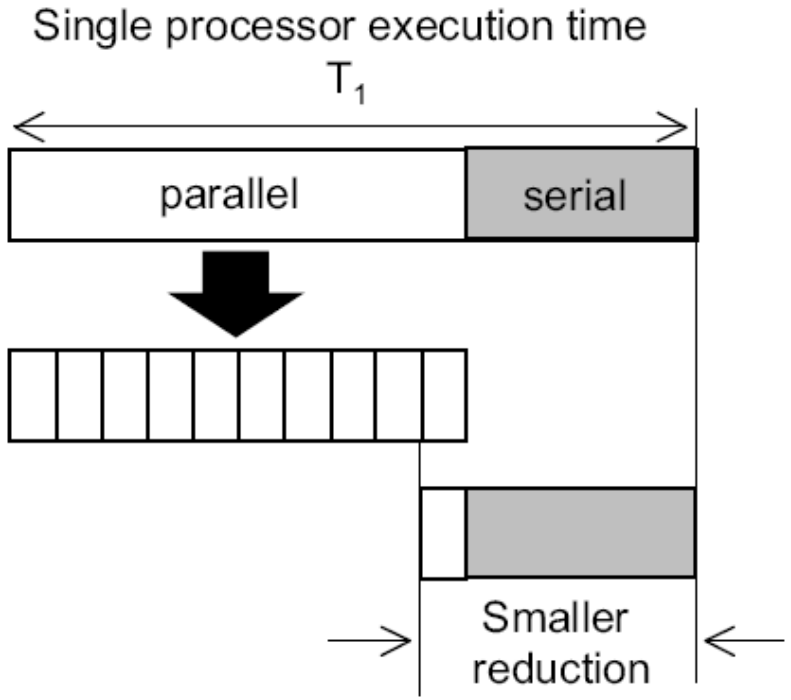


Fig. 4.5. Amdahl's law recognizes that ideal parallelism (Fig. 4.4) cannot be achieved in general because there are certain portions of the workload that can only be executed sequentially (*gray*). That aggregate portion of the total execution time is called the serial fraction

From: Gunter, Guerillia Capacity Planning

Amdahls Law

$$\text{Speedup} = \frac{1}{(1 - \text{Parallel Fraction}) + \frac{\text{Parallel Fraction}}{\text{Number of Processors}}}$$

Very soon adding processors does not increase speedup!

The impact of multiuser scaleup on response time is shown in Fig. 4.7.

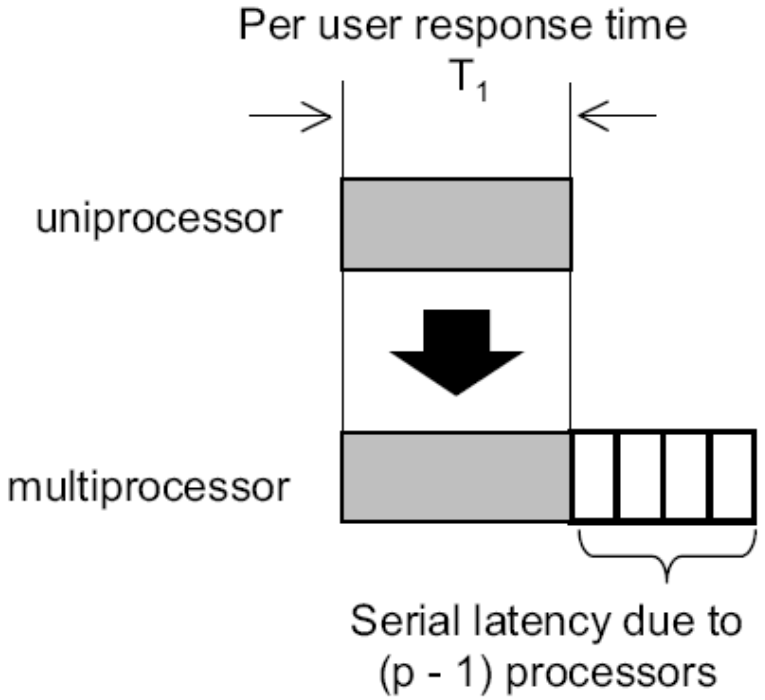


Fig. 4.7. The effect of multiuser scaleup is to stretch the response time in proportion to the number of physical processors

From: Gunter, Guerillia Capacity Planning

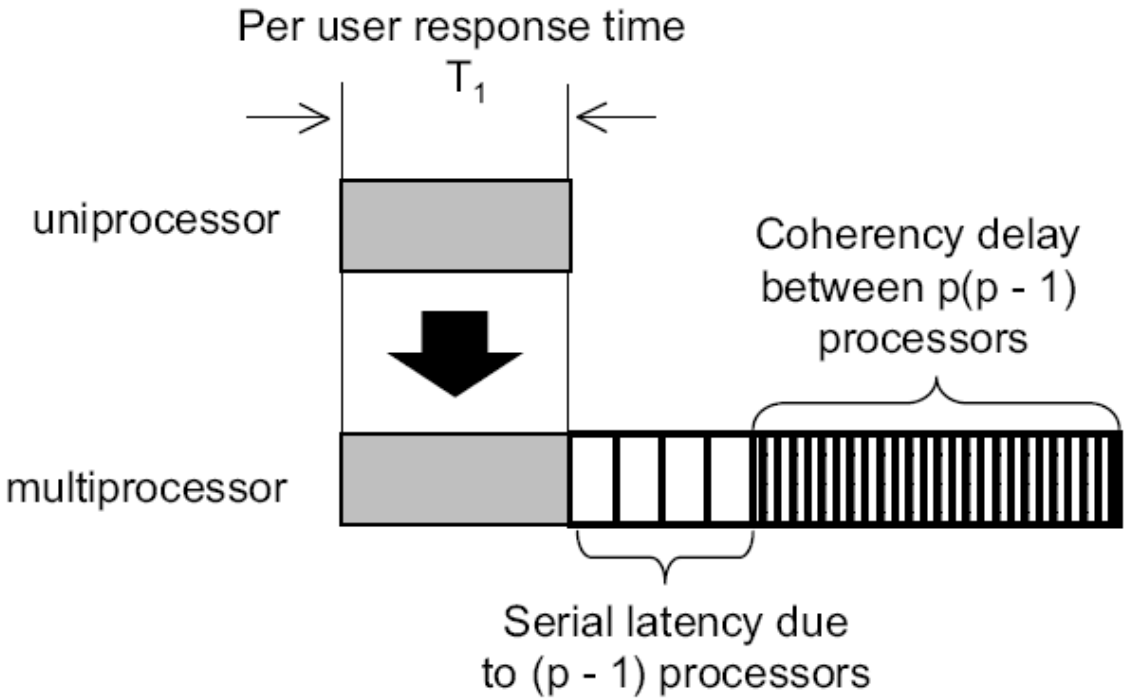


Fig. 4.8. Multiuser scaleup showing the per-user response time growing linearly with the number of processors due to serial delays (cf. Fig. 4.7), and the additional, but smaller, coherency delays increasing quadratically due to point-to-point exchanges between processors

