Peer-to-Peer Application Project

Authors:
Mikhail Rudakov m.rudakov@innopolis.university
Grigorii Kostarev g.kostarev@innopolis.university
Anna Startseva a.startseva@innopolis.university

Introduction

Peer-to-peer networks are networks where all communication links between peers are in use, opposed to centralized networks, where all clients communicate only with the server. Hence, utilization of peer-to-peer networks may increase speed of data transmission due to usage of additional links. To achieve additional speedup, peer-to-peer networks typically use UDP for communication rather than TCP to avoid connection overheads.

In this work, we implement simple peer-to-peer console application for file sharing among peers in the network. It allows users to distribute a file between peers, acting as a sender peer, or to download a file from current sender peer, acting as a receiver peer. The application uses UDP to send datagrams of the distributed file. The application is implemented in C programming language with the use of Linux threads and tested on Ubuntu operating system. Project can be found in [1].

Application Algorithm Description

Application's flow is divided into two phases. During first phase, peers in the network are receiving information about file to be downloaded (size, number of chunks it is divided into, name of the file), while sender peer is sending this information to the peers. This is done with the use of C sockets programming with TCP to ensure that data is delivered. Once peers received all information about the file, second phase begins. During second phase, clients are trying to get every chunk of the file. It is done via asking all neighbours in the network for missing chunks. The peer does not have any information about what it's neighbor peer has and does not have, so it simply asks everyone. Therefore, sender peer itself does not initiate transfer of the file. Instead, all other peers are asking it to send certain data chunks -

parts of the distributed file. Similar to the first phase, peers communicate via C sockets, but using UDP protocol[3] for faster downloading time. Notice that during second phase, not only sender peer, but other peers can send file chunks on request as well. This shows that additional communication links between peers are in use.

User-defined Data Structures

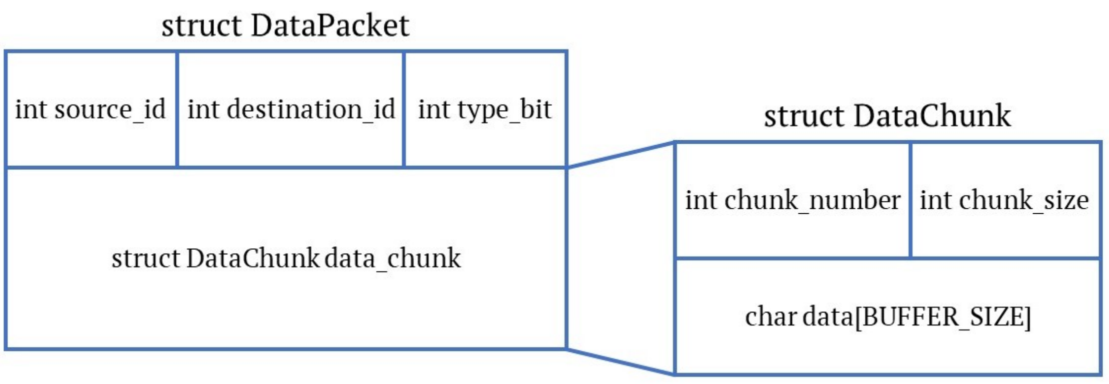


Figure 1. DataPacket and DataChunk structures.

This is structure of DataPacket. Data packets move from one peer to another over network (UDP).
Fields:

- source_id - id of source peer
- destination_id - id of destination peer
- type_bit - bit which is equal to 0 - request for chunk of data, 1 - response with chunk of data
- data_chunk - data chunk itself (filled only if type_bit = 1)

DataChunk is small part of the file to be sent.
Fields:

- chunk_number - serial number of the chunk
- chunk_size - size of this chunk
- data[BUFFER_SIZE] - data in this chunk (BUFFER_SIZE is size of buffer string to be sent via UDP. By default is 512)

