Proxy: Web & Concurrency

15-213: Introduction to Computer Systems
Recitation 13: Monday, Nov. 18th, 2013

Marjorie Carlson
Section A

Proxy Mechanics

- **Reminder: no partners this year.**
  - No code review (for malloc either!).
  - Partially autograded, partially hand-graded.
- **Due Tuesday, Dec. 3.**
  - You can use two grace days *or* late days.
  - Last day to turn in: Thursday, Dec. 5.
- **Just to orient you…**
  - One week from today: more proxy lab.
  - Two weeks from today: exam review.
  - Three weeks from today: final exam begins.

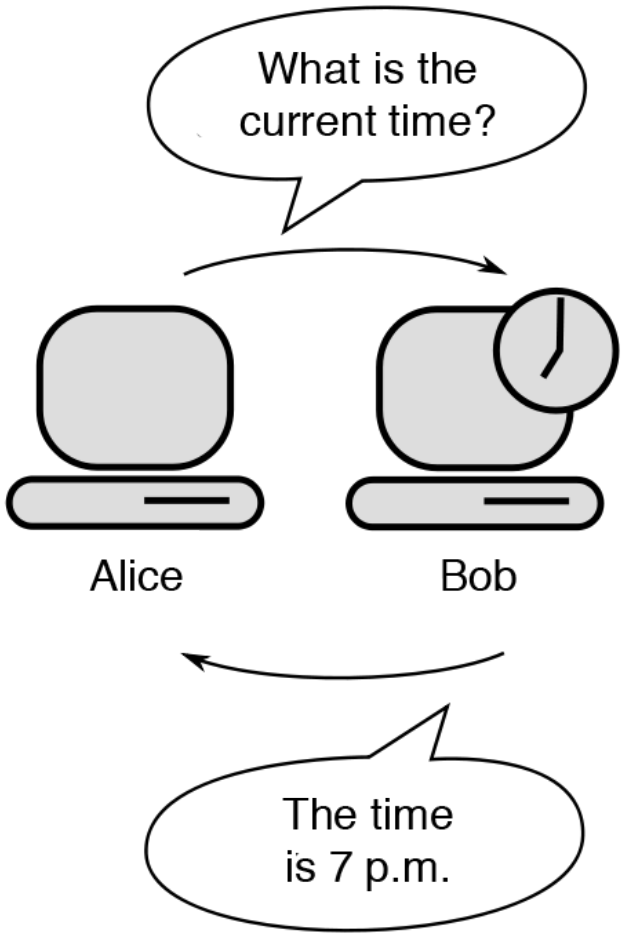
Outline — Proxy Lab

- **Step 1: Implement a sequential web proxy**
- **Step 2: Make it concurrent**
- **Step 3: …***
- **Step 4: PROFIT**

