Network Applications: High-Performance Server Design (Proactive Async Servers; Operational Analysis; Multi-Servers)

Y. Richard Yang

http://zoo.cs.yale.edu/classes/cs433/

10/11/2018

Outline

- Admin and recap
- High performance server
  - Thread design
  - Asynchronous design
  - Operational analysis
- Multiple servers

Admin

- Assignment Three (HTTP server) Part 1 check point
- Assignment Part 2 posted (there is one to-do place to be fixed today)

Recap: Multiplexed, Reactive I/O

- A different approach for avoiding blocking: peek system state, issue function calls only for those that are ready

Completed connection

sendbuf full or has space

recvbuf empty or has data

- Basic abstractions
  - Channel (source)
  - Selector
  - PCB

server

128.36.232.5
128.36.230.2

TCP socket space

state: listening
address: {*.**6789**, *:*}
completed connection queue: C1; C2
sendbuf:
recvbuf:

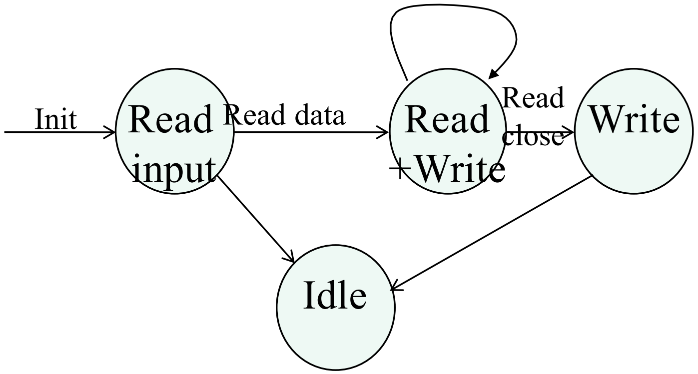
state: established
address: {128.36.232.5:**6789**, 198.69.10.10.**1500**}
sendbuf:
recvbuf:

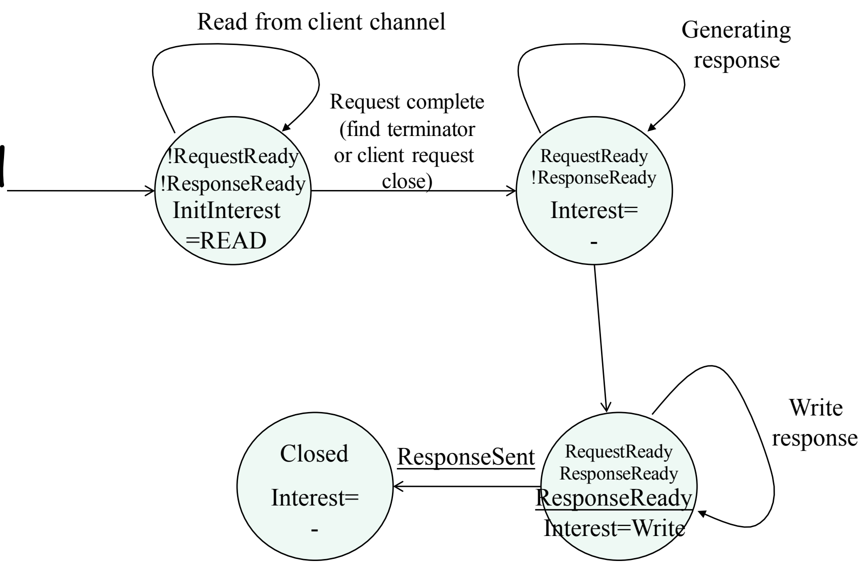
state: established
address: {128.36.232.5:**6789**, 198.69.10.10.**1500**}
sendbuf:
recvbuf:

state: listening
address: {*.**25**, *:*}
completed connection queue:
sendbuf:
recvbuf:

FSM and Reactive Programming

- Designing a good FSM is key for a good non-blocking select design
- There can be multiple types of FSMs
  - Staged: first read request and then write response
  - Mixed: read and write mixed
- Choice depends on protocol and tolerance of complexity, e.g.,
  - HTTP/1.0 channel may use staged
  - HTTP/1.1/2/Chat channel may use mixed





Outline

- Admin and recap
- High performance servers
  - Thread design
    - Per-request thread
    - Thread pool
      - Busy wait
      - Wait/notify
  - Asynchronous design
    - Overview
    - Nonblocking, selected servers--reactive programming
    - Proactive programming

Basic Idea: Asynchronous Initiation and Callback

- Issue of only peek:
  - Cannot handle initiation calls (e.g., read file, initiate a connection by a network client)

- Idea: asynchronous initiation (e.g., aio_read) and program specified completion handler (callback)
  - Also referred to as proactive (Proactor) nonblocking

Asynchronous Channel using Future/Completion Handler

- Java 7 introduces ASynchronousServerSocketChannel and ASynchornousSocketChannel beyond ServerSocketChannel and SocketChannel
  - accept, connect, read, write return Futures or have a callback.

https://docs.oracle.com/javase/7/docs/api/java/nio/channels/AsynchronousServerSocketChannel.html

https://docs.oracle.com/javase/7/docs/api/java/nio/channels/AsynchronousSocketChannel.html

Asynchronous I/O

Asynchronous I/O	Description
AsynchronousFileChannel	An asynchronous channel for reading, writing, and manipulating a file
AsynchronousSocketChannel	An asynchronous channel to a stream-oriented connecting socket
AsynchronousServerSocketChannel	An asynchronous channel to a stream-oriented listening socket
CompletionHandler	A handler for consuming the result of an asynchronous operation
AsynchronousChannelGroup	A grouping of asynchronous channels for the purpose of resource sharing

- https://docs.oracle.com/javase/8/docs/api/java/nio/channels/package-summary.html

Example Async Calls

abstract Future <AsynchronousSocketChannel>	accept(): Accepts a connection.
abstract <A> void	accept(A attachment, CompletionHandler<AsynchronousSocketChannel,? super A> handler): Accepts a connection.

abstract Future<Integer>	read(ByteBuffer dst): Reads a sequence of bytes from this channel into the given buffer.
abstract <A> void	read(ByteBuffer[] dsts, int offset, int length, long timeout, TimeUnit unit, A attachment, CompletionHandler<Long,? super A> handler): Reads a sequence of bytes from this channel into a subsequence of the given buffers.

https://docs.oracle.com/javase/8/docs/api/java/nio/channels/AsynchronousServerSocketChannel.html

Using Future

```
SocketAddress address
   = new InetSocketAddress(args[0], port);
AsynchronousSocketChannel client
  =  AsynchronousSocketChannel.open();
Future<Void> connected
  = client.connect(address);

ByteBuffer buffer = ByteBuffer.allocate(100);

// wait for the connection to finish
connected.get();

// read from the connection
Future<Integer> future = client.read(buffer);

// do other things...

// wait for the read to finish...
future.get();

// flip and drain the buffer
buffer.flip();
WritableByteChannel out
   = Channels.newChannel(System.out);
out.write(buffer);
```

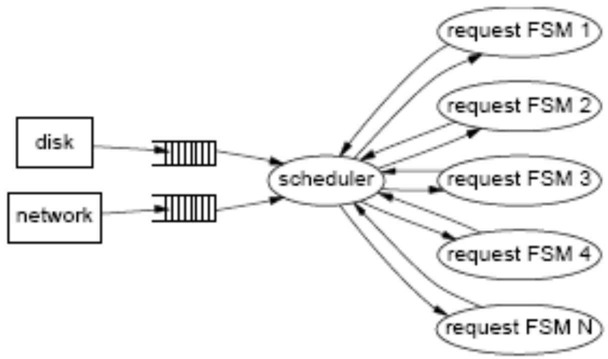
Using CompletionHandler

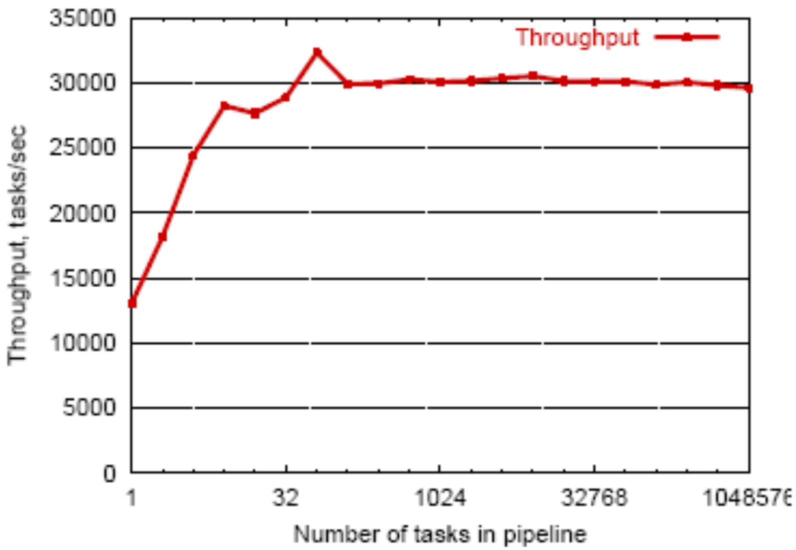
```
class LineHandler implements
CompletionHandler<Integer, ByteBuffer> {

  @Override
  public void completed(Integer result, ByteBuffer buffer)
{
    buffer.flip();
    WritableByteChannel out
         = Channels.newChannel(System.out);
    try {
      out.write(buffer);
    } catch (IOException ex) {
      System.err.println(ex);
    }
  }

  @Override
  public void failed(Throwable ex,
                           ByteBuffer attachment) {
    System.err.println(ex.getMessage());
  }
}

ByteBuffer buffer = ByteBuffer.allocate(100);
CompletionHandler<Integer, ByteBuffer>
       handler = new LineHandler();
channel.read(buffer, buffer, handler);
```

Asynchronous Channel Implementation

- Asynchronous is typically based on Thread pool. If you are curious on its implementation, please read https://docs.oracle.com/javase/8/docs/api/java/nio/channels/AsynchronousChannelGroup.html

