École polytechnique de Bruxelles
OPERA Wireless Communications Group

ELEC-H-417: BitTorrent-like Project

Author: Jérémy Pagé

Professor: Jean-Michel Dricot

Academic Year 2016–2017

Contents

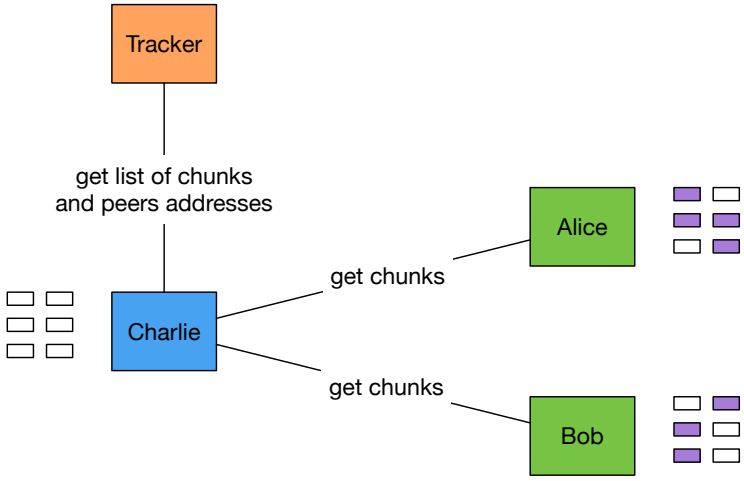


Figure 1: Schema Example

1 Goal

By groups of 3, you will create a *BitTorrent*[1]-like client and tracker. The project will have te be finished for December 21 at 12:21 PM. The grade obtained for the project will count for 25% of the final grade for the course. In addition to the source code, you will make a report explaining your solution. The choice for the programming language is left to you, but we highly recommend to use Python[2] (version 3).

You will implement 2 peers: Alice and Bob, the client: Charlie, and the Tracker. The tracker provides Charlie with the list of chunks of a file, and the peers which have some of these chunks, Alice and Bob. Charlie will download each chunk from Alice or Bob in order to recover the file (see Figure 1).

The project is divided in 3 steps (last one is optional):

1. Creation of the client *Charlie* that will download chunks of a file from 2 peers: *Alice* and *Bob*.

2. Creation of the tracker that will provide your client *Charlie* with the list of chunks to download and where these chunks are located.

3. Discover the tracker with a *broadcast* message.

In order to implement your solution, we will provide you with:

- A basic project structure (see Appendix A).
- Configuration files (see Appendix B).

[1] `https://en.wikipedia.org/wiki/BitTorrent`
[2] `https://www.python.org/`

- Useful scripts (see Appendix C).
- The definition of the protocol messages format (see Appendix D).

Besides those requirements, you will have the possibility to earn *bonus* points by completing 2 challenges:

1. You will earn **1** extra point (over 20) by implementing the third step.
2. You will earn **2** extra points (over 20) by contacting our own tracker running in the lab and successfully download the file we will provide in chunks.

2 Description of the Steps

Be sure to complete one step at a time and to be sure that the step is correctly implemented before going any further.

2.1 Step 1

You need to implement 3 entities: Alice, Bob and Charlie. Alice and Bob will have the same job: they will provide Charlie with the chunks they have. Charlie will ask Alice and Bob for missing chunks.

Alice and Bob Alice and Bob need to read the configuration file `config/peers.ini` to know their IP address and port number where they can be contacted. They do not need to read `config/file.ini`. Indeed, they can know which chunks they have by looking at their own directory: `chunks/alice/*.bin` and `chunks/bob/*.bin`.

Alice and Bob will listen for incoming requests (on their IP address and port number):

- If the request is malformed (invalid message format), they will send back an `ERROR` (see Appendix D.7) message with the error code `INVALID_MESSAGE_FORMAT`.
- If the request is not of type `GET_CHUNK` (see Appendix D.5), they will send back an `ERROR` (see Appendix D.7) message with the error code `INVALID_REQUEST`.
- If the chunk cannot be found (look to the directory content), they will send back an `ERROR` (see Appendix D.7) message with the error code `CHUNK_NOT_FOUND`.
- Otherwise, they will send back a `CHUNK` (see Appendix D.6) message with the content obtained from the file.

