Socket Programming
EE 122: Intro to Communication Networks
Fall 2006 (MW 4-5:30 in Donner 155)
Sukun Kim / Vern Paxson
TAs: Dilip Antony Joseph and Sukun Kim
http://inst.eecs.berkeley.edu/~ee122/
Materials with thanks to Artur Rivilis, Jennifer Rexford, Ion Stoica
and colleagues at Princeton and UC Berkeley

Overview

• Socket Programming: how applications use the network
  – Sockets are a C-language programming interface
    between Layer 7 (applications) and Layer 4 (transport)
  – Interface is quite general and fairly abstract
  – Use of interface differs somewhat between clients & servers

Socket: End Point of Communication

• Sending message from one process to another
  – Message must traverse the underlying network
• Process sends and receives through a "socket"
  – In essence, the doorway leading in/out of the house
• Socket as an Application Programming Interface
  – Supports the creation of network applications

Identifying the Receiving Process

• Sending process must identify the receiver
  – Address of the receiving end host
  – Plus identifier (port) that specifies the receiving process
• Receiving host
  – Destination address uniquely identifies the host
    • IP address is a 32-bit quantity
• Receiving process
  – Host may be running many different processes
  – Destination port uniquely identifies the socket
    • Port number is a 16-bit quantity

Using Ports to Identify Services

Announcements

• Project #1 out today, due Thu Sep 28.
• Vern & Dilip away this week.
  – I will be covering Dilip’s Friday section + office hours (in my office)
• No Lecture Wed Sept. 13
Knowing What Port Number To Use

- Popular applications have "well-known ports"
  - E.g., port 80 for Web and port 25 for e-mail
  - Well-known ports listed at [http://www.iana.org](http://www.iana.org)
    - Or see `/etc/services` on Unix system
- Well-known vs. ephemeral ports
  - Server has a well-known port (e.g., port 80)
    - By convention, between 0 and 1023; privileged
  - Client gets an unused "ephemeral" (i.e., temporary) port
    - By convention, between 1024 and 65535
- Uniquely identifying the traffic between the hosts
  - The two IP addresses plus the two port numbers
    - Sometimes called the "four-tuple"
  - And: underlying transport protocol (e.g., TCP or UDP)
    - With the above, called the "five-tuple"

Delivering the Data: Division of Labor

- Network
  - Deliver data packet to the destination host
  - Based on the destination IP address
- Operating system
  - Deliver data to the destination socket
  - Based on the protocol and destination port #
- Application
  - Read data from the socket
  - Interpret the data (e.g., render a Web page)

UNIX Socket API

- Socket interface
  - Originally provided in Berkeley UNIX
  - Later adopted by all popular operating systems
  - Simplifies porting applications to different OSes
- 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
- System calls for sockets
  - Client: create, connect, write/send, read, close
  - Server: create, bind, listen, accept, read/recv, write/send, close

Types of Sockets

- Different types of sockets implement different service models
  - Stream: SOCK_STREAM
  - Datagram: SOCK_DGRAM
- Stream socket (TCP)
  - Connection-oriented (includes establishment + termination)
  - Reliable (lossless) and in-order delivery guaranteed.
  - At-most-once delivery, no duplicates
    - E.g., SSH, HTTP (Web), SMTP (email)
- Datagram socket (UDP)
  - Connectionless (just data-transfer of packets)
    - "Best-effort" delivery, possibly lower variance in delay
    - E.g., IP Telephony (Skype), simple request/reply protocols
      (hostname → address lookups via DNS; time synchronization via NTP)

Using Stream Sockets

- No need to packetize data
- Data arrives in the form of a "byte stream"
- Receiver needs to separate messages in stream

Recovering message boundaries

- Stream socket data separation:
  - Use records (data structures) to partition data stream
  - How do we implement variable length records?
    - Use records (data structures) to partition data stream
    - How do we implement variable length records?
  - What if field containing record size gets corrupted?
    - Not possible! Why?
Datagram Sockets

• User packetizes data before sending
• Maximum size of 64 KB
• Using previous example, "Hi there!" and "Hope you are well" definitely sent in separate packets at network layer
  – Message boundaries preserved
  – But note: your message had better fit within 64 KB or else you’ll have to layer your own boundary mechanism on top of the datagram delivery anyway

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
  • Note, single socket supports both reading and writing
  – Do stuff with the data (e.g., render a Web page)
• Close the socket

Creating a Socket: socket()

• Operation to create a socket
  – int socket(int domain, int type, int protocol)
  – Returns a descriptor (or handle) for the socket
  – Originally designed to support any protocol suite
• Domain: protocol family
  – Use 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. No need for us to specify, since PF_INET plus SOCK_STREAM already implies TCP, or SOCK_DGRAM implies UDP.

