CS244A Review Session #1

Topics: Socket Programming
Assignment #1
Coding Guidelines

Friday, January 11, 2007

Clay Collier

(based off slides by Ari Greenberg & William Chan)

Announcements

- Assignment questions:
  - Check FAQs (on the web page & the assignment page)
  - Look at RFC 1945
  - Post on the newsgroup: su.class.cs244a

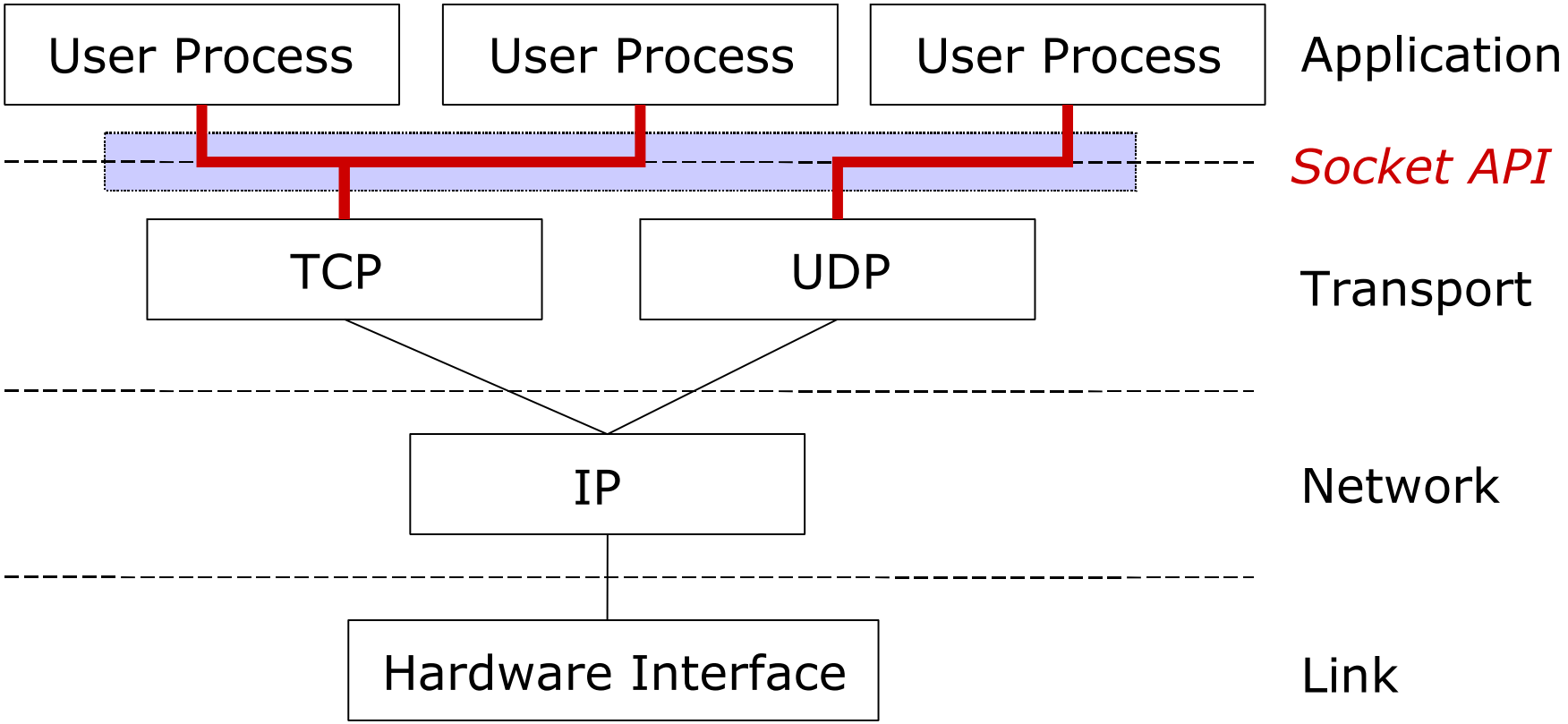
General Comments

- Read the project guidelines document
- Read through Stevens Chapter 3 & 4
- Kurose & Ross chapter 2
- Use Linux machines: *myth*, *bramble*, *hedge*, *vine*
- Debugging: *make*, *gdb, valgrind* and *telnet*
- Some links to socket programming references are found on the assignment page

The Socket API

- A collection of system calls to write a networking program at user-level.
- API is similar to Unix file I/O in many respects: open, close, read, write.
  - Data written into socket on one host can be read out of socket on other host
  - Difference: networking has notion of client and server.

