Network Programming

- ISO OSI Model
- ISO OSI Model: Higher Layers
- Alphabet Soup of Protocols
- Alphabet Soup of Protocols Continued
- Applications of Protocols
- Internet Addressing
- Ports
- Some Well Known Ports
- Network Debugging Tools
- Example using `curl`
- Connecting to TCP Services Using `telnet`
- Example using `netcat`
- Connection Parameters
- Networking Variations
- Sockets Overview
- Socket API Overview
- Socket API Overview Continued
- TCP Connection Phone Analogy
- TCP Client Socket Usage
- TCP Server Socket Usage
- Communication Domains
- Common Socket Domains

- Socket Type
- Creating a Socket
- Socket Endpoint Addresses
- IP Socket Addresses
- `bind()` Call
- `listen()` Call
- `accept()` Call
- `connect()` Call
- Socket I/O
- Echo Server Log
- Example Connection-Oriented Server
- Connection-Oriented Server Continued
- Connection-Oriented Server Continued
- Example Connection-Oriented Client
- Example Connection-Oriented Client Continued
- Example Connection-Oriented Client Continued
- Auxiliary Routines: `gethostbyname()`
- `gethostbyname()` Log
- `gethostbyname()` Program
- Shutting Down a Socket
- Socket `shutdown()` Function
- Datagrams
- Sending Datagrams
- Receiving Datagrams

ISO OSI Model

International Standards Organization defines 7 layers of protocols known as the *Open Systems Interconnection* model.

1] **Physical Layer**: Electrical/physical standards. Example: physical Ethernet hardware like NICs and cabling.

2] **Data-Link Layer**: Divide a bit-stream into higher-level entities like packets. Error detections. Example: Ethernet protocol (CSMA/CD).

3] **Network Layer**: Includes concepts of destination addressing and routing. Must handle network congestion. Example: IP operates at this level.

4] **Transport Layer**: End-to-end reliability. Examples: TCP and UDP.

ISO OSI Model: Higher Layers

WRT TCP/IP protocols, the above layers are usually implemented within kernel-space. However, the following application layers usually operate in user-space. They do not really map well into the TCP/IP protocol suite.

5] **Session layer**: allows remote terminal access. Authentication.

6] **Presentation layer**: transformation services like compression.

7] **Application layer**: end-user program.

Alphabet Soup of Protocols

IPv4 *Internet Protocol version 4*
Reliable package delivery. Uses 32-bit addresses. Accessed using *raw sockets*.

IPv6 *Internet Protocol version 6*
Mid-90s redesign of IPv4. Uses 128-bit addresses. Accessed using *raw sockets*.

TCP *Transmission Control Protocol*
Connection oriented protocol. Provides a reliable full-duplex byte stream. Can use IPv4 or IPv6. Accessed using *stream sockets*.

UDP User Datagram Protocol
Connectionless protocol. Unreliable: no guarantee that datagrams reach destination. Can use IPv4 or IPv6. Accessed using *datagram sockets*.

Alphabet Soup of Protocols Continued

ICMP *Internet Control Message Protocol*
Used for error and control information by networking software. Also used by popular `ping` program.

ARP *Address Resolution Protocol*
Used for converting a IPv4 address into a hardware address (like a 6-byte ethernet address).

RARP *Reverse Address Resolution Protocol*
Used for converting a hardware address into a IPv4 address. Used to have diskless computers figure out their IPv4 address.

Applications of Protocols

- `ping` uses ICMP.
- `traceroute` uses ICMP and UDP.
- BOOTP (bootstrap protocol), DHCP (bootstrap protocol), NTP (time protocol), TFTP (trivial FTP), SNMP (network management) use UDP.
- SMTP (email), `telnet` (remote login), `ftp` (file transfer), HTTP (web protocol), NNTP (net news) use TCP.
- DNS (domain name system), NFS (network file system) Sun RPC (remote procedure call) use both UDP and TCP.

Internet Addressing

- IPv4 uses 32-bit binary addresses.
- Instead of an absolute number, IPv4 addresses are often represented in dotted notation as 4 decimal numbers separated by periods. Example: `128.226.6.4`.
- Domain name system used to map human-friendly hostnames into binary addresses.
- Domain name system functions as a distributed hierarchical database.
- Routines like `gethostbyname()`, `inet_addr`(), `inet_aton`() used for conversion.
- Multi-homed hosts may have multiple addresses and names.

