15-213 Recitation
Networking and Proxies

Your TAs

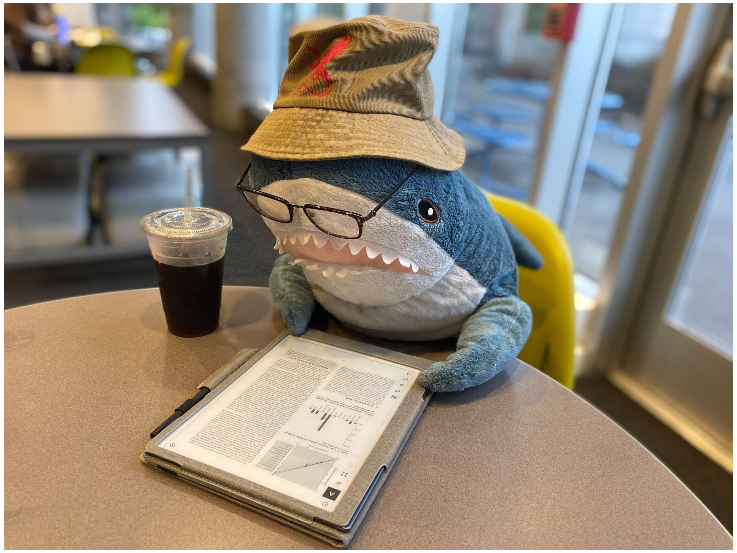
Friday, November 14th

Reminders

- **`proxylab`** is released *November 18th. Due November 25*
- **`sfslab`** will be released on November 25th
  - Due *December 04*
- ***Written 10*** due *November 20th*
- Code Reviews:
  - **`tshlab`**

Apply to be a TA!

- TA Applications are open! See Ed #888 :-)
  - First round of interviews happening in 1-2 weeks!
- What qualifications are we looking for?
  - Decent class performance
  - Strong communication skills
  - Reasonable ability to gauge schedule and responsibilities



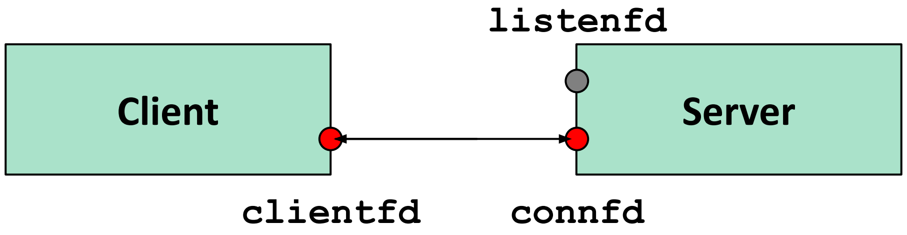
Agenda

- Network Review
- Activity: Telephone
- Proxy Lab
  - What is a proxy?
  - Getting started

Review: Networks

Networking Refresher

- UNIX File Abstraction: communicate over the network by reading from and writing to *file descriptors (fd's)*.
- Once we establish a connection and setup the fd's, we can send and receive data over those file descriptors.



Review: Telnet

- **`telnet`** is a network protocol for text-based communication
- Can run via: **`$ telnet <host> <port>`** to create a connection to the specified user
- Will be useful for this activity (as well as **`proxylab`**)!!

Activity: Telephone

Activity

- Download this week’s handout from the *Schedule* page.
- **Get into groups of 3-4 people!**
  - Just open up the source code under **`telephone.c`**.
  - We’ll take each component incrementally together

```
$ wget https://www.cs.cmu.edu/~213/activities/f25-rec11.tar
$ tar -xvf f25-rec11.tar
$ cd f25-rec11
```

Objective: Telephone Game

- Our goal is to create a player in the telephone game
- We should be able to:
  1. Receive messages from a person
  2. Pass along the message to a specified person
  3. Know when to stop sending messages
- We communicate through a network!

