Sockets

Based on

Section 17.3 of Computer Networking with Internet Protocols and Technology, by William Stallings, Prentice Hall.

book chapter 12.6.2
Sockets

The concept of sockets and sockets programming was developed in the 1980s in the UNIX environment as the Berkeley Sockets Interface.

- A socket enables communication between a client and server process and may be connection-oriented or connectionless.

- The Berkeley Sockets Interface is the de facto standard application programming interface (API) for developing networking applications.

- Windows Sockets (WinSock) is based on the Berkeley specification.

- The sockets API provides generic access to interprocess communications services.
Sockets

1. Server creates a named socket.

2. Client creates an unnamed socket and requests a connection.

3. Client makes a connection. Server retains original named socket.
Sockets

- TCP and UDP header includes source port and destination port fields, IP header includes IP address.
  - TCP/UDP: The port values identify the respective users (applications) of the two TCP entities.
  - IP (IPv4 and IPv6): header includes source address and destination address fields.
    - these IP addresses identify the respective host systems.

Definition of a Socket

- The concatenation of a port value and an IP address forms a socket, which is unique throughout the Internet.
Sockets

- The socket is used to define an API, which is a generic communication interface for writing programs that use TCP or UDP.

- In practice, when used as an API, a socket is identified by the triple (protocol, local address, local process).

- The local address is an IP address and the local process is a port number. Because port numbers are unique within a system, the port number implies the protocol (TCP or UDP).
The Sockets API recognizes two types of sockets:

- **Stream sockets (SOCK_STREAM)**
  - make use of TCP, which provides a connection-oriented reliable data transfer.
  - with stream sockets, all blocks of data sent between a pair of sockets are guaranteed for delivery and arrive in the order that they were sent.
The Sockets API recognizes two types of sockets:

- Datagram sockets, (SOCK_DGRAM)
  - make use of UDP, which does not provide the connection-oriented features of TCP.
  - with datagram sockets, delivery is not guaranteed, nor is order necessarily preserved.

There is a third type of socket provided by the Sockets API: raw sockets, (SOCK_RAW)

- Raw sockets
  - allow direct access to lower layer protocols, such as IP.
Sockets

- Socket Interface Calls

- To use sockets, it is a three-step process:
  - 1. Socket Setup
  - 2. Socket Connection
  - 3. Socket Communication

- Any program that uses sockets must include
  - /usr/include/sys/types.h
  - /usr/include/sys/socket.h
Sockets

The typical TCP client’s communication involves four basic steps:

1. Create a TCP socket using `socket()`.
2. Establish a connection to the server using `connect()`.
3. Communicate using `send()` and `recv()`.
4. Close the connection with `close()`.
Sockets

Socket Setup

The first step in using Sockets is to create a new socket using the socket() command. There are three parameters:

1. The *protocol family* is always PF_INET for the TCP/IP protocol suite.
2. The *type* specifies whether this is a stream or datagram socket.
3. The *protocol* specifies either TCP or UDP.

The reason that both type and protocol need to be specified is to allow additional transport-level protocols to be included in a future implementation.
Sockets

- After socket is created, it must have an address to listen to.

- The `bind()` function binds a socket to a socket address. The address has the structure:

```c
struct sockaddr_in {
    short int sin_family; // Address family (TCP/IP)
    unsigned short int sin_port; // Port number
    struct in_addr sin_addr; // Internet address
    unsigned char sin_zero[8]; // Same size as struct sockaddr
};
```
Sockets

Socket Connection

Stream socket

- once the socket is created, a connection must be set up to a remote socket.
- one side functions as a client, and requests a connection to the other side, which acts as a server.
Sockets

The server side of a connection setup requires two steps:

1. a server application issues a `listen()`,
   - indicates that socket is ready to accept incoming connections.
   - parameter `backlog` is the number of connections allowed on the incoming queue.
   - Each incoming connections is placed in this queue until a matching `accept()` is issued by the server side.
Sockets

The server side of a connection setup requires two steps:

2. the `accept()` call is used to remove one request from the queue.

