References

- The Pocket Guide TCP/IP Sockets – C Version, Michael J. Donahoo, Kenneth L. Calvert
- TCP/IP ¼ÒÄÏÇÁ·Î±×·¡¹Ö (C ¹öÀü), ¹ÚÁØÃ¶¿ª
- http://cs.ecs.baylor.edu/~donahoo/practical/CSockets/
- http://www.mkp.com/socket

- W. R. Stevens, UNIX Network Programming, Vol I, Prentice-Hall,
- Douglas E. Comer, Internetworking with TCP/IP, Vol. I, Prentice-Hall,
Introduction(1)

- Internet Protocol
  - Internet base protocol
  - Network layer (Host-to-Host transmission)
- Transmission Control Protocol
  - Transport layer (Process-to-Process transmission)
  - reliable transfer
- User Datagram Protocol
  - Transport layer
  - unreliable transfer

- IETF (Internet Engineering Task Force)
## Introduction(2)

<table>
<thead>
<tr>
<th>ISO OSI Model</th>
<th>Internet Model</th>
<th>User Model</th>
</tr>
</thead>
<tbody>
<tr>
<td>7 Application</td>
<td>Process / Application</td>
<td>Network Applications</td>
</tr>
<tr>
<td>6 Presentation</td>
<td>Transport</td>
<td></td>
</tr>
<tr>
<td>5 Session</td>
<td></td>
<td>Network Infrastructure</td>
</tr>
<tr>
<td>4 Transport</td>
<td>Internet</td>
<td></td>
</tr>
<tr>
<td>3 Network</td>
<td>Network</td>
<td></td>
</tr>
<tr>
<td>2 Data link</td>
<td>Link</td>
<td></td>
</tr>
<tr>
<td>1 Physical</td>
<td>Physical</td>
<td></td>
</tr>
</tbody>
</table>
### Introduction (3)

<table>
<thead>
<tr>
<th>Layer</th>
<th>Protocols</th>
</tr>
</thead>
<tbody>
<tr>
<td>Process/Application</td>
<td>TELNET (login)</td>
</tr>
<tr>
<td></td>
<td>FTP files</td>
</tr>
<tr>
<td></td>
<td>SMTP (mail)</td>
</tr>
<tr>
<td></td>
<td>DNS (names)</td>
</tr>
<tr>
<td></td>
<td>NTP (time)</td>
</tr>
<tr>
<td></td>
<td>NFS (files)</td>
</tr>
<tr>
<td>Transport</td>
<td>TCP</td>
</tr>
<tr>
<td></td>
<td>UDP</td>
</tr>
<tr>
<td>Internet</td>
<td>IP</td>
</tr>
<tr>
<td>Network</td>
<td>Ethernet</td>
</tr>
<tr>
<td></td>
<td>ISO 8802-2</td>
</tr>
<tr>
<td></td>
<td>X.25</td>
</tr>
<tr>
<td></td>
<td>ISO 7776</td>
</tr>
<tr>
<td></td>
<td>SLIP</td>
</tr>
<tr>
<td></td>
<td>PPP</td>
</tr>
<tr>
<td>Link</td>
<td>ISO 8802-3</td>
</tr>
<tr>
<td></td>
<td>IEE 802.3 (CSMA/CD)</td>
</tr>
<tr>
<td></td>
<td>ISO 8802-5 (HDL)</td>
</tr>
<tr>
<td></td>
<td>X.25 LAPB (serial)</td>
</tr>
<tr>
<td>Physical</td>
<td>(various)</td>
</tr>
</tbody>
</table>

[2-2] IP Protocol stacks
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
- 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)
  - Add ports and data checksum to IP
  - Best-effort
- Transmission Control Protocol (TCP)
  - Add ports and data checksum to IP
  - 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)