Note 1: The default value for the size of a chunk and the one use in the script `src/create_chunks.py` is 512 kB. However, your implementation must handle any (reasonable) chunk size (*e.g.*, 1 MB and 2 MB, which are commonly used in *BitTorrent*). You can change this value in the script to test your implementation.

Note 2: If in `config/file.ini`, Alice is set to have a specific chunk, it does not mean that she really has this chunk! Indeed, if you remove the chunk from Alice's directory, she must signal that she does not have the chunk. If Bob has the chunk, it **should** work without any problem. Do not remove a chunk if the other peer does not have it, otherwise it will not be possible to recover the file. Indeed, at least one of the peers must have the chunk.

Charlie Charlie needs to read the 2 configuration files to know which chunks to download and which peers have it. It will connect to the peer and ask for a chunk by sending a `GET_CHUNK` (see Appendix D.5) message with the chunk hash. If the peer has the chunk, it will send it back, otherwise, Charlie needs to ask to another peer.

Note: Try to minimize the opening and closing of *sockets*.

2.2 Step 2

You need to implement the tracker. Its job is to provide to Charlie the list of chunks and the peers for each chunk. Charlie will not use the configuration files anymore, except to get the IP address and port number of the tracker (`config/peers.ini`).

Tracker The tracker needs to read the 2 configuration files to know its IP address and port number, and to know the list of chunks and peers for each chunk.

The tracker will listen for incoming requests:

- If the request is malformed (invalid message format), it will send back an `ERROR` (see Appendix D.7) message with the error code `INVALID_MESSAGE_FORMAT`.
- If the request is not of type `GET_FILE_INFO` (see Appendix D.3), it will send back an `ERROR` (see Appendix D.7) message with the error code `INVALID_REQUEST`.
- Otherwise, it will send back a `FILE_INFO` (see Appendix D.4) message with the list of chunks and peers for each chunk.

Charlie Charlie will connect to the tracker and send a `FILE_INFO` (see Appendix D.4) message. The tracker will respond with the description of the chunks, and Charlie can now contact Alice and Bob to get the chunks (same as the previous step).

2.3 Step 3 (optional)

You need to add the possibility to discover a tracker. The tracker will respond to discover requests and Charlie will broadcast these requests to discover the IP address and port number of the tracker. Charlie will not use the configuration files, no exception.

Note: There can be several available trackers with different IP addresses. You will need to choose one if this is the case.

Tracker The tracker will listen for broadcasting messages (UDP) on port **9000**:

- If the request is malformed (invalid message format), it will send back an `ERROR` (see Appendix D.7) message with the error code `INVALID_MESSAGE_FORMAT`.
- If the request is not of type `DISCOVER_TRACKER` (see Appendix D.1), it will send back an `ERROR` (see Appendix D.7) message with the error code `INVALID_REQUEST`.
- Otherwise, it will send back a `TRACKER_INFO` (see Appendix D.2) message with its IP address and port number (from the configuration file).

Charlie Charlie will broadcast a `DISCOVER_TRACKER` (see Appendix D.1) message on port **9000** and listen for responses. Amongst the responses, Charlie will choose one tracker and request for file information (same as the previous step).

3 Execution

If you use Python as programming language, you will execute your application with the following commands.

3.1 Step 1

Run Alice and Bob in 2 different terminals:

```
python3 src/alice.py
python3 src/bob.py
```

Then, run Charlie:

```
python3 src/charlie.py 1
```

where `1` is the step number.

> **Note:** The parsing of the step number argument is already done for you in the file `src/charlie.py`.

3.2 Step 2

Run Alice, Bob and the tracker in 3 different terminals:

```
python3 src/alice.py
python3 src/bob.py
python3 src/tracker.py
```

Then, run Charlie:

```
python3 src/charlie.py 2
```

3.3 Step 3

Run Alice, Bob and the tracker in 3 different terminals:

```
python3 src/alice.py
python3 src/bob.py
python3 src/tracker.py
```

Then, run Charlie:

```
python3 src/charlie.py 3
```

4 Report Content

You report will contain:

- Functional description of your solution.
- 1 sequence diagram for each step.
- Difficulties encountered and how you solved them.
- Configuration of every host, *i.e.*, for every *socket*, include:
  - IP address and port number (if defined by you)
  - UDP or TCP
  - Mode: *client* or *server*

5 Submission

You will create a ZIP[3] containing your source code and your report. Make sure to include everything and that your code is working!