Connecting the Socket to the Server

• Translate the server’s name to an address
  – struct hostent *gethostbyname(char *name)
  – name: the name of the host (e.g., "www.cnn.com")
  – Returns a structure that includes the host address
  • Or NULL if host doesn’t exist
• Identifying the service’s port number
  – struct servent *getservbyname(char *name, char *proto)
  – E.g., getservbyname("http", "tcp")
• Establishing the connection
  – int connect(int sockfd, struct sockaddr *server_address, socketlen_t addrlen)
  – Arguments: socket descriptor, server address, and address size
  – Returns 0 on success, and -1 if an error occurs

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)

Sending and Receiving Data, con’t

• Note: instead of using write(), you can instead use send(), which is intended for use with sockets.
  – Only difference is send() takes one additional argument of flags, which for most purposes don’t matter
• Similarly, instead of using read(), you can instead use recv() 
  – Again, only difference is one additional argument of flags
• Important to realize they’re basically equivalent, since you see both pairs of calls used (sometimes intermingled).
Byte Ordering

- We talk about two numeric presentations:
  - **Big Endian**
    - Architectures: Sun SPARC, Motorola 68000, PowerPC 970, IBM System/360
    - The most significant byte stored in memory at the lowest address.
    - Example: 4A3B2C1D hexadecimal will be stored as: 4A 3B 2C 1D
    - This is network-byte order.
  - **Little Endian**
    - Architectures: Intel x86, AMD64, DEC VAX
    - The least significant byte stored in memory at the lowest address.
    - Example: 4A3B2C1D hexadecimal will be stored as: 1D 2C 3B 4A

- What problems can arise because of this?
- What can we do to solve them?

Byte Ordering Solution

- The networking API provides us the following functions:
  - `uint16_t htons(uint16_t host16bitvalue);`
  - `uint32_t htonl(uint32_t host32bitvalue);`
  - `uint16_t ntohs(uint16_t net16bitvalue);`
  - `uint32_t ntohl(uint32_t net32bitvalue);`
- Use for all 16-bit and 32-bit binary numbers (short, int) to be sent across network
- ‘h’ stands for “host order”
- These routines do nothing on big-endian hosts
- Note: common mistake is to forget to use these

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

Servers Differ From Clients

- Passive open
  - Prepare to accept connections
  - … but don’t actually establish one
  - … until hearing from a client
- Hearing from multiple clients
  - Allow a backlog of waiting clients
  - … in case several try to start a connection at once
- Create a socket for each client
  - Upon accepting a new client
  - … create a new socket for the communication

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
Server Preparing its Socket

- 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

- Define how many connections can be pending
  - `int listen(int sockfd, int backlog)`
  - Arguments: socket descriptor and acceptable backlog
  - Returns 0 on success, and -1 on error

Accepting a New Client Connection

- Accept a new connection from a client
  - `int accept(int sockfd, struct sockaddr *addr, socklen_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

Questions

- What happens if no clients are around?
  - The `accept()` call blocks waiting for a client
- What happens if too many clients are around?
  - Some connection requests don’t get through
  - ... But, that’s okay, because the Internet makes no promises

Putting it All Together

Server
socket() \[\rightarrow\] bind() \[\rightarrow\] listen() \[\rightarrow\] accept() \[\rightarrow\] connection established \[\rightarrow\] send request \[\rightarrow\] write() \[\rightarrow\] block \[\rightarrow\] read() \[\rightarrow\] process request \[\rightarrow\] write() \[\rightarrow\] send response \[\rightarrow\] read()

Client
socket() \[\rightarrow\] connect() \[\rightarrow\] accept() \[\rightarrow\] send() \[\rightarrow\] recv() \[\rightarrow\] send() \[\rightarrow\] close() \[\rightarrow\] close()

Same, But Emphasizing the Client

- Stream socket: Transmission Control Protocol (TCP)
- Note, does not create any state inside the network
- Sequence of actions:
  - I want to check out latest Cal Football stats.
  - socket()
  - socket()
  - bind(80)
  - listen()
  - connect()
  - [connection established]
  - send()
  - recv()
  - send()
  - close()
  - close()

Let's Look At Some Code

(though you’ll still need to read the manual pages too ...)

