Network Applications: HTTP/2; High-Performance Server Design (Thread)

Y. Richard Yang

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

10/2/2018

Outline

- Admin and recap
- HTTP
  - Basic service
  - Basic protocol
  - Basic HTTP client/server workflow
  - Stateful interaction using stateless HTTP
  - HTTP "acceleration"
- High-performance network server design
  - Overview
  - Threaded servers

Admin

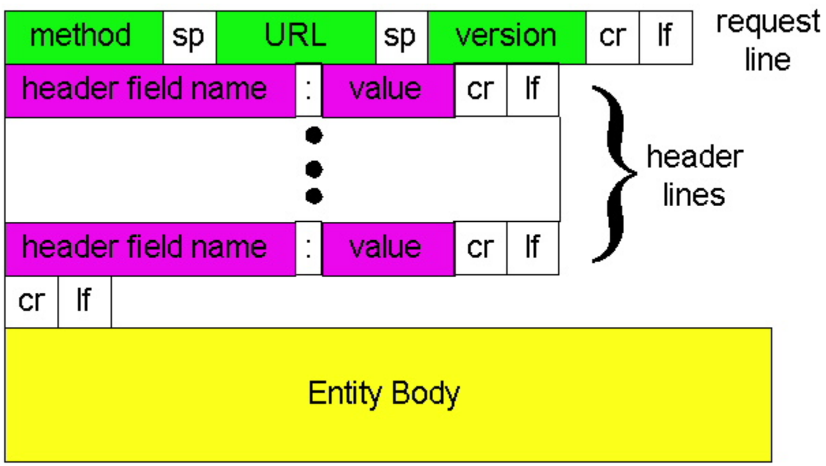
- Assignment Two status

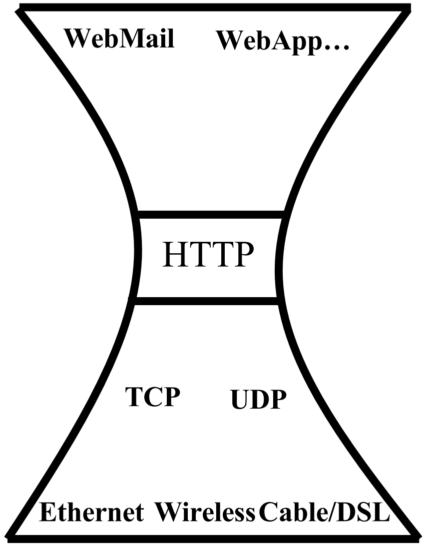
- Assignment Three (HTTP server) Assignment Part 1 to be posted on Wednesday

- Exam 1 date?

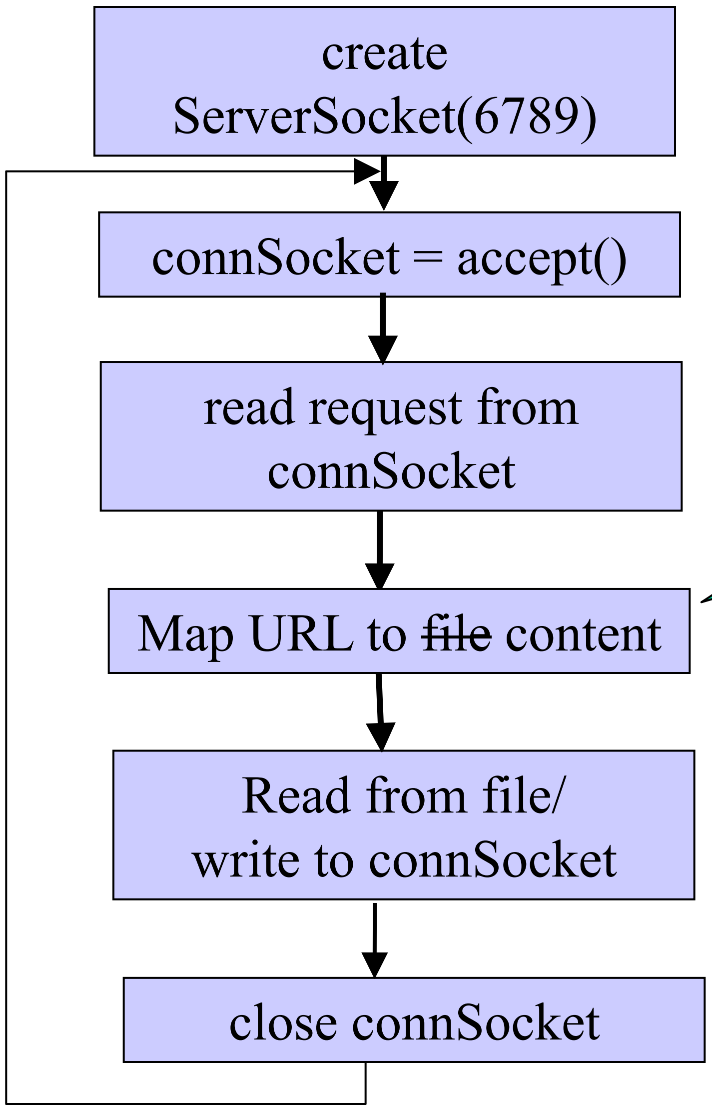
Recap So Far: HTTP

- C-S app serving hypertext
  - message format
    - simple methods, rich headers (e.g., content negotiation), entity body
  - message flow
    - stateless server, thus states such as cookie and authentication are needed in each message
- Wide use of HTTP for Web applications
  - Example: RESTful API http://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm





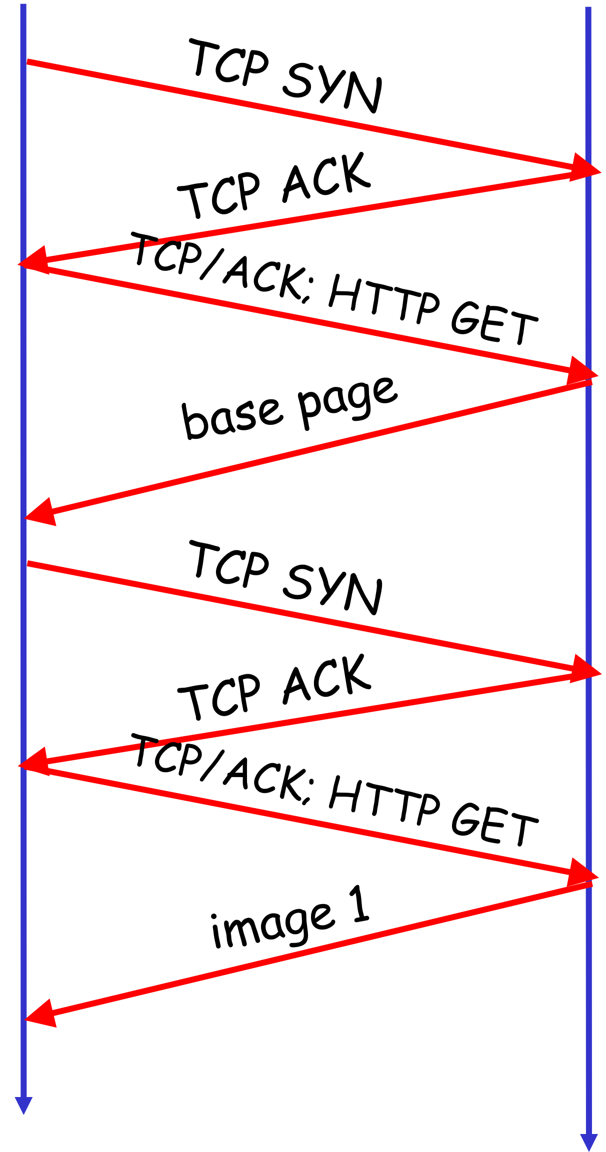
Recap: Basic HTTP Server Workflow



A major power of Web/HTTP is the URI abstraction

Recap: Latency 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)
- Assume
  - RTT ~ 40 ms, 75 objects =>
  - 6 sec load time



Recap: Substantial Efforts to Speedup Basic HTTP/1.0

- Reduce the number of objects fetched [Browser cache]

- Reduce data volume [Compression of data]
- Header compression [HTTP/2]

- Reduce the latency to the server to fetch the content [Proxy cache]
- Remove the extra RTTs to fetch an object [Persistent HTTP, aka HTTP/1.1]