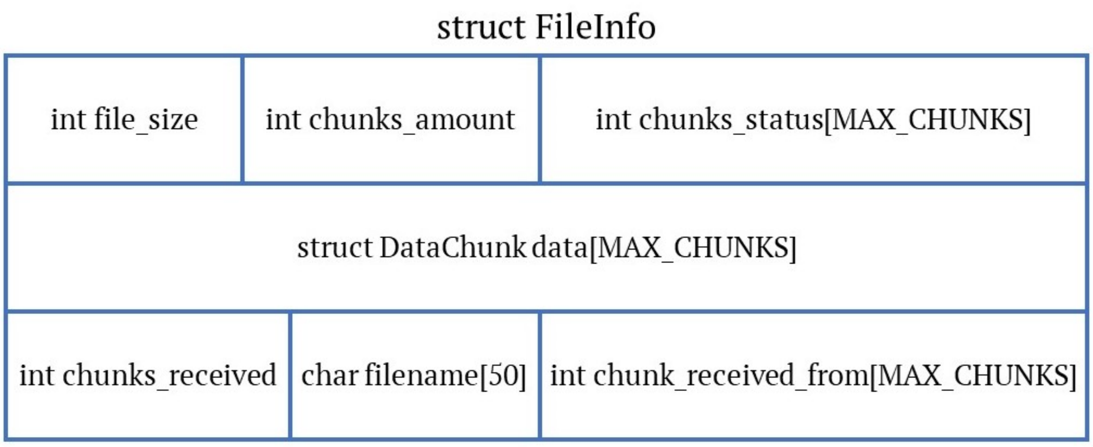

struct FileInfo
int file_size
int chunks_amount
int chunks_status[MAX_CHUNKS]
struct DataChunk data[MAX_CHUNKS]
int chunks_received
char filename[50]
int chunk_received_from[MAX_CHUNKS]

Figure 2. FileInfo structure.

This is structure of FileInfo. It is information about the file - its data and additional log info.
Fields:

- file_size - size of the distributed file (in bytes)
- chuncks_amount - amount of chunks this file is divided into
- chuncks_status[MAX_CHUNCKS] - for each chunk, is this chunk received yet or not (MAX_CHUNCKS is maximum number of chunks in the file. By default is 1 000 000)
- data[MAX_CHUNCKS] - data of the file itself
- chunck_recieved_from[MAX_CHUNCKS] - from which peer did this peer receive this chunk
- chuncks_recieved - how many chunks are received by now
- filename[50] - name of the file (for saving purposes)

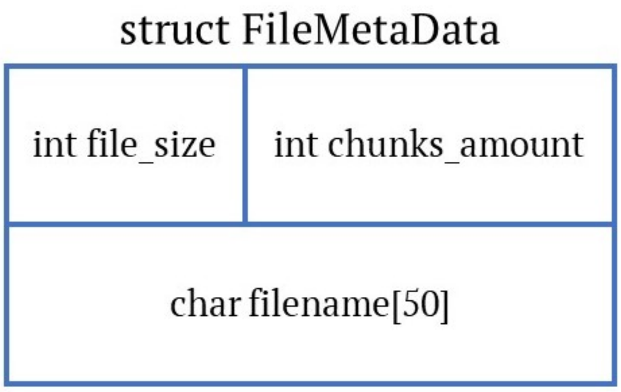


Figure 3. FileMetaData structure.

This is structure of FileMetaData. It is metadata about file to be sent - distributed around peers by sender peer.
Fields:

- file_size - size of the file (in bytes)
- chuncks_amount - amount of chunks this file is divided into
- filename[50] - name of this file (for saving purposes)

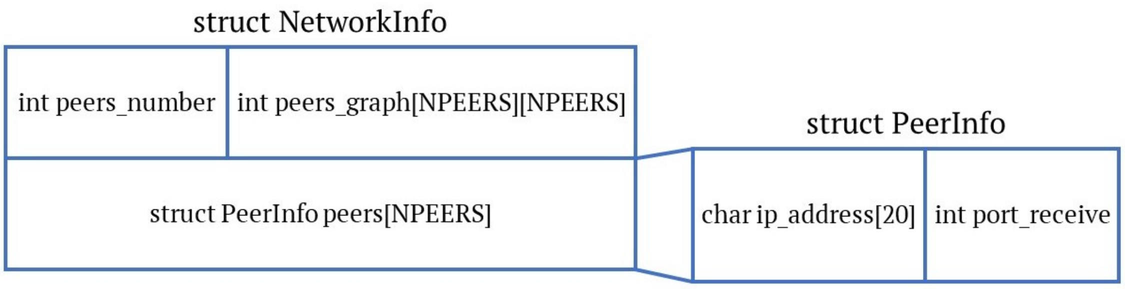


Figure 4. NetworkInfo and PeerInfo structures.

This is structure of NetworkInfo and PeerInfo. NetworkInfo is information about the network - peers and links between them.
Fields:

- peers_number - total number of peers in the network
- peers_graph[NPEERS][NPEERS] - graph of links between peers - peers_graph[i][j] = 1 if link exists, 0 otherwise

- peers[NPEERS] - information about each peer

PeerInfo is network information about a peer.
Fields:

- ip_address[20] - string - IP address
- port_recieve - port to send packets to this peer to

Thread Communication

Application uses linux threads to send and receive packets. Threads are needed to be able to operate with several requests at the same time, not with just the only one. There are 4 threads concurrently running on each peer's application.

