Application protocols

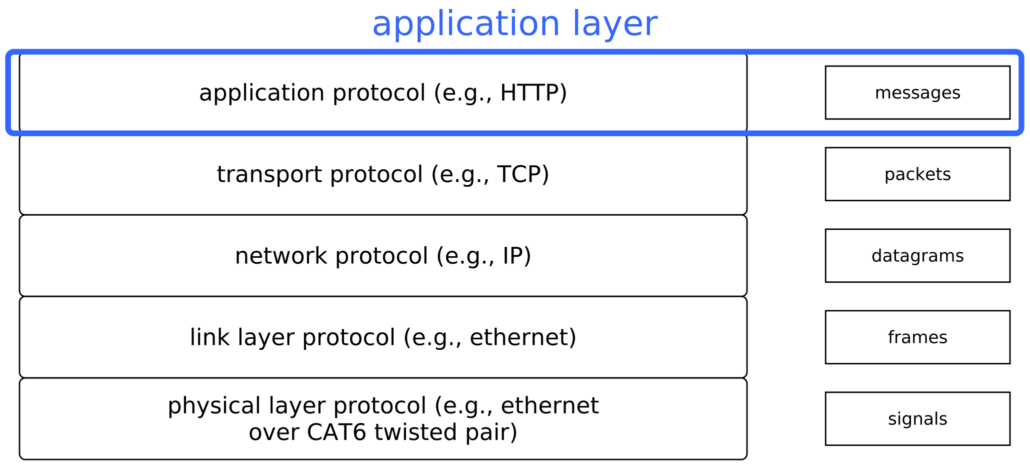
David Hovemeyer

18 November 2019

Application layer

In the network protocol stack, the *application layer* is at the top

- Consists of applications: web browsers/servers, email clients/servers, P2P file sharing apps, etc.


application layer
application protocol (e.g., HTTP)
messages
transport protocol (e.g., TCP)
packets
network protocol (e.g., IP)
datagrams
link layer protocol (e.g., ethernet)
frames
physical layer protocol (e.g., ethernet over CAT6 twisted pair)
signals

Application protocols: define how peer applications communicate with each other

Example: HTTP

HTTP

HTTP history

Invented by Tim Berners-Lee at CERN in 1989

- Initial goal: online sharing of scientific data

Application protocol underlying the *World Wide Web*

Most important content type is HTML: *HyperText Markup Language*

- ...but flexible enough for access to any kind of data

HTTP

A synchronous client/server protocol used by web browsers, web servers, web clients, and web services

- HTTP 1.1: https://tools.ietf.org/html/rfc2616

Client sends request to server, server sends back a response

- Each client request specifies a *verb* (GET, POST, PUT, etc.) and the name of a *resource*

Requests and responses may have a *body* containing data

- The body's *content type* specifies what kind of data the body contains

HTTP message format

All HTTP messages have the same general form:

- First line: describes meaning of message
- Zero or more *headers*: metadata about message
- Optional *body*: payload of actual application data (HTML document, image, etc.)

Protocol is text-based, with lines used to delimit important structures

- Each line terminated by CR (ASCII 13) followed by LF (ASCII 10)
- Line continuation using backslash (\) allowed for headers

HTTP headers

An HTTP *header* has the form

```
Name: Content
```

Each header provides metadata to help the recipient understand the meaning of the message

HTTP has evolved significantly over time: headers help the communicating peers understand each other's capabilities

Examples:

- $\boxed{\texttt{Host: placekitten.com}}$ specify which host server is accessed
- $\boxed{\texttt{Content-Type: text/html}}$ specify that body is an HTML document

HTTP request

An HTTP *request* is a message from a client to a server

Specifies a *method* and a *resource*

- Method: the verb specifying what action the client is requesting the server perform (GET, PUT, POST, etc.)
- Resource: the data resource on the server to which the client is requesting access

For HTTP 1.1, first line also specifies protocol version

A request can have a body (payload): examples include

- Submitted form data
- File upload data

HTTP request example

Example HTTP request:

```
GET /1024/768 HTTP/1.1
Host: placekitten.com
User-Agent: curl/7.58.0
Accept: */*
```

- $\boxed{\texttt{GET}}$ is the method (request to get resource data)
- $\boxed{\texttt{/1024/768}}$ is the resource
- The $\boxed{\texttt{Host}}$ header specifies which website is being accessed (a web server can host multiple sites)
- The $\boxed{\texttt{Accept}}$ header indicates what file types the client is prepared to receive