Sockets and the TCP/IP Suite



Socket Parameters

A socket connection has 5 general parameters:

- The protocol
  - Example: TCP, UDP etc.
- The local and remote address
  - Example: 171.64.64.64
- The local and remote port number
  - Needed to determine to which application packets are delivered
  - Some ports are reserved (e.g. 80 for HTTP- see /etc/services)
  - Root access require to listen on port numbers below 1024

Selecting the Protocol
Connection-Oriented vs Connectionless

- Connection oriented (streams)

```
sd = socket(PF_INET, SOCK_STREAM, 0);
```

- Connectionless (datagrams):

```
sd = socket(PF_INET, SOCK_DGRAM, 0);
```

- For the internet (**`PF_INET`**) this corresponds to TCP and UDP respectively
- **`socket()`** returns a socket descriptor, an `int` similar to a file descriptor.

Selecting the Remote Address and Port
Example: A Client using TCP

- Use **`connect()`** on a socket that was previously created using **`socket()`**:

```
err = connect(int sd, struct sockaddr*,
              socklen_t addrlen);
```

- Remote address and port are in struct sockaddr:

```
struct sockaddr_in {
  u_short sa_family;
  u_short sin_port;
  struct in_addr sin_addr;
  char sin_zero[8];
};
```

A Connection-oriented client

- `socket()` create the socket descriptor
- `connect()` connect to the remote server.
- `read(),write()` communicate with the server
- `close()` end communication by closing socket descriptor

A Connection-oriented server

- `socket()` create the socket descriptor
- `bind()` associate the local address
- `listen()` wait for incoming connections from clients
- `accept()` accept incoming connection
- `read()`,`write()` communicate with client
- `close()` close the socket descriptor

Server Operation

- The socket returned by `accept()` is not the same socket that the server was listening on!
- A new socket, bound to a random port number, is created to handle the connection
- New socket should be closed when done with communication
- Initial socket remains open, can still accept more connections

Main structures

```
struct sockaddr {
    u_short sa_family;
    char sa_data[14];
};
```

Generic address
<sys/socket.h>

```
struct in_addr {
    u_long s_addr;
};
```

IP address
<netinet/in.h>

```
struct sockaddr_in {
    u_short sa_family;
    u_short sin_port;
    struct in_addr sin_addr;
    char sin_zero[8];
};
```

TCP/UDP address
(includes port #)
<netinet/in.h>

Supporting function calls

`gethostbyname()`	get address for given host name (e.g. 171.64.64.64 for name "cs.stanford.edu");
`getservbyname()`	get port and protocol for a given service e.g. ftp, http (e.g. "http" is port 80, TCP)
`getsockname()`	get local address and local port of a socket
`getpeername()`	get remote address and remote port of a socket

Byte order

- Order of bytes in a 32-bit word depends on hardware architecture
- Big-endian vs little-endian
- Network byte order used for portability
- Use appropriate functions in your assignment
  `htonl(), htons(), ntohl(), ntohs()`
  – Don't forget to use them for `connect()` etc.
- For documentation see man page
  `man 3 byteorder`

Other useful stuff…

- Address conversion routines
  - Convert between system's representation of IP addresses and readable strings (e.g. "171.64.64.64")

```
unsigned long inet_addr(char* str);
char * inet_ntoa(struct in_addr inaddr);
```

- Important header files:
  `<sys/types.h>`, `<sys/socket.h>`, `<netinet/in.h>`, `<arpa/inet.h>`
- man pages (section 2)
  - `socket`, `accept`, `bind`, `listen`
- That's it for sockets—any questions?