Header Files

```
#include <unistd.h> /* access to system calls */
#include <sys/types.h> /* widely used types */
#include <errno.h> /* error codes / perror() */
#include <netdb.h> /* gethostbyname() & friends */
#include <arpa/inet.h> /* htons() & friends */
#include <sys/socket.h> /* socket structs */
#include <netinet/in.h> /* Internet sockets, including sockaddr_in */
```
Socket Creation (Client + Server)

```c
int sock;
if ((sock = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP)) < 0) {
    perror("Failed to create TCP socket");
    abort();
}
```

• Handling errors that occur rarely usually consumes most of systems code

Server Reusing Its Port

• After TCP connection closes, it waits several minutes before freeing up the associated port
• But server port numbers are fixed ⇒ must be reused
• Solution:

```c
int optval = 1;
if ((sock = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
    perror("couldn't create TCP socket");
    abort();
}
/* Note, this call must come before call to bind() */
if (setsockopt(sock, SOL_SOCKET, SO_REUSEADDR, &optval, sizeof(optval)) < 0) {
    perror("couldn't reuse address");
    abort();
}
```

Serving Binding Port to Socket

• Want port at server end to use a particular number

```c
struct sockaddr_in sin;
memset(&sin, 0, sizeof(sin));
sin.sin_family = AF_INET;
sin.sin_addr.s_addr = INADDR;
sin.sin_port = htons(server_port);
if (bind(sock, (struct sockaddr *)&sin, sizeof(sin)) < 0) {
    perror("cannot bind socket to address");
    abort();
}
```

Server waits for incoming connections

• Backlog parameter specifies max number of established connections waiting to be accepted (using accept()) - What would happen if you didn’t bother with a backlog?

```c
if (listen(sock, 5) < 0) {
    perror("error listening");
    abort();
}
```

Client Establishes connection

```c
struct sockaddr_in sin;
struct hstent *host = gethostbyname(argv[1]);
in_addr_t server_addr = *(in_addr_t *) host->h_addr_list[0];
unsigned short server_port = atoi(argv[2]);
memset(&sin, 0, sizeof(sin));
sin.sin_family = AF_INET;
sin.sin_addr.s_addr = server_addr; /* already in network order */
sin.sin_port = htons(server_port);
if (connect(sock, (struct sockaddr *)&sin, sizeof(sin)) < 0) {
    perror("cannot connect to server");
    abort();
}
```

Server Accepts Incoming Connection

```c
sockaddr_in addr;
int addr_len = sizeof(addr);
int sock;
sock = accept(tcp_sock, (struct sockaddr *)&addr, &addr_len);
if (sock < 0) {
    perror("error accepting connection");
    abort();
}
```

Must do this for all networking API calls.
**Sending Data Stream**

- Now that the connection is established, we want to send data:

  ```c
  int send_packets (char *buffer, int buffer_len)
  {
    sent_bytes = send (sock, buffer, buffer_len, 0);
    if (sent_bytes < 0)
      perror ("send() failed");
    return 0;
  }
  ```

  Again, can use `write()` instead of `send()`.

**Receiving Data Stream**

- Receiving is nearly symmetric:

  ```c
  int receive_packets (char *buffer, int buffer_len)
  {
    num_received = recv (sock, buffer, buffer_len, 0);
    if (num_received < 0)
      perror ("recv() failed");
    else if (num_received == 0)
      /* sender has closed connection */
    return EOF;
    else
    return num_received; /* might not be a full record*/
  }
  ```

  Again, can use `read()` instead of `recv()`.

**Datagram Sockets**

- Similar to stream sockets, except:
  - Sockets created using `SOCK_DGRAM` instead of `SOCK_STREAM`
  - No need for connection establishment and termination
  - Uses `recvfrom()` and `sendto()` in place of `recv()` (or `read()`) and `send()` (write()) respectively
  - Data sent in packets, not byte-stream oriented

**How to handle multiple connections?**

- Where do we get incoming data?
  - Stdin / keyboard input
  - Sockets (both datagram and stream)
  - Asynchronous: don’t know when data will arrive

- Solution: I/O multiplexing using `select()`
  - Efficient for our purposes (preferred method).
  - Solution: I/O multiplexing using polling
    - Inefficient - avoid.
  - Solution: Multithreading (POSIX et al)
    - More complex, but can scale further
    - Not covered, but feel free to try.