<table>
<thead>
<tr>
<th>Application</th>
<th>WWW</th>
<th>E-mail</th>
<th>Telnet</th>
</tr>
</thead>
<tbody>
<tr>
<td>Port</td>
<td>80</td>
<td>25</td>
<td>23</td>
</tr>
</tbody>
</table>

192.18.22.13
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
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/IP Sockets

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

<table>
<thead>
<tr>
<th>Family</th>
<th>Type</th>
<th>Protocol</th>
</tr>
</thead>
<tbody>
<tr>
<td>TCP</td>
<td>PF_INET</td>
<td>SOCK_STREAM</td>
</tr>
<tr>
<td>UDP</td>
<td></td>
<td>SOCK_DGRAM</td>
</tr>
</tbody>
</table>

- Socket reference
  - File (socket) descriptor in UNIX
  - Socket handle in WinSock
Specifying Addresses

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

- 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) */
  }
### Specifying Addresses

<table>
<thead>
<tr>
<th>sockaddr</th>
<th>Family</th>
<th>Blob</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>2 bytes</td>
<td>4 bytes</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>sockaddr_in</th>
<th>Family</th>
<th>Port</th>
<th>Internet address</th>
<th>Blob</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Not used</td>
</tr>
</tbody>
</table>
Clients and Servers

- **Client:** Initiates the connection

  Client: Bob

  "Hi. I’m Bob."

  Server: Jane

  "Hi, Bob. I’m Jane"

  "Nice to meet you, Jane."

- **Server:** Passively waits to respond
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()`
TCP Client/Server Interaction

Server starts by getting ready to receive client connections...

<table>
<thead>
<tr>
<th>Client</th>
<th>Server</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Create a TCP socket</td>
<td>1. Create a TCP socket</td>
</tr>
<tr>
<td>2. Establish connection</td>
<td>2. Assign a port to socket</td>
</tr>
<tr>
<td>3. Communicate</td>
<td>3. Set socket to listen</td>
</tr>
<tr>
<td>4. Close the connection</td>
<td>4. Repeatedly:</td>
</tr>
<tr>
<td></td>
<td>a. Accept new connection</td>
</tr>
<tr>
<td></td>
<td>b. Communicate</td>
</tr>
<tr>
<td></td>
<td>c. Close the connection</td>
</tr>
</tbody>
</table>
TCP Client/Server Interaction

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

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

bind system call

```c
#include <sys/types.h>
#include <sys/socket.h>

int bind(int sockfd, struct sockaddr *myaddr, int addrlen);
```

- **myaddr**: protocol-specific address
- **addrlen**: size of address structure
bind() Socket System Calls

- Three uses of bind

  - Servers register their well-known address with the system. It tells the system "this is my address and any messages received for this address are to be given to me".

  - A client can register a specific address for itself.

  - A connectionless client needs to assure that the system assigns it some unique address, so that the other end (the server) has a valid address to its response to.

- bind system call fills in the local-addr and local-process elements of the association 5-tuple(protocol, load-addr, local-process, foreign-addr, foreign-process)
TCP Client/Server Interaction

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

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

- **listen** system call

  ```c
  int listen(int sockfd, in backlog);
  ```

- `sockfd` : socket descriptor
- `backlog` : specifies how many connection requests can be queued by the system while it waits for the server to execute the `accept` system call.

- **listen** system call is used by a connection-oriented server to indicate that it is willing to receive connections.

- In the time that takes a server to handle the request of an `accept`, it is possible for additional connection requests to arrive from other clients.
TCP Client/Server Interaction

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

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

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

**accept** system call

```c
#include <sys/types.h>
#include <sys/socket.h>

int accept(int sockfd, struct sockaddr *peer, int *addrlen);
```

- **sockfd**: socket descriptor
- **peer**: used to return the address of the connection peer process (client)
- **addrlen**: value-result argument

- caller sets **addrlen** to the size of the sockaddr structure whose address is passed as the peer argument.