Brainstorm

- What components do we need to implement to implement the telephone game? Try to think in “networking terms.”
  - Connections?
  - File descriptors?
  - Other routines?

Component Roadmap

We will generally follow this roadmap!

1. Listen for any incoming connections
2. Connect to an incoming connection
3. Read messages from the accepted connection
4. Parse these messages and handle them accordingly
    a. FORWARD, STOP, and General messages

Phase 1: Listening for Connections

- In your groups, implement the component of setting up a file descriptor to listen for incoming connections.
- Take a moment to get familiar with the `csapp` library!

Phase 1: Solution

- We want to use **`open_listenfd(argv[1])`**, where the first argument holds the port.
- How can we test for its correctness?
  - Use verbose print statements to check for error/success!
  - What indicates failure?

Phase 2: Accepting a Connection

- In your groups, implement the accepting of any incoming connections!
- Keep in mind: how will we know if a connection is requested?

Phase 2: Solution

- We want to use **`accept(listenfd,…)`** , which sets up a file descriptor associated with out connection.
- How can we test its correctness?
  - Use **`telnet`** to attempt a connection!
  - Use verbose printing to report success.

Phase 3: Reading Inputs

- In your groups, implement the reading inputs from the connection we just accepted!
- Hint: use the **`RIO`** (robust I/O) package in **`CSAPP`**

Phase 3: Solution

- We want to:
  1. Initialize a RIO object via `rio_readinitb(...)`
     a. What fd should we associate with the RIO object?
  2. Fill the line buffer using `rio_readlineb(...)`
- How can we test its correctness?
  - ○ Use `telnet` to connect + send a message!
  - ○ Print the received message and check for consistency

Phase 4: Handling General Messages

- In your groups, implement sending general messages to some (later specified) destination.
- Hint: we want to send the message via **`outputfd`**
- How can we test this?
  - Since the default **`outputfd`** is **`STDOUT_FILENO`**, see if we get the appropriate prints.

Phase 5: Handling FORWARD

- In your groups, implement the FORWARD state.
- Remember that an user should only receive one *valid* FORWARD message, which defines the destination user to write to.
- Note: We provide you with a parsing function at the top of the file!

Phase 5: Solution

- We are given a host and port to connect to *(parse from input)*
- Open a file descriptor to this destination user
  - Via **`open_clientfd(host,port)`**
- Update **`outputfd`** and **`forwarding_state`**
- How can we test this?
  - Use 2 terminals and send messages via **`telnet`**!

Phase 6: Handling STOP

- In your groups, implement the STOP state.
- Remember that an user should also pass on the STOP message (so that other users can stop), before stopping themselves.
- How can we test this?
  - Same test as forward, but make sure that all instances terminate!

Phase 7: Cleanup

- Close any open file descriptors!
- What could go wrong if we don’t?

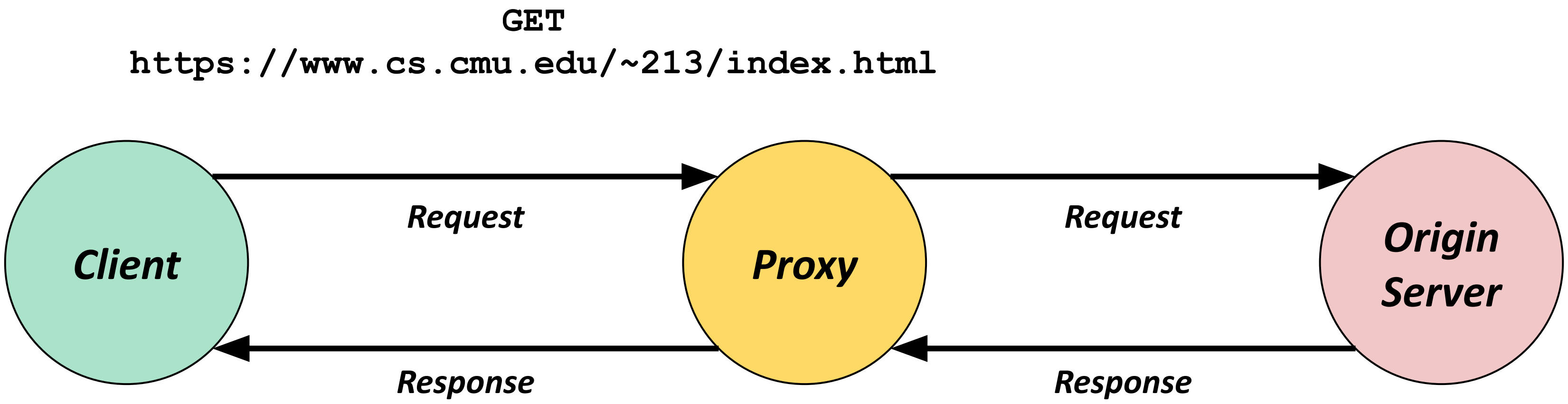
Connection to `proxylab`?

