

CSIS0234B Computer and Communication Networks (Class B)

Tutorial 3

File transfer using UDP

In this tutorial, we modify a UDP server and a UDP client to perform file transfer. UDP is a connectionless and unreliable service, so unreliable that if you are not careful, messages can be lost even in a reliable network. We will see tricks to avoid such problems.

Two programs, `Server.cc` and `Client.cc`, are stored in the computer. They make a UDP socket (`bind()`'ed to port 12345 in the server), and then call `dg_ftp_serv()` and `dg_ftp_cli()` respectively. Your task is to write these functions to perform file transfer. The client needs a command-line argument specifying where the server is, while the server needs one to specify what file to serve. The client should write the received file to standard output, to be redirected to a file using shell output redirection (`>` or `>|`).

Your tasks

1. Write the `dg_ftp_cli()` and `dg_ftp_serv()` functions. In the client, `dg_ftp_cli()` starts the transfer by sending a 0-byte message to the server (if you use a connected UDP socket, you need to use `send()` instead of `write()` for this to work). In the server, `dg_ftp_serv()` replies with the content of the file. Use a buffer of 128KiB to read and receive the file. Use the script `gen-random-file` to generate a 1KiB random file to test your programs. Once the client completes, use `ls -l` to see whether the file is of the correct size.
2. Generate a file with 112000 bytes, and transfer the file with your programs. It fails: the maximum message size is exceeded. Modify your programs so that the server sends multiple UDP packets, each of only 1400 bytes. A message of less than 1400 bytes notifies the end of file (if the file size is a multiple of 1400, the server sends a 0-byte message at the end). Check that your programs can start transferring the file, although the file is incomplete. Some messages are lost because the client buffer is not sufficient to hold the file. Compare the result when the file size is 111999. Explain the difference.
3. Implement a simple “flow-control” mechanism to avoid overrunning the client buffer: every time the server sends one message, it waits for the client’s “acknowledgement” before continuing. The acknowledgement is a 0-byte message, which is sent by the client whenever it receives a message. Check that the file is transferred completely.
4. The program fails when there are multiple clients. If time allows, change the server to a concurrent server to correct the problem, and use `connect()` in the server to make sure client won’t interfere. You’ll also have to open the file in each child process separately to avoid them to interfere. Test the server using a large file (e.g., of 5MiB) so that you have enough time to create another client when one file transfer is in progress.

Note

The loopback interface (and the Ethernet network) used in this tutorial is reliable, i.e., frames sent are always received, in correct order, unless buffer overruns. But if we run the server in `virtue` and connect to it from our lab, the programs fail. This is because the inter-network, consisting of multiple Ethernet networks, is unreliable: frames may be lost even if we send slowly. Better scheme must be used to correct the problem. Alternatively, one might simply use TCP.