* Cache web objects

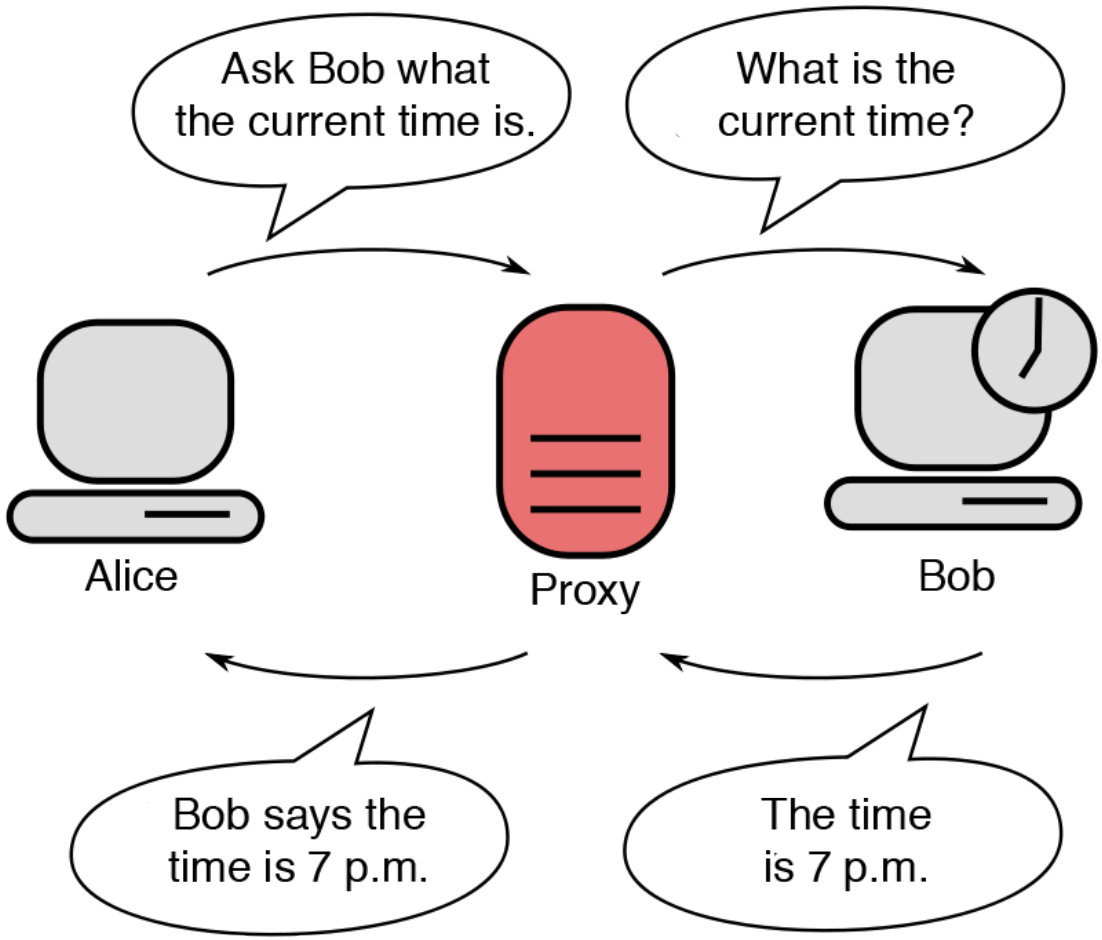
Step 1: Implement a Proxy

- **In the "textbook" version of the web, there are clients and servers.**
  - Clients send requests.
  - Servers fulfill them.
- **Reality is more complicated. In this lab, you're writing a proxy.**
  - A server to the clients.
  - A client to the server(s).



Images here & following based on http://en.wikipedia.org/wiki/File:Proxy_concept_en.svg