You will send the created ZIP to the email address: `jerepage@ulb.ac.be` with the following subject:

```
[ELEC-H-417] FIRSTNAME1 LASTNAME1 - FIRSTNAME2 LASTNAME2 - FIRSTNAME3 LASTNAME3
```

[3] `https://en.wikipedia.org/wiki/Zip_(file_format)`

A Project Structure

You will receive[4] this basic structure:

```
elec-h-417-project
├── chunks
│   ├── alice
│   ├── bob
│   └── charlie
├── config
│   └── peers.ini
├── files
├── src
│   ├── lib
│   ├── alice.py
│   ├── bob.py
│   ├── charlie.py
│   ├── check_file.py
│   ├── create_chunks.py
│   ├── merge_chunks.py
│   └── tracker.py
├── MESSAGES.md
├── README.md
└── statement.pdf
```

You will only modify those files:

- `config/peers.ini` : configure the IP addresses and port numbers of the entities.
- `src/alice.py` , `src/bob.py` : provide the chunks of the file.
- `src/charlie.py` : download the chunks of the file.
- `src/tracker.py` : provide the list of chunks and the peers for each chunk.

If you need additional files (highly recommended to split your code!), you will add them to the `src/lib/` directory and import them.

Note:

- `MESSAGES.md` [a] describes the protocol messages format as in Section D.
- `README.md` [b] explains how to use the scripts and how to run the different steps.
- `statement.pdf` is this document.

[a] `https://github.com/jpagex/elec-h-417-project/blob/master/MESSAGES.md`
[b] `https://github.com/jpagex/elec-h-417-project/blob/master/README.md`

[4] Available online at `https://github.com/jpagex/elec-h-417-project`.

B Configuration Files

There is 2 different configuration files:

- `config/peers.ini` : Define the IP addresses and port numbers of Alice, Bob and the tracker. You can edit this file to change these values.
- `config/file.ini` : This file list the chunks in the order with their hash and the peers (Alice or Bob) that have the chunk. You do not need to edit this file, it is auto-generated with the script `src/create_chunks.py` .

The default `config/peers.ini` file is:

```
[tracker]
ip_address = 127.0.0.1
port_number = 8000

[alice]
ip_address = 127.0.0.1
port_number = 8001

[bob]
ip_address = 127.0.0.1
port_number = 8002
```

Here is an example of a `config/file.ini` file:

```
[description]
filename = test.txt
chunks_count = 3

[chunks]
0 = 7b1c9c5713f25e3ff8f1d84b6dfa01045f67fcc7
1 = 9473c0216c709b1d24e8089626900a0343cdee36
2 = bd0be5d2b74e26ec4430c23c806dfd646e610cbb

[chunks_peers]
0 = alice
1 = alice, bob
2 = bob
```

C Useful Scripts

3 Python scripts have been made for you:

- `scr/create_chunks.py` : Create chunks of 512 kB from a file inside the `files/` directory, and create the configuration file `config/file.ini` . The first third of chunks will be added to Alice, the second third to Alice **and** Bob, and the last third to Bob.
- `scr/merge_chunks.py` : Merge the chunks from `chunks/charlie/` directory into a single file.

- `scr/check_file.py`: Check if the file obtained from the chunks is identical to the one provided in `files/`.

To use them, run these commands:

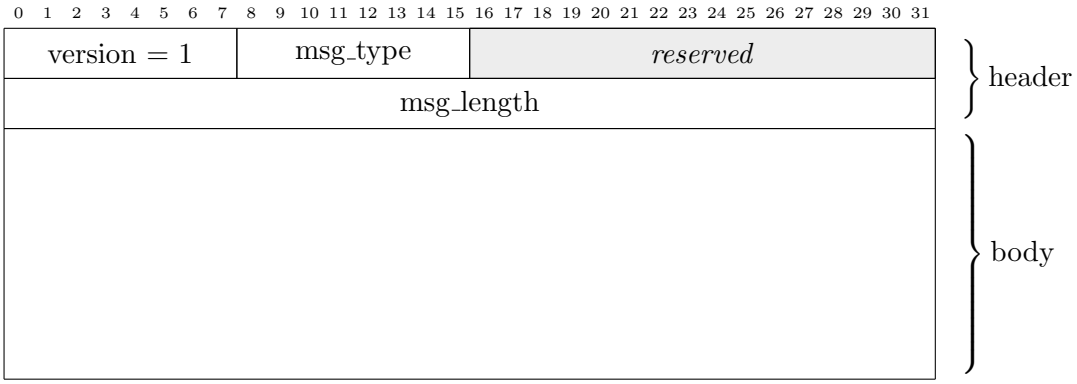
```
python3 src/create_chunks.py [filename]
python3 src/merge_chunks.py
python3 src/check_file.py
```

where `filename` is the filename of the file in the `files/` directory.

