TCP/IP Sockets in C

Great and inexpensive book:

Michael J. Donahoo
Kenneth L. Calvert

Morgan Kaufmann Publisher
$14.95 Paperback

Computer Chat

- How do we make computers talk?

- How are they interconnected?

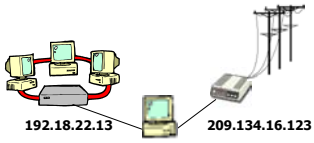
Internet Protocol (IP)

Internet Protocol (IP)

- Datagram (packet) protocol
- Best-effort service
  - Loss
  - Reordering
  - Duplication
  - Delay
- Host-to-host delivery

IP Address

- 32-bit identifier (IPv4, IPv6=128 bits)
- Dotted-quad: 192.118.56.25
- www.mkp.com -> 167.208.101.28
- Identifies a host interface (not a host)



Transport Protocols

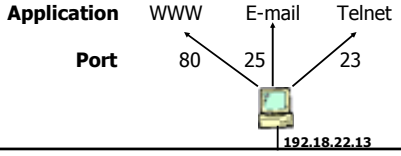
Best-effort not sufficient!

- Add services on top of IP
- User Datagram Protocol (UDP)
  - Data checksum
  - Best-effort
- Transmission Control Protocol (TCP)
  - Data checksum
  - Reliable byte-stream delivery
  - Flow and congestion control

Ports

Identifying the ultimate destination

- IP addresses identify hosts
- Host has many applications
- Ports (16-bit identifier) 1-65,535



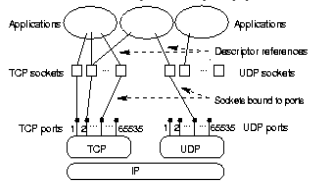
Socket

How does one speak TCP/IP?

- Sockets provides interface to TCP/IP
- Generic interface for many protocols

Sockets

- Identified by protocol and local/remote address/port
- Applications may refer to many sockets
- Sockets accessed by many applications



TCP/IP Sockets

- mySock = socket(family, type, protocol);
- TCP/IP-specific sockets

	Family	Type	Protocol
TCP	PF_INET	SOCK_STREAM	IPPROTO_TCP
UDP		SOCK_DGRAM	IPPROTO_UDP

- Socket reference
  - File (socket) descriptor in UNIX

Specifying Addresses

Generic

```
struct sockaddr
{
        unsigned short sa_family;  /* Address family (e.g., AF_INET) */
        char sa_data[14];          /* Protocol-specific address information */
};
```

IP Specific

```
struct sockaddr_in
{
        unsigned short sin_family; /* Internet protocol (AF_INET) */
        unsigned short sin_port;   /* Port (16-bits) */
        struct in_addr sin_addr;   /* Internet address (32-bits) */
        char sin_zero[8];          /* Not used */
};
struct in_addr
{
        unsigned long s_addr;      /* Internet address (32-bits) */
};
```

Clients and Servers

- Client: Initiates the connection

Client: Bob Server: Jane

"Hi. I'm Bob." →

← "Hi, Bob. I'm Jane"

"Nice to meet you, Jane." →

- Server: Passively waits to respond

TCP Client/Server Interaction

Server starts by getting ready to receive client connections…

Client
1. Create a TCP socket
2. Establish connection
3. Communicate
4. Close the connection

Server
1. Create a TCP socket
2. Assign a port to socket
3. Set socket to listen
4. Repeatedly:
   a. Accept new connection
   b. Communicate
   c. Close the connection

TCP Client/Server Interaction

```
/* Create socket for incoming connections */
if ((servSock = socket(PF_INET, SOCK_STREAM, IPPROTO_TCP)) < 0)
    DieWithError("socket() failed");
```

Client

1. Create a TCP socket
2. Establish connection
3. Communicate
4. Close the connection

Server

1. Create a TCP socket
2. Bind socket to a port
3. Set socket to listen
4. Repeatedly:
   a. Accept new connection
   b. Communicate
   c. Close the connection