   - If the queue is empty, the `accept()` blocks the process until a connection request arrives.
   - If there is a waiting call, then `accept()` returns a new file descriptor for the connection.
   - This creates a new socket, which has the IP address and port number of the remote party, the IP address of this system, and a new port number.
Sockets

- A client application issues a `connect()` that specifies both a local socket and the address of a remote socket.

- If the connection attempt is unsuccessful `connect()` returns the value 1.

- If the attempt is successful, `connect()` returns a 0 and fills in the file descriptor parameter to include the IP address and port number of the local and foreign sockets.

- Recall that the remote port number may differ from that specified in the `foreignAddress` parameter because the port number is changed on the remote host.
Sockets

Socket Communication

For stream communication, the functions `send()` and `recv()` are used to send or receive data over the connection identified by the sockfd parameter.

In the `send()` call, the `*msg` parameter points to the block of data to be sent and the `len` parameter specifies the number of bytes to be sent.

The `flags` parameter contains control flags, typically set to 0.

The `send()` call returns the number of bytes sent, which may be less than the number specified in the `len` parameter.
Sockets

Socket Communication cont.

In the `recv()` call, the `*buf` parameter points to the buffer for storing incoming data, with an upper limit on the number of bytes set by the `len` parameter.

At any time, either side can close the connection with the `close()` call, which prevents further sends and receives. The `shutdown()` call allows the caller to terminate sending or receiving or both.
Socket System Calls

- **socket()**
  - Open communication endpoint

- **bind()**
  - Register well-known address with system

- **listen()**
  - Establish client’s connection; request queue size

- **accept()**
  - Accept first client connection request on the queue
  - Blocks until connection from client
  - `accept()` creates a new socket to server the new client request

- **receive()**
  - Process request

- **send()**
  - Data (request)
  - Send/receive data

- **close()**
  - Shutdown

**Figure 17.6** Socket System Calls for Connection-Oriented Protocol
Sockets

Datagram Communication

For datagram communication, the functions `sendto()` and `recvfrom()` are used.

- The `sendto()` call includes all the parameters of the `send()` call plus a specification of the destination address (IP address and port).
- Similarly, the `recvfrom()` call includes an address parameter, which is filled in when data are received.
Sockets

Example from socket tutorial at

http://www.linuxhowtos.org/C_C++/socket.htm

the server and client code for this site are shown below
server code

/* A simple server in the internet domain using TCP
The port number is passed as an argument */
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>

void error(char *msg)
{
    perror(msg);
    exit(1);
}
```c
int main(int argc, char *argv[]) {
    int sockfd, newsockfd, portno, clilen;
    char buffer[256];
    struct sockaddr_in serv_addr, cli_addr;
    int n;
    if (argc < 2) {
        fprintf(stderr,"ERROR, no port provided\n");
        exit(1);
    }
    sockfd = socket(AF_INET, SOCK_STREAM, 0);
    if (sockfd < 0)
        error("ERROR opening socket");
    bzero((char *) &serv_addr, sizeof(serv_addr));
    portno = atoi(argv[1]);
    serv_addr.sin_family = AF_INET;
    serv_addr.sin_addr.s_addr = INADDR_ANY;
    serv_addr.sin_port = htons(portno);
    if (bind(sockfd, (struct sockaddr *) &serv_addr,
             sizeof(serv_addr)) < 0)
        error("ERROR on binding");
    listen(sockfd,5);
    clilen = sizeof(cli_addr);
    newsockfd = accept(sockfd,
                        (struct sockaddr *) &cli_addr,
                        &clilen);
    if (newsockfd < 0)
        error("ERROR on accept");
    bzero(buffer,256);
    n = read(newsockfd,buffer,255);
    if (n < 0) error("ERROR reading from socket");
    printf("Here is the message: %s\n",buffer);
    n = write(newsockfd,"I got your message",18);
    if (n < 0) error("ERROR writing to socket");
    return 0;
}
```
client code

```c
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netinet/in.h>
#include <netdb.h>

void error(char *msg)
{
    perror(msg);
    exit(0);
}
```
int main(int argc, char *argv[]) {
    int sockfd, portno, n;
    struct sockaddr_in serv_addr;
    struct hostent *server;

