In the name of Allah

Computer Engineering Department
Sharif University of Technology

CE443- Computer Networks

Socket Programming

Acknowledgments: Lecture slides are from Computer networks course thought by Jennifer Rexford at Princeton University. When slides are obtained from other sources, a reference will be noted on the bottom of that slide.
Socket: End Point of Net. Comm.’s

• Socket as an Application Programming Interface
  – Supports the creation of network applications

• Two ends communicate through a “socket”
  – Sending messages from one process to another
  – The transportation details are transparent to the programmer
Delivering the Data: Division of Labor

• **Application**
  – Read data from and write data to the socket
  – Interpret the data (e.g., render a Web page)

• **Operating system**
  – Deliver data to the destination socket
  – Based on the destination port number

• **Network**
  – Deliver data packet to the destination host
  – Based on the destination IP address
Identifying the Receiving Process

- **Sending process must identify the receiver**
  - The receiving end host machine
  - The specific socket in a process on that machine

- **Receiving host**
  - Destination address that uniquely identifies the host
  - An IPv4 address is a 32-bit quantity

- **Receiving socket**
  - Host may be running many different processes
  - Destination port that uniquely identifies the socket
  - A port number is a 16-bit quantity
Identifying the Receiving Process

Service request for 128.2.194.242:80 (i.e., the Web server)

Service request for 128.2.194.242:7 (i.e., the echo server)
Knowing What Port Number To Use

• Popular applications have well-known ports
  – E.g., port 80 for Web and port 25 for e-mail
  – See http://www.iana.org/assignments/port-numbers

• Well-known vs. ephemeral ports
  – Server has a well-known port (e.g., port 80)
    • Between 0 and 1023
  – Client picks an unused ephemeral (i.e., temporary) port
    • Between 1024 and 65535

• Uniquely identifying the traffic between the hosts
  – Two IP addresses and two port numbers
  – Underlying transport protocol (e.g., TCP or UDP)
Port Numbers are Unique on Each Host

- Port number uniquely identifies the socket
  - Cannot use same port number twice with same address
  - Otherwise, the OS can’t demultiplex packets correctly

- Operating system enforces uniqueness
  - OS keeps track of which port numbers are in use
  - Doesn’t let the second program use the port number

- Example: two Web servers running on a machine
  - They cannot both use port “80”, the standard port #
  - So, the second one might use a non-standard port #
  - E.g., http://www.cnn.com:8080
UNIX Socket API
UNIX Socket API

• **Socket interface**
  – Originally provided in Berkeley UNIX
  – Later adopted by all popular operating systems
  – Simplifies porting applications to different OSes (even to the Windows!)

• **In UNIX, everything is like a file**
  – All input is like reading a file
  – All output is like writing a file
  – File is represented by an integer file descriptor

• **API implemented as system calls**
  – E.g., connect, read, write, close, …
Typical Client Program

• Prepare to communicate
  – Create a socket
  – Determine server address and port number
  – Initiate the connection to the server

• Exchange data with the server
  – Write data to the socket
  – Read data from the socket
  – Do stuff with the data (e.g., render a Web page)

• Close the socket
Typical Server Program

• Prepare to communicate
  – Create a socket
  – Associate local address and port with the socket

• Wait to hear from a client (passive open)
  – Indicate how many clients-in-waiting to permit
  – Accept an incoming connection from a client

• Exchange data with the client over new socket
  – Receive data from the socket
  – Do stuff to handle the request (e.g., get a file)
  – Send data to the socket
  – Close the socket

• Repeat with the next connection request
Putting it All Together

**Server**
- `socket()`
  - `bind()`
    - `listen()`
      - `accept()`
        - `read()`
        - `write()`
          - `block`

**Client**
- `socket()`
  - `connect()`
    - `write()`
      - `establish connection`
      - `send request`
      - `read()`
Wanna See Real Clients and Servers?

- **Apache Web server**
  - Open source server first released in 1995
  - Name derives from “a patchy server” ;-)
  - Software available online at [http://www.apache.org](http://www.apache.org)

- **Mozilla Web browser**

- **Sendmail**
  - [http://www.sendmail.org/](http://www.sendmail.org/)

- **BIND Domain Name System (Datagram)**
  - Client resolver and DNS server

- ...
Wanna to have fun? Okay…

Client Programming
int `socket`(int `domain`, int `type`, int `protocol`)

- **Operation to create a socket**
  - Returns a descriptor (or handle) for the socket
  - Originally designed to support any protocol suite

- **Domain: protocol family**
  - PF_INET for the Internet

- **Type: semantics of the communication**
  - SOCK_STREAM: reliable byte stream
  - SOCK_DGRAM: message-oriented service

