A4: Computer Networking (II)

Computer Systems 2021
Department of Computer Science
University of Copenhagen

Kenneth Skovhede and David Gray Marchant

Due: Sunday, 5th of December, 16:00
Version 1 (November 22, 2021)

This is the fifth assignment in the course on Computer Systems 2021 at DIKU and the second on the topic of Computer Networks. We encourage pair programming, so please form groups of 2 or 3 students. Groups cannot be larger than 3, and we strongly recommend that you do not work alone.

For this assignment you will receive a mark out of 4 points; you must attain at least half of the 8 possible point between A3 and A4 to be addmitted to the exam. For details, see the *Course description* in the course page. This assignment belongs to the CN category together with A3. Resubmission is not possible.

It is important to note that only solving the theoretical part will result in 0 points. Thus, the theoretical part can improve you score, but you *have* to give a shot at the programming task.

The web is more a social creation than a technical one. I designed it for a social effect — to help people work together — and not as a technical toy. The ultimate goal of the Web is to support and improve our weblike existence in the world. We clump into families, associations, and companies. We develop trust across the miles and distrust around the corner.

— Tim Berners-Lee, Weaving the Web (1999)

Overview

This assignment has two parts, namely a theoretical part (Section 1) and a programming part (Section 2). The theoretical part deals with questions that have been covered by the lectures. The programming part requires you to fill in the blanks of a file transfer service using socket programming in C. The implementation task of the complete P2P network is spread over this assignment and the previous one (A3). You are encouraged to use your own submission for A3 as a basis for this assessment, though a sample submission for A3 has been provided. In this assignment, the programming effort relies solely on building the client portion of the architecture. More details will follow in the programming part (Section 2).

1 Theoretical Part (25%)

Each section contains a number of questions that should be answered **briefly** and **precisely**. Most of the questions can be answered within 2 sentences or less. Annotations have been added to questions that demand longer answers, or figures with a proposed answer format. Miscalculations are more likely to be accepted, if you account for your calculations in your answers.

1.1 Transport protocols

The questions relating to TCP in this section do not assume knowledge of the flow and congestion control parts of TCP.

1.1.1 TCP reliability and utilization

Part 1: Is the 3-way-handshake in TCP really needed? Can you suffice with less? Explain why or why not.

Part 2: How does TCP facilitate a *full-duplex*[1] connection?

1.1.2 Reliability vs overhead

Part 1: Considering the data transferred over a network, how does a TCP connection add overhead relative to UDP?

Part 2: Explain the TCP notion of reliable data transfer. In which way are TCP connections reliable? Are packets **always** delivered?

1.2 TCP: Principles and practice

Some of the questions below refer to one or more *Request For Comments* (RFC) memorandums posted by the *Internet Engineering Task Force* (IETF). In most cases, you can find answers to the question in the book, but we recommend that you browse through these RFCs to see how internet standards are formulated and distributed in practice.

Be careful when answering the questions below. If any ambiguity may arise, remember to specify the point of view of your answers – is it from the server or the client or both?

1.2.1 TCP headers

Part 1: The TCP segment structure is shown in figure 3.29 in K&R (or section 1.3 of RFC793[2]).

1.1. What is the purpose of the RST bit? Give an example of when a TCP stack may return a RST packet.

[1] Full speed bidirectional data transfer

[2] `https://tools.ietf.org/html/rfc793#section-3.1`

1.2. What does the sequence number and acknowledgement number headers refer to for a given connection? Is there any relation between server and client acknowledgement and sequence numbers?

1.3. What is the purpose of the window? What does a positive window size signify?

1.4. If the window size of a TCP receiver is 0, what happens then at the sender? How can the sender find out when the receiver window is not 0?

Part 2: In one of the previous questions, you were asked why the 3-way handshake was necessary. One of the practical reasons for this 3-way-handshake was the synchronization of sequence numbers. In some situations, the client can initiate its transfer of data after having received the SYN-ACK-packet. Explain when this is possible. *(Hint: What does the server need to tell the client in the SYN-ACK-packet?)*

1.2.2 Flow and Congestion Control

Part 1: Explain the type of congestion control that modern TCP implementations employ. Is it network assisted congestion control?

