THE SOCKETS API, PART 2

Internet Protocols

CSC / ECE 573

Fall, 2005

N. C. State University

Today's Lecture

I. TCP Clients-Server Interaction: Iterative Request Processing

II. Ways to Handle Concurrency in Servers

III. Server Design Issues

IV. Using Raw Sockets

Announcements

HW3 Part 2

Problems 2 and 4 revised this afternoon, new data files

After class: in east wing classroom

TCP SERVERS: ITERATIVE

Client-Server Interaction: TCP

Step #	Server	Client
1	Create socket with `socket()`	Create socket with `socket()`
2	Bind socket to <u>local</u> port and IP address with `bind()`	
3	Place socket into "passive" mode with `listen()`	

Client-Server Interaction: TCP (cont'd)

Step #	Server	Client
4	Wait for a connection request from a client using `accept();` creates socket, and binds <u>remote</u> address and port to socket	
5		Bind socket to <u>remote</u> IP address and port with `connect();` (this connection request may also serve as the request to the server)

Client-Server Interaction: TCP (cont'd)

Step		
6	Execute `send()` to respond to client's request (execute as many times as needed)	
7	Close the socket with `close()` and go back to step 4	Get data from the connection using `recv()`
8		Repeat step 7 until `recv()` returns SOCKET_ERROR, indicating no more data
9		Close the socket with `close();`

Step #3: Establish a Connection Queue

```
int  listen(int s, int backlog);
```

- Only used for stream sockets
- Used by connection-oriented servers to place a socket in passive mode
  - makes it ready to accept incoming connections
  - the remote port # / IP address of the socket = any port, any IP address
- Allows **backlog** pending connections to be waiting for **accept()**

Step #4: Accept an Incoming Connection Request

```
int  accept(int new_s, struct sockaddr *new_adr,
          int addrlen);
```

- Functions
  1. blocks until a connection request arrives
  2. removes the next connection request from queue (or waits til one arrives)
  3. creates a new socket
  4. binds remote address (from connection request) to socket

Step #5: Connecting To A Server

```
int  connect(int s, struct sockaddr *srvrsock,
          int srvrsocklen);
```

- Functions
  - uses 3-way handshake to establish connection (active open)
  - binds a remote address to a socket
  - chooses a local endpoint (IP address and port number) if the socket does not have one
- May fail (and returns error code). E.g., …
  - host does not have active service bound to port
  - connection timeout

Sending Data

```
int  write(int s, char* buf, int buflen);

int  send(int s, char* buf, int buflen,
          int flags);
```

- Flags control transmission
  - e.g., specify urgent data
- **write()** may not be able to write all **buflen** bytes (on a nonblocking socket)

Receiving Data

```
int  read(int s, char* buf, int buflen);

int  recv(int s, char* buf, int buflen,
          int flags);
```

- Flags control reception, e.g., get urgent data
- Reading is "stream-oriented"
  - # of bytes returned by **read()** may not = # of bytes specified by **write()** on the other end
- If other end closed the connection, and no more data to read, **read()** returns 0 to indicate EOF

Closing A Connection

```
int  close(int s);
```

- Actions
  - decrements reference count for socket
  - terminates communication gracefully and releases socket when reference count = 0 (why needed???)
  - any unread data from the other end will be discarded
- Problem: is other end of connection ready to terminate?

Partially Closing A Connection

```
int  shutdown(int s, int direction);
```

- Direction
  - 0 to close the input (reading) end
    - "I'm not listening anymore, but I have something more to say yak yak yak…" ?!
  - 1 to close the output (writing) end
    - "I have nothing more to say, but keep talking, I'm still listening"
  - 2 for both (same as `close()`)

TCP Data Transfer Example

- Client code
  - sends length of message first so server knows how much data to receive
  - then sends the data
- Server code
  - reads the message length
  - then reads that amount of data
- No error checking shown!

TCP Data Transfer: Client (producer)

```
u_short  msg_len;
char     msgout[MAXLEN] = "…some data here…";

msg_len = htons( strlen( &msgout ) );

/* send length of message to receiver */
write(s, (char *) &msg_len, sizeof(msg_len));

/* now send the message data – 1 write() will do it */
write(s, msgout, msg_len);
```


TCP Data Transfer: Server (consumer)

```
u_short  msg_len;
char     *ptr, msgin[MAXLEN];

ptr = (char *) &msg_len;

/* read length of msg */
for ( i=0; i < sizeof(msg_len); i+=n, ptr+=n )
    n = read( s, ptr, sizeof(msg_len)-i );

…more on next page…
```