First thread is a receiver thread. This thread in an endless loop receives data packets (struct DataPacket) from other peers in the network and stores them to requests queue (implemented using [2]) (incoming_requests), which is to be processed by another thread. While operating with the queue, mutex is locked to ensure that other threads do not change it while receiver works with it. Once request is pushed, mutex is unlocked to let other threads use the queue.

Second thread is response_generator thread. In an endless loop, it takes front message from incoming_requests queue, locking corresponding mutex in advance. Then it processes incoming message obtained. If this is a request from peer and it can provide data chunk, it generates response and puts it into outgoing_requests queue. If this a data chunk peer previously asked for, it then saves this information into struct FileInfo. So, this thread is responsible for data transferring logic between peers.

Third thread is requests_generator thread. While some chunks are missing for this peer, it continuously polls all other peers, as described in algorithm description section. When all chunks are present in peer's FileInfo, it calls log_info() function to save the file and additional logging data. Finally, after calling that function, thread terminates

Fourth (and last) thread is sender thread. It takes messages from outgoing_requests queue and sends them via UDP to specified peers. IP address and port number of peer is obtained by destination_id and NetworkInfo data structure, where all information about peers is stored. This thread also uses mutex lock and unlock on outgoing_requests queue to avoid race conditions.

Thread communication diagram is shown below. Threads are depicted ellipses, and resources that they used are rectangles. Arrows indicate that some thread accesses (read or write) certain resource.

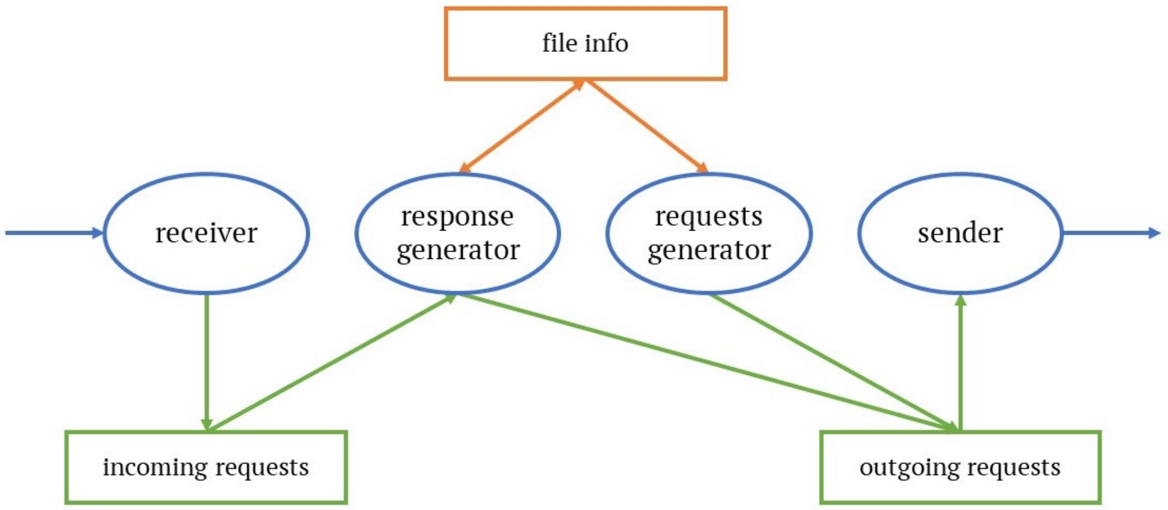
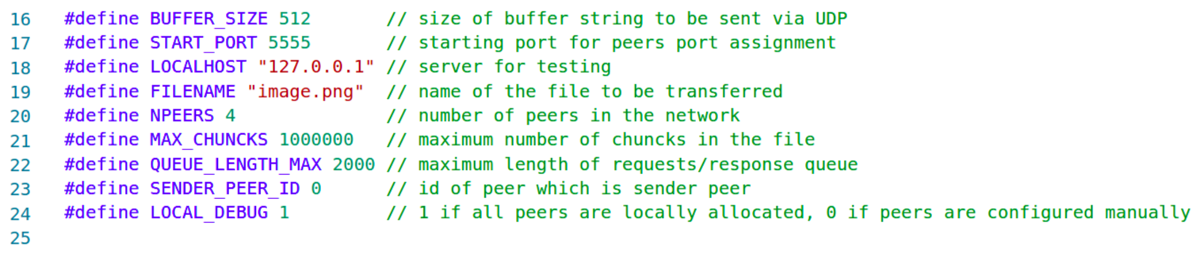


Figure 5. Threads and resources communication in the application.