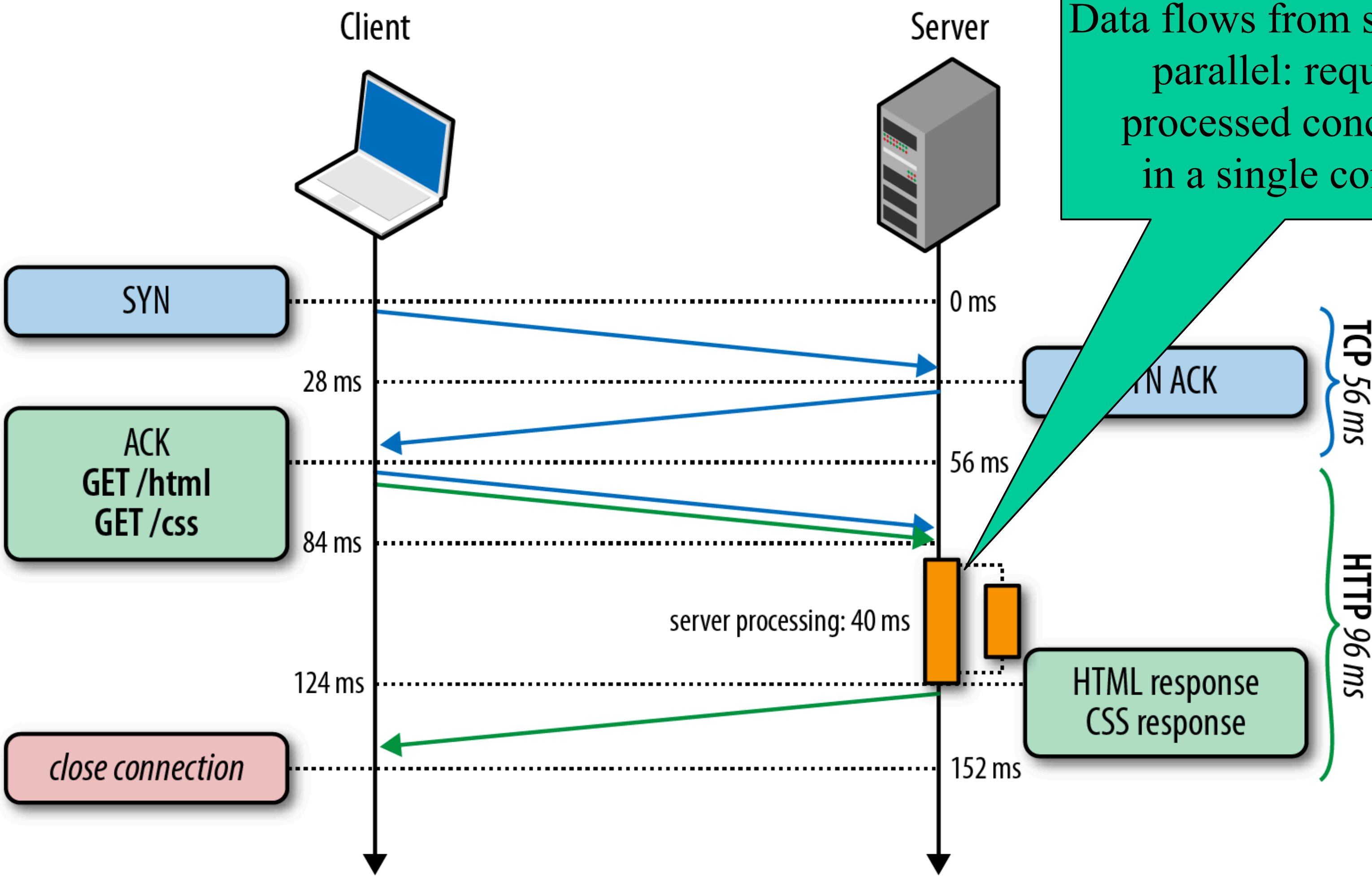
- Increase TCP concurrency [Multiple TCP connections]
- Asynchronous fetch (multiple streams) using a single TCP [HTTP/2]

- Server push [HTTP/2]

Office Exercise

- telnet cpsc.yale.edu and see the headers to see many things going on behind the scene
  - `telnet cpsc.yale.edu 80`
    `GET / HTTP/1.0`

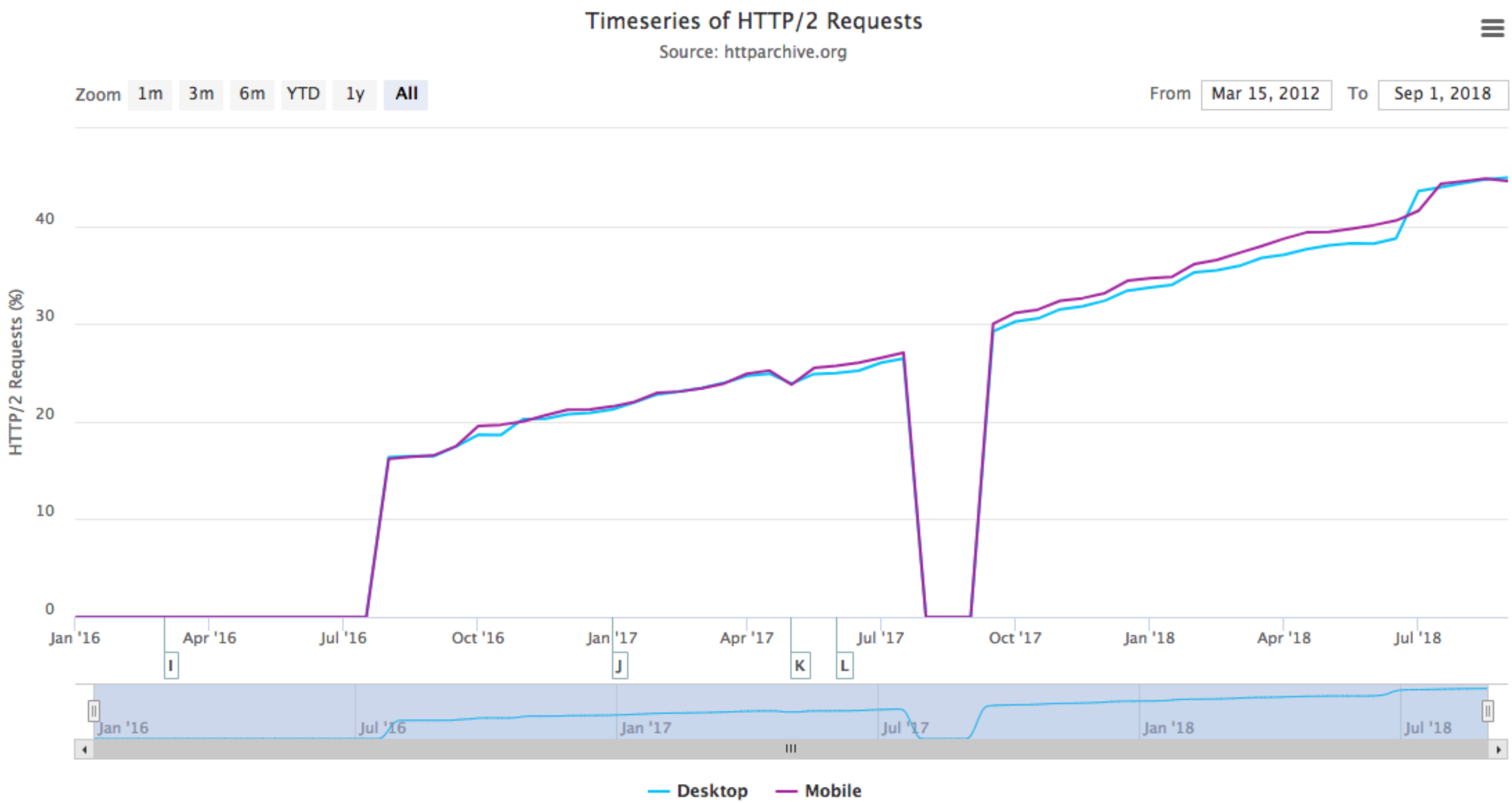
Recap: HTTP/2 Basic Idea: Remove Head-of-Line Blocking in HTTP/1.1



Data flows from sequential to parallel: requests be processed concurrently, in a single connection

