Network Applications: Network App Programming: HTTP/1.1/2; High-performance Server Design

Y. Richard Yang

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

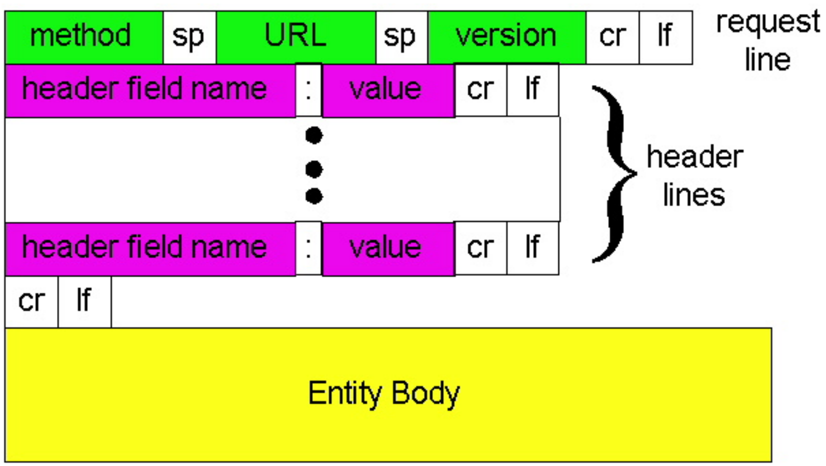
2/15/2016

Outline

- Admin and recap
- HTTP "acceleration"
- Network server design

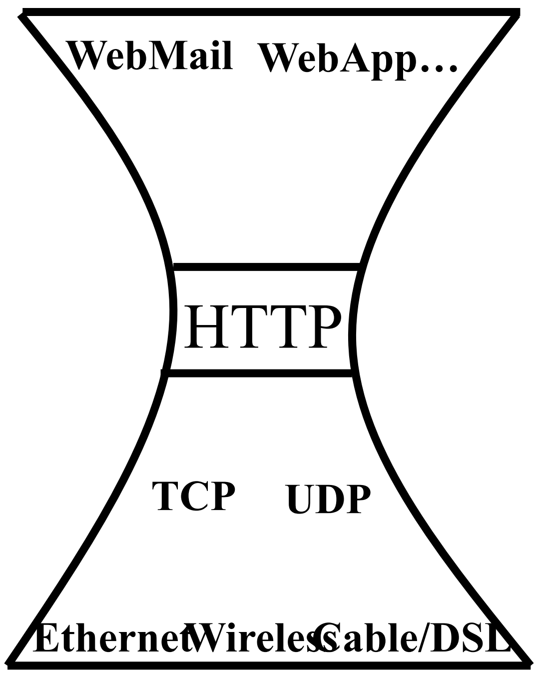
Recap: HTTP

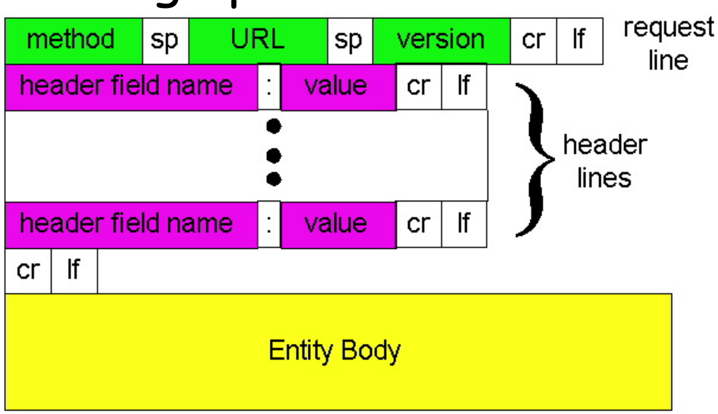
- C-S app serving Web pages
  - message format
    - request/response line, header lines, entity body
    - simple methods, rich headers
  - message flow
    - stateless server, thus states such as cookie and authentication are needed in each message



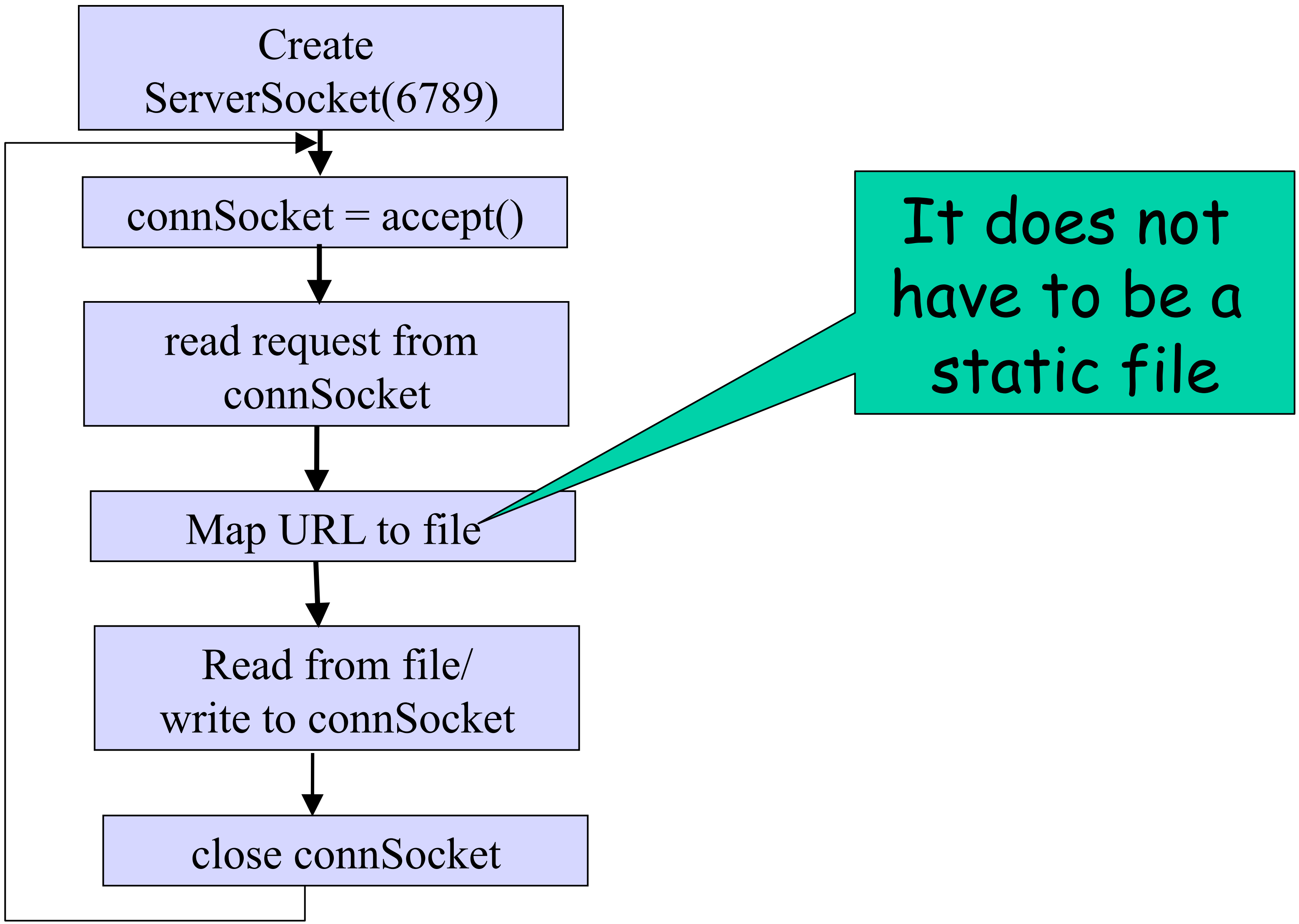
Recap: HTTP

- Wide use of HTTP for Web applications
- Example: RESTful API
  - RESTful design
    - http://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm
    - http://docs.oracle.com/javaee/6/tutorial/doc/giepu.html



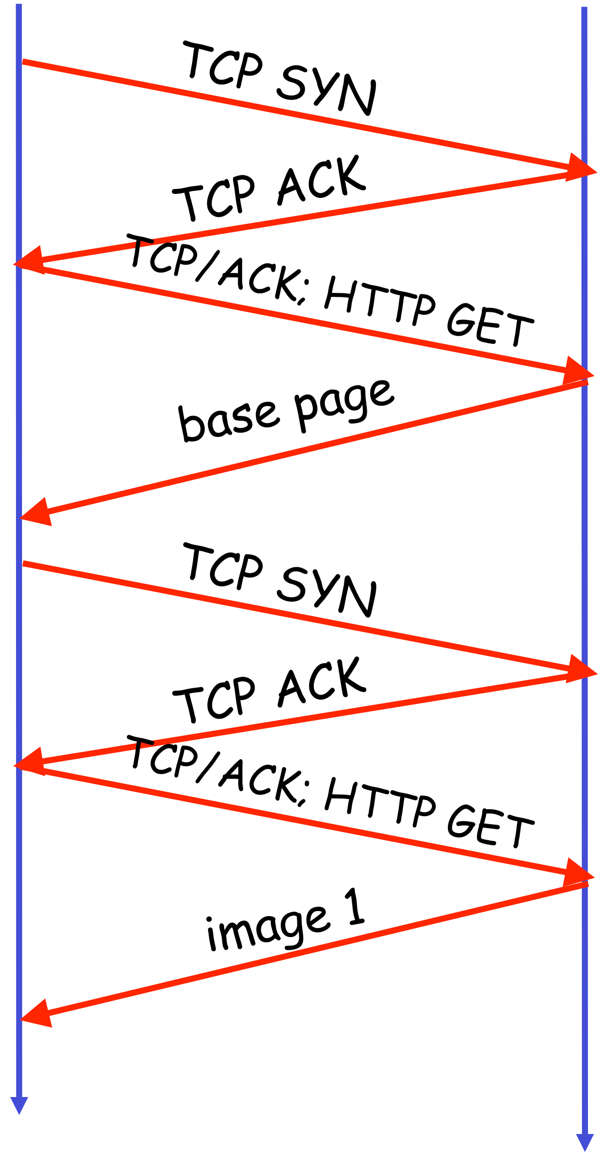


Recap: Basic HTTP/1.0 Server



Recap: Protocol Flow of Basic HTTP/1.0

- >= 2 RTTs per object:
  - TCP handshake --- 1 RTT
  - client request and server responds --- at least 1 RTT (if object can be contained in one packet)