Application Usage Example

In this section, we will show how to run the application.

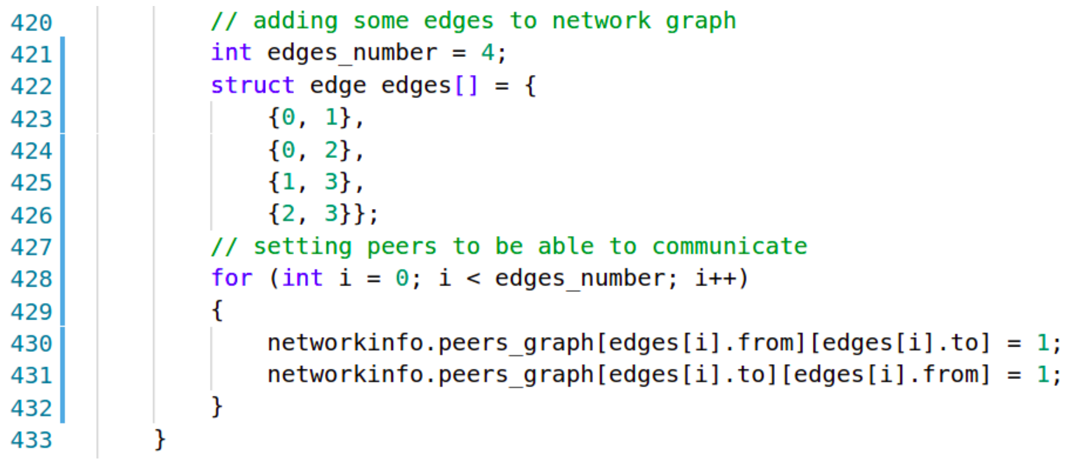
Firstly, in constants definition block, one need to choose mode of application. We will use locally created peers, so define"LOCAL_DEBUG" = 1. We also define that there are only 4 peers (line 20) and file to be transferred is an image (line 19).



```
16  #define BUFFER_SIZE 512       // size of buffer string to be sent via UDP
17  #define START_PORT 5555       // starting port for peers port assignment
18  #define LOCALHOST "127.0.0.1" // server for testing
19  #define FILENAME "image.png"  // name of the file to be transferred
20  #define NPEERS 4              // number of peers in the network
21  #define MAX_CHUNCKS 1000000   // maximum number of chuncks in the file
22  #define QUEUE_LENGTH_MAX 2000 // maximum length of requests/response queue
23  #define SENDER_PEER_ID 0      // id of peer which is sender peer
24  #define LOCAL_DEBUG 1         // 1 if all peers are locally allocated, 0 if peers are configured manually
25
```

Figure 6. Constants definition block.

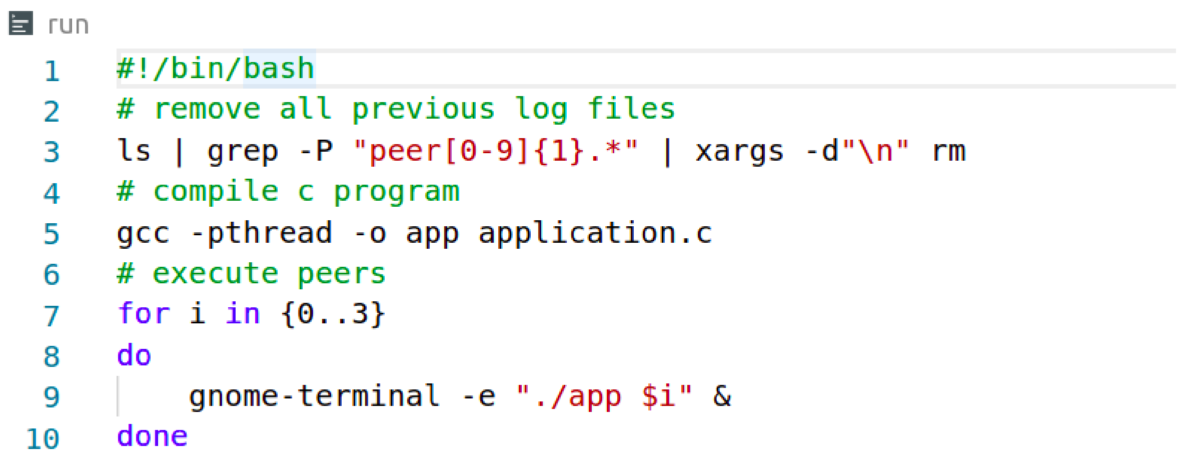
We also simulate absence of some communication links by manipulating with networkinfo.peers_graph.



```c
        // adding some edges to network graph
        int edges_number = 4;
        struct edge edges[] = {
            {0, 1},
            {0, 2},
            {1, 3},
            {2, 3}};
        // setting peers to be able to communicate
        for (int i = 0; i < edges_number; i++)
        {
            networkinfo.peers_graph[edges[i].from][edges[i].to] = 1;
            networkinfo.peers_graph[edges[i].to][edges[i].from] = 1;
        }
    }
```