Step 1: Implement a Proxy

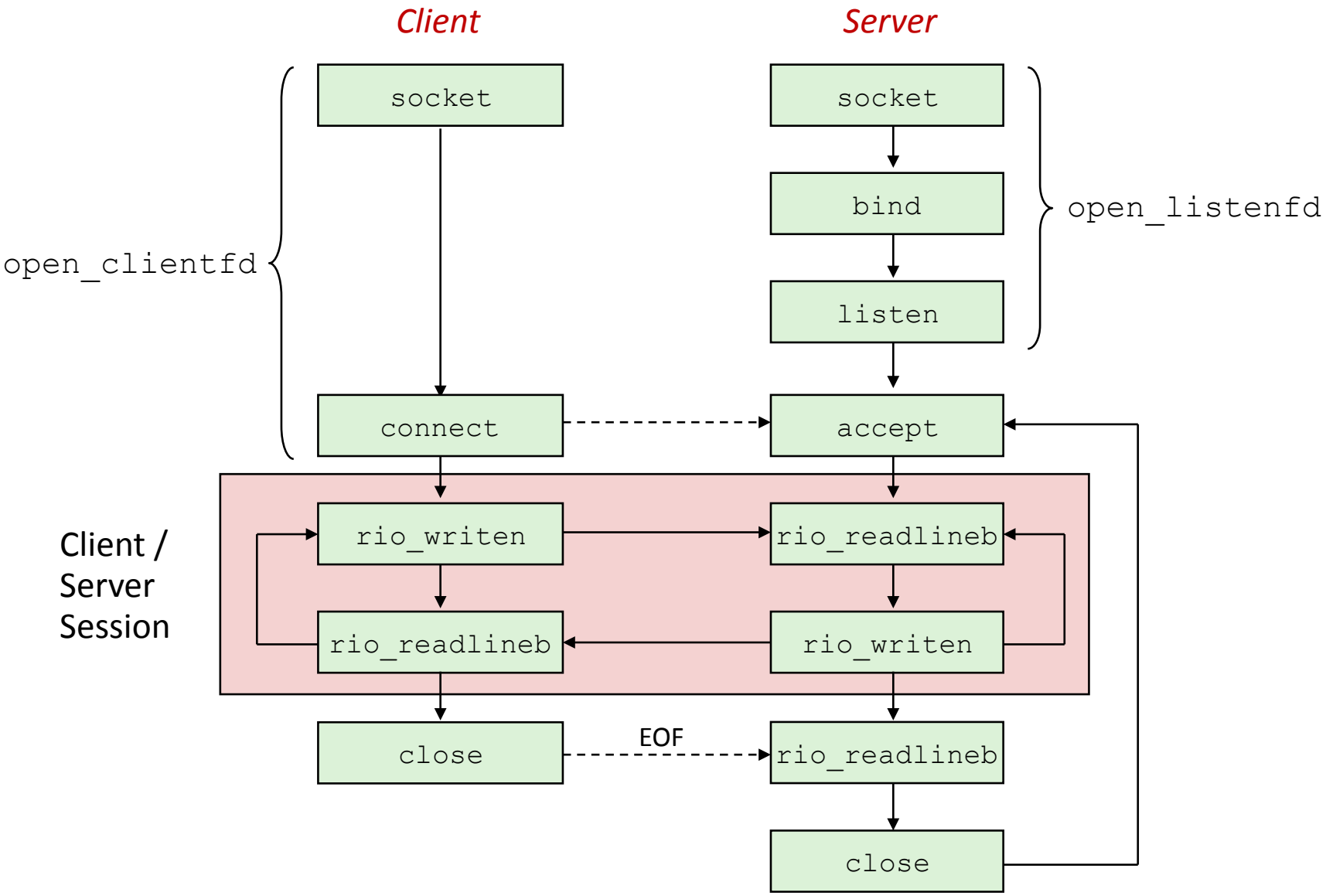


Step 1: Implement a Proxy

- **Proxies are handy for a lot of things.**
  - To filter content … or to bypass content filtering.
  - For anonymity, security, firewalls, etc.
  - For caching — if someone keeps accessing the same web resource, why not store it locally?

- **So how do you make a proxy?**
  - It's a server and a client at the same time.
  - You've seen code in the textbook for a client and for a server; what will code for a proxy look like?
  - Ultimately, the control flow of your program will look more like a server's. However, when it's time to serve the request, a proxy does so by forwarding the request onwards and then forwarding the response back to the client.

