ECE 252: Systems Programming and Concurrency Spring 2019

Lecture 8 — Network Communication

Jeff Zarnett *2018-07-30*

Network Communication, Continued

We now have a good idea about how to establish a connection for communication, but have thus far not actually sent or received any data! Let's assume that all the setup from before is done and we have established a connection. If we want to send datagrams, the workflow is different and we'll come back to that afterwards.

Either side can send or receive data. Figuring out whose turn it is to talk and whose turn it is to listen is the job of client and server themselves. Also what the content is, and how to interpret it is the province of the client and server.

Send. The function for sending data is [SR13]:

```
int send( int sockfd, const void* msg, int length, int flags );
```

The first parameter is the socket file descriptor and this is either the one returned by the call to `socket()` (client-side) or `accept()` (server-side). Then there is the data to send (some arbitrary bytes). The third parameter is the length of whatever we are sending. The last parameter is the flags and for a simple use case we can just give `0` to the function for this.

The return value of the function is the number of bytes sent. If something went wrong, this will be -1 and the `errno` variable will tell you more about what exactly went wrong. Under ideal circumstances, the number of bytes sent equals the length parameter.

Let's consider a modified example from [Hal15]:

```
char *msg = "Hello world!"
int len = strnlen( msg, 13 );
int sent = send( sockfd, msg, len, 0 );
```

And there you are, the data is sent! Assuming it went well. In real life it might be best to check for -1; unlike something like `malloc()` where the failure case is rare (is the memory allocator really going to be unable to give you an `int` often?), networks are tricky and can fail. Checking is worthwhile...

But wait a minute, we said under ideal circumstances the number of bytes is equal to the length. As you might have guessed, there is a limit to the amount of data that you can send in one chunk. The actual amount you can send in one chunk is reasonably-sized (somewhere around 1KB) but you can't just memorize a number and assume that will always be true across all systems. So if you have a significant chunk of data to send, you'll need to check how much was sent and then you are responsible for sending out the rest.

If the data being sent is large enough that this is a concern, you should track the number of bytes sent and keep calling send, updating the pointer as you advance. The code below is modified from [Hal15] that sends as many times as necessary until we're down to 0 bytes left to send:

```
int sendall( int socket, char *buf, int *len ) {
  int total = 0;        // how many bytes we've sent
  int bytesleft = *len; // how many we have left to send
  int n;

  while( total < *len ) {
    n = send( socket, buf + total, bytesleft, 0 );
    if (n == -1) {
      break;
```

```
    }
    total += n;
    bytesleft -= n;
  }
  *len = total; // return number actually sent here
  return n == -1 ? -1 : 0; // return -1 on failure, 0 on success
}
```

Receive. And if you'd like to receive data, the call for this is `recv()`; its signature is almost the same but not identical [SR13]:

```
int recv( int sockfd, void * buffer, int length, int flags );
```

The socket file descriptor is the place to receive data from. The buffer parameter is the destination where the data goes, and the length is the maximum size of that buffer. Flags can also be 0 here.

The return value is the number of bytes actually read into the buffer. If you got back -1, then an error occurred and check `errno` for more details. If you got back 0, it means the other side hung up on you: they closed the socket. So no point in waiting anymore!

Knowing that the other side is finished sending data is not necessarily all that easy. That's something to work out with the sender. This is why in movies when characters communicate with walkie-talkies, they say "over" when they have finished their transmission... so the recipient will know they are finished talking. In network communication, we might know in advance the size of data we are supposed to get, or we might be told as part of the (negotiated) protocol, or we might need to wait for the other side to close the connection (then we know they are definitely done).

Suppose we are sending more than just a string. Can we do a fancy thing and write directly to a `struct` by making the buffer location the location of thtat `struct` and the length the `sizeof` that type? Yes, you can but this requires that the representation you receive over the network to be exactly the same as your struct. A more sensible approach is to serialize your data in some way, and then de-serialize it on the other side.

Serialization is the process of converting the data to some sort of byte-representation so that it can be sent across the network (or stored in a database, or anything really) and then later reconstructed via the deserialization process. This means that no particular data format is needed and systems that don't use the same software or architecture, even, can communicate easily.