Ports

- It is not enough to merely specify which host you want to connect to. It is also necessary to specify which program instance on the host you want to connect to.
- The program instance is referred to using a 16-bit number referred to as a *port number*.
- TCP and UDP use disjoint port number spaces.
- Ports 0-1023 are referred to as *well-known ports*. Ports 1024-49151 are *registered ports*. Ports 49152-65535 are *ephemeral ports*.
- Under Unix, ports 0-1023 can only be accessed by `root`.
- Ports can also be referred to by name. Translation between service name and port number using `getservbyname()`.

Some Well Known Ports

The file /etc/services contains a mapping from service names to the port numbers used. An extract for some well-known services:

```
echo            7/tcp
echo            7/udp
discard         9/tcp           sink null
discard         9/udp           sink null
discard         9/udp
daytime         13/tcp
daytime         13/udp
ftp-data        20/tcp
ftp             21/tcp
ssh             22/tcp                          # SSH Remote Login Protocol
ssh             22/udp
telnet          23/tcp
smtp            25/tcp          mail
time            37/tcp          timserver
time            37/udp          timserver
http            80/tcp          www             # WorldWideWeb HTTP
http            80/udp                          # HyperText Transfer Protocol
https           443/tcp                         # http protocol over TLS/SSL
https           443/udp
```

Network Debugging Tools

Since many network protocols use textual protocols, it is relatively easy to understand network interactions.

`telnet`

Specify the TCP port running the service as an additional argument.

`curl`

Wide variety of options.

`netcat`

Much enhanced network debugging tool.

Example using `curl`

```
$ curl -D bing.headers http://www.binghamton.edu >/dev/null
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100 59557    0 59557    0     0   139k      0 --:--:-- --:--:-- --:--:--  139k
$ cat bing.headers
HTTP/1.1 200 OK
Date: Wed, 20 Apr 2016 21:39:57 GMT
Server: Apache/2.4.12 (Unix) mod_python/3.5.0- Python/2.6.6 OpenSSL/1.0.1e-fips
X-Frame-Options: SAMEORIGIN
Set-Cookie: insideee_last_visit=1145828397; expires=Thu, 20-Apr-2017 21:39:57 GMT; Max-Age=31536000; path=/
Set-Cookie: insideee_last_activity=1461188397; expires=Thu, 20-Apr-2017 21:39:57 GMT; Max-Age=31536000; path=/
Set-Cookie: insideee_tracker=a%3A1%3A%7Bi%3A0%3Bs%3A5%3A%22index%22%3B%7D; path=/
Expires: Mon, 26 Jul 1997 05:00:00 GMT
Last-Modified: Wed, 20 Apr 2016 21:39:57 GMT
Pragma: no-cache
Vary: Accept-Encoding
Transfer-Encoding: chunked
Content-Type: text/html; charset=utf-8

$
```

Connecting to TCP Services Using `telnet`

Before Binghamton moved to gmail:

```
$ telnet mail.binghamton.edu 25
Trying 128.226.1.18...
Connected to mail.binghamton.edu.
Escape character is '^]'.
220 Binghamton ESMTP University SMTP Server.
Usage is monitored.  Spamming is NOT allowed.
HELP
HELP
214-This is Sendmail version 8.9.3
214-Topics:
 ....
214-For local information send email to ...
214 End of HELP info
QUIT
QUIT
221 bingnet2.cc.binghamton.edu closing connection
Connection closed by foreign host.
$
```

Example using `netcat`

```
$ nc -l 1234 &  #one-shot background TCP server
[2] 22213
$ echo ’hello nc’ | nc localhost 1234 #client
hello nc           #background server output
[2]+  Done                    nc -l 1234
$
$ printf "GET / HTTP/1.0\r\n\r\n" | nc zdu.binghamton.edu 80
HTTP/1.1 200 OK
Date: Wed, 20 Apr 2016 22:04:11 GMT
...
Content-Type: text/html;charset=UTF-8

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 3.2 Final//EN">
<html>
...
</body></html>
```

Connection Parameters

- A *connection* is defined by its 2 *endpoints*.
- Each *endpoint* specified by 2 numbers: a IP address and a port number.
- The two values specifying each endpoint: a IP address and a port number is referred to as a *socket*.
- A *socket pair* is a 4-tuple which specifies both endpoints of the connection.
- Routines in socket API successively fill in this 4-tuple.