TCP Client/Server Interaction

```
echoServAddr.sin_family = AF_INET;                /* Internet address family */
echoServAddr.sin_addr.s_addr = htonl(INADDR_ANY);/* Any incoming interface */
echoServAddr.sin_port = htons(echoServPort);      /* Local port */

if (bind(servSock, (struct sockaddr *) &echoServAddr, sizeof(echoServAddr)) < 0)
    DieWithError("bind() failed");
```

Client

1. Create a TCP socket
2. Establish connection
3. Communicate
4. Close the connection

Server

1. Create a TCP socket
2. Bind socket to a port
3. Set socket to listen
4. Repeatedly:
   a. Accept new connection
   b. Communicate
   c. Close the connection

TCP Client/Server Interaction

```
/* Mark the socket so it will listen for incoming connections */
if (listen(servSock, MAXPENDING) < 0)
    DieWithError("listen() failed");
```

Client

1. Create a TCP socket
2. Establish connection
3. Communicate
4. Close the connection

Server

1. Create a TCP socket
2. Bind socket to a port
3. Set socket to listen
4. Repeatedly:
   a. Accept new connection
   b. Communicate
   c. Close the connection

TCP Client/Server Interaction

```
for (;;) /* Run forever */
{
    clntLen = sizeof(echoClntAddr);

    if ((clntSock=accept(servSock,(struct sockaddr *)&echoClntAddr,&clntLen)) < 0)
        DieWithError("accept() failed");
```

Client

1. Create a TCP socket
2. Establish connection
3. Communicate
4. Close the connection

Server

1. Create a TCP socket
2. Bind socket to a port
3. Set socket to listen
4. Repeatedly:
   a. Accept new connection
   b. Communicate
   c. Close the connection

TCP Client/Server Interaction

Server is now blocked waiting for connection from a client

Client

1. Create a TCP socket
2. Establish connection
3. Communicate
4. Close the connection

Server

1. Create a TCP socket
2. Bind socket to a port
3. Set socket to listen
4. Repeatedly:
   a. Accept new connection
   b. Communicate
   c. Close the connection

TCP Client/Server Interaction

Later, a client decides to talk to the server…

Client

1. Create a TCP socket
2. Establish connection
3. Communicate
4. Close the connection

Server

1. Create a TCP socket
2. Bind socket to a port
3. Set socket to listen
4. Repeatedly:
   a. Accept new connection
   b. Communicate
   c. Close the connection

TCP Client/Server Interaction

```
/* Create a reliable, stream socket using TCP */
if ((sock = socket(PF_INET, SOCK_STREAM, IPPROTO_TCP)) < 0)
    DieWithError("socket() failed");
```

Client

1. Create a TCP socket
2. Establish connection
3. Communicate
4. Close the connection

Server

1. Create a TCP socket
2. Bind socket to a port
3. Set socket to listen
4. Repeatedly:
   a. Accept new connection
   b. Communicate
   c. Close the connection

TCP Client/Server Interaction

```
echoServAddr.sin_family      = AF_INET;              /* Internet address family */
echoServAddr.sin_addr.s_addr = inet_addr(servIP); /* Server IP address */
echoServAddr.sin_port        = htons(echoServPort); /* Server port */

if (connect(sock, (struct sockaddr *) &echoServAddr, sizeof(echoServAddr)) < 0)
    DieWithError("connect() failed");
```

Client

1. Create a TCP socket
2. Establish connection
3. Communicate
4. Close the connection

Server

1. Create a TCP socket
2. Bind socket to a port
3. Set socket to listen
4. Repeatedly:
   a. Accept new connection
   b. Communicate
   c. Close the connection

TCP Client/Server Interaction

```
echoStringLen = strlen(echoString);          /* Determine input length */

/* Send the string to the server */
  if (send(sock, echoString, echoStringLen, 0) != echoStringLen)
      DieWithError("send() sent a different number of bytes than expected");
```

