Compsys - A3

Fie Hammer - gsr530, Nikolaj Ingemann Gade - qhp695 og
Frederikke Levinsen

November 21, 2021

Part I
Theoretical Part

1 Store and Forward

1.1 Processing and Delay

There are a total of 4 typical types of delays when it comes to Packet switches networks.
Nodal processing, where the delay typically occurs due to the processing, and more specifically, due to the examination of the header and where it should be directed.
Queuing delay, where the delay typically occurs due to queuing and the data waiting to be transmitted until it is the datas turn.
Transmission delay, where like in queuing delay, other packets has to be transmitted and has their turn before our current packet.
Propagation delay (Wasn't needed to be explained)

1.2 Transmission Speed

1.2.1 Part 1

We will look at the round trip time for figure 1 given in the assignment. We note that the Propagation delay we assume that the propagation speed in all links visible is $2,4 \cdot 10^8$ m/s. We also note that the queuing delay is a part of the time stated as node delays in the figure. It is stated that the propagation delay may be removed from the calculation. We will calculate it, but the reason for leaving it out may be because of the very small delay times that it may cause. We are down in nanoseconds when talking about propagation delay. This is also why cables can not be extremely long as the frequency of the signal will then be off and cause complications.

So we can calculate the propagation delay for the 5 meter cable linking the access point and modem as that is the only cable where the prop. delay is not stated/included and the cable is visible. it is calculated as $\frac{d}{s}$, where d is the physical distance of the link and s is the propagation speed. So we get that $\frac{5m}{2.4 \cdot 10^8 m/s} \approx 2.08 \cdot 10^{-8} s$. so the propagation delay is 20.8 nanoseconds for the cable between the access point and modem.

The round trip time is the time it takes for a package to travel from the client to the server and back. As the round trip is the time is takes for the communication it includes packet-propagation delays, packet-queuing delays in intermediate routers and switches, and packet-processing delays. If we look at the nodal delay it includes Processing, queueing, transmission and propagation delay, such that the total nodal delay is given by: dnodal = dproc + dqueue + dtrans + dprop. If we add the calculated propagation delay to the first nodal delay, we should thereby be able to add all the nodal delays to calculate the RTT of an unknown package size.

RTT = 2 $\cdot (2ms + 20.8ns + 1ms + 5ms + 24ms) \approx 64ms$.

A RTT at around 64 ms is fine and causes no problems, whereas an RTT at above 100ms might cause problems and delays and above 375 ms the connection is terminated.

1.2.2 Part 2

We will now assume that the package has a size of 640 KB (kilo Bytes). The upload is complete when all bytes have been transferred. With the RTT from part 1 the total transmission time now includes the speed between the distances. That speed is given in bits per seconds. We will thereby look at the sent package in bits that we are going to send and we know that we have 1 byte for 8 bits. This makes the package the size of 640*8 = 5120 kb (kilo bits) or 5,12 Mb (Mega bits). This way we can calculate the seconds it takes for it to send between destinations.

The package has to be send over 54 Mb/s, 100Mb/s, 2 Mb/s and 1 Gb/s.

$\frac{5.12Mb}{54Mb/s} + \frac{5.12Mb}{100Mb/s} + \frac{5.12Mb}{2Mb/s} + \frac{5.12Mb}{1000Mb/s} = 0.09481s + 0.0512s + 2.56s + 0.00512s = 2.71113s$

So it takes around 2.71 seconds just to send the package. With all of the delay etc. (The RTT), we get the total time to be, $2.71113s + 0.064s = 2.77513s$, so it takes around 2,78 seconds for a person to send a 640 KB data package (including overhead) sent to the diku.dk webserver.

