

COMP2303/COMP7306
Semester One, 2011

Week 9

Network Programming

School of Information Technology and Electrical Engineering
The University of Queensland

COMP2303/COMP7306

Outline

- TCP/IP
 - More details next week
- Sockets
- Typical TCP client
- Typical TCP server

- Credits:
 - Glass and Ables, "UNIX for Programmers and Users"
 - Bryant and O'Halloran, "Computer Systems: A Programmer's Perspective"
 - Rochkind, "Advanced UNIX Programming"
 - Tanenbaum, "Computer Networks"

3

COMP2303/COMP7306

TCP/IP

- Protocol = Rules for communication
- **TCP** = Transmission Control Protocol
 - Provides communication between ports on two computers (hosts)
 - Bidirectional
 - Point-to-point
 - Reliable
 - Byte stream
 - Uses **IP** (Internet Protocol) to transmit small packets of data between two IP addresses
- More details on these protocols next week
- Today – how to write programs using TCP/IP

4

COMP2303/COMP7306

Client-Server Model

- Most network applications are based on the **client-server model**:
 - A *server* process and one or more *client* processes
 - Server manages some *resource*
 - Server provides *service* by manipulating resource for clients

```
graph LR; Client[Client process] -- "1. Client sends request" --> Server[Server process]; Server -- "2. Server handles request" --> Resource[(Resource)]; Server -- "3. Server sends response" --> Client; Client -- "4. Client handles response" --> Client;
```

Note: clients and servers are processes running on hosts (can be the same or different hosts).

5

COMP2303/COMP7306

TCP Connections

- Identified by
 - Source IP address
 - Source port number
 - Destination IP address
 - Destination port number

6

COMP2303/COMP7306

IP Addresses (v4)

- 32-bit numbers
- Often written in **dotted-decimal** notation for human consumption
 - each of the 4 bytes written in decimal
 - e.g. 130.102.2.15
- Some addresses have special meanings
 - More on this next week

7

Port Numbers

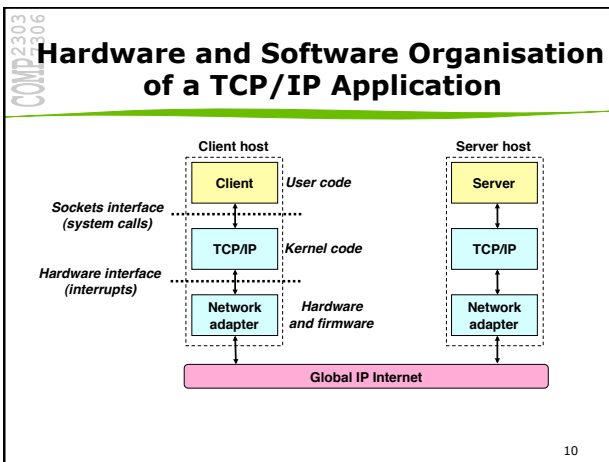
- 16 bits: 0 – 65535
- Below 1024
 - Well known ports
 - Reserved for standard services, e.g.
 - 23 - Telnet
 - 21 - FTP
 - 80 - HTTP
 - Look in /etc/services on a UNIX box

8

Sockets

- Introduced in Berkeley UNIX
- Sometimes called UNIX sockets
- Originally C based
 - Many other languages now
- A socket is a **communication endpoint**
 - Associated with a file descriptor in UNIX – can do file I/O on socket
 - Main distinction between regular file I/O and socket I/O is how the application "opens" the socket descriptors
- Many types of sockets
 - We'll look at stream sockets (TCP)

9



Socket System Calls

- **socket (...)**
 - Create new communication end-point
- **bind (...)**
 - Attach a local address to a socket
- **listen (...)**
 - Willing to accept connections, give queue size
- **accept (...)**
 - Wait for a connection attempt to arrive
- **connect (...)**
 - Attempt to establish a connection

11

Socket System Calls (cont.)

- **send (...)** OR **write (...)**
 - Send data over the connection
- **recv (...)** OR **read (...)**
 - Receive data over the connection
- **sendto (...)**
 - Send datagram
- **recvfrom (...)**
 - Receive datagram
- **close (...)**
 - Release the connection
- **shutdown (...)**
 - Close down one side of connection (or both sides)

● Not all are applicable in all circumstances!

