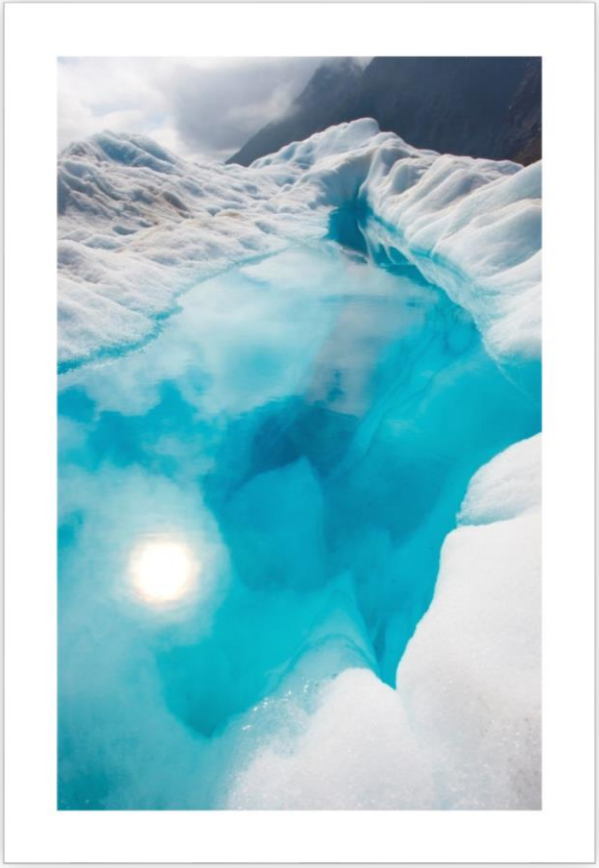
AAIT
ITSC – Distributed System Programming - 2016



[Group Members]

1. **Henok Getachew ATR-1651-06**
2. **Haileyesus Zemed ATR-3361-06**
3. **Mikiyas Birhanu ATR-3542-06**
4. **Yishak Abreham ATR-4241-06**

Project Title: Local/Offline Version Control System

Abstract

On this project we worked on an offline/local version control system where you can use the features that are available in most known version control systems within your local group of developers.

Design Implementation

Our system consists of Server go file and client go file which utilizes the socket mechanism to establish the communication between the server and the client.

Server.go	Client.go
main() – launches the server to start listening on localhost:9999	main() – launches the client to dial on localhost:9999
fillString() – fills in the data that has to be send to client with ":" b/c in tcp server promises the client how many bytes it will send it then the client will not stop until it has received all the bytes that has been promised by the server.	fillString() – same as the server's function
sendFileToClient(connection net.Conn) – used to send the versioned file to client by passing it as a byte by byte.	sendFileToServer() – sends the a version file to client by passing it byte by byte and if necessary adds ":" using fillString() b/c it will first promise the server some bytes and that many bytes need to be transferred or else the server waits infinitely

recieveFile(connection net.Conn, isFromBackup bool) – use to receive file from client byte by byte and removing it if it contains ":" which is added by fillString().	recieveFile(connection net.Conn, isFromBackup bool) – same as client
handleConnection() – handles the connection. Most operations are handled here on the handleConnection() function.	handleConnection() – handles all the necessary connection stuff. The command line arguments are entered here on the handleconnection() function. For example "lg -commit"

Why Did We Choose Socket?

Sockets are very simple to understand and work with instead of RPC or some other mechanisms.

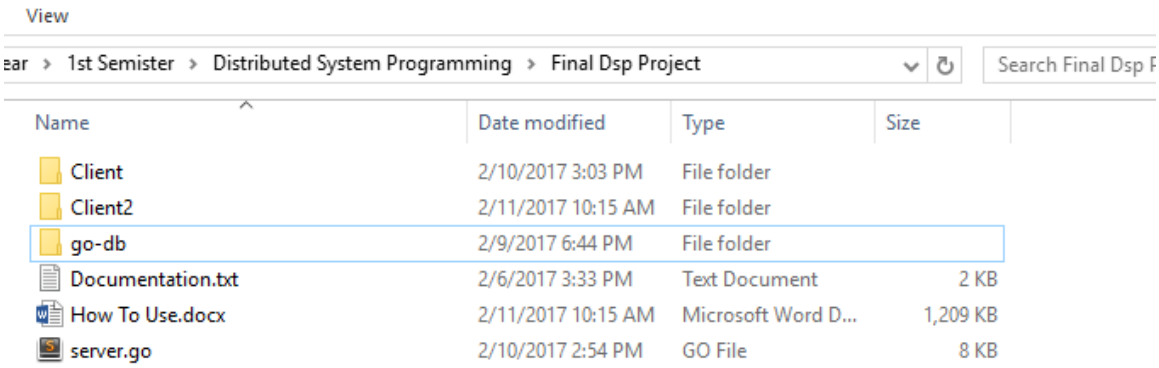
Sockets are also very easy to transfer files across multiple peers and our system mainly utilized the transfer of files.

In a system like ours which uses many file transfers we prefer using Socket b/c of its ease and flexible usage to transfer files across connected peers.

Content

How To Setup

Our Project contains the following files



Server.go – is the main server that will be used to handle the version control commands that come from clients