> **Note:** to create a file of a specific size with random data, use this command:
> `head -c [bytes_count] < /dev/urandom > files/[filename]`

D Message Format

The general format of a packet is comprised of a header of 8-bytes length and a body of variable length.



- `version` must be 1.
- `msg_type` must be between 0 and 6 (included) depending on the body type.
- `reserved` are reserved bits for future use and must be ignored.
- `msg_length` is the total length of the message in *dwords*[5] (*e.g.*, if the message has a body of 12 bytes, the `msg_length` will be $(8 + 12)/4 = 5$). It means that the message length in bytes must be a multiple of 4.

The possible values for `msg_type` are:

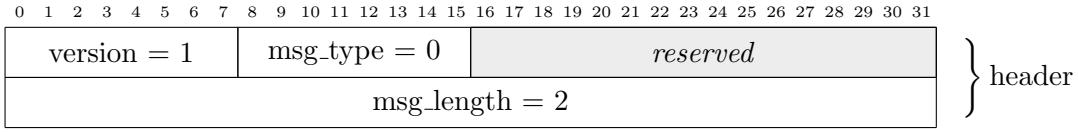
- 0 = `DISCOVER_TRACKER`
- 1 = `TRACKER_INFO`
- 2 = `GET_FILE_INFO`

[5] 1 *dword* = 4 bytes

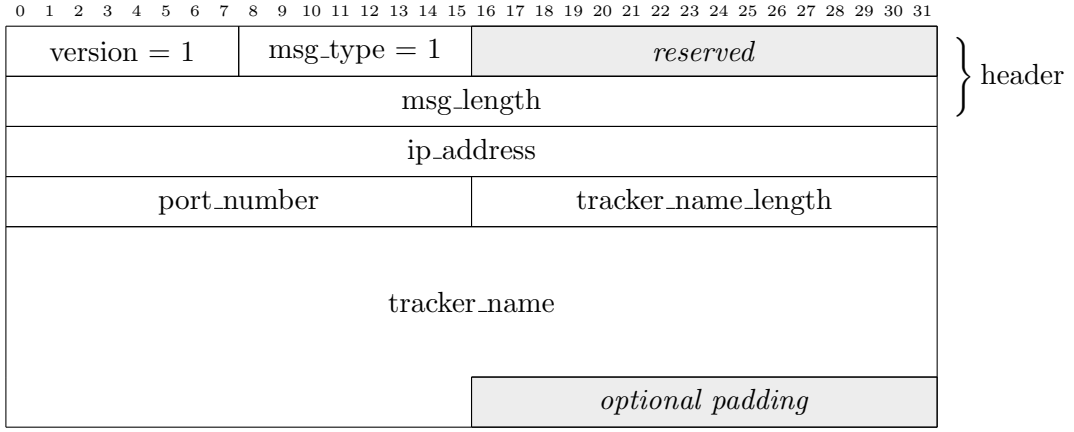
- 3 = `FILE_INFO`
- 4 = `GET_CHUNK`
- 5 = `CHUNK`
- 6 = `ERROR`

Note: The gray boxes are either *reserved* or *(optional) padding*. Their value do not matter but the space they are filling is important.

