Sockets Programming

Introduction

© Dr. Ayman Abdel-Hamid, CS4254 Spring 2006

Outline

• Sockets API and abstraction
• Simple Daytime client
• Wrapper functions
• Simple Daytime Server

Sockets API

API is Application Programming Interface

• Sockets API defines interface between application and transport layer
  ➢ two processes communicate by sending data into socket, reading data out of socket
• Socket interface gives a file system like abstraction to the capabilities of the network
• Each transport protocol offers a set of services
  ➢ The socket API provides the abstraction to access these services
• The API defines function calls to create, close, read and write to/from a socket

Sockets Abstraction

The socket is the basic abstraction for network communication in the socket API

➢ Defines an endpoint of communication for a process
➢ Operating system maintains information about the socket and its connection
➢ Application references the socket for sends, receives, etc
Simple Daytime Client

1. Source code available from [http://www.unpbook.com](http://www.unpbook.com)
2. Read `README` file first!
3. Source file is `daytimetcpcli.c`
4. Include “unp.h”
   - Textbook’s header file
   - Includes system headers needed by most network programs
   - Defines various constants such as `MAXLINE`
5. Create TCP Socket
   - `sockfd = socket(AF_INET, SOCK_STREAM, 0)`
   - Returns a small integer descriptor used to identify socket
   - If returned value < 0 then error

Socket Descriptors

- Operating system maintains a set of socket descriptors for each process
  - Note that socket descriptors are shared by threads
- Three data structures
  - Socket descriptor table
  - Socket data structure
  - Address data structure

Simple Daytime Client

1. Specify Server IP Address and Port
   - Fill an `Internet socket address structure` with server’s IP address and port
   - Set entire structure to zero first using `bzero`
   - Set address family to `AF_INET`
   - Set port number to 13 (well-known port for daytime server on host supporting this service)
   - Set IP address to value specified as command line argument
     - `argv[1]`
     - `inet_pton` to presentation to numeric, converts ASCII dotted-decimal command line argument (128.82.4.66) to proper format
   - `hton` to host to network short

2. Establish connection with server
   - `Connect(sockfd, (SA *) &servaddr, sizeof(servaddr))`
   - Establish a TCP connection with server specified by socket address structure pointed to by second argument
   - Specify length of socket address structure as third argument
   - `-SA` is defined to be `struct sockaddr` in `unp.h`
3. Read and Display server reply
   - Server reply normally a 26-byte string of the form `Mon May 26 20:58:40 2003`
   - TCP a `byte-stream` protocol, always code the `read` in a loop and terminate loop when `read` returns 0 (other end closed connection) or value less than 0 (error)
Simple Daytime Client

- Terminate program
  - Exit terminates the program `exit (0)`
  - Unix closes all open descriptors when a process terminates
  - TCP socket closed
- Program protocol dependent on IPv4, will see later how to change to IPv6 and even make it protocol independent

Error Handling: Wrapper Functions

- Check every function call for error return
- In previous example, check for errors from `socket`, `inet_pton`, `connect`, `read`, and `fputs`
  - When error occurs, call textbook functions `err_quit` and `err_sys` to print an error message and terminate the program
- Define wrapper functions in `lib/wrapsock.c`
  - Unix `errno` value
    - When an error occurs in a Unix function, global variable `errno` is set to a positive value indicating the type of error and the function normally returns -1
    - `err_sys` function looks at `errno` and prints corresponding error message (e.g., connection timed out)

Simple Daytime Server

- Source code in `daytimetcpsrv.c`
- Create a TCP Socket
  - Identical to client code
- Bind server well-known port to socket
  - Fill an Internet socket address structure
  - Call `Bind` (wrapper function) ➔ local protocol address bound to socket
  - Specify IP address as `INADDR_ANY`: accept client connection on any interface (if server has multiple interfaces)
- Convert socket to listening socket
  - Socket becomes a listening socket on which incoming connections from clients will be accepted by the kernel
  - `LISTENQ` (defined in `unph.h`) specifies the maximum number of client connections the kernel will queue for this listening descriptor

- Accept client connection, send reply
  - Server is put to sleep (blocks) in the call to `accept`
  - After connection accepted, the call returns and the return value is a new descriptor called the connected descriptor
  - New descriptor used for communication with the new client
- Terminate connection
  - Initiate a TCP connection termination sequence
- Some Comments
  - Server handles one client at a time
  - If multiple client connections arrive at about the same time, kernel queues them up, up to some limit, and returns them to accept one at a time (An example of an iterative server, other options?)
IPv4 Socket Address Structure

```c
struct in_addr {
    in_addr_t s_addr; // 32-bit, IPv4 network byte order (unsigned)
};
```

```c
struct sockaddr_in {
    uint8_t sin_len;   /* unsigned 8 bit integer */
    sa_family_t sin_family; /* AF_INET */
    in_port_t sin_port; /* 16 bit TCP or UDP port number */
    struct in_addr sin_addr; /* 32 bit IPv4 address */
    char sin_zero[8]; /* unused */
};
```

```c
struct sockaddr_in servaddr;
servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
```