Part 2: Congestion control translates directly to limiting the transmission rate. How does TCP determine the transmission rate when doing congestion control?

Part 3: How does a TCP sender infer the value of the congestion window, that is, what is the basis for calculating the congestion window? *(Answer with at most 10 sentences)*

Part 4: The fast retransmit extension, as described in RFC 2581[3] allows for retransmission of packets after receiving 3 duplicate ACKs for the same segment. What does the sender need to implement in order to allow for fast retransmit? What does the receiver need to implement? If a sender supports fast retransmit and needs to do fast retransmit for a specific segment, how many ACKs should the sender receive for the previous packet?

[3] `https://tools.ietf.org/html/rfc2581`

2 Programming Part (75 %)

For the programming part of this assignment, you will implement a peer in a distributed file-sharing service.

2.1 Design and overview

The file-sharing service comprises a metadata-file, a tracker, and the peers. Each peer in the real world would be both a client and a server, and for this second assignment we will implement the servert part, meaning that the *"peer"* will now be able to serve files to other peers. This will compliment the downloading of files acheived in the last asignment to form a complete peer implementation.

The design of the P2P network is a simpler version of the popular BitTorrent protocol and is named the Cascade protocol, which hints at the relation to torrents.

Like in the BitTorrent protocol, each file that is served on the network is divided into blocks of a fixed size. The client can then choose any of the peers as the source for downloading a particular block. To ensure that the block is correct and not corrupted by the network or a malicious peer, each file is described by a list of hashes in a metadata file, dubbed a *.cascade* file, which can then be used to verify the individual blocks.

Also like the BitTorrent network, the Cascade network relies on a tracker service for discovering peers. A client can retrieve a list of active clients from the tracker as well as register as a peer.

For this assignment you are going to fully implement a "peer", using either your submission for A3, or the sample provided as a base. This peer should be able to parse one or more `.cascade` files, obtain a list of peers, download the blocks required to reconstruct the file, store the file locally, make itself known to the tracker as a potential server, receive connections from external peers, and server files to those peers. It should also be designed in such a way as to be always capable of accepting incoming communications.

2.2 API and Functionality

2.2.1 Complete "peer"

The peer you will be implementing needs to perform the three basic tasks listed below. Each should be completed according to the protocol described in Section 3. The complete peer should:

- Subscribe to the tracker - The peer should be able to send the hash of a `.cascade` in order to make it known that the peer can now serve this file to other peers.
- Recieve requests from other peers - Blocks should be downloadable by other peers, using this peer as a server.
- Non-blocking server - Although not formally part of the protocol, your peer should be capable of recieving multiple communication attempts at

a time, and potentially of acting as both a client and a server at the same time for different `.cascade` files.

As well as these three basic tasks, the requirements of A3 are still present. These are repeated below so you can judge the suitability of your submission for A3 as a basis for A4. If you were not able to complete all of these for A3, you are encouraged to make use of the sample solution provided.

- Parse the `.cascade` file - After parsing the file, you will have a structure that contains metadata about the file to be downloaded, including the file size, the size of each block, the number of blocks, and the hash of each block
- Obtain a list of peers - Using the hash of a `.cascade` you can ask the tracker for a list of peers that serve that particular file.
- Download blocks from peers - Download each of the blocks required to restore the file. After obtaining a block, verify that the hash of the block matches.

2.2.2 Test environment

One of the difficulties with developing a system that relies on network communication is that besides a network, you need to have both the client and server running, and potentially debug both simultaneously.

To assist in developing and testing your code, we have provided implementations of the tracker and the full peer, but written in Python. You can use these to get started with a setup that runs fully contained on your own machine. By running a local tracker and a local peer it is possible to debug and monitor all pieces of the communication, which is often not possible with an existing system.