- Why is this activity relevant?
- Well, we just built a simple **proxy**!
  - We are the *intermediary* between the person sending a message and the person that is waiting to receive a message from us
- A lot of the testing behavior will be extremely useful for **`proxylab`** as well.

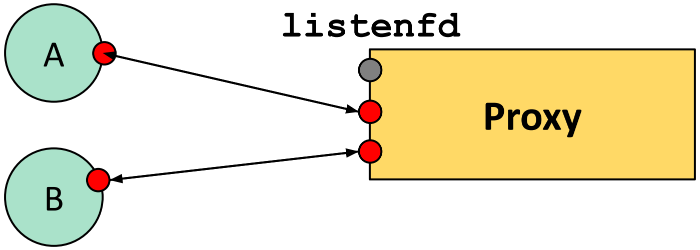
Proxy Lab

What is a proxy?

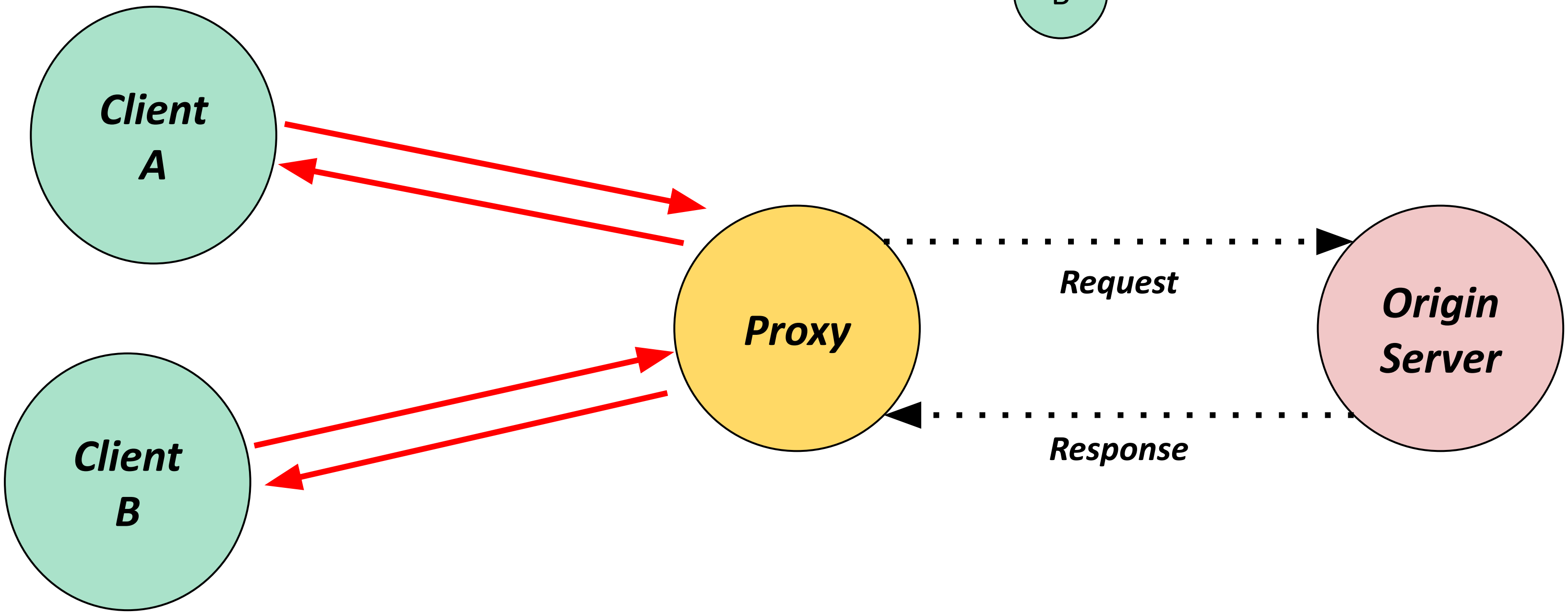
- Proxy sits between client and the server the client wants to talk to.