Figure 7. Peers links graph initialization.

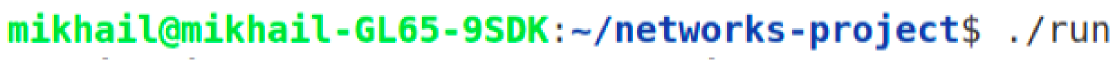
C code of the application is ready to be compiled now. Shell script "run" is created for this purposes. Inside it, one need to set the correct number of peers and just execute the code in command line: "./run".



run

```bash
#!/bin/bash
# remove all previous log files
ls | grep -P "peer[0-9]{1}.*" | xargs -d"\n" rm
# compile c program
gcc -pthread -o app application.c
# execute peers
for i in {0..3}
do
    gnome-terminal -e "./app $i" &
done
```

Figure 8. Script running all peers on local machine at once.

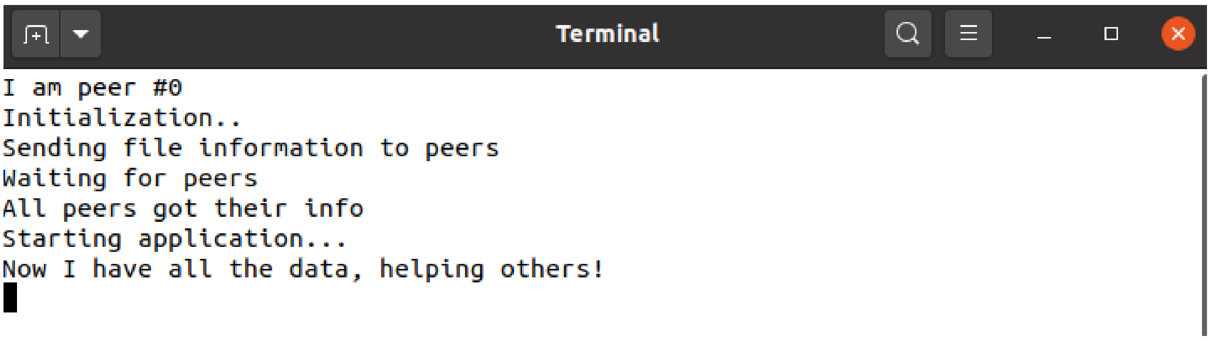


```
mikhail@mikhail-GL65-9SDK:~/networks-project$ ./run
```

