CIS 330: UNIX and C/C++

Network Servers

Prof. Kevin Butler
Winter 2012
Today

• Network programming
  ‣ server-side programming
Servers

• Pretty similar to clients, but with additional steps
  ‣ there are seven steps:
    1. figure out the address and port on which to listen
    2. create a socket
    3. **bind** the socket to the address and port on which to listen
    4. indicate that the socket is a **listen**ing socket
    5. **accept** a connection from a client
    6. **read** and **write** to that connection
    7. **close** the connection
Accepting a connection from a client

Step 1. Figure out the address and port on which to listen.

Step 2. Create a socket.

Step 3. **Bind** the socket to the address and port on which to listen.

Step 4. Indicate that the socket is a **listening** socket.
• Servers can have multiple IP addresses
  ‣ “multihomed”
  ‣ usually have at least one externally visible IP address, as well as a local-only address (127.0.0.1)

• When you bind a socket for listening, you can:
  ‣ specify that it should listen on all addresses
    • by specifying the address “INADDR_ANY” -- 0.0.0.0
  ‣ specify that it should listen on a particular address
bind() 

• The “bind()” system call associates with a socket:
  ‣ an address family
    • AF_INET: IPv4
    • AF_INET6: IPv6
  ‣ a local IP address
    • the special IP address INADDR_ANY ("0.0.0.0") means “all local IP addresses of this host”
  ‣ a local port number
The “listen( )” system call tells the OS that the socket is a listening socket to which clients can connect

- you also tell the OS how many pending connections it should queue before it starts to refuse new connections
  - you pick up a pending connection with “accept( )”
- when listen returns, remote clients can start connecting to your listening socket
  - you need to “accept( )” those connections to start using them
Server socket, bind, listen

- see server_bind_listen.cc
Accepting a connection from a client

Step 5. Accept a connection from a client.

Step 6. read( ) and write( ) to the client.

Step 7. close( ) the connection.
The “accept( )” system call waits for an incoming connection, or pulls one off the pending queue

- it returns an active, ready-to-use socket file descriptor connected to a client
- it returns address information about the peer
  - use inet_ntop( ) to get the client’s printable IP address
  - use getnameinfo( ) to do a reverse DNS lookup on the client
Server accept, read/write, close

• see server_accept_rw_close.cc
Something to note...

• Our server code is not concurrent
  ‣ single thread of execution
  ‣ the thread blocks waiting for the next connection
  ‣ the thread blocks waiting for the next message from the connection

• A crowd of clients is, by nature, concurrent
  ‣ while our server is handling the next client, all other clients are stuck waiting for it
Sequential

pseudocode:

listen_fd = Listen(port);
while(1) {
    client_fd = accept(listen_fd);
    buf = read(client_fd);
    write(client_fd, buf);
    close(client_fd);
}

• look at `echo_sequential.cc`
Whither sequential?

• **Benefits**
  ‣ super simple to build

• **Disadvantages**
  ‣ incredibly poorly performing
    • one slow client causes all others to block
    • poor utilization of network, CPU
Concurrency with processes

- The **parent** process blocks on `accept()` waiting for a new client to connect
  - when a new connection arrives, the parent calls `fork()` to create a **child** process
  - the child process handles that new connection, and `exit()`'s when the connection terminates

- Remember that children become “zombies” after death
  - option a) parent calls `wait()` to “reap” children
  - option b) use the double-fork trick
Graphically

server
Graphically

client

server
Graphically
Graphically

client

server

fork(x)

child

fork( )child

server
Graphically

client -> server
server -> server

fork() grandchild
Graphically

client

server

child exit( )’s / parent wait( )’s
Graphically

client → server

server

parent closes its client connection
Graphically
Graphically

client → server

fork(child) → child

fork(grandchild) → grandchild

exit( )
Graphically

client → server

server

client → server
Graphically
Concurrent with processes

- look at `echo_concurrent_processes.cc`
Concurrent processes

• **Benefits**
  ‣ almost as simple as sequential
    • in fact, most of the code is identical!
  ‣ parallel execution; good CPU, network utilization

• **Disadvantages**
  ‣ processes are heavyweight
    • relatively slow to fork
    • context switching latency is high
  ‣ communication between processes is complicated
How slow is fork?

• run `forklatency.cc`
Implications?