Outline

- Admin and recap
- HTTP "acceleration"

Substantial Efforts to Speedup Basic HTTP/1.0

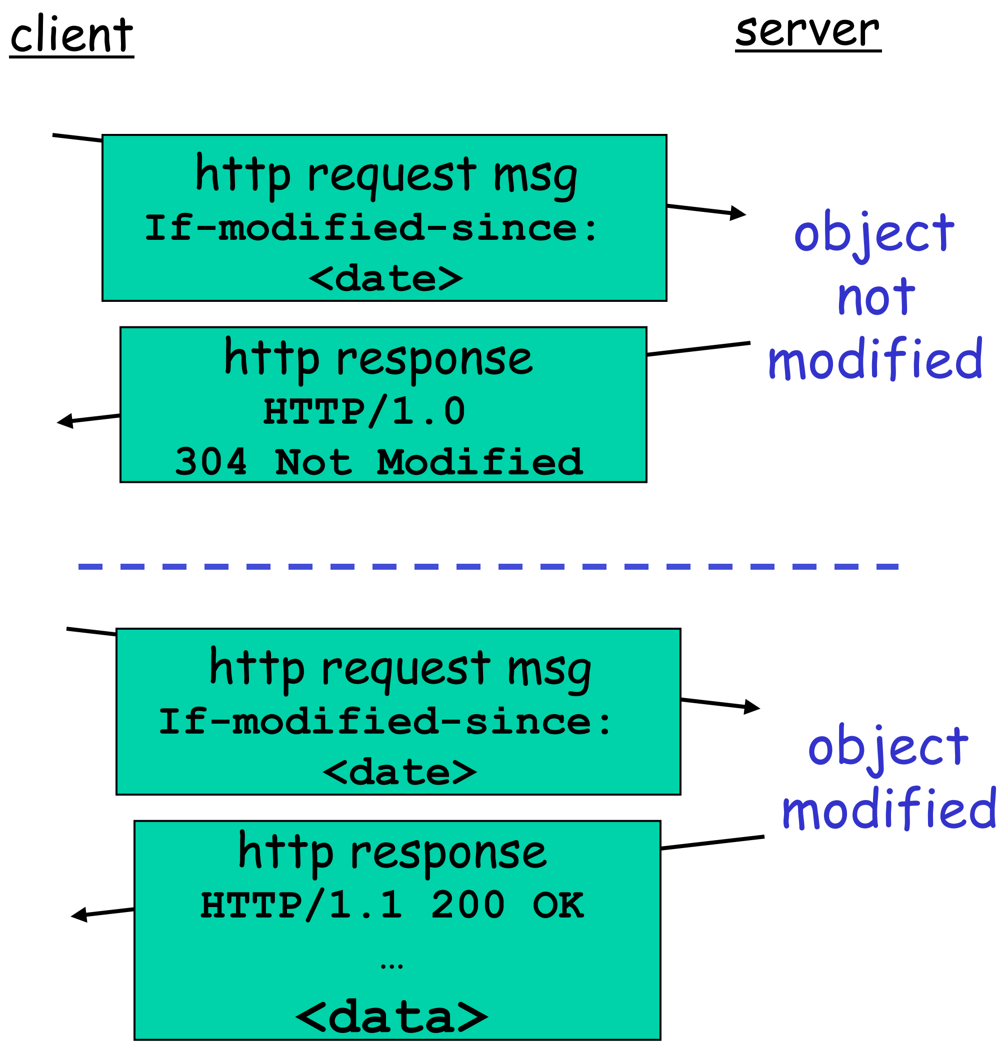
- Reduce the number of objects fetched [Browser cache]
- Reduce data volume [Compression of data]
- Reduce the latency to the server to fetch the content [Proxy cache]
- Increase concurrency [Multiple TCP connections]
- Remove the extra RTTs to fetch an object [Persistent HTTP, aka HTTP/1.1]
- Asynchronous fetch (multiple streams) using a single TCP [HTTP/2]
- Server push [HTTP/2]
- Header compression [HTTP/2]

Browser Cache and Conditional GET

- Goal: don't send object if client has up-to-date stored (cached) version
- client: specify date of cached copy in http request

  If-modified-since: <date>
- server: response contains no object if cached copy up-to-date:

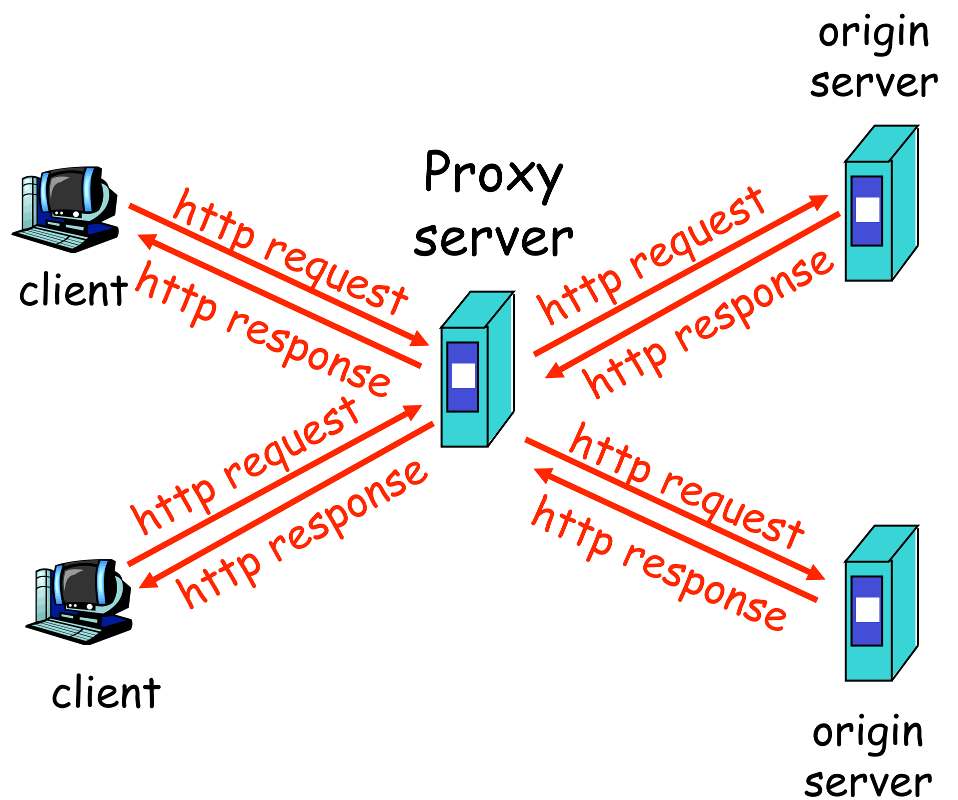
  HTTP/1.0 304 Not Modified



Web Caches (Proxy)

Goal: satisfy client request without involving origin server

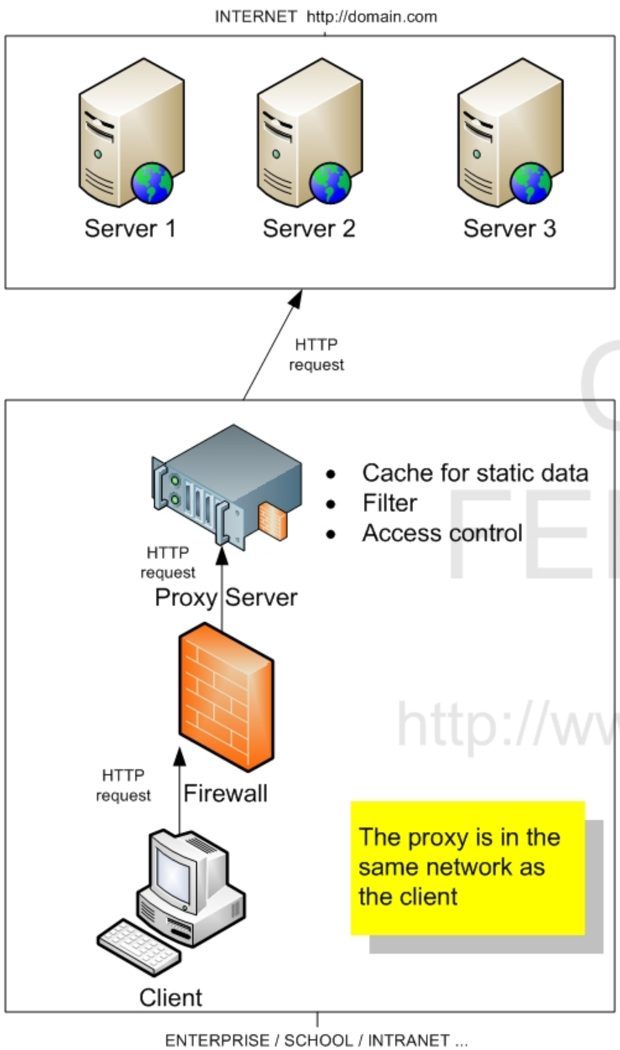
- Two types of proxies
  - Forward proxy
    - Typically in the same network as the client
  - Reverse proxy
    - Typically in the same network as the server