Networking Variations

When setting up a connection, a program must decide:

- Whether it wants to act as a *client* or a *server*.
- Whether it needs *connection-oriented service* (using TCP) or *datagram service* (using UDP).
- Above gives rise to 4 basic combinations.
- When designing servers, another basic decision is whether to use a *concurrent* or *iterative* server.

Sockets Overview

- Sockets are referred to via file-descriptors and normal I/O routines (like `read()`, `write()`) operate on them.
- Sockets can be used for many protocols other than the TCP/IP protocol suite.
- Besides basic socket API, need auxiliary routines for looking up host addresses, port numbers for different services and protocol numbers.

Socket API Overview

- `socket()` creates a descriptor for use in network communication.
- `connect()` connects to a remote peer (used by a client).
- `close()` terminates communication and deallocates descriptor.
- `bind()` binds a local IP address and port to a socket.
- `listen()` places a socket in passive mode and sets a limit on the number of incoming TCP connections which can be enqueued.
- `accept()` accepts the next incoming connection (used by a server).

Socket API Overview Continued

- `recv()`, `recvmsg()`, `recvfrom()` receives the next incoming datagram.
- `send()`, `sendmsg()`, `sendto()` sends a outgoing datagram.
- `shutdown()` terminates a TCP connection in one or both directions.
- `getpeername()` returns remote endpoint address (hostaddress and port).
- `getsockopt()`, `setsockopt()` allows control over socket options.

TCP Connection Phone Analogy

- `socket()` creates a telephone.
- `bind()` sets your phone number.
- `listen()` turns on the ringer.
- `connect()` dials the published phone number.
- `accept()` picks up the phone (and gets the identity of the caller, somewhat like *caller-id*, except caller-id works before pick-up).
- DNS similar to telephone book. `gethostbyname()`, `getservbyname()` look up numbers in phone book.

TCP Client Socket Usage

1. `socket()`: create socket.
2. `connect()`: connect to server.
3. `write()`: send request.
4. `read()`: read response.
5. Loop back to 3 if necessary.
6. `close()`: close connection, free socket.

TCP Server Socket Usage

1. `socket()`: create socket.
2. `bind()` address (ip address, protocol number) to socket.
3. `listen()` in passive mode for incoming connections, queueing requests if necessary.
4. `accept()` incoming connection from client.
5. `read()` request from client.
6. `write()` response to client.
7. Loop back to 5 if necessary.
8. `close()` connection to client. Loop back to 4.

Communication Domains

Socket API evolved in an environment with a large number of network protocols including TCP/IP, Xerox XNS, IBM SNA, etc. Hence API tried to abstract out details of actual network protocols by defining a communication domain:

Protocol Family

The networking protocol being used.

Address Family

The format of addressing used for the protocol

The intent was that the same protocol could support different kinds of addresses. In practice, there was a 1:1 mapping and hence POSIX currently only specifies AF_* constants for address family constants.

Common Socket Domains

`AF_UNIX`

Intra-host communication via kernel. Address format is simply a pathname and address structure is of type `sockaddr_un`.

`AF_INET`

Inter-host communication via IPv4 networking. Address format is a 32-bit net address + 16-bit port number and address structure is of type `sockaddr_in`.

`AF_INET6`

Inter-host communication via IPv6 networking. Address format is a 128-bit net address + 16-bit port number and address structure is of type `sockaddr_in6`.

Socket Type

SOCK_DGRAM

Datagram (message boundaries preserved) connection-less communication without any guaranteed delivery. UDP internet transport protocol.

SOCK_STREAM

Stream oriented (no message boundaries) connection oriented communication with guaranteed delivery. TCP internet transport protocol.

Creating a Socket

```
int socket(int domain, int type, int protocol);
```

`domain`

Specified using `AF_*` constant like `AF_UNIX`, `AF_INET`, etc.