- **0.18 ms** per fork
  - maximum of \((1000 / 0.18) = 5,555.5\) connections per second
  - 0.5 billion connections per day per machine
    - fine for most servers
    - too slow for a few super-high-traffic front-line web services
      - Facebook serves \(O(750\) billion\) page views per day
      - guess \(~1-20\) HTTP connections per page
      - would need 3,000 -- 60,000 machines just to handle fork\((), \)
        i.e., without doing any work for each connection!
Concurrency with threads

- A single *process* handles all of the connections
  - but, a parent *thread* forks (or dispatches) a new thread to handle each connection
  - the child thread:
    - handles the new connection
    - exits when the connection terminates
Graphically

server

accept( )
Graphically

client

connect

accept()

server
Graphically

client

pthread_create()

server
Graphically
Graphically

client → pthread_create() → server

client → server
Graphically
Concurrent with threads

- look at `echo_concurrent_threads.cc`
Concurrent threads

• Benefits
  ‣ straight-line code, line processes or sequential
    • still the case that much of the code is identical!
  ‣ parallel execution; good CPU, network utilization
    • lower overhead than processes
  ‣ shared-memory communication is possible

• Disadvantages
  ‣ synchronization is complicated
  ‣ shared fate within a process; one rogue thread can hurt you badly
How fast is `pthread_create`?

- run `threadlatency.cc`
Implications?

- **0.021 ms** per thread create; 10x faster than process forking
  - maximum of \( (1000 / 0.021) = \approx 50,000 \) connections per second
  - 4 billion connections per day per machine
    - much, much better

- But, writing safe multithreaded code is serious voodoo
OSI Layer Model

- Application protocols
  - the format and meaning of messages between application entities
  - e.g., HTTP is an application level protocol that dictates how web browsers and web servers communicate
- HTTP is implemented on top of TCP streams
Packet Encapsulation

Packet encapsulation -- same as before!

- Ethernet header
- IP header
- TCP header
- HTTP header

HTTP payload (e.g., HTML page)
TCP payload
IP payload

Destination address
Source address
Data

Ethernet header
Ethernet payload
Let’s dive down into HTTP

I’d like “foo.html”

Found it, here it is: (foo.html)

• A client establishes one or more TCP connections to a server
  ‣ the client sends a request for a web object over a connection, and the server replies with the object’s contents
  ‣ we have to figure out how let the client and server communicate their intentions to each other clearly
  ‣ we have to define a protocol
HTTP is a “protocol”

- **Protocol**: the rules governing the exchange of messages, and the format of those messages, in a computing system
  - what messages can a client exchange with a server?
    - what do the messages mean?
    - what are legal replies to a message?
    - what is the syntax of a message?
  - what sequence of messages is legal?
    - how are errors conveyed?
  - A protocol is (roughly) the network equivalent of an API
HTTP

• **Hypertext transport protocol**
  ‣ a request / response protocol
  ‣ a client (web browser) sends a request to a web server
    • the server processes the request, sends a response
  ‣ typically, a request asks the server to retrieve a resource
    • a resource is an object or document, named by a URI
  ‣ a response indicates whether the server succeeded
    • and, if so, it provides the content of the requested response
An HTTP request

- [METHOD] [request-uri] HTTP/[version]\r\n
- [fieldname1]: [fieldvalue1]\r\n
- [fieldname2]: [fieldvalue2]\r\n
- [...]

- [fieldnameN]: [fieldvalueN]\r\n
- \r\n
- [request body, if any]

let's use “nc” to see a real request

HTTP methods

- There are three commonly used HTTP methods:
  - **GET**: “please send me the named document”
  - **POST**: “I’d like to submit data to you, such as form content”
  - **HEAD**: “send me the headers for the named object, but not the object. (I’d like to see if my cached copy is still valid.)”