- on return, **addrlen** contains the actual number of bytes that the system call stores in the peer argument.
accept() Socket System Calls

- An actual connection from some client process is waited for by having the server executing the `accept` system call.

- `accept` takes the first connection request on the queue and creates another socket with the same properties as `sockfd`. If there is connection requests pending, this call blocks until one arrives.

- `accept` automatically creates a new socket descriptor, assuming the server is a concurrent server.
# TCP Client/Server Interaction

<table>
<thead>
<tr>
<th>Client</th>
<th>Server</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Create a TCP socket</td>
<td>1. Create a TCP socket</td>
</tr>
<tr>
<td>2. Establish connection</td>
<td>2. Bind socket to a port</td>
</tr>
<tr>
<td>3. Communicate</td>
<td>3. Set socket to listen</td>
</tr>
<tr>
<td>4. Close the connection</td>
<td>4. Repeatedly:</td>
</tr>
<tr>
<td></td>
<td>a. Accept new connection</td>
</tr>
<tr>
<td></td>
<td>b. Communicate</td>
</tr>
<tr>
<td></td>
<td>c. Close the connection</td>
</tr>
</tbody>
</table>

Server is now blocked waiting for connection from a client.
## TCP Client/Server Interaction

Later, a client decides to talk to the server...

<table>
<thead>
<tr>
<th>Client</th>
<th>Server</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Create a TCP socket</td>
<td>1. Create a TCP socket</td>
</tr>
<tr>
<td>2. Establish connection</td>
<td>2. Bind socket to a port</td>
</tr>
<tr>
<td>3. Communicate</td>
<td>3. Set socket to listen</td>
</tr>
<tr>
<td>4. Close the connection</td>
<td>4. Repeatedly:</td>
</tr>
<tr>
<td></td>
<td>a. Accept new connection</td>
</tr>
<tr>
<td></td>
<td>b. Communicate</td>
</tr>
<tr>
<td></td>
<td>c. Close the connection</td>
</tr>
</tbody>
</table>
TCP Client/Server Interaction

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

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

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

```c
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");
```
connect() Socket System Calls

connect system call

```c
#include <sys/types.h>
#include <sys/socket.h>

int connect(int sockfd, struct sockaddr *servaddr, int addrlen);
```

- `servaddr` : pointer to a socket address
- `addrlen` : size of socket address

- the `connect` system call results in the actual establishment of a connection between the local system and foreign system.
connect() Socket System Calls

- The connection typically causes these four elements of the association 5-tuple to be assigned: local-addr, local-process, foreign-addr, foreign-process.

- For a connectionless protocol, all that is done by the `connect` system call is to store the servaddr specified by the process, so that the system knows where to send any future data that process writes to the sockfd descriptor.

- One advantage of connecting a socket associated with a connectionless protocol is we don't need to specify the destination address for every datagram that we send.
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

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

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

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

close(sock);
close(clntSocket)
Socket System Calls for Connection Oriented Protocol

Server
- socket()
- bind()
- listen()
- accept()

blocks until connection from client

connection establishment
- read()
- process request
- write()

Client
- socket()
- connect()
- write()

read()
TCP Tidbits

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

---

**Client**

```
send("Hello Bob")
```

**Server**

```
recv() -> "Hello 
recv() -> "Bob"
send("Hi 
send("Jane")
recv() -> "Hi Jane"
```
Closing a Connection

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

**Client**

```plaintext
send(string)
while (not received entire string)
  recv(buffer)
  send(buffer)
close(socket)
```

**Server**

```plaintext
recv(buffer)
while(client has not closed connection)
  send(buffer)
  recv(buffer)

close(client socket)
```
Concurrent Server

