Lecture 8 — Network Communication

Prepared by Jeff Zarnett, taught by Seyed Majid Zahedi
`jzarnett@uwaterloo.ca`, `smzahedi@uwaterloo.ca`

Department of Electrical and Computer Engineering
University of Waterloo

On the Line

We established a connection, but so far have neither sent nor received any data.

Either side can send or receive.
- Figuring out whose turn it is will be the job of client and server.

Same for what the content is.

Sending Data

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

`sockfd`: Socket to send the data to.

`msg`: Bytes of data to be sent.

`length`: Size of the message.

`flags`: Options; giving in 0 will suffice.

Return value: number of bytes sent.

Send Fail?

If something went wrong, the function returns -1..

The `errno` variable will tell you more about what exactly went wrong.

Under ideal circumstances, the number of bytes sent equals the length parameter.

Otherwise – not all data was sent!

Send Message

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

In real life it might be best to check for -1.

Networks are tricky and can fail. Checking is worthwhile...

If I Can Just Repack It..

Under ideal circumstances the number of bytes is equal to the length.

There is a limit to the amount of data that you can send in one chunk.

How Much Space Do We Have?

The actual amount you can send in one chunk is reasonably-sized ($\approx$1KB).

You can’t memorize a number & assume that will 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.

Blowing Up Our Phones...

Track the number of bytes sent and keep calling send, updating the pointer as you advance:

```c
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
}
```

Send is called as many times as necessary.

Receiving You Loud and Clear...

And if you'd like to receive data, the call for this is `recv()`:

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

`sockfd`: Where to receive data from.

`buffer`: Where the data goes

`length`: the maximum size of that buffer.

`flags`: flags can also be 0 here.

Return value: the number of bytes actually read into the buffer.

Negative Numbers are Bad

Return values with special meaning:

-1: then an error occurred and check `errno` for more details.

0: the other side hung up on you: they closed the socket.

Are you *still* talking?!

Knowing that the other side is finished sending data may not be easy!

Options:

- Pre-defined length of message
- Negotiated at the start of communication
- Wait for them to hang up

String Theory?

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 that `struct` and the length the `sizeof` that type?

Struct Theory

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.

Serial Killers

Serialize

Serialization is the process of converting the data to some sort of byte-representation.

Then later it can be 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.

Do Not Reinvent the Wheel

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.

No, You Hang Up First!

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

Calling is for old people and we just want to text people!

```
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 )
```

Each send has parameters for where to send the data to and each receive tells you where the data is being received from.

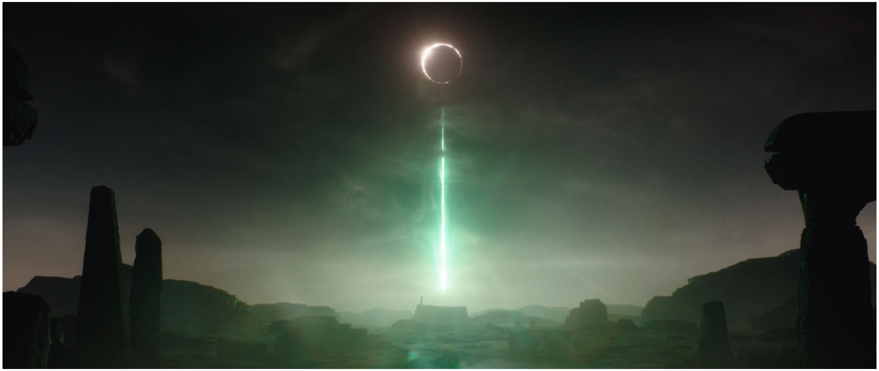
Top Connections

If you call `connect()` on a datagram socket, you can then skip some of this.

Then you can 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.

That’s No Moon!



So cURL me maybe?

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.

It is only for the client-side and isn’t meant to be used for server-side operations.

Webservices!

Imagine that you want to access a webservice.

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).

`GET course/ece252/grade`

If we wished to communicate, for example, a GET request, then we can put together a connection via a socket.

Write the “"GET / HTTP/1.0\r\n"” into a string and send that message via `send()`.

But: no need to do it by hand because we can do this very easily with libcurl.

Starting Example of cURL

```c
#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;
}
```

Recycle, Reduce, Reuse

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`.

Setting up Callbacks

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.

Read Callback, Write Callback

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.

Write Callback Function

A write function has to have the following signature

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

The name can be anything you like.

`size_t`: represents a size and can be treated like an integer.

`ptr`: points to whatever data we received.

`nmemb`: the size of that data.

`size`: always 1.

`user data`: arbitrary structure we get to pass directly to this function.

Return value: number of bytes processed.

Did We Get ’Em All?

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.

Read Callback Function

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

`buffer`: the area where you are going to put the data to send.

`size`: the size of each data element.

`nitems`: the number of items.

In practice you will just want to calculate the maximum buffer size by multiplying these two things together.

Return value: the number of bytes successfully put there; 0 signals end-of-file.

Registration

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 );
```

Callback Example

```c
#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;
};
```

Callback Example

```c
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 */
}
```

Callback Example

```
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;
  }
```

Callback Example

```
  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;
}
```

Not Just Coincidence

In these examples and our brief overview, we have really only scratched the surface of what the curl library can do.

But this is enough to get the flavour of how it works and to begin to do useful work with it.

Such as, perhaps, a lab?