Final Project: Computer Network

Niloofar Safi Samghabadi (ID:1463946)

November 27, 2017

1 Problem Description

The goal of the project is to implement a metaCDN. We use different cloud services to run our servers. There is also a client that uploads a given file on the server, and a HTTP proxy that decides from which server, we should download a requested file. In the following sections, we discuss the implementation details of each part of the project.

2 File Upload on Server

For uploading a file on one of the running servers, we define two different method: Json and Protobuf3. The client (upload.py) reads the input file, convert it to one of these two formats using base64 encoding and send it to the server through a TCP protocol by opening a socket. We can specify the format that we are going to use by an input argument. It adds three fields as metadata to the encoded content of input file to create the packet:

1. The size of the file
2. The name of the file
3. A flag named "distributed" which is initially set to "F" (server uses this flag to decide whether to distribute the uploaded file or not)

The list of server IPs and ports for connecting to the servers on each of service providers hard-coded in the code. The client chooses the server on which it wants to upload the files by an input argument. In the case that the name of input server does not exist in the list it will return an error. When each of the servers receives a file from client, it first decodes the content of the file, and save it (this would be exactly the same file we have on the client side), then changes the flag distributed to "T" and send the same packets as what it received to the other servers which are alive. So, if the mentioned flag is "T", the server does not distribute the file to the other servers. But, the question is that how a particular server knows the other live servers. We implement it by making the servers able to send heartbeat messages to the other servers. The mechanism is that once we run a particular server, it has the list of other servers in Json format (containing their IPs and ports). Each server can update its own list when it receives a heartbeat message from other server. The heartbeat message contains the port and address of each server. Therefore, for example, whenever server "A" does not receive a heartbeat message from server "B", it knows that server "B" is dead, and remove it from its list of servers. On the other hand, if server "A" receives a heartbeat message from a new server "C", it adds server "C" information to its list. We set two different timeout here, one for sending the heartbeat to the other servers, and another one to decide if any of the other servers is dead. Please note that we need to be careful about setting the timeout for receiving heartbeat not be too small, because the list of servers on all servers would be empty. Also, the interval of sending heartbeat should not be too short, since in this case multiple thread will run and the network would be very complicated. We set the heartbeat send timeout and heartbeat receive timeout to 40s and 80s respectively.

3 Download with Proxy

Following the previous section, all the servers have a copy of uploaded file. So, we can ask any of them to download the desired file (if it is already uploaded on one of the servers). In order to decide which server is the best option to download the required file (in terms of price and download time), we implement a HTTP proxy. There is a server configuration API which provides the proxy with the list of available servers for downloading the file.

To know the cost of downloading files from different servers, we implement a server that provide an API to respond to all queries about the costs. This API works with a Json file in server side which includes all price information for different servers. The proxy can send a get request to this server to know all price information of different servers. This API also provides us with POST and DELETE (a sub version of post) requests to update the price information for a particular server, or delete the cost information of a server if it leaves the cloud.

Knowing the price may not be enough to decide which server is the best for downloading the file. Sometime it is more important to download the file as fast as possible, so we need to know how long does it take to download a file from each of the servers. In order to estimate the delay, the proxy send a packet to each of servers it knows and measures the round-trip time (it pings all the servers).

Now, we have cost and delay for all the servers. The proxy provides the functionality for the user to choose which factor is more important by setting an input argument. It can consider only the cost or the delay or mix them together and calculate a weighted measurement of them (here also user can decide about the weight by setting it as an input argument). Finally, the proxy ranks the servers by the score they get and find the best one (fastest or cheapest or both). Once the best server selected, the proxy connects to its API, and sends a request to download the desired file. This API is the same one that provides the proxy with the list of the available servers and their information (IP and port). It also supports a POST request to update the list of available servers.

The only concern remained is that sometimes the client is on a slow link, so in case of downloading bigger files like images, even choosing the best server cannot be that much helpful. To solve this problem, we add a functionality to server to resize the image and send the smaller version to the client in this situations. We implement this part with two different scenarios:

- **Static:** Whenever the server receives an image, resizes it to two different lower quality images by changing the size, and saves them along with the original image. So, when the proxy sends a request to server API for downloading the required image, first the API responds with a random 10Mb file to the proxy. Proxy measures the download time for that file by getting the time of the system before and after the download, and sends it back to the API (by another request), so the API has the data rate of the link (by dividing 10Mb to the time measured by proxy). Now, API can decide if the link is slow or not, and send the appropriate version of the image (original, lower, or lowest quality) to the proxy.

- **Dynamic:** In this scenario, server only saves the original image. When the proxy sends request to a server API for downloading the image, API follow the same instructions, as the previous case, to calculate the data rate of the link. The only difference between this approach and the previous one is that having the data rate, API opens the original image and resizes it regarding to the measured data rate, and sends the resized version to the proxy.

In more details, if the 10Mb random file can be sent less than 10ms, the server API sends the original version to proxy. If it takes 10-100ms that proxy receives the random file, API sends the lower quality image. Otherwise, if sending random file takes more time, API sends the lowest quality version of the image to the proxy. We choose these thresholds by running the whole system using different Networks (home Wireless and Mobile Hotspot).

As the proxy handles multi-threading, we can request multiple files for downloading in the same time. For testing the proxy, we configure our Firefox to loop back to the local address (127.0.0.1). Overall, in this project, the role of proxy is to sitting between the client and servers, get the requests from client, choose the best server to download the client's required files, and download them for the client.

4 SSL

Another part of the project is to provide a secure connection among the servers on different clouds. In this section, for generating the certificate and random key we follow a the self-signed approach. Actually, we create one random certificate and one random key files, and share them among all servers. As we are opening a SSL connection in server side, we also need to have the certificate file in client side (upload.py). In this case, for opening the socket on server side, we need both certificate and key files, but in receiver side, we only need the certificate file. SSL makes the connection between the servers more secure, but it is not as fast as the simple socket connection, since hand-shaking between two sides takes more time in this scenario. SSL connection can be useful in case we need to send or receive messages in a secure manner like online banking.

5 Analysis: Json vs. Protobuf?

As we mentioned before, our upload implementation supports both Json and Protobuf3 encoding. But the question is that which method is better. Figure 1 compares the size of packets sent with Json and Protobuf3 encodings. For this experiment, we randomly generate 1Kb, 10Kb, 100Kb, 1Mb, and 10Mb text files, convert them to Json and Protobuf3 formats and calculate the byte size of the generated packets. However, the difference is not that much sensible (when we draw the line plot of measurement, the difference cannot be caught by eyes), but it is obvious that the size of packets when we use Protobuf3 is smaller, so transferring the data would be faster in this case. The other advantage of Protobuf3 is that we can specify the type of elements in this format (if the value for a specific key is integer, string, etc). But using Protobuf3 also has a negative side that is the installation part is a little bit hard and not that much straightforward.

	Packet Size (byte)	
File Size	Json	Protobuf3
1K	1466	1385
10K	13754	13673
100K	136634	136554
1M	1398202	1398122
10M	13981114	13981035

Table 1: Comparison between packet size using Json and Protobuf3