Step 1: Implement a Proxy

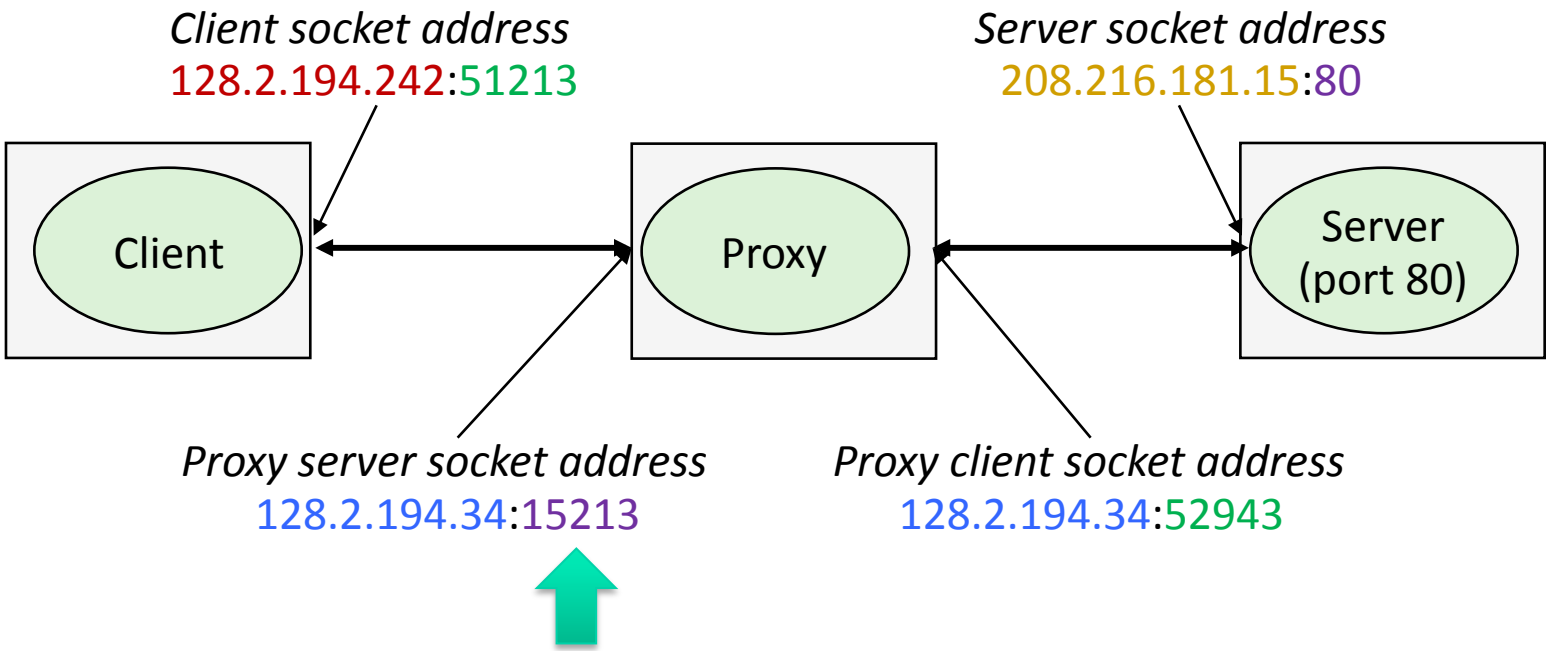


Step 1: Implement a Proxy

- **Your proxy should handle HTTP/1.0 GET requests.**
  - Luckily, that's what the web uses most, so your proxy should work on the vast majority of sites.
    - Reddit, Vimeo, CNN, YouTube, NY Times, etc.
- **Features that require a POST operation (i.e., sending data to the server) will not work.**
  - Logging in to websites, sending Facebook messages, etc.
- **HTTPS is expected *not* to work.**
  - Google (and some other popular websites) now try to push users to HTTPS by default; watch out for that.
- **Your server should be robust. It shouldn't crash if it receives a malformed request, a request for an item that doesn't exist, etc. etc.**

Step 1: Implement a Proxy

- **What you end up with will resemble:**



This is the port number you need to worry about. Use `./port_for_user.pl <your andrewid>` to generate a unique port # to use during testing. When you run your proxy, give that number as a command-line argument, and configure your client (probably Firefox) to use that port.

Aside: Telnet Demo

- **Telnet (an interactive remote shell – like ssh, minus the s)**
  - You must build the HTTP request manually. This will be useful for testing your response to malformed headers.

```
[03:30] [ihartwig@lemonshark:proxylab-handout-f13]% telnet www.cmu.edu 80
Trying 128.2.42.52...
Connected to WWW-CMU-PROD-VIP.ANDREW.cmu.edu (128.2.42.52).
Escape character is '^]'.
GET http://www.cmu.edu/ HTTP/1.0

HTTP/1.1 301 Moved Permanently
Date: Sun, 17 Nov 2013 08:31:10 GMT
Server: Apache/1.3.42 (Unix) mod_gzip/1.3.26.1a mod_pubcookie/3.3.4a mod_ssl/2.8.31 OpenSSL/0.9.8e-
fips-rhel5
Location: http://www.cmu.edu/index.shtml
Connection: close
Content-Type: text/html; charset=iso-8859-

<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML 2.0//EN">
<HTML><HEAD>
<TITLE>301 Moved Permanently</TITLE>
</HEAD><BODY>
<H1>Moved Permanently</H1>
The document has moved <A HREF="http://www.cmu.edu/index.shtml">here</A>.<P>
<HR>
<ADDRESS>Apache/1.3.42 Server at <A HREF="mailto:webmaster@andrew.cmu.edu">www.cmu.edu</A> Port 80</
ADDRESS>
</BODY></HTML>
Connection closed by foreign host.
```

Aside: cURL Demo

- **cURL: "URL transfer library" with command-line program**
  - Builds valid HTTP requests for you!

```
[03:28] [ihartwig@lemonshark:proxylab-handout-f13]% curl http://www.cmu.edu/
<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML 2.0//EN">
<HTML><HEAD>
<TITLE>301 Moved Permanently</TITLE>
</HEAD><BODY>
<H1>Moved Permanently</H1>
The document has moved <A HREF="http://www.cmu.edu/index.shtml">here</A>.<P>
<HR>
<ADDRESS>Apache/1.3.42 Server at <A HREF="mailto:webmaster@andrew.cmu.edu">www.cmu.edu</A> Port 80</
ADDRESS>
</BODY></HTML>
```

  - Can also be used to generate HTTP proxy requests:

```
[03:40] [ihartwig@lemonshark:proxylab-conc]% curl --proxy lemonshark.ics.cs.cmu.edu:3092 http://
www.cmu.edu/
<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML 2.0//EN">
<HTML><HEAD>
<TITLE>301 Moved Permanently</TITLE>
</HEAD><BODY>
<H1>Moved Permanently</H1>
The document has moved <A HREF="http://www.cmu.edu/index.shtml">here</A>.<P>
<HR>
<ADDRESS>Apache/1.3.42 Server at <A HREF="mailto:webmaster@andrew.cmu.edu">www.cmu.edu</A> Port 80</
ADDRESS>
</BODY></HTML>
```

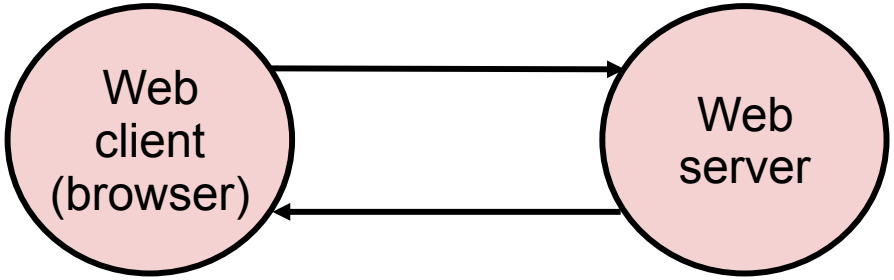
Outline — Proxy Lab

- Step 1: Implement a sequential web proxy
- **Step 2: Make it concurrent**
- Step 3: …*
- Step 4: PROFIT

* Cache web objects

Step 2: Make it Concurrent

- In the textbook version of the web, a client requests a page, the server provides it, and the transaction is done.