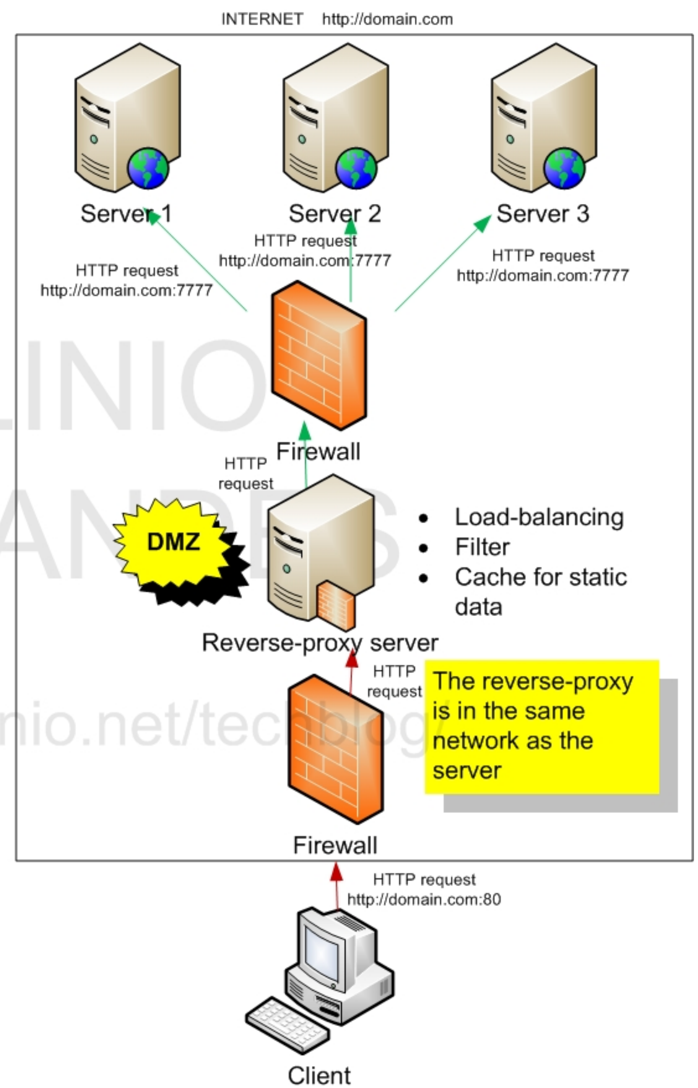
http://www.celinio.net/techblog/?p=1027

Two Types of Proxies

(FORWARD) PROXY server



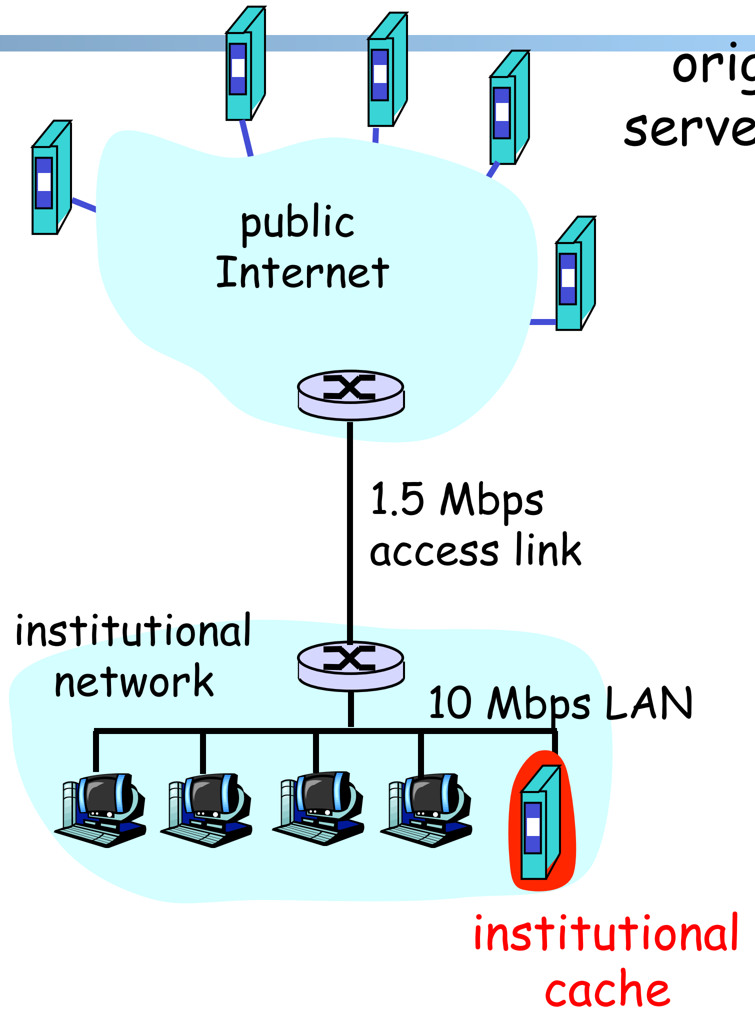
REVERSE-PROXY server



Benefits of Forward Proxy

Assume: cache is "close" to client (e.g., in same network)

- smaller response time: cache "closer" to client
- decrease traffic to distant servers
  - link out of institutional/ local ISP network often bottleneck



No Free Lunch: Problems of Web Caching

- The major issue of web caching is how to maintain consistency
- Two ways
  - pull
    - Web caches periodically pull the web server to see if a document is modified
  - push
    - whenever a server gives a copy of a web page to a web cache, they sign a lease with an expiration time; if the web page is modified before the lease, the server notifies the cache

HTTP/1.1: Persistent (keepalive/pipelining) HTTP

- On same TCP connection: server parses request, responds, parses new request, …
- Client sends requests for all referenced objects as soon as it receives base HTML
- Fewer RTTs

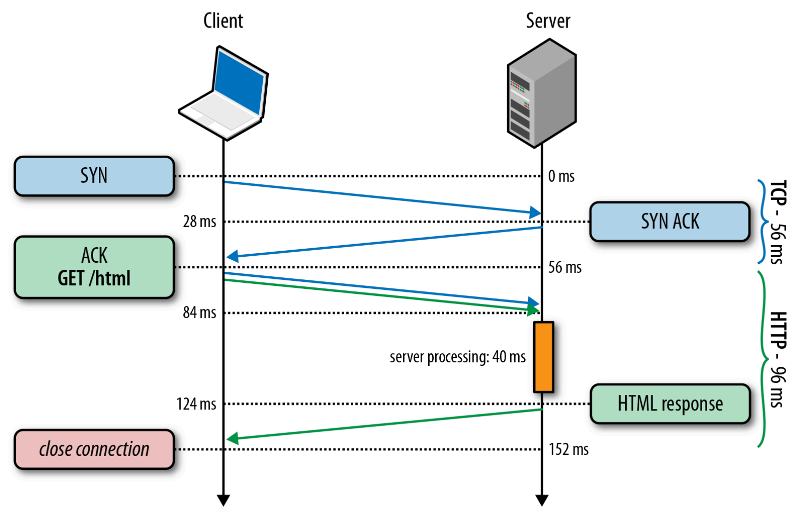
See Joshua Graessley WWDC 2012 talk: 3x within iTunes

Example

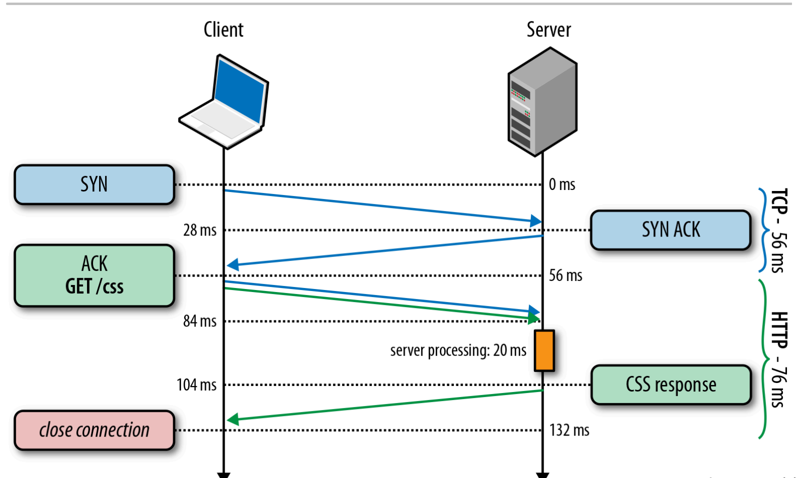
- Visit cs home page using Chrome

HTTP/1.0, Keep-Alive, Pipelining

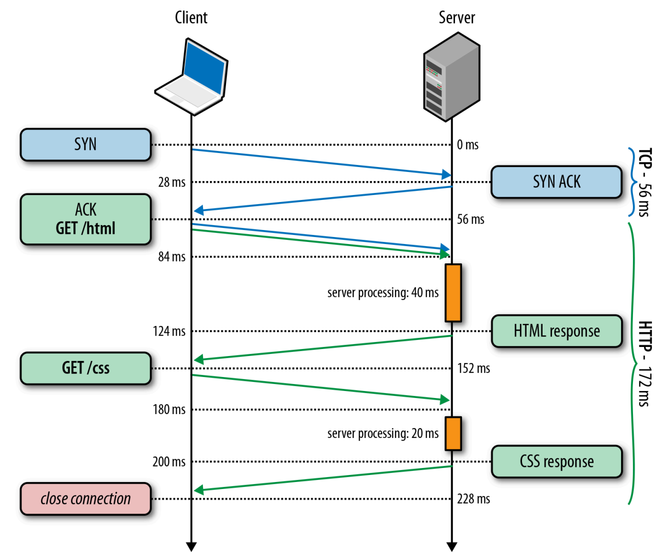
TCP connection #1, Request #1: HTML request

