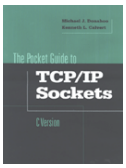
TCP/IP Sockets in C: Practical Guide for Programmers



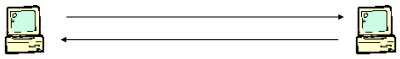
Michael J. Donahoo
Kenneth L. Calvert

Morgan Kaufmann Publisher
$14.95 Paperback

http://www.cs.uga.edu/~maria/classes/4730-Fall-2009/project3sockets/

Computer Chat

- How do we make computers talk?



- How are they interconnected?

  Internet Protocol (IP)

A protocol – an agreement on how to communicate – e.g., how is the information structured (length, address location)

TCP/IP protocol suit.

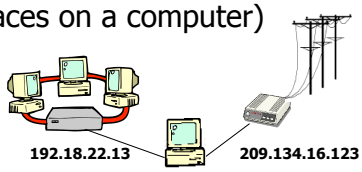
Internet Protocol (IP)

- Datagram (packet) protocol
- Best-effort service
  - Loss
  - Reordering
  - Duplication
  - Delay
- Host-to-host delivery (we are not at 'application' level yet – more later)

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 computer. could multiple interfaces on a computer)

Like a Street Address

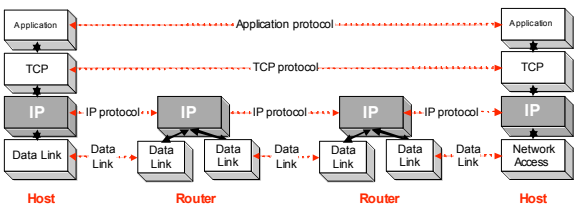


Transport Protocols

Best-effort not sufficient!

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

- Organize Protocols in Layers



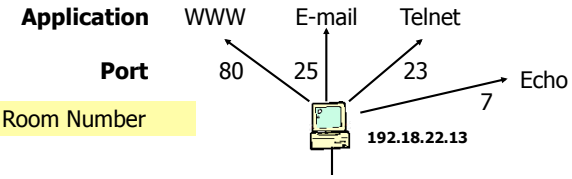
Ports

Internet Assigned Numbers Authority

Well-known	1-1,023
Registered	1,024-49,151
Dynamic	49,152-65,535

Identifying the ultimate destination

- IP addresses identify hosts
- Host has many applications
- Ports (16-bit identifier) 1-65,535 (about 2000 are reserved).



Like a Room Number

Internet Phone Book

- Domain Name Service (DNS)
  - Data base maps domain names to internet addresses

Socket

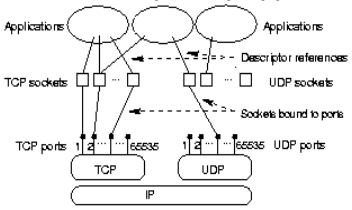
How does one speak TCP/IP?

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

Like a File Descriptor for a file

Sockets

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



TCP/IP Sockets: Creates end point (and flavor)

- Int 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

Type: Semantics of transmission: e.g., is it reliable, best-effort, boundaries (packets, streams)

Specifying Addresses

Generic

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

Family	Blob (14 bytes)		
2 bytes	2 bytes 16 bits	4 bytes 32 bits	8 bytes
Family	Port	Internet Address	Unused