- Can do useful things in-between (not assessed in **`proxylab`**):
  - Caching, logging, anonymization, transcoding, etc.

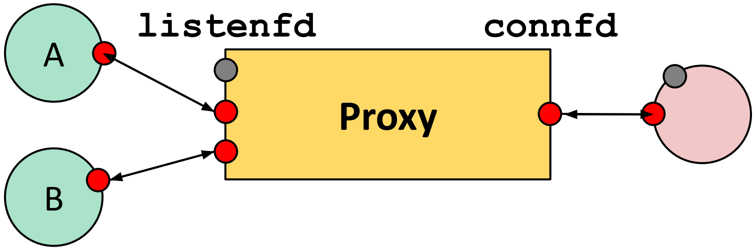


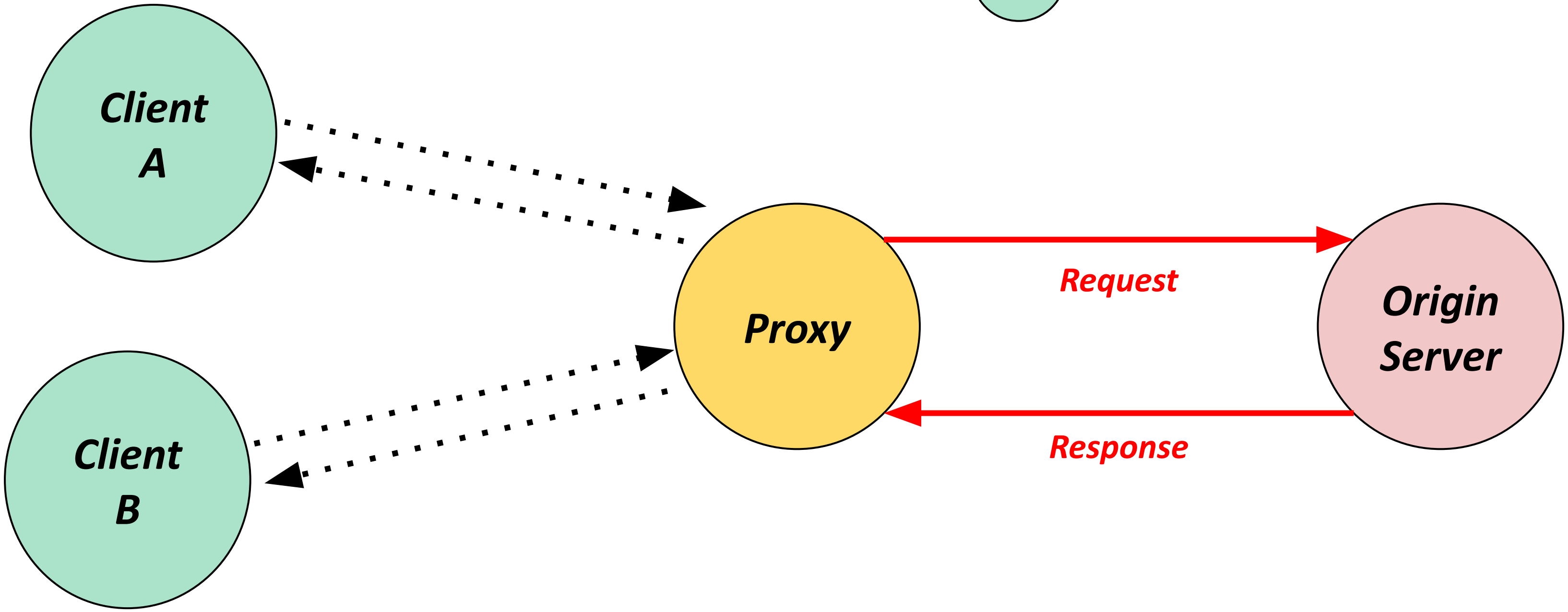
Proxies are *Servers*



- Proxies need to listen for and handle requests from clients.
- Ideally, they should be able to do so for multiple clients at the same time!

Proxies are *Clients*





- Proxy parses headers in client's request to figure out which server to contact.
- Then connects to a server to get the data the client asked for.

Proxy Lab: Overview

- You'll implement a web proxy like the one on the previous slide!

Part I

- Accept connections from clients.
- Parse headers to determine origin server (see **`http_parser.h`**)
- Fetch data from the server, and forward response back to the client.