`type`

One of `SOCK_DGRAM` or `SOCK_STREAM`.

`protocol`

Specified as 0 to allow system to automatically choose protocol suitable for first 2 arguments; can use `IPPROTO_RAW` for raw sockets.

Successful return with non-negative file descriptor; -1 on error.

Socket Endpoint Addresses

Addresses and protocol numbers must be specified in *network-order* which is big-endian. Use `htons()`, `htonl()`, `ntohs()` and `ntohl()` to convert between host and network unsigned shorts and longs.

A endpoint address is specified using a pointer to `struct sockaddr`. This is a generic structure for different protocols:

```
struct sockaddr {
  sa_family_t sa_family;          /* Address family (AF_* constant) */
  char        sa_data[14];        /* Socket address (size varies
                                     according to socket domain) */
};
```

IP Socket Addresses

TCP/IP sockets will pass a pointer to a `struct sockaddr_in` (cast to a `struct sockaddr`). Initialize to 0. Copy in components using `gethostbyname()` and `getservbyname()`.

```
struct sockaddr_in {
  sa_family_t sin_family; /* address family: AF_INET */
  u_int16_t sin_port;     /* port in network byte order */
  struct in_addr sin_addr;/* internet address */
  char sin_zero[8];       /* always zero */
};

struct in_addr {
  u_int32_t s_addr;/* IPv4 address in network byte order */
};
```

`bind()` Call

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

`sockfd`

Descriptor of previously created socket.

`my_addr`

Pointer to socket address. Usually cast from pointer to struct `sockaddr_in`. Use `INADDR_ANY` to listen to all local host interfaces.

`addrlen`

Length of 2nd argument.

`bind()` assigns a name to a socket. Used by server to setup local socket. Instead of specifying the local address explicitly, specifying `INADDR_ANY` allows the system to choose appropriate local address (as well as allows listening on any of the local addresses if the host is multi-homed).

`listen()` Call

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

`sockfd`

Descriptor of previously created socket.

`backlog`

Maximum length the queue of pending connections may grow to. May be silently rounded down to maximum implementation value `SOMAXCONN`.

Applies only to sockets of type `SOCK_STREAM`. Marks socket as *passive*. Cannot be applied to sockets returned by `accept()` or on a connected socket (on which `connect()` has been called).

`accept()` Call

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

`sockfd`

Descriptor of previously created socket.

`addr`

Pointer to socket address of peer connecting socket. Type depends on domain as for `bind()`. Can be specified as NULL if do not need peer address.

`addrlen`

Value-result parameter containing length of 2nd argument. Specify as 0 if not interested in peer address.

Blocks until a connection is received. Returns file descriptor to **new** socket which is the socket connected to the peer socket (performing the `connect()`). `sockfd` can be used to accept more connections by having `accept()` called again.

`connect()` Call

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

`sockfd`

Descriptor of previously created socket.

`addr`

Pointer to socket address. Usually cast from pointer to struct `sockaddr_in` as for `bind()`.

`addrlen`

Length of 2nd argument.

If of type `SOCK_DGRAM`, this call specifies the address to which datagrams are to be sent or received. If of type `SOCK_STREAM`, this call attempts to make a connection to another socket specified by `addr`.

Socket I/O

- Since sockets are file descriptors, can simply use `read()` and `write()` for performing I/O (specific routines for datagrams).
- Can even use C's standard I/O after opening a `FILE` stream on socket using `fdopen()`.

Echo Server Log

```
$ ./echo-server 2234 &
[1] 31085
$ ./echo-client 127.0.0.1 2234
hello world
*** hello world
hello again
*** hello again
^D
$
```

Example
Connection-Oriented Server

```
enum { MAX_BUF = 100 };
enum { QLEN = 5 };

int main(int argc, char *argv[])
{
  int port;
  int s;
  struct sockaddr_in sin;
  if (argc != 2 ||
      (port = atoi(argv[1])) < 1024) {
    fprintf(stderr, "%s PORT\n", argv[0]); exit(1);
  }
  if ((s = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
    perror("socket"); exit(1);
  }
```