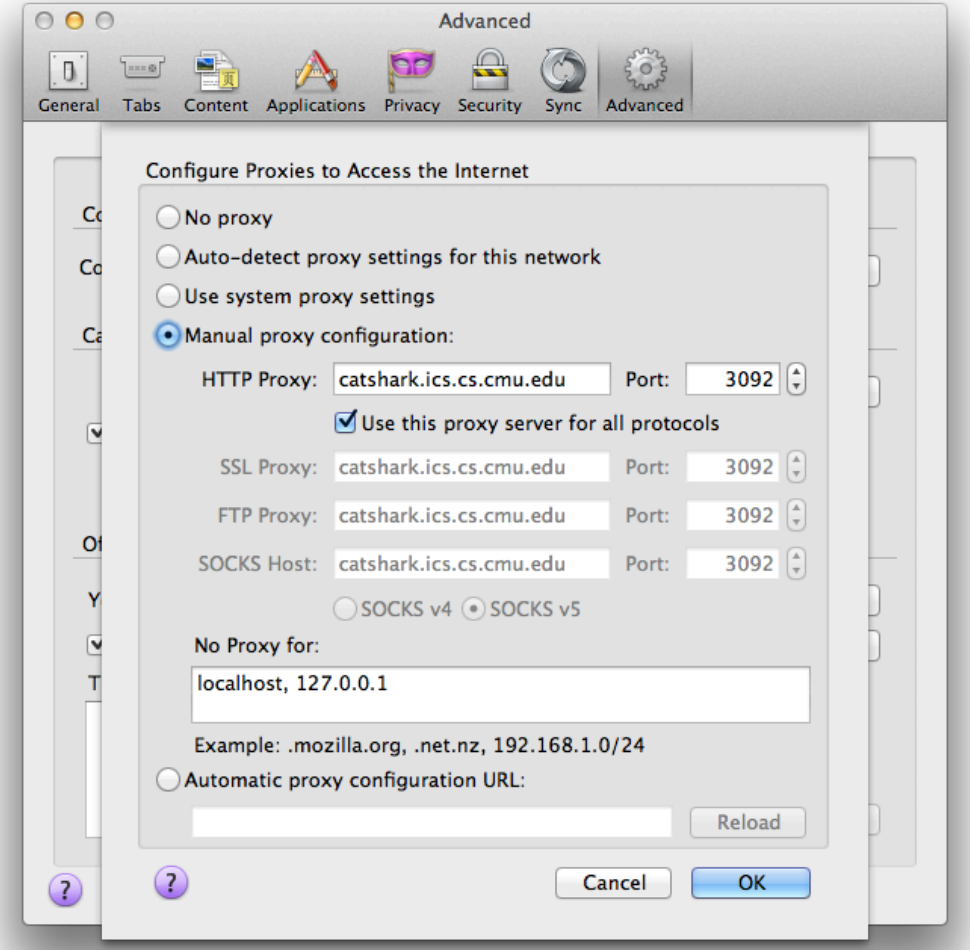


- A sequential server can handle this. We just need to serve one page at a time.
- This works great for simple text pages with embedded styles (a.k.a., the Web circa 1997).

Step 2: Make it Concurrent

- **Let's face it, what your browser is really doing is a little more complicated than that.**
  - A single HTML page may depend on 10s or 100s of support files (images, stylesheets, scripts, etc.).
  - Do you really want to load each of those one at a time?
  - Do you really want to wait for the server to serve every other person looking at the web page before they serve you?
- **To speed things up, you need concurrency.**
  - Specifically, concurrent I/O, since that's generally slower than processing here.
  - You want your server to be able to handle lots of requests at the same time.
- **That's going to require threading. (Yay!)**

