Networking Basics and Concurrent Programming

Shiva (sshankar)

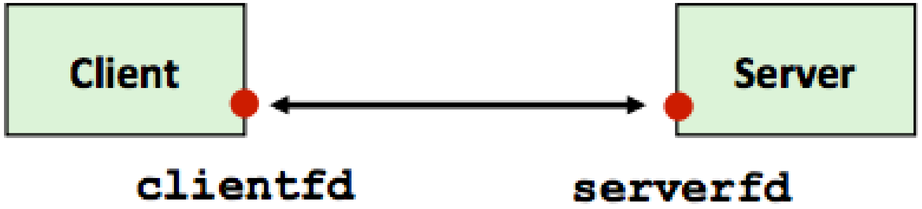
Section M

Topics

- Networking Basics
- Concurrent Programming
- Introduction to Proxy Lab

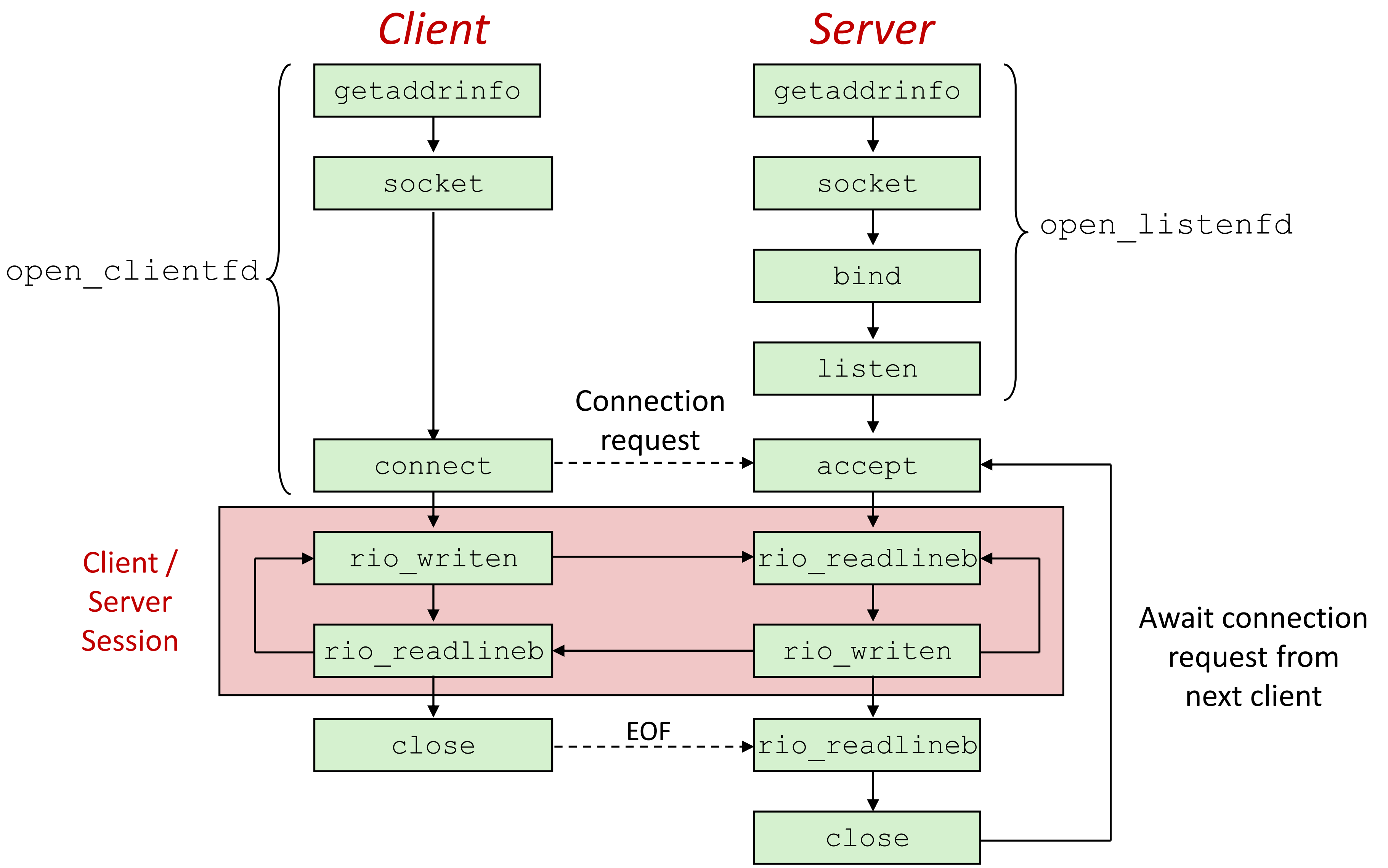
Sockets

- **What is a socket?**
  - To an application, a socket is a file descriptor that lets the application read/write from/to the network
  - (all Unix I/O devices, including networks, are modeled as files)
- **Clients and servers communicate with each other by reading from and writing to socket descriptors**