2 HTTP

2.1 HTTP Semantics

2.1.1 Part 1

The method field in request is important as it can take different values such as GET which is always defined. The GET value/method i responsible for retrieving the data which is identified by the Uniform ressorce identifier (URI) and the data is returned/retrieved that it relates to. The POST method on the other hand creates an object and binds to a specific object. A message-id field of the object is then set by client of server and the Uniform Resource Locator (URL) allocated by the server for that object is returned to the client. This can be done if a client fills in a form such that the typed information is saved. However this can also be done by the method GET, but the link is then altered/extended to include the typed information from the client.

2.1.2 Part 2

The Host header is important and necessary as it specifies the host and is required by Web proxy caches.

2.2 HTTP Headers and Fingerprinting

2.2.1 Part 1

Cookies are used from webpages to store user data for instance what users put in their shopping cards, for spacial access for that particular user or to store information about log ins for the user. They are used to track the users activity at a webpage. The SET -Cookie is for storing the info and Cookie is the identification used to get the stored data/cookies. Cookies can also be used to direct commercials and such to a client. If A client searches a lot after training gear, Commercials can be directed at that.

2.2.2 Part 2

The Entity tags (ETag) uses a weak algorithm and for a webpage to be considered identical to another does not need it to be identical down to each byte. HEAD returns only the Header, not the content associated with the link/header.

3 Domain Name System

3.1 DNS Provisions

The domain name system (DNS) is an application layer which makes it easier for people to read the network addresses.Each ressource on the network is identified by their unique domain name. The DNS must be fault tolerant which is a critical point as if it crashes it will mean that so does the internet. It needs to be efficient which means that there needs to be servers in some big cities around the world. This means that not all access must com from a single point. It will also lower the maintenance as it does not require updating for each person in the world, but only a part. So there exist root DNS servers, top-level domain (TLD) DNS servers and authoritative DNS servers. TLD servers include dot com etc. and dot (the initials for a country) and so forth.

Three of the most important goals of DNS are to ensure fault tolerance, scalability and efficiency. Explain how these insurances can (and are) met in practice. (Answer with 2-4 sentences.)

3.2 DNS Lookup and Format

3.2.1 Part 1

3.2.2 Part 2

Part II
Programming Part

1 Introduction

We are aiming to create an implementation of a download-only torrent peer in C. The program must be able to parse cascade files, request peers from a tracker, request blocks from a peer, and receive and process blocks from peers.

2 Implementation

2.1 `csc_parse_file()`

This function is given the file name of a cascade file and returns an instance of the `csc_file_t` structure, which contains all pertinent data. After checking whether the file exists and if the 8 starting bytes are correct, the function initializes a `csc_file_t` instance and starts setting the members using the data from the cascade file's header.

```c
csc_file_t* casc_file_data = (csc_file_t*)malloc(sizeof(csc_file_t));

casc_file_data->targetsize = be64toh(*((unsigned long long*)&header[16]));
casc_file_data->blocksize = be64toh(*((unsigned long long*)&header[24]));
uint8_t x[32];
for (int i = 0; i < 32; i++) {
    x[i] = (uint8_t)(*((unsigned long long*)&header[32+i]));
}

csc_hashdata_t* hash = malloc(sizeof(csc_hashdata_t));
memcpy(hash->x,x,SHA256_HASH_SIZE);

casc_file_data->targethash = *hash;
```

Then, the last members are calculated and set:

```c
casc_file_data->blockcount = 1 + floor(
    (casc_file_data->targetsize - 1.0)/casc_file_data->blocksize
);
casc_file_data->trailblocksize = casc_file_data->targetsize - (
    (casc_file_data->blockcount - 1) * casc_file_data->blocksize
);
casc_file_data->completed = 0;
```

The final member to be set, `completed`, was added to the structure by us. It shows whether all blocks have been downloaded correctly.

