PROXY

This programming problem is a test of your network programming skills. You are free to choose any appropriate programming language to implement your solution. However, do try to use only standard APIs and libraries. Feel free to explain or discuss your design choices/assumptions as comments in your code.

Problem

Write a server-client program that achieves the following: server listens on a port; client makes a TCP connection and server sends a file to the client. In addition to acting as a server, the server can also act like a proxy.

Your solution should have two parts: a server program and a client program. The server program is run with a port number as an input. It listens at the specified port for incoming connections and can function as either a proxy or a file server depending on the request made by the client.

The client program is run with an input file in the following format:

```
filename
proxy1-ip-address/hostname port
proxy2-ip-address/hostname port
proxy3-ip-address/hostname port
...
proxyn-ip-address/hostname port
server-ip-address/hostname port
```

Example

The following is a sample input file that is read by the client program:

```
myfile
123.42.12.123 100
clementi.comp.nus.edu.sg 1000
12.23.156.1 99
```

What the client will do is to connect to `123.42.12.123` at port 100, `123.42.12.123` will act as a proxy and connect to `clementi.comp.nus.edu.sg` at port 1000, which in turn acts as a proxy to connect to `12.23.156.1` at port 99. `12.23.156.1` then returns the contents of file `myfile` to `clementi.comp.nus.edu.sg` which forwards it to `123.42.12.123`, which in turn forwards it to the client. The client then saves the file locally to disk with the name "`myfile`".

Requirements

1. The file must be streamed *continuously* from the server through the proxies. For example, it is not acceptable for `clementi.comp.nus.edu.sg` to forward the file to `123.42.12.123` only after it has fully downloaded the file from

`12.23.156.1`. Instead, as blocks are received from `12.23.156.1` they would be forwarded to `123.42.12.123`.

2. The server must support *concurrent connections*. Multiple clients may send requests at the same time and each server must be able to handle multiple requests *in parallel*. In other words, for each server, before it finishes processing current request, it should be able to accept *and* process new request. As an example, one server may be streaming two files to two different locations simultaneously, while at the same time relaying request to another server. Note that it is not acceptable for each server to process incoming requests in sequence, which is inefficient.

   You may assume that the number of concurrent connections to a server will not exceed 4. But a good implementation should work without such assumption.

3. The server must be able to handle requests as long as it is still running. In other words, it should continue functioning even after serving the first request.

4. The server and client must be able to handle *any* type of file, regardless of whether it is binary or text.

5. Your code *must compile* and the programs should execute *without crashing or stalling*.