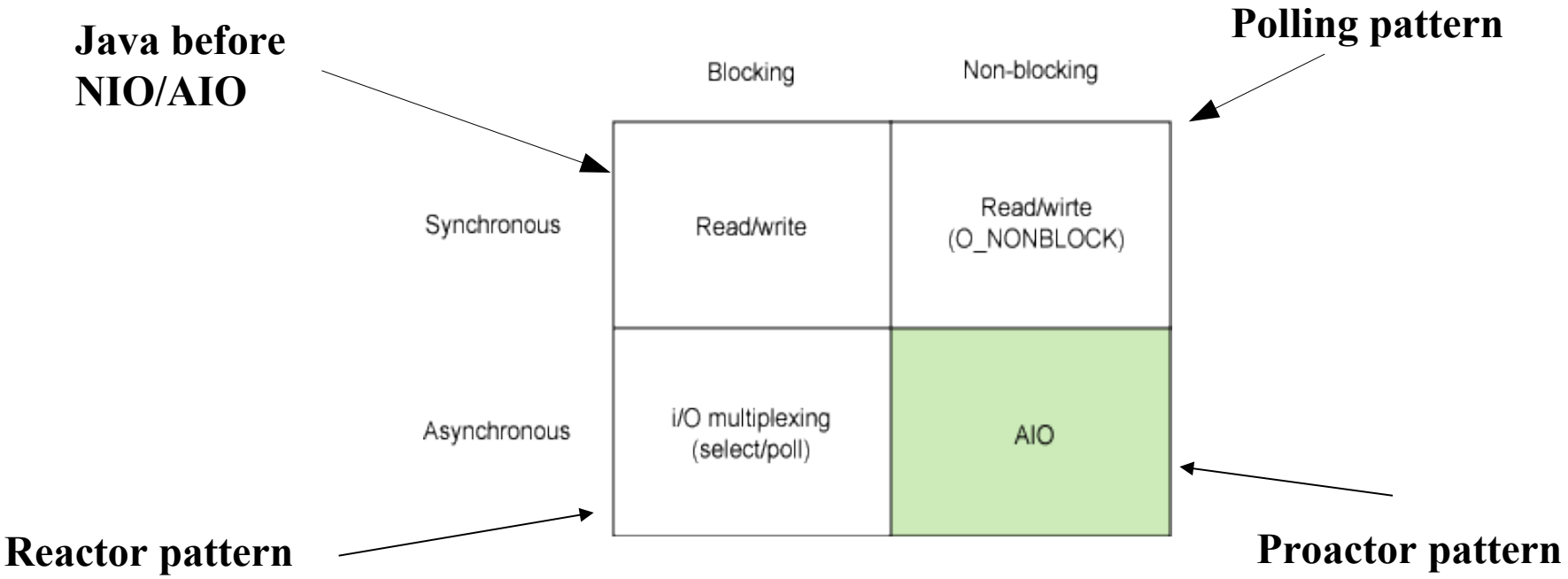
From: Gunter, Guerillia Capacity Planning

Questions for Process Models

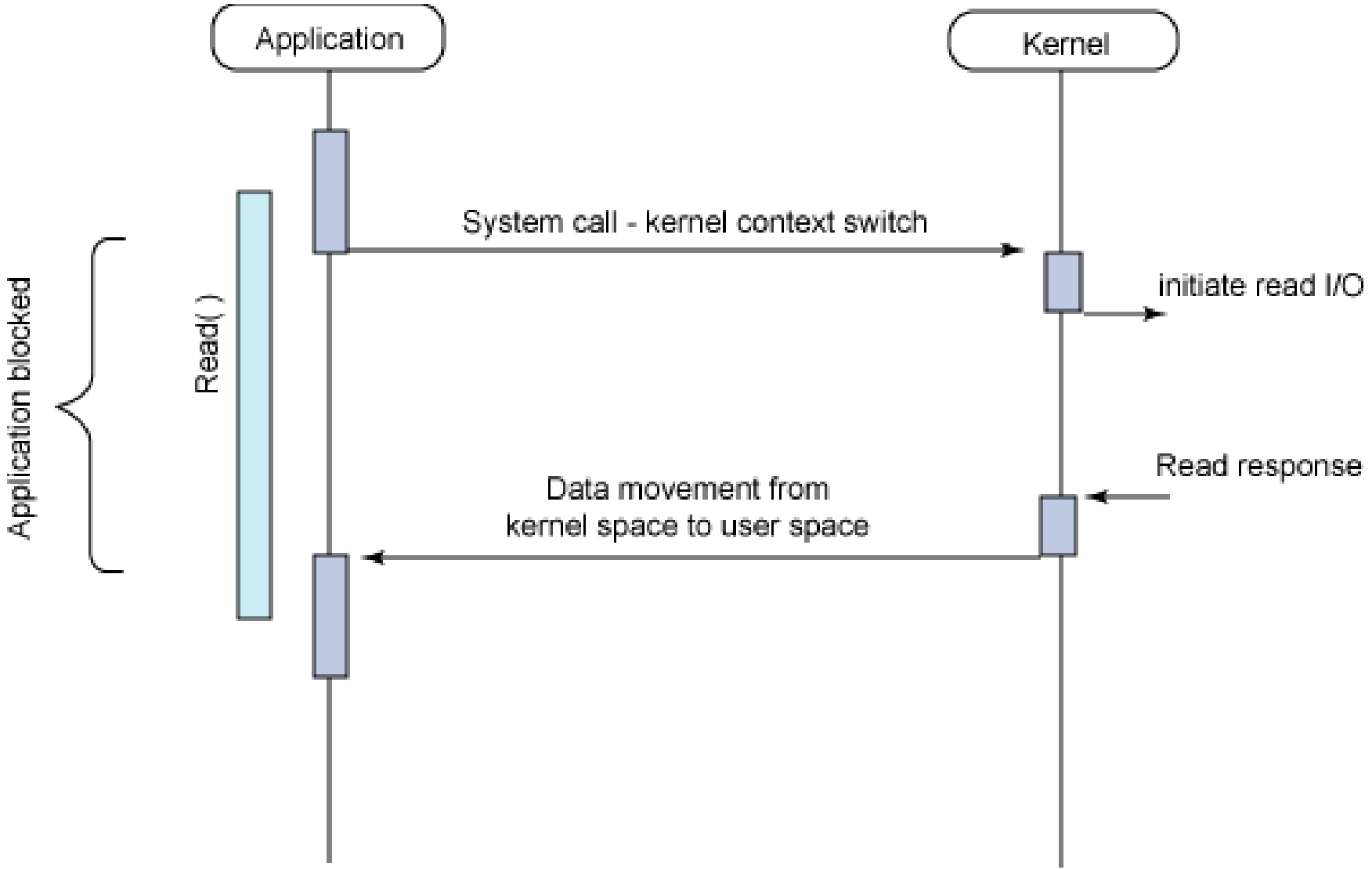
- can it use available cores/CPUs?
- what is the ideal number of threads?
- how does it deal with delays/(b)locking?
- how does it deal with slow requests/uploads?
- Is there observable non-determinism aka race conditions?
- is locking/synchronization needed?
- what is the overhead of context switches and memory?

We are talking request-level parallelism here. Requests won't get faster but we can handle more of them (throughput).