- concurrent server: typical scenario

```c
int sockfd, newsockfd;
if((sockfd =socket(...)) < 0)
    err_sys("socket error");
if(bind(sockfd,...) < 0)
    err_sys("bind error");
if(listen(sockfd, 5) < 0)
    err_sys("listen error");
for(;;) {
    newsockfd =accept(sockfd,...); /* blocks */
    if(newsockfd < 0)
        err_sys("accept error");
    if(fork() == 0) {
        close(sockfd); /* child */
        doit(newsockfd); /* process the request */
        exit(0);
    }
    close(newsockfd); /* parent */
}
Concurrent Server

- Concurrent Server

When a connection request is received and accepted, the process forks, with the child process servicing the connection and the parent process waiting for another connection request.
Iterative Server

- Iterative server: typical scenario

```c
int sockfd, newsockfd;

if((sockfd = socket(...) < 0)
   err_sys("socket error");
if(bind(sockfd,...) < 0)
   err_sys("bind error");
if(listen(sockfd, 5) < 0)
   err_sys("listen error");
for(;;) {
   newsockfd = accept(sockfd,...); /* blocks */
   if(newsockfd < 0)
      err_sys("accept error");
   doit(newsockfd); /* process the request */
   close(newsockfd);
}
```
UDP Client/Server Interaction (Connectionless Protocol)

Server starts by getting ready to receive client messages...

**Client**
- Create a UDP socket
- Send a message
- Receive a reply
- Close the connection

**Server**
- Create a UDP socket
- Assign a port to socket
- Repeatedly:
  - Wait for message to arrive
  - Reply
- Close the connection
UDP Client/Server Interaction
(Connectionless Protocol)

Client
- Create a UDP socket
- Send a message
- Receive a reply
- Close the connection

Server
- Create a UDP socket
- Assign a port to socket
- Repeatedly:
  - Wait for message to arrive
  - Reply
- Close the connection

/* Create socket for sending/receiving datagrams */
if ((sock = socket(PF_INET, SOCK_DGRAM, IPPROTO_UDP)) < 0)
    DieWithError("socket() failed");
UDP Client/Server Interaction
(Connectionless Protocol)

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

Client
- Create a UDP socket
- Send a message
- Receive a reply
- Close the connection

Server
- Create a UDP socket
- Assign a port to socket
- Repeatedly:
  - Wait for message to arrive
  - Reply
- Close the connection
UDP Client/Server Interaction (Connectionless Protocol)

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

    /* Block until receive message from a client */
    if ((recvMsgSize = recvfrom(sock, echoBuffer, ECHOMAX, 0, 
     (struct sockaddr *) &echoClntAddr, &cliAddrLen)) < 0)
        DieWithError("recvfrom() failed");

Client
- Create a UDP socket
- Send a message
- Receive a reply
- Close the connection

Server
- Create a UDP socket
- Assign a port to socket
- Repeatedly:
  - Wait for message to arrive
  - Reply
- Close the connection
UDP Client/Server Interaction
(Connectionless Protocol)

Client
- Create a UDP socket
- Send a message
- Receive a reply
- Close the connection

Server
- Create a UDP socket
- Assign a port to socket
- Repeatedly:
  - Wait for message to arrive
  - Reply
- Close the connection

/* Create a datagram/UDP socket */
if ((sock = socket(PF_INET, SOCK_DGRAM, IPPROTO_UDP)) < 0)
  DieWithError("socket() failed");
UDP Client/Server Interaction (Connectionless Protocol)

Client
- Create a UDP socket
- Send a message
- Receive a reply
- Close the connection

Server
- Create a UDP socket
- Assign a port to socket
- Repeatedly:
  - Wait for message to arrive
  - Reply
- Close the connection

/* Construct the server address structure */
    echoServAddr.sin_family = AF_INET;                 /* Internet addr family */
    echoServAddr.sin_addr.s_addr = inet_addr(servIP);  /* Server IP address */
    echoServAddr.sin_port   = htons(echoServPort);     /* Server port */

/* Send the string to the server */
    if (sendto(sock, echoString, echoStringLen, 0, (struct sockaddr *) &echoServAddr,
                sizeof(echoServAddr)) != echoStringLen)
        DieWithError("sendto() sent a different number of bytes than expected");
UDP Client/Server Interaction
(Connectionless Protocol)

/* Send received datagram back to the client */
if (sendto(sock, echoBuffer, recvMsgSize, 0,
    (struct sockaddr *) &echoClntAddr, sizeof(echoClntAddr)) != recvMsgSize)
    DieWithError("sendto() sent a different number of bytes than expected");

Client
- Create a UDP socket
- Send a message
- Receive a reply
- Close the connection

Server
- Create a UDP socket
- Assign a port to socket
- Repeatedly:
  - Wait for message to arrive
  - Reply
- Close the connection
**UDP Client/Server Interaction**  
*(Connectionless Protocol)*

---

**Client**
- Create a UDP socket
- Send a message
- Receive a reply
- Close the connection

**Server**
- Create a UDP socket
- Assign a port to socket
- Repeatedly:
  - Wait for message
  - Reply
- Close the connection

---