Connection-Oriented Server Continued

```
memset(&sin, 0, sizeof(sin));
sin.sin_family = AF_INET;
sin.sin_addr.s_addr = INADDR_ANY;
sin.sin_port = htons((unsigned short)port);

if (bind(s, (struct sockaddr *)&sin,
         sizeof(sin)) < 0) {
  perror("bind"); exit(1);
}
if (listen(s, QLEN) < 0) {
  perror("listen"); exit(1);
}
```

Connection-Oriented Server Continued

```
  while (1) {
    struct sockaddr_in rsin;
    socklen_t rlen = sizeof(rsin);
    char buf[MAX_BUF];
    FILE *rf;
    int rs = accept(s, (struct sockaddr*)&rsin, &rlen);
    if (rs < 0) {
      perror("accept"); exit(1);
    }
    if (!(rf = fdopen(rs, "r"))) {
      perror("fdopen"); exit(1);
    }
    while ((fgets(buf, MAX_BUF, rf))) {
      write(rs, "*** ", 4);
      write(rs, buf, strlen(buf));
    }
  }

}
```

Example
Connection-Oriented Client

```
enum { MAX_BUF = 100 };

int main(int argc, char *argv[])
{
  int port;
  int s;
  struct sockaddr_in sin;
  if (argc != 3 ||
      (port = atoi(argv[2])) < 1024) {
    fprintf(stderr, "%s HOST-ADDR PORT\n", argv[0]); exit(1);
  }
  if ((s = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
    perror("socket"); exit(1);
  }
```

Example Connection-Oriented Client Continued

```
memset(&sin, 0, sizeof(sin));
sin.sin_family = AF_INET;
if (inet_pton(AF_INET, argv[1], &sin.sin_addr.s_addr) <= 0) {
  fprintf(stderr, "cannot convert %s:", argv[1]);
  exit(1);
}
sin.sin_port = htons((unsigned short)port);

if (connect(s, (const struct sockaddr*)&sin, sizeof(sin)) < 0) {
  fprintf(stderr, "cannot connect:");
  exit(1);
}
```

Example Connection-Oriented Client Continued

```
  char *line = NULL;
  int lineSize = 0;
  int nRead;
  FILE *in = fdopen(s, "r");
  if (in == NULL) {
    fprintf(stderr, "cannot open file on socket %d:", s);
    exit(1);
  }
  while ((nRead = getline(&line, &lineSize, stdin)) > 0) {
    int nWrite = 0;
    while (nWrite < nRead) {
      int n = write(s, line + nWrite, nRead - nWrite);
      if (n <= 0) break;
      nWrite += n;
    }
    if (nWrite < nRead) {
      fprintf(stderr, "could not write");
      exit(1);
    }
    if (getline(&line, &lineSize, in) > 0) {
      printf("%s", line);
    }
  }
  free(line);
}
```

Auxiliary Routines: `gethostbyname()`

```
struct hostent {
  char  *h_name;            /* official name of host */
  char **h_aliases;         /* NULL-terminated alias list */
  int    h_addrtype;        /* host address type: AF_INET or AF_INET6 */
  int    h_length;          /* length of address in bytes */
  char **h_addr_list;       /* NULL-terminated address list */
}
#define h_addr h_addr_list[0] /* for backward compatibility */

struct hostent *gethostbyname(const char *name);
```

gethostbyname() Log

```
$ ./gethostbyname www.binghamton.edu
www.binghamton.edu  128.226.136.6
$ ./gethostbyname www.google.com
www.google.com  216.58.217.164
$ ./gethostbyname www.yahoo.com
www.yahoo.com  98.139.183.24
  98.138.253.109
  98.139.180.149
$
```

`gethostbyname()` Program

In ./programs/gethostbyname.c:

```
int main(int argc, char *argv[])
{
  int i;
  for (i = 1; i < argc; i++) {
    struct hostent *p = gethostbyname(argv[i]);
    if (!p) {
      fprintf(stderr, "cannot resolve %s\n", argv[i]);
    }
    else {
      int addrLen = p->h_length;
      int a;
      printf("%s", argv[i]);
      for (a= 0; p->h_addr_list[a]; a++) {
        char *aP = p->h_addr_list[a];
        int j;
        for (j = 0; j < addrLen; j++) {
          printf("%s%d", (j > 0) ? "." : "  ",
                 (unsigned char)aP[j]);
        }
        printf("\n");
      }
    }
  }
}
```

Shutting Down a Socket

- If `close()` is called and there is still data waiting to be transmitted over the socket, then normally `close()` tries to complete the transmission.