Part II

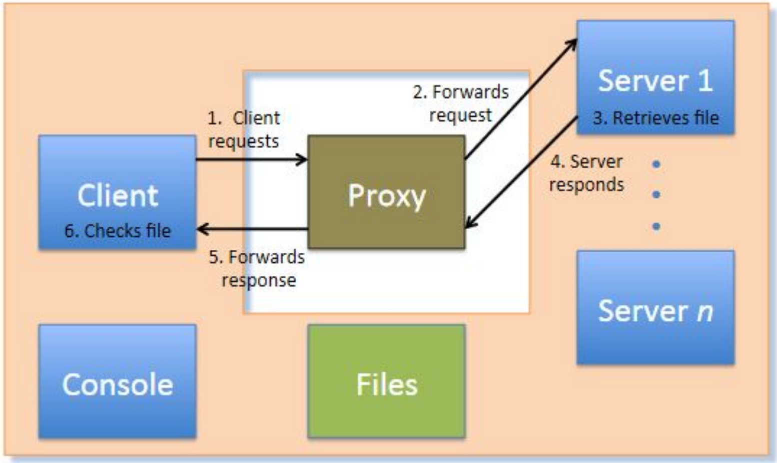
- Handle *concurrent requests* with POSIX threads (Tuesday's lecture)!

Proxy Lab: Getting Started

- Worth 4% of course grade.
- You have one week to complete the lab. *Start early!*
- Resources:
  - *Network Programming* Lectures
  - Textbook: Chapter 11
  - Write-up!
- Make sure you're familiar with the provided libraries before you start:
  - **`csapp.h`** – Networking Wrappers, **`rio`**
  - **`http_parser.h`** – For parsing *requests*

Debugging Proxy Lab: PxyDrive !

- Testing framework for Proxy Lab
  - Autolab uses it to grade your code…
  - You can use it to debug!
- PxyDrive workflow:
  - Generate text and binary data
  - Create server(s)
  - Build *transactions*
  - Trace transactions to inspect headers and response data.