You can of course also peek into or alter the Python peer and get inspiration for implementing you own client, but beware that what is a sensible design choice in Python *may* not be a good choice for C. You are also reminded that although you can alter the Python as much as you want to provide additional debugging or the like, they are currently correct implementations of the defined protocols. Be careful in making changes that you do not alter the implemenation details, as this may lead your C implementation astray.

To start an appropriate test environment you can use the same setup as in A3. An ideal starting point would be to run the tracker using the command below, run in the `python/tracker` directory:

```
python3 tracker.py config.json
```

This will start a tracker at 127.0.0.1:8888, assuming you have not altered `config.json`. You can then start an ideal peer by running another command prompt at `python/peer`. Note that this is a single very long line:

```
python3 peer.py shakespeare.1kib.txt.cascade:shakespeare
.10kib.txt.cascade:shakespeare.100kib.txt.cascade:shakes
peare.1mib.txt.cascade:shakespeare.10mib.txt.cascade 127
.0.0.1:8888 127.0.0.1:6666
```

This will start a peer capable of handling request for all five of the handed out cascade files, and will host at 127.0.0.1:6666. From here you can either add a second Python peer, or a C peer to test the system. If you are adding a second Python peer, make sure you copy it to a new directory without the data files as otherwise no data will need to be downloaded.

2.3 Run-down of handed-out code

- `src/cascade.c` contains the sample client. This acts as a solution to A3, though should not be taken as a perfect answer as numerous limitations may be present. No specific `TODO`'s have been provided this time as you are expected to be able to alter either your own design or the sample to achieve the specified protocol. Note that the structure presented in A3 was suitable for the first assignment, but significant changes may be necessary to answer A4. You can of course copy parts from the provided `c/cascade.c` and incorporate them into your own solution if you wish.
- `src/cascade.h` contains a number of useful data structures, defined to contain the various data items needed by the peer.
- `python/tracker/tracker.py` contains the tracker, written in Python. The peer can run with Python 3.7 or newer, and relies on a config file in JSON format.
- `python/peer/peer.py` contains a peer implementation, written in Python. The tracker can run with Python 3.7 or newer, and relies on command-line inputs. Next to this peer a number of pregenerated cascade files have been provided for ease of testing.
- `python/fielgen/filegen.py` contains the a small program for generating `.cascade` files. To test the system with varying sizes, the tool will generate `.cascade` files for different block sizes. Some pregenerated cascade files along with the data file they relate to are also provided for ease of testing.
- `src/sha256.c` and `c/sha256.h` contain an open-source stand-alone implementation of the SHA256 algorithm, as used throughout the network.
- As usual, we provide `csapp.c` and a `Makefile` within the `src` directory.

2.4 Testing

For this assignment, it is *not* required of you to write formal, automated tests, but you *should* test your implementation to such a degree that you can justifiably convince yourself that each API functionality implemented works, and are able to document those which do not.

Simply running the program, emulating regular user behaviour and making sure to verify the result file should suffice, but remember to note your results.

One final resource that has been provided to assist you is that an instance of tracker.py and peer.py have been remotely hosted, and which you can connect to. These are functionally identical to those contained in the handout, with the obvious difference that any communications with them will be propperly over a network.

The tracker and the peer are designed to be robust, so should be resilient to any malformed messages you send, but as they are each only a single small resource be mindful of swamping them with requests and only use them once you are confident in your system.

```
Test Tracker: 130.225.104.138:5555
```

Note that depending on whatever firewalls or other network configuration options you have, you may not be able to reach the tracker from home, but that you should be able to from within the university.

2.5 Recommended implementation progress

Unlike in A3, relatively little guidance has been provided within the code itself as to how to proceed. This checklist *may* be of some assistance in guiding progress through the assignment, and can serve as a checklist for your project. *We recommend* that you work on them in the order presented here. Note that time is dedicated at the start to planning. This is an important step in any concurrent system as careless implementation can be both disastrous and extremely difficult to debug.