- Note that a socket descriptor can be duplicated using `dup()` and `close()` will deallocate socket end-point only when last descriptor referring to it is closed.

- Sometimes it is necessary to indicate to one process that there is no more data coming in one direction ... we need to shutdown the socket in one direction while keeping it open in the other direction.

Socket `shutdown()` Function

```
int shutdown(int sockfd, int how);
```

`how` governs how the socket is to be shut down:

`SHUT_RD` (0)
: Stop receiving data for this socket. If further data arrives, reject it.

`SHUT_WR` (1)
: Stop trying to transmit data from this socket. Discard any data waiting to be sent. Stop looking for acknowledgement of data already sent; don't retransmit it if it is lost.

`SHUT_RDWR` (2)
: Disable both transmission and reception.

Datagrams

- Uses UDP.
- Sockets created using `SOCK_DGRAM`.
- No guarantee of datagram delivery.
- Use with higher-level protocol which attempts datagram retransmission.
- Alternatively, use with applications which can stand data loss like audio/video or real-time games.
- More efficient than connection oriented protocols like TCP for short 1-off messages.

Sending Datagrams

```
int sendto(int socket, void *buffer, size_t size, int flags,
           struct sockaddr *addr, socklen_t length);
int send(int socket, void *buffer, size_t size, int flags);
```

- Send `size` bytes of data from `buffer` via `socket` to remote socket specified by `addr/length`.

- `send()` can only be used with connected sockets.

- `flags` are bitwise-or of:

  `MSG_OOB`
  Send out-of-band data.

  `MSG_DONTROUTE`
  Bypass routing table lookup.

  `MSG_DONTWAIT`
  Non-blocking.

- Returns # of bytes sent; -1 on (local) error.

Receiving Datagrams

```
int recvfrom(int socket, void *buffer, size_t size, int flags,
             struct sockaddr *addr, socklen_t *lengthPtr);

int recv(int socket, void *buffer, size_t size, int flags);
```

- Receive upto `size` bytes of data into `buffer` via `socket`. Return remove socket address in `addr`/`*lengthPtr`.
- The `recv()` call normally used on a connected socket and is identical to `recvfrom()` call with a NULL `addr` parameter.
- If the datagram is longer than `size` bytes, then get the first `size` bytes only. There is no way to get the rest of the datagram --- it is lost. Hence using datagram protocols requires knowing datagrapm lengths.

Receiving Datagrams Continued

- `flags` are bitwise-or of:

  `MSG_OOB`
  Send out-of-band data.

  `MSG_PEEK`
  Peek at incoming message.

  `MSG_DONTWAIT`
  Non-blocking.

  `MSG_WAITALL`
  Wait until `size` bytes have been received.

- Returns # of bytes received. -1 on error.

Upcase Client-Server Log

Adapted from text:

```
$ ./upcase-server 1234 &
[3] 21983
 ./upcase-client ::1 1234 hola mundo
Server received 4 bytes from (::1, 42681)
Response 1: HOLA
Server received 5 bytes from (::1, 42681)
Response 2: MUNDO
$ kill 21983
$
```

Upcase Server

In ./programs/upcase-server.c:

```
int
main(int argc, char *argv[])
{
  int port;
  if (argc != 2 ||
      (port = atoi(argv[1])) < 1024) {
    fprintf(stderr, "%s PORT\n", argv[0]); exit(1);
  }
  int sfd = socket(AF_INET6, SOCK_DGRAM, 0);
  if (sfd == -1) {
    fprintf(stderr, "cannot create socket: %s\n", strerror(errno));
    exit(1);
  }
  struct sockaddr_in6 svaddr;
  memset(&svaddr, 0, sizeof(struct sockaddr_in6));
  svaddr.sin6_family = AF_INET6;
  svaddr.sin6_addr = in6addr_any;     /* Wildcard address */
  svaddr.sin6_port = htons(port);
  if (bind(sfd, (struct sockaddr *) &svaddr,
           sizeof(struct sockaddr_in6)) == -1) {
    fprintf(stderr, "cannot bind socket: %s\n", strerror(errno));
    exit(1);
  }
```