PxyDrive Demo

- Let’s run through some of the features of PxyDrive!
- If you want to follow along:

```
$ wget http://www.cs.cmu.edu/~213/activities/rec11.tar
$ tar -xvf rec11.tar
$ cd pxydrive-tutorial
```

PxyDrive – Getting Started

- Run the REPL and try entering some commands!

```
$ ./pxy/pxydrive.py
> help
  #         XXXXXXXX                    Rest of line treated as comment
  check     ID [CODE]                   Make sure request ID handled
properly and generated expected CODE
  delay     MS                          Delay for MS milliseconds
```

- Running with a specific proxy:

```
$ ./pxy/pxydrive.py -p ./proxy-ref
Proxy set up at nurseshark.ics.cs.cmu.edu:12168
>
```

PxyDrive – Tutorial 1

- Take a look at **`s01-basic-fetch.cmd`**
- Then try running the commands yourself in the REPL:

```
$ ./pxy/pxydrive.py -p ./proxy-ref
> generate data1.txt 1k
...
```

- **`generate data1.txt 1k`** – Generates a 1K text file called **`data1.txt`**
- **`serve s1`** – Launches a server called **`s1`**
- **`fetch f1 data1.txt s1`** – Fetches **`data1.txt`** from server **`s1`**, in a transaction called **`f1`**
- **`trace f1`** – Traces the transaction **`f1`**
- **`check f1`** – Checks the transaction **`f1`**

PxyDrive – Tutorial 1

- Try running the trace again with the **`-f`** flag:

```
$ ./pxy/pxydrive.py -f s01-basic-fetch.cmd -p ./proxy-ref
```

- Can you identify:
  - **`GET`** command
  - Host header? Other headers?
  - Request from *client to proxy*
  - Request from *proxy to server*
  - Response by *server to proxy*
  - Response by *proxy to client*

PxyDrive – Tutorial 1

- Let’s try a different trace

```
$ ./pxy/pxydrive.py -f s02-basic-request.cmd  -p ./proxy-ref
```

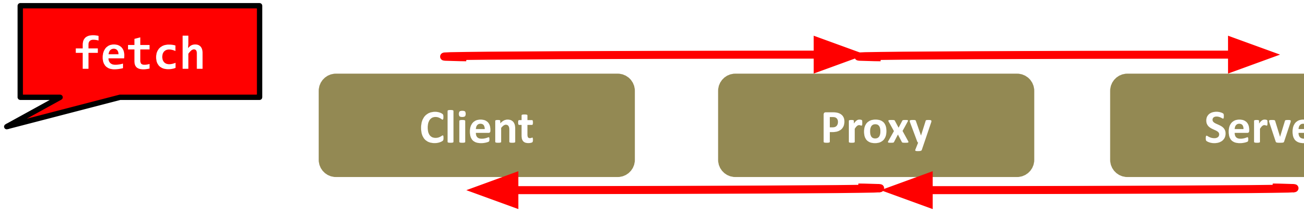
- You should get a different output.
- Why do we see “Response NOT sent by server” after the first `trace` command?

PxyDrive – Tutorial 1

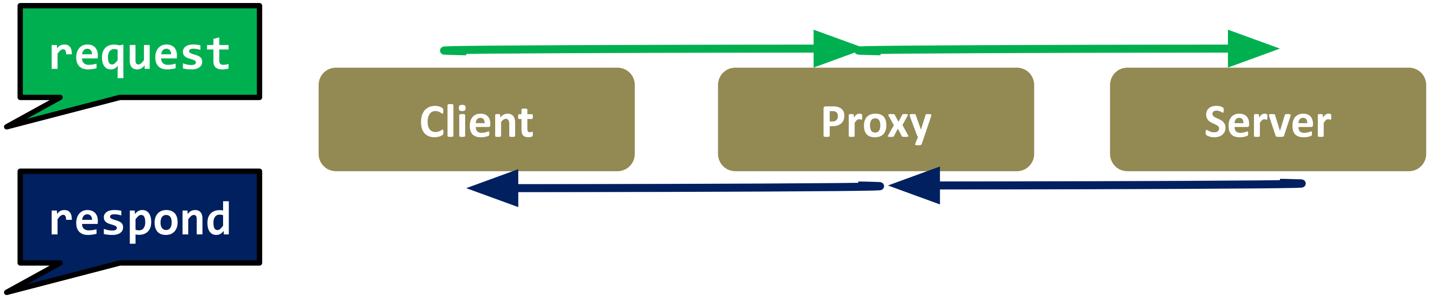
- **`generate data1.txt 1K`**
- **`serve s1`**
- **`request r1 data1.txt s1`** – Requests **`data1.txt`** from server **`s1`**, in a transaction called **`r1`**
- **`trace r1`**
- **`respond r1`**
- **`wait *`** – Allow server to respond to the transaction **`r1`**
- **`trace r1`**
- **`check r1`** – Checks the transaction **`r1`**