Summary: Event-Driven (Asynchronous) Programming

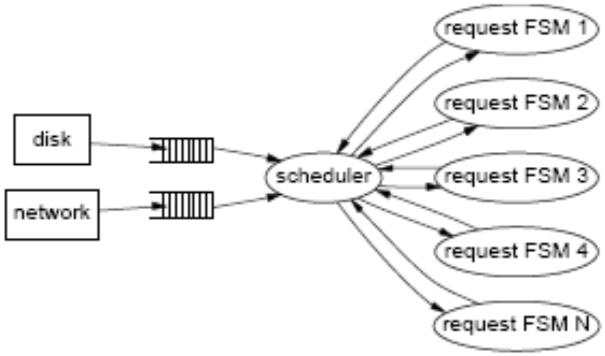




- Advantages
  - Single address space for ease of sharing
  - No synchronization/thread overhead
- Many examples: Google Chrome (libevent), Dropbox (libevent), nginx, click router, NOX controller, …

Problems of Event-Driven Server

- Obscure control flow for programmers and tools
- Difficult to engineer, modularize, and tune
- Difficult for performance/failure isolation between FSMs



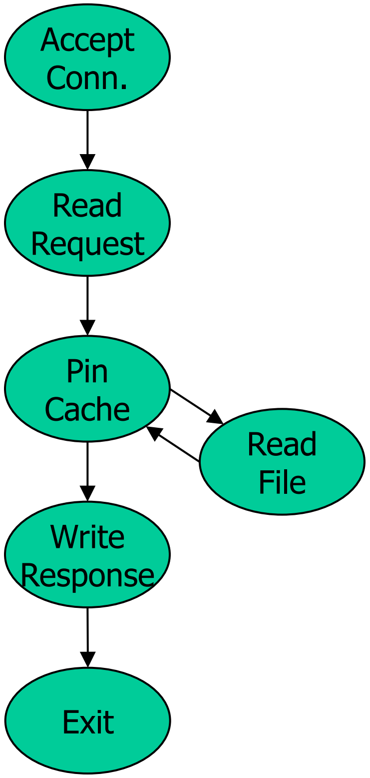
Another view

- Events obscure control flow
  - For programmers *and* tools

Threads	*Events*

<pre>thread_main(int sock) {
    struct session s;
    accept_conn(sock, &s);
    read_request(&s);
    pin_cache(&s);
    write_response(&s);
    unpin(&s);
}
 	
pin_cache(struct session *s) {
    pin(&s);
    if(!in_cache(&s))
        read_file(&s);
}</pre>
	<pre>AcceptHandler(event e) {
	    struct session *s = new_session(e);
	    RequestHandler.enqueue(s);
	}
	RequestHandler(struct session *s) {
	    …; CacheHandler.enqueue(s);
	}
	CacheHandler(struct session *s) {
	    pin(s);
	    if(!in_cache(s)) ReadFileHandler.enqueue(s);
	    else ResponseHandler.enqueue(s);
	}
	. . .
	ExitHandlerr(struct session *s) {
	    …; unpin(&s); free_session(s); }</pre>

Web Server



[von Behren]

Summary: The High-Performance Network Servers Journey

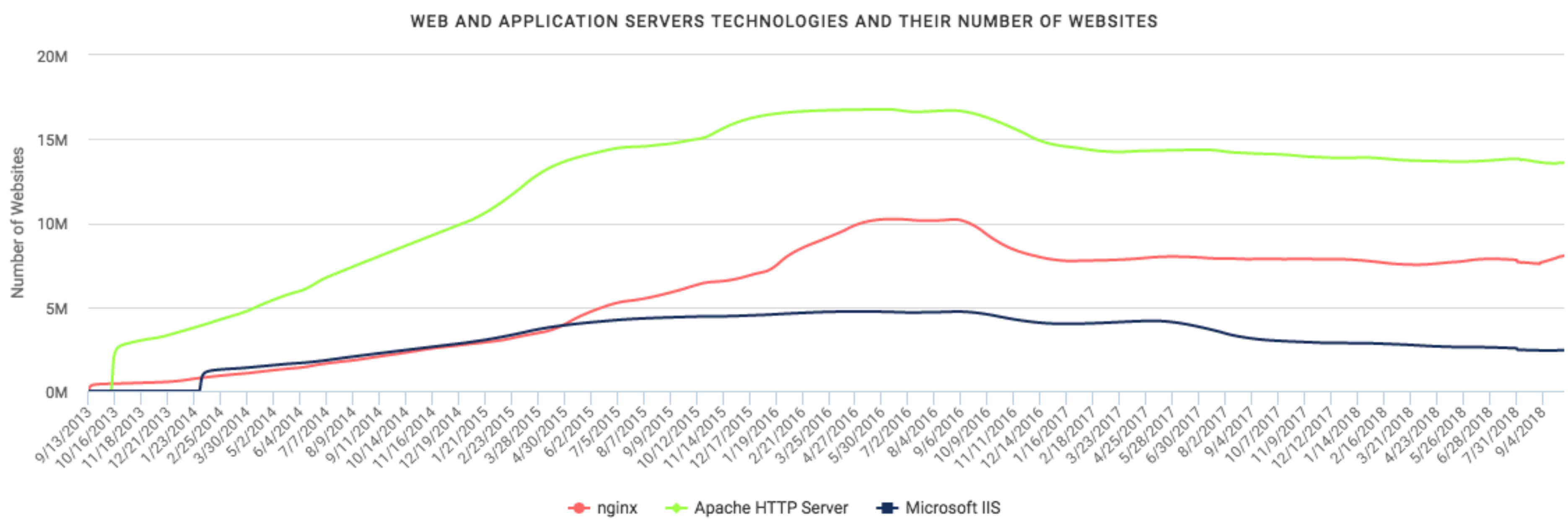
- Avoid blocking (so that we can reach bottleneck throughput)
  - introduce threads, async select, async callback
- Limit unlimited thread overhead
  - Thread pool (share welcome, share Q)
- Coordinating data access
  - synchronization (lock, condition, synchronized)
- Coordinating behavior: avoid busy-wait
  - wait/notify; select FSM, Future/Listener
- Extensibility of SW/robustness
  - language support/design using interfaces

Beyond Java: Design Patterns

- We have seen Java as an example
- C++ and C# can be quite similar. For C++ and general design patterns:
  - http://www.cs.wustl.edu/~schmidt/PDF/OOCP-tutorial4.pdf
  - http://www.stal.de/Downloads/ADC2004/pra03.pdf

HTTP Servers

Ranking	Technology	Domains	Market Share
1	Apache HTTP Server	13,593,009	52.41%
2	nginx	8,244,455	31.79%
3	Microsoft IIS	2,443,642	9.42%



https://www.datanyze.com/market-share/web-and-application-servers

Summary: Server Software Architecture

- Architectures
  - Multi threads
  - Asynchronous
  - Hybrid
    - Assigned reading: SEDA
    - Netty design

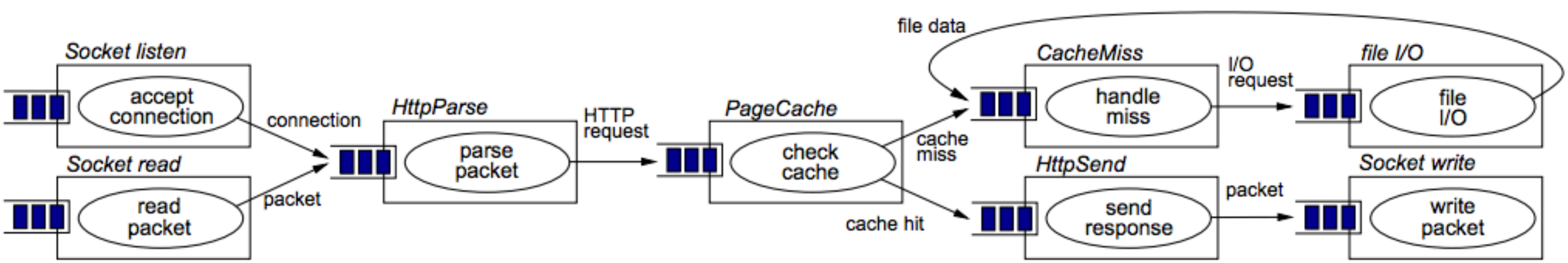
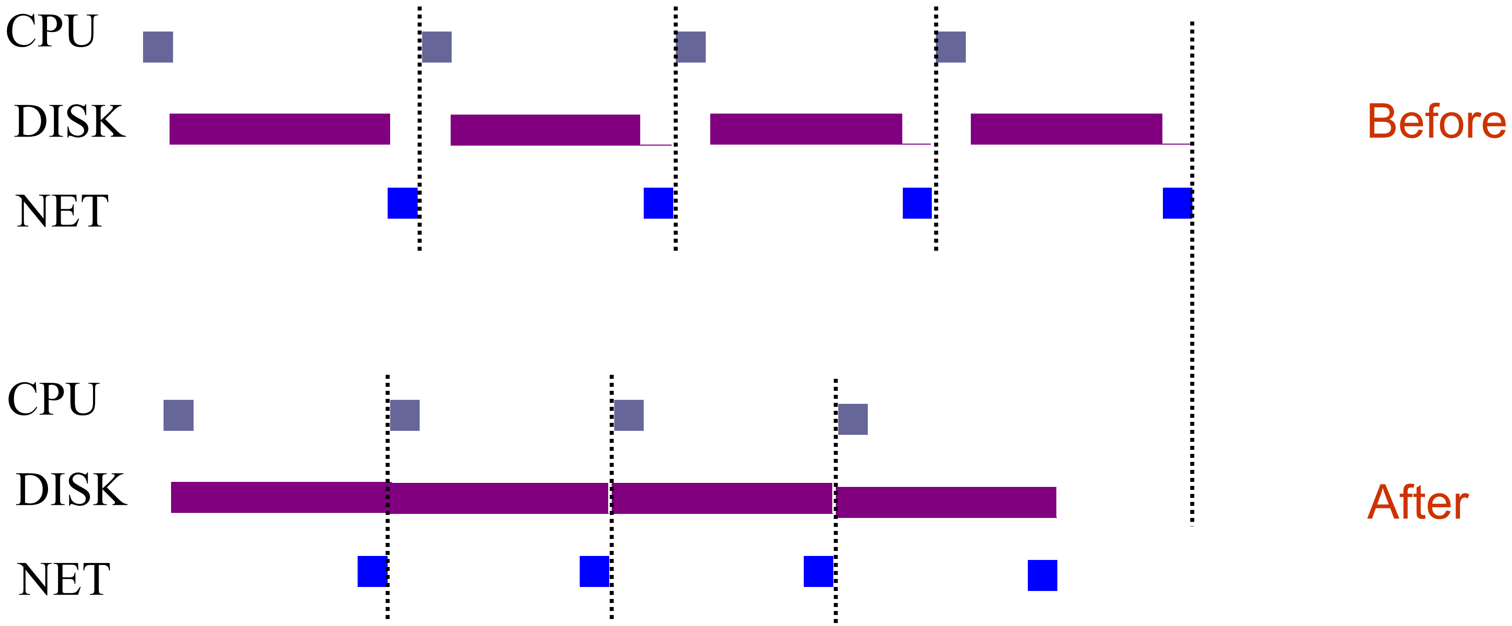
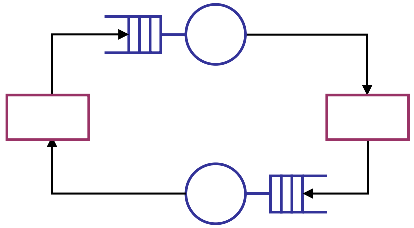