Client

1. Create a TCP socket
2. Establish connection
3. Communicate
4. Close the connection

Server

1. Create a TCP socket
2. Bind socket to a port
3. Set socket to listen
4. Repeatedly:
   a. Accept new connection
   b. Communicate
   c. Close the connection

TCP Client/Server Interaction

```
if ((clntSock=accept(servSock,(struct sockaddr *)&echoClntAddr,&clntLen)) < 0)
      DieWithError("accept() failed");
```

Client

1. Create a TCP socket
2. Establish connection
3. Communicate
4. Close the connection

Server

1. Create a TCP socket
2. Bind socket to a port
3. Set socket to listen
4. Repeatedly:
   a. Accept new connection
   b. Communicate
   c. Close the connection

TCP Client/Server Interaction

```
/* Receive message from client */
if ((recvMsgSize = recv(clntSocket, echoBuffer, RCVBUFSIZE, 0)) < 0)
    DieWithError("recv() failed");
```

Client

1. Create a TCP socket
2. Establish connection
3. Communicate
4. Close the connection

Server

1. Create a TCP socket
2. Bind socket to a port
3. Set socket to listen
4. Repeatedly:
   a. Accept new connection
   b. Communicate
   c. Close the connection

TCP Client/Server Interaction

```
close(sock);
```

```
close(clntSocket)
```

Client

1. Create a TCP socket
2. Establish connection
3. Communicate
4. Close the connection

Server

1. Create a TCP socket
2. Bind socket to a port
3. Set socket to listen
4. Repeatedly:
   a. Accept new connection
   b. Communicate
   c. Close the connection

TCP Tidbits

- Client knows server address and port
- No correlation between `send()` and `recv()`

Client	Server
send("Hello Bob")	
	recv() -> "Hello "
	recv() -> "Bob"
	send("Hi ")
	send("Jane")
recv() -> "Hi Jane"	

Closing a Connection

- close() used to delimit communication
- Analogous to EOF

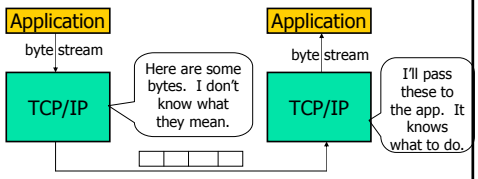
Client	Server
send(*string*)	
	recv(*buffer*)
while (not received entire string)	while(client has not closed connection)
recv(*buffer*)	send(*buffer*)
send(*buffer*)	recv(*buffer*)
close(*socket*)	
	close(*client socket*)

Constructing Messages

…beyond simple strings

TCP/IP Byte Transport

- TCP/IP protocols transports bytes



- Application protocol provides semantics

Application Protocol

- Encode information in bytes
- Sender and receiver must agree on semantics
- Data encoding
  - Primitive types: strings, integers, and etc.
  - Composed types: message with fields

Primitive Types

- String
  - Character encoding: ASCII, Unicode, UTF
  - Delimit: length vs. termination character

	0	77	0	111	0	109	0	10
	M		o		m		\n	
3	77		111		109			

Primitive Types

- Integer
  - Strings of character encoded decimal digits

49	55	57	57	56	55	48	10
'1'	'7'	'9'	'9'	'8'	'7'	'0'	\n

  - Advantage: 1. Human readable
    2. Arbitrary size
  - Disadvantage: 1. Inefficient
    2. Arithmetic manipulation

Primitive Types

- Integer
  - Native representation

Little-Endian

0	0	92	246

23,798

Big-Endian

246	92	0	0

4-byte two's-complement integer

  - Network byte order (Big Endian)
    - Use for multi-byte, binary data exchange
    - htonl(), htons(), ntohl(), ntohs()

Message Composition

- Message composed of fields
  - Fixed length fields

integer	short	short

  - Variable length fields

M	i	k	e		1	2	\n

"Beware the bytes of padding" -- Julius Caesar, Shakespeare

- Architecture alignment restrictions
- Compiler pads structs to accommodate

```
struct tst {
  short x;
  int y;
  short z;
};
```

x	[pad]	y	z	[pad]