In a practical scenario there's no need to write your own (de)serialization routine; there exist libraries like `protobuf-c` that are designed explicitly for this purpose. Pick a good one and use it.

But anyway, this is it! That's how we send and receive data. When we're done, we just call `close()` on the socket and that is the end. We now know how to communicate over the network.

Datagrams

But if you don't want to establish a communication channel and you just want to send a message (that is, calling is for old people and we just want to text people) there are two related functions for that instead [Hal15]:

```
int sendto( int sockfd, const void* msg, int length, unsigned int flags,
    const struct sockaddr* to, socklen_t tolength )

int recvfrom( int sockfd, void* buffer, int length, unsigned int flags,
    struct sockaddr* from, int* fromlength )
```

Given what we've covered here nothing is really surprising. Because there's no connection established, each send has parameters for where to send the data to and each receive tells you where the data is being received from. Send still returns the number of bytes sent and may be less than you were expecting if sending a large amount of data.

If you call `connect()` on a datagram socket, incidentally, you can then skip some of this and just use the regular `send` and `recv` operations – the transport is still UDP, but the source and destination don't need to be added every time.

In our next lecture, we will do an in-class exercise related to using sockets. The in-class exercise will provide an opportunity to use the socket code we've learned in a way that is more engaging than just seeing yet another code example in the lecture notes.

cURL

In most situations, however, we don't work with sockets directly when dealing with URLs. Instead we are likely to use cURL (or similar), a network communication and transfer request library. There is an associated command-line tool, but the client-side library is the area of interest because we can build on it in our program and save ourselves a lot of hassle. The curl library can do a lot of things and communicate via a lot of different protocols. It is quite flexible in that way. But it is only for the client-side and isn't meant to be used for server-side operations.

Imagine, if you will, that you want to access a webservice. Lots of things run via web service now and you are probably already somewhat familiar with them. In short, servers have "endpoints" that clients connect to via HTTP, and then the client can get a response. There are numerous examples of services that use this mechanism and they often adhere to some design principles like REST (REpresentational State Transfer). If we wished to communicate, for example, a GET request (to "get" some resource), then we can put together a connection via a socket and write the ""GET / HTTP/1.0\r\n"" into a string and send that message via `send()`. But there's no need to do it all by hand because we can do this very easily with libcurl.

As you might imagine, the information in this section is based on the official libcurl documentation. We don't cover every option or every detail, so it might be worth it to read some of the official documentation if you get stuck: `https://curl.haxx.se/libcurl/c/`.

This example, modified from `https://curl.haxx.se/libcurl/c/https.html` outlines all the key parts we need to use the curl library. We'll go through it and see what each of the parts do:

```
#include <stdio.h>
#include <curl/curl.h>

int main( int argc, char** argv ) {
  CURL *curl;
  CURLcode res;

  curl_global_init(CURL_GLOBAL_DEFAULT);

  curl = curl_easy_init();
  if( curl ) {
    curl_easy_setopt(curl, CURLOPT_URL, "https://example.com/" );
    res = curl_easy_perform(curl);

  if( res != CURLE_OK) {
      fprintf(stderr, "curl_easy_perform() failed: %s\n", curl_easy_strerror(res));
    }
    curl_easy_cleanup(curl);
  }

  curl_global_cleanup();
  return 0;
}
```

Starting from the top then: we need to include the header `curl/curl.h`. If you are trying to compile and run a program on your own system and this header is not found, you need to install `libcurl-dev` on the system.

At the beginning of `main` there are two variables declared. The `CURL` structure is our "handle". This needs to be, like many structures, initialized before use and cleaned up when we're done with it. The `CURLcode` type is also a useful structure for finding out how operations went. Unlike in many other scenarios where success is assumed, remember, in the network we should always check and see what happened.

All the code that uses the network is bracketed in the `curl_global_init()` and `curl_global_cleanup()`. For the initialization, we take the defaults with `CURL_GLOBAL_DEFAULT` which is sufficient. If we forget to call the global initialization function then the later call to `curl_easy_init` will do it for us, but this is dangerous because having it called more than once is an issue. And of course we should not forget to clean up when done.