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

HTTP/2 Adoption

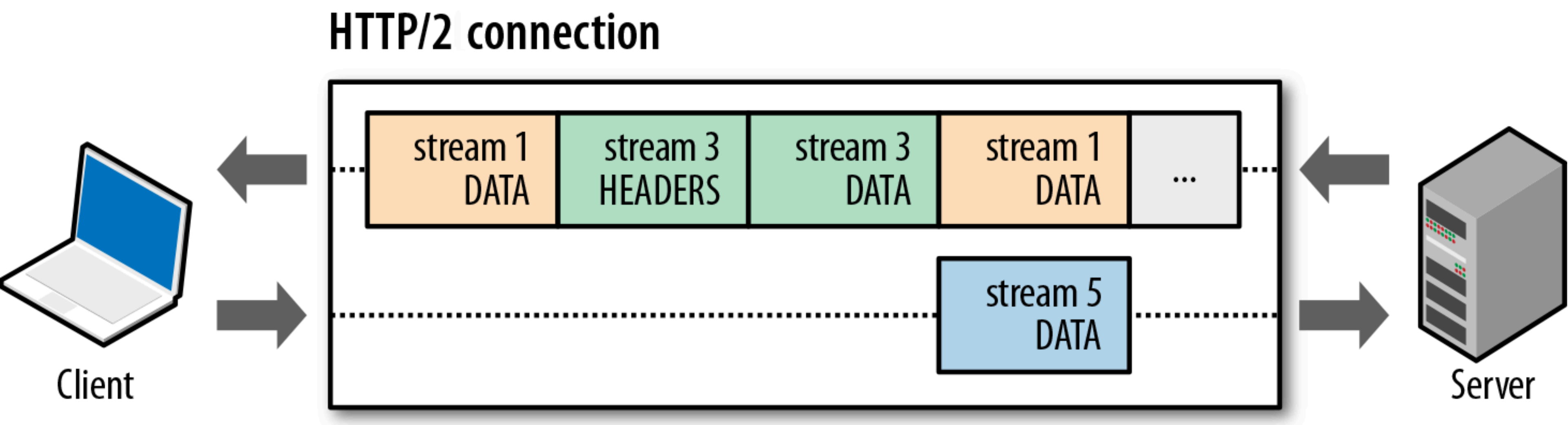


https://httparchive.org/reports/state-of-the-web#h2

HTTP/2 Design: Application-Layer Multiplexing

HTTP/2 Binary Framing to support multiplexing/concurrency

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



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

Ways to Observe HTTP/2

- Visit HTTP/2 pages, such as https://http2.akamai.com
- Wireshark capture
  - export SSLKEYLOGFILE=/tmp/keylog.txt
  - Start Chrome, e.g. /Applications/Google Chrome.app/Contents/MacOS/Google Chrome
- See chrome://net-internals/#http2
- Use a debugging command
  - nghttp -v -n -a -H "CS433533Header: Foo" https://www.akamai.com > out
- Use curl:
  - curl -v --http2 https://akah2san.h2book.com/hello-world.html

HTTP/2 Frame Types

Name	ID	Description
DATA	0x0	Carries the core content for a stream
HEADERS	0x1	Contains the HTTP headers and, optionally, priorities
PRIORITY	0x2	Indicates or changes the stream priority and dependencies
RST_STREAM	0x3	Allows an endpoint to end a stream (generally an error case)
SETTINGS	0x4	Communicates connection-level parameters
PUSH_PROMISE	0x5	Indicates to a client that a server is about to send something
PING	0x6	Tests connectivity and measures round-trip time (RTT)
GOAWAY	0x7	Tells an endpoint that the peer is done accepting new streams
WINDOW_UPDATE	0x8	Communicates how many bytes an endpoint is willing to receive (used for flow control)
CONTINUATION	0x9	Used to extend HEADER blocks

Features: Everything is a Header

- GET / HTTP/1.1
  Host: www.example.com
  User-agent: Next-Great-h2-browser-1.0.0
  Accept-Encoding: compress, gzip

  HTTP/1.1 200 OK
  Content-type: text/plain
  Content-length: 2
  …

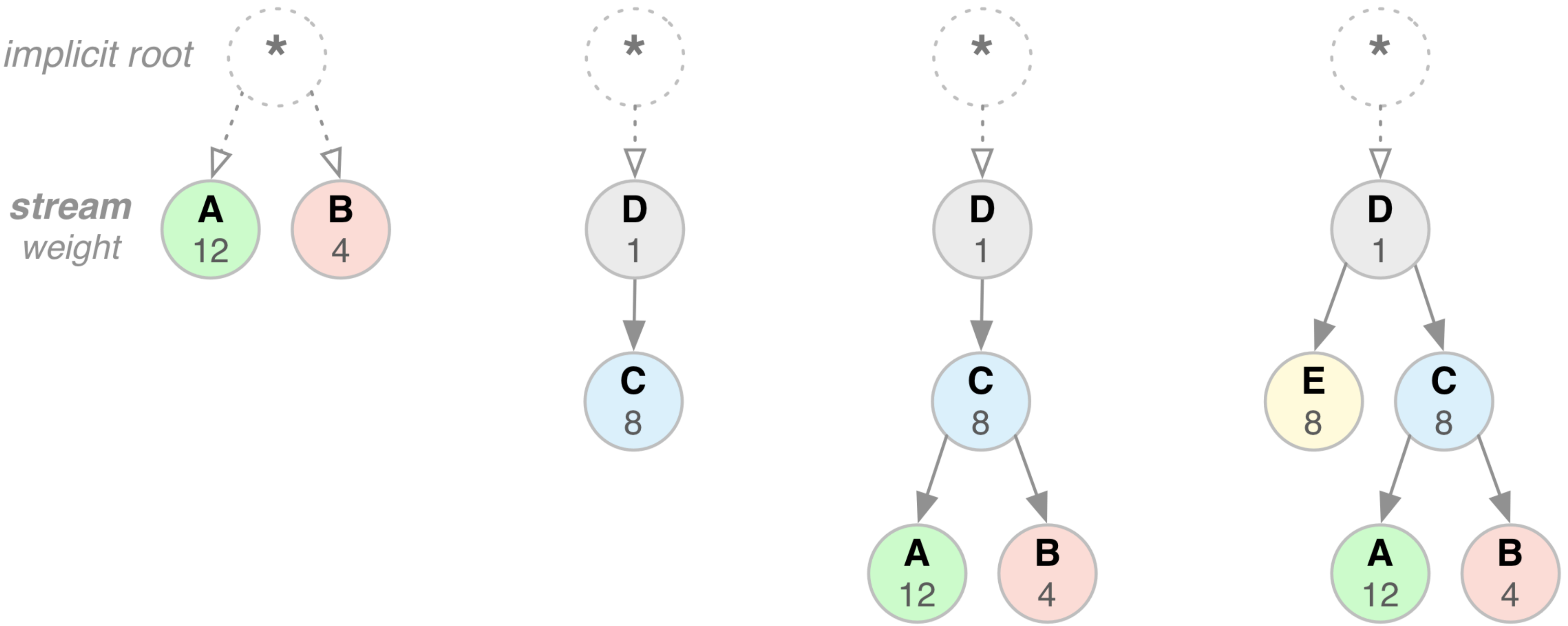
- :scheme: https
  :method: GET
  :path: /
  :authority: www.example.com
  User-agent: Next-Great-h2-browser-1.0.0
  Accept-Encoding: compress, gzip

Features: Allow Priority (Weights) During Multiplexing

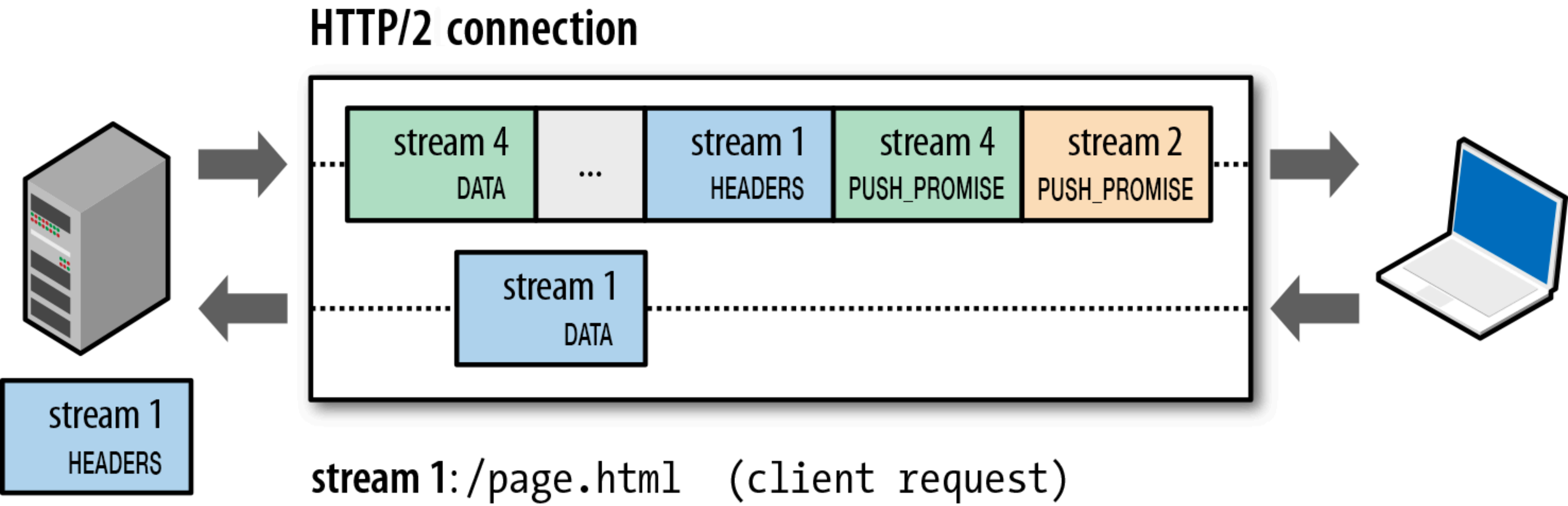
index.html
- *header.jpg*
- *critical.js*
- *less_critical.js*
- *style.css*
- *ad.js*
- *photo.jpg*

index.html
- *style.css*
  - *critical.js*
    - *less_critical.js* (weight 20)
    - *photo.jpg* (weight 8)
    - *header.jpg* (weight 8)
    - *ad.js* (weight 4)