Generic Socket Address Structure

- A socket address structure always passed by reference when passed as an argument to any socket function.
- How to declare the pointer that is passed?
- Define a generic socket address structure

```c
struct sockaddr {
    uint8_t sa_len;   /* unsigned 8 bit integer */
    sa_family_t sa_family; /* AF_INET */
    char sa_data[14]; /* protocol specific address */
};
```

Prototype for bind

```c
int bind (int, struct sockaddr * socklen_t)
```

```c
struct sockaddr_in serv;
bind (sockfd, (struct sockaddr *) &serv, sizeof(serv));
```

Or #define SA struct sockaddr

```c
bind (sockfd, (SA *) &serv, sizeof(serv));
```

Value-Result Arguments

- Length of socket passed as an argument
- Method by which length is passed depends on which direction the structure is being passed (from process to kernel, or vice versa)
- Value-only: bind, connect, sendto (from process to kernel)
- Value-Result: accept, recvfrom, getsockname, getpeercname (from kernel to process, pass a pointer to an integer containing size)
  - Tells process how much information kernel actually stored

```c
struct sockaddr_in clientaddr;
socklen_t len;
int sockfd, connectfd;
len = sizeof(clientaddr);
connectfd = accept(listenfd, (SA *) &clientaddr, &len);
```

Byte Ordering Functions

- Two ways to store 2 bytes (16-bit integer) in memory
  - Low-order byte at starting address ➔ little-endian byte order
  - High-order byte at starting address ➔ big-endian byte order
- In a big-endian computer ➔ store 4F52
  - Stored as 4F52 ➔ 4F is stored at storage address 1000, 52 will be at address 1001, for example
- In a little-endian system ➔ store 4F52
  - it would be stored as 524F (52 at address 1000, 4F at 1001)
- Byte order used by a given system known as host byte order
- Network programmers use network byte order
- Internet protocol uses big-endian byte ordering for integers (port number and IP address)
Byte Ordering Functions 2/4

Little-endian byte order:
- High-order byte
- Address A
- Increasing memory
- Address A+1
- Low-order byte

Big-endian byte order:
- High-order byte
- Address A
- Increasing memory
- Address A+1
- Low-order byte

Byte Ordering Functions 3/4

Sample program to figure out little-endian or big-endian machine

Source code in byteorder.c

Byte Ordering Functions 4/4

To convert between byte orders
- Return value in network byte order
  - htons (s for short word 2 bytes)
  - htonl (l for long word 4 bytes)
- Return value in host byte order
  - ntohs
  - ntohl

Must call appropriate function to convert between host and network byte order

On systems that have the same ordering as the Internet protocols, four functions usually defined as null macros

servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
servaddr.sin_port = htons(13);

Byte Manipulation Functions

#include <string.h>

void bzero (void *dest, size_t nbytes);
// sets specified number of bytes to 0 in the destination

void bcopy (const void *src, void *dest, size_t nbytes);
// moves specified number of bytes from source to destination

void bcmp (const void *ptr1, const void *ptr2, size_t nbytes);
// compares two arbitrary byte strings, return value is zero if two byte strings are identical, otherwise, nonzero
Address Conversion Functions 1/2

Convert an IPv4 address from a dotted-decimal string “206.168.112.96” to a 32-bit network byte order binary value

#include <arpa/inet.h>

int inet_aton (const char* strptr, struct in_addr *addrptr);
// return 1 if string was valid, 0 on error. Address stored in *addrptr

in_addr_t inet_addr (const char * strptr);
// returns 32 bit binary network byte order IPv4 address, currently deprecated

char * inet_nota (struct in_addr inaddr);
//returns pointer to dotted-decimal string

Address Conversion Functions 2/2

To handle both IPv4 and IPv6 addresses

#include <arpa/inet.h>

int inet_pton (int family, const char* strptr, void *addrptr);
// return 1 if OK, 0 on error. 0 if not a valid presentation, -1 on error, Address
// stored in *addrptr

Const char * inet_ntop (int family, const void* addrptr, char *strptr,
size_t len);
// return pointer to result if OK, NULL on error

if (inet_pton(AF_INET, argv[1], &servaddr.sin_addr) <= 0)
    err_quit("inet_pton error for %s", argv[1]);

ptr = inet_ntop (AF_INET,&addr.sin_addr,str,sizeof(str));

Reading and Writing Functions 1/2

- int send (int socket, char *message, int msg_len, int flags) (TCP)
- int sendto (int socket, void *msg, int len, int flags, struct sockaddr *
to, int tolen ); (UDP)
- int write(int socket, void *msg, int len); /* TCP */
- int recv (int socket, char *buffer, int buf_len, int flags) (TCP)
- int recvfrom(int socket, void *msg, int len, int flags, struct sockaddr *rom, int *fromlen); (UDP)
- int read(int socket, void *msg, int len); (TCP)

Reading and Writing Functions 2/2

- Stream sockets (TCP sockets) exhibit a behavior with read and
  write that differs from normal file I/O
- A read or write on a stream socket might input or output fewer
  bytes than requested (not an error)

- readn function
- writen function
- readline function