- Problem: Alignment restrictions vary
- Solution: 1) Rearrange struct members
  2) Serialize struct by member

TCPEchoClient.c

```
#include <stdio.h>      /* for printf() and fprintf() */
#include <sys/socket.h> /* for socket(), connect(), send(), and recv() */
#include <arpa/inet.h>  /* for sockaddr_in and inet_addr() */
#include <stdlib.h>     /* for atoi() */
#include <string.h>     /* for memset() */
#include <unistd.h>     /* for close() */

#define RCVBUFSIZE 32   /* Size of receive buffer */

void DieWithError(char *errorMessage);  /* Error handling function */

int main(int argc, char *argv[])
{
    int sock;                       /* Socket descriptor */
    struct sockaddr_in echoServAddr; /* Echo server address */
    unsigned short echoServPort;    /* Echo server port */
    char *servIP;                   /* Server IP address (dotted quad) */
    char *echoString;               /* String to send to echo server */
    char echoBuffer[RCVBUFSIZE];    /* Buffer for echo string */
    unsigned int echoStringLen;     /* Length of string to echo */
    int bytesRcvd, totalBytesRcvd;  /* Bytes read in single recv()
                                       and total bytes read */

    if ((argc < 3) || (argc > 4))    /* Test for correct number of arguments */
    {
        fprintf(stderr, "Usage: %s <Server IP> <Echo Word> [<Echo Port>]\n",
                argv[0]);
        exit(1);
    }
```

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

if (argc == 4)
    echoServPort = atoi(argv[3]); /* Use given port, if any */
else
    echoServPort = 7;  /* 7 is the well-known port for the echo service */

/* Create a reliable, stream socket using TCP */
if ((sock = socket(PF_INET, SOCK_STREAM, IPPROTO_TCP)) < 0)
    DieWithError("socket() failed");

/* Construct the server address structure */
memset(&echoServAddr, 0, sizeof(echoServAddr));     /* Zero out structure */
echoServAddr.sin_family      = AF_INET;             /* Internet address family */
echoServAddr.sin_addr.s_addr = inet_addr(servIP);   /* Server IP address */
echoServAddr.sin_port        = htons(echoServPort); /* Server port */

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

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

/* Send the string to the server */
if (send(sock, echoString, echoStringLen, 0) != echoStringLen)
    DieWithError("send() sent a different number of bytes than expected");
```

```
/* receive the same string back from the server */
    totalBytesRcvd = 0;
    printf("Received: ");                /* Setup to print the echoed string */
    while (totalBytesRcvd < echoStringLen)
    {
        /* Receive up to the buffer size (minus 1 to leave space for
           a null terminator) bytes from the sender */
        if ((bytesRcvd = recv(sock, echoBuffer, RCVBUFSIZE - 1, 0)) <= 0)
            DieWithError("recv() failed or connection closed prematurely");
        totalBytesRcvd += bytesRcvd;   /* Keep tally of total bytes */
        echoBuffer[bytesRcvd] = '\0';  /* Terminate the string! */
        printf(echoBuffer);            /* Print the echo buffer */
    }

    printf("\n");    /* Print a final linefeed */

    close(sock);
    exit(0);
}

void DieWithError(char *errorMessage)
{
    perror(errorMessage);
    exit(1);
}
```

TCPEchoServer.c

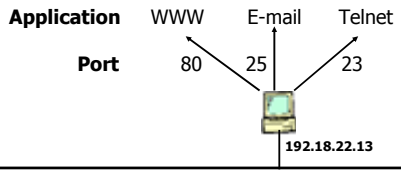
```
#include <stdio.h>      /* for printf() and fprintf() */
#include <sys/socket.h> /* for socket(), bind(), and connect() */
#include <arpa/inet.h>  /* for sockaddr_in and inet_ntoa() */
#include <stdlib.h>     /* for atoi() */
#include <string.h>     /* for memset() */
#include <unistd.h>     /* for close() */