HTTP response

An HTTP response indicates protocol version, *status code*, and *reason phrase*

The status code specifies how the client should interpret the response: e.g. 200 (OK), 403 (Forbidden), 404 (Not Found)

- Full list: https://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html

The reason phrase is informational and does not affect the meaning of the response

HTTP response example

Example HTTP response:

```
HTTP/1.1 200 OK
Date: Wed, 13 Nov 2019 12:33:20 GMT
Content-Type: image/jpeg
Transfer-Encoding: chunked
Connection: keep-alive
Set-Cookie: __cfduid=de2a22cdd3ed939398e0a56f41ce0e4a31573648400; expi
Access-Control-Allow-Origin: *
Cache-Control: public, max-age=86400
Expires: Thu, 31 Dec 2020 20:00:00 GMT
CF-Cache-Status: HIT
Age: 51062
Server: cloudflare
CF-RAY: 5350c608682a957e-IAD
```

Headers followed by blank line and 40,473 bytes of data

HTTP response example

Features of example HTTP response:

- Response code was $\boxed{\texttt{200}}$, indicating success
- The $\boxed{\texttt{Content-Type}}$ header indicates resource is an image
- The $\boxed{\texttt{Transfer-Encoding}}$ header indicates that the body is encoded using ‘‘chunked’’ encoding (commonly used for streaming content, but also used for static content)
- The $\boxed{\texttt{Connection: keep-alive}}$ header invites the client to keep the connection open, to be reused for subsequent requests

Message bodies

An HTTP request or response can have a body containing arbitrary data

Various encodings are possible: raw binary, chunked (chunks consist of byte count followed by specified amount of data)

Compression can be used

Content types

The $\boxed{\texttt{Content-Type}}$ header indicates what kind of data the message body contains

The content of the header is a *MIME type*, e.g.

- $\boxed{\texttt{text/html}}$ HTML document
- $\boxed{\texttt{text/html; charset=utf-8}}$ HTML document with UTF-8 character set
- $\boxed{\texttt{image/jpeg}}$ JPEG image

Official registry of MIME types:
https://www.iana.org/assignments/media-types/media-types.xhtml

Experimenting with HTTP

One of the best ways to learn about HTTP is to examine actual HTTP message exchanges

The curl program is a command-line HTTP client: use the -v option to have it print the first line and headers of the HTTP request and HTTP response

Example:

```
    curl -v http://placekitten.com/1920/1080 -o kitten.jpg
```

HTTP server implementation

Implementing an HTTP server

HTTP server: listens for incoming TCP connections, reads client requests, sends back responses

Example implementation on web page: webserver.zip

Section 11.6 in textbook also presents an example web server

Lecture will highlight interesting implementation issues, see code for gory details

Code uses csapp.h/csapp.c functions, see textbook for details about these

The server loop

```
int main(int argc, char **argv) {
  if (argc != 3) { fatal("Usage: webserver <port> <webroot>"); }

  const char *port = argv[1];
  const char *webroot = argv[2];

  int serverfd = open_listenfd((char*) port);
  if (serverfd < 0) { fatal("Couldn't open server socket"); }

  while (1) {
    int clientfd = Accept(serverfd, NULL, NULL);
    if (clientfd < 0) { fatal("Error accepting client connection"); }
    server_chat_with_client(clientfd, webroot);
    close(clientfd);
  }
}
```

`open_listenfd`: create server socket

`Accept`: wait for incoming connection

Chat with client

The server_chat_with_client function reads a client request and generates an appropriate response:

```
void server_chat_with_client(int clientfd, const char *webroot) {
  struct Message *req = NULL;
  rio_t in;

  rio_readinitb(&in, clientfd);

  req = message_read_request(&in);
  printf("got request for resource %s\n", req->resource);
  if (req) {
    server_generate_response(clientfd, req, webroot);
    message_destroy(req);
  }
}
```

Header and Message types

It's useful to have data types representing protocol messages:

```
/* data type for message headers */
struct Header {
  char *name;
  char *content;
};

/* Message data type, represents a request from a client */
struct Message {
  int num_headers;           /* number of headers */
  struct Header **headers;   /* array of headers */
  char *method;              /* the method */
  char *resource;            /* the resource requested */
};
```