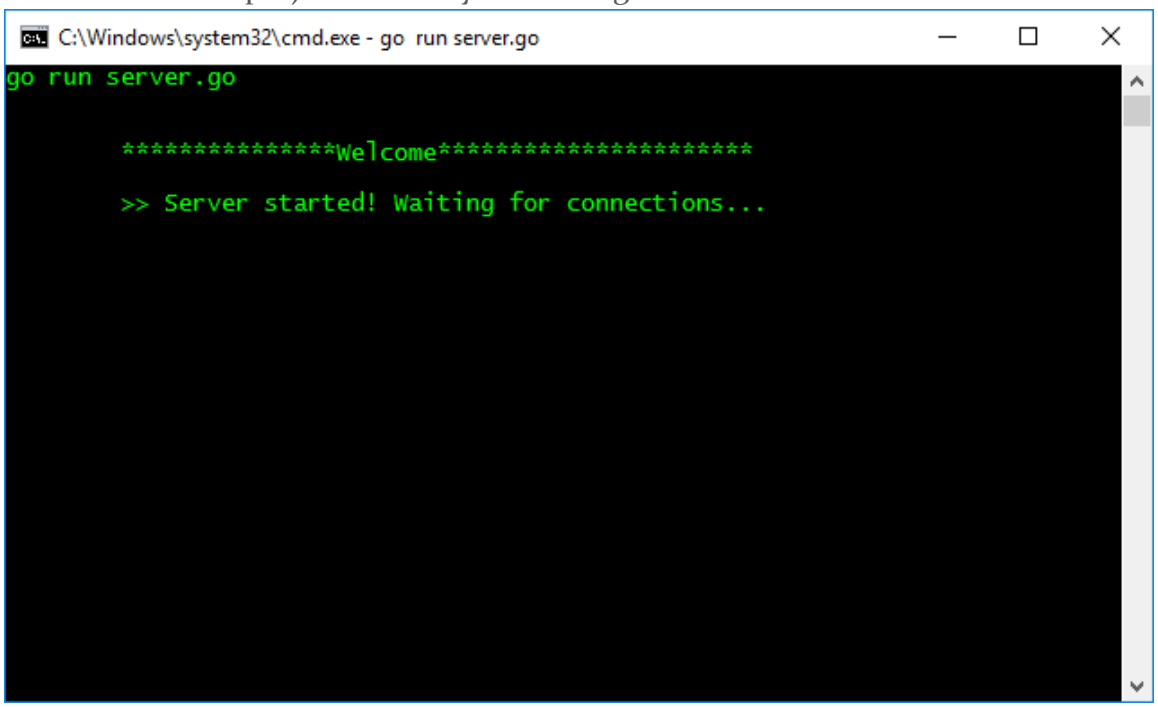
Inside Client and Client2 folders there exists a client.go file which enables us to be a client.

– Our Local Git works as the following.

First a group of programmers get together and create a hotspot/wifi to connect to each other without the need for internet. So you will need two pc's

Then one pc will be chosen as the server(that pc can also simultaneously be a client as well). You can run the server with the following cmd: go run server.go

While still in the project directory. You will get this welcome text.

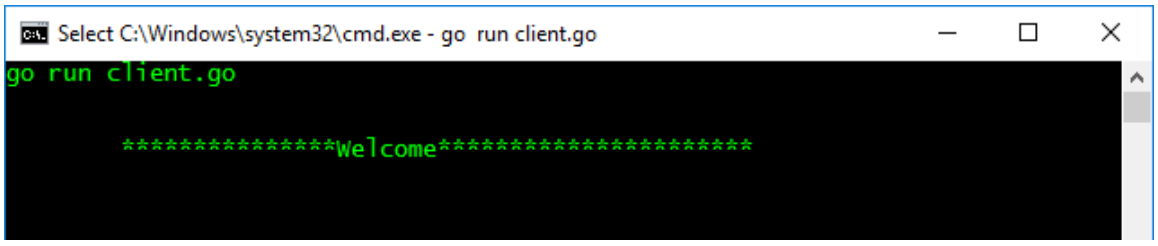


That's it now the pc is a server and its waiting for a connection on ***localhost***

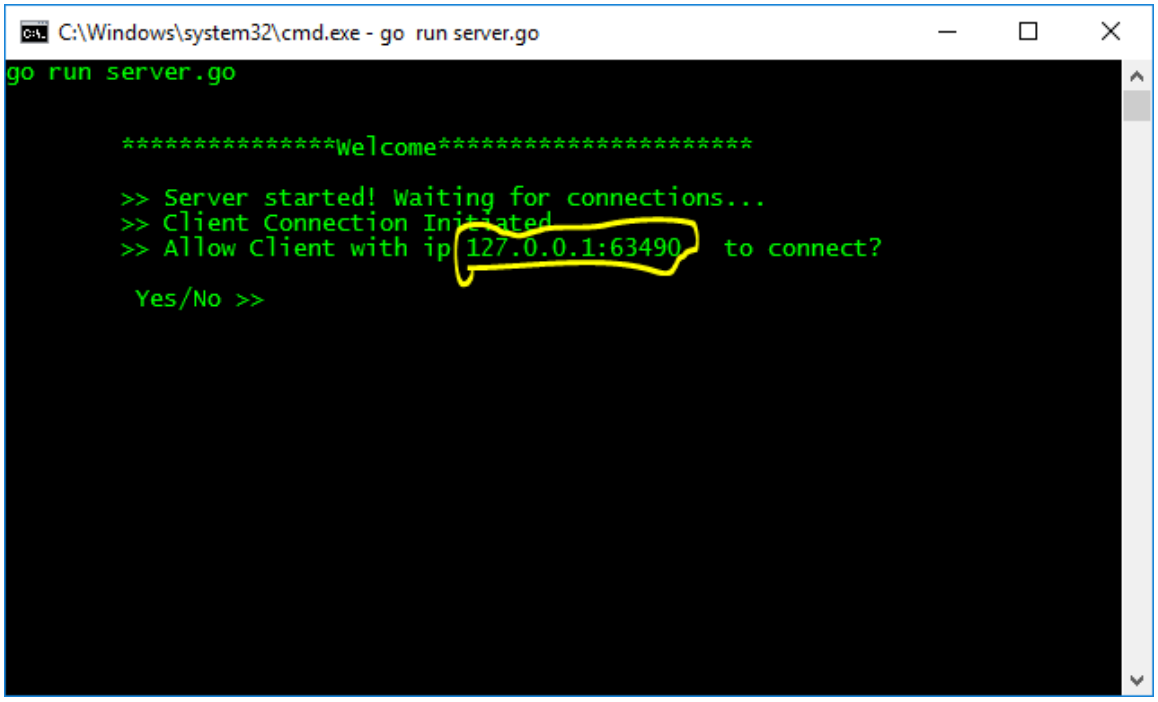
Now in order to be a client and connect to that pc we need to enter the server ip in the source code of client.go in the line **25**

```
// connection, err := net.Dial("tcp", "10.5.12.114:9999")
connection, err := net.Dial("tcp", "localhost:9999")
```