TCP Data Transfer: Server (cont'd)

```
msg_len = ntohs(msg_len);
ptr = (char *) msgin;

/* read message data */
for ( i=0; i < msg_len; i+=n, ptr+=n )
      n = read(s, ptr, msg_len-i);
```

Possible Results

- For **msg_len**:
  - One transfer (2 bytes)
  - or two transfers (1+1)
- For **msgin**:
  - One transfer (msg_len bytes)
  - or two transfers (1+(msg_len-1)), or (2+(msg_len-2)),…
  - or three transfers (1+1+(msg_len-2)), or(1+2+(msg_len-3)), …
  - or…

CONCURRENCY IN SERVERS

Concurrency

- Servers are frequently bombarded with requests!
  - concurrent execution of request handling is necessary
- 3 ways to handle concurrent client requests
  1. the server process dynamically creates a "slave" process to handle each incoming connection request
  2. the server process dynamically creates a "slave" thread to handle each incoming connection request
  3. the server process polls the active connections using the **select()** function call and processes them

Concurrency Issues

- Overhead (memory, processing time) required
- Ease of programming
- Likelihood of programming errors
- Degree of control desired (execution order)
  - OS, or application control?

1. With Slave Processes

- Master process
  1. create a master socket and bind to a well-known address
  2. place the master socket in passive mode (listening)
  3. call **accept()** to receive next request from a client, and create a slave socket to handle communication with this specific client

1. With Slave Processes (cont'd)

- Slave process
  1. "De-references" the master socket
  2. receives requests from the client on the slave socket, sends back the responses
  3. when client is finished, closes the slave socket and exits

Concurrent Server Example

```
while ( 1) {
  new_s = accept(s, ...);          /* blocks until connection */
                                   /* request is received */
  if (fork() == 0) {               /* this is the child process*/
    close(s);                      /* child stops using parent's socket */
    process_request(new_s);        /* get request, respond */
    exit(0);                       /* child process is done! */
  }
  close(new_s);                    /* this is the parent process */
                                   /* parent has stopped using child's socket */
}
```

Accepting Connection Requests

- The listening socket **s** remains open
- Master process calls **accept()** again to obtain the next connection request
- How deliver a TCP (SYN) segment to the correct listening socket **s**?
  - segment destination = **s**'s local address & port, and
  - there is no "slave" socket with a remote address & port = source address & port of the incoming segment

Cleaning Up Child Processes

- *See Comer+Stevens, Volume III for details*

2. With Separate Threads

- Advantages of threads: shared memory space
  - less overhead to create than multi-processes
  - inter-thread communication is trivial; uses shared variables
  - is there communication between threads of a server?
- Disadvantage of threads: shared memory space!
  - programmer must protect shared, global variables
  - must use "thread-safe" libraries

3. Polling with select()

- Avoids the overhead of process creation and gives more concurrency control to application
  but makes program design more complicated
- Allows a single process to wait for connections on multiple sockets
  - checks descriptors 0 through **nfds**-1

```
int  select(int nfds,
            fd_set *readfds,
            fd_set *writefds,
            fd_set *exceptfds,
            struct timeval *timeout)
```


3. Polling with select() (cont'd)

- **select()** (normally) blocks until
  - at least 1 socket has something to be read, written, or an exception occurs
  - or a timeout occurs

The `select()` Timeout Parameter

```
struct timeval {
    long tv_sec;            /* seconds */
    long tv_usec;       /* microseconds */
}
```

- The value of the **timeout** argument determines the behavior of **select()**
  - NULL pointer: **select()** blocks, but may wait indefinitely
  - Non-zero value: **select()** blocks, but only waits up to the time specified
  - Zero: **select()** returns immediately after checking the descriptors, no blocking (i.e., used for polling)

`select()` Details

nfds: highest number assigned to a socket descriptor

readfds: set of descriptors we want to read from

writefds: set of descriptors we want to write to

exceptfds: set of descriptors to watch for exceptions

timeout: maximum time select should wait

```
FD_ZERO(fd_set *fdset)              /* clear all bits in  */
                                    /* fdset              */
FD_SET(int fd, fd_set *fdset)       /* turn bit for fd on  */
FD_CLR(int fd, fd_set *fdset)       /* turn bit for fd off */
FD_ISSET(int fd, fd_set *fdset)     /* test if bit fd is on */
```


Using `select()`

1. Create all the sockets needed
2. Create the bit field **fd_set**
3. Clear **fd_set** with **FD_ZERO**
4. Add each socket you want to watch using **FD_SET**
5. Call **select()**
6. When **select()** returns, use **FD_ISSET** to find out which socket needs attention, then do the processing for that socket