Note that with additional fields, struct Message could also represent a response

Reading lines

HTTP uses lines (terminated by CRLF) to structure messages, so a function to read a line of text robustly is very helpful:

```
ssize_t readline(rio_t *in, char *usrbuf, size_t maxlen) {
  ssize_t len = rio_readlineb(in, usrbuf, maxlen);
  if (len > 0 && usrbuf[len-1] == '\n') {
    /* trim trailing LF (newline) */
    usrbuf[len-1] = '\0';
    len--;
  }
  if (len > 0 && usrbuf[len-1] == '\r') {
    /* trim trailing CR */
    usrbuf[len-1] = '\0';
    len--;
  }
  return len;
}
```

Heavy lifting done by rio_readlineb

Reading a Message

```
struct Message *message_read_request(rio_t *in) {
  struct Message *result = NULL;

  read first line (method and resource)

  read 0 or more headers

  read optional body

  return result;
}
```

This is a fairly complicated function

Read first line of request

First line of request has essential information: method, resource, and protocol

```
len = readline(in, linebuf, MAX_LINE_LEN);
if (len < 0) { goto read_done; }
char *savep, *method, *resource, *proto;
method = strtok_r(linebuf, " ", &savep);
if (!method) { goto read_done; }
resource = strtok_r(NULL, " ", &savep);
if (!resource) { goto read_done; }
proto = strtok_r(NULL, " ", &savep);
if (!proto || strcmp(proto, "HTTP/1.1") != 0) { goto read_done; }
```

strtok_r used to tokenize the line

Error handling simplified using goto: not as awful as it sounds, see full code

Reading a header

Body of loop to read 0 or more headers (greatly simplified)

```
len = readline(in, linebuf, MAX_LINE_LEN);
if (strcmp(linebuf, "") == 0) {
  done_with_headers = 1;  /* read a blank line */
} else {
  /* try to read a header */
  char *p = strchr(linebuf, ':');
  if (p) {
    separate header into name and content parts
    struct Header *hdr = create and initialize Header object
    msg->headers[num_headers] = hdr;
    num_headers++;
  }
}
```

strchr function used to find ':' separating name and content of header

Headers are terminated by a blank line

Reading request body

TODO, example implementation doesn't attempt to read request body

Will be fairly complicated due to encoding schemes, compression, etc.

Left as exercise for reader ☺

Handling untrusted data

It is incredibly important to realize that data read from the client is *untrusted*

A network application which connects to untrusted peers must assume that they are malicious!

In general, *never* under any circumstances:

- Trust that data is properly formatted
- Trust no special characters are present
- Trust that message size limits are not exceeded

Example security vulnerability

Let’s say we have a line-oriented protocol, and a line will have two fields containing (per the protocol spec) at most 100 characters each

Can you spot the problem in the following code to parse a line?

```
    char buf[1024], field1[256], field2[256];
    rio_readlineb(in, buf, 1024);
    sscanf(buf, "%s %s", field1, field2);
```

Buffer overflows

A *buffer overflow* occurs when malicious peer sends more data than can be received into the recipient's buffer

If the recipient's buffer is stack allocated, the malicious client could overwrite the return address in the current stack frame with an arbitrary value

When the function returns, it jumps to an address controlled by the malicious peer

For example: it could be possible for the client to cause the program to call the system function, which executes an arbitrary program as a subprocess

Another example

From the textbook's web server implementation, reading the first line of an HTTP request:

```
    char buf[MAXLINE], method[MAXLINE], uri[MAXLINE], version[MAXLINE];

    Rio_readlineb(&rio, buf, MAXLINE);
    sscanf(buf, "%s %s %s", method, uri, version);
```

Is this code vulnerable to buffer overflow?

Responding to a request

```
void server_generate_response(int clientfd, struct Message *req, const char *webroot) {
  if (strcmp(req->method, "GET") != 0) {
    server_generate_text_response(clientfd, "403", "Forbidden",
      "only GET requests are allowed");
    goto response_done; }
  filename = concat(webroot, req->resource);
  struct stat s;
  if (stat(filename, &s) < 0) {
    server_generate_text_response(clientfd, "404", "Not Found",
      "requested resource does not exist");
    goto response_done; }

  writestr(clientfd, "HTTP/1.1 200 OK\r\n");
  write Content-Type and Content-Length headers
  read data from file and copy to clientfd
response_done:
  cleanup
}
```