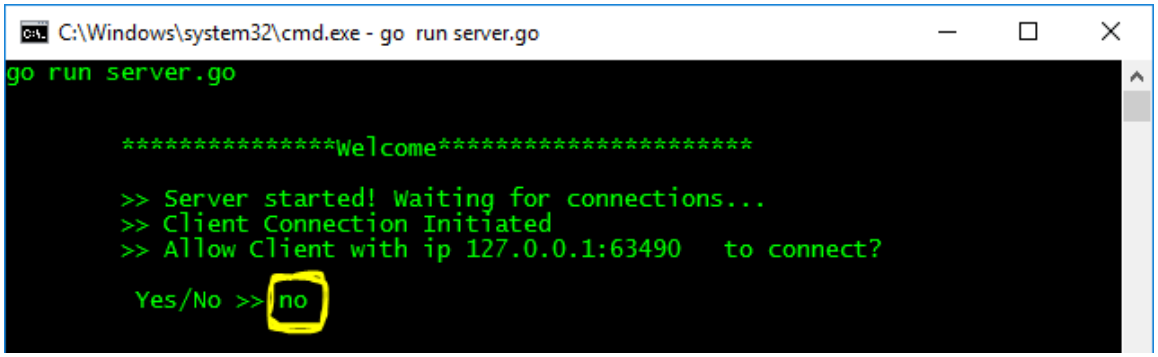
Then run client.go as: go run client.go and you will see the following welcome screen.



As soon as you run client.go you will be dialing to server's ip and the server will display the following



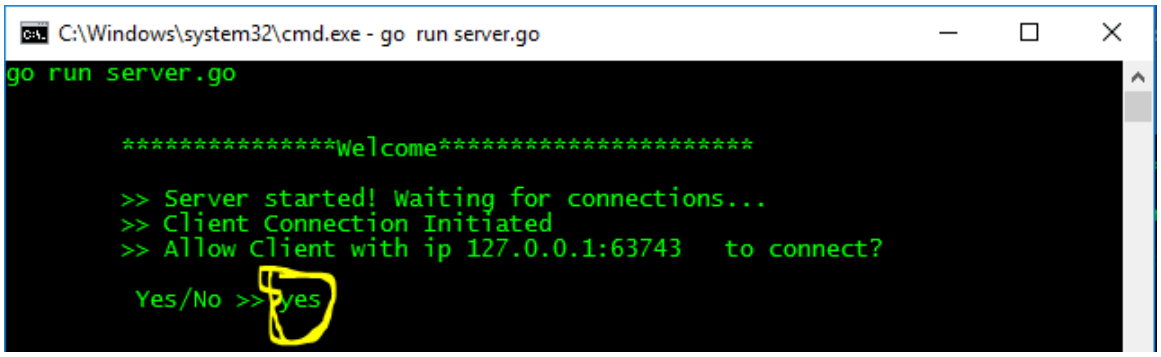
You will be prompted to allow the client with the ip address to join the local version control system. If you say no. client will be rejected and connection closed.



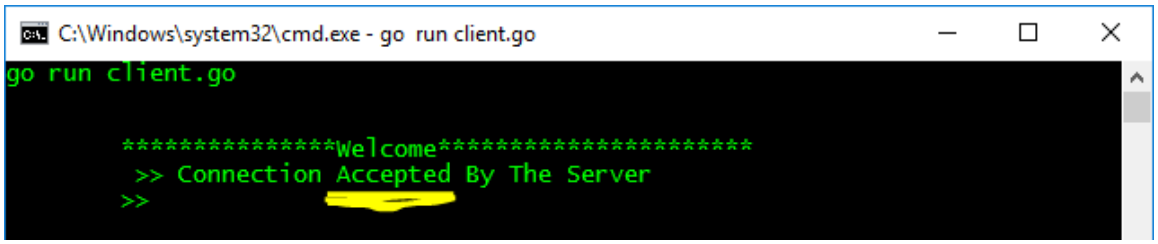
But if you say yes the connection will be granted and client will be able to

- Commit
- Create
- Update
- Backup

- Update To Version Number



If you say yes as the above picture shows you will receive a confirmation In the client side as well



Now the connection has been set and we are ready to start using our version control system.

How To Use

These are the commands line arguments you can enter, every command is preceded with “lg” then the argument

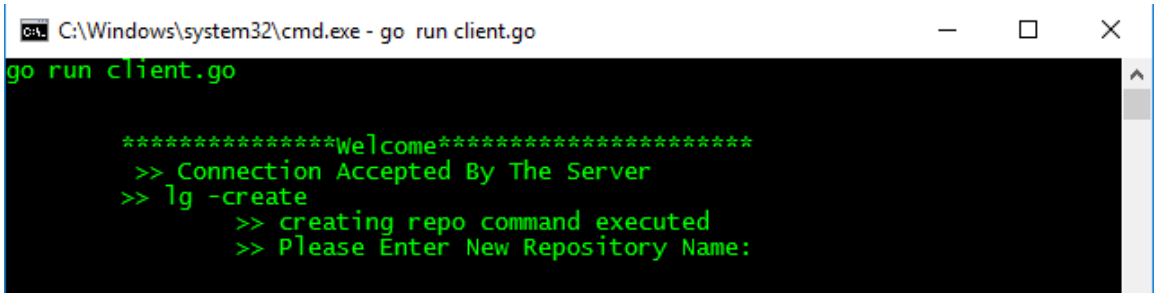
1. “`lg -create`” - to create a new repository on the server
2. “`lg -commit`” - to commit a file to the repository
3. “`lg -update`” - to update your version of the file from the repository
4. “`lg -uptovX`” - to rollback to a specific version of a file
    a. Here X means the version number example lg -uptov3, lg -uptov2
5. “`lg -backup`” - to allow the server to backup and replicate its files to your computer
6. “`lg -logout`” - to exit the session that has been established

The above commands can be executed after a connection to the server has been established.