- **Protocol: specific protocol**
  - UNSPEC: unspecified
  - (PF_INET and SOCK_STREAM already implies TCP)
Client: Learning Server Address/Port

- Server typically known by name and service
  - “www.google.com” and “http”
- Which must be translated into IP address and port #
- Translating the server’s name to an address
  - int `getaddrinfo`(const char *node, const char *service, const struct addrinfo *hints, struct addrinfo **res);
  - void `freeaddrinfo`(struct addrinfo *res);
  - int `getnameinfo`(const struct sockaddr *sa, socklen_t salen, char *host, size_t hostlen, char *serv, size_t servlen, int flags);
- Check Linux Man pages for details
Client: Learning Server Address/Port

- `struct addrinfo {
  int ai_flags;
  int ai_family;
  int ai_socktype;
  int ai_protocol;
  socklen_t ai_addrlen;
  struct sockaddr *ai_addr;
  char *ai_canonname;
  struct addrinfo *ai_next;
};`
IP Address Data Structures

```c
#include <netinet/in.h>

// All pointers to socket address structures are often cast to pointers
// to this type before use in various functions and system calls:

struct sockaddr {
    unsigned short  sa_family;   // address family, AF_xxx
    char            sa_data[14]; // 14 bytes of protocol address
};

// IPv4 AF_INET sockets:

struct sockaddr_in {
    short           sin_family; // e.g. AF_INET, AF_INET6
    unsigned short  sin_port;  // e.g. htons(3490)
    struct in_addr  sin_addr;  // see struct in_addr, below
    char            sin_zero[8]; // zero this if you want to
};

struct in_addr {
    unsigned long  s_addr;     // load with inet_pton()
};
```
Client: Connecting Socket to the Server

```c
int connect(int sockfd, struct sockaddr *server_address, socklen_t addrlen)
```

- Client contacts the server to establish connection
  - Associate the socket with the server address/port
  - Acquire a local port number (assigned by the OS)
  - Request connection to server, who will hopefully accept

- Establishing the connection
  - Arguments: socket descriptor, server address, and address size
  - Returns 0 on success, and -1 if an error occurs
Client: Sending and Receiving Data

• Sending data
  ssize_t write(int sockfd, void *buf, size_t len)
  – Arguments: socket descriptor, pointer to buffer of data to send, and length of the buffer
  – Returns the number of characters written, and -1 on error

• Receiving data
  ssize_t read(int sockfd, void *buf, size_t len)
  – Arguments: socket descriptor, pointer to buffer to place the data, size of the buffer
  – Returns the number of characters read (where 0 implies “end of file”), and -1 on error

• Closing the socket
  int close(int sockfd)
Not enough fun? Okay… face a headache!

Server Programming
Servers Differ From Clients

• Passive open
  – Prepare to accept connections
  – …. but don’t actually establish
  – …. until hearing from a client

• Hearing from multiple clients
  – Allowing a backlog of waiting clients
  – … in case several try to communicate at once

• Create a socket for each client
  – Upon accepting a new client
  – …. create a new socket for the communication
Remember: Typical Server Program

• Prepare to communicate
  – Create a socket
  – Associate local address and port with the socket

• Wait to hear from a client (passive open)
  – Indicate how many clients-in-waiting to permit
  – Accept an incoming connection from a client

• Exchange data with the client over new socket
  – Receive data from the socket
  – Do stuff to handle the request (e.g., get a file)
  – Send data to the socket
  – Close the socket

• Repeat with the next connection request
Remember: The Big Picture

Server

- `socket()`
  - `bind()`
  - `listen()`
  - `accept()`
  - `read()`
  - `block`
  - `process request`
  - `write()`

Client

- `socket()`
  - `connect()`
  - `write()`
  - `send request`
  - `establish connection`
  - `read()`
Server: Server Preparing its Socket

• Server creates a socket and binds address/port
  – Server creates a socket, just like the client does
  – Server associates the socket with the port number
    (and hopefully no other process is already using it!)