Sending a text response

Error messages (such as 403 and 404 responses) are sent back as text

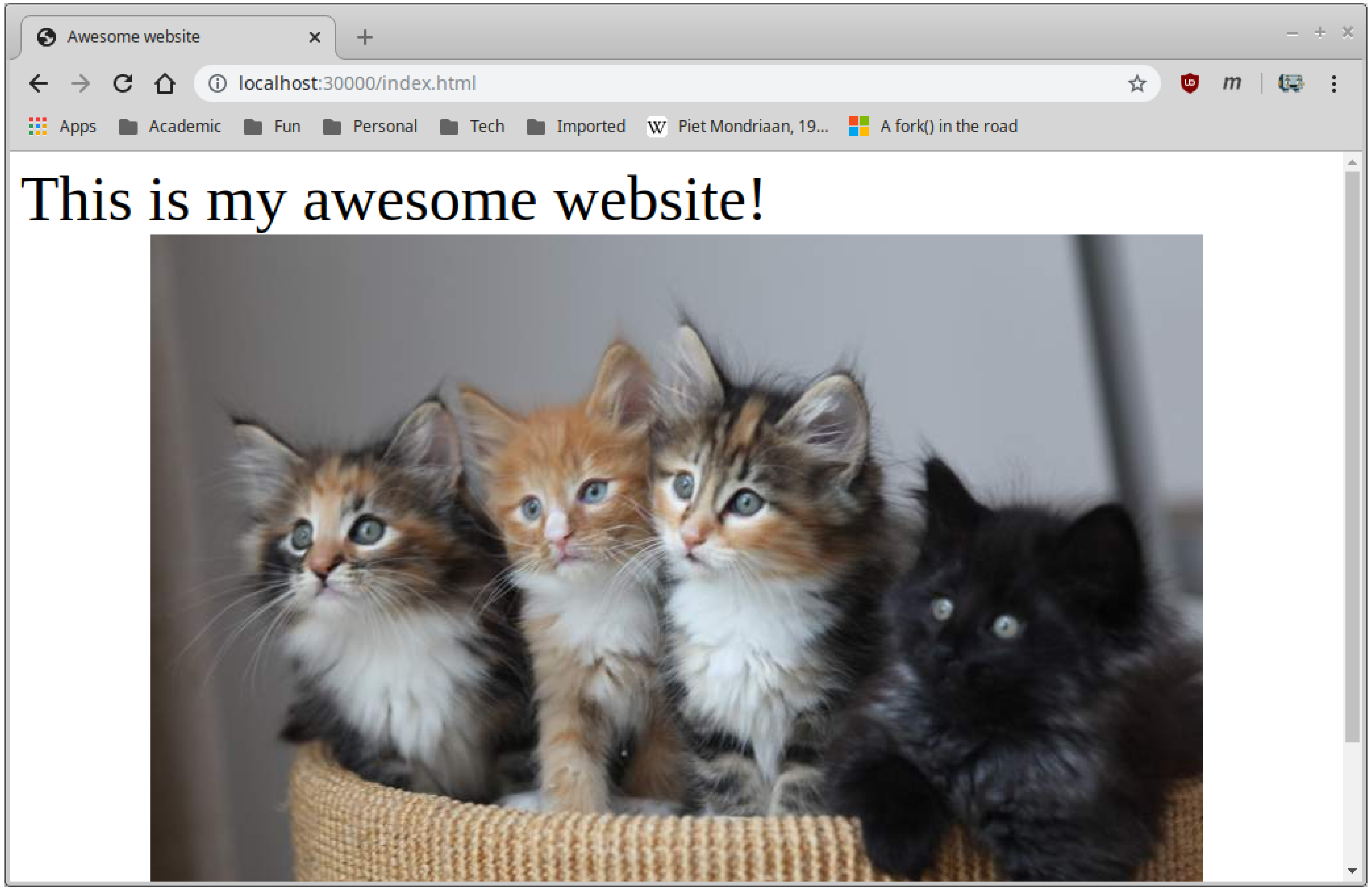
```
void server_generate_text_response(int clientfd, const char *response_code,
  const char *reason, const char *msg) {
  writestr(clientfd, "HTTP/1.1 ");
  writestr(clientfd, response_code);
  writestr(clientfd, " ");
  writestr(clientfd, reason);
  writestr(clientfd, "\r\n");
  /* could generate headers... */
  writestr(clientfd, "\r\n");
  writestr(clientfd, msg);
}
```

Putting it all together

```
$ gcc -o webserver main.c webserver.c csapp.c -lpthread
$ mkdir site
$ curl http://placekitten.com/800/600 -o site/kittens.jpg
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100 45798    0 45798    0     0   813k      0 --:--:-- --:--:-- --:--:--  813k
$ cat > site/index.html
<html><head><title>Awesome website</title></head>
<body>
  <div style="font-size: 300%;">
    This is my awesome website!
  </div>
  <center>
    <img src="kittens.jpg">
  </center>
</body></html>
$ ./webserver 30000 ./site
```

Result

Connect to: `http://localhost:30000/index.html`



This is my awesome website!