Handling Multiple Connections

- Your proxy must handle multiple concurrent connections
- Can be implemented however you wish- threading, fork() + exec()
- Don't leak threads/processes- your proxy will grind to a halt over time
- Enforce a reasonable connection limit- document how/why in README.
- Look for tutorial info online

Stevens Chapter 5 – TCP Client/Server Example

- Simple echo server
- Shows socket system calls as they would be used in real client/server programs
- Your proxy will include both client and server components
- Pg 121-125

Assignment #1

- Due Friday January 18, 12 pm
- Please post questions to the newsgroup su.class.cs244a
- Should take you <= 15-20 hrs: If not, reconsider taking this class
- Office hours posted on web site
- Download testing script and sample Makefile from assignment page (use `wget` if working from a Stanford machine)

Testing

- For testing, run proxy_tester.py:
  - We'll do more testing for actual grading
  - Free to add additional tests to the list of URL's in the script
- Public tests test basic functionality; our private tests will include tests of:
  - Error handling
  - How well you conformed to the RFC specs
  - Concurrent connection handling

Overview

- proxy <port #>
- Specifications on assignment page
- Should conform to RFC 1945
- You need to parse incoming HTTP requests to make sure that they are legal
- "Be lenient about what you accept, strict about what you produce" (but don't discard other people's data!)
- Pay attention to programming style!
  - modularity, code reuse, comments, readability
- Make sure your Makefile has 'tabs' and not 'spaces'

Deliverables

- Read the assignment and programming guidelines
- Tarball containing the following:
  - README
    - Architecture, design decisions, comments (short, please!)
    - Any extensions that you added to the proxy
  - Your source code & Makefile
- Submit everything via the script on the webpage before 12:00 PM
- Submission link is on web site.

HTTP

- Check out Kurose and Ross on HTTP
- Read the RFC
- You'll need to send both HTTP requests and responses- you're acting as both a server (to the webrowser/client) and a client (to the remote web server)
- Take a look at the example of communicating with a HTTP server via telnet
- Also check out "HTTP Made Really Easy" (link on the assignment page)

Miscellany

- Some things to watch out for…
  - Remember that you may be transferring both binary and ASCII data
  - Memory leaks- use Valgrind to ensure that you are freeing all your allocated blocks, or your server will gradually grind to a halt
  - Look at the HTTP response codes- things like 404 errors should be passed back to the client

Coding Guidelines

- 10% of assignment grade is reserved for "style and design"
  - In essence: Readability, efficiency, and maintainability
- Read over hints and suggestions on web page
  - Most of this is common sense, and should be second nature to you by now
  - There are some examples of both good and bad code in the coding guidelines—be sure to take a look!

External Libraries

- No 3rd party libraries
- One exception- you may (but aren't required) use the ustr library for string processing
  - May be easier/cleaner, may be a pain if you're already familiar with C strings
- Must be ANSI C, but can use Gnu C99 extensions (try -std=gnu99)

Some easy ways to lose points

- Global variables
  - It's ok to have global options like debugging output
  - There really isn't much need for global variables in assignment 1
- Copy and paste code between functions
- Crash on malformed HTTP requests
- No comments

Some more ways to lose points

- Dump everything into main(), and maybe a token helper function or two
  - Functions in general really shouldn't be longer than say, 150 lines or so, especially in this program
- Write non-portable code
  - Yes, we will check for this! Be sure to check return values of system calls, **endianness** assumptions, etc.
- Use lines longer than 80 characters
- Don't indent, or use an indention < 4 or > 8 characters

Even more ways to lose points

- Not watching out for partial sends / recvs
  - May want to write wrapper
- Blindly sending out malformed requests
  - Be lenient about what you accept, strict about what you emit
- Statically allocate giant buffers
- Memorize the operator precedence tables, and expect that everyone else has too

Ways to make your life easier

- Document assumptions in your code
  - Make liberal use of assert()
    - Make it clear what preconditions/postconditions *must* hold for your code to work
      - Catch incorrect assumptions before they trigger other problems in your code…
    - If you haven't used this before, start now!
      - This can save you some time on HW#1 and HW#2, and probably a ton of time on HW#3
- Fix the warnings
  - Your code should compile silently with -Wall