Figure 5: **Staged event-driven (SEDA) HTTP server:** *This is a structural representation of the SEDA-based Web server, described in detail in Section 5.1. The application is composed as a set of stages separated by queues. Edges represent the flow of events between stages. Each stage can be independently managed, and stages can be run in sequence or in parallel, or a combination of the two. The use of event queues allows each stage to be individually load-conditioned, for example, by thresholding its event queue. For simplicity, some event paths and stages have been elided from this figure.*

Recap: Best Server Design Limited Only by Resource Bottleneck





Some Questions

- When is CPU the bottleneck for scalability?
  - So that we need to add helper threads
- How do we know that we are reaching the limit of scalability of a single machine?

- These questions drive network server architecture design

- Some basic performance analysis techniques are good to have

Outline

- Admin and recap
- High performance server
  - Thread design
  - Asynchronous design
  - Operational analysis
- Multiple servers

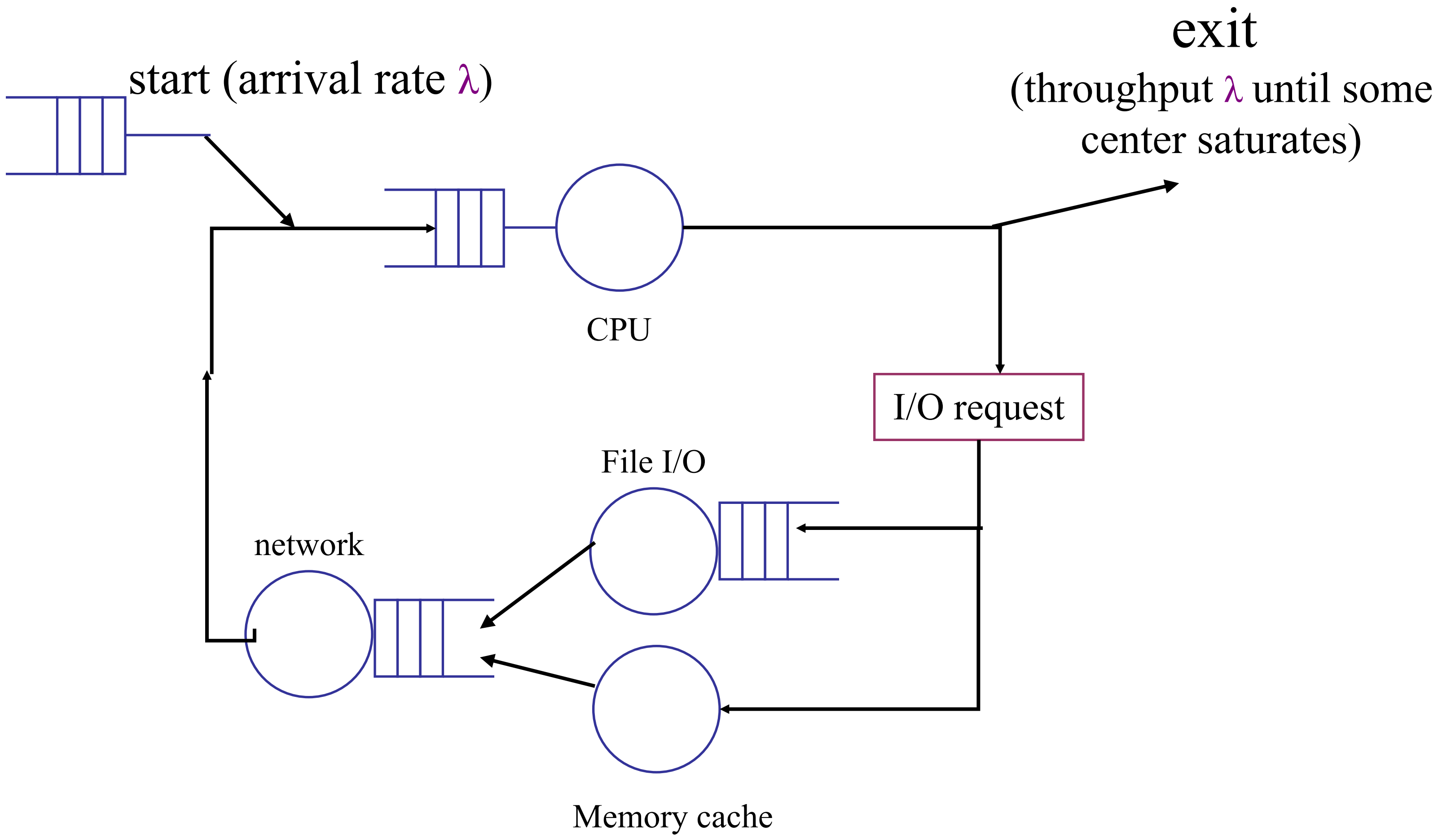
Operational Analysis

- Relationships that do not require any assumptions about the distribution of service times or inter-arrival times
  - Hence focus on measurements

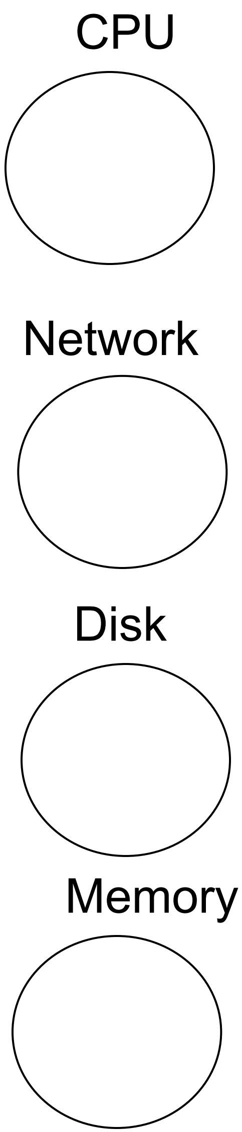
- Identified originally by Buzen (1976) and later extended by Denning and Buzen (1978).

- We touch only some techniques/results
  - In particular, bottleneck analysis
- More details see linked reading

Under the Hood (An example FSM)



Operational Analysis: Resource Demand of a Request



V_{CPU} visits for S_{CPU} units of resource time per visit

V_{Net} visits for S_{Net} units of resource time per visit

V_{Disk} visits for S_{Disk} units of resource time per visit

V_{Mem} visits for S_{Mem} units of resource time per visit

Operational Quantities

- T: observation interval
- Bi: busy time of device i
- i = 0 denotes system

Ai: # arrivals to device i
Ci: # completions at device i

$$\text{arrival rate } \lambda_i = \frac{A_i}{T}$$

$$\text{Throughput } X_i = \frac{C_i}{T}$$

$$\text{Utilization } U_i = \frac{B_i}{T}$$

$$\text{Mean service time } S_i = \frac{B_i}{C_i}$$

Utilization Law

$$\text{Utilization } U_i = \frac{B_i}{T}$$

$$= \frac{C_i}{T}\frac{B_i}{C_i}$$

$$= X_i S_i$$

- The law is independent of any assumption on arrival/service process
- Example: Suppose NIC processes 125 pkts/sec, and each pkt takes 2 ms. What is utilization of the network NIC?

Deriving Relationship Between R, U, and S for one Device

- Assume flow balanced (arrival=throughput), Little's Law:

$$Q = \lambda R = XR$$

- Assume PASTA (Poisson Arrival--memory-less arrival--Sees Time Average), a new request sees Q ahead of it, and FIFO

$$R = S + QS = S + XRS$$

- According to utilization law, U = XS

$$R = S + UR \longrightarrow R = \frac{S}{1-U}$$

Forced Flow Law

- Assume each request visits device i Vi times

$$\text{Throughput } X_i = \frac{C_i}{T}$$

$$= \frac{C_i}{C_0}\frac{C_0}{T}$$

$$= V_i X$$

Bottleneck Device

$$\text{Utilization } U_i = X_i S_i$$

$$= V_i X S_i$$

$$= X V_i S_i$$

- Define Di = Vi Si as the total demand of a request on device i
- The device with the highest Di has the highest utilization, and thus is called the bottleneck

Bottleneck vs System Throughput

$$\text{Utilization } \mathrm{U}_i = XV_iS_i \le 1$$

$$\rightarrow \; X \le \frac{1}{D_{\max}}$$

Example 1

- A request may need
  - 10 ms CPU execution time
  - 1 Mbytes network bw
  - 1 Mbytes file access where
    - 50% hit in memory cache
- Suppose network bw is 100 Mbps, disk I/O rate is 1 ms per 8 Kbytes (assuming the program reads 8 KB each time)
- Where is the bottleneck?

Example 1 (cont.)