**I/O Multiplexing: Select (1)**

- Select()
  - Wait on multiple file descriptors/sockets and timeout
  - Application does not consume CPU while waiting
  - Return when file descriptors/sockets are ready to be read or written or they have an error, or timeout exceeded

- Advantages
  - Simple
  - At smaller scales (up to many dozens of descriptors), efficient

- Disadvantages
  - Does not scale to large number of descriptors/sockets
  - More awkward to use than it needs to be

**I/O Multiplexing: Select (2)**

```c
fd_set read_set;
struct timeval time_out;
while (1) {
  FD_ZERO (read_set);
  FD_SET (fileno(stdin), read_set);
  FD_SET (sock, read_set);
  time_out.tv_usec = 100000; time_out.tv_sec = 0;
  select_retval = select(MAX(stdin, sock) + 1, &read_set, NULL, NULL, &time_out);
  if (select_retval < 0) {
    perror ("select");
    abort();
  } else if (select_retval > 0) {
    if (FD_ISSET(stdin, read_set)) {
      if (read_user(user_buffer, user_buffer_len, &user_bytes_read) != 0)
        break;
    } else if (FD_ISSET(sock, read_set)) {
      if (receive_packets(buffer, buffer_len, &bytes_read) != 0)
        break;
    }
  }
}
```
I/O Multiplexing: Select (3)

Explanation:
- FD_ZERO(fd_set *set) -- clear a file descriptor set
- FD_SET(int fd, fd_set *set) -- adds fd to the set
- FD_CLR(int fd, fd_set *set) -- removes fd from the set
- FD_ISSET(int fd, fd_set *set) -- tests to see if fd is in the set

When does the call return?
- An error occurs on an fd.
- Data becomes available on an fd.
- (Other cases you needn't worry about now)

What do I check?
- You use FD_ISSET to see if a particular fd is set, and if it is then you need to handle it in some way.
- All non-active fds are cleared (so you need to reset the fd_set if you want to select again on a certain fd).
- More than one fd set may be set after select returns.

Common Mistakes + Hints

- Common mistakes:
  - Byte-ordering!
  - Use of select() - Separating records in TCP stream
  - Not coping with a read() that returns only part of a record
  - Server can’t bind because old connection hasn’t yet gone away
  - Not knowing what exactly gets transmitted on the wire
    - Use tcpdump (covered in Section next week) or Ethereal

- Hints:
  - Use man pages
  - Check out Web (ex. Beej’s Guide), programming books
  - Look at sample code.

Next Lecture

- IP: the “waist” of the Internet’s layering “hourglass”
- Read P&D: 3.1, 4.1.1, 4.1.2
- No lecture on Wed Sept. 13
- Next lecture is Mon Sept. 18

Socket API Reference

- struct hostent *gethostbyname(const char *name);
  - struct hostent
    - name: Official name of the host
      - char *h_name: A terminated array of alternate names for the host
      - int h_length: The length of the address in bytes
      - int h_addrtype: The type of address being returned; usually AF_INET
      - char **h_addr_list: A zero-terminated array of network addresses for the host. Host addresses are in Network Byte Order
      - int gethostbyname(char *hostname, size_t *size);
        - Returns the name of the computer that your program is running on
      - int socket(int domain, int type, int protocol);
      - int bind(int sockfd, struct sockaddr *, int addrlen);
      - int connect(int sockfd, const struct sockaddr *, int addrlen);
      - int accept(int sockfd, void *addr, int *addrlen);
      - int send(int sockfd, void *, int len, int flags);
      - int recv(int sockfd, void *buf, int len, int flags);
      - int setsockopt(int sockfd, int level, int optname, const void *optval, socklen_t optlen);

C Programming Reference

- void *memcpy(void *, void *, size_t num);
  - Copies num bytes from src buffer to memory location pointed by dest.
- void *memset(void *, int c, size_t num);
  - Sets the first num bytes pointed by the value specified by c parameter.
- void *memmove(void *, void *, size_t num);
  - Copies num bytes from memory areas s1 to s2. It returns s1. If s1 and s2 overlap, all bytes are copied in a preserving manner.
- void perror(const char *s);
  - Print a system error message.
- void *select(int nfds, fd_set *, fd_set *, fd_set *, struct timeval *timeout);