12

Typical TCP Server

- Create socket socket (...)
- Bind to address/port bind (...)
- Specify willingness to accept connections listen (...)
- Block waiting for connection accept (...)
 - **accept (...)** returns a new socket
 - Original socket continues to listen
- Deal with request
 - e.g. spawn process or thread send (...) OR write (...)
 - recv (...) OR read (...)
- Continue

13

COMP2303
COMP7306

Typical TCP Client

- Create socket `socket (...)`
- Connect to server (at a particular address) `connect (...)`
- Send/receive data as necessary `send (...)` or `write (...)`
`recv (...)` or `read (...)`
- Close connection `close (...)`
- Clients don't normally use `bind (...)`
 - don't care what the outgoing port is

14

COMP2303
COMP7306

Other Programming Languages

- Sockets interface is available in many programming languages
 - Interface is similar (but not identical) across all of them
- We'll be concentrating on the C sockets interface

15

COMP2303
COMP7306

IP Addresses in C

- 32-bit IP addresses are stored in an *IP address struct*
 - IP addresses are always stored in memory in network byte order (big-endian byte order)
 - True in general for any integer transferred in a packet header from one machine to another
 - E.g., the port number used to identify an Internet connection

```
typedef uint32_t in_addr_t;
/* Internet address structure */
struct in_addr {
    in_addr_t s_addr; /* network byte order (big-endian) */
};
```

- Handy network byte-order conversion functions:
 - `htonl()`: convert `uint32_t` from host to network byte order.
 - `htons()`: convert `uint16_t` from host to network byte order.
 - `ntohl()`: convert `uint32_t` from network to host byte order.
 - `ntohs()`: convert `uint16_t` from network to host byte order.

17

COMP2303
COMP7306

IP Addresses in C (cont.)

- Users (applications) often write IP addresses using dotted decimal notation
 - e.g. IP address `0x8002C2F2 = 128.2.194.242`
- Functions for converting between binary IP addresses and dotted decimal strings:
 - `inet_addr(...)`: converts a dotted decimal string to an IP address in network byte order
 - `inet_ntoa(...)`: converts an IP address in network byte order to its corresponding dotted decimal string
 - "n" denotes network representation. "a" denotes application representation

18

COMP2303
COMP7306

Address Handling Code Examples

- *To be discussed/commented in class*
 - See separate handout

19

COMP2303
COMP7306

Creating a Socket

- `socket (...)` creates a socket descriptor
 - `AF_INET`: indicates that the socket is associated with Internet protocols
 - `SOCK_STREAM`: selects a reliable byte stream connection (TCP)

```
int fd; /* socket descriptor */

/* Create a TCP socket descriptor */
fd = socket(AF_INET, SOCK_STREAM, 0);
if (fd < 0) {
    perror("Socket creation failed");
    exit(1);
}
```

20

COMP2303
COMP7306

Socket Address Structures

- Generic socket address:
 - For address arguments to `connect`, `bind`, and `accept`.
 - Necessary only because C did not have generic (`void *`) pointers when the sockets interface was designed

```

struct sockaddr {
    sa_family_t sa_family; /* protocol family */
    char sa_data[14]; /* address data. */
};
    
```

- Internet-specific socket address:
 - Must cast (`struct sockaddr_in *`) to (`struct sockaddr *`) for `connect`, `bind`, and `accept`

```

struct sockaddr_in {
    sa_family_t sin_family; /* address family (always AF_INET) */
    in_port_t sin_port; /* port num in network byte order */
    struct in_addr sin_addr; /* IP addr in network byte order */
    unsigned char sin_zero[8]; /* pad to sizeof(struct sockaddr) */
};
    
```

COMP2303
COMP7306

Socket Address Structures

```

struct sockaddr {
    sa_family_t sa_family; /* protocol family */
    char sa_data[14]; /* address data. */
};
    
```

```

struct sockaddr_in {
    sa_family_t sin_family; /* address family (always AF_INET) */
    in_port_t sin_port; /* port num in network byte order */
    struct in_addr sin_addr; /* IP addr in network byte order */
    unsigned char sin_zero[8]; /* pad to sizeof(struct sockaddr) */
};
    
```

COMP2303
COMP7306

Typical Client (Stream based)

- Create socket `socket (...)`
- Connect to server (at a particular address) `connect (...)`
- Send/receive data as necessary `send (...)` or `write (...)`
`recv (...)` or `read (...)`
- Close connection `close (...)`

24

COMP2303
COMP7306

Sample Client Code

- To be discussed/commented in class*

25

COMP2303
COMP7306

Typical Server (Stream based)

- Create socket `socket (...)`
- Bind to address/port `bind (...)`
- Specify willingness to accept connections `listen (...)`
- Block waiting for connection `accept (...)`
 - `accept (...)` returns a new socket
 - Original socket continues to listen
- Deal with request
 - e.g. spawn process/thread & communicate `send (...)` or `write (...)`
`recv (...)` or `read (...)`
- Continue

27

COMP2303
COMP7306

accept ()