Concurrent Server Example with `select()`

```
int master_s, new_s;
fd_set rfds, afds;              /* read, active descriptors */
master_s = socket(...);         /* create master socket     */
bind(master_s, ...);            /* bind to a port           */
listen(master_s, 5);            /* listen for requests      */
FD_ZERO(&afds);
FD_SET(master_s, &afds);        /* only respond to requests */
                                /* on master socket         */
while ( 1 ) {
   bcopy(&rfds, &afds, sizeof(rfds));
   if (select(nfds, &rfds, (fd_set *)0, (fd_set *)0,
             (struct timeval *)0 ) < 0)
      (print error and exit)
...
```

Concurrent Server Example (cont'd)

```
   if (FD_ISSET(master_s, &rfds)) {
        /* i.e., a new connection request has arrived */
       new_s = accept(master_s, ... );
       FD_SET(new_s, &afds);
   }
   /* now start checking the slave sockets */
   for(fd=0; fd < nfds; fd++){
   /* process slave requests */
       if(fd != master_s && FD_ISSET(fd, &rfds)) {
           process_request(fd);
           if (finished(fd))
               FD_CLR(fd, &afds);
       }
   }
}
```

Questions

- In what order are client requests processed?
- Does any client request get higher priority than others?
- If there are no new connection requests, will existing client requests be starved?
- As client load increases, will the rate of accepting new connection requests increase, decrease, or stay the same?
- After a new connection request is accepted, will a client request for that connection be immediately processed, or will it have to wait for other client requests to be processed first?

SOME SERVER DESIGN ISSUES

Multiservice Servers (TCP, UDP)

- Typically, individual server for each service
  - dozens of server processes (daemons)
  - routed, snmpd, in.rlogind, ftpd, mountd, telnetd, ...
- Drawbacks
  - most of the servers rarely receive requests, but consume system resources
  - much of the development effort is the same for each type of server; actual request processing may be trivial
- Solution: consolidate many services into a single server

Multiservice Servers (TCP, UDP) (cont'd)

- Server manages multiple services
  - one "master" port per service
- UDP
  - dedicate one port for each service
  - handle requests in order received, iteratively
  - port received on identifies service needed
- TCP
  - dedicate one port per service to listen for connection requests
  - create a socket for each request to concurrently handle connections

Process *Preallocation*

- Master
  - opens socket for well-known port
  - creates *N* slaves
    - each inherits the socket for the well-known port
  - master can then exit
- Each slave calls `accept()`
  - when a connection request arrives, one of the slaves is unblocked and handles the connection
  - when finished, it closes connection and calls `accept()` again

Process Preallocation (cont'd)

- Assessment
  - good: can respond quickly to short requests
  - not so good: maximum concurrency level limited to *N*

Delayed Process Allocation

- Tradeoff
  - short requests favor iterative server
  - long requests favor concurrent server
  - compromise: start iteratively, switch to concurrent if time becomes "too long"
- Timer signals when "too long" occurs
- Switching from iterative to concurrent
  - `fork()` creates exact duplicate of all variables, including open sockets
  - child process can continue from exactly the same point

RAW SOCKETS

Raw Sockets

- Allows reading and writing packet types that are also handled by the kernel (e.g. IGMP, ICMPv4, ICMPv6)
- Allows reading and writing packet types that are not handled at all by the kernel (e.g. OSPF)

Raw Sockets (cont'd)

- With **`IP_HDRINCL`** socket option, allows writing your own IPv4 header
- Why use?
  - specialty programs, like **`ping`** and **`traceroute`**
  - new protocol implementations
  - exploits

Using Raw Sockets

- **`socket(AF_INET, SOCK_RAW, `*`protocol`*`)`**
- *protocol* is usually a constant like **`IPPROTO_ICMP`**
- Optionally, set **`IP_HDRINCL`**
  - **`IP_HDRINCL`** lets you include the whole IP header
  - otherwise, the kernel writes the header for you
- **`bind()`** and **`connect()`** are optional, similar to use in UDP

Reading / Writing

- Without **`IP_HDRINCL`**
  - send packet with **`sendto()`**
- With **`IP_HDRINCL`**
  - send packet, including IP header, with **`send()`** or **`write()`**

Summary

1. UDP applications relatively easy to write: send a datagram, wait for a response
2. The sockets API for TCP requires the application to be prepared to read and write data a byte at a time
3. Servers have to handle concurrent requests, using either (a) processes, (b) threads, or (c) polling with **select()**
   - easiest: processes
4. Preallocation / delayed allocation help tune performance of servers

Next Lecture

- Subnetting, Classless Addresses, and CIDR