Figure 9. Script execution

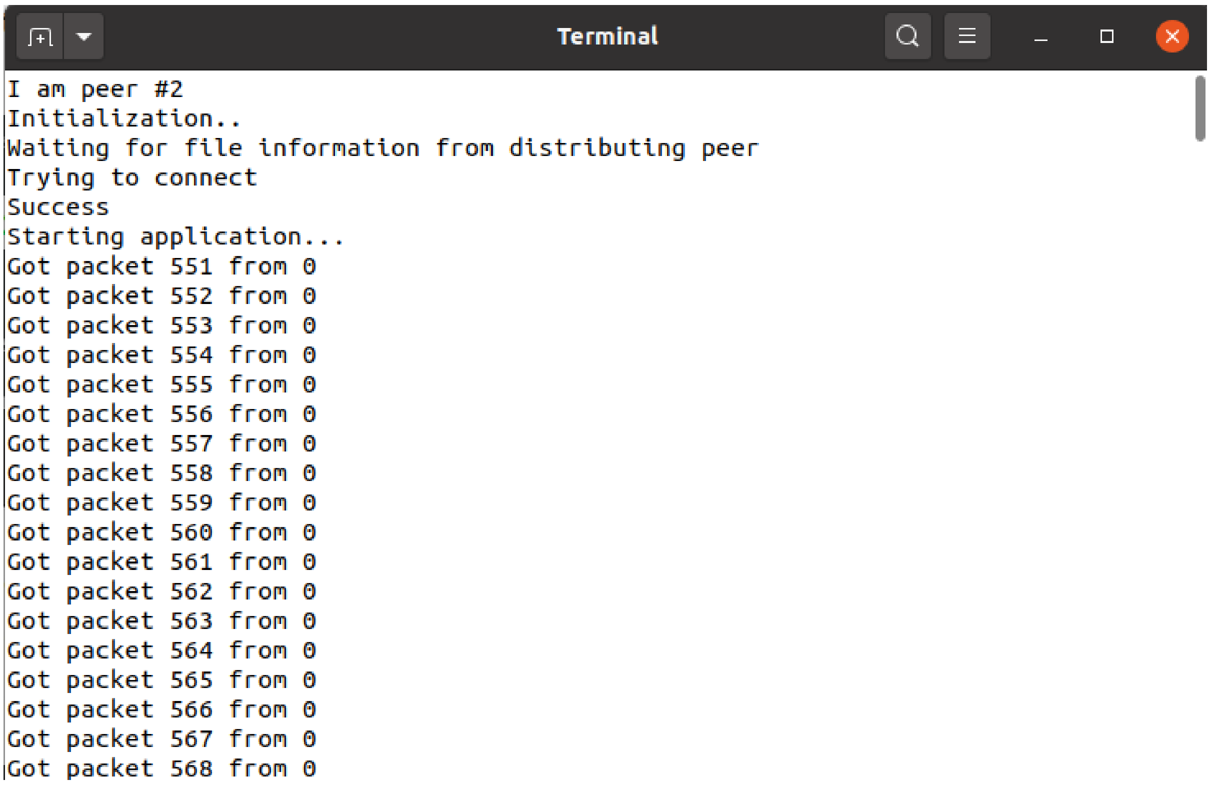
After running command listed on Figure 9, new terminals windows will open, one per peer.

In this terminals, log information about packets flow will be printed.



```
I am peer #0
Initialization..
Sending file information to peers
Waiting for peers
All peers got their info
Starting application...
Now I have all the data, helping others!
```

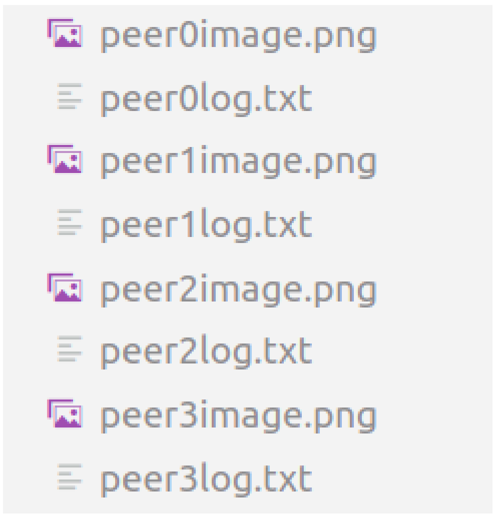
Figure 10. Sender peer console output.



```
I am peer #2
Initialization..
Waiting for file information from distributing peer
Trying to connect
Success
Starting application...
Got packet 551 from 0
Got packet 552 from 0
Got packet 553 from 0
Got packet 554 from 0
Got packet 555 from 0
Got packet 556 from 0
Got packet 557 from 0
Got packet 558 from 0
Got packet 559 from 0
Got packet 560 from 0
Got packet 561 from 0
Got packet 562 from 0
Got packet 563 from 0
Got packet 564 from 0
Got packet 565 from 0
Got packet 566 from 0
Got packet 567 from 0
Got packet 568 from 0
```

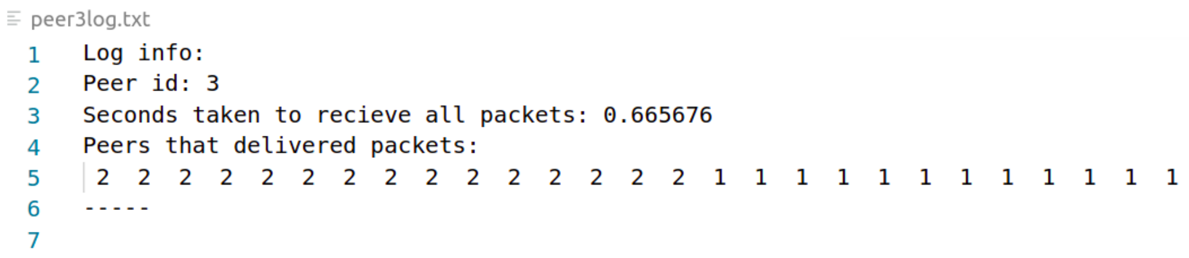
Figure 11. Receiver peer console output.

