

Peer-to-Peer Application Project

Authors:

Mikhail Rudakov m.rudakov@innopolis.university

Grigorii Kostarev g.kostarev@innopolis.university

Anna Startseva a.startseva@innopolis.university

1. 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 can be found in [1].

2. Overview of Application's Functionality

In this section, we provide high-level application overview.

Application allows users to share a file between peers that are in the same network. Using all communication links between peers, it achieves maximum performance. Figure 1 provides general example of network with peer hosts sharing a file.

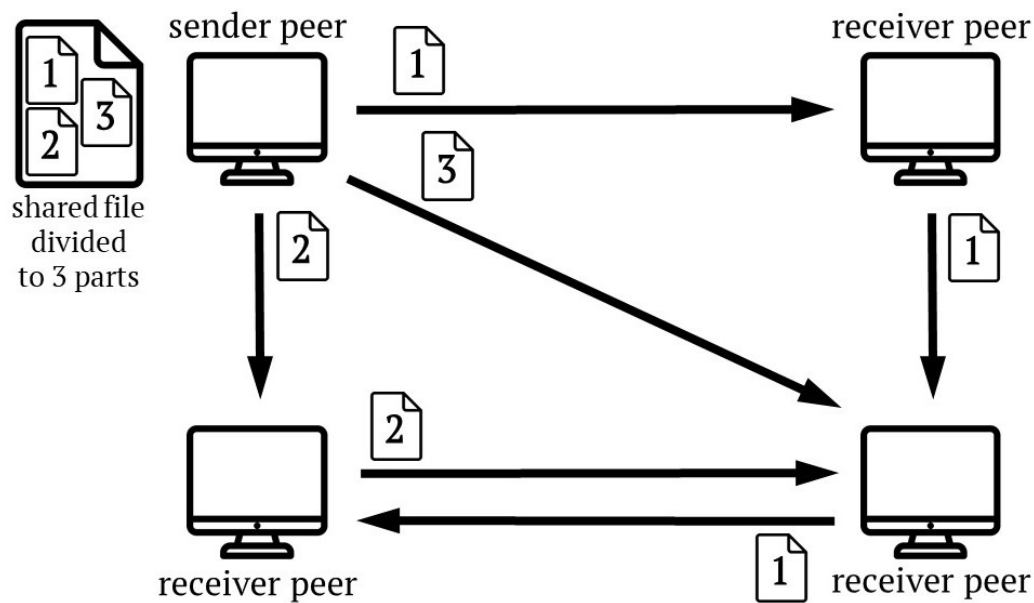


Figure 1. Example of hosts exchanging parts of a shared file in a peer-to-peer network

One peer, which wants to distribute a file, acts like a sender peer. It starts file sharing in the network. File is divided into digestible chunks to send it to the others.

Other peers, acting as a receiver peers, try to receive all chunks of a file, exchanging parts that they possess. To send and receive such parts, peers use custom communication protocol, which will be described in Section 3.

After all, every peer has every part of a file, meaning that file is successfully distributed (Figure 2).