16 byte/128 bit data structure

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

struct sockaddr_in  /* TCP/IP structure form */
    {
          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) */
    };
```

Historically the intent was that a single protocol family might support multiple address families

Note:

- In Theory: Protocol family to socket (PF_INET) for internet family are different from the addressing scheme (AF_INET) – here it is 1-1 but does not need to be.
- In Practice: AF_XXXX and PF_XXXX constants are interchangeable.
  - Values are the same AF_XXXX = PF_XXXX

Clients and Servers

- Server: Waits until needed
- Client: Initiates the connection

Client: Bob Server: Jane

"Hi. I'm Bob." →

← "Hi, Bob. I'm Jane"

"Nice to meet you, Jane." →

Two separate programs – on the samemachine or remote

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

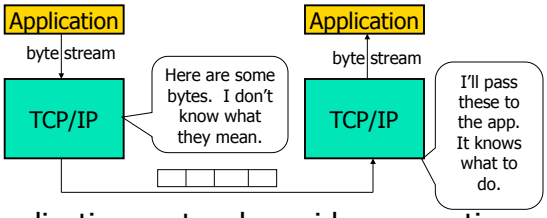
Client	Server
send(*string*)	
	recv(*buffer*)
while (not received entire string)	while(client has not closed connection)
recv(*buffer*)	send(*buffer*)
send(*buffer*)	recv(*buffer*)
close(*socket*)	
	close(*client socket*)

Constructing Messages

… something to think about…

TCP/IP Byte Transport

- TCP/IP protocols transports bytes



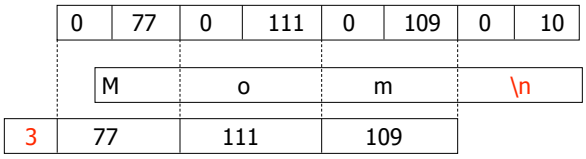
- Application protocol provides semantics
  - TCP does not examine or modify bytes

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



Primitive Types

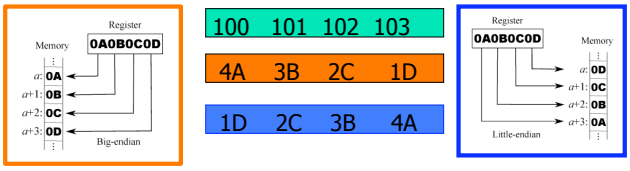
- Integer
  - Strings of character encoded decimal digits

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

  - Advantage:
    - 1. Human readable
    - 2. Arbitrary size (in contrast to native integer format -fixed).
  - Disadvantage:
    - 1. Inefficient (space 1 byte (char) to represent 0-9, while only really need 4 bits).
    - 2. Arithmetic manipulation (must convert to integers)

Byte Ordering

- Computers orders bytes differently.
  - Big endian machines stores high order byte at lowest address (Big end first). Example: Intel Processor, Network
  - Little endian stores low order byte at lowest address (Little end first). Example: Motorola processor



Network Byte Order Functions

- Network byte order is in Big-Endian and used for multi-byte, binary data exchange
- htonl(), htons(), ntohl(), ntohs()
  - Converts between **host byte order** and **network byte order**
    - 'h' = host byte order
    - 'n' = network byte order (big endian)
    - 'l' = long (4 bytes), converts IP addresses
    - 's' = short (2 bytes), converts port numbers
- **Implementation:** If the byte order is already big-endian, then the [hn]to[ns][sl]( ) functions are no-ops.

Message Composition

- Message composed of fields
  - Fixed-length fields

integer	short	short

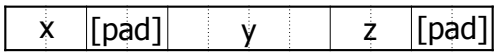
  - Variable-length fields

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;
};
```



- 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 */
}
```

Socket I/O: accept()

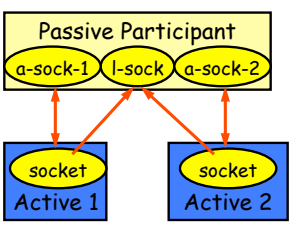
```
struct sockaddr_in cli;              /* used by accept() */
int newfd;                         /* returned by accept() */
int cli_len = sizeof(cli);     /* used by accept() */