When every peer indicates that it received all the packets, new files will appear in project folder. Each peer includes its id into filename, as depicted on Figure 12.



```
peer0image.png
peer0log.txt
peer1image.png
peer1log.txt
peer2image.png
peer2log.txt
peer3image.png
peer3log.txt
```

Figure 12. Files generated by peers.



```
peer3log.txt
Log info:
Peer id: 3
Seconds taken to recieve all packets: 0.665676
Peers that delivered packets:
 2  2  2  2  2  2  2  2  2  2  2  2  2  2  2  1  1  1  1  1  1  1  1  1  1  1  1
-----
```

Figure 13. Log information generated by peer.

peer3image.png

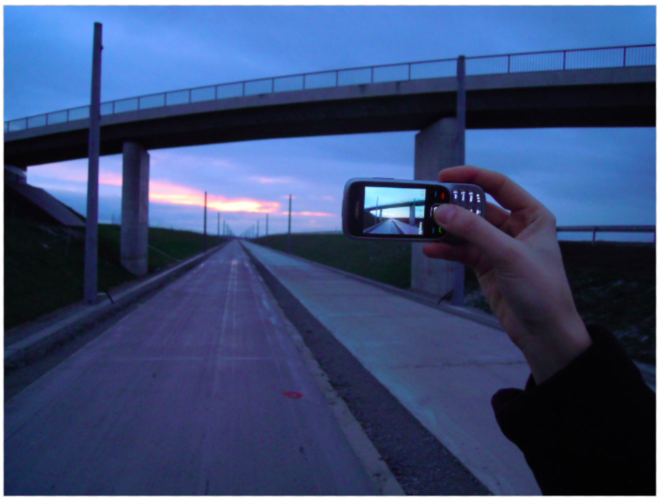


Figure 14. Picture downloaded by peer.

To summarize, we saw how application is run locally, what are the parameters to configure and what is the output of each peer.

Experiments On File Transmission

We conducted some experiments to see how transmission time depends on program parameters.

The first experiment, which results is on Figure 15, we see how average file downloading time depends on number of peers. As expected, as amount of work increases, the time also increases near linearly.

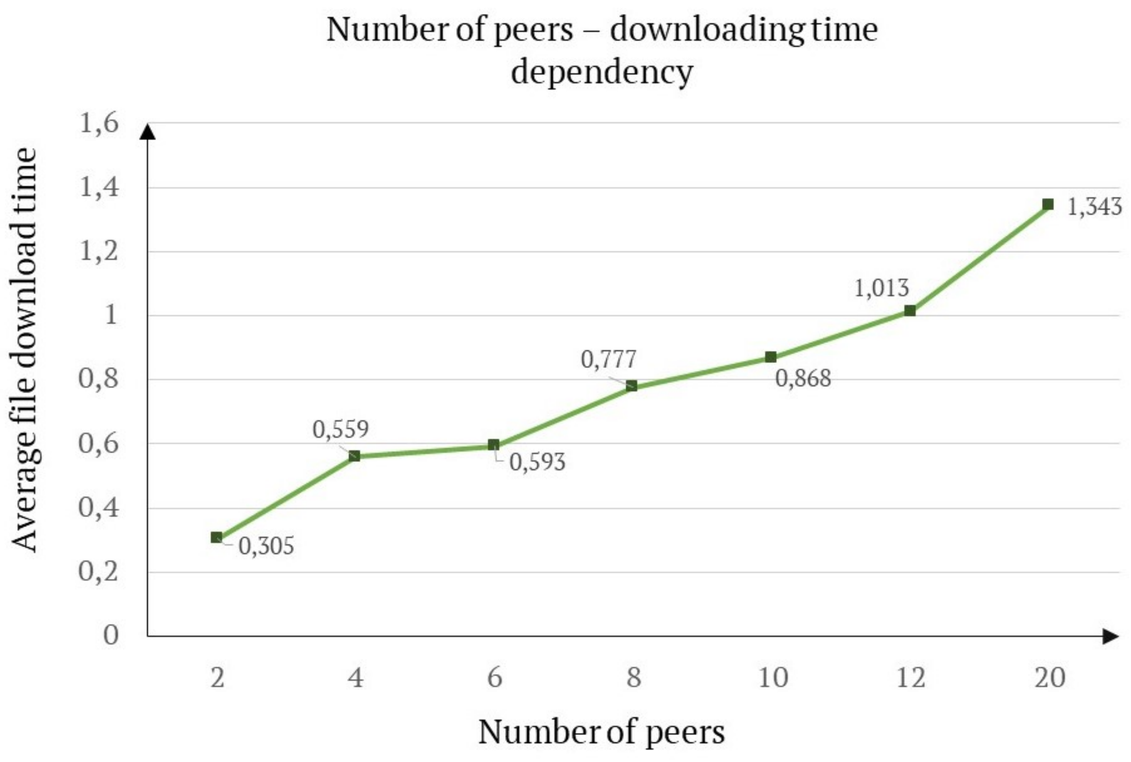


Figure 15. Dependency of download time on number of peers.