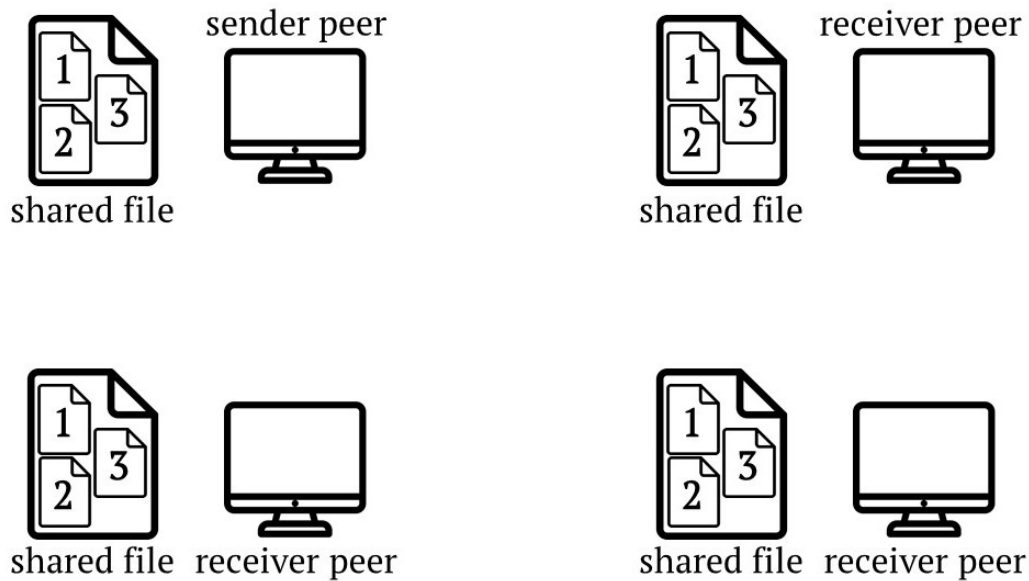


Figure 2. All parts of a file are successfully distributed in a peer-to-peer network

3. Implementation Details

This section provides technical details about peer-to-peer application.

3.1 Technology Stack

To implement the application, we use the following technologies:

- C programming language - general-purpose and powerful
- C socket programming with UDP - exchanging information between hosts. UDP is used to enhance performance [2]
- POSIX Threads - processing information and sending packets concurrently on each peer, described in Section 3.3
- Tailq - C implementation of FIFO queue for queues of packets [3]

3.2 Communication Protocol

We developed a custom communication protocol for file exchange between peers.

Before transmission of any data, each peer need to know size of the file and amount of parts. Such information is sent to each peer by sender peer with the use of FileMetaData packet, which structure is described in Table 1. Each peer get single FileMetaData packet from sender peer, getting ready to receive parts of a file.

Table 1. FileMetaData Structure

<u>Field Name</u>	<u>Datatype</u>	<u>Field Description</u>
<u>file_size</u>	int	Size of distributed file in bytes
<u>chunks_amount</u>	int	Amount of parts file is divided into
<u>filename</u>	char[]	String with name of the shared file

Initially, receiver peers do not have any chunks of a distributed file. These peers start to poll each adjacent peer whether it has a chunk that is missing. If such chunk is already downloaded by that peer, it sends a packet back with desired chunk inside. Otherwise, it does not respond. Figure 3 describes such interaction.

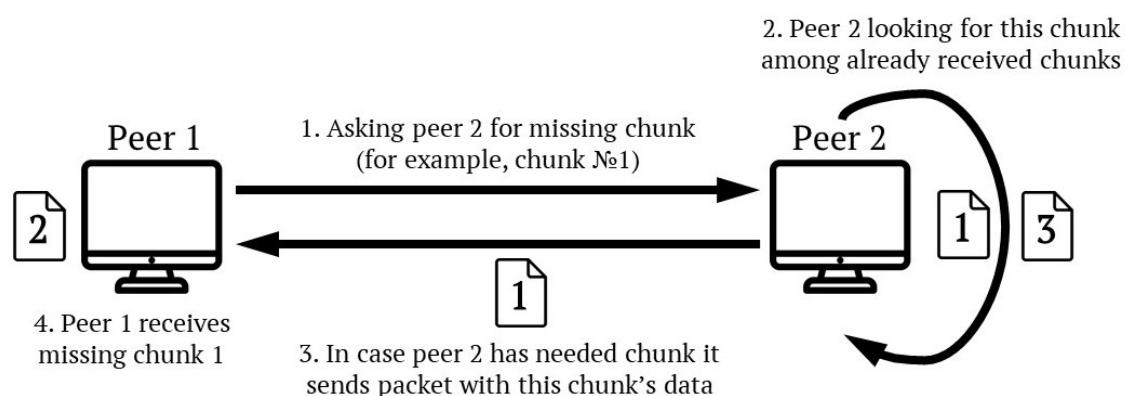





Figure 3. Peers communication when one wants to get some chunk from the other




Every packet that is sent from one peer to another has a certain structure, described in Table 2. Such packets are used either to ask for a missing chunk of a file or to send a packet to peer that asked for the chunk.