Upcase Server Continued

```
/* Receive messages, convert to uppercase, and return to client */
for (;;) {
  struct sockaddr_in6 claddr;
  char buf[BUF_SIZE];
  char claddrStr[INET6_ADDRSTRLEN];

  socklen_t len = sizeof(struct sockaddr_in6);
  ssize_t numBytes = recvfrom(sfd, buf, BUF_SIZE, 0,
                     (struct sockaddr *) &claddr, &len);
  if (numBytes == -1) {
    fprintf(stderr, "recvfrom error: %s\n", strerror(errno));
    exit(1);
  }
  if (inet_ntop(AF_INET6, &claddr.sin6_addr, claddrStr,
                INET6_ADDRSTRLEN) == NULL) {
    fprintf(stderr, "Couldn't convert client address to string: %s\n",
            strerror(errno));
    exit(1);
  }
  else {
    printf("Server received %ld bytes from (%s, %u)\n",
           (long) numBytes, claddrStr, ntohs(claddr.sin6_port));
  }
```

Upcase Server Continued

```
    for (int j = 0; j < numBytes; j++) {
      buf[j] = toupper((unsigned char) buf[j]);
    }
    if (sendto(sfd, buf, numBytes, 0,
               (struct sockaddr *) &claddr, len) != numBytes) {
      fprintf(stderr, "sendto for %zd bytes failed: %s\n",
              numBytes, strerror(errno));
      exit(1);
    }
  } //for (;;)
}
```

Upcase Client

In ./programs/upcase-client.c:

```
int
main(int argc, char *argv[])
{
  int port;
  if (argc < 4 || strcmp(argv[1], "--help") == 0 ||
      (port = atoi(argv[2])) < 1024) {
    fprintf(stderr, "usage: %s host-address port msg...\n", argv[0]);
    exit(1);
  }
  int sfd = socket(AF_INET6, SOCK_DGRAM, 0);  //create client socket
  if (sfd == -1) {
    fprintf(stderr, "socket creation error: %s\n", strerror(errno));
    exit(1);
  }
  struct sockaddr_in6 svaddr;
  memset(&svaddr, 0, sizeof(struct sockaddr_in6));
  svaddr.sin6_family = AF_INET6;
  svaddr.sin6_port = htons(port);
  if (inet_pton(AF_INET6, argv[1], &svaddr.sin6_addr) <= 0) {
    fprintf(stderr, "inet_pton failed for address '%s': %s\n",
            argv[1], strerror(errno));
    exit(1);
  }
```

Upcase Client Continued

```
  /* Send messages to server; echo responses on stdout */
  for (int j = 3; j < argc; j++) {
    char resp[BUF_SIZE];
    size_t msgLen = strlen(argv[j]);
    if (sendto(sfd, argv[j], msgLen, 0, (struct sockaddr *) &svaddr,
               sizeof(struct sockaddr_in6)) != msgLen) {
      fprintf(stderr, "cannot sendto %zu bytes: %s\n",
              msgLen, strerror(errno));
      exit(1);
    }
    ssize_t numBytes = \
      recvfrom(sfd, resp, BUF_SIZE, 0, NULL, NULL);
    if (numBytes == -1) {
      fprintf(stderr, "recvfrom error: %s", strerror(errno));
      exit(1);
    }
    printf("Response %d: %.*s\n", j - 2, (int) numBytes, resp);
  }
  exit(EXIT_SUCCESS);
}
```

References

Text: Chs 56, 58, 59.

APUE: Ch. 16.

Adolfo Rodriguez, John Gatrell, John Karas and Roland Peschke, *TCP/IP Tutorial and Technical Overview*, IBM Redbooks, http://www.redbooks.ibm.com/pubs/pdfs/redbooks/gg243376.pdf.

Samuel J. Leffler, Robert S. Fabry, William N. Joy and Phil Lapsley, *An Advanced 4.4BSD InterprocessC ommunication Tutorial*, http://docs.freebsd.org/44doc/psd/20.ipctut/paper.pdf.