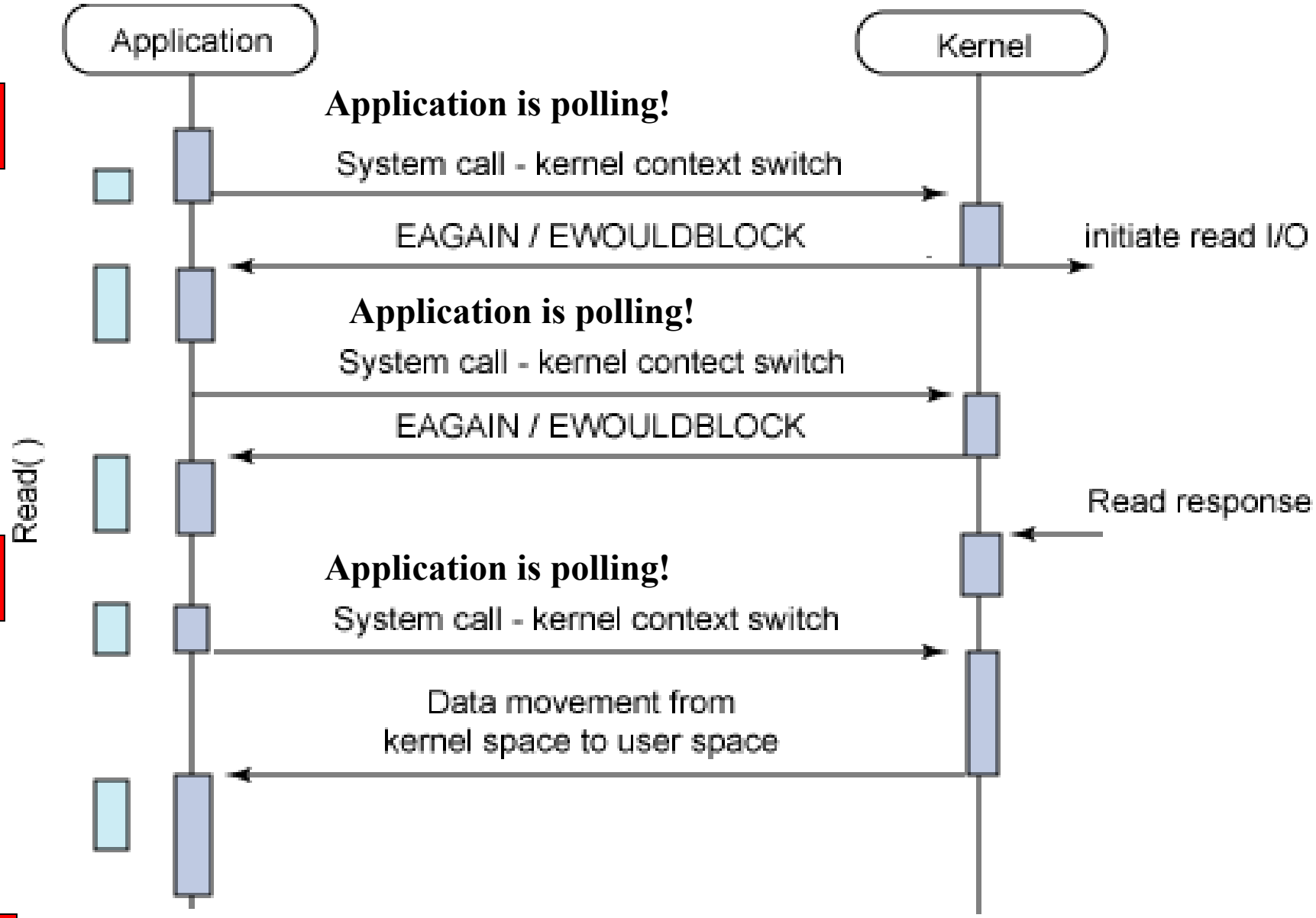
I/O Models



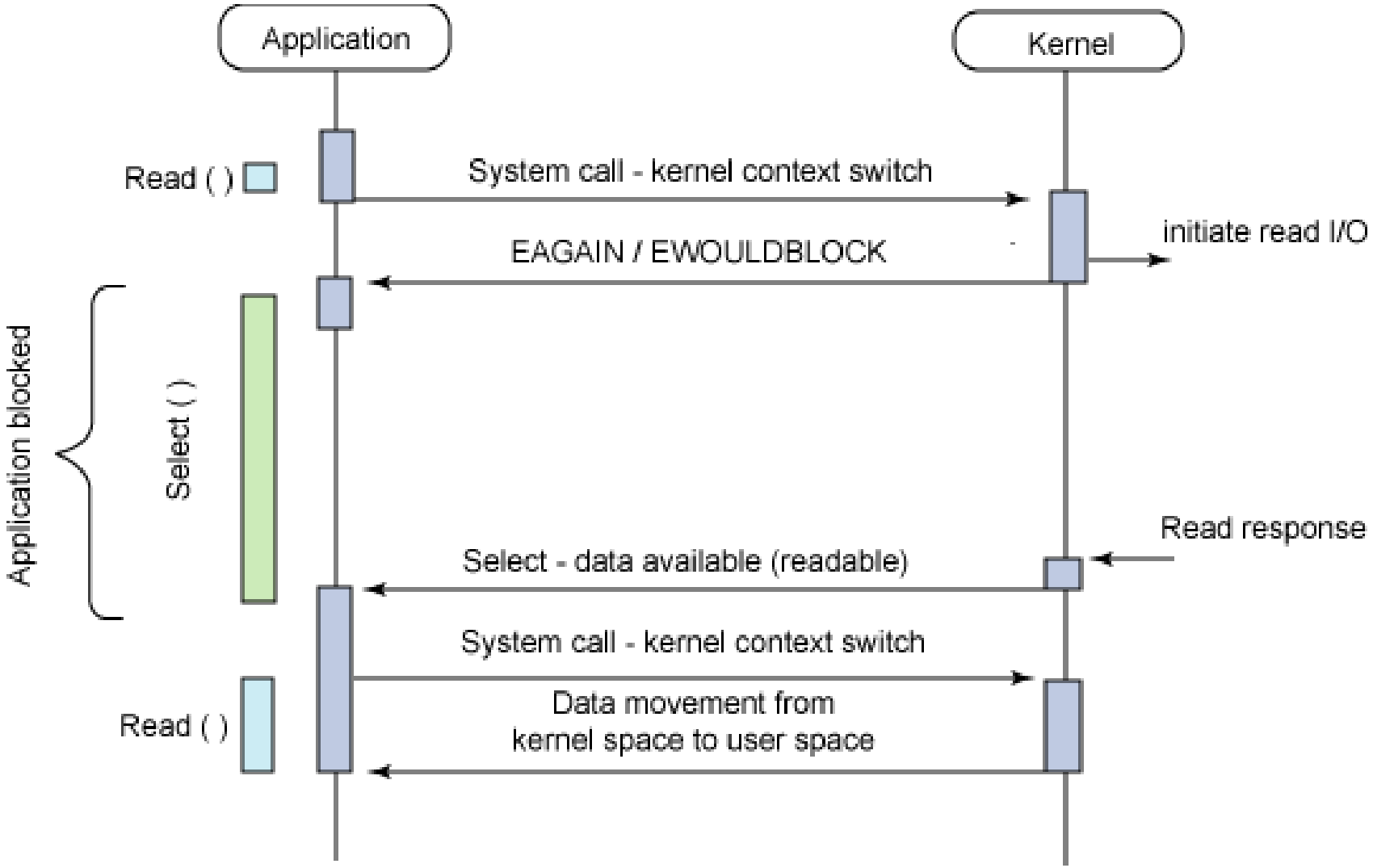
Adapted from: T.Jones Boost application performance using asynchronous I/O. Think About threads in this context! Which model needs tons of threads to handle more Channels?