#define MAXPENDING 5    /* Maximum outstanding connection requests */

void DieWithError(char *errorMessage);  /* Error handling function */
void HandleTCPClient(int clntSocket);   /* TCP client handling function */

int main(int argc, char *argv[])
{
    int servSock;                    /* Socket descriptor for server */
    int clntSock;                    /* Socket descriptor for client */
    struct sockaddr_in echoServAddr; /* Local address */
    struct sockaddr_in echoClntAddr; /* Client address */
    unsigned short echoServPort;     /* Server port */
    unsigned int clntLen;            /* Length of client address data structure */

    if (argc != 2)     /* Test for correct number of arguments */
    {
        fprintf(stderr, "Usage:  %s <Server Port>\n", argv[0]);
        exit(1);
    }

    echoServPort = atoi(argv[1]);  /* First arg:  local port */
```

```
/* Create socket for incoming connections */
    if ((servSock = socket(PF_INET, SOCK_STREAM, IPPROTO_TCP)) < 0)
        DieWithError("socket() failed");

    /* Construct local address structure */
    memset(&echoServAddr, 0, sizeof(echoServAddr));   /* Zero out structure */
    echoServAddr.sin_family = AF_INET;                /* Internet address family */
    echoServAddr.sin_addr.s_addr = htonl(INADDR_ANY); /* Any incoming interface */
    echoServAddr.sin_port = htons(echoServPort);      /* Local port */

    /* Bind to the local address */
    if (bind(servSock, (struct sockaddr *) &echoServAddr, sizeof(echoServAddr)) < 0)
        DieWithError("bind() failed");

    /* Mark the socket so it will listen for incoming connections */
    if (listen(servSock, MAXPENDING) < 0)
        DieWithError("listen() failed");
```

```
    for (;;) /* Run forever */
    {
        /* Set the size of the in-out parameter */
        clntLen = sizeof(echoClntAddr);

        /* Wait for a client to connect */
        if ((clntSock = accept(servSock, (struct sockaddr *) &echoClntAddr,
                               &clntLen)) < 0)
            DieWithError("accept() failed");

        /* clntSock is connected to a client! */

        printf("Handling client %s\n", inet_ntoa(echoClntAddr.sin_addr));

        HandleTCPClient(clntSock);
    }
    /* NOT REACHED */
}
```

Transport Protocols

- Best-effort, host-to-host insufficient
- Transport protocols add services to IP
- Ports make end-to-end protocols



Transport Protocols

- User Datagram Protocol (UDP)
  - Adds ports and data checksum to IP
  - Best effort
- Transmission Control Protocol (TCP)
  - Adds ports and data checksum to IP
  - Reliable byte stream delivery
  - Flow and congestion control
  - Etc.

Socket Protocols

- Protocol Family
  - PF_INET – Internet Protocol Family
- Protocol Type
  - SOCK_STREAM – Reliable byte stream
  - SOCK_DGRAM – Best effort datagram
- Protocol
  - IPPROTO_TCP
  - IPPROTO_UDP

TCP Client

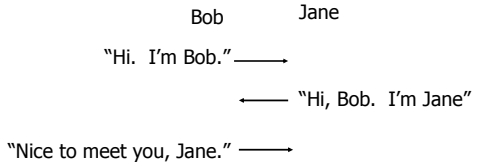
1. Create a TCP socket using `socket()`
2. Establish a connection to the server using `connect()`
3. Communicate using `send()` and `recv()`
4. Close the connection with `close()`

TCP Server

1. Create a TCP socket using `socket()`
2. Assign a port number to the socket with `bind()`
3. Tell the system to allow connections to be made to that port, using `listen()`
4. Repeatedly do the following:
   a. Call `accept()` to get a new socket for each client connection.
   b. Communicate with the client via that new socket, using `send()` and `recv()`
   c. Close the client connection using `close()`

Protocols

- We want to make computers "talk"
- Protocol: Agreement on exchange



TCPEchoServer-Fork.c

