Mychael Walker
12/6/16
CSCI 156

Client & Server BLOB Data Transfer

The implementation of TCP and UDP data transfer is not a very easy one, but it is a rewarding endeavour. The following code accomplishes a specific streaming strategy using socket programming, which involves a Client and a Server with both TCP and UDP protocols. The following report will discuss the highlights of the coding process, the structure of the code, and an analysis of the completed project.

The Client sends UDP packets to the Server at a specific loss rate implemented, by default it is a 10% miss chance rate. The Client sends large data structures encoded in JSON to the Server. The Server returns keeps track of what packets were missed, and requests the missed packets from the Client via TCP. When the Client receives the TCP input in the form of a struct with a data member of an slice (GO's vector) of ints, it attempts to resend the packet. Maintaining all of these asynchronous tasks was a difficult task, and the different functions running on goroutines (virtualized threads). The next page is a diagram showing the Client and Server,and all of the goroutines and which functions they are running.

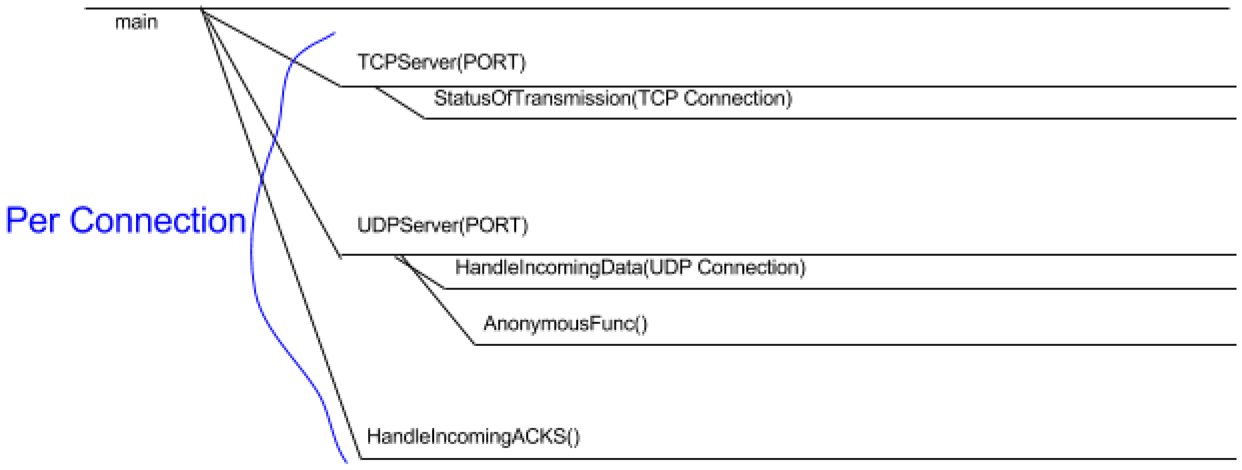
The two applications use channels to share resources between the threads, and the applications use waitgroups to have specific goroutines wait for information from other goroutines. The channels act as a stack, and if the stack is empty, a goroutine will hold and wait until another goroutine passes an item to the stack. Go's TCP and UDP connections have built in protection, which allows multiple goroutines to read and write without having any danger of corrupting data, allowing duplex reading and writing via goroutines.

The program is set up in such a way that any number of connections can be made, because each connection gets its own set of data and runs on its own threads. Goroutines are an amazing tool that was given to us, and many go applications can use hundreds of thousands of goroutines. They are extremely inexpensive to start up, and take only 1kb of memory, so excessive use of them can be extremely beneficial when done correctly. Refer to the last page of the report for a diagram regarding each goroutine.

