CS442
Comer Networking API
Chapter 3

Chapter three of the textbook presents an API to perform network programming in the C language. While this chapter does not cover everything about network programming, it does demonstrate an important idea:

> A programmer can create Internet application software without understanding the underlying network technology or communication protocols.

In fact, a majority of network applications are now written at such a level that the actual networking routines are abstracted out (e.g., writing XML documents or .NET framework services).

The API in this chapter is written using Sockets. We will examine how sockets work in an upcoming lecture.

Network Communication

Recall the layering model of communications today. The applications really don't need to be aware of the low-level details of the network, only with the interface to the next level down. Applications are really just concerned with talking to other applications on the receiving side. In turn, the network neither generates nor understands the actual content of the data being sent.

Client-Server computing

How do programs find one another in a network like the Internet? First, one application must start and wait for contact. Then, the second application tries to contact it. The second application must know the location of the first application. The application that is waiting for contact is called a *server* and the program that initiates contact is called a *client*. So for example, a web server is a program waiting for a web browser (client) to contact it.

In the Internet, a location is loosely specified by a pair of identifiers:

(Computer, Application)

Computer identifies the computer, e.g., the hostname or IP address.
Application identifies the particular application on that computer, e.g. the port number

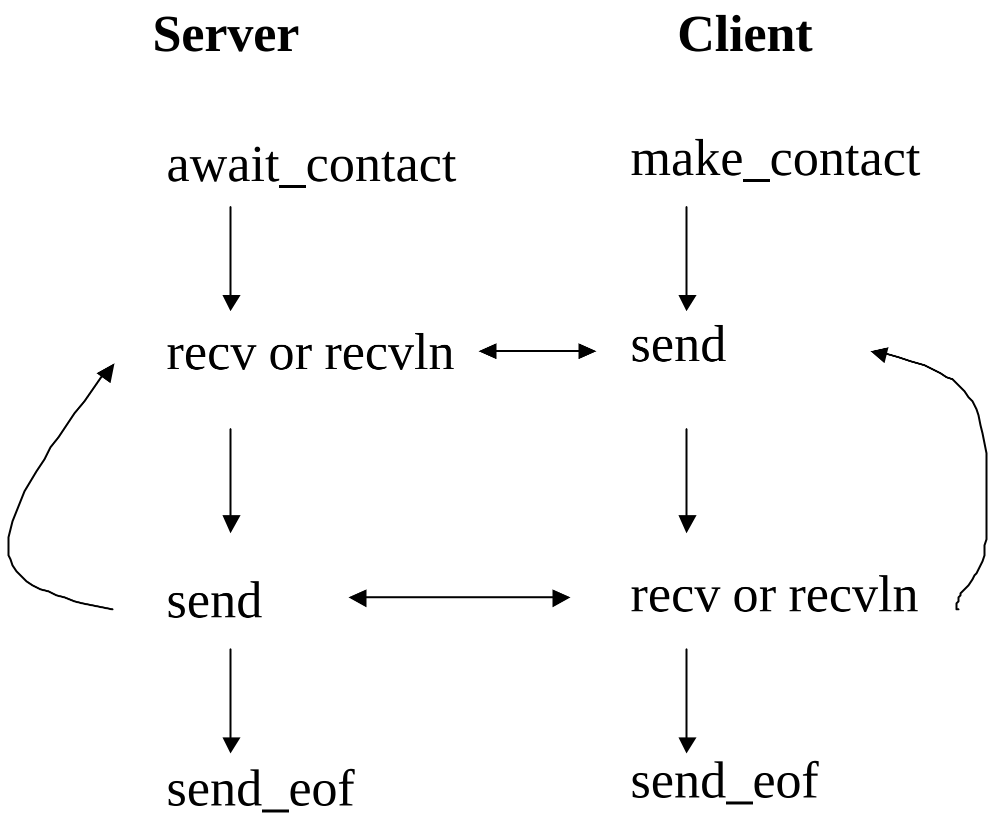
To recap, the basic steps involved are:

1. Server application starts first and waits for contact from a client
2. Client contacts the server by specifying its location and requests a communication channel be set up
3. The client and server exchange messages using some protocol
4. The client and server each send an "end of file" message to terminate communication

All of these steps are a little hairy to perform directly using sockets (we'll look at it though!)   The whole process is made a bit easier by using the API library provided by Comer.