```
#include "TCPEchoServer.h"  /* TCP echo server includes */
#include <sys/wait.h>       /* for waitpid() */

int main(int argc, char *argv[])
{
    int servSock;                   /* Socket descriptor for server */
    int clntSock;                   /* Socket descriptor for client */
    unsigned short echoServPort;    /* Server port */
    pid_t processID;                /* Process ID from fork() */
    unsigned int childProcCount = 0; /* Number of child processes */

    if (argc != 2)     /* Test for correct number of arguments */
    {
        fprintf(stderr, "Usage:  %s <Server Port>\n", argv[0]);
        exit(1);
    }

    echoServPort = atoi(argv[1]);  /* First arg:  local port */

    servSock = CreateTCPServerSocket(echoServPort);
```

```
    for (;;) /* Run forever */
    {
        clntSock = AcceptTCPConnection(servSock);
        /* Fork child process and report any errors */
        if ((processID = fork()) < 0)
            DieWithError("fork() failed");
        else if (processID == 0)  /* If this is the child process */
        {
            close(servSock);   /* Child closes parent socket */
            HandleTCPClient(clntSock);

            exit(0);           /* Child process terminates */
        }

        printf("with child process: %d\n", (int) processID);
        close(clntSock);       /* Parent closes child socket descriptor */
        childProcCount++;      /* Increment number of outstanding child processes */

        while (childProcCount) /* Clean up all zombies */
        {
            processID = waitpid((pid_t) -1, NULL, WNOHANG);  /* Non-blocking wait */
            if (processID < 0)  /* waitpid() error? */
                DieWithError("waitpid() failed");
            else if (processID == 0)  /* No zombie to wait on */
                break;
            else
                childProcCount--;  /* Cleaned up after a child */
        }
    }
}
```

```
#define MAXPENDING 5    /* Maximum outstanding connection requests */

void DieWithError(char *errorMessage);  /* Error handling function */

int CreateTCPServerSocket(unsigned short port)
{
    int sock;                        /* socket to create */
    struct sockaddr_in echoServAddr; /* Local address */

    /* Create socket for incoming connections */
    if ((sock = socket(PF_INET, SOCK_STREAM, IPPROTO_TCP)) < 0)
        DieWithError("socket() failed");

    /* Construct local address structure */
    memset(&echoServAddr, 0, sizeof(echoServAddr));   /* Zero out structure */
    echoServAddr.sin_family = AF_INET;                /* Internet address family */
    echoServAddr.sin_addr.s_addr = htonl(INADDR_ANY); /* Any incoming interface */
    echoServAddr.sin_port = htons(port);              /* Local port */

    /* Bind to the local address */
    if (bind(sock, (struct sockaddr *) &echoServAddr, sizeof(echoServAddr)) < 0)
        DieWithError("bind() failed");

    /* Mark the socket so it will listen for incoming connections */
    if (listen(sock, MAXPENDING) < 0)
        DieWithError("listen() failed");

    return sock;
}
```

```
#include <stdio.h>      /* for printf() */
#include <sys/socket.h> /* for accept() */
#include <arpa/inet.h>  /* for sockaddr_in and inet_ntoa() */

void DieWithError(char *errorMessage);  /* Error handling function */

int AcceptTCPConnection( int servSock )
{
    int clntSock;                    /* Socket descriptor for client */
    struct sockaddr_in echoClntAddr; /* Client address */
    unsigned int clntLen;            /* Length of client address data structure */

    /* Set the size of the in-out parameter */
    clntLen = sizeof(echoClntAddr);

    /* Wait for a client to connect */
    if ((clntSock = accept(servSock, (struct sockaddr *) &echoClntAddr,
          &clntLen)) < 0)
        DieWithError("accept() failed");

    /* clntSock is connected to a client! */

    printf("Handling client %s\n", inet_ntoa(echoClntAddr.sin_addr));

    return clntSock;
}
```

Resources/References

- http://cs.ecs.baylor.edu/~donahoo/practical/CSockets
- http://www.ecst.csuchico.edu/~beej/guide
  - Network programming
  - Unix Interprocess communication