• Create a socket
  int socket(int domain, int type, int protocol)

• Bind socket to the local address and port number
  int bind(int sockfd, struct sockaddr *my_addr, socklen_t addrlen)
  – Arguments: socket descriptor, server address, address length
  – Returns 0 on success, and -1 if an error occurs
Server: Allowing Clients to Wait

- Many client requests may arrive
  - Server cannot handle them all at the same time
  - Server could reject the requests, or let them wait
  - Define how many connections can be pending: backlog

- Wait for clients
  ```
  int listen(int sockfd, int backlog)
  ```
  - Arguments: socket descriptor and acceptable backlog
  - Returns a 0 on success, and -1 on error

- What if too many clients arrive?
  - Some requests don’t get through
  - The Internet makes no promises…
  - And the client can always try again
Server: Accepting Client Connection

• Now all the server can do is wait…
  – Waits for connection request to arrive
  – Blocking until the request arrives
  – And then accepting the new request

• Accept a new connection from a client
  
  ```c
  int accept(int sockfd, struct sockaddr *addr, socketlen_t *addrlen)
  ```
  – Arguments: socket descriptor, structure that will provide client address and port, and length of the structure
  – Returns descriptor for a new socket for this connection
Server: One Request at a Time?

• Serializing requests is inefficient
  – Server can process just one request at a time
  – All other clients must wait until previous one is done

• May need to time share the server machine
  – Alternate between servicing different requests
    • E.g. use multi-threading
  – Or, start a new process to handle each request
    • Allow the operating system to share the CPU across processes
  – Or, some hybrid of these two approaches
Client and Server: Cleaning House

• Once the connection is open
  – Both sides and read and write
  – Two unidirectional streams of data
  – In practice, client writes first, and server reads
  – … then server writes, and client reads, and so on

• Closing down the connection
  – Either side can close the connection
  – … using the close() system call

• What about the data still “in flight”
  – Data in flight still reaches the other end
  – So, server can close() before client finishing reading
The Problem of Interoperability
Byte Order

• Hosts differ in how they store data
  – E.g., four-byte number (byte3, byte2, byte1, byte0)

• Little endian (“little end comes first”) ← Intel PCs!!!
  – Low-order byte stored at the lowest memory location
  – Byte0, byte1, byte2, byte3

• Big endian (“big end comes first”)
  – High-order byte stored at lowest memory location
  – Byte3, byte2, byte1, byte0

• Makes it more difficult to write portable code
  – Client may be big or little endian machine
  – Server may be big or little endian machine
IP is Big Endian

• But, what byte order is used “on the wire”
  – That is, what do the network protocol use?
• The Internet Protocols picked one convention
  – IP is big endian (aka “network byte order”)
• Writing portable code require conversion
  – Use htons() and htonl() to convert to network byte order
  – Use ntohs() and ntohl() to convert to host order
• Hides details of what kind of machine you’re on
  – Use the system calls when sending/receiving data structures longer than one byte
Why Can’t Sockets Hide These Details?

• Dealing with endian differences is tedious
  – Couldn’t the socket implementation deal with this
  – … by swapping the bytes as needed?

• No, swapping depends on the data type
  – Two-byte short int: (byte 1, byte 0) vs. (byte 0, byte 1)
  – Four-byte long int: (byte 3, byte 2, byte 1, byte 0) vs. (byte 0, byte 1, byte 2, byte 3)
  – String of one-byte charters: (char 0, char 1, char 2, …) in both cases

• Socket layer doesn’t know the data types
  – Sees the data as simply a buffer pointer and a length
  – Doesn’t have enough information to do the swapping
The Web as an Example Application
The Web: URL, HTML, and HTTP

• Uniform Resource Locator (URL)
  – A pointer to a “black box” that accepts request methods
  – Formatted string with protocol (e.g., http), server name (e.g., www.cnn.com), and resource name (coolpic.jpg)

• HyperText Markup Language (HTML)
  – Representation of hypertext documents in ASCII format
  – Format text, reference images, embed hyperlinks
  – Interpreted by Web browsers when rendering a page

• HyperText Transfer Protocol (HTTP)
  – Client-server protocol for transferring resources
  – Client sends request and server sends response
Example: HyperText Transfer Protocol

Request

```
GET /courses/archive/spring08/cos461/ HTTP/1.1
Host: www.cs.princeton.edu
User-Agent: Mozilla/4.03
<CRLF>
```

Response

```
HTTP/1.1 200 OK
Date: Mon, 4 Feb 2008 13:09:03 GMT
Server: Netscape-Enterprise/3.5.1
Content-Type: text/plain
Last-Modified: Mon, 4 Feb 2008 11:12:23 GMT
Content-Length: 21
<CRLF>
Site under construction
```
In Fact, Try This at a UNIX Prompt…

labpc: telnet www.cnn.com 80
GET /index.html HTTP/1.1
Host: www.cnn.com
<CRLF>

And you’ll see the response…
Web Server

• Web site vs. Web server
  – Web site: collections of Web pages associated with a particular host name
  – Web server: program that satisfies client requests for Web resources

• Handling a client request
  – Accept the socket
  – Read and parse the HTTP request message
  – Translate the URL to a filename
  – Determine whether the request is authorized
  – Generate and transmit the response