- CPU:
  - D_{CPU}= 10 ms (e.q. 100 requests/s)
- Network:
  - D_{Net} = 1 Mbytes / 100 Mbps = 80 ms (e.q., 12.5 requests/s)
- Disk I/O:
  - Ddisk = 0.5 * 1 ms * 1M/8K = 62.5 ms (e.q. = 16 requests/s)

Example 2

- A request may need
  - 150 ms CPU execution time (e.g., dynamic content)
  - 1 Mbytes network bw
  - 1 Mbytes file access where
    - 50% hit in memory cache
- Suppose network bw is 100 Mbps, disk I/O rate is 1 ms per 8 Kbytes (assuming the program reads 8 KB each time)
- Bottleneck: CPU -> use multiple threads to use more CPUs, if available, to avoid CPU as bottleneck

Interactive Response Time Law

- System setup
  - Closed system with N users (e.g., remote desktops)
  - Each user sends in a request, after response, think time, and then sends next request
  - *Notation*
    - *Z = user think-time, R = Response time*
  - The total cycle time of a user request is $R+Z$

In duration T, #requests generated by each user: $T/(R+Z)$ requests

Interactive Response Time Law

- *If N users and flow balanced:*

$$\text{System Throughput X} = \text{Toal\# req./T}$$

$$= \frac{N\frac{T}{R+Z}}{T}$$

$$= \frac{N}{R+Z}$$

$$R = \frac{N}{X} - Z$$

Bottleneck Analysis

$$X(N) \le \min\{\frac{1}{D_{max}}, \frac{N}{D+Z}\}$$

$$R(N) \ge \max\{D, ND_{max} - Z\}$$

- Here D is the sum of Di

Proof

$$X(N) \le \min\{\tfrac{1}{D_{\max}}, \tfrac{N}{D+Z}\}$$

$$R(N) \ge \max\{D, ND_{\max} - Z\}$$

- We know

$$X \le \frac{1}{D_{\max}} \qquad R(N) \ge D$$

Using interactive response time law:

$$R = \frac{N}{X} - Z \Longrightarrow R \ge ND_{\max} - Z$$

$$X = \frac{N}{R+Z} \Longrightarrow X \le \frac{N}{D+Z}$$

Summary: Operational Laws

- Utilization law: U = XS
- Forced flow law: Xi = Vi X
- Bottleneck device: largest Di = Vi Si
- Little's Law: Qi = Xi Ri
- Bottleneck bound of interactive response (for the given closed model):

$$X(N) \le \min\{\frac{1}{D_{max}}, \frac{N}{D+Z}\}$$

$$R(N) \ge \max\{D, ND_{max} - Z\}$$

In Practice: Common Bottlenecks

- No more file descriptors
- Sockets stuck in TIME_WAIT
- High memory use (swapping)
- CPU overload
- Interrupt (IRQ) overload

[Aaron Bannert]

Offline, Optional Read - Start

http://video.google.com/videoplay?docid=-6304964351441328559#

You Tube

- 02/2005: Founded by Chad Hurley, Steve Chen and Jawed Karim, who were all early employees of PayPal.
- 10/2005: First round of funding ($11.5 M)
- 03/2006: 30 M video views/day
- 07/2006: 100 M video views/day
- 11/2006: acquired by Google
- 10/2009: Chad Hurley announced in a blog that YouTube serving well over 1 B video views/day (avg = 11,574 video views /sec)

Pre-Google Team Size

- 2 Sysadmins
- 2 Scalability software architects
- 2 feature developers
- 2 network engineers
- 1 DBA
- 0 chefs

YouTube Design Alg.

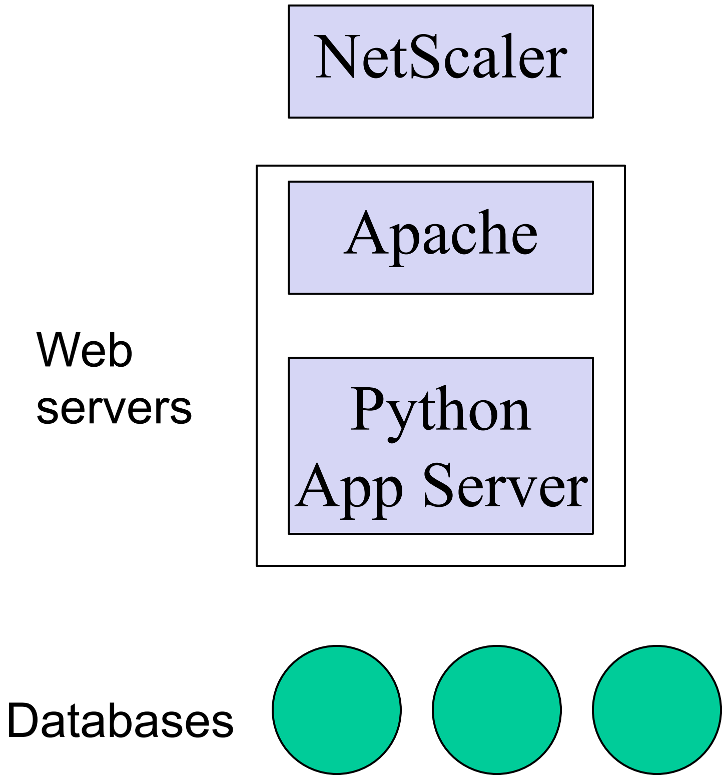
```
while (true)
{
  identify_and_fix_bottlenecks();
  drink();
  sleep();
  notice_new_bottleneck();
}
```

YouTube Major Components

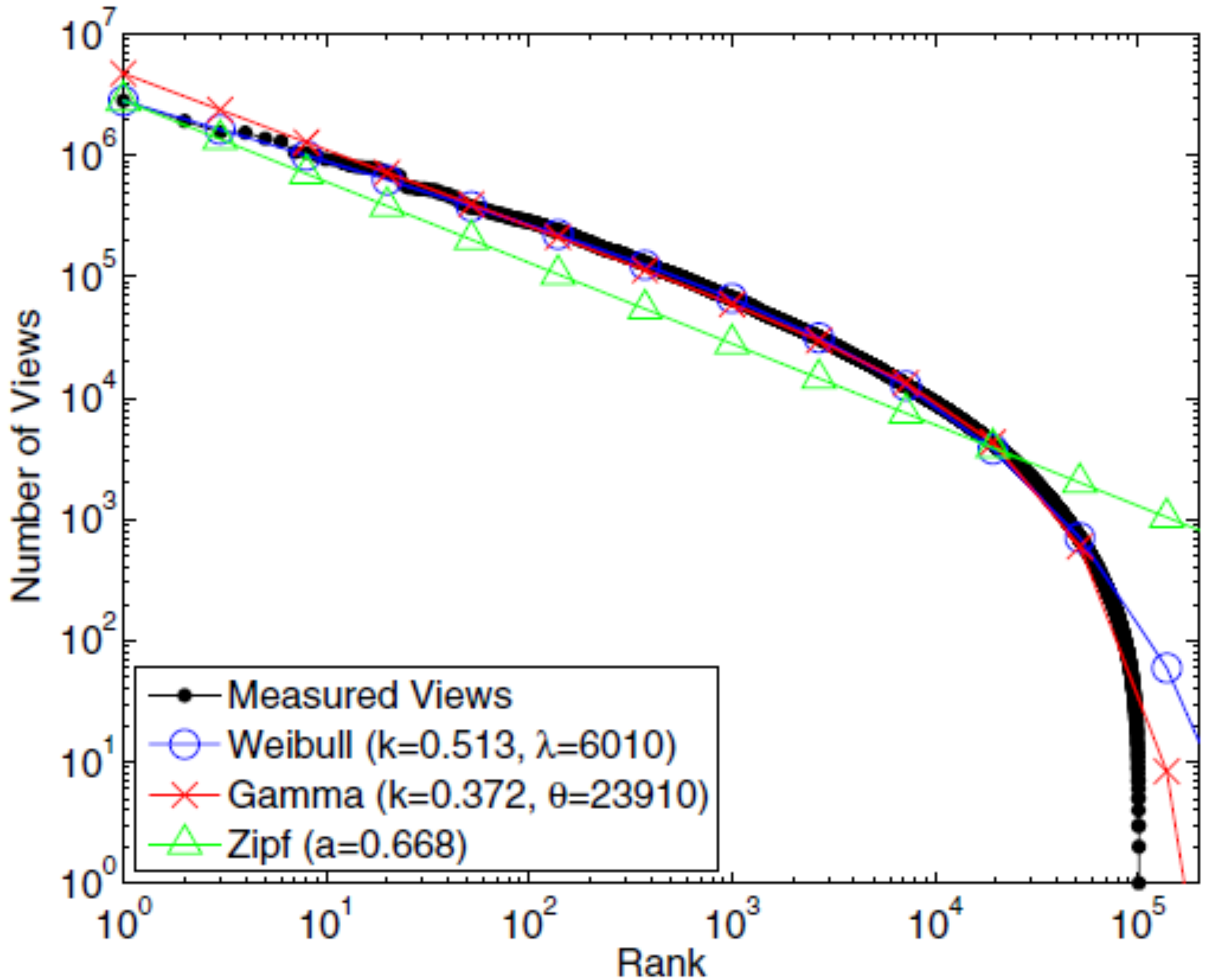
- Web servers
- Video servers
- Thumbnail servers
- Database servers

YouTube: Web Servers

- Components
  - Netscaler load balancer; Apache; Python App Servers; Databases
- Python
  - Web code (CPU) is not bottleneck
    - JIT to C to speedup
    - C extensions
    - Pre-generate HTML responses
  - Development speed more important



YouTube: Video Popularity



See “Statistics and Social Network of YouTube Videos”, 2008.