PxyDrive – Tutorial 1

- The **`fetch`** command makes the server *immediately* respond to a request.
- All steps of a transaction are complete after a fetch.



- The **`request`** command does not complete a transaction.
- A request needs a **`respond`** to complete its transaction.



PxyDrive – Tutorial 2

- Let’s see what happens with a buggy proxy…

```
$ ./pxy/pxydrive.py -f s01-basic-fetch.cmd -p ./proxy-corrupt
```

- What happens?

```
# Make sure it was retrieved properly
> check f1
ERROR: Request f1 generated status 'error'.  Expecting 'ok'
(Mismatch between source file ./source_files/random/data1.txt and
response file ./response_files/f1-data1.txt starting at position
447: 'F' (hex 0x46) != 'G' (hex 0x47))
> quit
ERROR COUNT = 1
```

- Proxy clobbers response from server.

PxyDrive – Tutorial 3

- Let’s try another buggy proxy…

```
$ ./pxy/pxydrive.py -f s01-basic-fetch.cmd -p ./proxy-strip -S 3
```

- **-S** denotes strictness level.
- What happens?

```
Response status: bad_request (Missing Request-ID header)
  Source file in ./source_files/random/data1.txt
Request status:  bad_request (Bad request)
  Result file in ./response_files/f1-status.html
```

- Proxy does not correctly forward **Request-ID** header from client to server.

PxyDrive – Tutorial 4

- Let's try another buggy proxy…

```
$ ./pxy/pxydrive.py -f s03-overrun.cmd -p ./proxy-overrun
```

- Is the error message helpful?

```
ERROR: Request f1 generated status 'error'.  Expecting 'ok'
(Socket closed after reading 106386/200000 bytes)
```

- Let's use **gdb**!

PxyDrive – Multi-Window Debugging

- Use **gdb** to run **proxy-overrun** in a fresh window.

```
$ gdb ./proxy-overrun
(gdb) run <port>
```

- Now run **pxydrive** in another window (same Shark):

```
$ ./pxy/pxydrive.py -P localhost:<port> -f s03-overrun.cmd
```

- When debugging **proxylab**, run **./port-for-user.pl** to get a unique port number, so your debugging doesn't conflict with other students.

PxyDrive – Multi-Window Debugging

- Multi-Window debugging is helpful even without **gdb**:

```
$ ./proxy-overrun <port>
```

```
$ ./pxy/pxydrive.py -P localhost:<port> -f s03-overrun.cmd
```