Comer Networking Application Program Interface (CNAPI)

The API usage generally proceeds as shown below for a client and server:



Here are details on each function. To use the library, ensure that <cnaiapi.h> is included as a header file, and the linker settings can find the library object files (this is done in the Makefile in Unix, and in the Visual Studio project in Windows).

Data Types

appnum	- a number used to identify an application (port number, 2 bytes)
computer	- a number used to identify a computer (IP address, 4+ bytes)
connection	- a value used to identify the connection between a client/server (a socket, which binds the computer and appnum)

appname_to_appnum

```
appnum   appname_to_appnum( char *a)
```

This function takes an ASCII value of an application, and returns the associated number for that application (if it exists). –1 is returned in the event of failure. For example:

```
appnum x  =  appname_to_appnum("www");
```

This assigns the value 80 to x, since 80 is the port number associated with the web.

await_contact

```
connection   await_contact(appnum a)
```

This call takes one argument of type appnum and returns a value of type connection. The argument is a number that specifies the port or the application on the server. If this call fails, the connection value returned is –1. Connection will be used to send subsequent data.

Ex:

```
appnum x  =  appname_to_appnum("www");
connection c1 = await_contact(appnum);
connection c2 = await_contact(8000);          // Use port 8000 to listen on
```

cname_to_comp

```
computer cname_to_comp(char *c)
```

This call takes an ASCII argument that represents the application-layer name of the computer to contact (e.g., www.math.uaa.alaska.edu) and returns the network ID of that computer (i.e., the IP address). A negative value is returned in the event of failure.

Ex:

```
computer  comp = cname_to_comp("echidna.math.uaa.alaska.edu");
```

make_contact

```
connection make_contact(computer c,  appnum a)
```

This function is called by the client to initiate contact with a server. It requires that the computer and the application number be known. The client will use the return value, connection, to transfer data. In the event of failure, the connection value returned is negative.

Ex:

```
computer  comp = cname_to_comp("echidna.math.uaa.alaska.edu");
appnum app = 8045;
connection conn = make_contact(comp, app);
```

send

```
int send(connection con, char *buffer, int length, int flags)
```

This call sends data. It requires a connection, an address for the buffer of which data is to be sent, the length of the data in bytes, and the fourth argument is zero for normal transfer. Send returns back the number of bytes actually sent, or a negative value if there was an error.

Ex:

```
computer  comp = cname_to_comp("echidna.math.uaa.alaska.edu");
appnum app = 8045;
connection conn = make_contact(comp, app);
char buffer[100];
strcpy(buffer,"Hello there!\n");
send(conn, buffer, strlen(buffer), 0);
```

recv

```
int recv(connection con, char *buffer, int length, int flags)
```

Both clients and servers use recv to access data that arrives across the network. The first argument specifies the connection that was created. The second is the buffer address where data is to be retrieved. Length is the maximum size of data that should be placed in the buffer. The flags argument is normally 0.

Recv returns the number of bytes placed in the buffer, or zero to indicate that an *end-of-file* has been received, or a negative value in case of an error.

Ex:

```
int numbytes;
char buffer[100];
connection   conn = await_contact(8000);
numbytes = recv(conn, buffer, 100, 0);
```

This call is **blocking**; the program will wait at this instruction until data is received.

recvln

```
int recvln(connection con, char *buffer, int length)
```

recvln functions just like recv, except it will repeatedly call recv until an entire line of text has been read.

This call is **blocking**; the program will wait at this instruction until data is received.

send_eof

```
int send_eof(connection con)
```

This function signals that the sender is done transmitting data and wants to close the connection by sending an end of file character. On the receiving side, recv will return a 0 when the EOF is received. If an error occurred, this function returns a negative value.