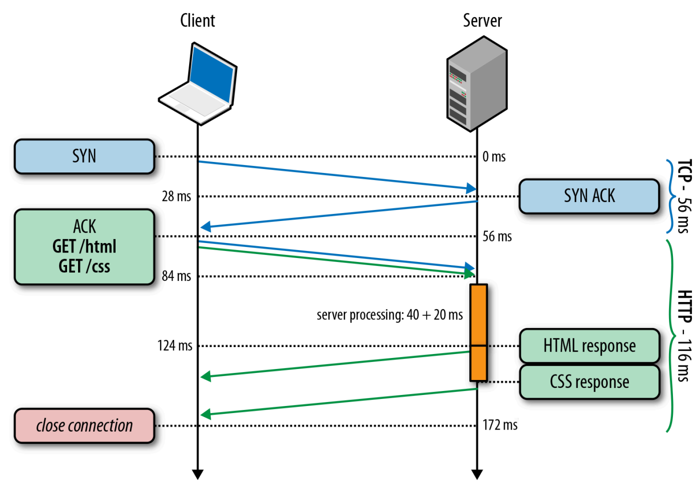


TCP connection #2, Request #2: CSS request



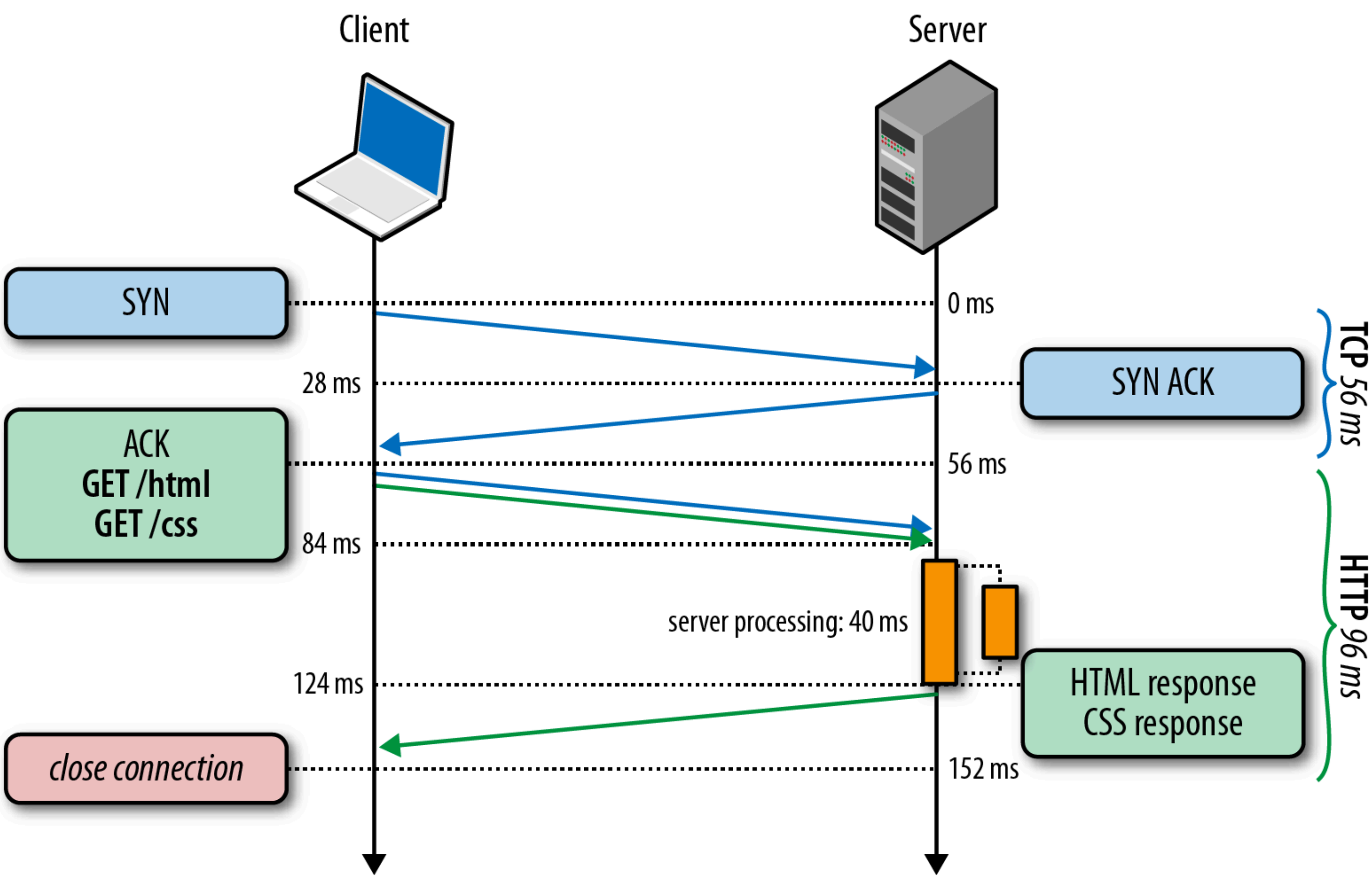
TCP connection #1, Request #1-2: HTML + CSS





Source: http://chimera.labs.oreilly.com/books/1230000000545/ch11.html

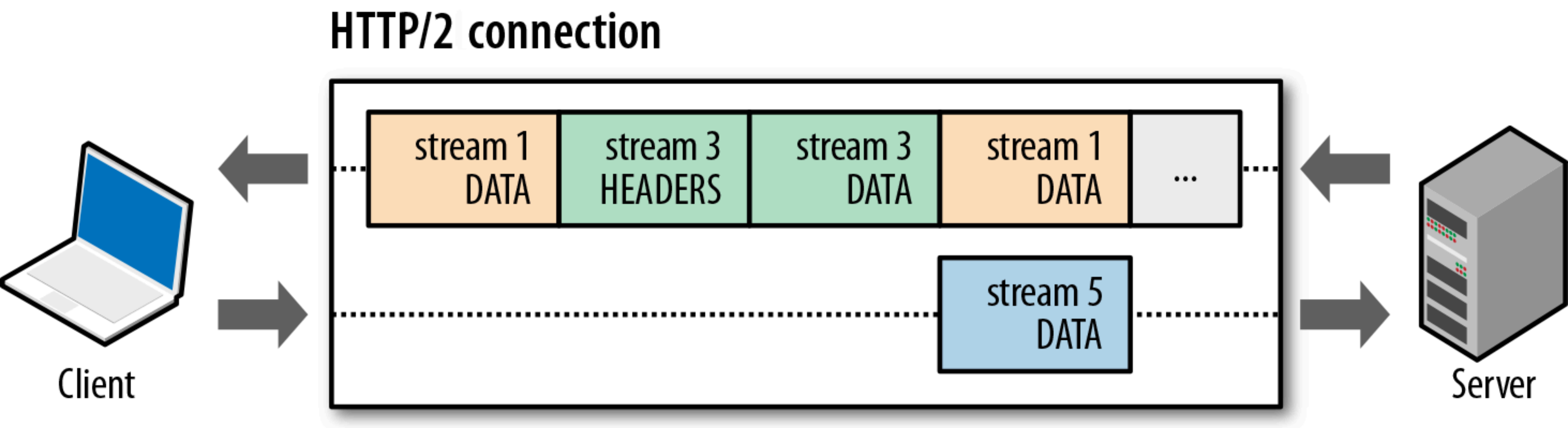
Remaining Problem of Pipelining



Head-of-line blocking

Source: *http://chimera.labs.oreilly.com/books/1230000000545/ch11.html*

HTTP/2 Multi-Streams Multiplexing

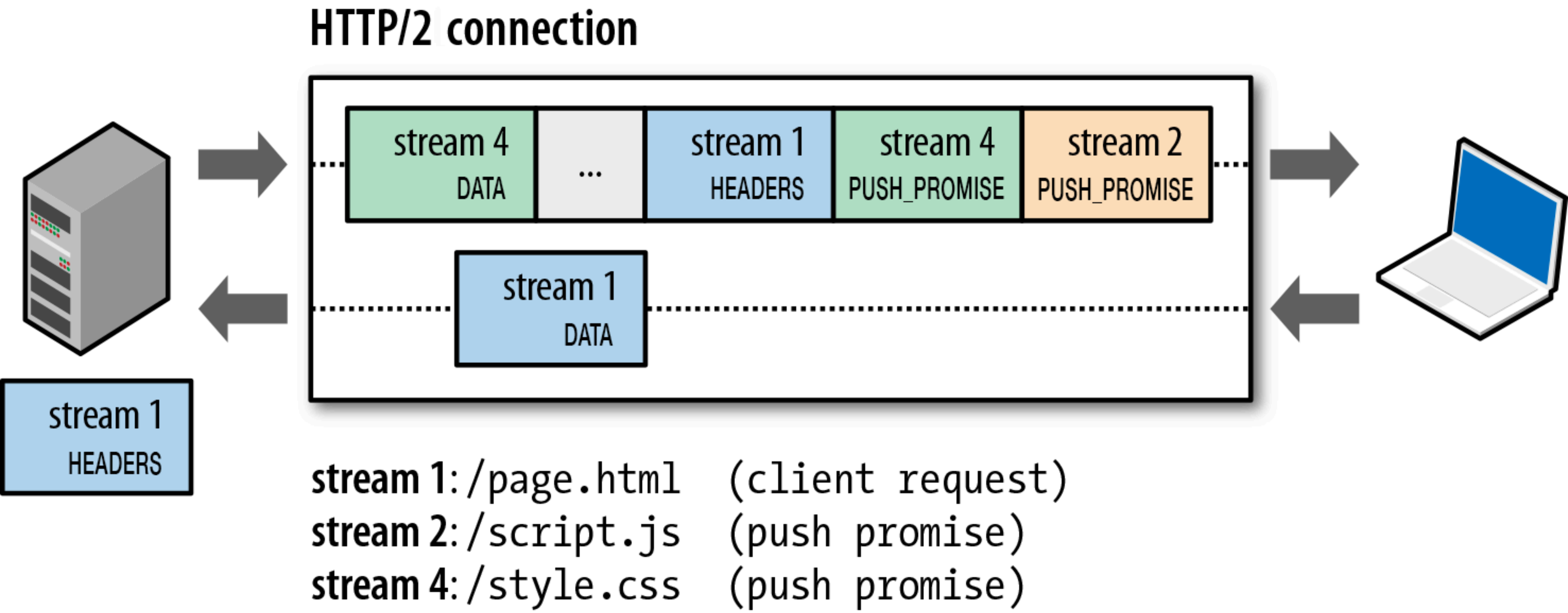


Bit	+0..7		+8..15	+16..23	+24..31
0	Length				Type
32	Flags				
40	R	Stream Identifier			
...	*Frame Payload*				