After parsing the header of the cascade file, the rest of the file (which contains all of the block hashes) is used to initialize instances of the `csc_block_t` structure.

```
csc_block_t* block_list = malloc(
    sizeof(csc_block_t) * casc_file_data->blockcount
);

casc_file_data->blocks = block_list;

for (unsigned long long b = 0;b < casc_file_data->blockcount; b++) {
    csc_block_t* block = &(block_list[b]);
    block->index = b;
    block->offset = b * casc_file_data->blocksize;
    if (b == casc_file_data->blockcount - 1 ) {
        block->length = casc_file_data->trailblocksize;
    } else {
        block->length = casc_file_data->blocksize;
    }

    block->completed = 0;

    uint8_t block_x[32];
    if (fread(block_x, 1, 32, fp) != 32) {
        printf("Cascade file not readable\n");
        fclose(fp);
        return NULL;
    }
    csc_hashdata_t* hash = malloc(sizeof(csc_hashdata_t));
    memcpy(hash->x,block_x,SHA256_HASH_SIZE);
    block->hash = *hash;
}
```

Finally, the file is closed and `check_blocks` is called in order to check if any of the file has already been downloaded.

2.2 `check_blocks()`

This function takes in a `csc_file_t` instance and a `char*`, which is the path of the target destination, and returns the given `csc_file_t` with the completion status of each block and the entire file being marked correctly.

The function consists mostly of code that was originally intended for `casc_parse_file()`, but since the code is called several times during the program, it made sense to have the code in a separate function.

The function start off by setting the given `csc_file_t` instance's `completed` member to 1. This will be changed if any of the blocks, or the whole file, has an incorrect hash.

```
casc_file_data->completed = 1;
```

The function goes through the file and compares the hash of each block with its corresponding hash in the cascade file. If the hashes match up, the `completed` member of the corresponding `csc_block_t` instance is set to 1. If not, the `completed` member of both the block and the whole file is set to 0.

```
void* buffer = malloc(casc_file_data->blocksize);
```

```
SHA256_CTX shactx;
for(unsigned long long i = 0; i < casc_file_data->blockcount; i++)
{
    uint8_t* shabuffer = malloc(sizeof(uint8_t) * SHA256_HASH_SIZE);
    unsigned long long size = casc_file_data->blocks[i].length;
    if (fread(buffer, size, 1, fp) != 1)
    {
        break;
    }

    sha256_init(&shactx);
    sha256_update(&shactx, buffer, size);
    sha256_final(&shactx, shabuffer);

    if (memcmp((&(&casc_file_data->blocks[i])->hash)->x, shabuffer, 32) == 0) {
        (&casc_file_data->blocks[i])->completed = 1;
    } else {
        (&casc_file_data->blocks[i])->completed = 0;
        casc_file_data->completed = 0;
    }
}
free(buffer);
```

If all the blocks turn out to be fully downloaded, the `completed` member of the file will still be 1. If this is the case, the hash of the whole file will be checked against the one provided by the cascade file. If the hash is not correct, all blocks are set to not be completed. This is because at least one of the completed blocks must be wrong, but since all the block hashes are correct, we have no way of knowing which one.

```
if (casc_file_data->completed) {
    rewind(fp);
    buffer = malloc(casc_file_data->targetsize);
    uint8_t* shabuffer = malloc(sizeof(uint8_t) * SHA256_HASH_SIZE);
    fread(buffer, casc_file_data->targetsize, 1, fp);

    sha256_init(&shactx);
    sha256_update(&shactx, buffer, casc_file_data->targetsize);
    sha256_final(&shactx, shabuffer);

    if (!(memcmp((&casc_file_data->targethash)->x, shabuffer, 32) == 0)) {
        casc_file_data->completed = 0;
        for (unsigned long long i = 0;i < casc_file_data->blockcount;i++) {
            (&casc_file_data->blocks[i])->completed = 0;
        }
    }
    free(buffer);
}
```

Finally, the `csc_file_t` instance, containing all the data about which blocks are completed, is returned.