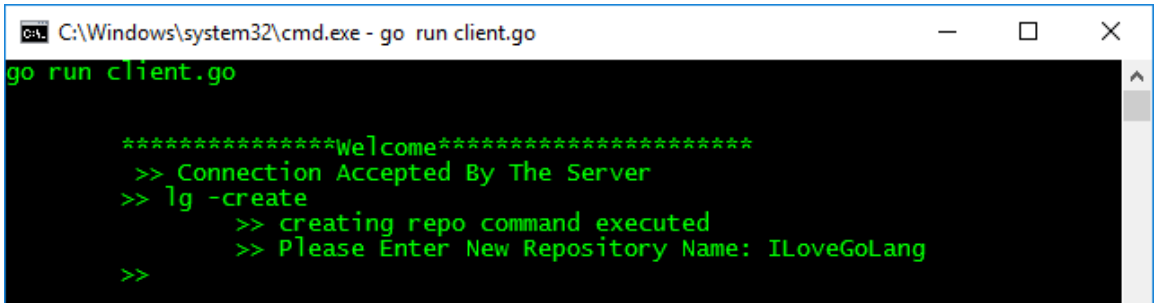
After the server replied with yes.

Creating A Repository

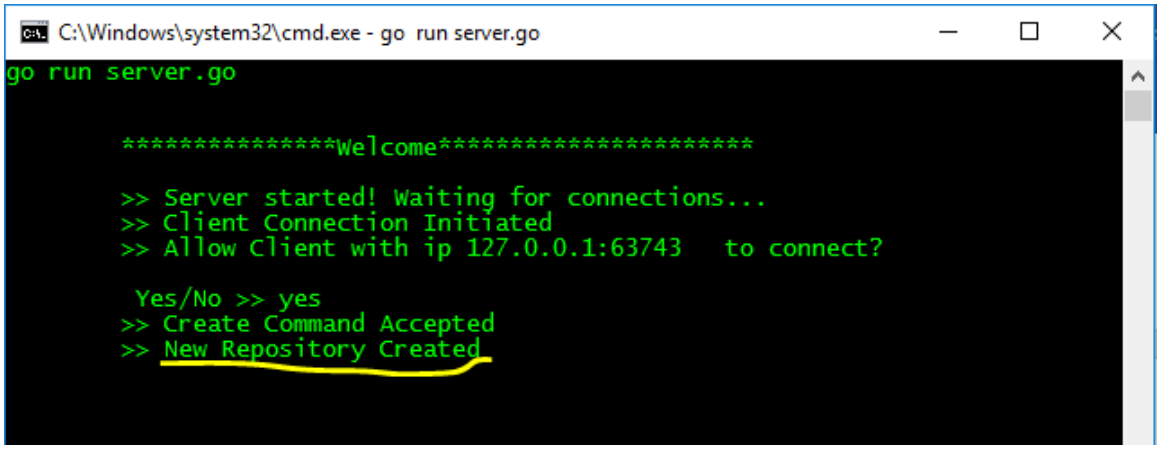
Our client will execute “lg -create” to create a repository



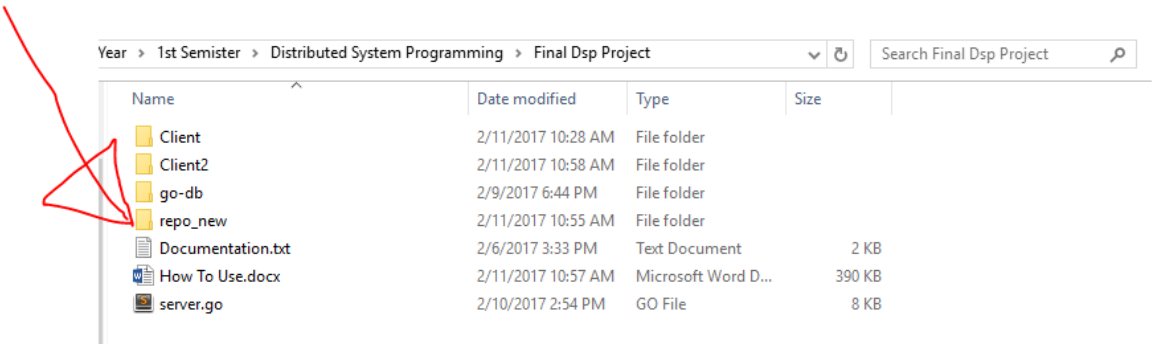
Then the client is asked to enter the repository name which we entered “ILoveGoLang”



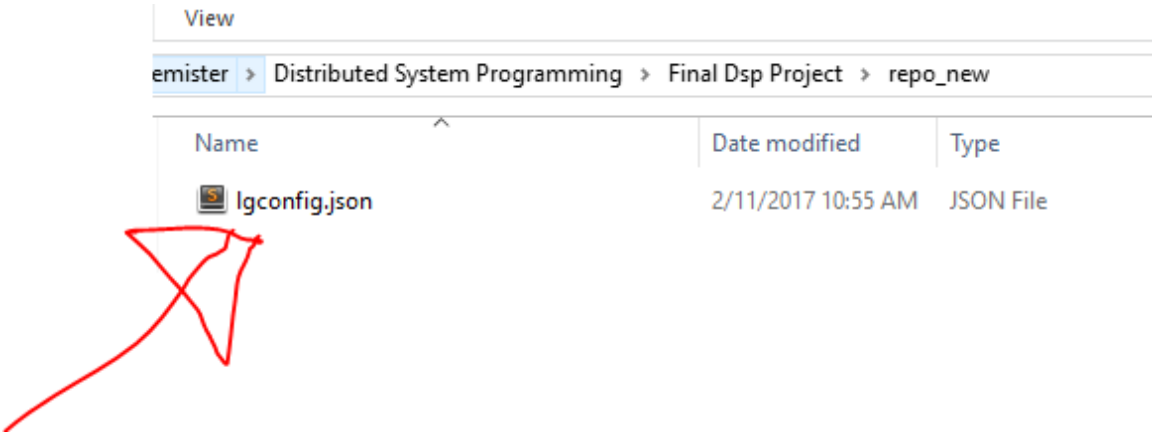
And the server immediately displays that it accepted a create repository command and when you enter the name it will display that it has created the repository.