YouTube: Video Popularity

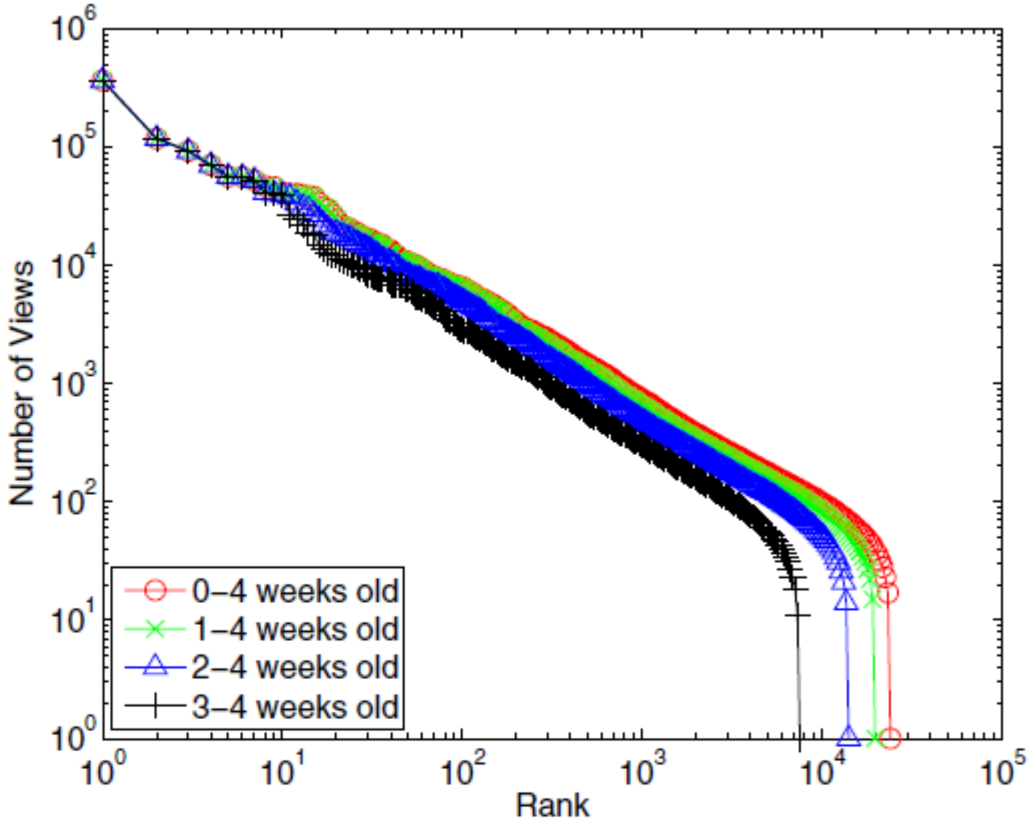


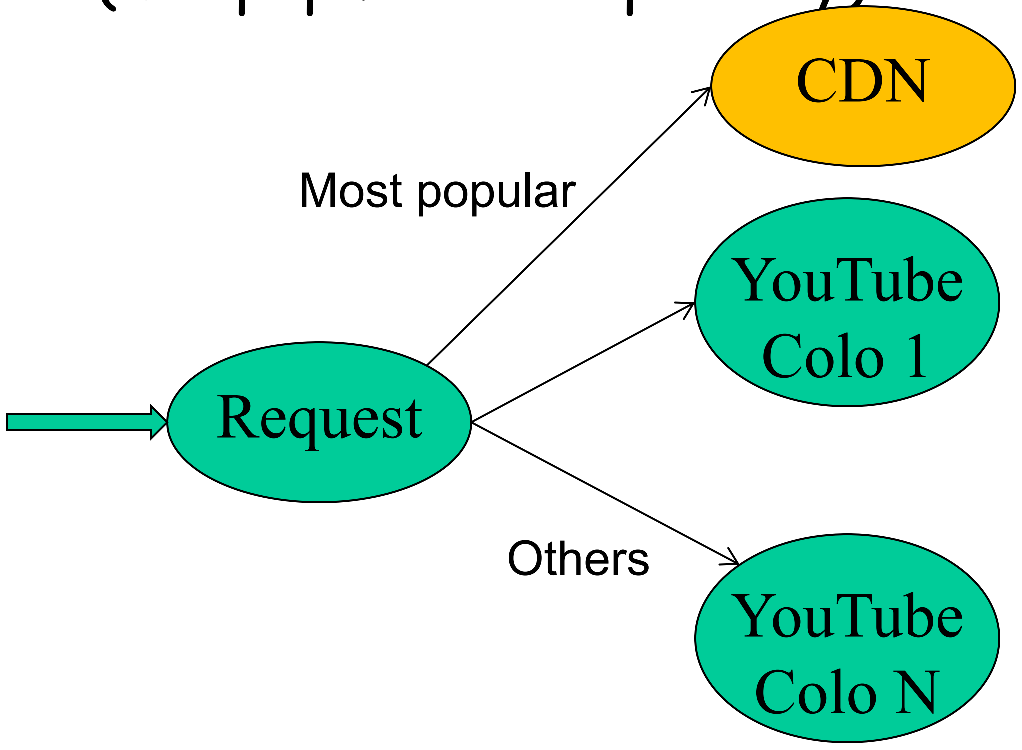
Fig. 8. Recently added YouTube videos rank by popularity

How to design a system to handle highly skewed distribution?

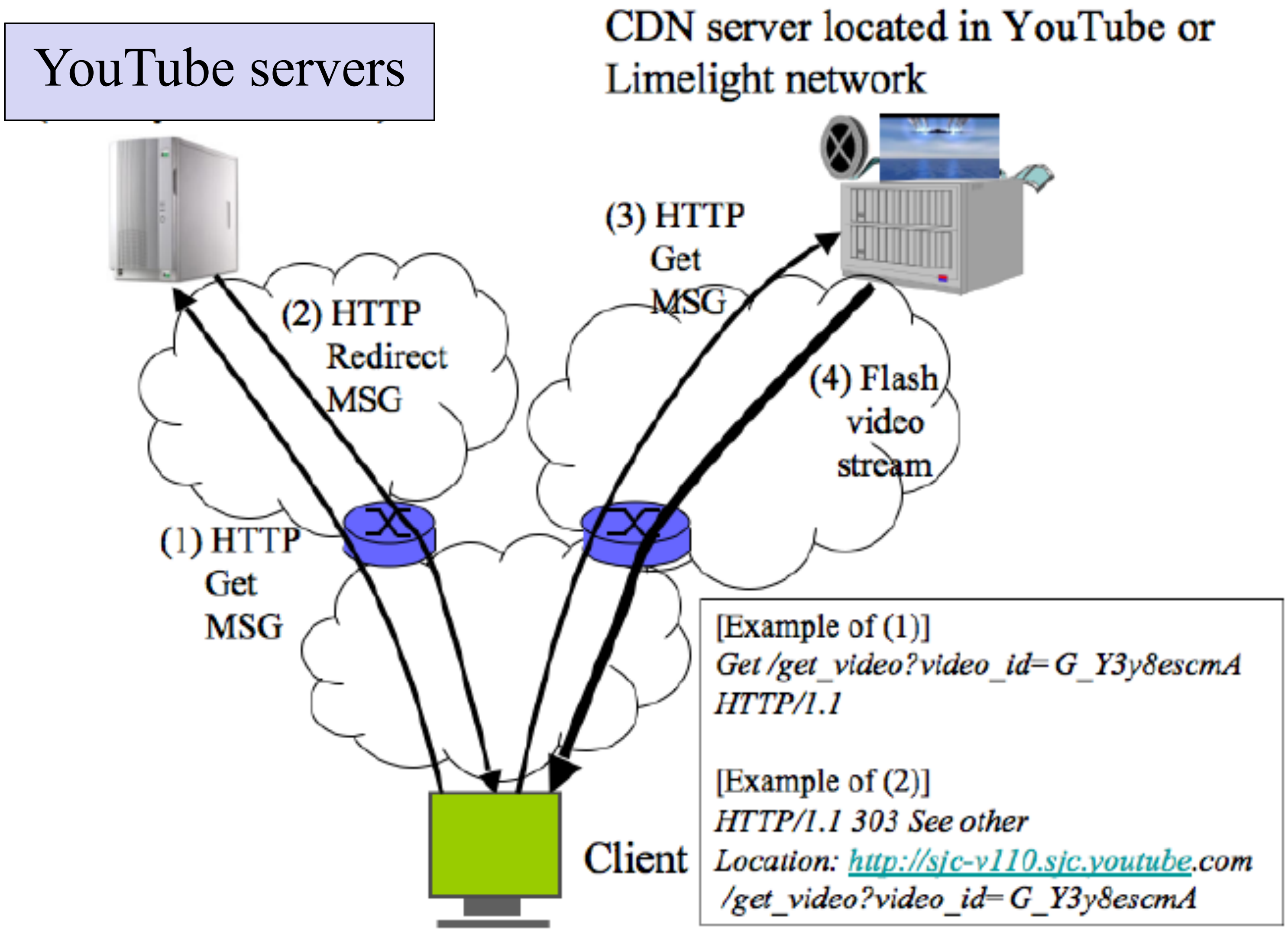
See "Statistics and Social Network of YouTube Videos", 2008.

YouTube: Video Server Architecture

- Tiered architecture
  - CDN servers (for popular videos)
    - Low delay; mostly in-memory operation
  - YouTube servers (not popular 1-20 per day)



YouTube Redirection Architecture



YouTube Video Servers

- Each video hosted by a mini-cluster consisting of multiple machines
- Video servers use the lighttpd web server for video transmission:
  - Apache had too much overhead (used in the first few months and then dropped)
  - Async io: uses epoll to wait on multiple fds
  - Switched from single process to multiple process configuration to handle more connections