HTTP client implementation

HTTP client

Next, let's develop a simple HTTP client (a bit like curl)

Full-blown web browser: incredibly complicated

Our client:

- Parse URL
- Connect to server
- Request resource
- Read headers
- Save body of resource to file

Full source code in webclient.zip on course web page

Webclient main

Overall main function:

```
int main(int argc, char **argv) {
   check command line arguments

   parse URL
   connect to server
   send request
   read response headers
   read response body
}
```

Parse URL

A URL (Uniform Resource Locator) encodes the protocol, name, and location of a data resource

Example: http://placekitten.com/1024/768

- Protocol is http
- Host (location) is placekitten.com
- Resource name is /1024/768

TCP port for (unencrypted) HTTP is 80

- Can be overridden in URL (e.g., http://localhost:30000)

Code to parse URL

```
/* parse URL */
if (!strncpy(url, "http://", 7) != 0)
  { fatal("only http URLs are supported"); }
url += 7;
char *host, *resource;
host = url;
resource = strchr(host, '/');
if (!resource) { fatal("Invalid URL"); }
*resource = '\0';
resource++;
/* see if a port was specified */
const char *port = "80"; /* default HTTP port */
char *p = strchr(host, ':');
if (p) {
  *p = '\0';
  port = p+1;
}
```

Connect, request resource

Once host and port have been determined, connecting and sending request is fairly straightforward:

```
/* Connect to server */
int connfd = Open_clientfd((char *)host, (char *)port);

/* Send request */
writestr(connfd, "GET /");
writestr(connfd, resource);
writestr(connfd, " HTTP/1.1\r\n");
writestr(connfd, "Host: ");
writestr(connfd, host);
writestr(connfd, "\r\nUser-Agent: jhucsf/0.1\r\n");
writestr(connfd, "Accept: */*\r\n\r\n");
```

Reading response

First line of response indicates status code:

```
/* Read response */
char linebuf[LINEBUF_SIZE];
Rio_readlineb(&in, linebuf, LINEBUF_SIZE);
printf("First line: %s\n", linebuf);

/* make sure response code was 200 */
char *p2 = strchr(linebuf, ' ');
if (!p2) { fatal("bad HTTP response?"); }
p2++;
if (strncmp(p2, "200", 3) != 0) { fatal("HTTP response not 200"); }
```

Reading headers

Client reads headers, looking for Transfer-Encoding and Content-Length

Client needs to know encoding of message body in order to know how to read and decode it

Our client will only support chunked encoding

Headers are terminated by a blank line, message body follows

Reading message body

The chunked encoding consists of chunks of binary data, each preceded with a size in bytes (encoded as a hexadecimal integer):

```
while (1) {
  /* determine size of next chunk */
  readline(&in, linebuf, LINEBUF_SIZE);
  unsigned chunk_len;
  sscanf(linebuf, "%x", &chunk_len);
  /* if chunk size is 0, body has ended */
  if (chunk_len == 0) { break; }
  /* read chunk, write to output file */
  char *buf = xmalloc(chunk_len);
  rio_readnb(&in, buf, chunk_len);
  rio_writen(outfd, buf, chunk_len);
  free(buf);
  /* need to read another \r\n following the chunk */
  Rio_readnb(&in, linebuf, 2);
}
```

Running the program

```
$ ./webclient http://placekitten.com/1152/864 kitten.jpg
host: placekitten.com
port: 80
resource: /1152/864
First line: HTTP/1.1 200 OK

Header: Date: Sat, 16 Nov 2019 01:15:05 GMT
Header: Content-Type: image/jpeg
Header: Transfer-Encoding: chunked
Chunked encoding
Header: Connection: keep-alive
Header: Set-Cookie: __cfduid=dd8f10ca64c91fce89a52b57aa4ccede71573866904; expires=Sun, 1
Header: Access-Control-Allow-Origin: *
Header: Cache-Control: public, max-age=86400
Header: Expires: Thu, 31 Dec 2020 20:00:00 GMT
Header: CF-Cache-Status: MISS
Header: Server: cloudflare
Header: CF-RAY: 53659c9958fbea34-IAD
Download successful?
```

Result

Conclusions

Conclusions

HTTP is a complicated protocol!

- Lots of historical cruft
- But: solves genuine problems informed by experience

Our example server and client are not fully general

- Hopefully demonstrate the elements of a more general approach