    char buffer[256];
    if (argc < 3) {
        fprintf(stderr,"usage %s hostname port\n", argv[0]);
        exit(0);
    }
    portno = atoi(argv[2]);
    sockfd = socket(AF_INET, SOCK_STREAM, 0);
    if (sockfd < 0)
        error("ERROR opening socket");
    server = gethostbyname(argv[1]);
    if (server == NULL) {
        fprintf(stderr,"ERROR, no such host\n");
        exit(0);
    }
bzero((char *) &serv_addr, sizeof(serv_addr));
serv_addr.sin_family = AF_INET;
bcopy((char *)server->h_addr,
    (char *)&serv_addr.sin_addr.s_addr,
    server->h_length);
serv_addr.sin_port = htons(portno);
if (connect(sockfd,&serv_addr,sizeof(serv_addr)) < 0)
    error("ERROR connecting");
printf("Please enter the message: ");
bzero(buffer,256);
fgets(buffer,255,stdin);
n = write(sockfd,buffer,strlen(buffer));
if (n < 0)
    error("ERROR writing to socket");
bzero(buffer,256);
n = read(sockfd,buffer,255);
if (n < 0)
    error("ERROR reading from socket");
printf("%s\n",buffer);
return 0;
Sockets

Another example
Sample Application

/* TCPEchoClient4.c */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

#include "Practical.h"

int main(int argc, char *argv[]) {
    if (argc < 3 || argc > 4) // Test for correct number of arguments
        DieWithUserMessage("Parameter(s)",
            "<Server Address> <Echo Word> [<Server Port>]");

    char *servIP = argv[1]; // First arg: server IP address (dotted quad)
    char *echoString = argv[2]; // Second arg: string to echo

    // Third arg (optional): server port (numeric). 7 is well-known echo port
    in_port_t servPort = (argc == 4) ? atoi(argv[3]) : 7;

    // Create a reliable, stream socket using TCP
    int sock = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
    if (sock < 0)
        DieWithSystemMessage("socket() failed");
// Construct the server address structure
struct sockaddr_in servAddr; // Server address
memset(&servAddr, 0, sizeof(servAddr)); // Zero out structure
servAddr.sin_family = AF_INET; // IPv4 address family
// Convert address
int rtnVal = inet_pton(AF_INET, servIP, &servAddr.sin_addr.s_addr);
if (rtnVal == 0)
    DieWithUserMessage("inet_pton() failed", "invalid address string");
else if (rtnVal < 0)
    DieWithSystemMessage("inet_pton() failed");
servAddr.sin_port = htons(servPort); // Server port

// Establish the connection to the echo server
if (connect(sock, (struct sockaddr *) &servAddr, sizeof(servAddr)) < 0)
    DieWithSystemMessage("connect() failed");

size_t echoStringLen = strlen(echoString); // Determine input length

// Send the string to the server
ssize_t numBytes = send(sock, echoString, echoStringLen, 0);
if (numBytes < 0)
    DieWithSystemMessage("send() failed");
else if (numBytes != echoStringLen)
    DieWithUserMessage("send()", "sent unexpected number of bytes");
// Receive the same string back from the server
unsigned int totalBytesRcvd = 0; // Count of total bytes received
fputs("Received: ", stdout); // Setup to print the echoed string

while (totalBytesRcvd < echoStringLen)
{
    char buffer[BUFSIZE]; // I/O buffer
    /* Receive up to the buffer size (minus 1 to leave space for
     * a null terminator) bytes from the sender */
    numBytes = recv(sock, buffer, BUFSIZE - 1, 0);
    if (numBytes < 0)
        DieWithSystemMessage("recv() failed");
    else if (numBytes == 0)
        DieWithUserMessage("recv()", "connection closed prematurely");

    totalBytesRcvd += numBytes; // Keep tally of total bytes
    buffer[numBytes] = '\0'; // Terminate the string!
    fputs(buffer, stdout); // Print the echo buffer
}

fputc('
', stdout); // Print a final linefeed

close(sock);
exit(0);