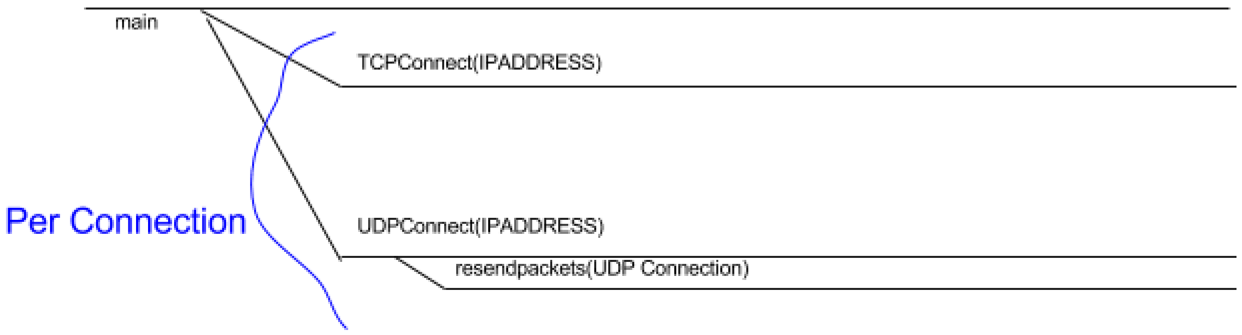
Each time a packet is sent from the client, when the server receives it, it puts it into a data type that is used to notify the client on which packets to resend every 5 seconds. Implementing this via goroutines makes it so that it can continue its original file transfer, even while transmitting lost packets. The chance that a packet will be sent is simulated each time the client attempts to upload a file to the server. If the server has not received anything for 25 seconds, it assumes that the file transfer is done and closes out the connections. The server places each packet that needs to be resent onto a map, of ints to ints. The key of the map is the packetid, and the value of the key is how many times it has been tried to be resent. If it has tried to send up to times “retrylimit” it will delete it from the resend map, and move forward eliminating the chance that we will get that packet again.

Implementing this specific type of TCP and UDP file transfer was very interesting, and used large amounts of technology that I have never used before. While I have played with goroutines for very minimal operations in the past, using them in this robust developed environment shows exactly how capable they are. The concurrency required for this application is paramount, because a udp and tcp server would not be able to be ran at the same time without it. This project tested my knowledge of threads much more than the 144 project this semester.

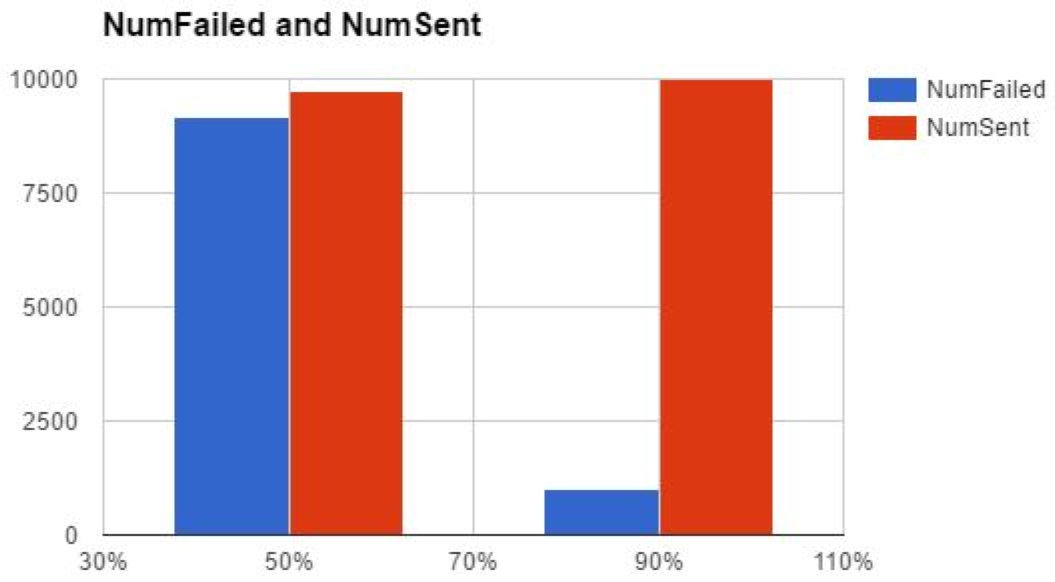
Server (receives file from Client)



Client(Sends file To Server)



Percent to Pass	NumFailed	NumSent
50%	9182	9719
90%	1020	9996


NumFailed and NumSent
NumFailed
NumSent