2.3 get_peers_list()

The function starts off by establishing a connection to the tracker:

```
rio_t rio;
uint8_t rio_buf[MAX_LINE];

int tracker_socket;

tracker_socket = Open_clientfd(tracker_ip, tracker_port);
Rio_readinitb(&rio, tracker_socket);
```

Then a request for a list of peers is created and sent. The request is made using the `RequestHeader` and `RequestBody` structures.

```
struct RequestHeader request_header;
memcpy(request_header.protocol, "CASC", 4);
request_header.version = htonl(1);
request_header.command = htonl(1);
request_header.length = htonl(BODY_SIZE);
memcpy(rio_buf, &request_header, HEADER_SIZE);

struct RequestBody request_body;
memcpy(request_body.hash, hash, 32);

inet_aton(my_ip, &request_body.ip);
request_body.port = be16toh(atol(my_port));
memcpy(&rio_buf[HEADER_SIZE], &request_body, BODY_SIZE);

Rio_writen(tracker_socket, rio_buf, MESSAGE_SIZE);
```

The reply is then read and the header of the response is copied into a `char[]` named `reply_header`. The length of the message is read from the header.

```
Rio_readnb(&rio, rio_buf, MAX_LINE);

char reply_header[REPLY_HEADER_SIZE];
memcpy(reply_header, rio_buf, REPLY_HEADER_SIZE);

uint32_t msglen = ntohl(*(uint32_t*)&reply_header[1]);
```

After some error-checking, the peers are created as instances of the `csc_peer_t` structure, and the members of each are set using the data from the reply. Finally, the total amount of peers is returned.

```
int peercount = msglen/12;
*peers = malloc(peercount * sizeof(csc_peer_t));

for (int i = 0;i<peercount;i++) {
    csc_peer_t peer;
    uint8_t* peer_data = &rio_buf[REPLY_HEADER_SIZE + 12*i];
    sprintf(peer.ip, "\%u.\%u.\%u.\%u", peer_data[0], peer_data[1], peer_data[2], peer_data[3]);
    sprintf(peer.port, "\%u", be16toh(*((uint16_t*)&peer_data[4])));

    (*peers)[i] = peer;
}
Close(tracker_socket);
return peercount;
```

2.4 get_block()

This function attempts to download a given block from a given peer and write it to the target file.

It starts by initializing a buffer large enough to hold the entire block:

```
int buffer_size;
if (block->length + PEER_RESPONSE_HEADER_SIZE > MAX_LINE) {
    buffer_size = block->length + PEER_RESPONSE_HEADER_SIZE;
} else {
    buffer_size = MAX_LINE;
}
char rio_buf[buffer_size];
```

Then, a connection is established to the peer, and a request is created with the `ClientRequest` structure and sent:

```
rio_t rio;
int peer_socket;
peer_socket = Open_clientfd(peer.ip, peer.port);
Rio_readinitb(&rio, peer_socket);

struct ClientRequest request;
memcpy(request.protocol, "CASCADE1", 8);
for (int i = 0;i<16;i++) {
    request.reserved[i] = 0;
}
request.block_num = be64toh(block->index);
memcpy(request.hash, hash, 32);

memcpy(rio_buf, &request, PEER_REQUEST_HEADER_SIZE);
Rio_writen(peer_socket, rio_buf, PEER_REQUEST_HEADER_SIZE);
```

Like in `get_peer_list`, we read the header of our reply and figure out the message length.

The SHA256 hash of the given data is computed and compared with the expected hash:

```
uint8_t* shabuffer = malloc(sizeof(uint8_t) * SHA256_HASH_SIZE);

SHA256_CTX shactx;
sha256_init(&shactx);
sha256_update(&shactx, &rio_buf[PEER_RESPONSE_HEADER_SIZE], msglen);
sha256_final(&shactx, shabuffer);

if (memcmp(shabuffer, (&block->hash)->x, SHA256_HASH_SIZE) != 0) {
    printf("Not the same hash\n");
    Close(peer_socket);
    return;
}
```

Finally, the data is written to the target file:

```
FILE* fp = fopen(output_file, "rb+");
if (fp == 0)
{
    printf("Failed to open destination: %s\n", output_file);
    Close(peer_socket);
    return;
}

fseek(fp, block->offset, SEEK_SET);
fwrite(&rio_buf[PEER_RESPONSE_HEADER_SIZE],msglen,1,fp);
```

2.5 download_only_peer()

The `download_only_peer()` is the main function of the program, and performs the necessary actions to download the requested file.