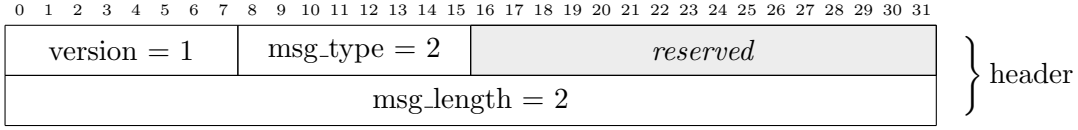
D.1 DISCOVER_TRACKER



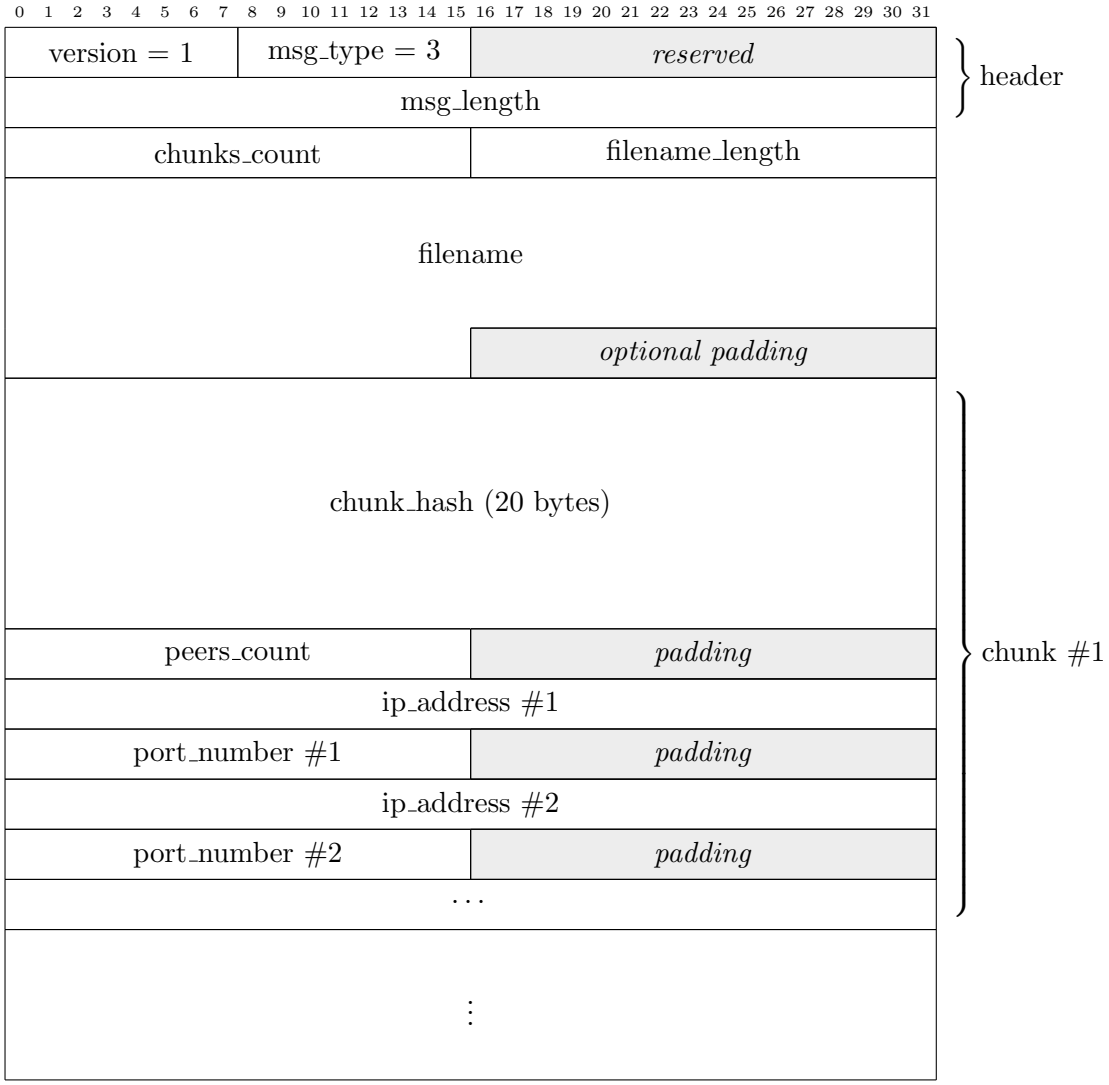
D.2 TRACKER_INFO



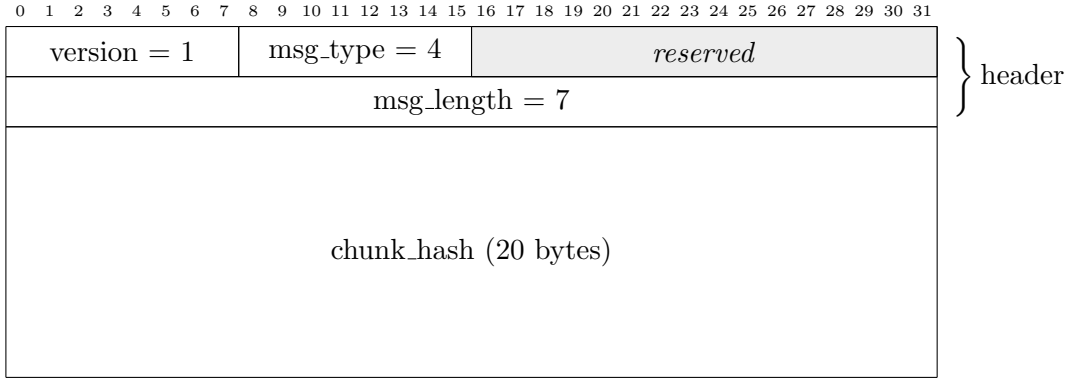
D.3 GET_FILE_INFO



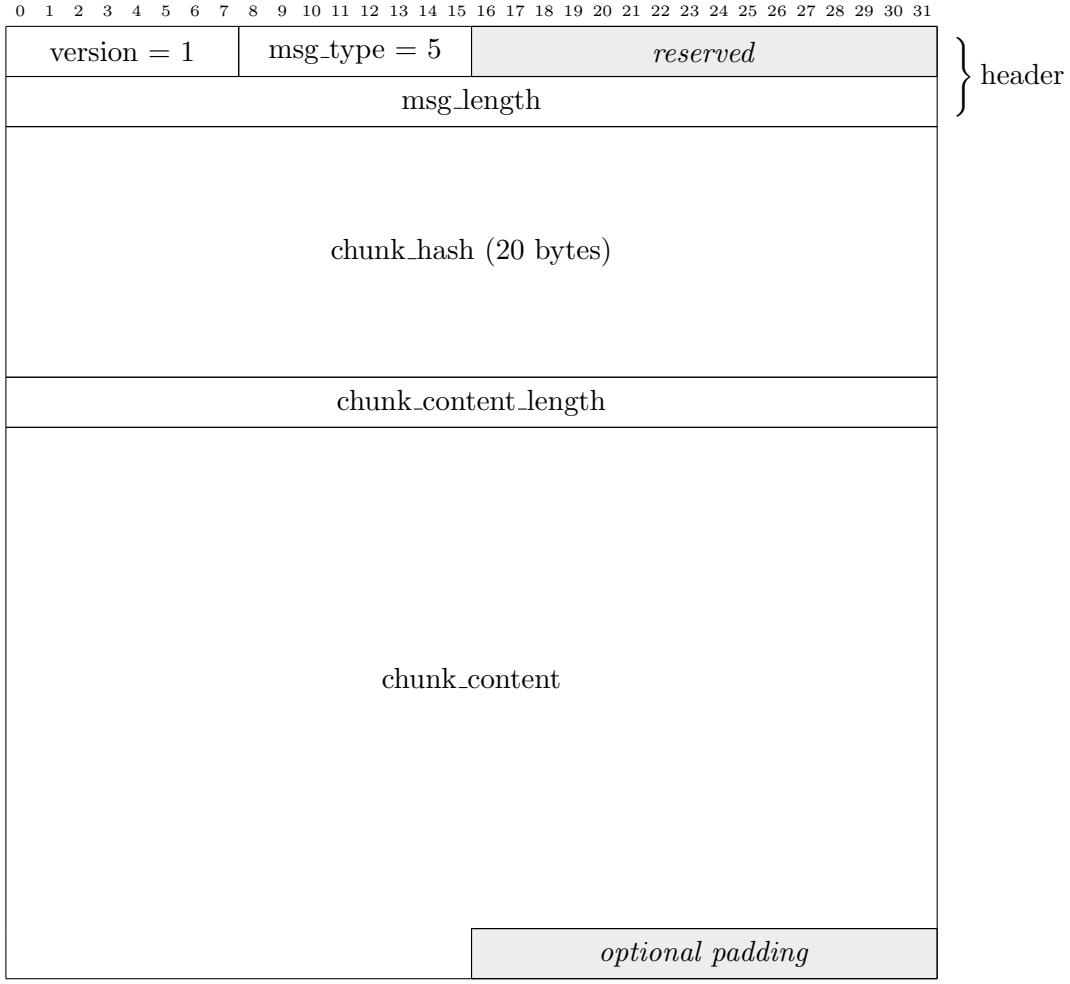
D.4 FILE_INFO



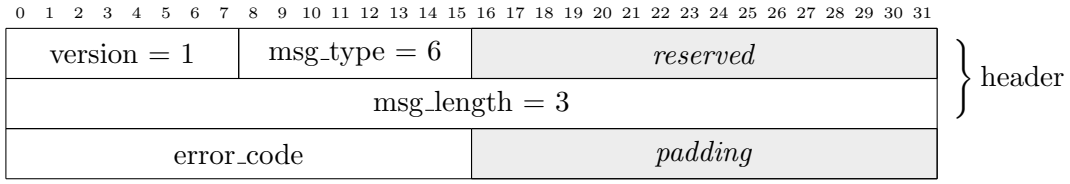
D.5 GET_CHUNK



D.6 CHUNK



D.7 ERROR



The error codes are:

- 0 = `INVALID_MESSAGE_FORMAT`
- 1 = `INVALID_REQUEST`
- 2 = `CHUNK_NOT_FOUND`

E Python Help

Here are some hints to help you with Python.

E.1 Read Configuration Files

You will use the `configparser`[6] library. You can look at examples online (see the link) and at the scripts `src/merge_chunks.py` and `src/check_file.py` which are using this library.

E.2 Create a Simple Client and Server

You will use the `socket`[7] library and the `threading`[8] library. Here is a basic server implementation which accepts any clients, reads their messages and sends back the message in reverse:

```
import socket
import threading

sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
sock.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
sock.bind(("127.0.0.1", 1337))
sock.listen()

def handle_client(client_sock, client_addr):