HTTP/2 Stream Dependency and Weights



Features: HTTP/2 Server Push



stream 1: /page.html (client request)
stream 2: /script.js (push promise)
stream 4: /style.css (push promise)

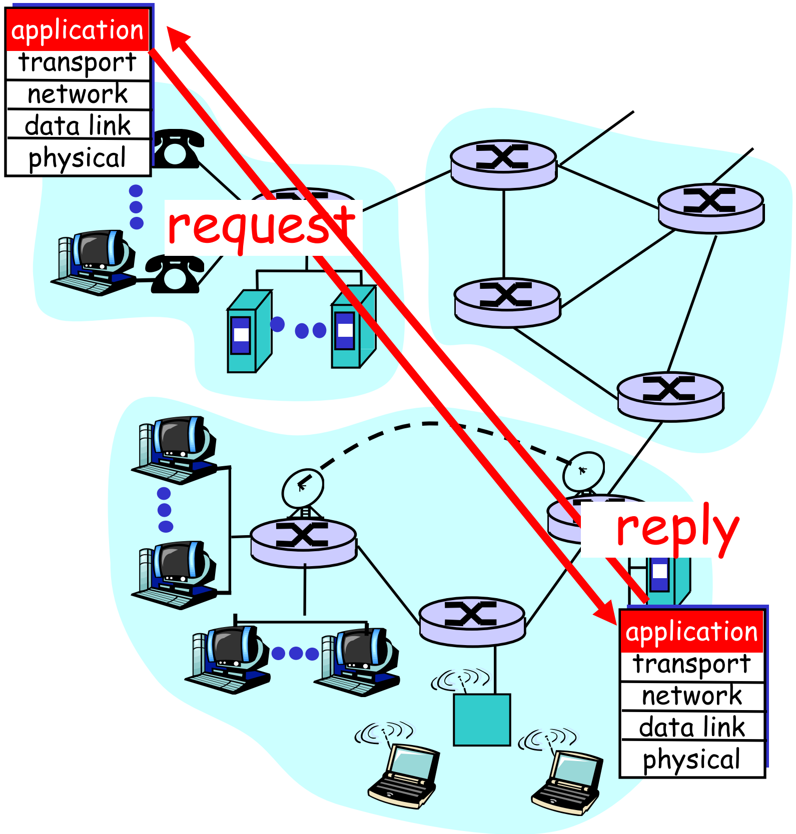
Demo: Put together

- Test www.yale.edu from http://www.webpagetest.org/

HTTP Evaluation

Key questions to ask about a C-S application

- Is the application extensible?
- Is the application scalable?
- How does the application handle server failures (being robust)?
- How does the application provide security?

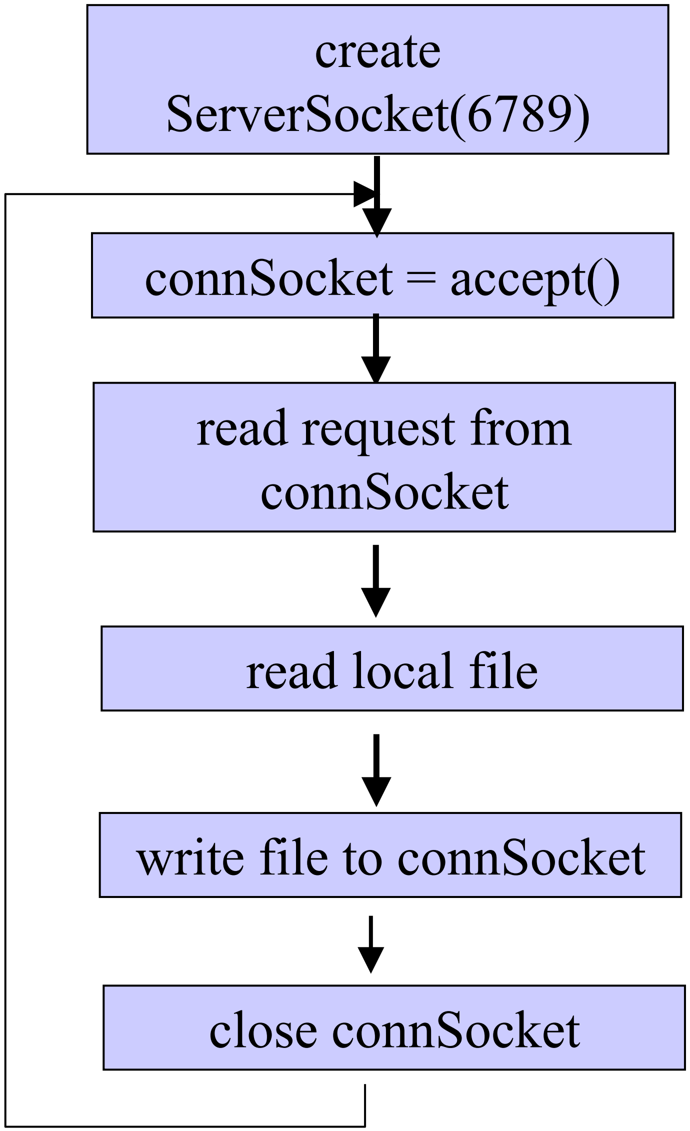


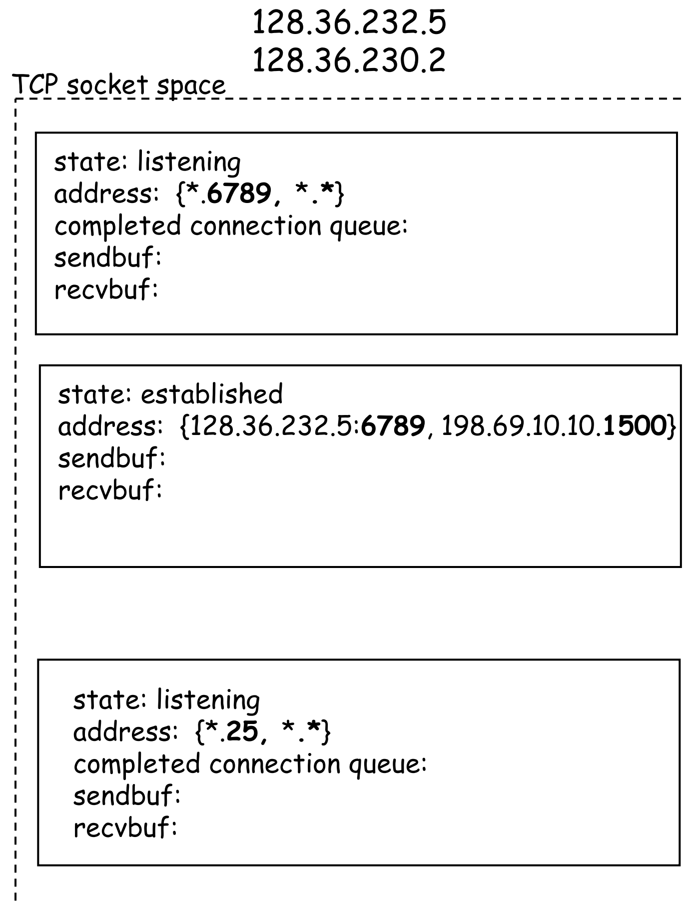
See HTTP extensions such as
- RFC7252 Constrained Application Protocol (CoA)

Outline

- Admin and recap
- HTTP
  - Basic service
  - Basic protocol
  - Basic HTTP client/server workflow
  - Stateful interaction using stateless HTTP
  - HTTP "acceleration"
- Network server design