Table 2. Packet Structure

 Field Name	 Datatype	 Field Description
<u>source_id</u>	int	Id of the peer message is sent from
<u>destination_id</u>	int	Id of the peer message is sent to
<u>type_bit</u>	int	Type of a packet: type_bit = 0 for chunk request, and type_bit = 1 for response with data of a desired chunk inside
<u>data_chunk</u>	DataChunk	Custom structure with id of chunk requested / transmitted and data for that packet for a response packet

When peer receives a packet with a chunk it does not have, it saves this fragment to a FileInfo structure. This structure contains information about all chunks status (received / not), data of that chunks and additional logging information. More information is provided in Table 3.

Table 3. FileInfo Structure

 Field Name	 Datatype	 Field Description
<u>file_size</u>	int	Size of distributed file in bytes
<u>chunks_amount</u>	int	Amount of parts file is divided into
<u>chunks_received</u>	int	Amount of chunks already received by a peer
<u>chunks_received_from</u>	int[]	For each chunk, Id of peer from which this part was received
<u>chunks_status</u>	int[]	For each chunk, chunks_status[i] = 1 if chunk is already received, 0 otherwise
<u>filename</u>	char[]	String with name of the shared file
<u>data</u>	DataChunk	Custom structure with id of chunk requested / transmitted and data

After a while, all of the file's chunks will be distributed among all the peers, and each peer will have the whole file saved in FileInfo.data field.

3.3 Threads Interaction

Application uses POSIX threads to send and receive packets. Threads are needed to be able to operate with several requests at the same time, sending requests as well concurrently. We also use mutexes to avoid race conditions on certain resources.

There are 4 threads concurrently running on each peer's application. Sample threads interaction is described on Figure 4.

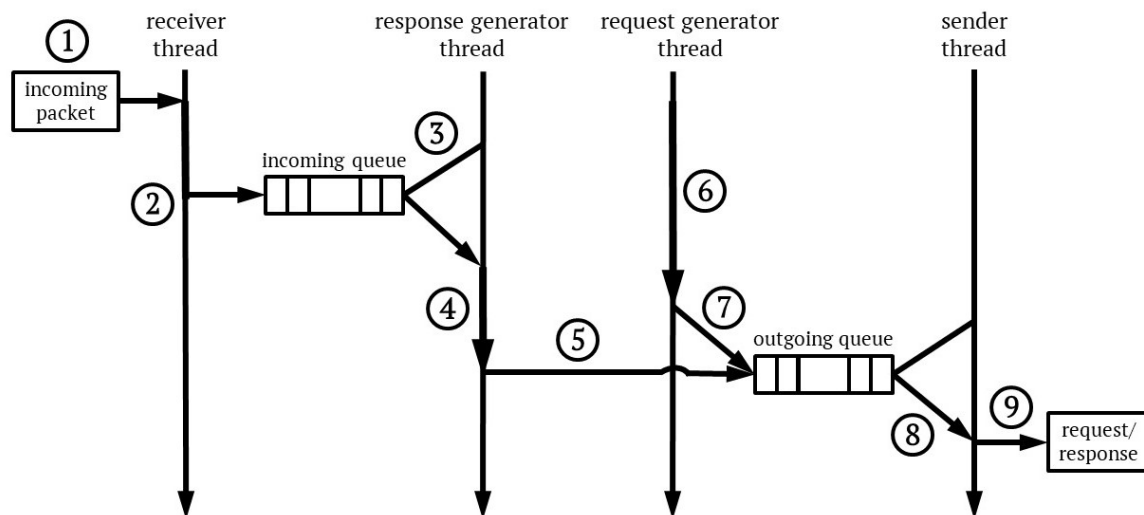


Figure 4. Example of threads interaction step-by-step.