  while True:
    msg = client_sock.recv(1024).decode("utf-8")
    if len(msg) == 0:
      client_sock.close()
      break
    print("%s:%d> %s" % (client_addr[0], client_addr[1], msg))
    client_sock.send(msg[::-1].encode("utf-8"))

while True:
  client_sock, client_addr = sock.accept()
  th = threading.Thread(target=handle_client, args=(client_sock,
↪  client_addr))
  th.daemon = True
  th.start()
```

Here is a basic client implementation which reads from the input and sends your messages to the server:

[6] https://docs.python.org/3/library/configparser.html
[7] https://docs.python.org/3/library/socket.html
[8] https://docs.python.org/3/library/threading.html

```python
import socket

sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
sock.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
sock.connect(("127.0.0.1", 1337))

while True:
  msg = input("> ")
  sock.send(msg.encode("utf-8"))
  res = sock.recv(1024).decode("utf-8")
  if len(res) == 0:
    sock.close()
    break
  print("Server> %s" % res)
```

E.3 Pack and Unpack your Data in Binary

You will use the `struct`[9] library.

Here is a basic implementation which encodes and decodes a number and a string. The number is encoded in 2 bytes. The string is encoded in 1 byte for its length and 1 byte for each of its bytes (UTF-8).

For the number `42` and the string `Hello World!`, the binary data is:

`0x00 0x2a` `0x0c` `0x48 0x65 0x6c 0x6c 0x6f 0x20 0x57 0x6f 0x72 0x6c 0x64 0x21`

[9] https://docs.python.org/3/library/struct.html

```
import struct

def encode(num, string):
  string = string.encode("utf-8")
  string_length = len(string)
  fmt = "!HB%ds" % string_length
  return struct.pack(fmt, num, string_length, string)

def decode(binary):
  fmt1 = "!HB"
  fmt1_size = struct.calcsize(fmt1)
  num, string_length = struct.unpack(fmt1, binary[:fmt1_size])

  fmt2 = "!%ds" % string_length
  (string,) = struct.unpack(fmt2, binary[fmt1_size:])

  return (num, string.decode("utf-8"))

num = 42
string = "Hello World!"
print("Original:\t", num, string)

binary = encode(num, string)
print("Encoded:\t", " ".join(map(hex, bytes(binary))))

new_num, new_string = decode(binary)
print("Decoded:\t", new_num, new_string)
```

The output is:

```
Original:  42 Hello World!
Encoded:   0x0 0x2a 0xc 0x48 0x65 0x6c 0x6c 0x6f 0x20 0x57 0x6f 0x72 0x6c
↪  0x64 0x21
Decoded:   42 Hello World!
```