2.1. Consider the requirements of a peer acting as both client and server

2.2. Create a diagram/graph/plan of the concurrent system

2.3. Check for complete data files corresponding to `.cascade` files

2.4. Subscribe to the tracker as a potential server for any compelte `.cascade` files

2.5. Recieve requests from fellow peers and server blocks as requested

2.6. Recieve and respond to requests in a non-blocking manner

2.7. Act as both client and server at the same time for different `.cascade` files

2.8. Manually test your implementation, documenting bugs found and how you fix them (if you are able to).

2.9. Meanwhile, do not neglect the theoretical questions :D they may be relevant to your understanding of the implementation task.

2.6 Report and approximate weight

The following approximate weight sums to 75 % and includes the implementation when relevant.

Please include the following points in your report:

- Document technical implementations you made for the peer - cover in short each of the points for `cascade.c` presented above as well as any additional changes you made. Each change made should be breifly justified. You should also describe what input/data your implementation is capable of processing and what it is not. (Approx. weight: 30 %)
- Highlight how you achieved a peer that can respond to communication requets in a non-blocking manner. What options did you consider and why did you use your final design? What are the limitations of your decision?

  You should use appropriate diagrams to describe your design and any possible interactions between different processes/threads/event loops. These should demonstrate the correctness and completeness of your approach. (Approx. weight: 20%)
- Discuss how your design was tested. What data did you use on what machines? Did you automate your testing in any way? Remember that you are not expected to have built a perfect solution that can manage any and all input, but you are always meant to be able to recongise what will break your solution. (Approx. weight: 10%)
- Discuss any shortcomings of your implementation, and how these might be fixed. It is not necessarily expected of you to build a fully functional peer, but it *is* expected of you to reflect on the project. (Approx. weight: 15%)

As always, remember that it is also important to document half-finished work. Remember to provide your solutions to the theoretical questions in the report pdf.

3 Protocol description

This section defines the implementation details of the components used in the `cascade` P2P file-sharing network.

3.1 Format of `cascade` files

Each `cascade` files contains information about a single file. The layout of the file header is as follows:

```
8 bytes - "CASCADE1", a string literal
8 bytes - Reserved, SHOULD be zero
8 bytes - Total file length in bytes, unsigned integer in
          network byte-order
```

```
8 bytes - Block size in bytes, unsigned integer in network
          byte-order
32 bytes - Complete file SHA256 hash
```

Following the header, the file contains a sequence of 32 bytes SHA256 hashes for each block of the file. The first hash is computed by extracting the bytes $[0, block\_size - 1]$ from the source file, and computing the SHA256 hash for this block, the next hash corresponds to the bytes $[block\_size, 2 * block\_size - 1]$ and so on.

The number of blocks (i.e. the number of hashes in the file) MUST match the computed number of blocks, using the formula:

$$blocks = floor(\frac{file\_length + block\_size - 1}{block\_size}).$$

As an example, a source file of 1000 bytes and a block size of 64 bytes yields 16 block hashes, each with a length of 32 bytes. As the header is 64 bytes the total length of the `cascade` file is $64 + 32 * 16 = 576$ bytes.

Once a file is downloaded entirely, it should have the size mentioned in the header and a SHA256 hash corresponding to the value in the header.

3.2 Tracker API

The tracker keeps information about which clients are currently active in the network (as in: it tracks the peers).

A client MAY request a list of peers currently serving a particular file by sending a request to the tracker.