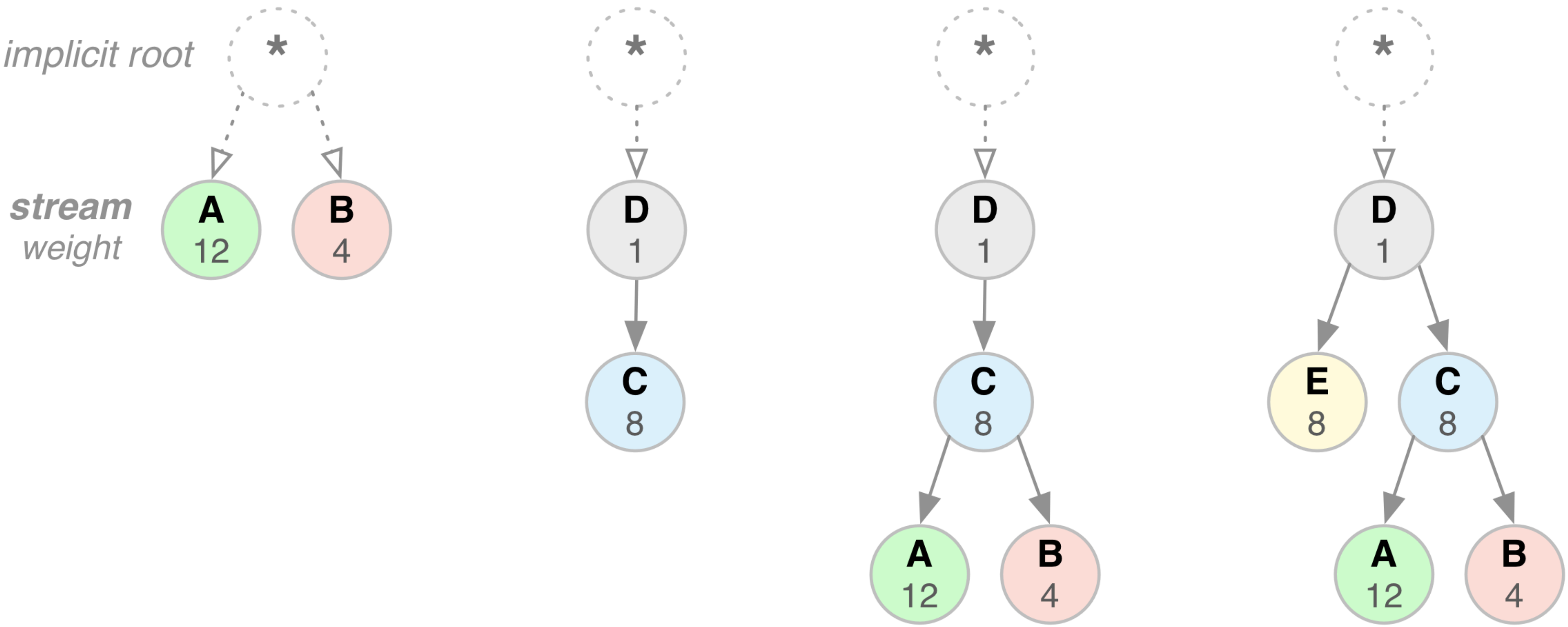
HTTP/2 Binary Framing

https://tools.ietf.org/html/rfc7540

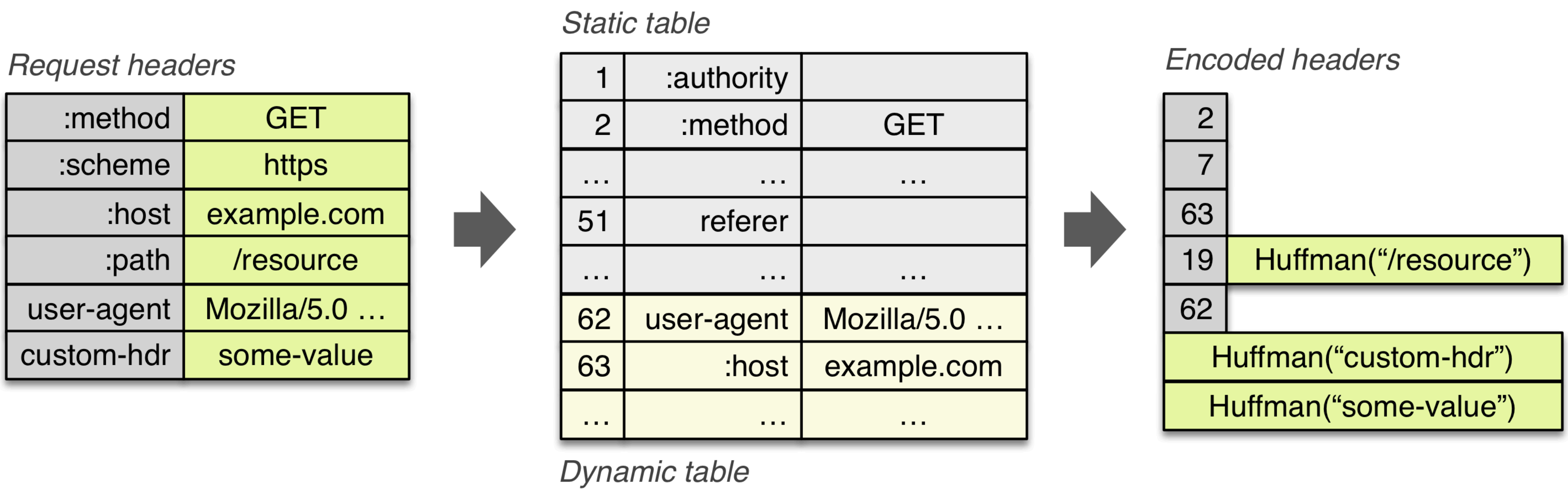
HTTP/2 Server Push



HTTP/2 Stream Dependency and Weights



HTTP/2 Header Compression



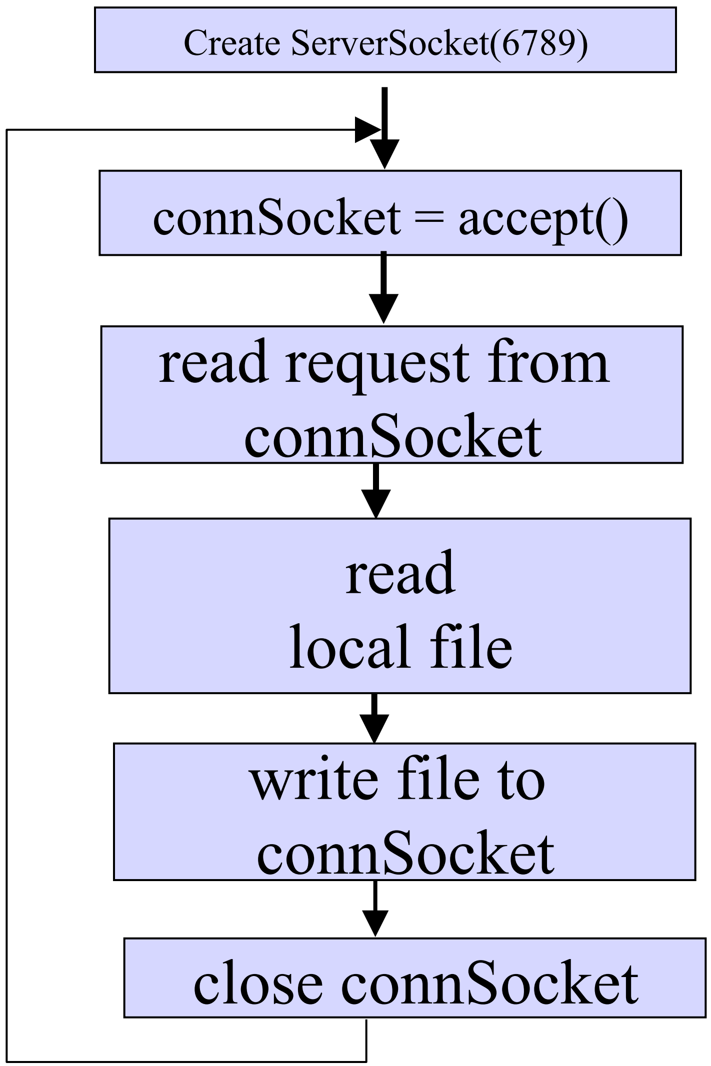
Example

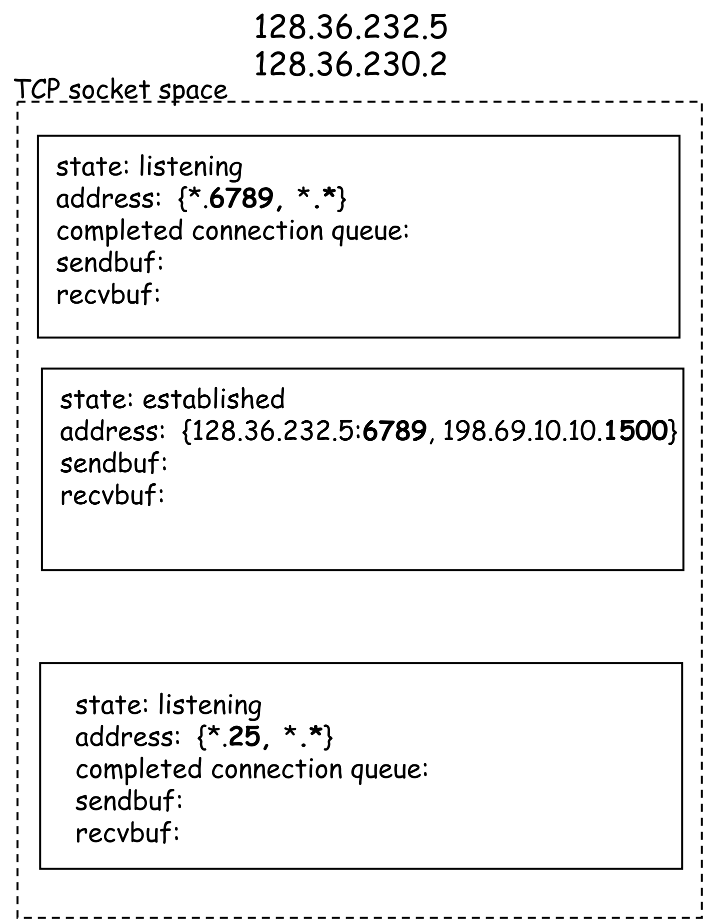
- Visit HTTP/2 pages, such as https://http2.akamai.com
- See chrome://net-internals/#http2