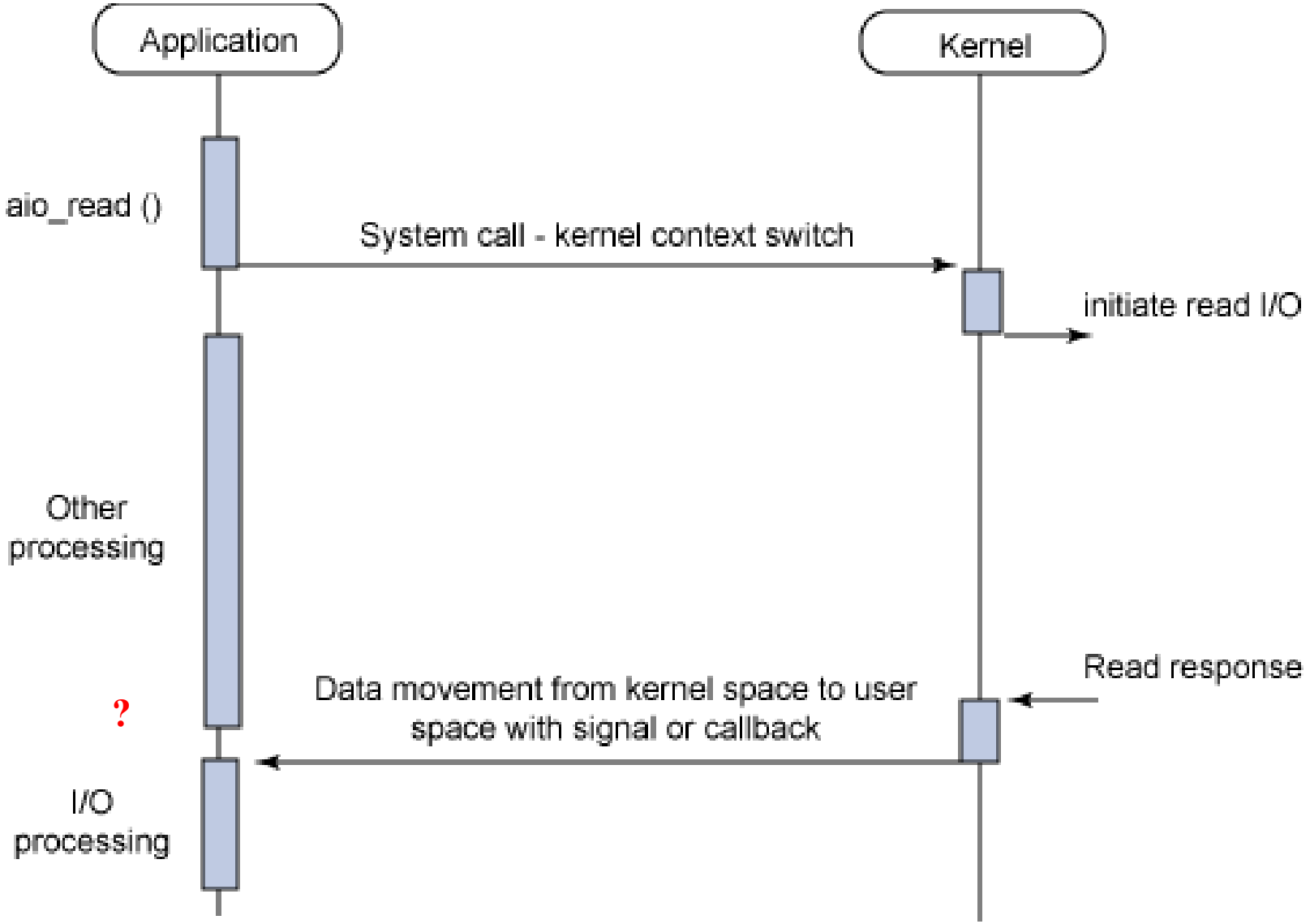
From: [Jones], Blocking Synchronous I/O



Adapted from: [Jones]. Non-Blocking Synchronous I/O
Allows alternating I/O and other app. processing

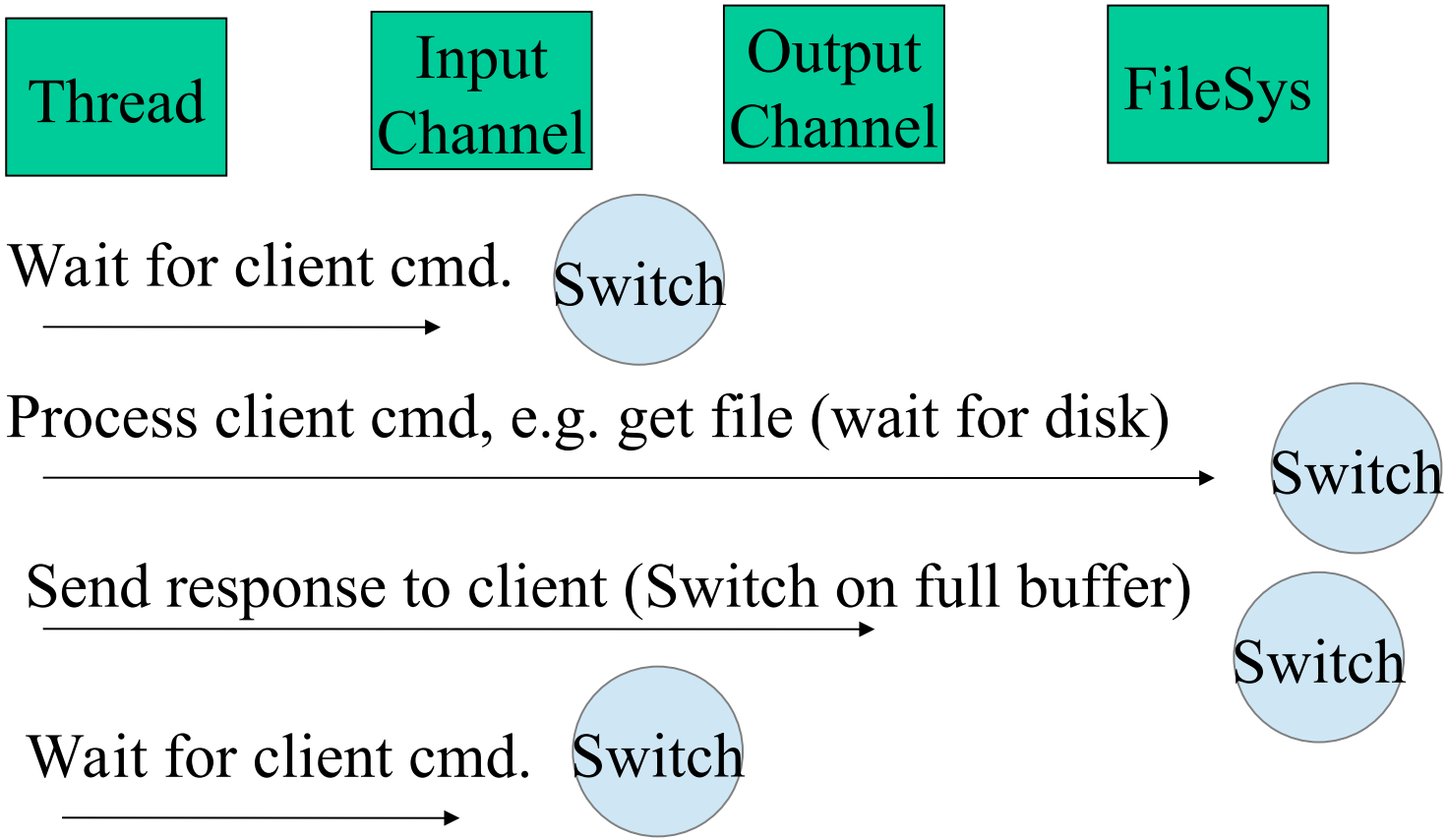


From: [Jones] Blocking Asynchronous I/O (Event loop)



From: [Jones]. True Asynchronous Non-Blocking I/O. How are data moved? Is application processing interrupted? When is completion signaled? Does application wait for completion signals? Are data-races possible?

Synchronous I/O (blocking calls)



Many threads are required to stay responsive. Many context switches occur and each thread needs extra memory. Latency hiding through multiple threads see: Aruna Kalaqanan et.al. http://www-128.ibm.com/developerworks/java/library/j-javaio