```c
/* Recv a response */
fromSize = sizeof(fromAddr);
if ((respStringLen = recvfrom(sock, echoBuffer, ECHOMAX, 0,
    (struct sockaddr *) &fromAddr, &fromSize)) != echoStringLen)
    DieWithError("recvfrom() failed");
```
UDP Client/Server Interaction
(Connectionless Protocol)

Client
- Create a UDP socket
- Send a message
- Receive a reply
- Close the connection

Server
- Create a UDP socket
- Assign a port to socket
- Repeatedly:
  - Wait for message to arrive
  - Reply
- Close the connection

close(sock);
Socket System Calls for Connectionless Protocol

**Server**
- `socket()`
- `bind()`
- `recvfrom()`

**Client**
- `socket()`
- `sendto()`
- `recvfrom()`

- `recvfrom()` blocks until data received from a client
- `data(request)`
- `data(reply)`
- `sendto()`
- `process request`

Timeline of socket system calls for connectionless protocol.
Constructing Messages

...beyond simple strings
TCP/IP Byte Transport

- TCP/IP protocols transports bytes

- Application protocol provides semantics

Here are some bytes. I don’t know what they mean.

I’ll pass these to the app. It knows what to do.
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

<p>| | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>77</td>
<td>111</td>
<td>109</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>M</td>
<td>o</td>
<td>m</td>
<td>\n</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>111</td>
<td></td>
<td></td>
<td>109</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Primitive Types

- Integer
  - Strings of character encoded decimal digits

<p>| | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>49</td>
<td>55</td>
<td>57</td>
<td>57</td>
<td>56</td>
<td>55</td>
<td>48</td>
<td>10</td>
</tr>
</tbody>
</table>

- Advantage: 1. Human readable
  2. Arbitrary size

- Disadvantage: 1. Inefficient
  2. Arithmetic manipulation
Primitive Types

- **Little-Endian**
  - Specifies that the least significant byte is stored in the lowest-memory address, which is the address of the data.
  - The Intel 80X86 and Pentium and DEC Alpha RISC processors are Little Endian.
  - Windows NT and OSF/1 are Little Endian.

- **Big-Endian**
  - Storing the most significant byte in the lowest memory address, which is the address of the data.
  - Motorola 680x0 microprocessors, Hewlett-Packard PA-RISC, and Sun SuperSPARC processors are Big Endian.
  - Most UNIXes (for example, all System V) and the Internet are Big Endian.
Primitive Types

- Integer
  - Native representation
    - Little-Endian: 0 0 92 246
    - Big-Endian: 246 92 0 0
      - 4-byte two’s-complement integer
    - 23,798
  - 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

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

- Problem: Alignment restrictions vary
- Solution: 1) Rearrange struct members
  2) Serialize struct by-member
```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 fami ly */
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);
}
#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 */
        {          /* Error handling function */
            fprintf(stderr, "Usage:  %s <Server Port>
", 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 */