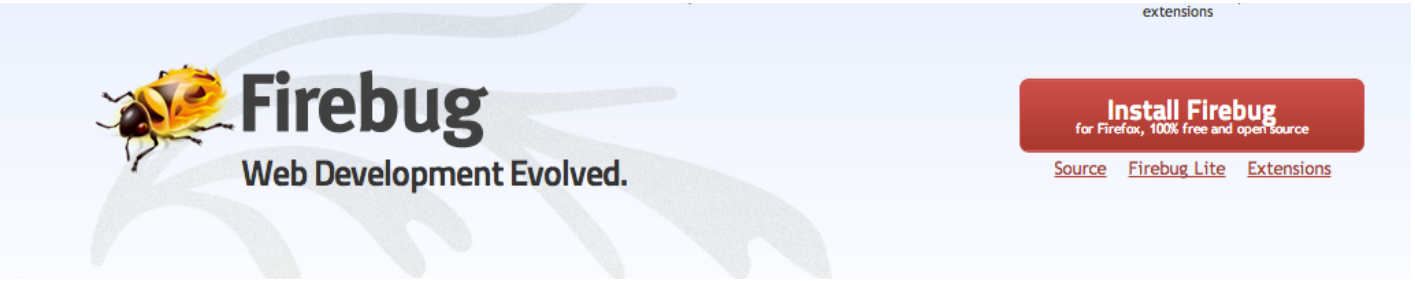
Aside: Setting up Firefox to use a Proxy



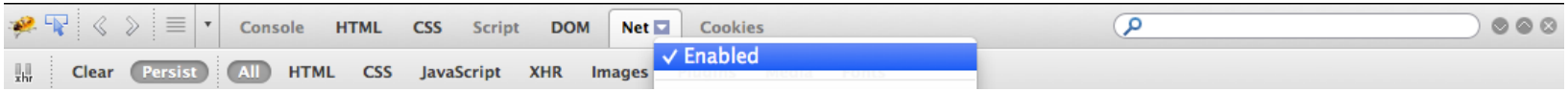
- You may use any browser, but we'll be grading with Firefox
- Preferences > Advanced > Network > Settings… (under Connection)
- Check "Use this proxy for all protocols" or your proxy will appear to work for HTTPS traffic.
- Also, turn off caching!

Aside: Using FireBug to Monitor Traffic

- Install Firebug (getfirebug.com).