Sample Programs

Here is code for a server that just echoes back whatever is sent to it, until an eof is received:

```
#include <stdlib.h>
#include <stdio.h>
#include <cnaiapi.h>

#define BUFFSIZE 256

int main(int argc, char *argv[])
{
      connection conn;
      int len;
      char buff[BUFFSIZE];
      if (argc != 2) {
            printf("Usage:  %s <appnum>\n", argv[0]);
            exit(1);
      }                       /* First parameter = port or appnum */
      conn = await_contact((appnum) atoi(argv[1]));
      if (conn < 0) exit(1);
      len = 1;
      while (len > 0) {
            len = recv(conn, buff, BUFFSIZE, 0);
            if (len > 0) {
               /* dangerous, buff is not necessarily a null
                  terminated string! */
               /* There may also be CR's or LF's at the end! */
               printf("%d:%s", buff);
               send(conn, buff, len, 0);
            }
      }
      send_eof(conn);   /* Signal other side we're done */
      return 0;
}
```

To do: Try this code and execute with some random port number (greater than 1024), and then telnet to that machine and port from another machine:

```
      echoserver 5031
```

From another machine:

```
      telnet <echoserver's machine> 5031
```

Try typing short text, followed by long text, and you should see that:

On the receiving side, not a null term (in fact it could be binary data)

We might have carriage returns and linefeeds for text data, warning if you do any string comparisons!

For string data, I find it useful to sometimes strip out cr's and lf's from the end and convert them to null:

```
      void stripcrlf(char *s, int len)
      {
       int i;
       for (i=0; i<len; i++) {
         if ((s[i]=='\n') || (s[i]=='\r'))
              s[i]='\0';
       }
      }
```

In while loop:

```
      if (len > 0) {
               /* strip cr's and lf's if we know it's text data */
               stripcrlf(buff, len);
               printf("%s\n", buff);
               send(conn, buff, len, 0);
      }
```

Here is code for a client that can be used with echoserver. It just connects to the destination, gets input from the user, and sends it on. This version is somewhat stripped down from the book's version (it doesn't so much error checking and other niceties).

```
#include <stdlib.h>
#include <stdio.h>
#include <cnaiapi.h>
#define BUFFSIZE 256

int main(int argc, char *argv[])
{
      computer comp;
      connection conn;
      char buff[BUFFSIZE];
      appnum app;
      int expect, received, len;

      if (argc != 3) {
            printf("Usage: %s <compname> <appnum>\n", argv[0]);
            exit(1);
      }

      /* First parameter should be host name */
      comp = cname_to_comp(argv[1]);
      /* Second parameter should be app num */
      app = (appnum) atoi(argv[2]);
      /* make contact */
      conn = make_contact(comp, app);
      if (conn < 0) {
            printf("Error!\n");
            exit(1);
      }
```

```
        printf(“Enter data> “);
        fflush(stdout);   /* show prompt without newline */

        /* Get input from user, send to server, receive
           reply from server, display to user, repeat. */

        while ((len = readln(buff, BUFFSIZE)) > 0) {
                /* Send what user typed in */
                send(conn, buff, len, 0);
                printf(“Received> “);
                fflush(stdout);
                /* Wait for reply */
                expect = len;
                received = 0;
                while (received < expect) {
                        len = recv(conn, buff, BUFFSIZE, 0);
                        if (len < 0) {          /* Server quit? */
                                send_eof(conn); exit(1);
                        }
                        /* Ensure null terminate */
                        /* DANGEROUS SHORTCUT – COULD OVERRUN BUFFER */
                        buff[len]=’\0’;
                        printf(“%s”,buff);
                        received += len;
                }
                printf(“\nEnter data> “);
                fflush(stdout);
        }
        send_eof(conn);
        return(0);
}
```

Several details make this a bit more complicated than we might like. First, the client calls the readln function to read a line of input. The fflush call ensures the text from the printf will be displayed on the screen.