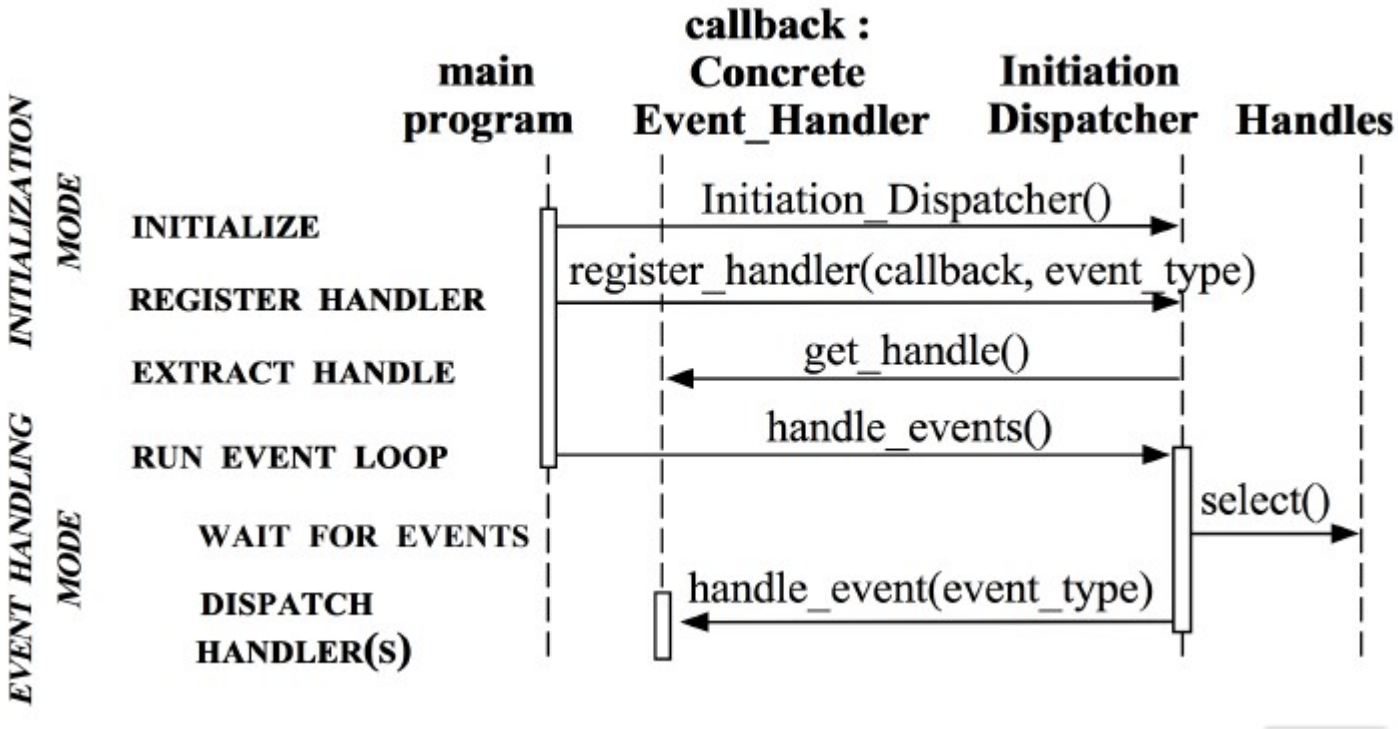
Non-Blocking: Reactor Pattern

"Server applications in a distributed system must handle multiple clients that send them service requests. Before invoking a specific service, however, the server application must demultiplex and dispatch each incoming request to its corresponding service provider. The Reactor pattern serves precisely this function. It allows event-driven applications to demultiplex and dispatch service requests, which are then delivered concurrently to an application from one or more clients."

The Reactor pattern is closely related to the Observer pattern in this aspect: all dependents are informed when a single subject changes. The Observer pattern is associated with a single source of events, however, whereas the Reactor pattern is associated with multiple sources of events."

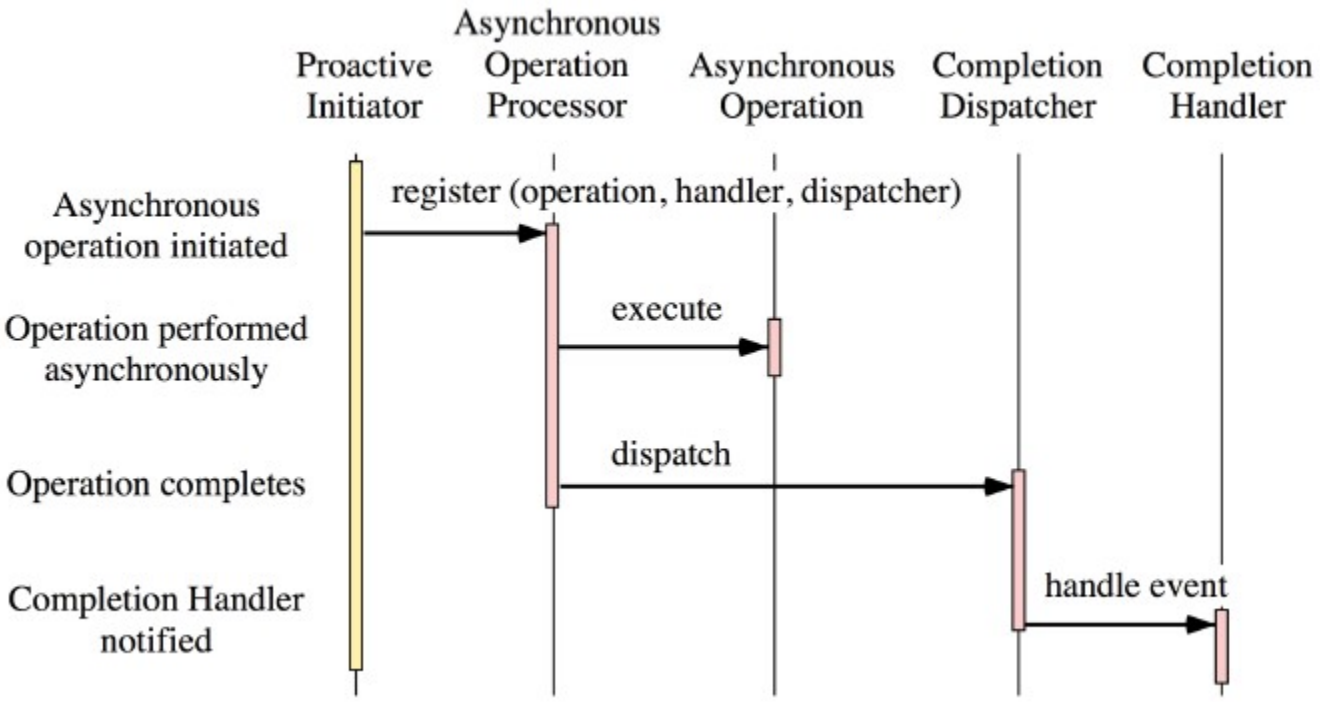
From: Aruna Kalaqanan et.al. http://www-128.ibm.com/developerworks/java/library/j-javaio. The downside: all processing needs to be non-blocking and the threads need to maintain the state of the processing between handler calls (explicit state management vs. implicit in normal multi-threaded designs).