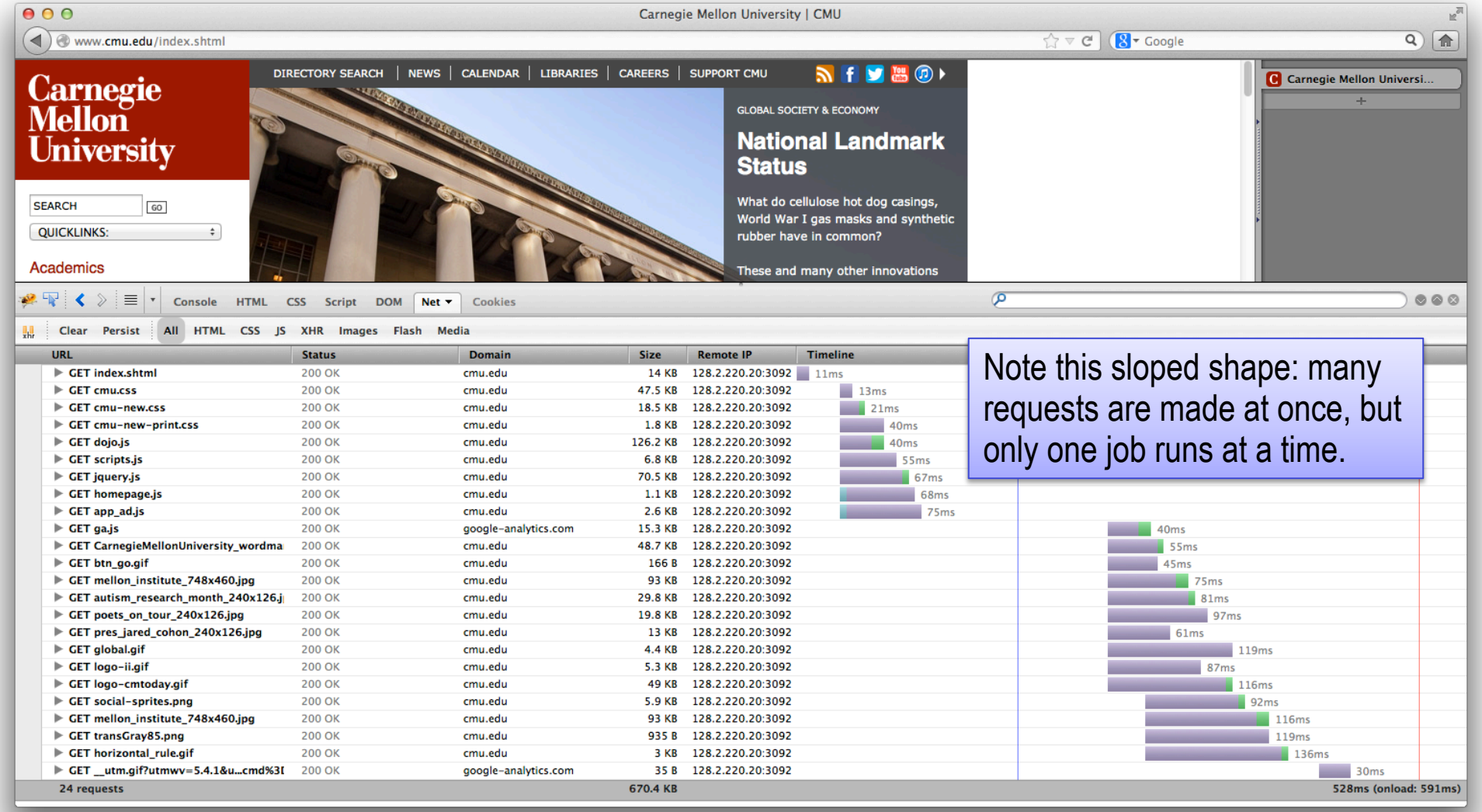


- Tools > Web Developer > FireBug > Open FireBug.
- Click on the triangle besides "Net" to enable it.

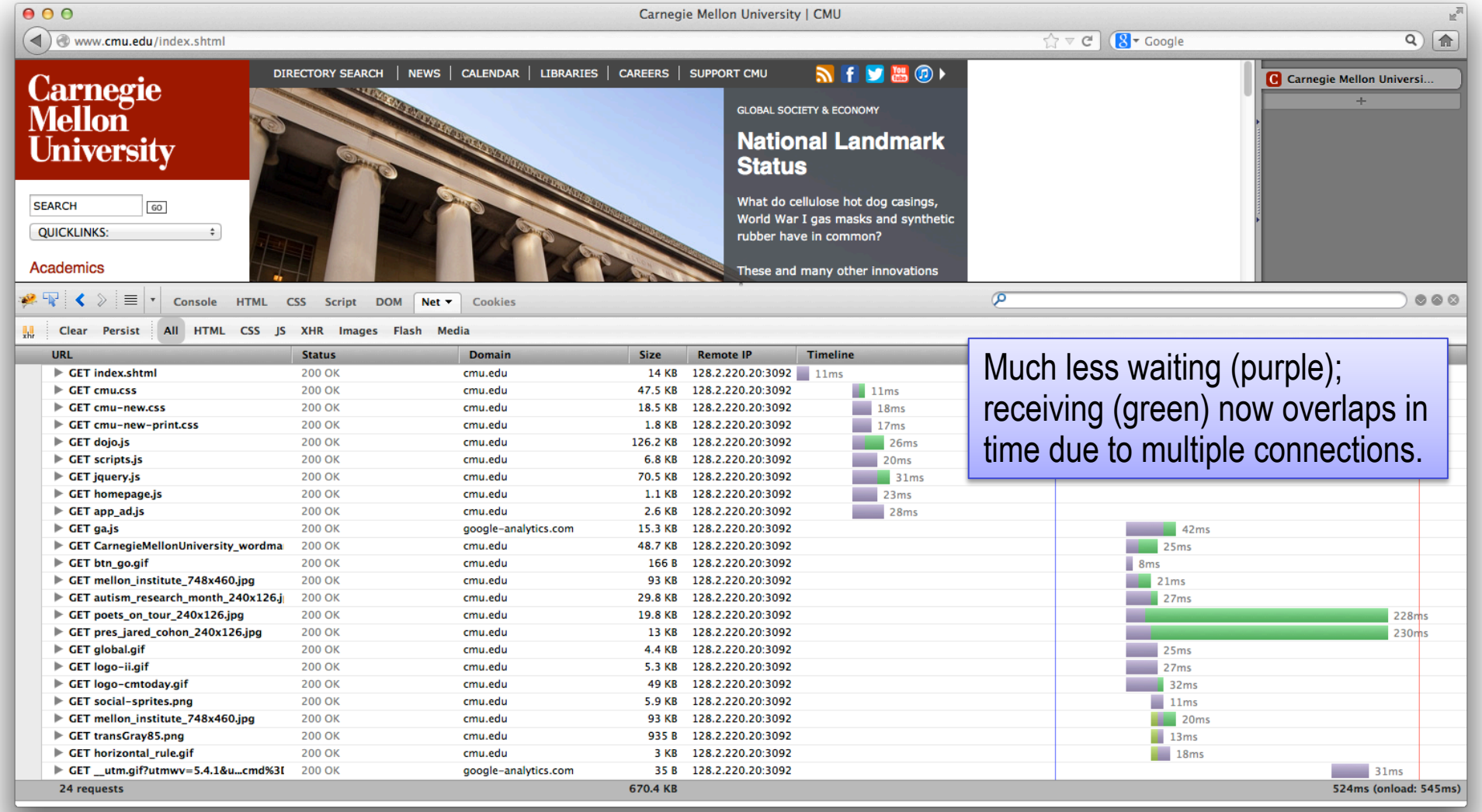


- Now load a web page; you will see each HTML request and see how it resolves, how long it takes, etc.