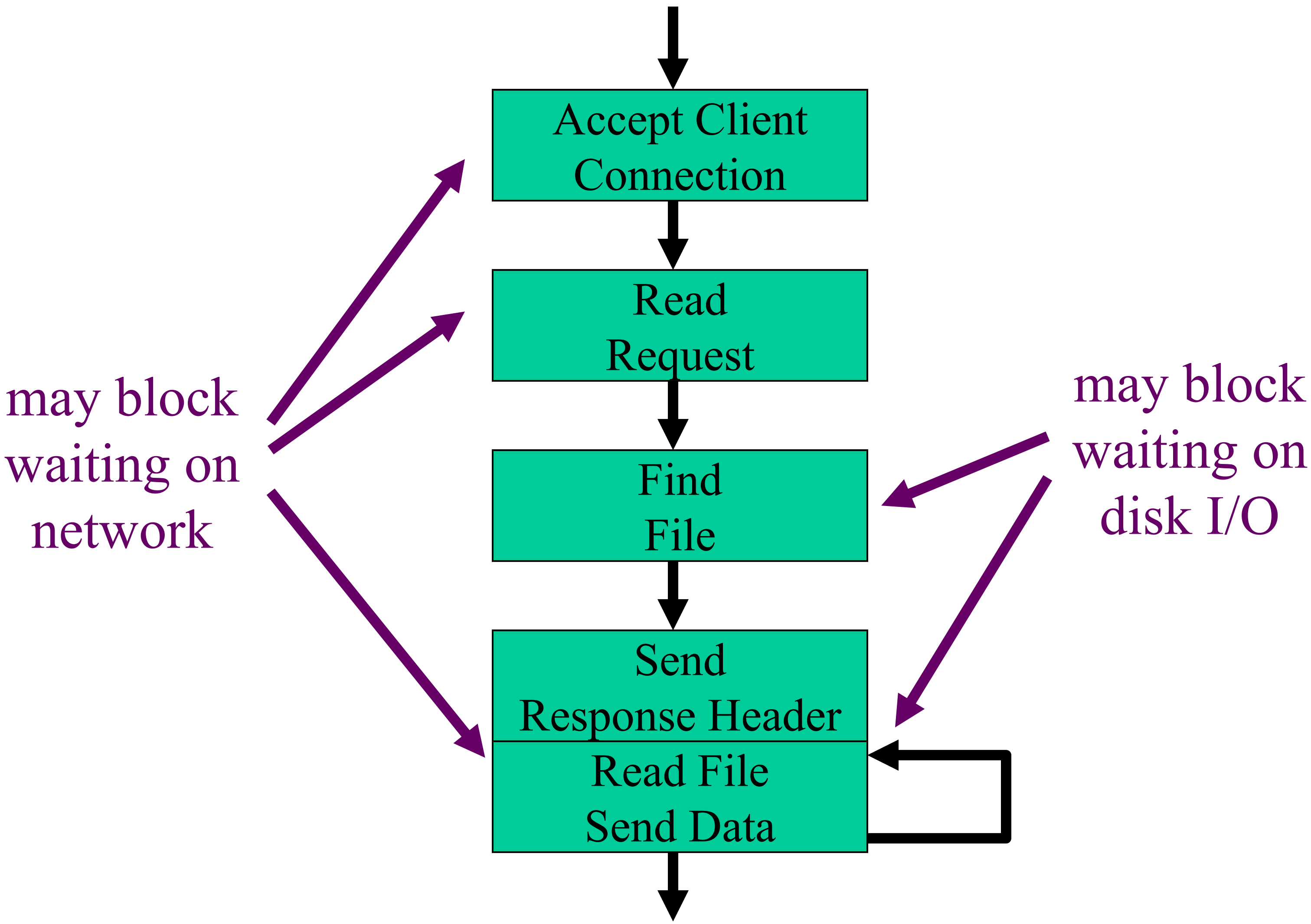
Basic WebServer Implementation





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

Server Processing Steps



Demo: Why Care Blocking?

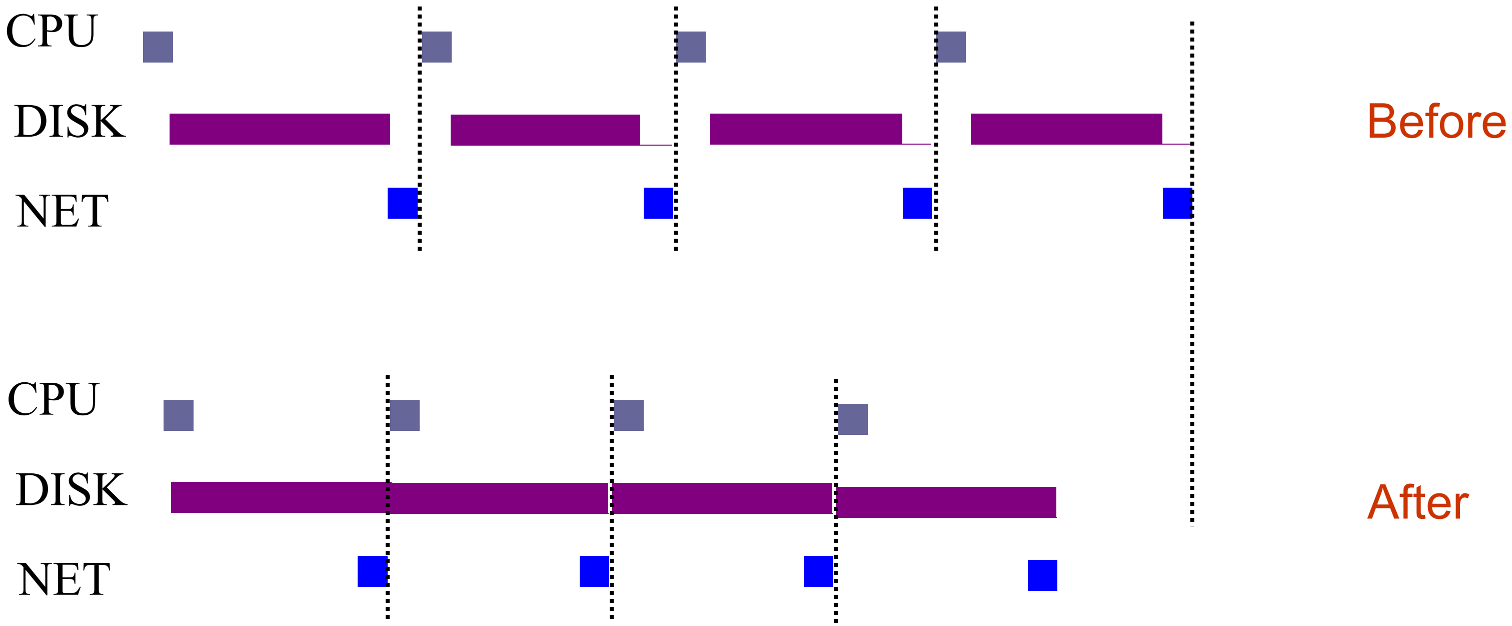
- Try WebServer
- Start two telnet
  - Client 1 starts early but slow input
  - Client 2 starts later but inputs first

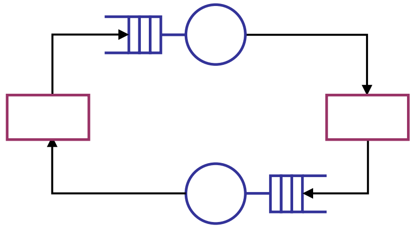
Writing High Performance Servers: Major Issues

- 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

- A blocking server can lead to both low performance and low availability

Goal: Limited Only by Resource Bottleneck



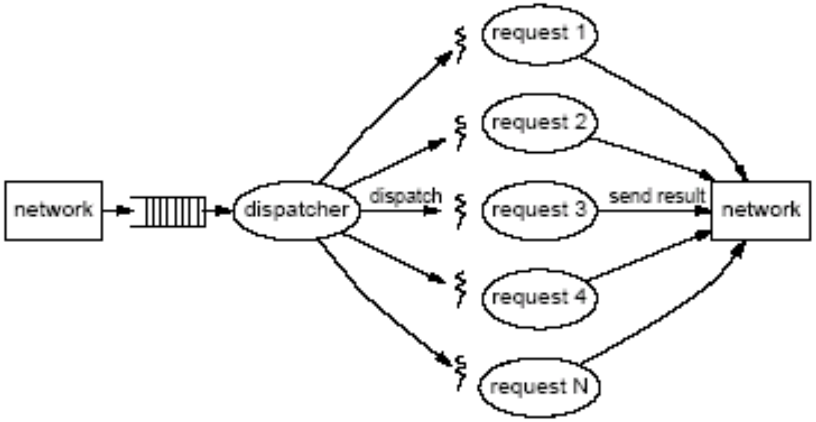


Outline

- Admin and recap
- HTTP
- Network server design
  - Overview
  - Multi-threaded network servers

Multi-Threaded Servers

- Motivation:
  - Avoid blocking the whole program
    (so that we can reach bottleneck throughput)