Reactor Pattern



From: Benedikt Hensle, Reaktive Programmierung

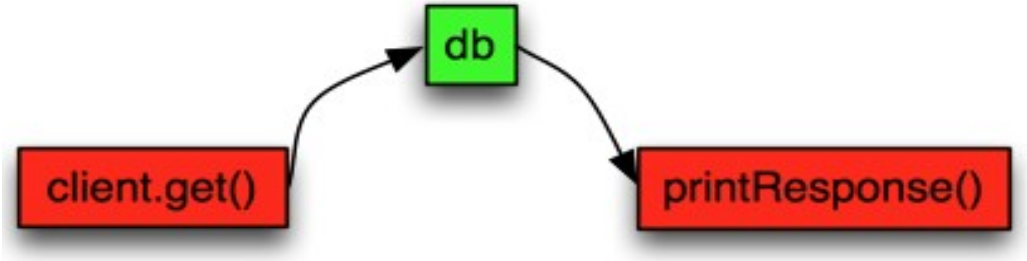
Proactor Pattern



From: Benedikt Hensle, Reaktive Programmierung

Example Node.js Event Loop

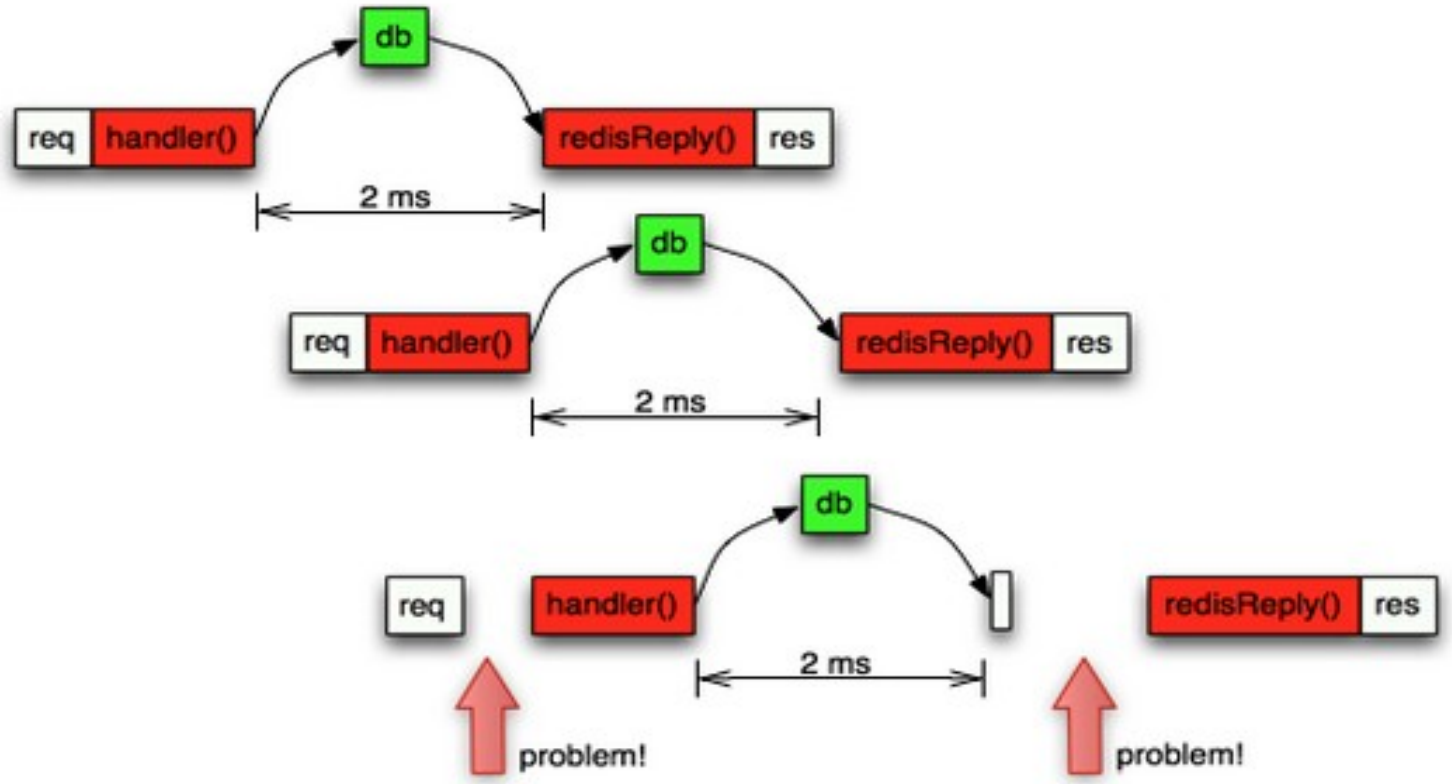
```
var redis = require('redis'), client = redis.createClient();
client.get("mykey", function printResponse(err, reply) { console.log(reply); });
```



1. client.get sends network packet and yields
2. eventloop sets marker for future response packet
3. network stack receives message
4. eventloop calls client with message.
5. redis client calls callback (printResponse)

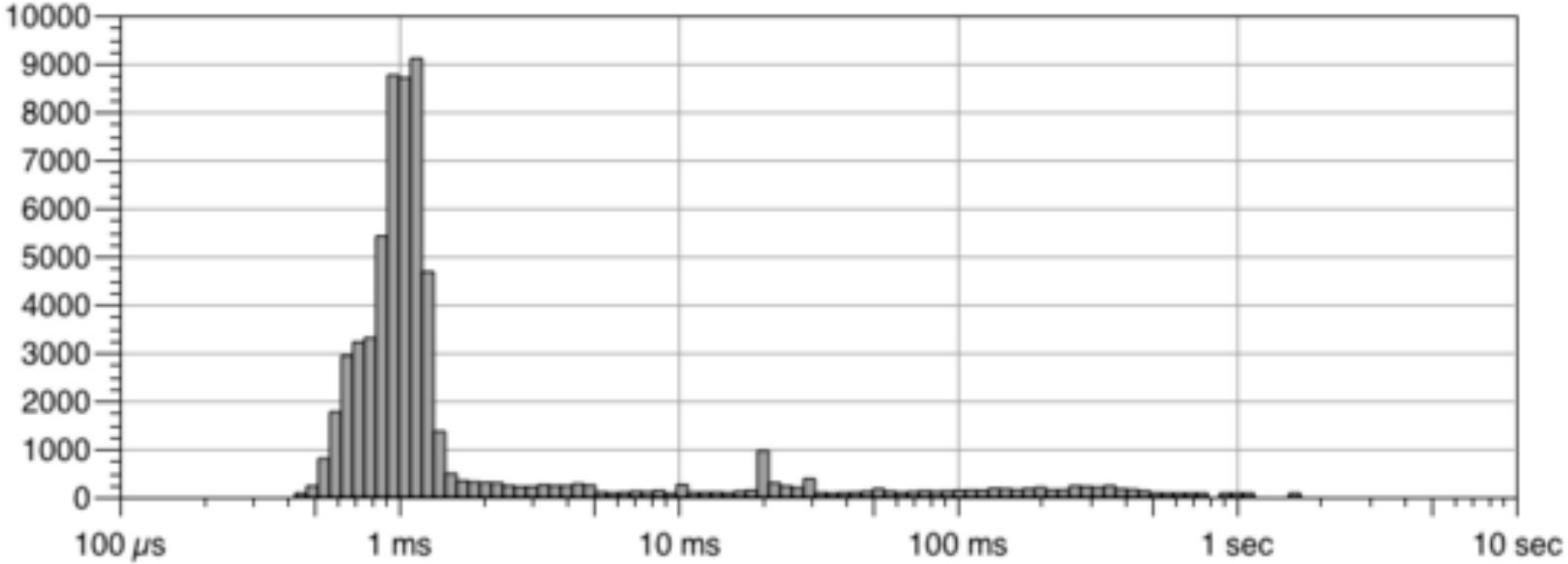
From: Juho Mäkinen, Problems with Node.js Event Loop, http://www.juhonkoti.net/2015/12/01/problems-with-node-js-event-loop. Excellent explanation of async-single-threaded processing of I/O

Event Loop Request Stalls



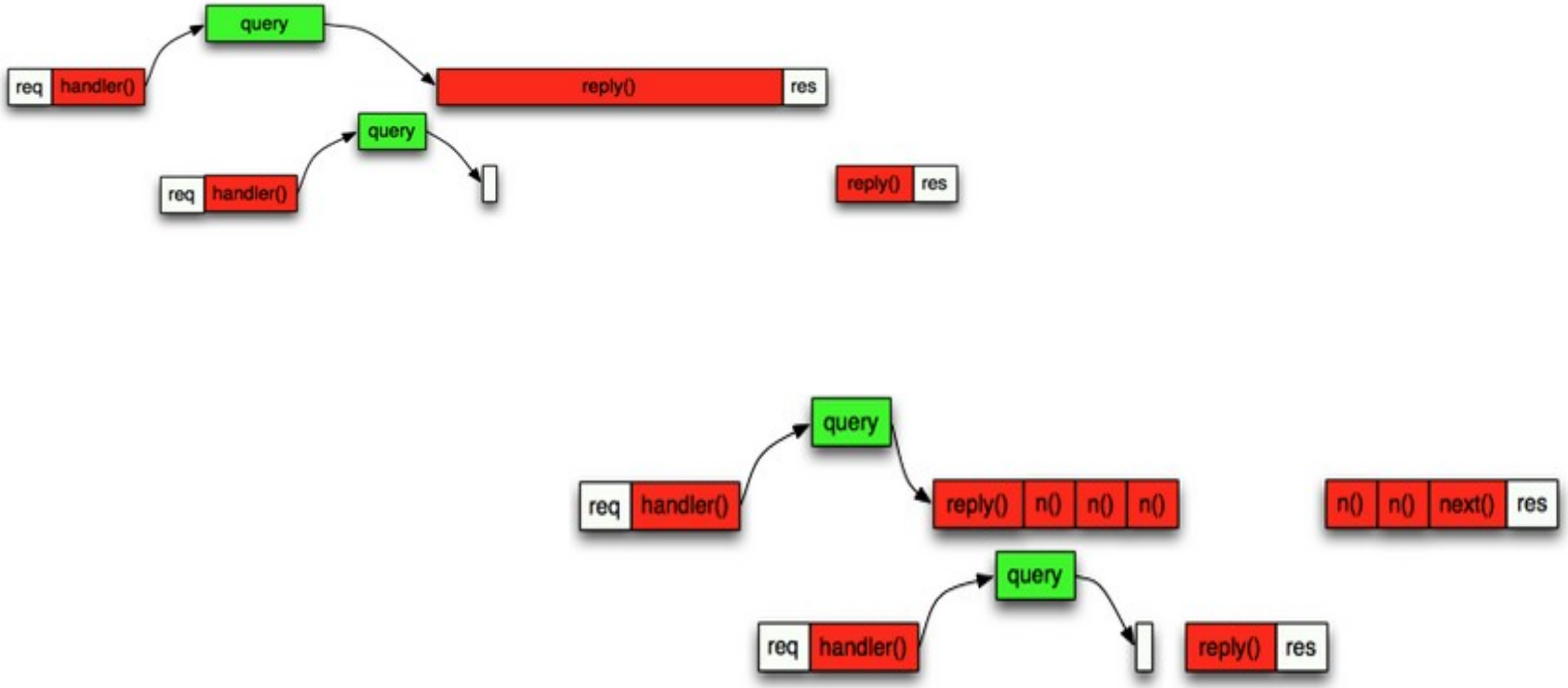
From: Juho Mäkinen, Problems with Node.js Event Loop, http://www.juhonkoti.net/2015/12/01/problems-with-node-js-event-loop. The EL is busy processing replies and can't deal with new requests.