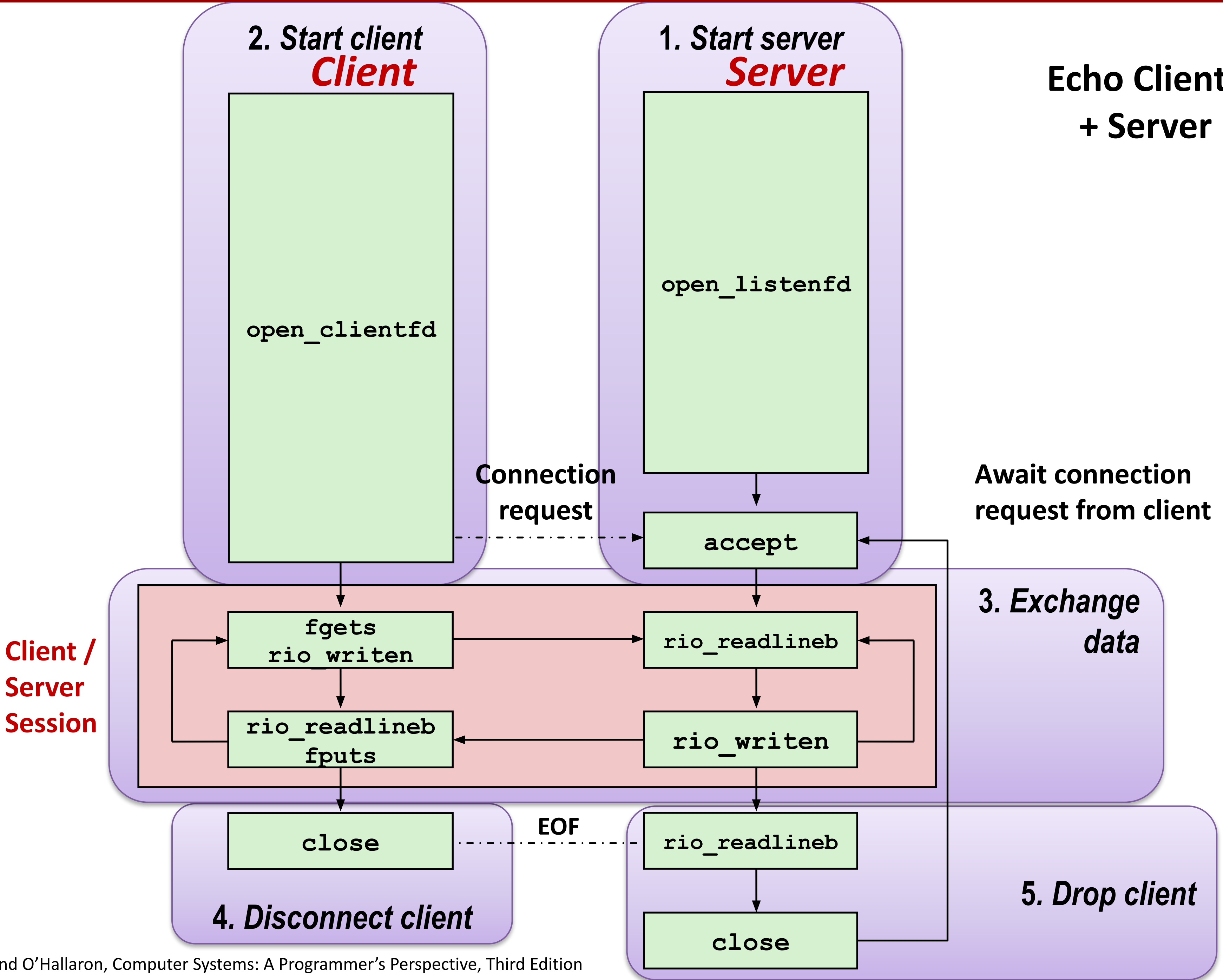
- Can redirect output of proxy to a file.
- If you include thread IDs in your print statements, can use **awk** to split thread outputs to different files for easier debugging.

Wrapping Up

- **`proxylab`** is due *November 25th*
- ***Written 10*** due *November 20th*
- Code reviews for **`tshlab`**
- Good luck on **`proxylab`** :-)

The End

Echo Client + Server


2. Start client
Client
open_clientfd
1. Start server
Server
open_listenfd
Connection request
accept
Await connection request from client
Client / Server Session
fgets
rio_writen
rio_readlineb
rio_readlineb
fputs
rio_writen
3. Exchange data
close
EOF
rio_readlineb
close
4. Disconnect client
5. Drop client