Internet protocols: structure and formats of messages

The internet protocols of interest in this course are HTTP, RTSP, SIP. We are also interested in SDP (the Session Description Protocol).

Internet protocols are mainly standardized by the IETF (Internet Engineering Task Force); they are normally client-server protocols, where a client sends a request to a server, and the server replies with a response.

Therefore, there are two main kinds of messages: request messages and response messages.

To describe the structure of messages, we will refer to the HTTP protocol; however, the same structure applies to the other protocols we are interested to.

HTTP request messages
A typical HTTP request message has the following structure:

First line\r\n
Header line\r\n
…
Header Line\r\n
\r\n
Body

The first line and the header lines are text strings, and they Are separated by the line separator \r\n (carriage return and line feed), that is, 0x0D0x0A.
At the end of the last header line there is an additional line separator, creating a sequence of two consecutive line separators, that marks the start of the body of the message.

The body is a sequence of bytes; there is no specific

the body can carry any kind of content, for example, text, a picture, a video clip, json-formatted data, …

The first line of a request message has the following structure:

method request-uri protocol-version

That is, the first line has three fields, separated by a single space character. The first field is the method, that specifies the kind of action that the requesting entity would like to perform. The second field specifies which is the resource on which the action must be performed, and the third field wield specifies the protocol identity and version.

For example:

GET /various_stuff/index.html HTTP/1.1

specifies that the requesting party is asking (GET method) for the web page various_stuff/index.html

The identity of the host to which the request is addressed is specified both implicitly and explicitly. The host is specified implicitly because before transmitting the request message, the requesting party must connect to the machine hosting the requested web page. The host is also specified explicitly through the Host header:

GET /various_stuff/index.html HTTP/1.1
Host: destination_machine.somewhere.com

The second field of the first line of a request message is defined as the request-uri, where uri is the Universal Resource Identifier. The complete URI for the above request could be, for example:

This would be the string that you need to write in the navigation bar of your browser if you want to display the specified html page.

The uri starts with http://, this means that you want to use the plain HTTP protocol. If the destination server implements HTTPS (HTTP with security features such as criptography), you would need to use http:// instead.
The format of the message would be the same, but the communication between the client and server would be different, because it would be encrypted and security-related handshakes would be performed.

Header lines provide additional information, such as the host Identity, as we have already seen). The structure of a header line is:

header_name: header value

The names of the headers are standardized. The headers are divided into four categories: general headers, request headers, response headers, and entity headers.

General headers in general provide details on the connection between client and server. Request header give additional details about a request. Response headers give additional details about a response, and entity headers provide information about the content carried in the body of the message.

In a message, headers should appear in the following order:
1. general headers
2. request headers
3. response headers
4. entity headers.

One of the most common request headers is the Host header, described above. Another important general header is the

1) Connection: close
2) Connection: keep-alive

If the Connection header has value “close”, the server shuts down the connection immediately after the response has been sent. If the Connection header is “keep-alive”, the server will keep alive for some time the connection after the response has been sent. This allows the client to send multiple requests over the same connection. This kind of behavior is called “persistent connection” and it is more efficient than the non-persistent case, because we don’t have to wait the connection setup time for each request. However, keeping connections up costs the server resources (file descriptors), thus, the server usually keeps alive the connection for a relatively short time, that it shuts it down if no further requests are received.

The Accept header (it is a request header), specifies which kind of content the client expects to find in the body of the response message. Typical values of this header are:

Accept: text/html
Accept: application/json
Accept: application/sdp

In the basic request case, the request message has no body. In some cases a request message has a body (for example when the request method is POST), but in our specific Context we are not interested in this kind of requests.

We have mentioned that there is a number of request methods, but we have made an example for the GET method only. The other request methods are CONNECT, DELETE, HEAD, OPTIONS, POST, PUT, TRACE. The DELETE and PUT methods are particularily interesting, as they are used to Delete content from the server and upload content to the server,

Response messages are used by the server to send the client a response to a request.

A response message has the following structure:

First line\r\n
Header line\r\n
…
Header Line\r\n
\r\n
Body

Thus, it has the same structure of a request message. However, The first line is different and a response message may use headers that are not used by request messages.

The first line of a response message is:

protocol_version reason_code reason_phrase

For example:

HTTP/1.1 200 OK

Each kind of response has a numerical code (the reason_code), and a text name (the reason phrase). Let us see a minimal Example of request/response, keeping on with the previous example of a web page request.

Request
GET /various_stuff/index.html HTTP/1.1
Connection: keep-alive
Host: destination_machine.somewhere.com
Accept: text/html
Content-length: 0

Response
HTTP/1.1 200 OK
Connection: keep-alive
Content-type: text/html
Content-length: 24

<html>Hello world</html>

In the request message, we can notice the Accept header. This header specifies that the client expacts an html page as a response. The entity header Content-length specifies the Length of the content in the body of the message. In this request, the message has no body so the content length is zero.