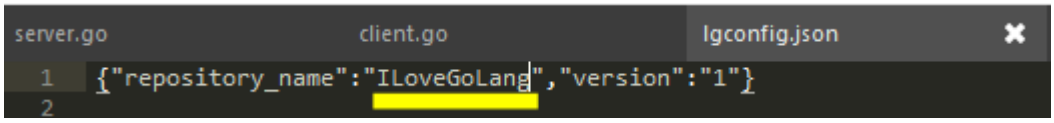
As you can clearly see repo_new folder was created on the same directory as the server.go file.



Inside repo_new folder



There is a file which will be called lgconfig.json which contains the repository name and the version number.



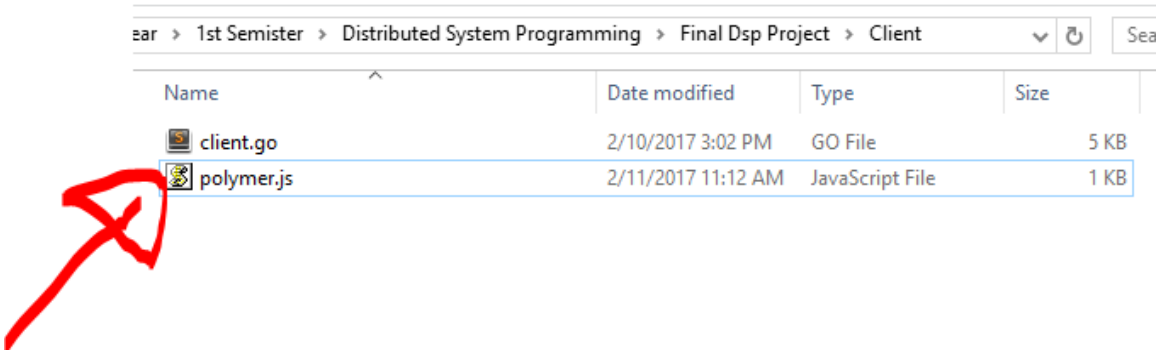
Inside lgconfig.json the repository name is “ILoveGoLang” and version number is set to 1 by default because nothing has been commited yet.

By that we are done creating a repository.

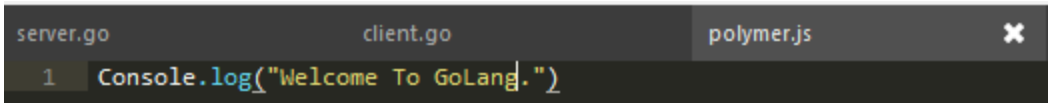
Committing to a Repository

Our client will execute “lg -commit” to commit a file to the repository.

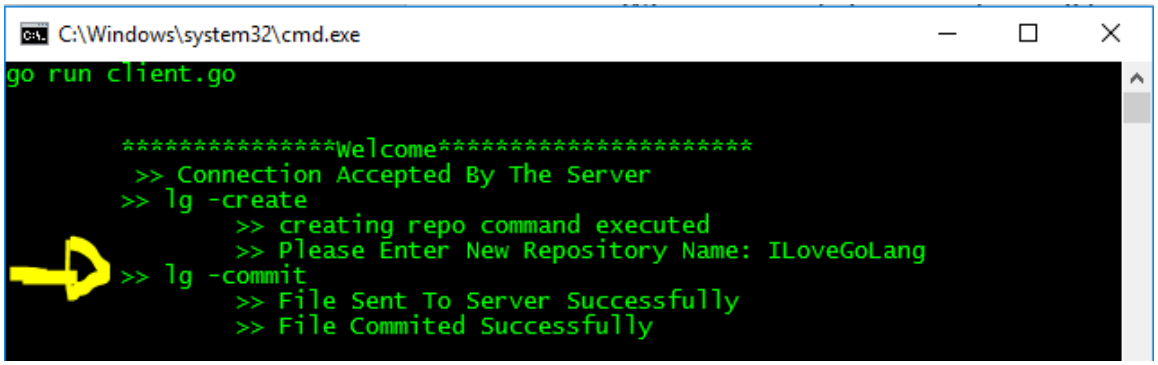
There is a sample file called Polymer.js in the same directory as the client.go file



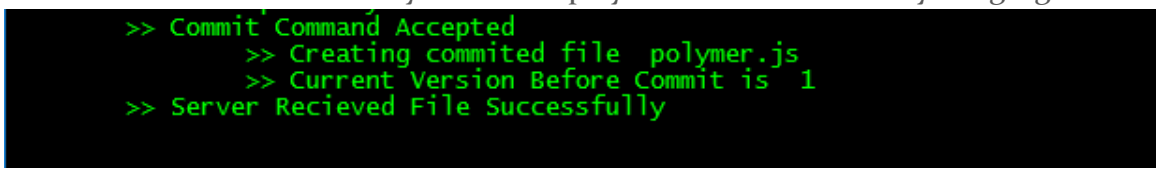
Which contains the following



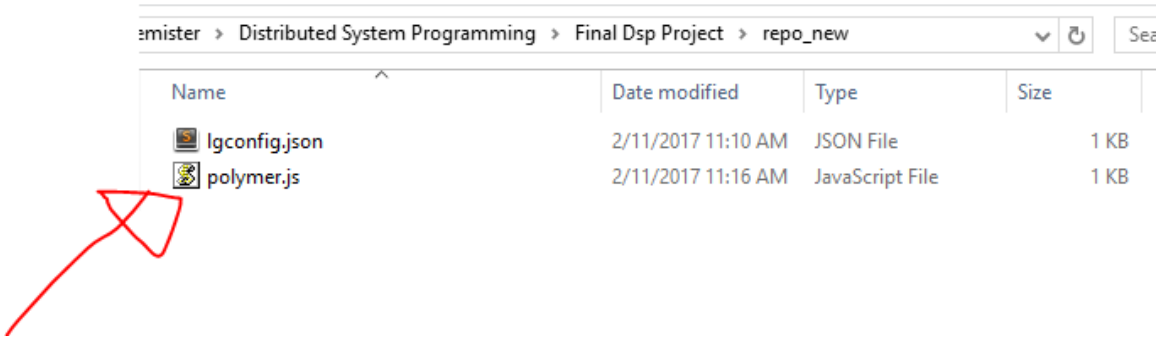
When we execute lg -commit we will be commiting polymer.js



File sent to server successfully will be display if we have done everything right



The above screenshot shows us that the commit was successfully done.