Thumbnail Servers

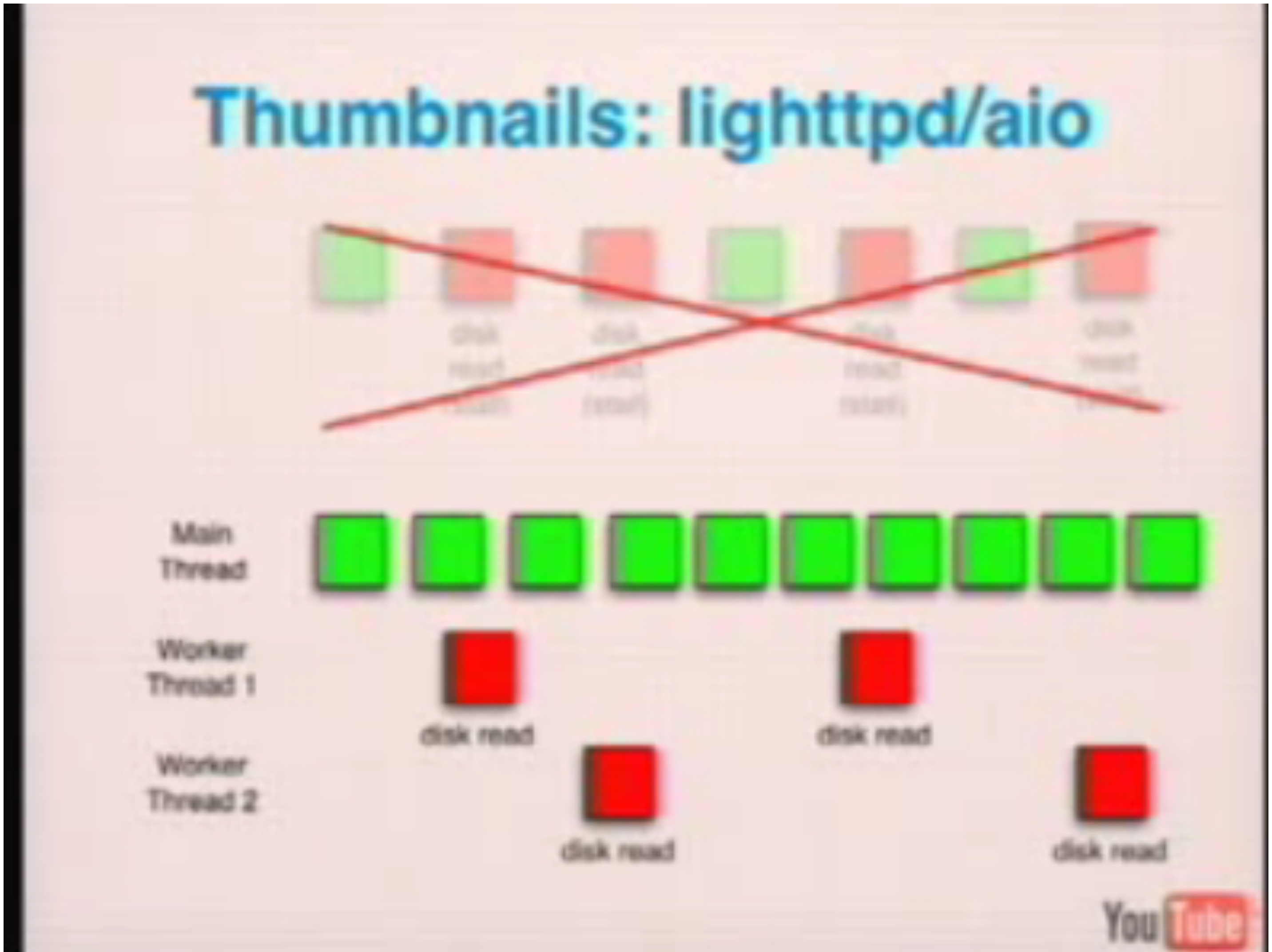
- Thumbnails are served by a few machines
- Problems running thumbnail servers
  - A high number of requests/sec as web pages can display 60 thumbnails on page
  - Serving a lot of small objects implies
    - lots of disk seeks and problems with file systems inode and page caches
    - may ran into per directory file limit

    - Solution: storage switched to Google BigTable

Thumbnail Server Software Architecture

- Design 1: Squid in front of Apache
  - Problems
    - Squid worked for a while, but as load increased performance eventually decreased: Went from 300 requests/second to 20
    - under high loads Apache performed badly, changed to lighttpd
- Design 2: lighttpd default: By default lighttpd uses a single thread
  - Problem: often stalled due to I/O
- Design 3: switched to multiple processes contending on shared accept
  - Problems: high contention overhead/individual caches

Thumbnails Server: lighttpd/aio



Offline, Optional Read - End

Summary: High-Perf. Network Server

- Avoid blocking (so that we can reach bottleneck throughput)
  - Introduce threads, async io
- Limit unlimited thread overhead
  - Thread pool
- Shared variables
  - Synchronization (lock, synchronized, condition)
- Avoid busy-wait
  - Wait/notify; FSM; asynchronous channel/Future/Handler
- Extensibility/robustness
  - Language support/Design for interfaces
- System modeling and measurements
  - Queueing analysis, operational analysis

Outline

- Admin and recap
- High performance server
  - Thread design
  - Asynchronous design
  - Operational analysis
- Multiple servers

Why Multiple Servers?

- Scale a single server that encounters bottleneck
- Scale a single server that has too large latency
- Add fault tolerance to a single server
- Match with settings where resources may be naturally distributed at different machines (e.g., run a single copy of a database server due to single license; access to resource from third party)
- Achieve modular software architecture (e.g., front end, business logic, and database)

FB Data Centers

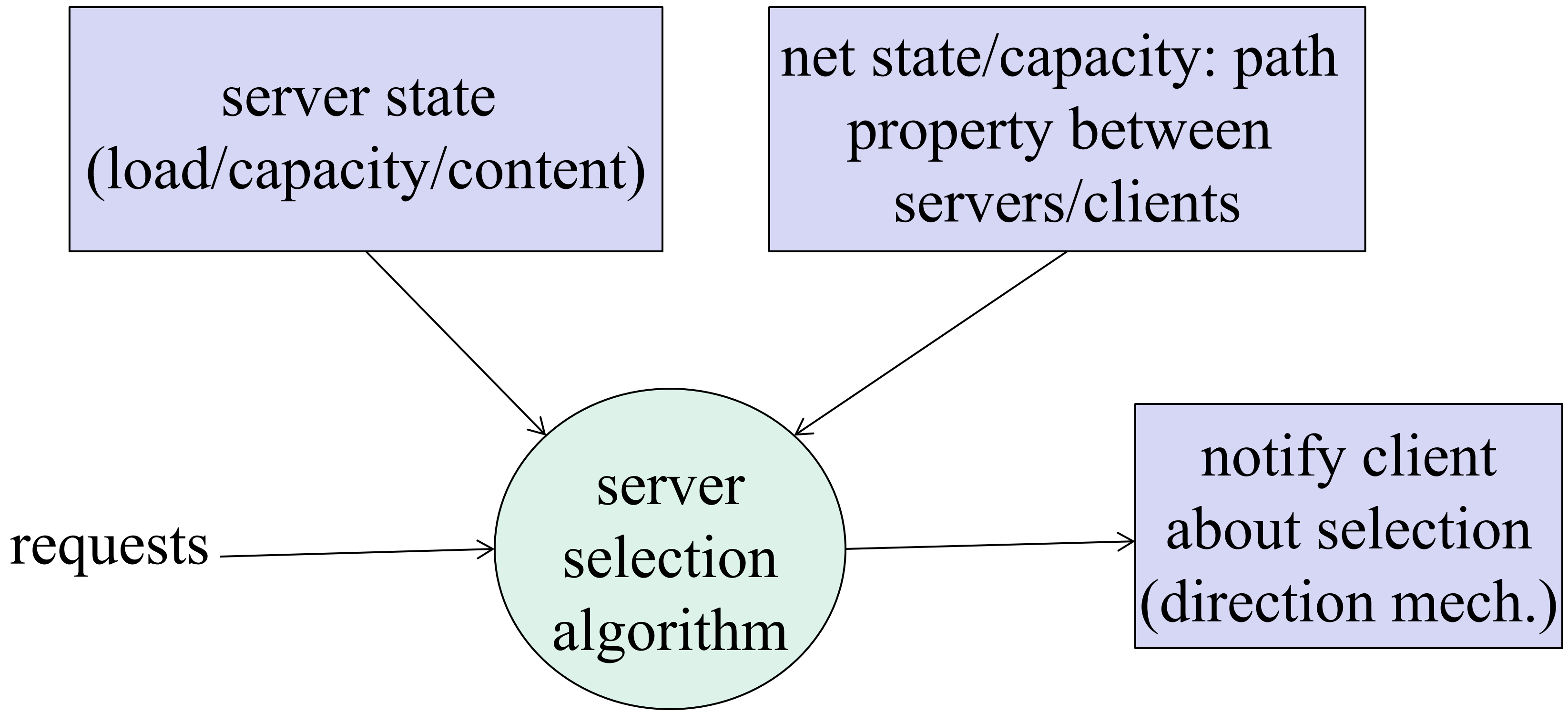
Discussion: Requirements in Designing Load-Balancing Multiple Servers

- Provide naming abstraction
- Optimize resource utilization/performance goal
- Achieve fault tolerance
- ...

Components of a Load-Balancing Multiple Servers System

- Service/resource discovery (static, zookeeper, etcs, consul)
- Health/state monitoring of servers/connecting networks
- Load balancing mechanisms/algorithm
  - Also called a request routing system