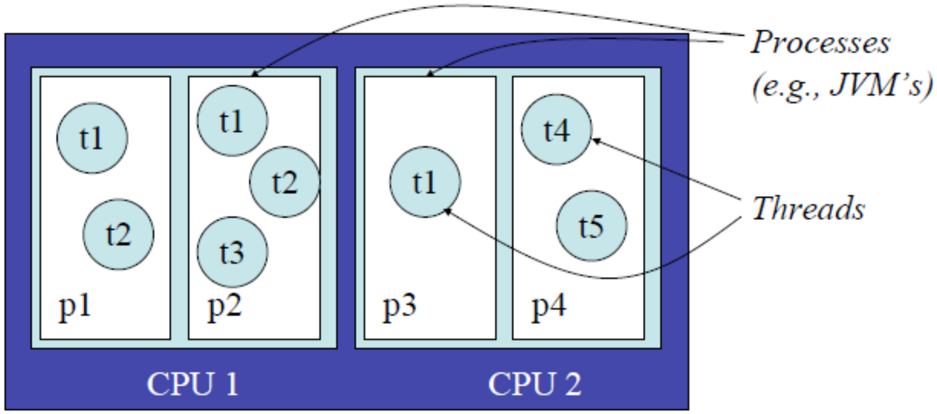
- Idea: introduce threads
  - A thread is a sequence of instructions which may execute in parallel with other threads
  - When a blocking operation happens, only the flow (thread) performing the operation is blocked

Background: Java Thread Model

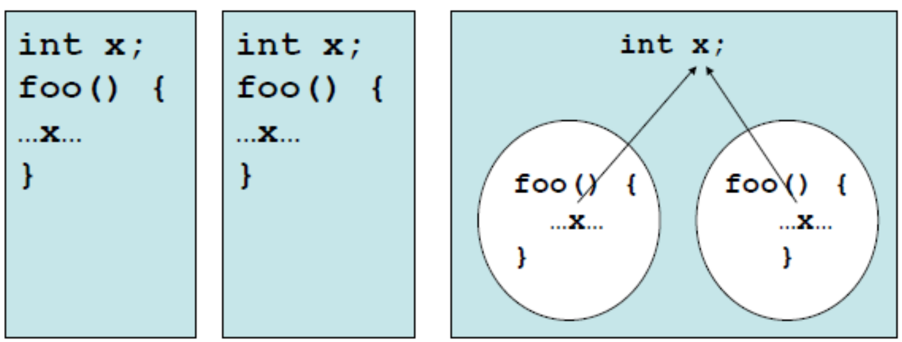
- Every Java application has at least one thread
  - The “main” thread, started by the JVM to run the application’s main() method
  - Most JVMs use POSIX threads to implement Java threads
- main() can create other threads
  - Explicitly, using the Thread class
  - Implicitly, by calling libraries that create threads as a consequence (RMI, AWT/Swing, Applets, etc.)

ps –e | grep java; jstack <pid>

Thread vs Process



A computer

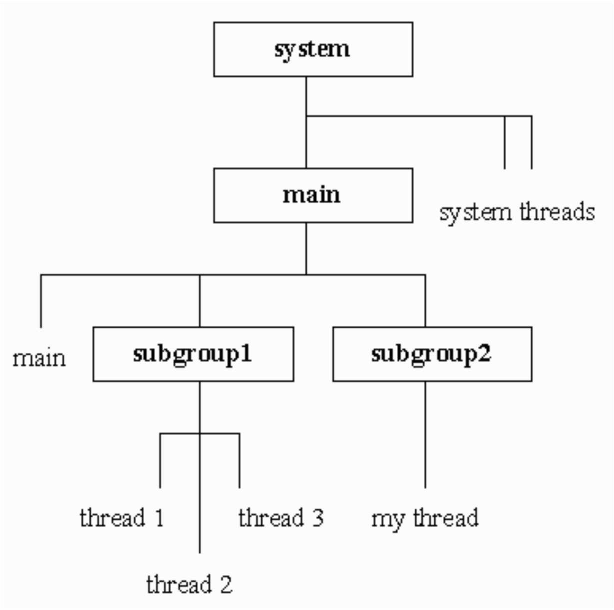


Processes do not share data

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 min, max;

    PrimeThread(long minPrime, long maxPrime) {
        this.min = minPrime; this.max = maxPrime
    }

    public void run() {
       // compute primes in [min, max]
    }
}

PrimeThread p = new PrimeThread(143, 10000);
p.start();
```

Option 2: Implement the Runnable Interface

```
class PrimeRun implements Runnable {
    long min, max;
    PrimeThread(long minPrime, long maxPrime) {
        this.min = minPrime; this.max = maxPrime
    }

    public void run() {
       // compute primes in [min, max] . . .
    }
}

 PrimeRun p = new PrimeRun(143, 10000);

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

Exercise: a Multi-threaded WebServer

- Turn WebServer into a multithreaded server by creating a thread for each accepted request
- Use jstack to list the threads as we add each telnet session