Each connection is accessed through the handle, and so the handle is initialized with `curl_easy_init()`. If something goes wrong, the handle is set to NULL (hence why it can be used in the if-statement). If everything went well then we can actually get to work. If you are wondering why there is "easy" in the function names, it's not because there is a "hard" interface, but there is a "multi" interface that we will learn about in the future. (Whether you think the multi interface is hard is a separate discussion).

The next thing to do is set options for the connection. The only mandatory option its the URL to use (otherwise we don't know where to connect to, after all). In this case it's a dummy URL but it will do for the purposes of the code example. In the future we will need to set other options to do anything useful with a response, but for now the simple example leaves that out. There are something like 200 options, according to the documentation, so you can configure just about anything you want.

With all setup done, then we can perform the request with `curl_easy_perform`. This is the step that actually sends data over the network and retrieves a response. The result is stored in `res`.

Then we need to check and see if everything worked: `CURLE_OK` is the response code if all went well. Otherwise we get back something else and the function `curl_easy_strerror` interprets an error for us making it a little easier to understand what happened... isn't this a lot better than having to go google what the given value of `errno` is?

A handle can be used multiple times if you need, although you may need to update the options that are set on it to reflect the new things you'd like to happen

If we wanted to re-use a handle but clear all the settings there is `curl_easy_reset`, but that's not the case in this scenario since the example is a one-time thing. Instead, we clean up. When we're done with a given handle we clean it up with `curl_easy_cleanup`. And then of course when all is finished there's the global cleanup. Excellent!

Setting up Callbacks. Almost certainly, however, we want to do something useful with the data we got. Or, we might have some data that we need to send. For each direction, what we want to set up is a *callback*.

In case you haven't been introduced to the concept, it's a lot like what happens when you tell someone "when you're finished, call me back". They were in the middle of something when you called, and when a certain condition is fulfilled, they call you. In the context of using the cURL library, we tell the curl handle what function we would like it to call when the appropriate time comes.

The read callback is used when you are uploading data to the server (sometimes this is a POST operation). The write callback is used when you are receiving data from the server (this can be a GET operation). You may set up a read or write callback (or both) for an operation. There can be different callbacks for different easy handles, of course.

A write function has to have the following signature (according to the official documentation):

```
size_t write_callback( char *ptr, size_t size, size_t nmemb, void *userdata );
```

The name of the function can be anything you like. The `size_t` type represents a size and can be treated like an integer. The `ptr` points to whatever data we received and `nmemb` is the size of that data. `size` is always 1 (the documentation does not explain why this is the case). And the last parameter is user data: we can pass some data to the processing function directly. This is helpful when we have some data that we would need to provide to make processing easier or possible. If, for example, we want to write the requested data to a file with a specified name, we can pass that name in the `userdata` pointer, and then our implementation of the write callback should know how to use it. Finally, the return type is also a `size_t`; the spec requires that the returned size is the number of bytes of the data successfully processed. If it's not equal to the size of `nmemb` then the library interprets that as an error in writing.

A read function is very similar:

```
size_t read_callback( char *buffer, size_t size, size_t nitems, void *inputdata );
```

Again, the function can be called whatever you like. The `buffer` is the area where you are going to put the data to send (and it can't be any bigger than the next parameters specify). `size` here is the size of each data element and `nitems` is the number of items. In practice you will just want to calculate the maximum buffer size by multiplying these two things together. And he return value is the number of bytes successfully put there. We might need to do this in several chunks, though. If you put 0, it signals end-of-file and means no further upload will take place.