Outline

- Admin and recap
- HTTP "acceleration"
- Network server design

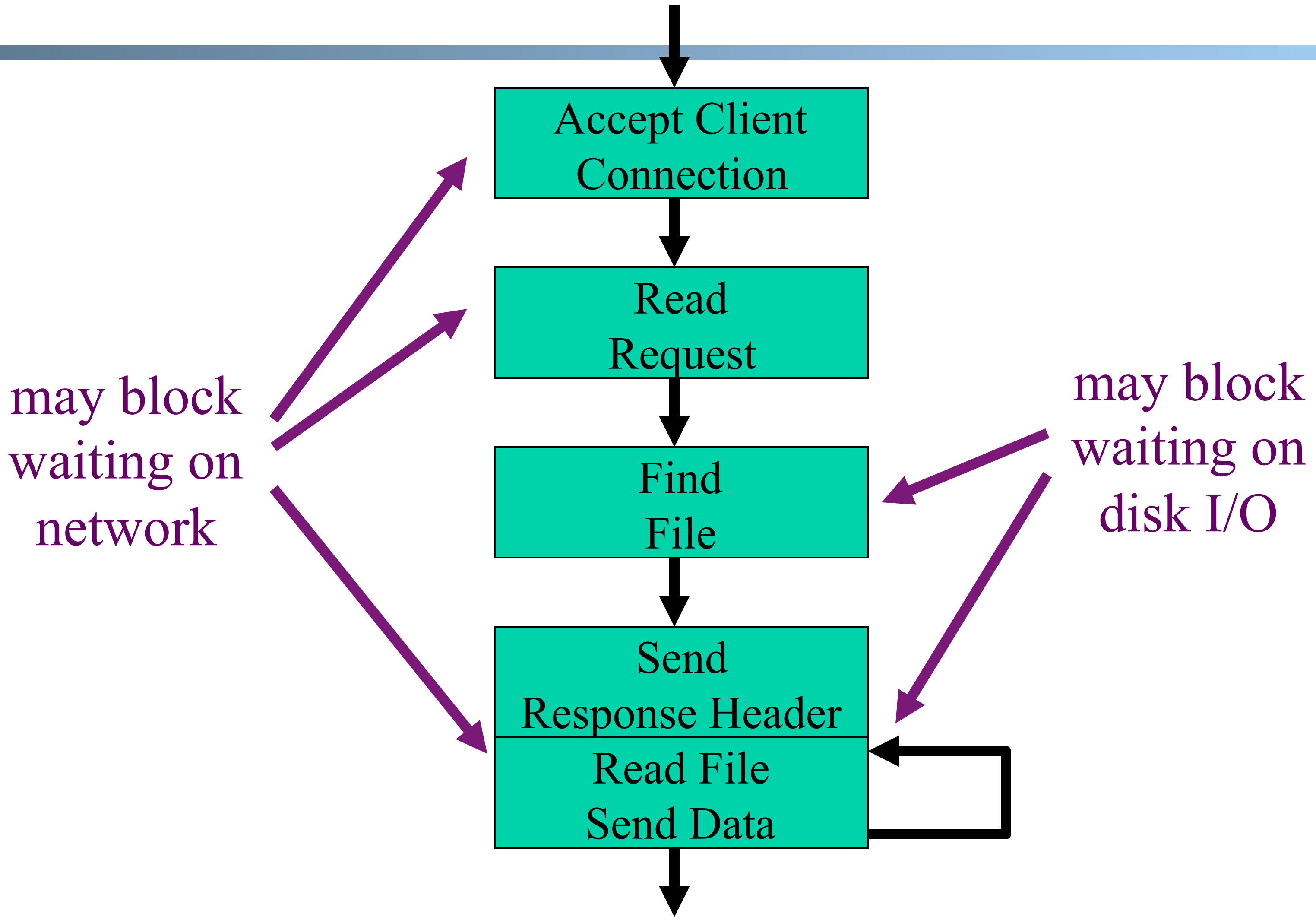
WebServer Implementation





Discussion: what does each step do and how long does it take?

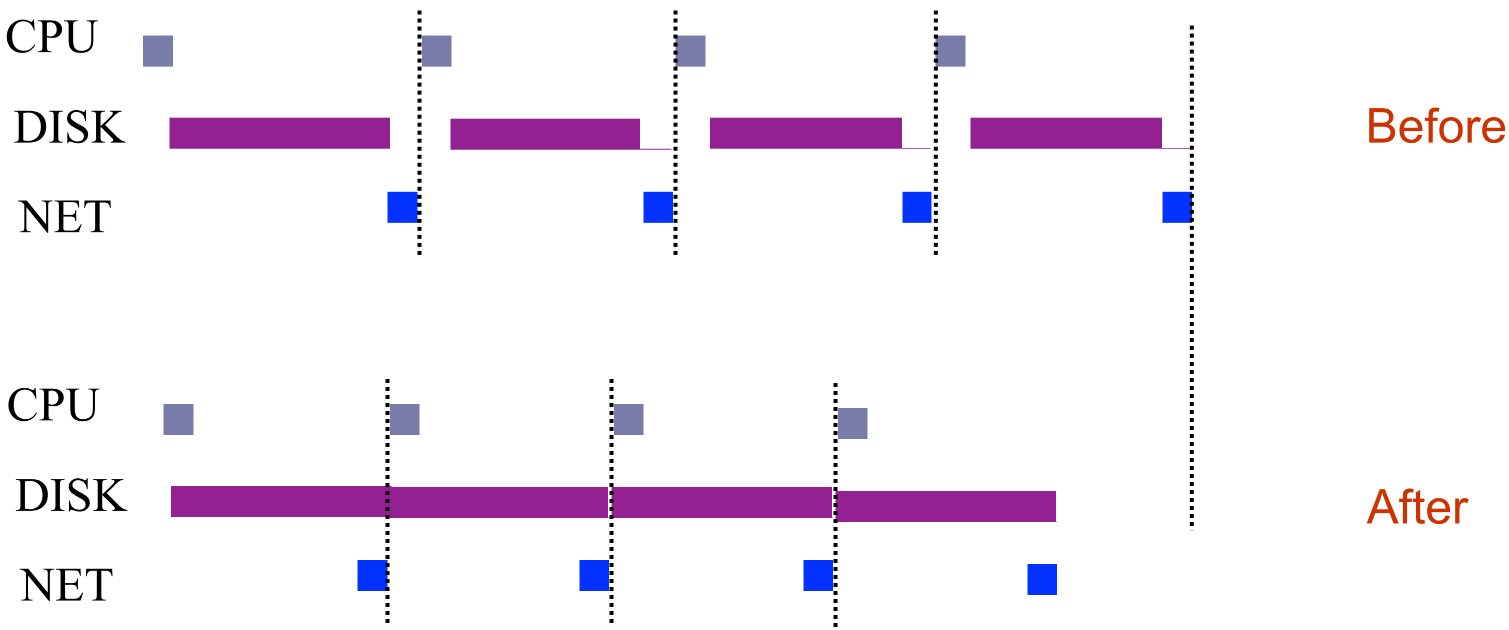
Server Processing Steps

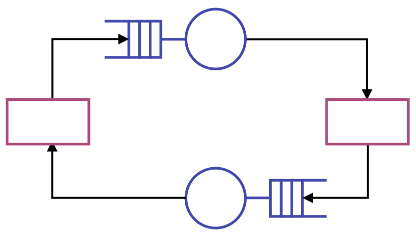


Writing High Performance Servers: Major Issues

- Many socket and IO operations can cause a process to block, e.g.,
  - `accept`: waiting for new connection;
  - `read` a socket waiting for data or close;
  - `write` a socket waiting for buffer space;
  - I/O `read/write` for disk to finish

Goal: Limited Only by the Bottleneck



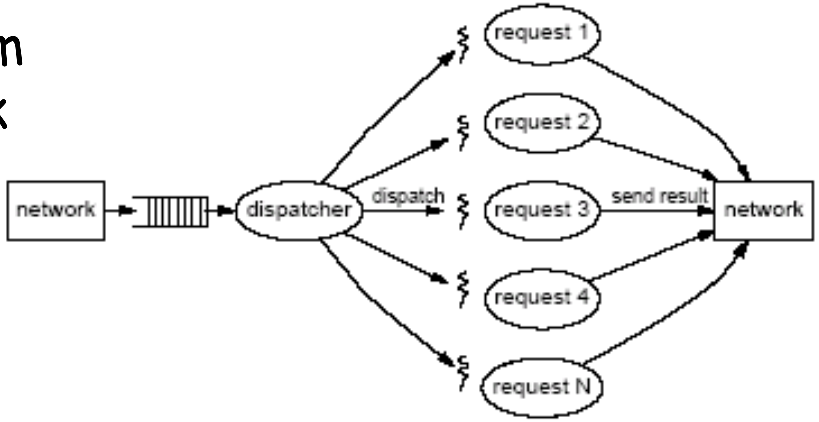


Outline

- Admin and recap
- HTTP “acceleration”
- Network server design
    - Overview
    - Multi-thread network servers

Multi-Threaded Servers

- Motivation:
  - Avoid blocking the whole program (so that we can reach bottleneck throughput)
- Idea: introduce threads
  - A thread is a sequence of instructions which may execute in parallel with other threads
  - When a blocking operation happens, only the flow (thread) performing the operation is blocked