The respose start with 200 OK, so it is a success. It confirms that the content type is text/html, and the Content-length header Tells us that the content is 24 bytes long. After the Content-length header, we have an additional line separator, and then the body follows.

It is important to note that messages have no start and end delimiters: the only way in which a receiver can understand where a message starts and ends is by counting the bytes of the Content. A wrong value in the Content-length header causes the connection to shut down.

Response codes (the reason codes) are organized by hundreds: 1xx, 2xx, 3xx, 4xx, 5xx.
The response codes in the 1xx hundred are provisional, that is

the server may send, before the final answer is given.

In the 2xx hundred, the most relevant reason code is 200, that means success. In the 3xx hundred we have redirections. For example, if a resource is on another server and the server we have contacted knows where it is, it may send us a 307 temporary redirect response, specifying the location of the resource in the location header. For example:

Request
GET /various_stuff/index.html HTTP/1.1
Connection: keep-alive
Host: destination_machine.somewhere.com
Accept: text/html
Content-length: 0

Response
HTTP/1.1 307 Temporary-redirect
Connection: keep-alive
Location: destination_machine.somewhereelse.com/various_stuff/index.html
Content-length: 0

In the 4xx responses we find errors, due to bad formatting of the request, while the 5xx response signal the presence of a server error.

How networked applications connect.
Now, we will study how an application (say, it client side) Connects to a remote application (say, the server side of the Application).

We need to remember that networked applications work on the TCP/IP or UDP/IP protocol stack, and they can connect and communicate by connecting two sockets, one on the client side and one on the server side of the application. A socket is, logically, a pair (ip address, port).

The scheme of this operation is that the server opens a socket on its side, on a port that clients know, and it LISTENS for Incoming connections. When a connection for that port reaches the server, the sever MAY accept it and create the connection.

Here we present a very simplified outline of what happens in a server application that wants to receive connections from remote clients.

Server side

```
…
port=30000; // this service receives connections on port 30000
strcpy (ipaddr, “10.20.30.40” ); // the ip address of this server, in text format
int fd; // file descriptors are referred to with an integer index
fd = socket ( AF_INET , SOCK_STREAM , 0 ); // create a TCP socket
struct sockaddr_in name;// this is how internet addresses are represented in C
name.sin_family = AF_INET; // this is an IP address
name.sin_port = htons ( (uint16_t) (port) ); // IP addresses and transport layer
                                             // ports are glued toghether
                                             // the port number must be
                                             // represented in the format used for
                                             // transfer in the network
name.sin_addr.s_addr = inet_addr ( ipaddr );// convert the string representing the
                                            // server’s ip address in network
                                            //format
bind (fd, ( struct sockaddr * )&name , sizeof(name)); // associate the logical
                                            // socket’s information (address and
                                            // port), to the physical resource
listen (fd, 100); // activate the listening operation on the socket, so that
                  // incoming connections are detected

// infinite loop
While (1)
{

    // the accept function monitors the socket continually, and it returns
    // only if a connection request is received, or on error
    // if it returns with a valid file descriptor index (>=0) it is a
    // success, and fd1 is a socket ready to be used for communication
    // with the client that placed the connection request
    struct sockaddr_in c_name; // prepare to register IP address and source port
                               // of the client that issues the connection request
    socklen_t client_name_size = sizeof (client_name);
    int fd1 = accept(fd_server, (struct sockaddr *) &c_name , &c_name_size);

    // comment of the accept function as used here: the accept function is blocking,
    // so the server program stays on the accept until a connection request
    // arrives, and it can’t do nothing else meanwhile. This is why this is a toy
    // example: things are done in a more complex way, in real life

    // if the program gets to this point, a connection request has been received
    // and, if fd1>=0, it has been accepted and fd1 is ready for communications
    // with the client’s socket

    // what happens now depends on the behavior of your server program: is it a
    // multi-process server or a multi-thread server? However, we do not proceed
    // because things get thicker from now on ...
}
```

XXX

Client side

```
…
port=30000; // this service receives connections on port 30000
strcpy (ipaddr, “10.20.30.40” ); // the ip address of this server, in text format
int fd = -1;
struct sockaddr_in addr;
addr.sin_family = AF_INET;
fd = socket(AF_INET,SOCK_STREAM,0);
addr.sin_port = htons (port);
addr.sin_addr.s_addr = inet_addr (ipaddr);
int res_conn = connect ( fd, (struct sockaddr *) (& addr), sizeof (addr) );
if ( res_conn < 0 )
{
    // failure ...
}
// a comment on connect(): it is blocking, thus if the server does no respond
// the client gets stuck forever. This is why in real life things are made
// more complex than explained here ...
...
```