1. Packet comes to the peer from some other host
2. Peer pushes received packet to the queue of incoming requests
3. Response generator thread gets front element of incoming requests queue
4. Request is processed: lookup for a chunk or saving of a chunk to FileInfo
5. If step 5 generated some answer, push it to the outgoing requests queue
6. Requests generator thread produces packets to ask other peers about
7. Packets with requests for some chunks are push to outgoing queue
8. Sender thread gets front element of outgoing queue
9. Packet is sent to the respective destination - some other peer

Overview of each thread's functionality:

- Receiver thread - listens for incoming UDP packets and store them to incoming packets queue
- Response generator thread - process packets from incoming queue and generates answer, if possible, storing results to the outgoing requests queue
- Request generator thread - generates packets to ask peers for chunks that are missing
- Sender thread - sends packets from outgoing queue to their destination




4. Details on application configuration

On Figure 5, one can see block with application configurations from application's source code. Table 4 provides information about each variable definition.

```
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_CHUNKS 1000000 // maximum number of chunks 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 5. Constants definition block

Table 4. Definitions Description

 Variable Name	 Datatype	 Variable Description
<u>BUFFER_SIZE</u>	int	Amount of bytes in a chunk - fragments initial file is divided into
<u>START_PORT</u>	int	Port number to start numeration for locally run peers - ports to communicate with other peers
<u>LOCALHOST</u>	char[]	Ip address of localhost
<u>FILENAME</u>	char[]	Name of the file to be distributed - information for sender peer
<u>NPEERS</u>	int	Amount of peers in the network where file is shared