newfd = accept(fd, (struct sockaddr*) &cli, &cli_len);
if(newfd < 0) {
      perror("accept");
      exit(1);
}
```

- How does the server know which client it is?
  - **cli.sin_addr.s_addr** contains the client's ***IP address***
  - **cli.sin_port** contains the client's ***port number***
- accept() is blocking (What does that mean?)
- Why does ***accept*** need to return a new descriptor?

Connection setup

- Passive(server) participant
  - step 1: listen (for incoming requests)
  - step 3: accept (a request)
  - step 4: data transfer
- Active participant
  - step 2: request & establish connection
  - step 4: data transfer
- The accepted connection is on a new socket already connected to the active participant
- The old socket continues to listen for other active participants



Dealing with blocking calls

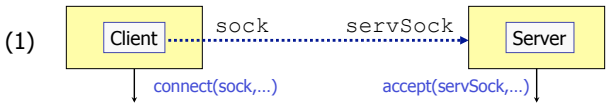
- How do we deal with blocking calls?
  - accept(): until a connection comes in
  - connect(): until the connection is established
  - recv(): until a packet (of data) is received
  - send(): until data is pushed into socket's buffer
    - Q: why not until received?
- For simple programs, blocking is convenient
- What about more complex programs?
  - multiple connections
  - simultaneous sends and receives
  - simultaneously doing non-networking processing

Dealing w/ blocking (cont'd)

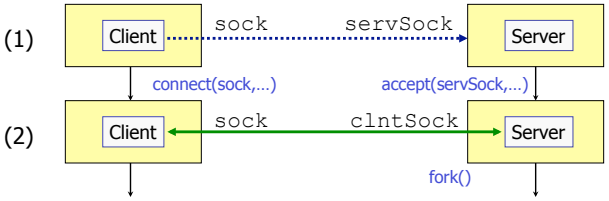
- Options:
  - Implement multi-tasking: create multi-process or multi-threaded code
  - turn off the blocking feature (e.g., using the fcntl file-descriptor control function)
  - use the select() function call (on your own…)

Why Multitasking?

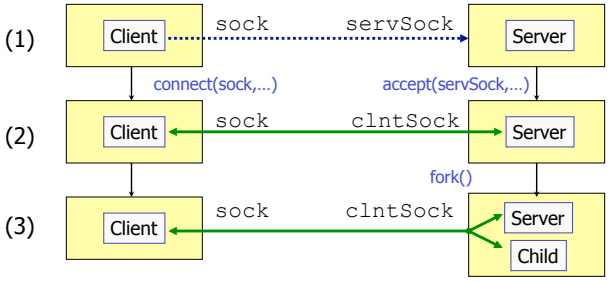
- Previously server could only server one client at a time (iteratively).
- For computational intensive tasks how can we server multiple clients?
  - Multitasking allows servers to farm out work to other processes or threads each executing independently (possibly as a copy of the original server).



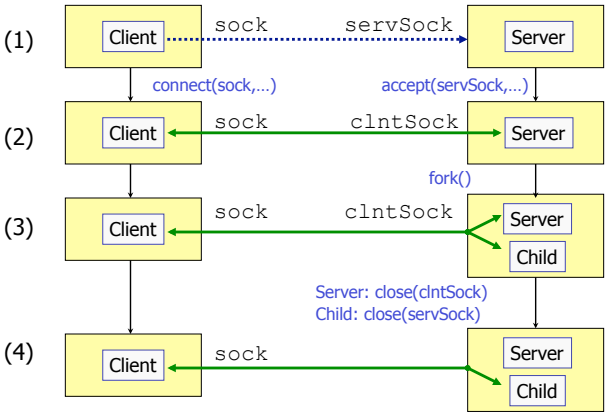
(1) Server runs forever and listens for connections at a specific port and repeatedly accepts incoming connections from clients



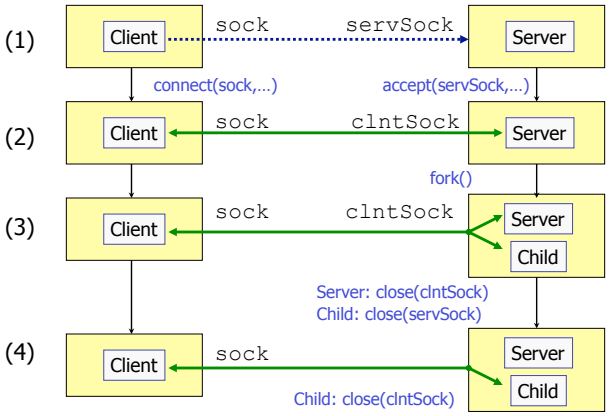
(2) When a client connects (after accept()) it creates a new process to handle that connection via fork()



(3) Fork creates a new child process and copies socket descriptors.



(4) The child now deals with the Client and it does not listen on the servSock any more so it closes that connection. The Server only needs to listen for other new Clients so it closes clntSock



(4) The Child then service the Client and later closes his connection. Child terminates via exit().

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