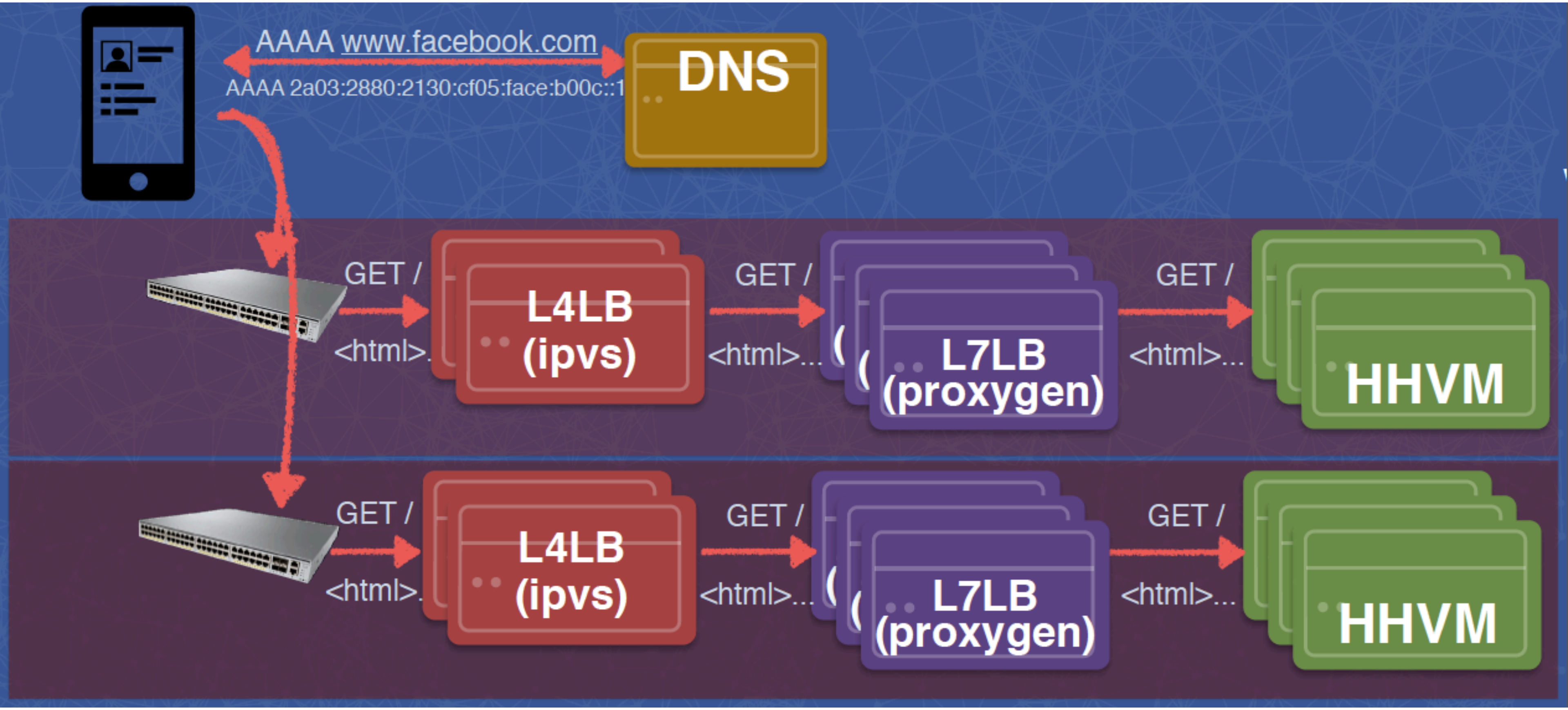
Request Routing: Basic Architecture



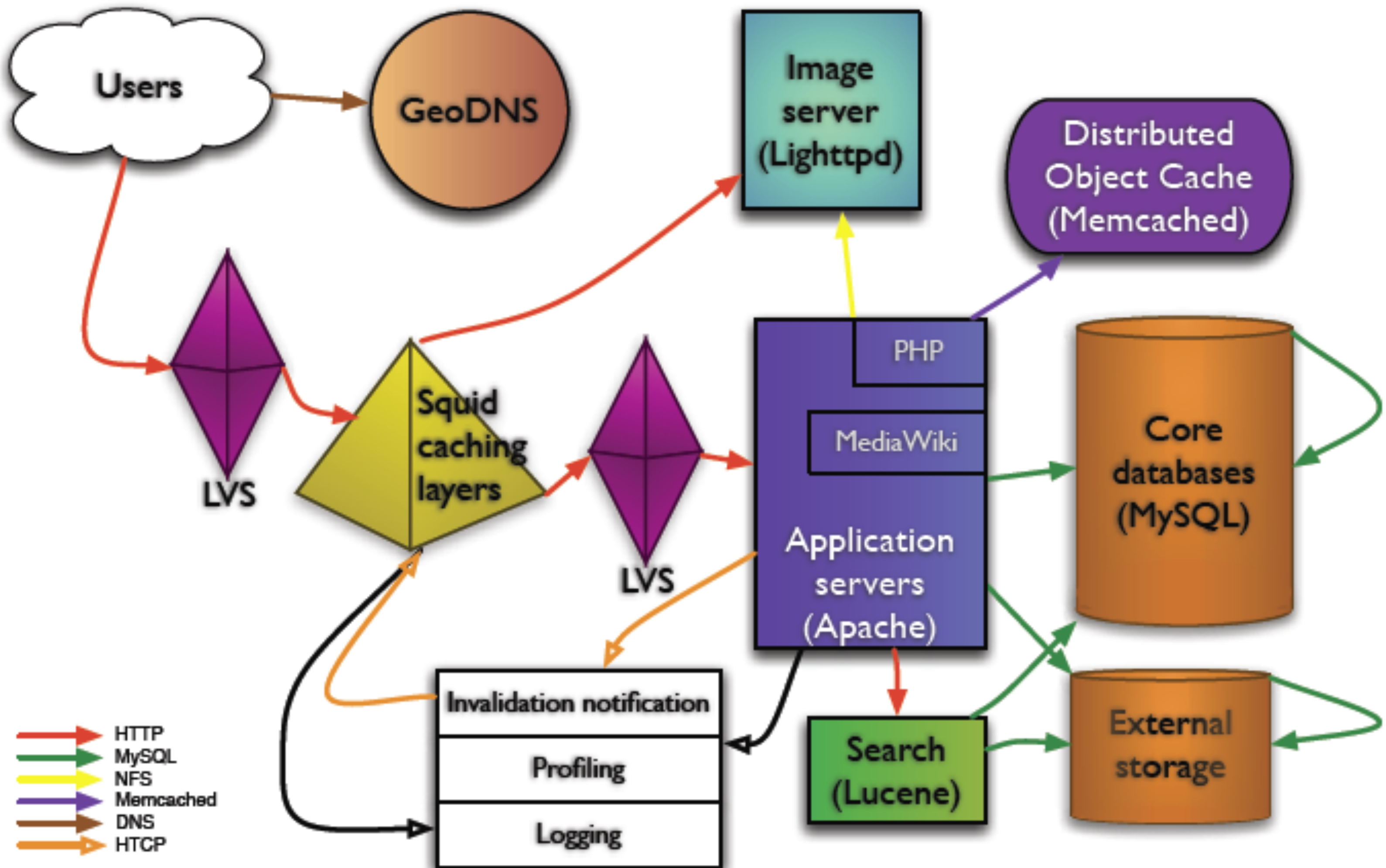
Request Routing Mechanisms

- DNS based request routing
- L4/network request routing
- L7/application request routing

Example: FB Architecture

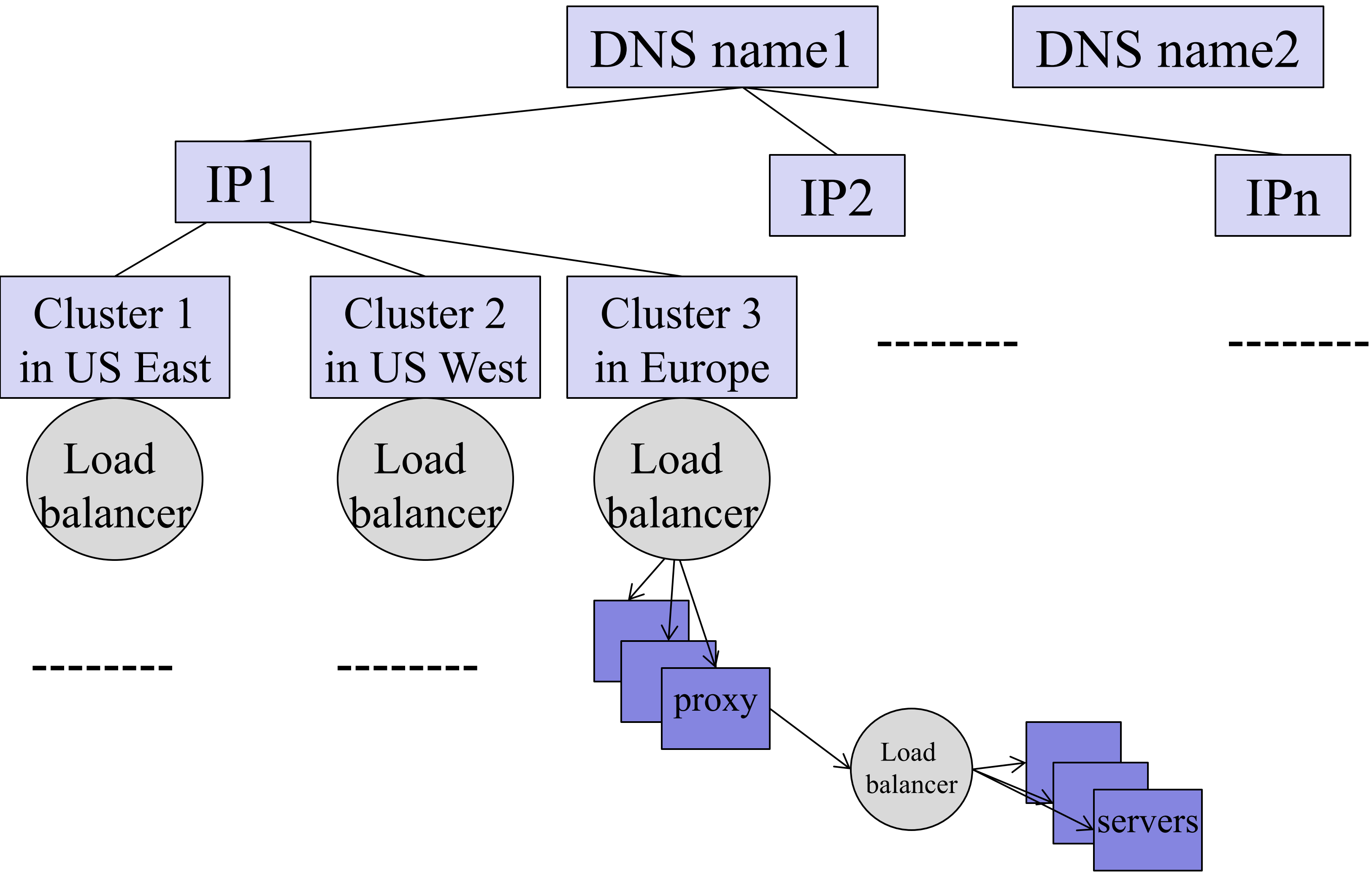


Example: Wikipedia Architecture



http://wikitech.wikimedia.org/images/8/81/Bergsma_-_Wikimedia_architecture_-_2007.pdf