The second experiments tests how download times depends on limiting size of requests. Figure 16 shows that size of queue is crucial parameter - if it is too small for current configuration, downloading time increases drastically.

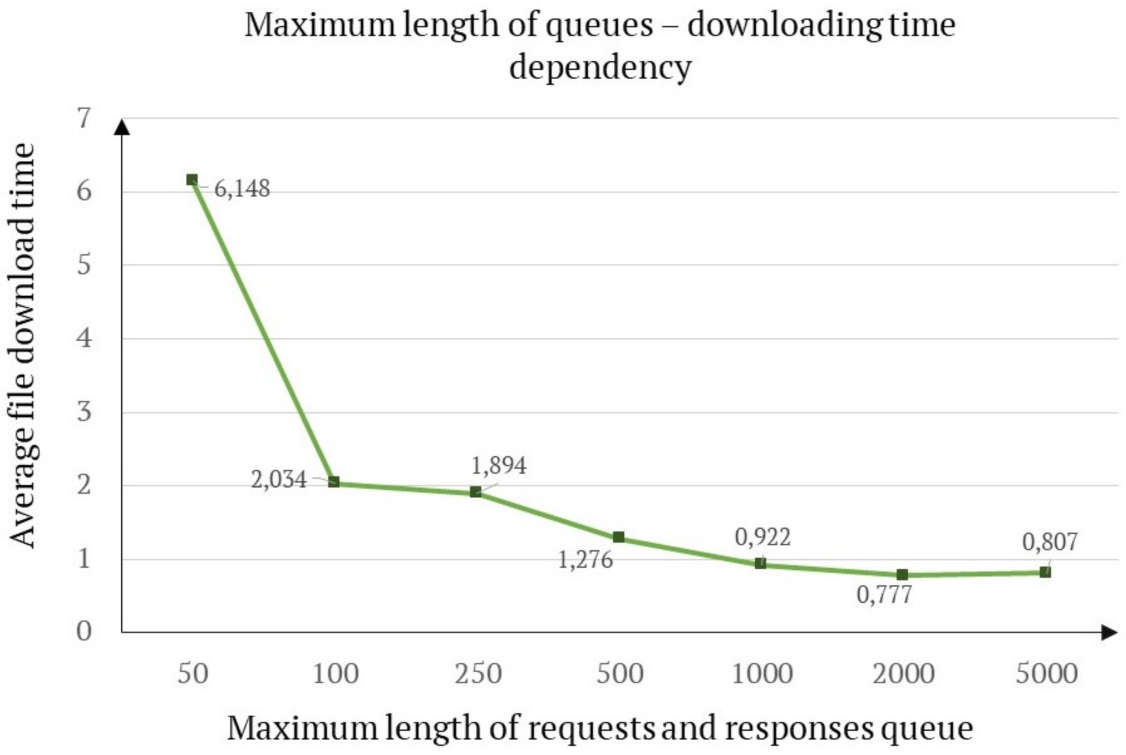


Figure 16. Dependency of download time on maximum length of queues.

Conclusion

In this work, we implement the peer-to-peer application using C sockets. The program works correctly on our tests. We try our algorithm to send some pictures from one host (source) to many other hosts (destination). In all cases, the photo has been delivered entirely. For the optimization process, there are two queues and four threads in the application. One queue stores all incoming requests, another one stores all outgoing requests and all responses. We implemented two threads for getting and sending requests (and responses), the other two for processing requests and generating new ones. We also think that our program can be improved by adding some new features. For example, analysis of bandwidth of links, getting IP addresses dynamically of peers in the network, and adding logical termination of our application, when all peers get the whole file. To sum up, we implemented the peer-to-peer application, which works correctly. However, the program is still imperfect in some aspects.

References

1. https://github.com/Glemhel/networks-project
2. https://man7.org/linux/man-pages/man3/stailq.3.html
3. https://www.geeksforgeeks.org/udp-server-client-implementation-c/