Stalls and Long Tails



From: Juho Mäkinen, Problems with Node.js Event Loop, http://www.juhonkoti.net/2015/12/01/problems-with-node-js-event-loop.

Long Processing and NextTick



From: Juho Mäkinen, Problems with Node.js Event Loop, http://www.juhonkoti.net/2015/12/01/problems-with-node-js-event-loop. “NextTick” allows yielding within a processing step – which in turn allows requests being handled by the EL

Computational Complexity and Event Loops

- calculations: partition and use event loop
- Avoid regex-DOS
- use worker-pools (com. Overhead)
- differentiate between I/O and compute worker
- watch out for resource exhaustion and back-pressure
- don’t do O(n) if n is determined by client input

From: Don’t block the Event Loop (or the worker pool)
https://nodejs.org/en/docs/guides/dont-block-the-event-loop/

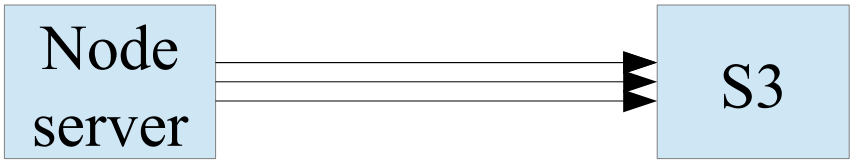
```
function asyncAvg(n, avgCB) {
  // Save ongoing sum in JS closure.
  var sum = 0;
  function help(i, cb) {
    sum += i;
    if (i == n) {
      cb(sum);
      return;
    }

    // "Asynchronous recursion".
    // Schedule next operation asynchronously.
    setImmediate(help.bind(null, i+1, cb));
  }

  // Start the helper, with CB to call avgCB.
  help(1, function(sum){
      var avg = sum/n;
      avgCB(avg);
  });
}

asyncAvg(n, function(avg){
  console.log('avg of 1-n: ' + avg);
});
```

Concept Exercise



Our troublesome Node service had a fairly straightforward purpose. Digg uses Amazon S3 for storage which is peachy, except S3 has no support for batch GET operations. Rather than putting all the onus on our Python web server to request up to 100+ keys at a time from S3, the decision was made to take advantage of Node's easy async code patterns and great concurrency handling. And so Octo, the S3 content fetching service, was born.

Node Octo performed well except for when it didn't. Once a day it needed to handle a traffic spike where the requests per minute jump from 50 to 200+. Also keep in mind that for each request, Octo typically fetches somewhere between 10–100 keys from S3. That's potentially 20,000 S3 GETs a minute. The logs showed that our service slowed down substantially during these spikes, but the trouble was it didn't always recover. As such, we were stuck bouncing our EC2 instances every couple weeks after Octo would seize up and fall flat on its face.

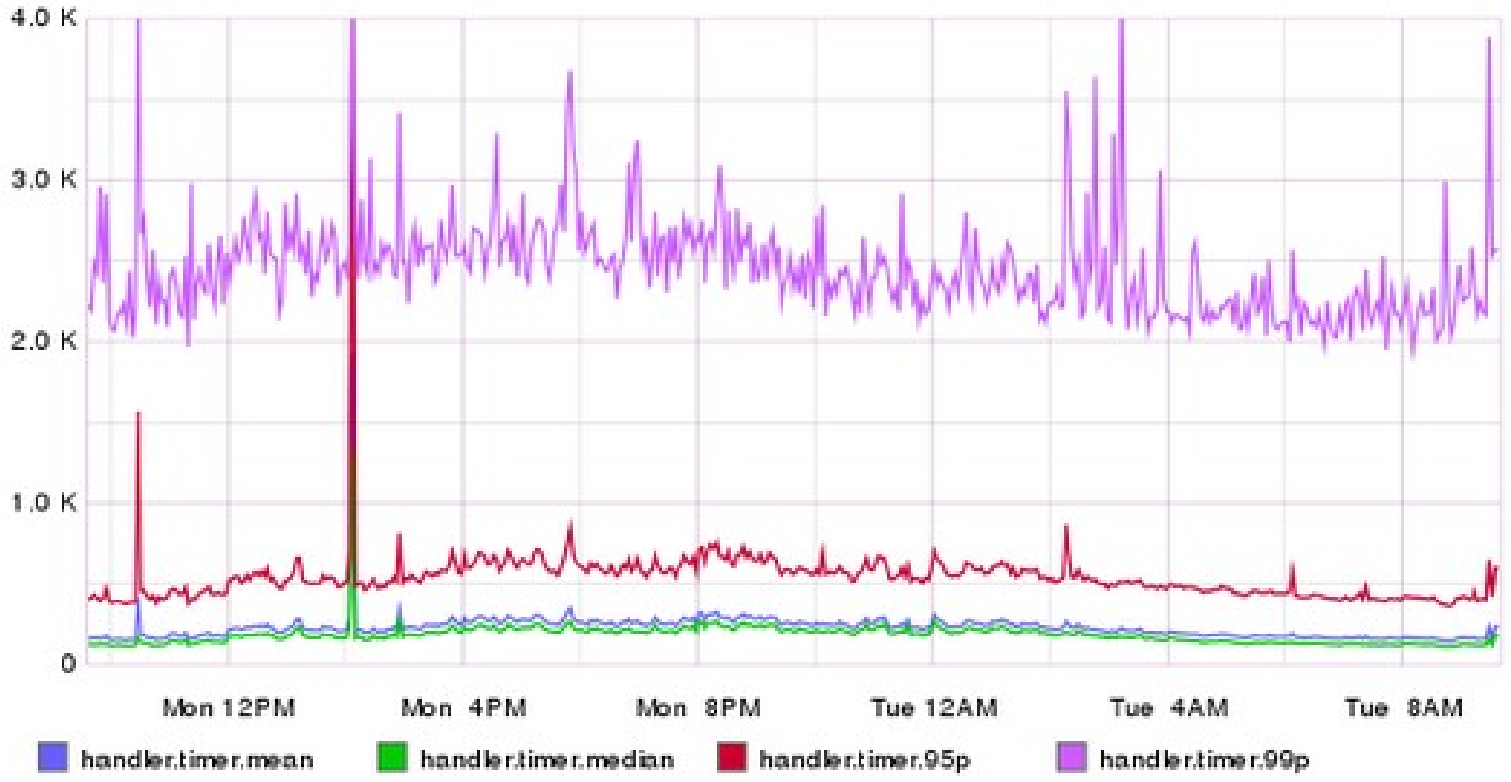
The requests to the service also pass along a strict timeout value. After the clock hits X number of milliseconds since receiving the request, Octo is supposed to return to the client whatever it has successfully fetched from S3 and move on. However, even with a max timeout of 1200ms, in Octo's worst moments we had request handling times spiking up to 10 seconds.

The code was heavily asynchronous and we were caching S3 key values aggressively. Octo was also running across 2 medium EC2 instances which we bumped up to 4.

I reworked the code three times, digging deeper than ever into Node optimizations, gotchas, and tricks for squeezing every last bit of performance out of it. I reviewed benchmarks for popular Node webserver frameworks, like Express or Hapi, vs. Node's built-in HTTP module. I removed any third party modules that, while nice to have, slowed down code execution. The result was three, one-off iterations all suffering from the same issue. No matter how hard I tried, I couldn't get Octo to timeout properly and I couldn't reduce the slow down during request spikes.