And our polymer.js file is in the repo_new directory where the server keeps the config file and the commited files.

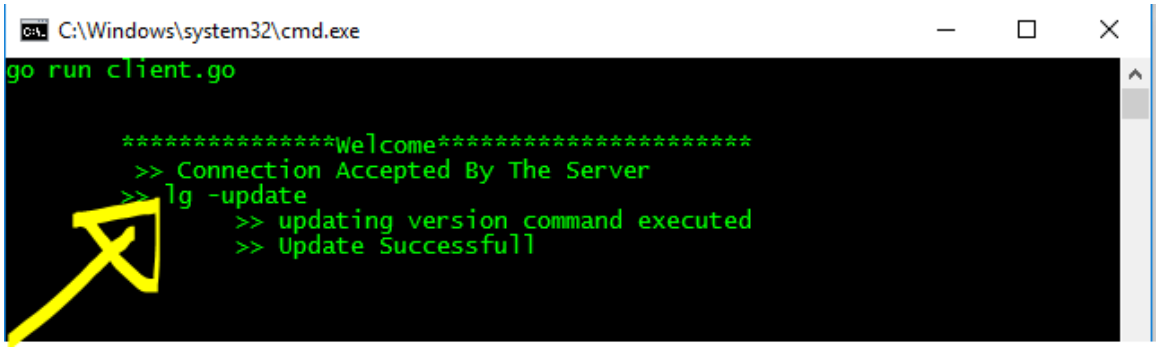
Checkout from a Repository

Our local version control system will treat checkout and update operations as the same

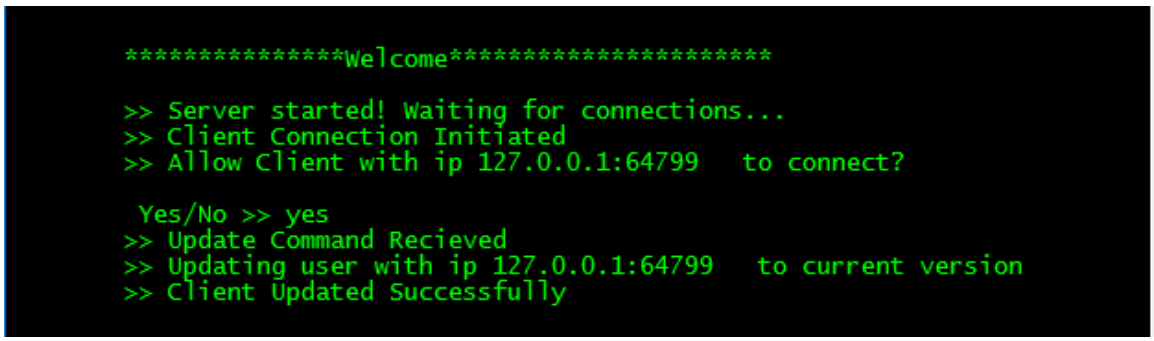
Our client will execute “lg -update” to checkout from a repository or update to the current/latest version of their file.

We now close the client.go file we have been working on and move to client**2** **folder** and run client.go as go run client.go and repeat the above steps to connect to server.

After successfully connecting enter lg -update to checkout from the repository to get the file that someone else has committed.

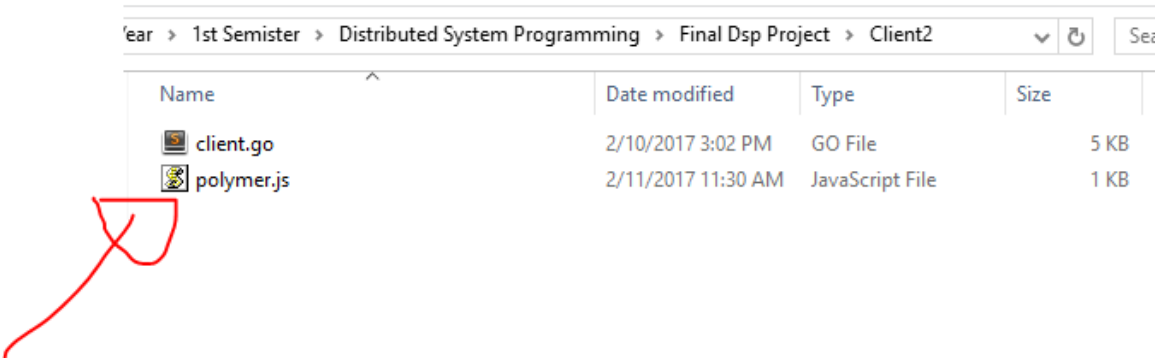


As we see from the above pic the update was successful and on the picture found below we can see the server updated the client successfully.



if we go and check our directory we will find the polymer.js file that the first client

committed with the same content.



Updating Version

Client2 can update the polymer.js file and commit as follows.

