CIS3210 — Computer Networks

Assignment 2

Due October 23^{rd} 2009 in/before midnight

You will create a client that will work with the server described below.

There is a TCP server that handles requests for documents. The server will accept a connection and serve one request; then it will close the connection is closed or it will timeout after 90 seconds.

In order to speed up the delivery of documents, the server will open $\mathcal{N}$ parallel connections to the client and send the document in chunks through all $\mathcal{N}$ sockets. One of the objectives is to see how the real–time delay depends on $\mathcal{N}$.

The application protocol

After your TCP client connects to the server, it sends one request to the server with the following contents:

Request⟶	N	p	document name
Length in bytes⟶	2B	2B	*variable*

Where **N** is the number of circuits that the client wants to use and **p** is the port number that the server is to connect to N times. The second line (not part of the request) gives the size of the fields in bytes. Note that only the last field is in **ASCII**; the other two are binary numbers in network byte ordering.

The server attempts to connect to the port N times, getting N parallel connections. If unsuccessful, it closes the main connection without any response.

If the server successfully creates N connections, it divides the desired document into N roughly equal parts and starts sending these parts in chunks over the N connections (each part is permanently associated with one connection).

The first chunk of each part starts with a 4 byte value: the offset of this part in the document, expressed as a binary integer in network byte ordering.

Example

1. The server receives from **IP:1234** a message:

03BBhttp://www.getrichfast.ca

where the first two bytes ("3") state that 3 connections are to be opened, followed by another 16–bit binary number in network byte ordering (if BB = $\boxed{12}\boxed{138}$ then it is port number 3210). That is followed by the name of the desired document.

2. The server decodes the port number: $12 \times 256 + 138$.
3. The server connects to **IP:3210** three times, each time using a different socket. If not all connections succeed, the server closes its connection with **IP:1234** and all the connections to **IP:3210** established so far.
4. All connections succeeded so the server now checks if the desired document is available. If not, the server closes the connection.
5. The server divides the document into 3 parts and starts sending it. This document has 49.4 kB, so the server chooses to send bytes 0–24000 to one connection, bytes 24001–48000 to another and bytes 48001–end to third one.
6. And so it does remembering to insert 4 bytes before the first byte of each part.
7. The server closes the connections.

The documents that can be requested

To reduce Internet traffic, you can ask only for one of a number of filenames that are stored on the same machine where the server is. At this point, the documents are *kakuro1.pdf, kakuro2.pdf, kakuro3.pdf, kakuro4.pdf*. This list will change before the assignment is due.

No escape through multi–threading

The server and the client must be single scheduling units, without any **threads** nor **sub–processes**. Threading or forking is not allowed, neither in your code nor in the code of any libraries you reference.

Your task

Develop your client in steps (each step adds code to the previous one):

1. Create a client that **accepts** 1 connection.

2. Upgrade your client to receive a document correctly. Make sure to find a way to display it.
3. Upgrade your client to survive disappearing connections.
4. Ask for 3 connections and assemble the document from input you get from all 3.
5. Be bold! Try 6 connections. Put a loop around your code so that it repeats these steps "forever":
    (a) Connect to the server asking for a document sent using 6 connections.
    (b) Receive the document.
    (c) Close all the 7 connections.
6. Add a timeout mechanism to handle a lazy server.

Deliverables

Your code will be written in C or C++ (no Java this time). It will use the Socket API function calls directly. All socket data traffic should be done by issuing system calls from the `send/receive` or `read/write` families (no wrappers).

You will submit a printout of your code with a self–evaluation page attached (see appendix).

You will sign up for a demonstration of your code and successfully demonstrate that your self assessment is correct.

Grading

You will receive marks for each step as listed above.

Step	1	2	3	4	5	6
Marks	4	2	2	6	4	2

Assignment 2

Self evaluation form

Name: ____________________ Student ID# __________ Total: [ ]

Step	Done correctly	Not done	Other (explanation)
Code uses only send/recv or read/write		If not, final grade = 0	
1			
2			
3			
4			
5			
6			

Grading

1	2	3	4	5	6
4	2	2	6	4	2