A theory eventually emerged...

The Danger of Percentiles



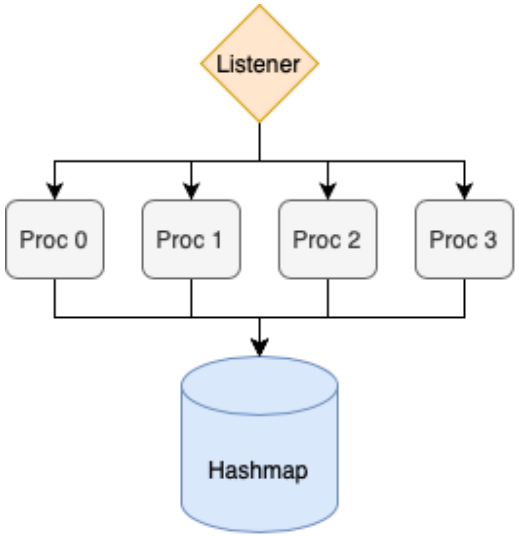
From: Juho Mäkinen, Problems with Node.js Event Loop, http://www.juhonkoti.net/2015/12/01/problems-with-node-js-event-loop. Only a very high percentile shows how bad the situation at the long tail really is.

Questions for I/O Models

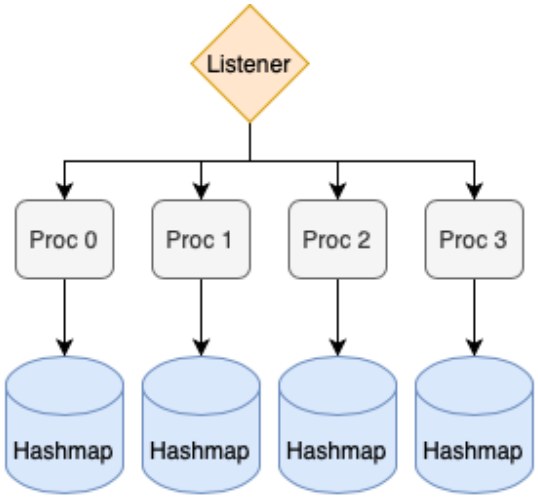
- Can it deal with ALL kinds of input/output?
- How are synchronous channels integrated?
- How hard is programming?
- Can it be combined with multi-cores?
- Scalability through multi-processes?
- Race conditions possible?

Event-driven programming can become really hard in the context of multiple cores.

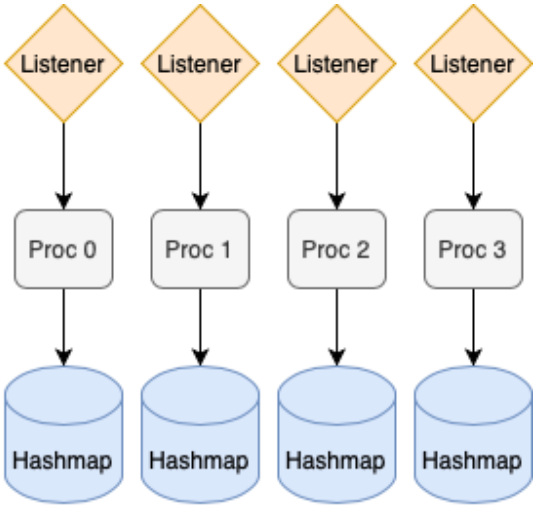
I/O Models and Multi-Cores???



Shared-everything
Redis Multithreading



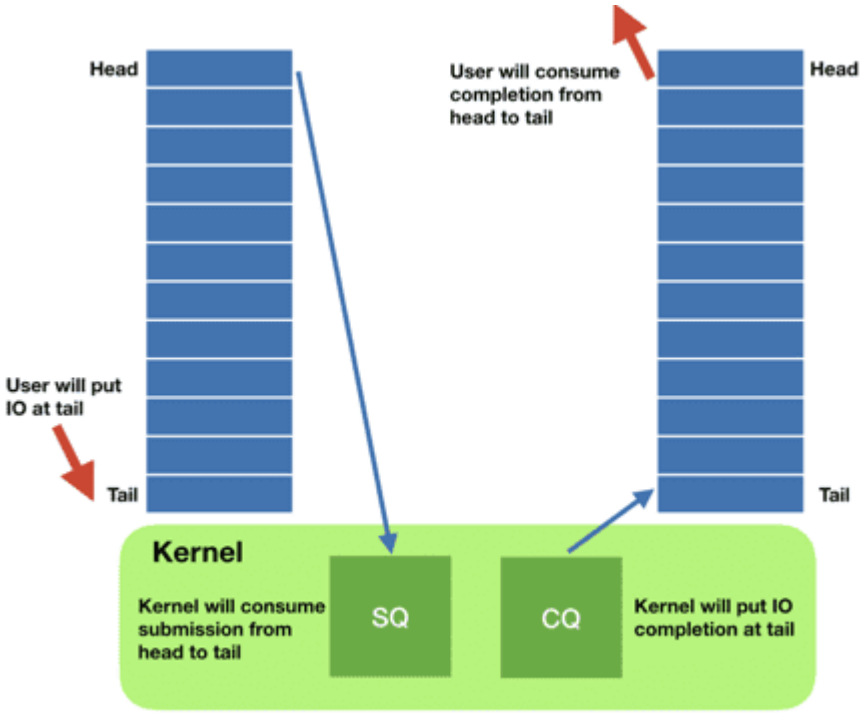
Shared-something
(thread-per-core)



Shared-nothing
Redis Cluster

How can Highly Concurrent Network-Bound Applications benefit from modern multi-core CPUs? By Lucas Crämer

True Async I/O without Locking: IO_uring



https://www.scylladb.com/2020/05/05/how-io_uring-and-ebpf-will-revolutionize-programming-in-linux/

Homework

Read sourcecode of server.java under

Gitlab.mi.hdm-stuttgart.de/kriha/
kriha_examples

And create a sequence diagram for requests

Where would one add

- persistence?
- security?

Resources

- Scaling Ruby Apps to 1000 Requests per Minute - A Beginner's Guide
- by Nate Berkopec, http://www.nateberkopec.com/2015/07/29/scaling-ruby-apps-to-1000-rpm.html
- David Flanagan, Java Examples in a Nutshell, O'Reilly, chapter 5. Code: www.davidflanagan.com/javaexamples3
- Ted Neward, Server Based Java Programming chapter 10, Code:www.manning.com/neward3
- Doug Lea, Concurrent Programming in Java
- Pitt, Fundamental Java Networking (Springer). Good theory and sources (secure sockets, server queuing theory etc.)
- Queuing Theory Portal: http://www2.uwindsor.ca/%7Ehlynka/queue.html
- Performance Analysis of networks: http://www2.sis.pitt.edu/~jkabara/syllabus2120.htm (with simulation tools etc.)
- Meet the experts: Stacy Joines and Gary Hunt on WebSphere performance (performance tools, queue theory etc.) http://www-128.ibm.com/developerworks/websphere/library/techarticles/0507_joines/0507_joines.html
- Doug Lea, Java NIO http://gee.cs.oswego.edu/dl/cpjslides/nio.pdf Learn how to handle thousands of requests per second in Java with a smaller number of threads. Event driven programming, Design patterns like reactor, proactor etc.
- Abhijit Belapurkar, CSP for Java programmers part 1-3. Explains the concept of communicating sequential processes used in JCSP library. Learn how to avoid shared state multithreading and its associated dangers.
- Core tips to Java NIO: http://www.javaperformancetuning.com/tips/nio.shtml
- Schmidt et.al. POSA2 book on design patterns for concurrent systems.
- Nuno Santos, High Peformance servers with Java NIO: http://www.onjava.com/pub/a/onjava/2004/09/01/nio.html?page=3 . Explains design alternatives for NIO. Gives numbers of requests per second possible.
- James Aspnes, Notes on Theory of Distributed Systems, Spring 2014, www.cs.yale.edu/homes/aspnes/classes/465/notes.pdf
- http://bravenewgeek.com/from-the-ground-up-reasoning-about-distributed-systems-in-the-real-world/