The request header MUST look like the following:

```
4 bytes - "CASC", string literal
4 bytes - Protocol version, must be 1, unsigned integer in
          network byte order
4 bytes - Command, unsigned integer in network byte-order
4 bytes - Length of request data, excluding this header,
          unsigned integer in network byte-order
```

The tracker only supports two commands and the command MUST be one of these:

```
1 - List peers
2 - Subscribe peer
```

For either command `1` or `2`, the request length MUST be $32 + 4 + 2 = 38$. The data in the request MUST be in the following format:

```
32 bytes - The cascade hash
4 bytes - The peer listening IP
2 bytes - The peer listening port, in network byte-order
```

The value for "cascade hash" is the SHA256 hash of the `.cascade` file and should not be confused with the SHA256 hash of the target file, nor the SHA256 hash of each block. To obtain the SHA256 hash value one can use the `shasum` tool found on most operating systems:

```
shasum - A 256 source.cascade
```

For either command, the tracker MUST respond the same, and the response header MUST have the format:

```
1 byte - Status code
4 bytes - Length of response, excluding this header, unsigned
          integer in network byte-order
```

The body of the response is always either a list of peers currently known by the tracker, or an error message. The makeup of this body depends on the 'Status code', and will be of the length given by the 'Response length'.

If there is an error, the status code MUST be `non-zero` and the response data MUST be a human-readable error message of the length provided.

If the response is valid, the status code MUST be `zero` and the response MUST be a sequence of entries of the following format:

```
4 bytes - Peer IP
2 bytes - Peer port in network byte-order
4 bytes - Last seen timestamp, seconds since UNIX epoch,
          unsigned integer in network byte-order
1 byte - "Good peer" flag, 1 meaning a course-hosted client
1 byte - Reserved, should be zero
```

This means that each record is 12 bytes and the number of records returned can be computed by dividing the response length by 12.

Note: The tracker may return peers in random order, and may return only a subset of all peers. Repeated requests MAY return different lists.

3.3 Client protocol

Once a client has obtained one or more peers that serve the desired file, they CAN request individual blocks from the file. The request sent to a client MUST follow the layout:

```
8 bytes - "CASCADE1", a string literal
16 bytes - Reserved, SHOULD be zero
8 bytes - The block number, unsigned integer in network
          byte-order
32 bytes - The cascade hash
```

The client receiving a request MUST return a response, with a header in the following format:

```
1 byte - Status code
8 bytes - Response length, excluding this header,
          in network byte-order
```

The response body MUST be either an error message, or the data contained in the requested block. The makeup of this body depends on the 'Status code', and will be of the length given by the 'Response length'.

If there is an error, the status code MUST be `non-zero` and the response data MUST be a human-readable error message of the length provided.

The following error codes are currently defined:

```
0 - Success
1 - Invalid hash (i.e. client does not have the file)
2 - Block not present (i.e. client has not downloaded it yet)
3 - Block number too large (i.e. client has requested a block
    that does not exist)
4 - Failed to parse request
```

The client SHOULD use the appropriate status codes to report error conditions.

If the client has the requested block it MUST respond with the status code `zero` and the length of the data. The length of the data MUST be the length of a block, unless the block is the last in the file, in which case it MAY be smaller. The data following the length MUST be the data from the block.

After sending a response, the peer MAY close the connection. If the peer supports multiple requests on a single connection, it MAY keep the connection open and serve the next request. If the status code indicates a problem with the request (i.e. is not zero or 2), the peer SHOULD close the connection as additional requests are likely to produce errors as well.

Submission

The submission should contain a file `src.zip` that contains the `src` directory of the handout; this should include any and all files necessary to run your code (including `csapp.c`, `csapp.h`, your `Makefile`, and any new files or test programs that you may have written).

Any Python code should also be included in your submission as it may form part of how you tested your C code (hint). As this course is not a Python course, you will not be marked according to the quality of your Python code.

Alongside the `src.zip` containing your code, submit a `report.pdf`, and a `group.txt`. `group.txt` must list the KU ids of your group members, one per line, and do so using *only* characters from the following set:

$$\{\texttt{0x0A}\} \cup \{\texttt{0x30}, \texttt{0x31}, \ldots, \texttt{0x39}\} \cup \{\texttt{0x61}, \texttt{0x62}, \ldots, \texttt{0x7A}\}$$

Please make sure your submission does not contain unnecessary files, including (but not limited to) compiled object files, binaries, or auxiliary files produced by editors or operating systems.