And to register the read and write callback respectively, there are two steps. One to register the function, and another to set the data:

```
CURLcode curl_easy_setopt( CURL *handle, CURLOPT_READFUNCTION, read_callback );
CURLcode curl_easy_setopt( CURL *handle, CURLOPT_READDATA, void *pointer );

CURLcode curl_easy_setopt( CURL *handle, CURLOPT_WRITEFUNCTION, write_callback );
CURLcode curl_easy_setopt( CURL *handle, CURLOPT_WRITEDATA, void *pointer );
```

Alright, on to an example, then, modified from `https://curl.haxx.se/libcurl/c/post-callback.html`.

```
#include <stdio.h>
#include <string.h>
#include <curl/curl.h>

const char data[]="Lorem ipsum dolor sit amet, consectetur adipiscing "
  "elit. Sed vel urna neque. Ut quis leo metus. Quisque eleifend, ex at "
  "laoreet rhoncus, odio ipsum semper metus, at tempus ante urna in mauris. "
  "Suspendisse ornare tempor venenatis. Ut dui neque, pellentesque a varius "
  "eget, mattis vitae ligula. Fusce ut pharetra est. Ut ullamcorper mi ac "
  "sollicitudin semper. Praesent sit amet tellus varius, posuere nulla non, "
  "rhoncus ipsum.";

struct data {
  char *readptr;
  size_t sizeleft;
};

size_t read_callback( void *dest, size_t size, size_t nmemb, void *userp ) {
  struct data *d = (struct data *) userp;
  size_t buffer_size = size * nmemb;

  if( d->sizeleft > 0 ) {
    /* copy as much as possible from the source to the destination */
    size_t copy_this_much = d->sizeleft;
    if ( copy_this_much > buffer_size ) {
       copy_this_much = buffer_size;
    }
    memcpy(dest, d->readptr, copy_this_much);

    d->readptr += copy_this_much;
    d->sizeleft -= copy_this_much;
    return copy_this_much;
  }
  return 0; /* no more data left to deliver */
}

int main( int argc, char** argv ) {
  CURL *curl;
  CURLcode res;
  struct data * d = malloc( sizeof( struct data ) );

  d->readptr = data;
  d->sizeleft = strlen( data );

  res = curl_global_init( CURL_GLOBAL_DEFAULT );
  if ( res != CURLE_OK ) {
    fprintf( stderr, "curl_global_init() failed: %s\n", curl_easy_strerror( res ) );
    return 1;
```

```
  }

  curl = curl_easy_init();
  if ( curl ) {
    curl_easy_setopt( curl, CURLOPT_URL, "https://example.com/index.cgi" );

    /* Now specify we want to POST data */
    curl_easy_setopt( curl, CURLOPT_POST, 1L );

    curl_easy_setopt( curl, CURLOPT_READFUNCTION, read_callback );
    curl_easy_setopt (curl, CURLOPT_READDATA, d );

    res = curl_easy_perform(curl);
    /* Check for errors */
    if(res != CURLE_OK) {
      fprintf( stderr, "curl_easy_perform() failed: %s\n", curl_easy_strerror( res ) );
    }
    curl_easy_cleanup( curl );
  }
  free( d );
  curl_global_cleanup();
  return 0;
}
```

A brief explanation is in order. Up at the top there is `data`, some amount of sample data and a little structure that is used to reference the data and to keep track of how much data there is to send. This is going to be important in the read callback function. And then there is of course the read callback function itself.

That function conforms, as you would expect, to the function signature as expected. If we are fortunate then it is possible to send the data in one chunk. But in this example we have intentionally made the data large. So we will now figure out how much we can actually fit in there by figuring out the size of the buffer and seeing if the data to be sent is bigger than the available space. Once we've figured out the limit, we can copy the data with `memcpy` to the buffer. Then we update where we are reading from and how much space is left. Then we return the amount copied. When all data has been copied, we return 0.

A large amount of the code of `main` is the same as in the previous example. Some new things of note, are obviously the initialization of the structure `d`. We set the option for the operation to be POST (upload data) since the default is GET (fetch some data), and then we have to register the read callback as well as send the data to be used when the read function runs.

References

[Hal15] Brian "Beej Jorgensen" Hall. Beej's guide to network programming: Using internet sockets, version 3.0.21, 2015. Online' accessed 5-July-2018. URL: `https://beej.us/guide/bgnet/html/single/bgnet.html`.

[SR13] W. Richard Stevens and Stephen A. Rago. *Advanced Programming in the UNIX Environment, Third Edition*. Addison-Wesley, 2013.