- **The main difference between regular file I/O and socket I/O is how the application "opens" the socket descriptors**

Overview of the Sockets Interface



Host and Service Conversion: `getaddrinfo`

- **`getaddrinfo`** is the modern way to convert string representations of host, ports, and service names to socket address structures.
  - Replaces obsolete `gethostbyname` - unsafe because it returns a pointer to a static variable
- **Advantages:**
  - Reentrant (can be safely used by threaded programs).
  - Allows us to write portable protocol-independent code(IPv4 and IPv6)
  - Given `host` and `service`, `getaddrinfo` returns `result` that points to a linked list of `addrinfo` structs, each pointing to socket address struct, which contains arguments for sockets APIs.

- **`getnameinfo` is the inverse of getaddrinfo, converting a socket address to the corresponding host and service.**
  - Replaces obsolete `gethostbyaddr` and `getservbyport` funcs.

Sockets API

- **int socket(int domain, int type, int protocol);**
  - Create a file descriptor for network communication
  - used by both clients and servers
  - int sock_fd = socket(PF_INET, SOCK_STREAM, IPPROTO_TCP);
  - One socket can be used for two-way communication

- **int bind(int socket, const struct sockaddr *address, socklen_t address_len);**
  - Associate a socket with an IP address and port number
  - used by servers
  - struct sockaddr_in sockaddr – family, address, port

Sockets API

- **int listen(int socket, int backlog);**
  - socket: socket to listen on
  - used by servers
  - backlog: maximum number of waiting connections
  - err = listen(sock_fd, MAX_WAITING_CONNECTIONS);

- **int accept(int socket, struct sockaddr *address, socklen_t *address_len);**
  - used by servers
  - socket: socket to listen on
  - address: pointer to sockaddr struct to hold client information after accept returns
  - return: file descriptor

Sockets API

- **int connect(int socket, struct sockaddr *address, socklen_t address_len);**
  - attempt to connect to the specified IP address and port described in address
  - used by clients

- **int close(int fd);**
  - used by both clients and servers
  - (also used for file I/O)
  - fd: socket fd to close

Sockets API

- **ssize_t read(int fd, void *buf, size_t nbyte);**
  - used by both clients and servers
  - (also used for file I/O)
  - fd: (socket) fd to read from
  - buf: buffer to read into
  - nbytes: buf length

- **ssize_t write(int fd, void *buf, size_t nbyte);**
  - used by both clients and servers
  - (also used for file I/O)
  - fd: (socket) fd to write to
  - buf: buffer to write
  - nbytes: buf length

Topics

- Networking Basics
- Concurrent Programming
- Introduction to Proxy Lab

Threads

- **Threads enable light-weight concurrency by sharing much of the same address space**
- **Similarities to processes**
  - each thread has its own logical control flow (its own registers, so its own EIP and ESP)
  - multiple threads can be in the middle of running at the same time, possibly on different cores
  - the kernel decides when to context switch to and from a thread (or a thread can voluntarily give up its share of cpu time by calling sleep, pause, sigsuspend, etc)
- **Differences with processes**
  - threads share code and data; processes generally don’t
  - threads are lesser overhead than processes (to create and reap)

Threads: pthreads interface

- **Creating/reaping threads**
  - pthread_create
  - pthread_join
- **To get your thread ID**
  - pthread_self
- **Terminating threads**
  - pthread_cancel
  - pthread_exit
- **synchronizing access to shared variables**
  - pthread_mutex_init
  - pthread_mutex_[un]lock
  - pthread_rwlock_init
  - pthread_rwlock_[wr]rdlock

Thread exit

- A thread terminates *implicitly* when its top-level thread routine returns
- A thread terminates *explicitly* by calling pthread_exit(NULL)
- pthread_exit(NULL) only terminates the current thread, NOT the process
- exit(0) terminates ALL the threads in the process (meaning the whole process terminates)
- pthread_cancel(tid) terminates the thread with id equal to tid

Threads - Reaping

- **Joinable threads can be reaped and killed by other threads**
  - must be reaped with pthread_join to free memory and resources
- **Detached threads cannot be reaped or killed by other threads**
  - resources are automatically reaped on termination
- **Default state is joinable**
  - use pthread_detach(pthread_self()) to make detached