Summary: Implementing Threads

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


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

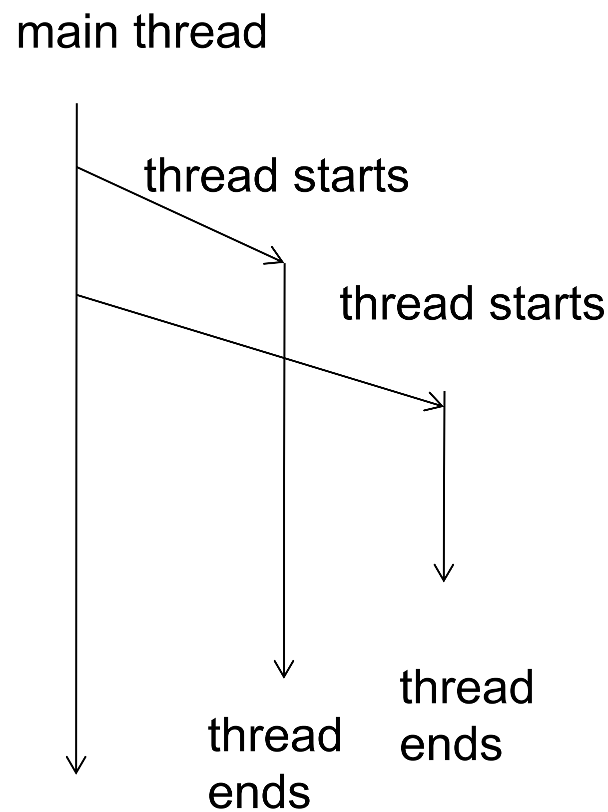
```
class RequestHandler
        implements Runnable {
   RequestHandler(Socket connSocket)
   {
     …
   }
   public void run() {
     // process request
   }
   …
}

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

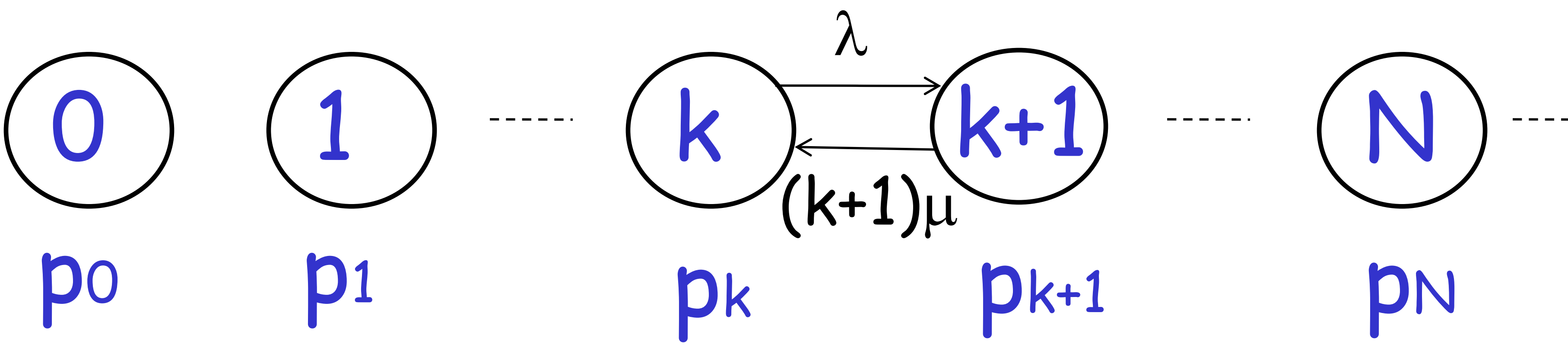
Summary: Per-Request Thread Server

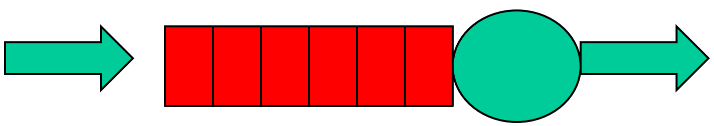
```
main() {
    ServerSocket s = new ServerSocket(port);
    while (true) {
        Socket conSocket = s.accept();
        RequestHandler rh
            = new RequestHandler(conSocket);
        Thread t = new Thread (rh);
        t.start();
    }
```

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

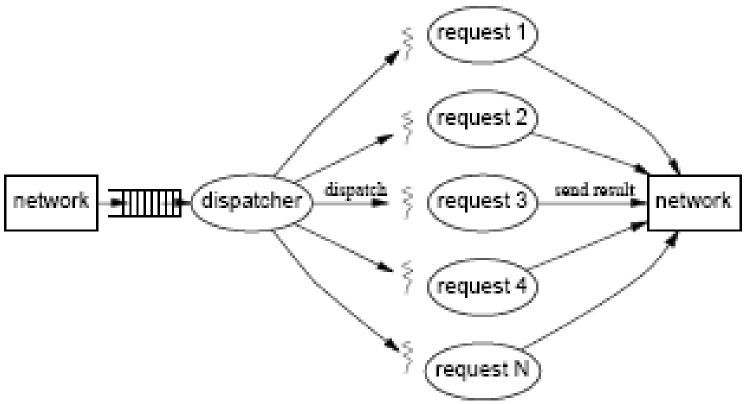


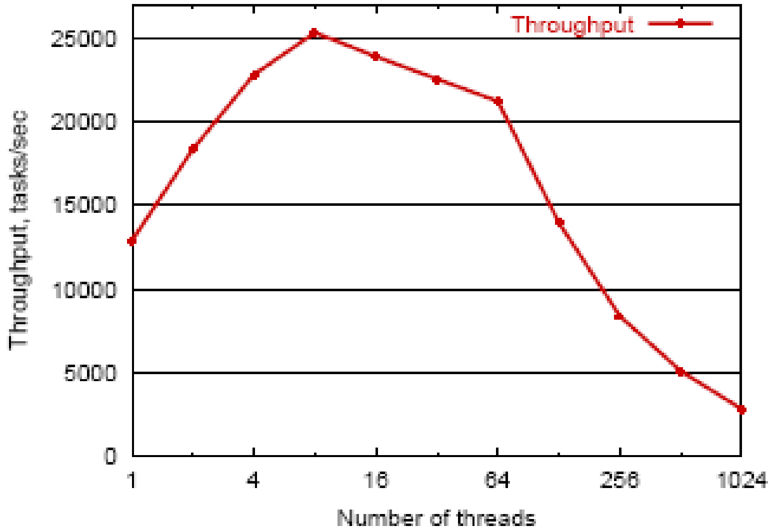
Modeling Per-Request Thread Server: Theory





Problem of Per-Request Thread: Reality



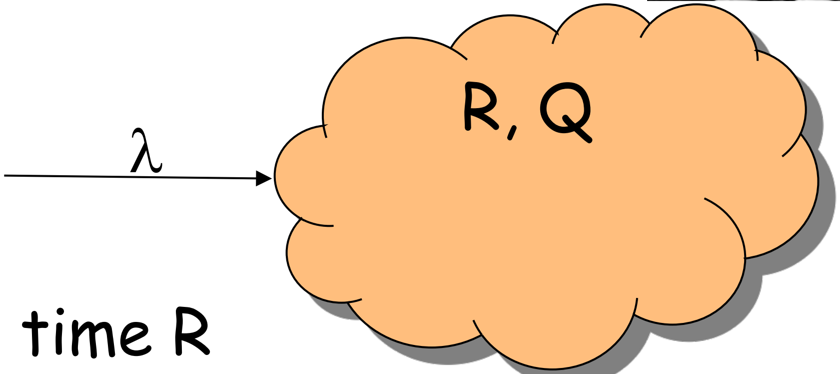


(937 MHz x86, Linux 2.2.14, each thread reading 8KB file)

- 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



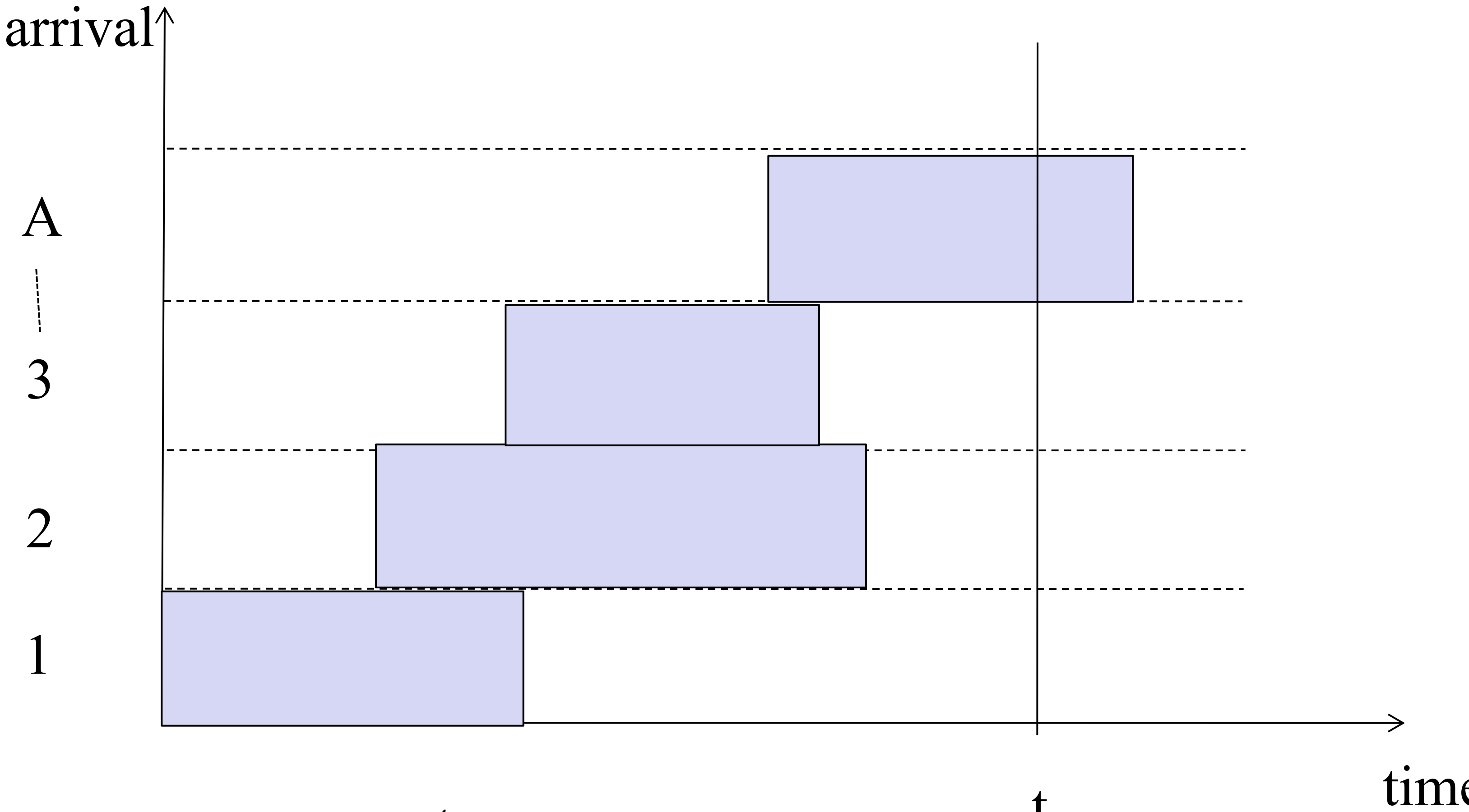
- Then relationship between Q, λ, and R:

$$Q = \lambda R$$

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

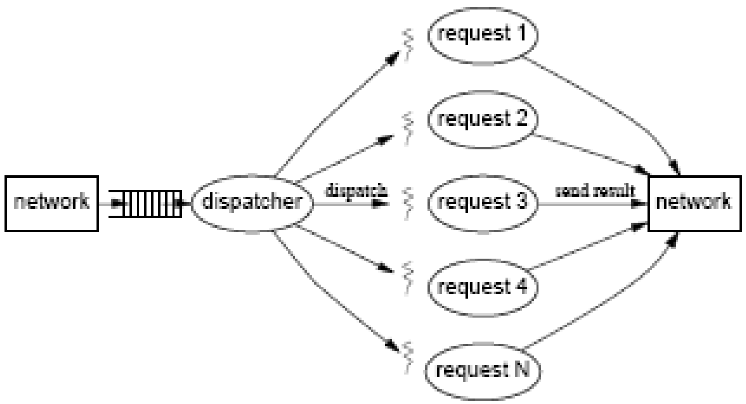
Little's Law

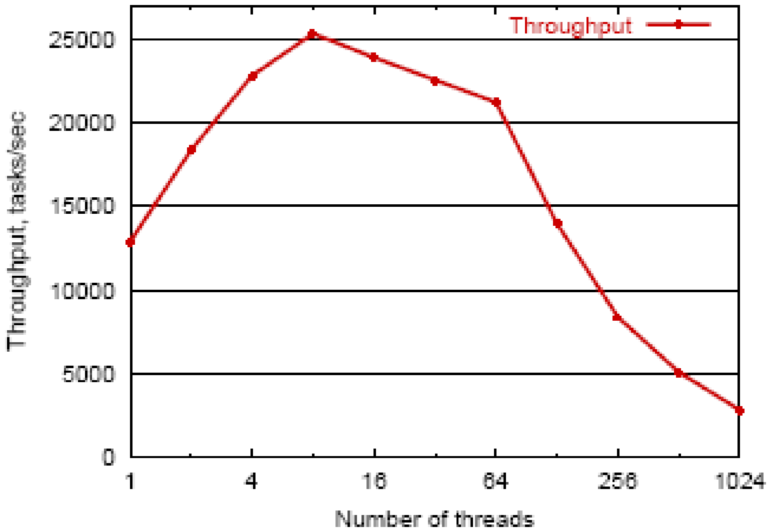
$$Q = \lambda R$$



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

Discussion: How to Address the Issue





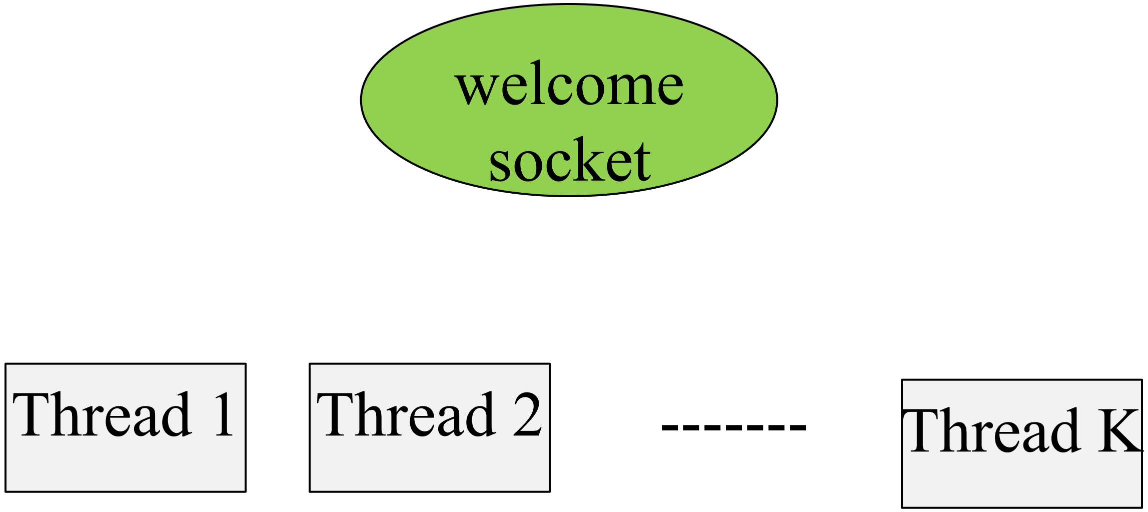
(937 MHz x86, Linux 2.2.14, each thread reading 8KB file)

Outline

- Admin and recap
- HTTP
- High-performance network server design
  - Overview
  - Threaded servers
    - Per-request thread
      - problem: large # of threads and their creations/deletions may let overhead grow out of control
    - Thread pool

Using a Fixed Set of Threads (Thread Pool)

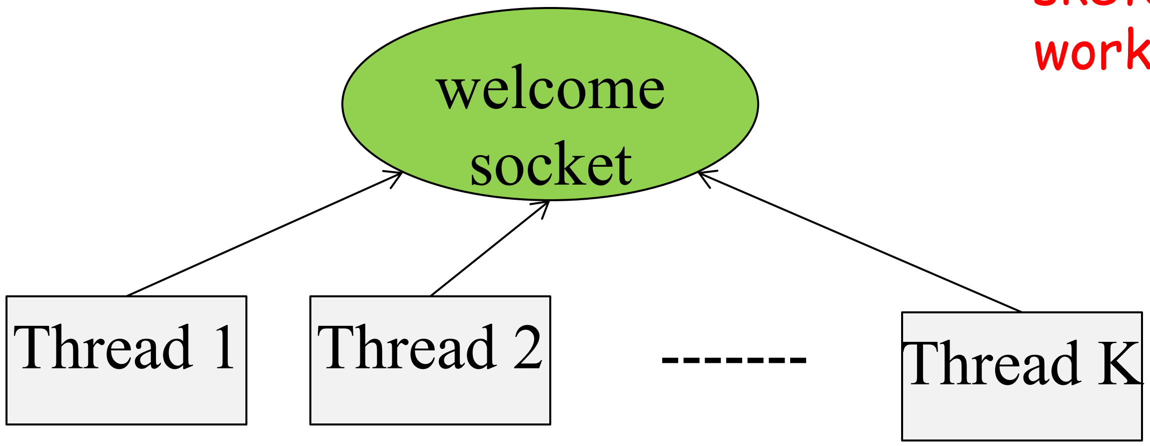
- Design issue: how to distribute the requests from the welcome socket to the thread workers



Design 1: Threads Share Access to the welcomeSocket