Before editing polymer.js has this content

server.go client.go polymer.js

```
1  Console.log("Welcome To GoLang.")
```

After editing

server.go client.go polymer.js

```
1  Console.log("Welcome To DSP.")
```

Now we have edited the polymer.js file, we will commit

On the server the version will be updated on the lgconfig file

```
        >> Commit Command Accepted
                >> Creating commited file  polymer.js
                >> Current Version Before Commit is  1
                >> Already Versioned File Exists so Updating Version...
                >> Current Version After Commit is  2
        >> Server Recieved File Successfully
```

The following shows the content of the lgconfig.json file after version has been updated.

server.go client.go lgconfig.json

```
1  {"repository_name":"ILoveGoLang","version":"2"}
2
```

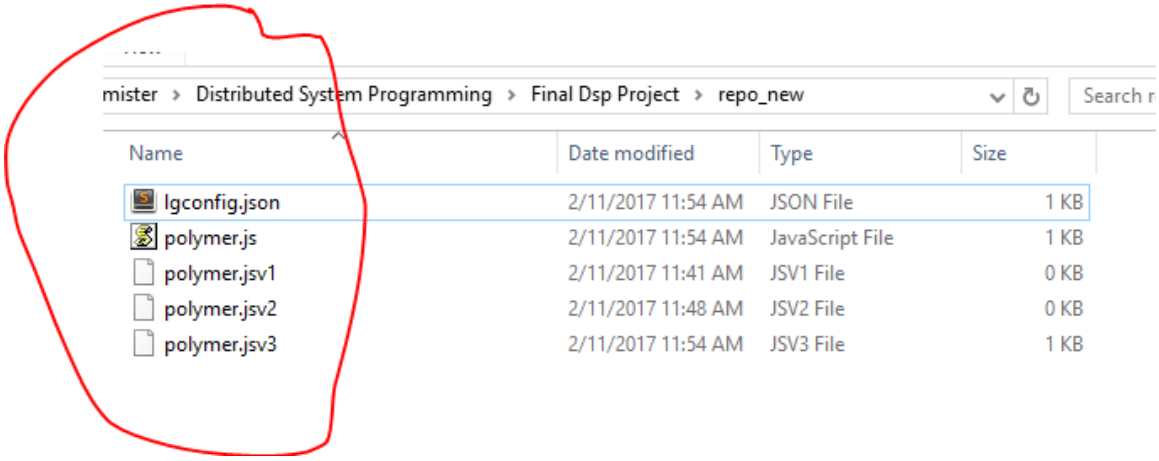
On the server the original polymer.js file has been renamed to **polymer.jsv1**

mister › Distributed System Programming › Final Dsp Project › repo_new

Name	Date modified	Type
lgconfig.json	2/11/2017 11:48 AM	JSON File
polymer.js	2/11/2017 11:48 AM	JavaScript File
polymer.jsv1	2/11/2017 11:41 AM	JSV1 File

The latest file is still polymer.js but the version 1 of the file is still in the repository and renamed to polymer.jsv1

We can go to many versioned files as we can.

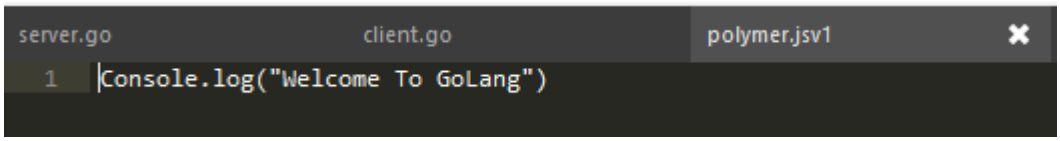


Rollback to previous Version

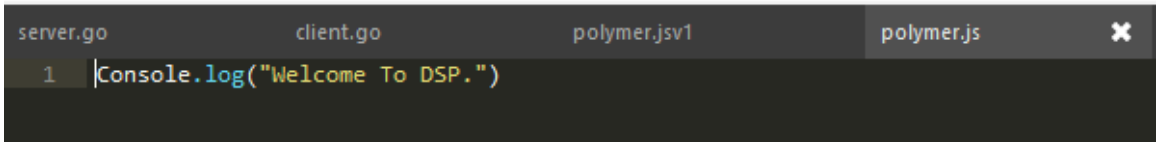
Clients can also rollback to a previous version of their code with the command

“lg -uptovX” X representing the version number they want to rollback to.

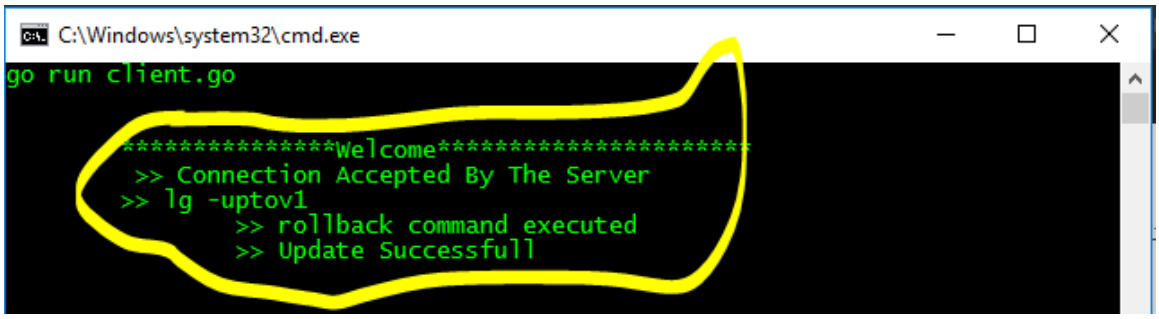
Currently polymer.js **Version One** has the following



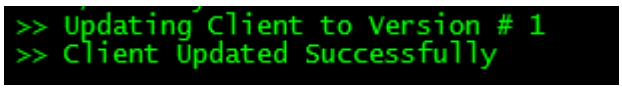
And **Version Four** has



Now on the client program we can do “lg -uptov1” to go back to the first version of the file



On the server we get successful rollback message.