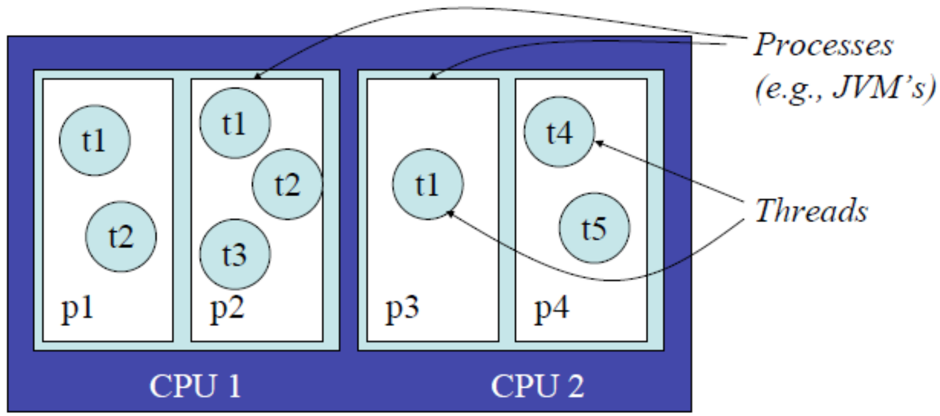


Background: Java Thread Model

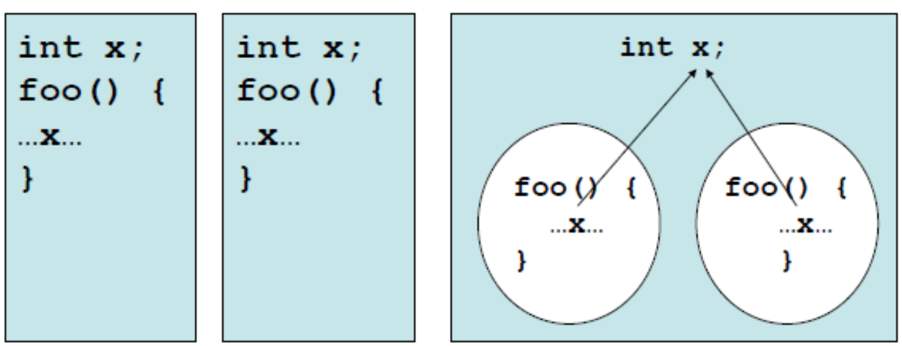
- Every Java application has at least one thread
  - The "main" thread, started by the JVM to run the application's main() method
  - Most JVM's use POSIX threads to implement Java threads

- main() can create other threads
  - Explicitly, using the Thread class
  - Implicitly, by calling libraries that create threads as a consequence (RMI, AWT/Swing, Applets, etc.)

Thread vs Process



A computer

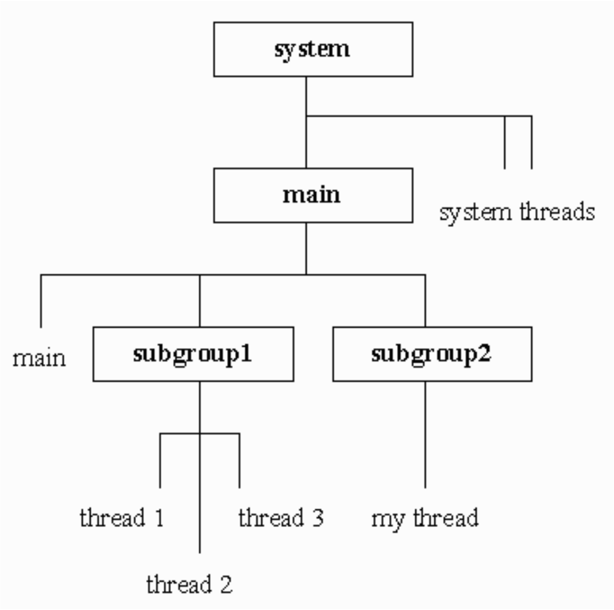


Processes do not share data

Threads share data within a process

Background: Java `Thread` Class

- Threads are organized into thread groups
  - A thread group represents a set of threads
    `activeGroupCount();`
  - A thread group can also include other thread groups to form a tree
  - Why thread group?



http://java.sun.com/javase/6/docs/api/java/lang/ThreadGroup.html

Creating Java Thread

- Two ways to implement Java thread
  1. Extend the `Thread` class
     - Overwrite the `run()` method of the `Thread` class
  2. Create a class C implementing the `Runnable` interface, and create an object of type C, then use a `Thread` object to wrap up C
- A thread starts execution after its `start()` method is called, which will start executing the thread's (or the `Runnable` object's) `run()` method
- A thread terminates when the `run()` method returns

http://java.sun.com/javase/6/docs/api/java/lang/Thread.html

Option 1: Extending Java Thread

```
class PrimeThread extends Thread {
    long minPrime;

    PrimeThread(long minPrime) {
        this.minPrime = minPrime;
    }

    public void run() {
       // compute primes larger than minPrime  . . .
    }
}

PrimeThread p = new PrimeThread(143);
p.start();
```

Option 1: Extending Java Thread

```
class RequestHandler extends Thread {
    RequestHandler(Socket connSocket) {
      // …
    }
    public void run() {
      // process request
    }
    …
}

Thread t =  new RequestHandler(connSocket);
t.start();
```

Option 2: Implement the Runnable Interface

```
class PrimeRun implements Runnable {
    long minPrime;
    PrimeRun(long minPrime) {
        this.minPrime = minPrime;
    }

    public void run() {
        // compute primes larger than minPrime  . . .
    }
}

 PrimeRun p = new PrimeRun(143);

 new Thread(p).start();
```

Option 2: Implement the Runnable Interface

```
class RequestHandler implements Runnable {
    RequestHandler(Socket connSocket) { … }
    public void run() {
      //
    }
   …
}
RequestHandler rh =  new RequestHandler(connSocket);
Thread t = new Thread(rh);
t.start();
```

Summary: Implementing Threads

```
class RequestHandler
        extends Thread {
   RequestHandler(Socket connSocket)
   {
     …
   }
   public void run() {
    // process request
   }
   …
}

Thread t =  new RequestHandler(connSocket);
t.start();
```

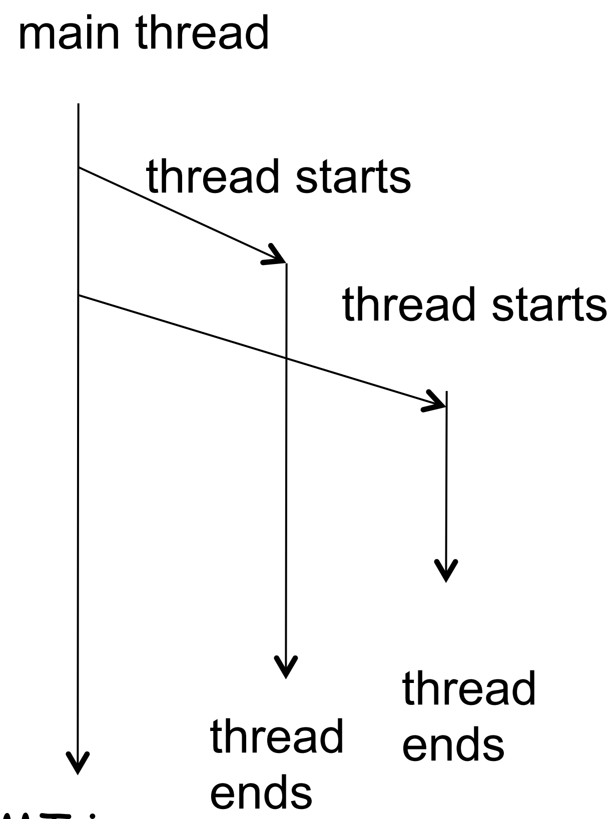
```
class RequestHandler
          implements Runnable {
   RequestHandler(Socket connSocket)
   {
     …
   }
   public void run() {
     // process request
   }
   …
}

RequestHandler rh =  new
         RequestHandler(connSocket);
Thread t = new Thread(rh);
t.start();
```

Example: a Multi-threaded TCPServer

- Turn TCPServer into a multithreaded server by creating a thread for each accepted request

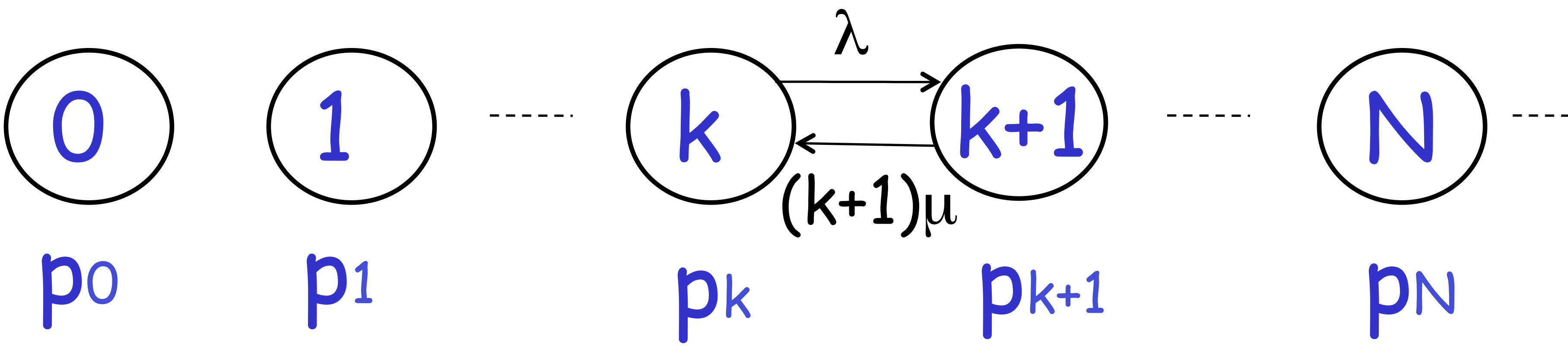
Per-Request Thread Server

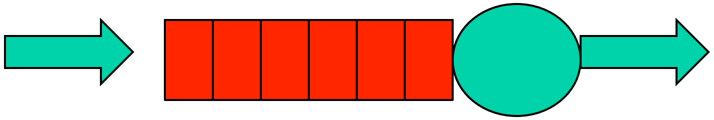
```
main() {
   ServerSocket s = new ServerSocket(port);
   while (true) {
      Socket conSocket = s.accept();
      RequestHandler rh
        =  new RequestHandler(conSocket);
      Thread t = new Thread (rh);
      t.start();
   }
```