The most important detail is notice the loop we made for receiving data. We do not merely issue one call to recv each time it receives data from the server. That is, a single recv call may not get all the data sent in a send function call.   The client enter a loop that repeatedly calls recv until it has received as many bytes as were sent:

> **A receiver cannot assume that data will arrive in the same size pieces as it was sent; a call to recv may return less data than was sent in call to send.**

We’ll explain why this is later (basically packets may shrink in size as they go through the network).

Let’s look at one final application, a super-simple and stripped down web server. First, it is a worthwhile exercise to run echoserver on some port, and then try to access it with a web browser. (Try using recvln instead of recv in echoserver, and print out the results with newlines and cr’s stripped):

e.g.:

```
        echoserver 8032
```

And then from a web browser, try to access http://machine:8032

You should see what the browser sends to the server upon connection, something like the following:

```
        GET / HTTP/1.1
        Accept: */*
        Accept-Language: en-us
        Accept-Encoding: gzip, deflate
        User-Agent: Mozilla/4.0 (compatible; MSIE 5.5; Windows NT 5.0)
        Host: www.math.uaa.alaska.edu:8032
        Connection: Keep-Alive
```

This is HTTP header information. It indicates the HTTP protocol, user agent, and other data, like cookies and data the browser can accept. There is a blank line at the end, signaling the end of data from the browser.

At this point the web browser is waiting for data to be returned to it. Here is a typical response from a web server:

```
        HTTP/1.1 200 OK
        Date: Fri, 07 Sep 2001 08:20:55 GMT
        Server: Apache/1.3.19 (Unix) Debian/Alpha PHP/4.0.5
        Last-Modified: Thu, 17 May 2001 01:21:39 GMT
        ETag: "3886d-1e5b-3b0327a3"
        Accept-Ranges: bytes
        Content-Length: 7771
        Connection: close
        Content-Type: text/html; charset=iso-8859-1

        <!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
        ... HTML  follows
```

The top part are more HTTP headers, specifying the size of data, type of data, etc. Let’s make a simple web server that just always spits back out the same data. We’ll fudge a bit and just use the basic HTTP headers. Our web server will just wait for a connection. When it gets one, it will read input until it reaches a blank line, discarding the data (this is the HTTP header info from the browser). Then we'll spit out a simple HTTP header, followed by HTML data:

```
#include <stdlib.h>
#include <stdio.h>
#include <cnaiapi.h>
#define BUFFSIZE 256

/* Insert stripcrlf function here, defined earlier */

int main(int argc, char *argv[])
{
        connection conn;
        int len;
        char buff[BUFFSIZE];

        if (argc != 2) {
                printf("Usage:  %s <appnum>\n", argv[0]);
                exit(1);
        }
        conn = await_contact((appnum) atoi(argv[1]));
        if (conn < 0) exit(1);

        /* Wait until we get a blank line to continue
           signals client browser waiting for data */
        strcpy(buff,".");
        while (strlen(buff)>0) {
                len = recvln(conn, buff, BUFFSIZE, 0);
                if (len > 0) {
                  stripcrlf(buff, len);
                }
        }
        /* Make a generic HTTP Header */
        strcpy(buff,"HTTP/1.1 200 OK\n");
        send(conn, buff, strlen(buff), 0);
        strcpy(buff,"Content-Type: text/html\n\n");
        send(conn, buff, strlen(buff), 0);

        /* Send fixed HTML data */
        strcpy(buff,"<h2>Some HTML Title</h2>");
        send(conn, buff, strlen(buff), 0);
        strcpy(buff,"<p><a href=http://www.yahoo.com>Yahoo!</a>");
        send(conn, buff, strlen(buff), 0);

        send_eof(conn); /* Signal other side we're done */
      return 0;
}
```

In a real web server, we wouldn't be ignoring the headers, and we'd likely be reading from the file system to retrieve the HTML page requested (or generate it by executing some CGI or script program).