<u>MAX_CHUNKS</u>	int	Maximum number of chunks file could be divided into - upper bound to avoid memory overflow
<u>QUEUE_LENGTH_MAX</u>	int	Maximum length of packets queue. Limit is set to control flow - prevent packets from filling up the peer
<u>SENDER_PEER_ID</u>	int	Id of peer that is considered to be sender - could be any peer number
<u>LOCAL_DEBUG</u>	int	Configuration of running mode. LOCAL_DEBUG = 1 - all peers are locally created on different ports, and file is shared among them. LOCAL_DEBUG = 0 - peers are real computers in the specified network (Ip addresses and ports are to be provided by the user)

5. Performance evaluation

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

The first experiment, which results is on Figure 6, 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.

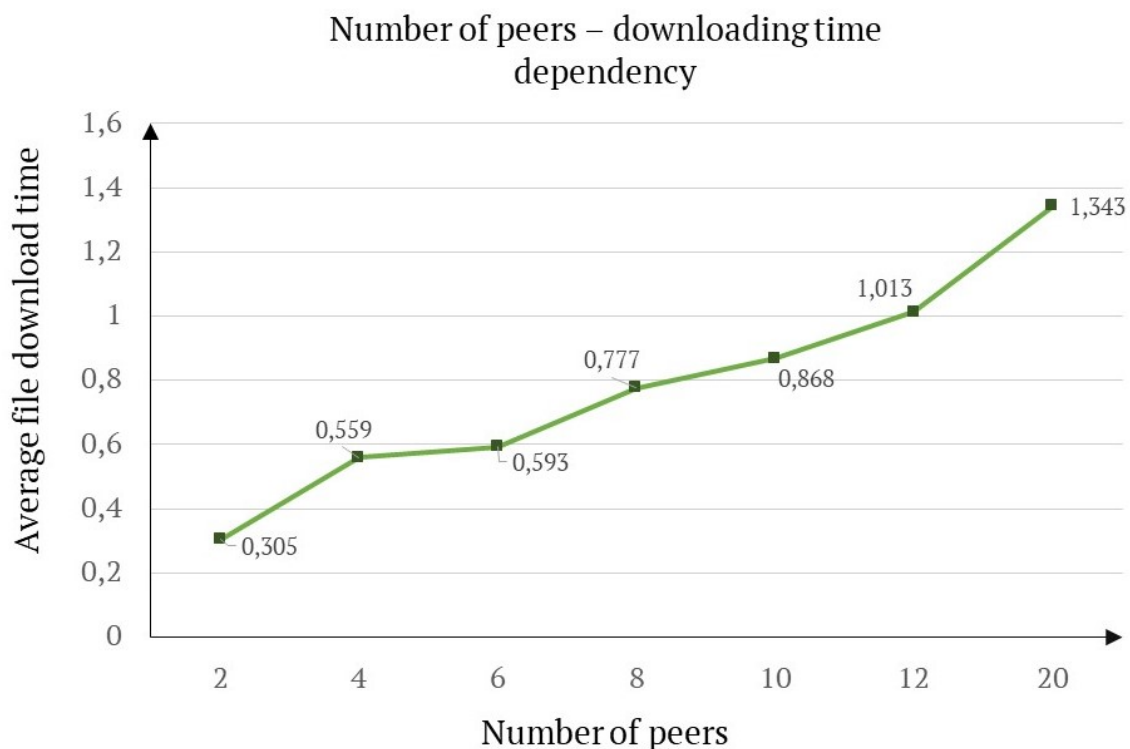


Figure 6. Dependency of download time on number of peers.

The second experiments tests how download times depends on limiting size of requests. Figure 7 shows that size of queue is crucial parameter - if it is too small for current configuration, downloading time increases drastically.

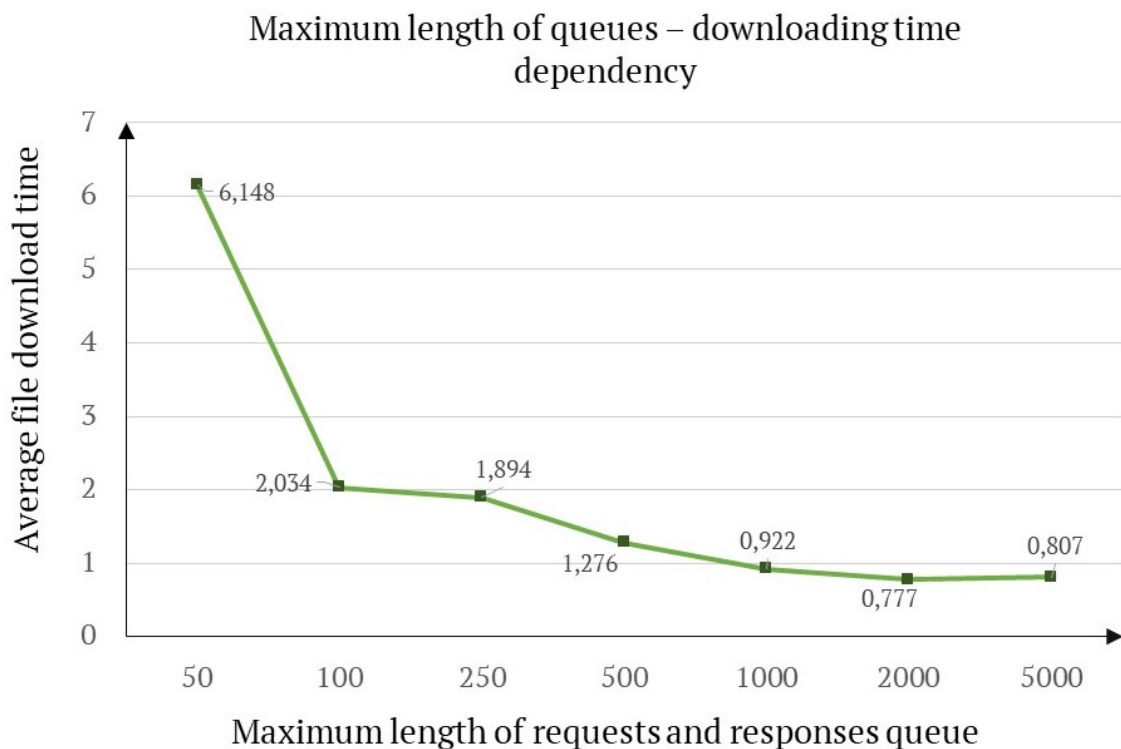


Figure 7. Dependency of download time on maximum length of queues.

6. Conclusion and Future Implications

In this work, we implemented peer-to-peer application for file sharing using C programming language. Application successfully distributes file among peers network, utilizing all available communication links between peers.

We also think that our program can be improved by adding some new features. For example, analysis of bandwidth of links to utilize them efficiently, getting IP addresses of peers in the network dynamically, and adding logical termination and command line interaction of our application, when all peers get the whole file.

References

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