- There are several rarely used methods:
  - PUT, DELETE, TRACE, OPTIONS, CONNECT, PATCH, ...
    - **TRACE**: “if there are any proxies or caches in between me and the server, please speak up!”
HTTP versions

• Most browsers and servers speak HTTP/1.1
  ‣ “version 1.1 of the HTTP protocol”
    • [http://www.w3.org/Protocols/rfc2616/rfc2616.html](http://www.w3.org/Protocols/rfc2616/rfc2616.html)
  ‣ introduced around 1996 to fix shortcomings of HTTP/1.0
    • better performance, richer caching features, better support for multi-homed servers, and much more
    • more complicated to implement than HTTP/1.0
Client headers

• The client can provide zero or more request “headers”
  ‣ they provide information to the server, or modify how the server should process the request

• You’ll encounter many in practice
  ‣ Host: the DNS name of the server [why?]
  ‣ User-Agent: an identifying string naming the browser [why?]
  ‣ Accept: the content types the client prefers or can accept
  ‣ Cookie: an HTTP cookie previously set by the server
GET /foo/bar.html HTTP/1.1
Host: futureproof.cs.washington.edu:5555
User-Agent: Mozilla/5.0 (Macintosh; U; Intel Mac OS X 10_6_7; en-us) AppleWebKit/533.21.1 (KHTML, like Gecko) Version/5.0.5 Safari/533.21.1
Accept: application/xml,application/xhtml+xml,text/html;q=0.9,text/plain;q=0.8,image/png,*/*;q=0.5
Accept-Language: en-us
Accept-Encoding: gzip, deflate
Cookie: __utma=59807807.1547453334.1214335349.1301330421.1301339949.30; __utmz=59807807.1300728257.27.14.utmcsr=google|utmccn=(organic)|utmcmd=organic|utmctr=csgordon@u.washington.edu;
__utma=80390417.1521666831.1201286098.1302710464.1302717901.34;
__utmz=80390417.1301950604.31.15.utmcsr=cs.washington.edu|utmccn=(referral)|utmcmd=referral|utmct=/education/courses/cse333/11sp/;
__qca=P0-1872143622-1294952393928
Connection: keep-alive
An HTTP response

- HTTP/[version] [status code] [reason]\r\n- [fieldname1]: [fieldvalue1]\r\n- [fieldname2]: [fieldvalue2]\r\n- [...]\n- [fieldnameN]: [fieldvalueN]\r\n- \r\n- [response body, if any]

*let’s use “telnet” to see a real response*
Status codes, reason phrase

• Code: a computer-readable outcome of the request
  ‣ three digit integer; first digit identifies the response category
    • 1xx: some kind of informational message
    • 2xx: success of some kind
    • 3xx: redirects the client to a different URL
    • 4xx: the client’s request contained some error
    • 5xx: the server experienced an error

• Reason phrase: human-readable explanation
  ‣ e.g., “OK” or “Moved Temporarily”
Common status lines

• HTTP/1.1 200 OK
  ‣ the request succeeded, the requested object is sent

• HTTP/1.1 404 Not Found
  ‣ the requested object was not found

• HTTP/1.1 301 Moved Permanently
  ‣ the object exists, but its name has changed
  ‣ the new URL is given in the “Location:” header

• HTTP/1.1 500 Server Error
  ‣ the server had some kind of unexpected error
Server headers

• The server can provide zero or more request “headers”
  ‣ they provide information to the client, or modify how the client should process the response

• You’ll encounter many in practice
  ‣ **Server**: a string identifying the server software
  ‣ **Content-Type**: the type of the requested object
  ‣ **Content-Length**: size of requested object [why?]
  ‣ **Last-Modified**: a date indicating the last time the request object was modified [why?]
Example

HTTP/1.1 200 OK
Date: Fri, 27 May 2011 17:05:53 GMT
Server: Apache/2.2.19 (Fedora)
Last-Modified: Fri, 27 May 2011 17:04:51 GMT
ETag: "2740640-52-4a444ef9392c0"
Accept-Ranges: bytes
Content-Length: 82
Content-Type: text/html
Content-Language: en
X-Pad: avoid browser bug

<html><body>
<font color="chartreuse" size="18pt">Awesome!!</font>
</body></html>
Cool HTTP/1.1 features

• Persistent connections
  ‣ establishing a TCP connection is costly
    • multiple network “round trips” just to set up the TCP connection
    • TCP has a feature called “slow start”; slowly grows the rate at which a TCP connection transmits to avoid overwhelming networks
  ‣ a web page consists of multiple objects, and a client probably visits several pages on the same server
    • bad idea: separate TCP connection for each object
    • better idea: single TCP connection, multiple requests
  • try it on www.cs.uoregon.edu