```
WorkerThread {
  void run {
    while (true) {
       Socket myConnSock = welcomeSocket.accept();
       // process myConnSock
       myConnSock.close();
    } // end of while
}
```

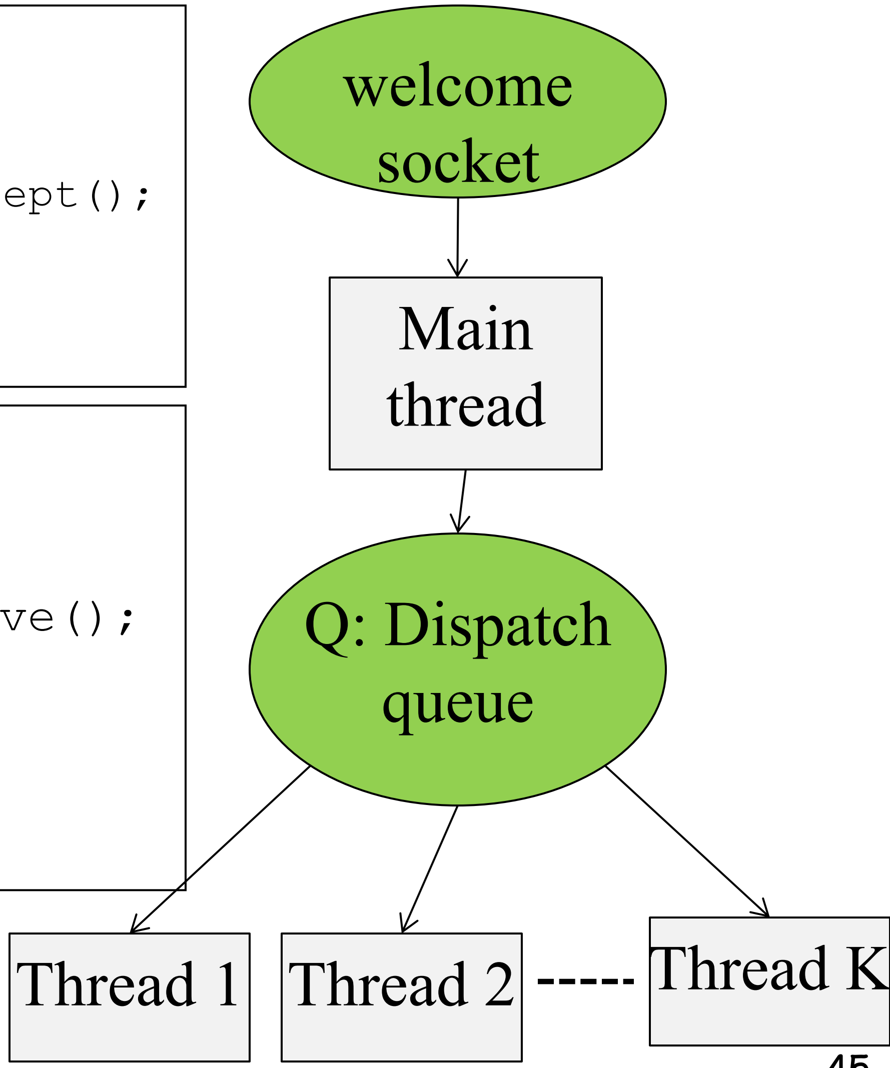
sketch; not working code



Design 2: Producer/Consumer

```
main {
  void run {
    while (true) {
       Socket con = welcomeSocket.accept();
       Q.add(con);
    } // end of while
}
```

```
WorkerThread {
  void run {
    while (true) {
       Socket myConnSock = Q.remove();
       // process myConnSock
       myConnSock.close();
    } // end of while
}
```



sketch; not working code