The functions starts by checking whether the cascade file exists and extracts the name of the target file. It then calls `casc_parse_file()` to get an instance of a `csc_file_t` structure with the correct cascade data.

```
printf("Managing download only for: \%s\n", cascade_file);
if (access(cascade_file, F_OK ) != 0 )
{
    fprintf(stderr, ">> File \%s does not exist\n", cascade_file);
    exit(EXIT_FAILURE);
}

char output_file[strlen(cascade_file)];
memcpy(output_file, cascade_file, strlen(cascade_file));
char* r = strstr(cascade_file, "cascade");
int cutoff = r - cascade_file ;
output_file[cutoff-1] = '\0';
printf("Downloading to: %s\n", output_file);

casc_file = csc_parse_file(cascade_file, output_file);
```

The function then loops through the blocks and creates a queue of the uncompleted ones.

```
int uncomp_count = 0;
queue = malloc(casc_file->blockcount * sizeof(csc_block_t*));
for (unsigned long long i = 0;i<casc_file->blockcount;i++) {
    if ((&casc_file->blocks[i])->completed == 0) {
        queue[uncomp_count] = &casc_file->blocks[i];
        uncomp_count++;
    }
}
```

The hash of the cascade file is then computed, so that it can be used to request peers from the tracker.

```
uint8_t hash_buf[32];
get_file_sha(cascade_file, hash_buf, 32);
```

The function now begins a while loop, looping indefinitely until the entire file is downloaded. The loop starts off with running `get_peer_list()` in a loop until at least 1 peer is found. Then, a peer is selected. If a good peer exists, that one is selected.

Then, a for loop is begun, calling `get_block()` for each uncompleted block, using the peer selected before. Once that loop is done, `check_blocks()` is called, to test if all the blocks were downloaded successfully.

When `check_blocks()` marks the file as completed, the while loop is exited. The function ends with freeing the used resources.

```
while (!casc_file->completed) {
    int peercount = 0;
    while (peercount == 0)
    {
        peercount = get_peers_list(&peers, hash_buf);
        if (peercount == 0)
        {
            printf("No peers were found. Will try again in %d seconds\n", PEER_REQUEST_DELAY);
            fflush(stdout);
            sleep(PEER_REQUEST_DELAY);
```

```
        }
        else
        {
            printf("Found %d peer(s)\n", peercount);
        }
    }

    csc_peer_t peer = (peers[0]);
    // Get a good peer if one is available
    for (int i=0; i<peercount; i++)
    {
        if (peers[i].good)
        {
            peer = (peers[i]);
        }
    }

    for (int i=0; i<uncomp_count; i++)
    {
        get_block(queue[i], peer, hash_buf, output_file);
    }

    printf("\n");
    casc_file = check_blocks(output_file,casc_file);
}

printf("File fully downloaded\n");
free_resources();
```

3 Result

Our code works as intended and is able to communicate with the provided python programs when running locally, as well as the provided remote tracker and peer.

The speed of the download is dependent on the size of the blocks. Smaller blocks results in a higher download time, likely because more code is being run and more requests are being sent, received, processed, and responded to. The difference in download time between files of different block sizes is even more clear when requesting blocks from a remote peer, which is likely a result of a larger ping.

4 Tests

We have tested our implementation by downloading the files described by each of the 5 cascade files, from both a local peer and a remote one. The resulting files are identical to the ones our peer provided.

5 Limitations and Potential problems

6 Conclusion

The code works as it should and communication has been made with the server (the provided python programs).