Make it Concurrent: Sequential Proxy Demo



Make it Concurrent: Concurrent Proxy Demo



Outline — Proxy Lab

- Step 1: Implement a sequential web proxy
- Step 2: Make it concurrent
- **Step 3: …***
- Step 4: PROFIT

*** Cache web objects**

Step 3: Cache Web Objects

- **Your proxy should cache previously requested objects.**
  - Don't panic! This has nothing to do with cache lab. We're just storing things for later retrieval, not managing the hardware cache.
  - Cache individual objects, not the whole page – so, if only part of the page changes, you only refetch that part.
  - The handout specifies a maximum object size and a maximum cache size.
  - **Use an LRU eviction policy.**
  - Your caching system must allow for *concurrent reads* while maintaining consistency.

Step 3: Cache Web Objects

- **Did I hear someone say… *concurrent* reads?**
  - Yup. A sequential cache would bottleneck a parallel proxy.
  - So…
- **Yay! More concurrency!**

- **Multiple threads = concurrency**
- **The cache = a shared resource**
- **So what should we be thinking about?**

Step 3: Cache — Mutexes & Semaphores

- **Mutexes**
  - Allow only one thread to run a section of code at a time.
  - If other threads are trying to run the critical section, they will wait.
- **Semaphores**
  - Allows a fixed number of threads to run the critical section.
  - Mutexes are a special case of semaphores, where the number of threads = 1.

Step 3: Cache — Reading & Writing

- **Reading & writing are sort of a special situation.**
  - Multiple threads can safely *read* cached content.
  - But what about *writing* content?
    - Two threads writing to same cache block?
    - Overwrite block while another thread reading?
- **So:**
  - if a thread is writing, no other thread can read *or* write.
  - if thread is reading, no other thread can write.
- **Potential issue: writing starvation**
  - If threads are always reading, no thread can write.
  - Solution: if a thread is waiting to write, it gets priority over any new threads trying to read.
  - What can we use to do this?

Step 3: Cache — Read-Write Locks

- **How would you make a read-write lock with semaphores?**
  - Luckily, you don't have to!

    `pthread_rwlock_*` handles that for you
    - `pthread_rwlock_t lock;`
    - `pthread_rwlock_init(&lock,NULL);`
    - `pthread_rwlock_rdlock(&lock);`
    - `pthread_rwlock_wrlock(&lock);`
    - `pthread_rwlock_unlock(&lock);`

Outline — Proxy Lab

- Step 1: Implement a sequential web proxy
- Step 2: Make it concurrent
- Step 3: …*
- **Step 4: PROFIT**

* Cache web objects

Step 4: Profit

- **New: Autograder**
  - Autolab and `./driver.sh` will check your proxy's ability to:
    - pull basic web pages from a server.
    - handle multiple requests concurrently.
    - fetch a web page from your cache.
  - Please don't use this grader to definitively test your proxy; there are many things not tested here.
- **Ye Olde Hand-Grading**
  - A TA will grade your code based on correctness, style, race conditions, etc., and will additionally visit the following sites on Firefox through your proxy:
    - http://www.cs.cmu.edu/~213
    - http://www.cs.cmu.edu/~droh
    - http://www.nfl.com
    - http://www.youtube.com/watch?v=ZOsLgnYeEk8

Step 4: Preparing to Profit…

- **Test your proxy liberally!**
  - We don't give you traces or test cases, but the web is *full* of special cases that want to break your proxy!
  - Use **telnet** and/or **cURL** to make sure your basics are working.
  - You can also set up **netcat** as a server and send requests to it, just to see how your traffic looks to a server.
  - When the basics are working, start working through **Firefox**.
  - To test caching, consider using your andrew web space (`~/www`) to **host test files**. (You can fetch them, take them down, and fetch them again, to make sure your proxy still has them.)
    - To publish your folder to the public server, you must go to https://www.andrew.cmu.edu/server/publish.html.

Confused where to start?

- **Grab yourself a copy of the echo server (pg. 910) and client (pg. 909) in the book.**
- **Also review the tiny.c basic web server code to see how to deal with HTTP headers.**
  - Note that tiny.c ignores these; you may not.
- **As with malloclab, this will be an iterative process:**
  - Figure out how to make a small, sequential proxy, and test it with telnet and curl.
  - Make it more robust. (You'll spend a lot of time parsing & dealing with headers.)
  - Make it concurrent.
  - Make it caching.
  - Repeat until you're happy with it.

Questions?