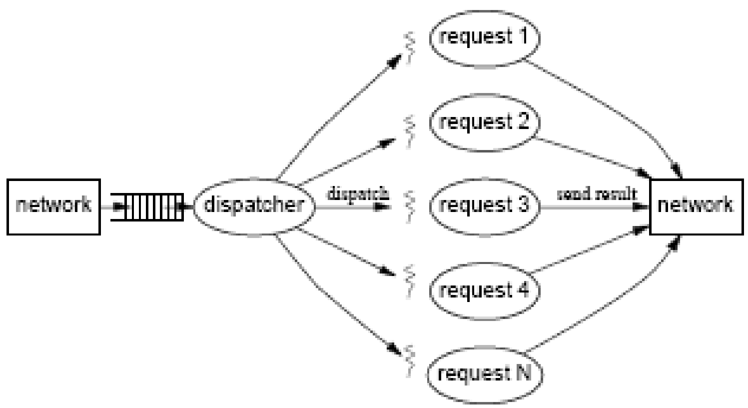
Try the per-request-thread TCP server: TCPServerMT.java

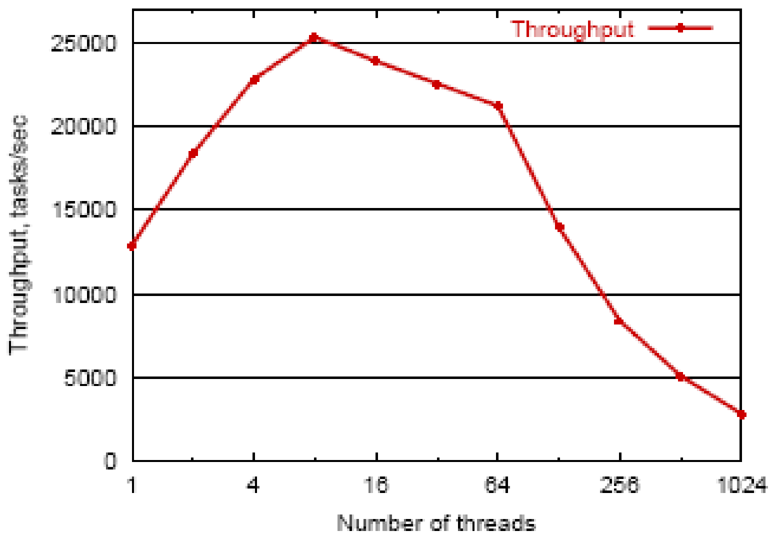
Modeling Per-Request Thread Server: Theory





Problem of Per-Request Thread: Reality





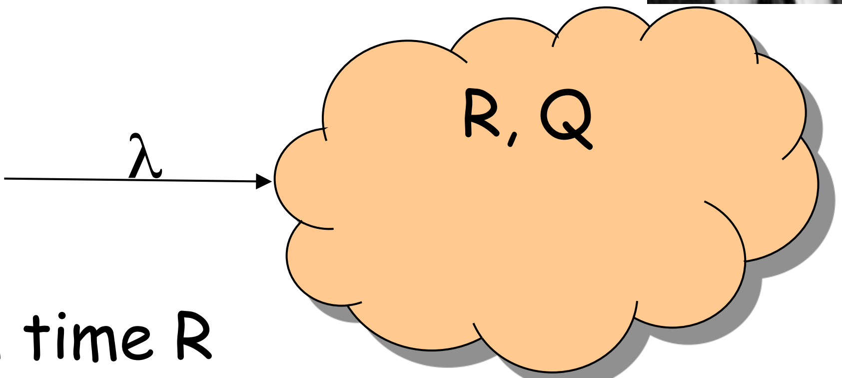
(937 MHz x86, Linux 2.2.14, each thread reading 8KB file)

- High thread creation/deletion overhead

- Too many threads → resource overuse → throughput meltdown → response time explosion
  - Q: given avg response time and connection arrival rate, how many threads active on avg?

Background: Little's Law (1961)

- For any system with no or (low) loss.
- Assume
  - mean arrival rate λ, mean time R at system, and mean number Q of requests at system



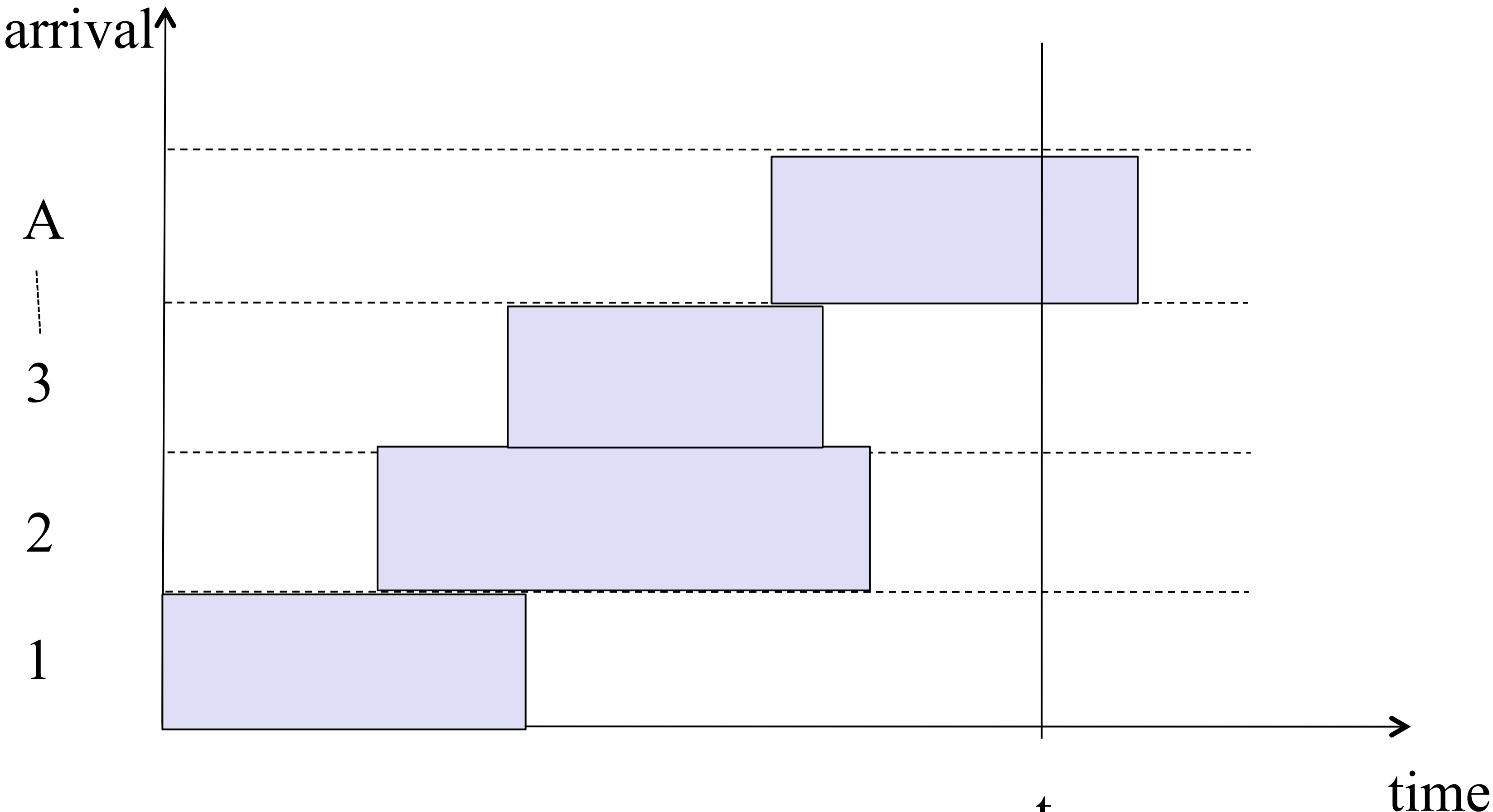
- Then relationship between Q, λ, and R:

$$Q = \lambda R$$

Example: Yale College admits 1500 students each year, and mean time a student stays is 4 years, how many students are enrolled?

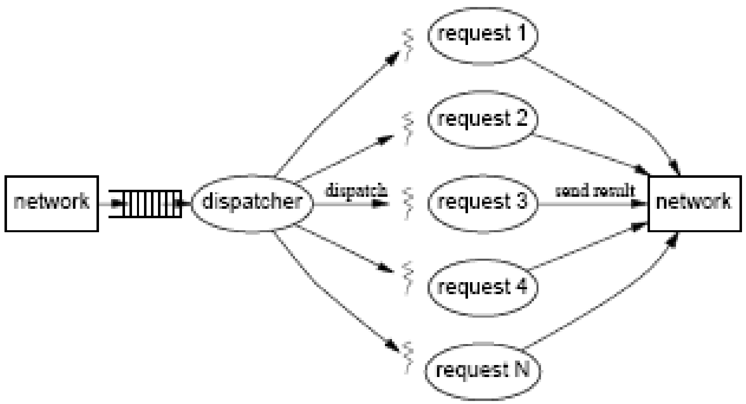
Little's Law

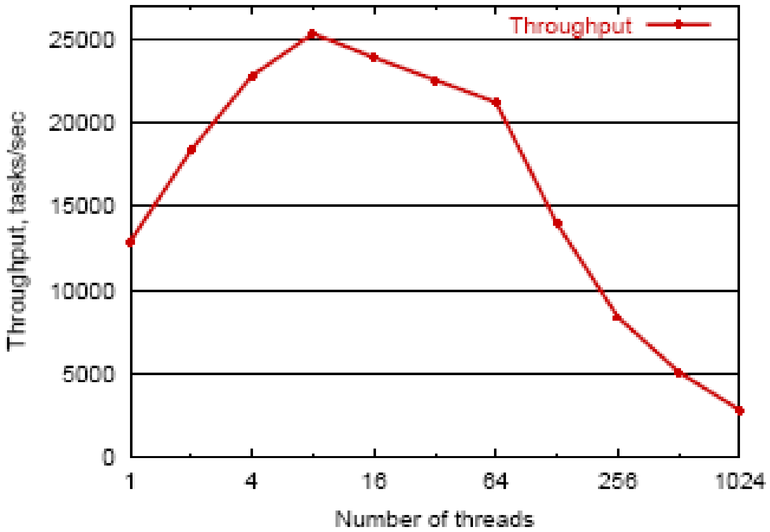
$$Q = \lambda R$$



$$\lambda = \frac{A}{t} \quad R = \frac{Area}{A} \quad Q = \frac{Area}{t}$$

Discussion: How to Address the Issue





(937 MHz x86, Linux 2.2.14, each thread reading 8KB file)