- `accept (...)` blocks waiting for a connection request

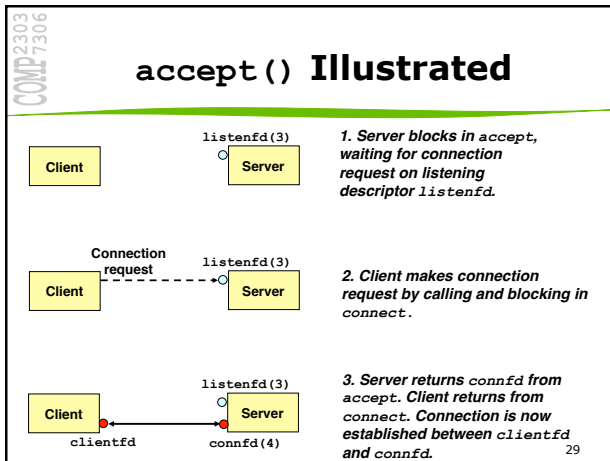
```

int listenfd; /* listening descriptor */
int connfd; /* connected descriptor */
struct sockaddr_in clientaddr;
int clientlen;

clientlen = sizeof(clientaddr);
connfd = accept(listenfd, (struct sockaddr*)&clientaddr, &clientlen);
    
```

- `accept (...)` returns a **connected descriptor** (`connfd`) with the same properties as the **listening descriptor** (`listenfd`)
 - Returns when the connection between client and server is created and ready for I/O transfers
 - All I/O with the client will be done via the connected socket
- `accept (...)` also fills in client's IP address.

28



Connected vs. Listening Descriptors

- **Listening** descriptor
 - End point for client connection requests
 - Created once and exists for lifetime of the server
- **Connected** descriptor
 - End point of the connection between client and server
 - A new descriptor is created each time the server accepts a connection request from a client
 - Exists only as long as it takes to service client
- Why the distinction?
 - Allows for *concurrent* servers that can communicate over many client connections simultaneously
 - e.g., each time we receive a new request, we fork a child process to handle the request

Socket Options: setsockopt

- The socket can be given some attributes
 - Many are integers

```
int optval = 1;
/* Eliminates "Address already in use" error from bind(). */
if (setsockopt(fd, SOL_SOCKET, SO_REUSEADDR,
              (const void *)&optval, sizeof(int)) < 0)
{
    perror("Unable to set socket option");
    exit(1);
}
```

- Handy trick that allows us to rerun a server immediately after we kill it
 - Otherwise we may have to wait about 30 secs+
 - Eliminates "Address already in use" error from `bind()`
 - Useful when debugging

Echo Server: Identifying the Client

- The server can determine the domain name and IP address of the client

```
int error;
char hostname[128]; /* Space to hold hostname */

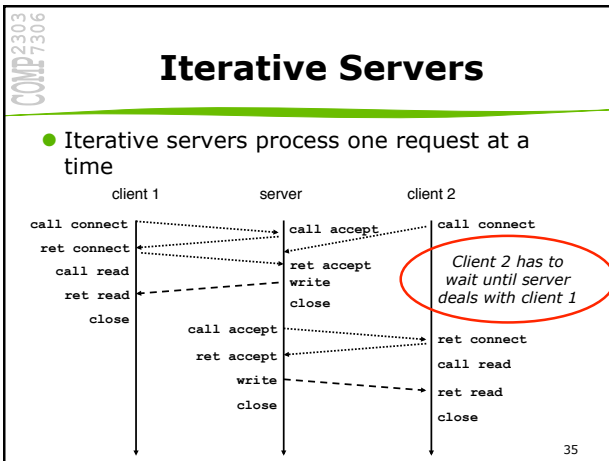
/* After accept() has populated clientaddr, clientlen */
error = getnameinfo((struct sockaddr*)&clientaddr, clientlen,
                  hostname, 128, NULL, 0, 0);
if(!error) {
    printf("Connection from %s (%s)\n", hostname,
          inet_ntoa(clientaddr.sin_addr));
}
```

Testing Servers Using netcat

- The `netcat` (`nc`) program is invaluable for testing servers that transmit over Internet connections
 - Our simple server
 - Web servers
 - Mail servers
- Usage:
 - `unix> nc <host> <portnumber>`
 - Creates a connection with a server running on `<host>` and listening on port `<portnumber>`
- Can also use `telnet` (Ctrl-] then "quit" to exit)
- `netcat` can also pretend to be a server
 - `unix> nc -l -p <portnumber>`

Sample Server Code

- *To be discussed/commented in class*



- 3 Basic Mechanisms for Creating Concurrent Flows**
- Processes
 - Kernel automatically interleaves multiple logical flows
 - Each flow has its own private address space
 - Threads
 - Kernel automatically interleaves multiple logical flows
 - Each flow shares the same address space
 - I/O multiplexing with `select()`
 - User manually interleaves multiple logical flows
 - Each flow shares the same address space
 - Popular for high-performance server designs
- 36

- Resources**
- Beej's Guide to Network Programming
 - <http://beej.us/guide/bqnet/>
 - Manual pages
 - On moss: `man <name>` or `man -s 2 <name>`
 - where `<name>` is `socket`, `bind`, `connect`, `listen`, `accept`, `recv`, `send`, ...
 - Glass & Ables, "UNIX for Programmers and Users"
 - Rochkind, "Advanced UNIX Programming"
 - Bryant and O'Halloran, "Computer Systems: A Programmer's Perspective"
 - Other UNIX Programming books...
 - See Reference text list in course profile
- 37