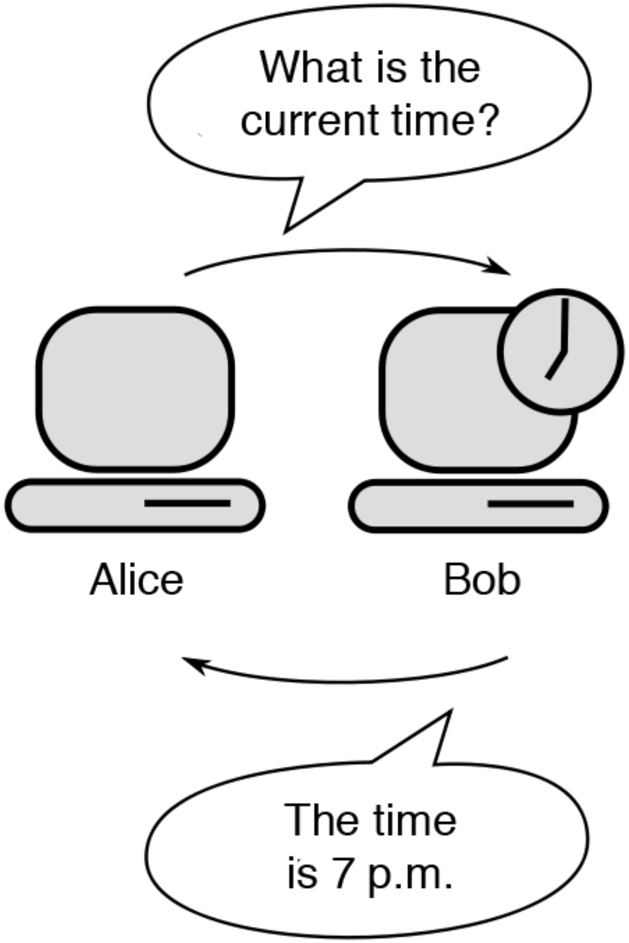
Mellon

Topics

- Networking Basics
- Concurrent Programming
- **Introduction to Proxy Lab**

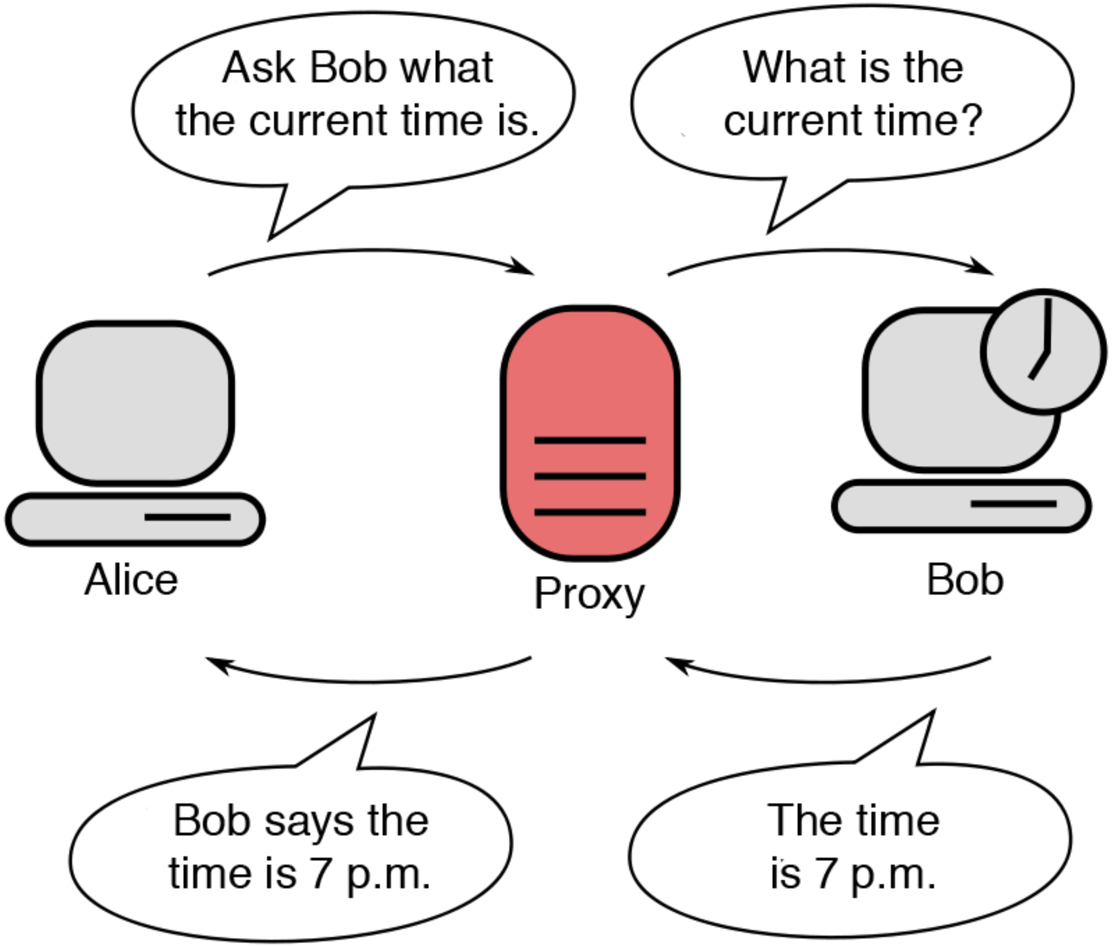
What is a Proxy?

