TCP/IP Sockets in C: Practical Guide for Programmers

Michael J. Donahoo
Kenneth L. Calvert
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
  (not application-to-application)
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)
  - Data checksum
  - Best-effort
- Transmission Control Protocol (TCP)
  - Data checksum
  - Reliable byte-stream delivery
  - Flow and congestion control
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
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
- 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) */
  };

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

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

- **Client:** Initiates the connection

  Client: Bob

  "Hi. I’m Bob."

  Server: Jane

  "Hi, Bob. I’m Jane"

- **Server:** Passively waits to respond

  "Nice to meet you, Jane."
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

```c
/* 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

- 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

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

```c
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");
```
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

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

/* 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

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

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

/* 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)
TCP Tidbits

- Client must know the server’s address and port
- Server only needs to know its own 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

### Echo Client

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

### Echo Server

- `recv(buffer)`
- `while (client has not closed connection)`
  - `send(buffer)`
  - `recv(buffer)`
- `close(client socket)`
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>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>77</td>
<td>0</td>
<td>111</td>
<td>0</td>
<td>109</td>
<td>0</td>
</tr>
<tr>
<td>M</td>
<td>o</td>
<td>m</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>77</td>
<td>111</td>
<td>109</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>\n</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>
<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>
<td></td>
</tr>
<tr>
<td>'1'</td>
<td>'7'</td>
<td>'9'</td>
<td>'9'</td>
<td>'8'</td>
<td>'7'</td>
<td>'0'</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **Advantage:**
  1. Human readable
  2. Arbitrary size

- **Disadvantage:**
  1. Inefficient
  2. Arithmetic manipulation
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

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

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