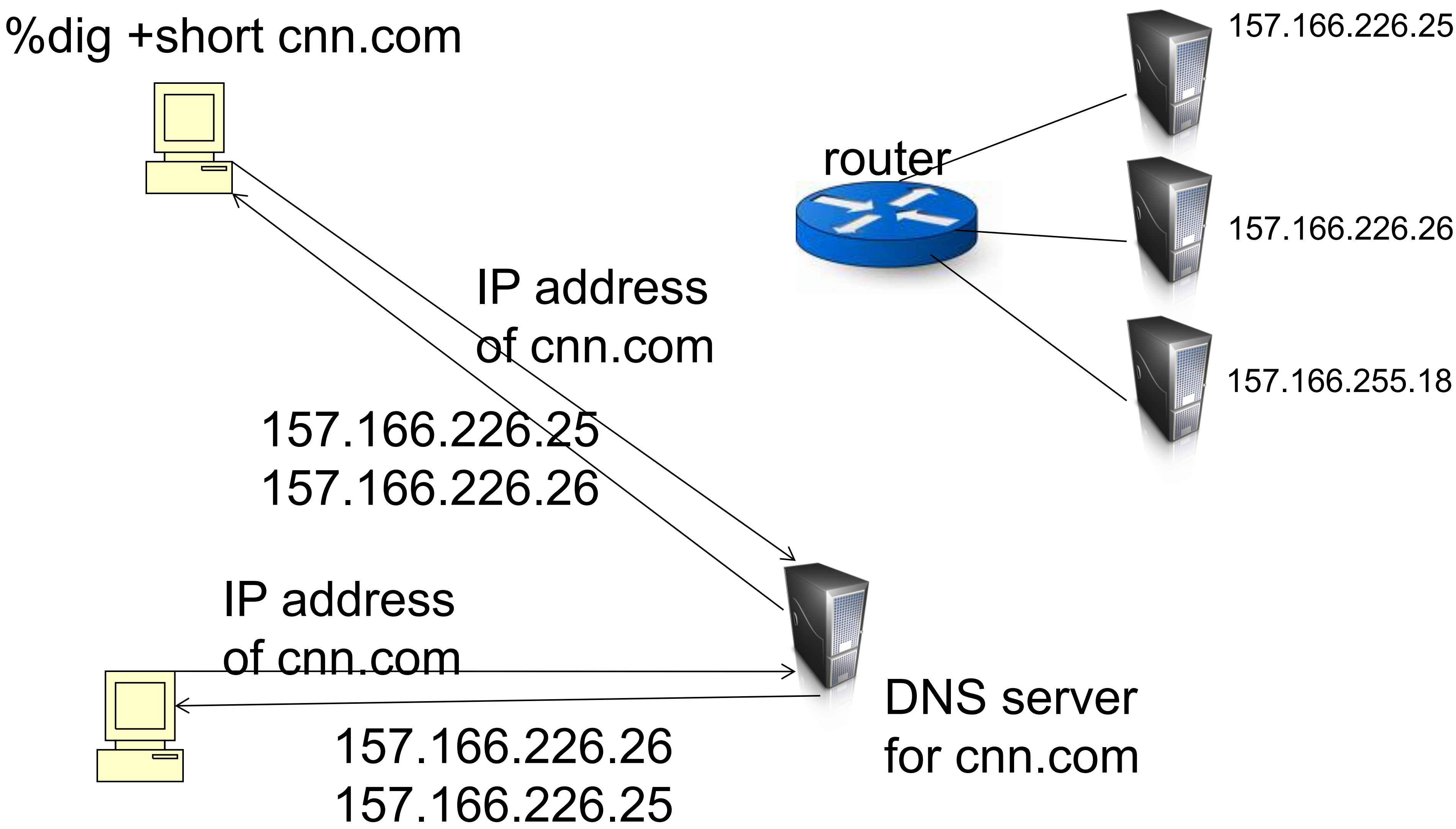
Example: Hybrid Request Routing View



Outline

- Recap
- Single network server
- Multiple network servers
  - Why multiple servers
  - Overview
  - DNS based request routing

Basic DNS Indirection and Rotation



CDN Using DNS (Akamai Architecture as an Example)

- Content publisher (e.g., cnn)
  - provides base HTML documents
  - runs origin server(s); but delegates heavy-weight content (e.g., images) to CDN
- Akamai runs
  - (~240,000) edge servers in 130 countries within 1700 networks
    - Claims 85% Internet users are within a single "network hop" of an Akamai CDN server.
  - customized DNS redirection servers to select edge servers based on
    - closeness to client browser, server load

Source: *https://www.akamai.com/us/en/about/facts-figures.jsp*

Linking to Akamai

- Originally, URL Akamaization of embedded content: e.g.,

  <IMG SRC= http://www.provider.com/image.gif >
  changed to

  <IMG SRC = http://a661. g.akamai.net/hash/image.gif>

  Note that this DNS redirection unit is per customer, not individual files.

- URL Akamaization is becoming obsolete and supported mostly for legacy reasons

Exercise

- Check any web page of cnn and find a page with an image
- Find the URL
- Use

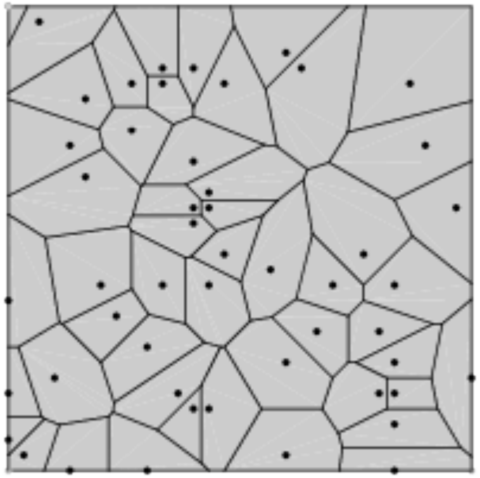
  `%dig +trace`

  to see DNS load direction

Akamai Load-Balancing DNS Name

- Akamai
  - e2466.dscg.akamaiedge.net (why two levels in the name?)

Two-Level Direction



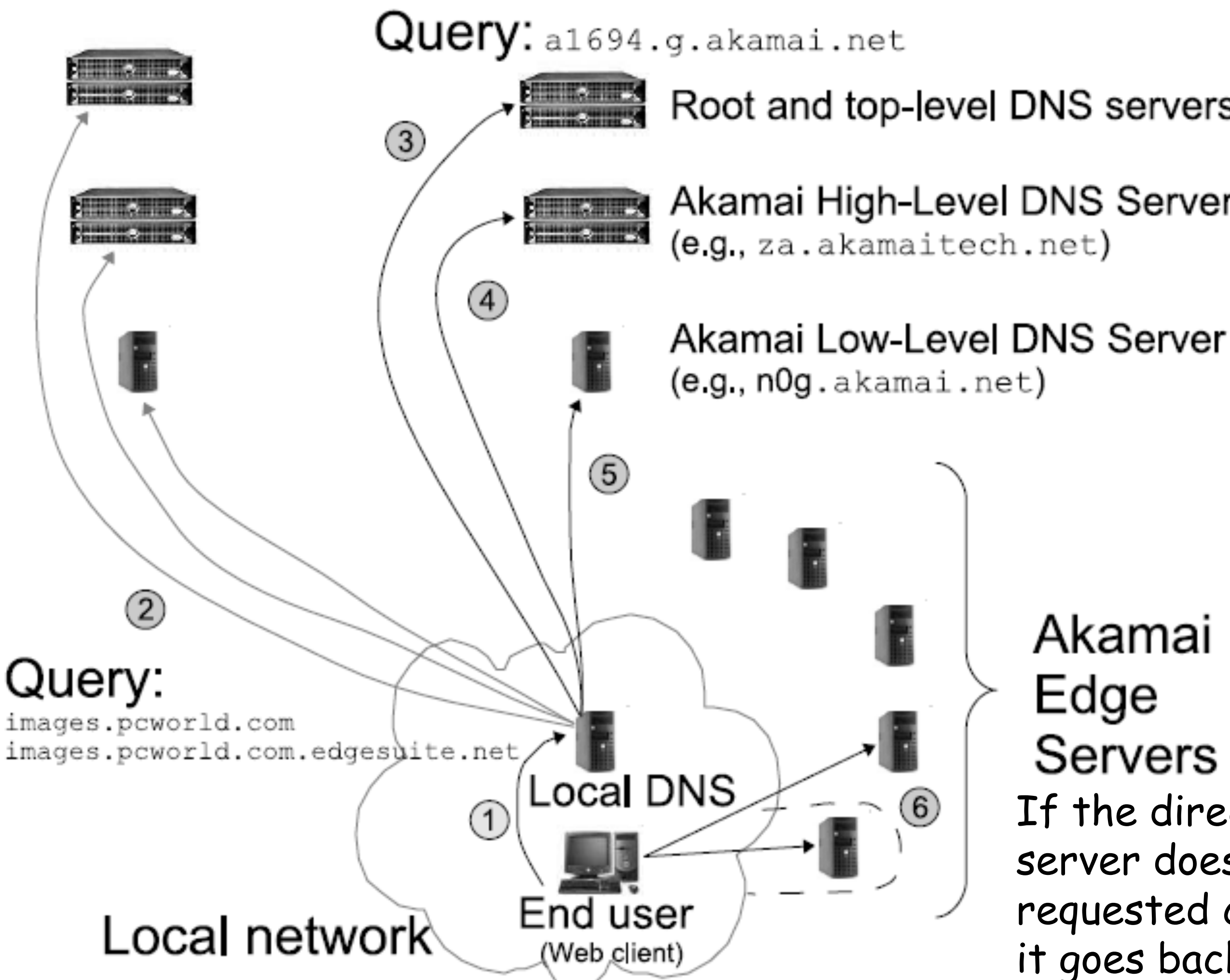
- high-level DNS determines proximity, directs to low-level DNS;

- low-level DNS: each manages a close-by cluster of servers

With query dscg.akamaiedge.net and client IP, directs to region (low-level)

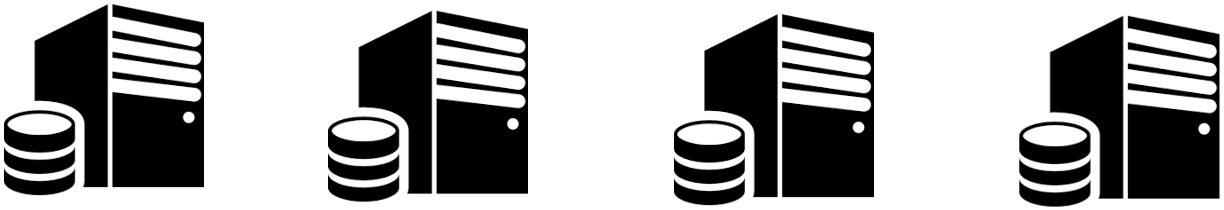
With query e2466.dscg.akamaiedge.net and client IP, directs to specific server

Akamai Load Direction



If the directed edge server does not have requested content, it goes back to the original server (source).

Local DNS Alg: Considerations



- Load on each edge server does not exceed its server capacity
- Maximize caching state of each server
- Minimize the number of busy servers

Example Local DNS Alg:

- Details of Akamai algorithms are proprietary
- A Bin-Packing algorithm (column 12 of Akamai Patent) every T second
  - Compute the load to each publisher k (called serial number)
  - (estimate the number of needed servers)
  - Sort the publishers from increasing load
  - For each publisher, compute a sequence of random numbers using a hash function
  - Assign the publisher to the first server that does not overload