We may need to rename our file after rolling back b/c we are not deleting the original file and replacing it the rollbacked file. The rollbacked file will be having the name POLYMER.JSV1 so renaming it ot polymer.js is our manual job

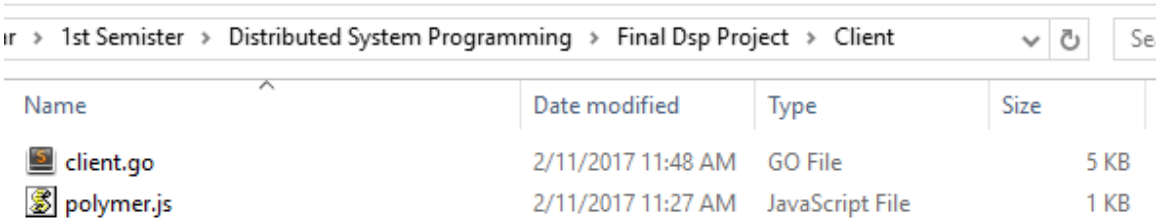
Backup And Replication

Our Distributed Local Version Control System (AKA local git)

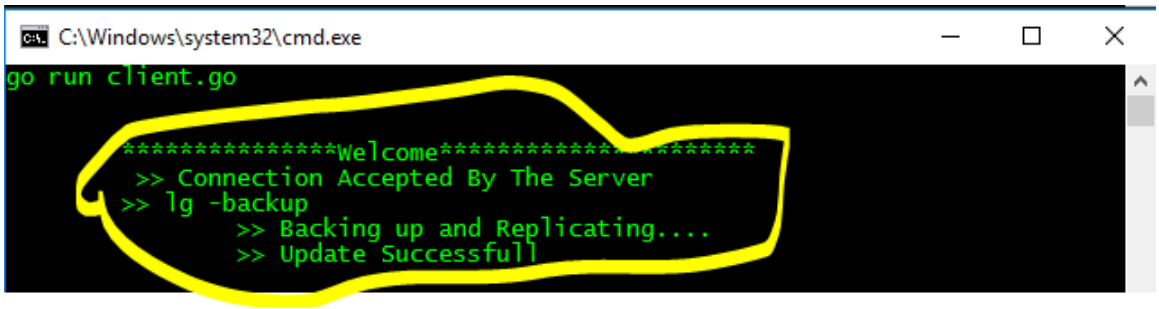
Has a backup and replication mechanism where one client can give permission to the server to backup and replicate its content to the client.

On the client program execute “lg -backup” to give the server permission to backup and replicate on its storage space.

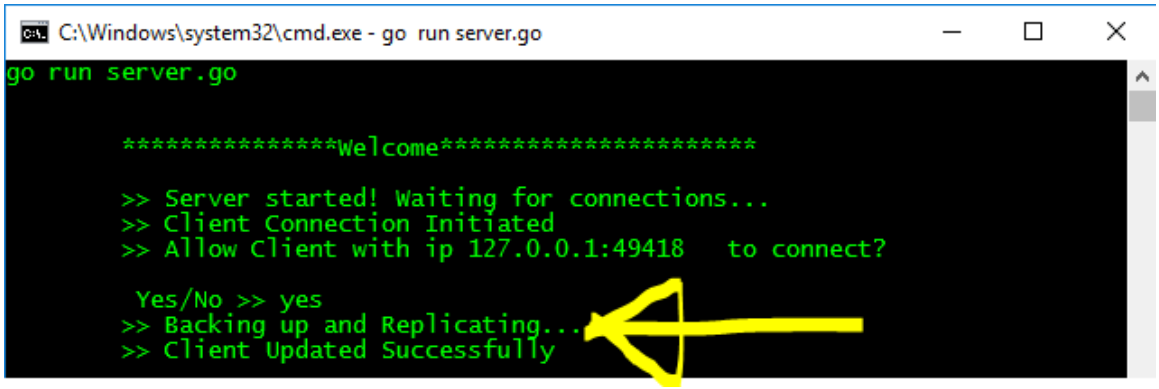
Now we go back to the first client that created the repository and the directory structure looks like this



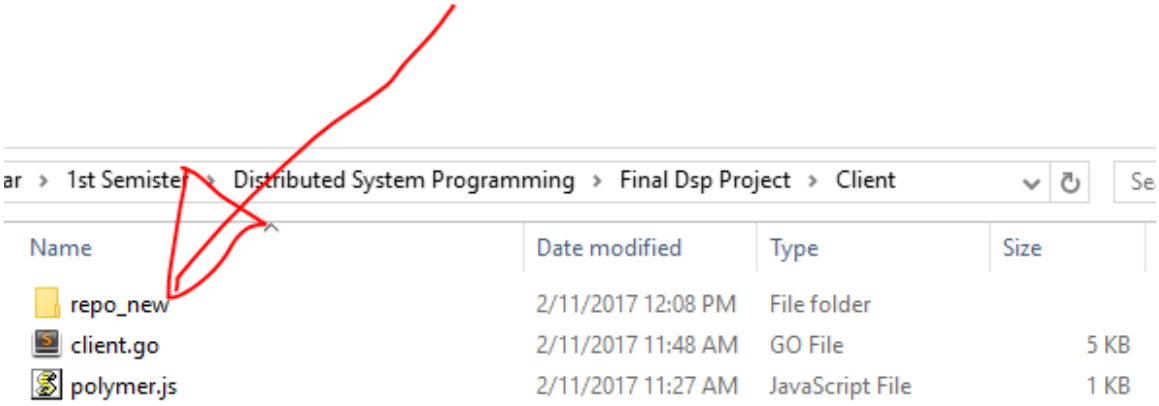
Execute “lg -backup” on client



After backup on server

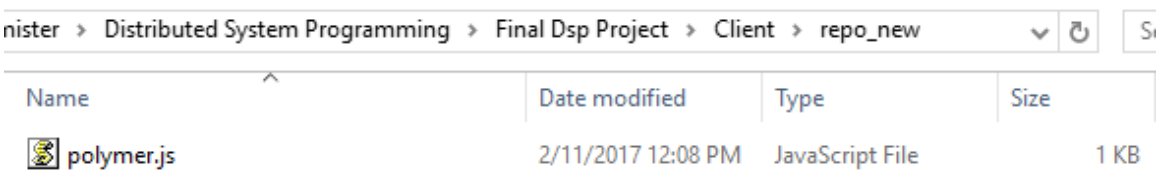


After backing up and replicating on the client the folder structure will looks as the following.



The server will backup and replicate the repo_new folder with the latest polymer.js that has been committed to the clients storage space.

Inside repo_new that has been backup to clients space



Conclusion

Our distributed system is flexible and usable to some extent. We have added the major functionalities of the worlds most known version control systems(GIT, SVN...) we have implemented some of the most crucial part of distributed system components like backup and replication and file consistency across our clients.

This local/offline version control system uses Socket to communicate between client and server.