- **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).



http://en.wikipedia.org/wiki/File:Proxy_concept_en.svg

What is a Proxy?



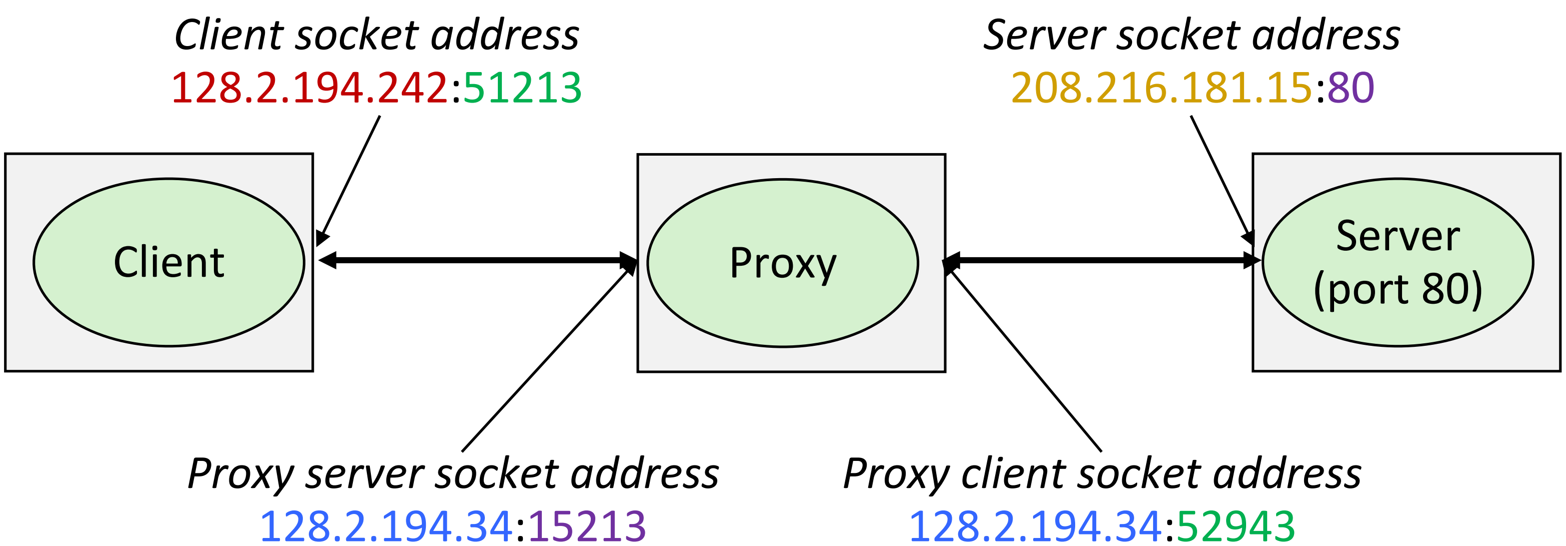
Why and How?

- **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 Proxy Lab

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

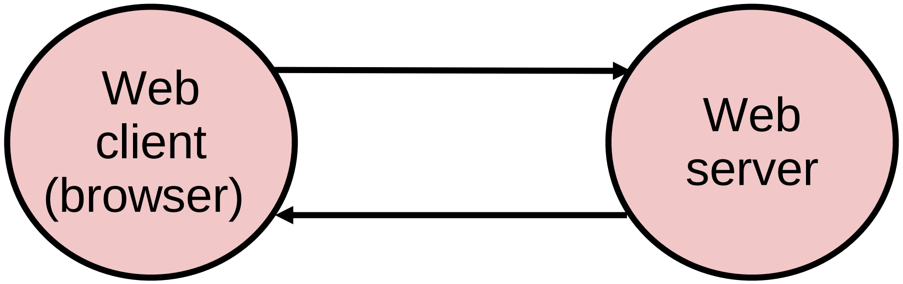


Proxy Lab

- **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.**

Sequential Proxy

- 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: Concurrent Proxy

- **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!)**

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. Concurrency? Shared Resource?

Questions?

(come to office hours if you need help)