Distributed Systems (Assignment-3):

Problem:

Design and implement a Remote Procedure Call framework to simulate the following:

- Implement a single server architecture with support for multiple clients
- The server maintains a list of graphs each associated with a distinct identifier.
- Clients can request to add a new graph, update an existing graph and query for the total weight of the minimum weight spanning tree of a given graph.
- Clients can request to add a new graph using **'add_graph <graph_identifier> n'**. This command will add a new graph on the server with the identifier 'graph_identifier' and 'n' number of nodes. The graph identifier is a string with a maximum length of 10 and it won't already exist. 'n' will be in the range: 1 <= n <= 100,000.
- Clients can request to add a new edge in a graph using **'add_edge <graph_identifier> <u> <v> <w>'**. This will add an undirected edge between the nodes **u** and **v** with weight **w**. **u** and **v** are the node numbers of the endpoints of the edge such that 1 <= u, v <= n and 0 <= w <= 10,000. **n** is the number of nodes in the specified graph. A graph with identifier graph identifier will already exist. There can be multiple edges and self-loops added to the graph.
- Clients can request for the total weight of the minimum weight spanning tree in a graph from the server using **'get_mst <graph_identifier>'**. The client will print the solution the server returns. In case the graph does not have a spanning tree, -1 should be printed. A graph with identifier graph_identifier will already exist.
- All values should fit in 32-bit signed integers.
- The server should be able to handle multiple clients simultaneously and should also work with clients on other machines.
- You are free to use any algorithm for MST

Solution Approach

- Programming language for Client and Server - C++
- Using Apache **'thrift'** package for Remote Procedure Calls.
- Create an ***interface.thrift*** IDL file for thrift compiler to generate the RPC service code. This file will contain the interfaces that will be exposed for the client to perform a remote procedure call to the server. Given below is the IDL file used for current assignment

```
service GraphService
{
  oneway void addGraph(1:string graphId,
                       2:i32 nodeCount)
   i32 addEdge(1:string graphId,
               2:i32 srcNode,
               3:i32 dstNode,
               4:i32 edgeWeight)
   i32 getMSTWeight(1:string graphId)
}
```

On Client Side:
Implement a menu-driven approach for performing all the above operations. A typical menu implemented in the client is as shown below:

```
 ____________________________________________________________________
| Add a Graph    ( Command: add_graph <graph_id_string> N )          |
| Add an Edge    ( Command: add_edge <graph_id_string> <u> <v> <w>) |
| Get MST Weight ( Command: get_mst <graph_id_string> )              |
|____________________________________________________________________|
|                          "-1" to EXIT                              |
|____________________________________________________________________|
Command: _
```

On Server Side:
- Use a std::map<key, value> as a graph container. The 'key' would be a string to identify the Graph and 'value' would be a pointer to the Graph.
- Implement a 'Graph' class encapsulating all the interfaces that can be called from the server. Use Kruskal's algorithm for computing the Minimum Spanning Tree for an undirected graph, using *Union-Find* data structure for better performance.

Environment Details

Operating System: *Mac OSX BigSur 11.2.2*
Package Installer : *Homebrew*
Compiler : *clang++*
RPC Package : *Apache Thrift*

Installation Instructions:
- Install CommandLineTools and Homebrew package manager on Mac OSX. Refer this link for detailed installation steps.
- The C++ compiler 'clang++' would come along with the CommandLineTools package.
- Using Homebrew package installer install Apache 'thrift' as follows:

```
brew install thrift
```

- Make a note of the include and the dynamic library paths in the Apache 'thrift' installation to update the Makefile. A typical include and library paths would look like:
  /usr/local/Cellar/thrift/0.14.0/include
  /usr/local/Cellar/thrift/0.14.0/lib
- Update the INCLUDE and LD_LIBRARY_PATH variables in the Makefile accordingly, using the above paths from your installation.

Source Tree

Given below is the source tree for the current assignment.

```
├── DS_Assignment3.pdf
├── Makefile
├── README.md
├── bin/
├── obj/
└── src/
    ├── interface.thrift
    ├── client/
    │   ├── Client.cpp
    │   ├── ClientUtils.cpp
    │   └── ClientUtils.h
    ├── server/
    │   ├── Graph.cpp
    │   ├── Graph.h
    │   ├── GraphServiceHandler.cpp
    │   ├── GraphServiceHandler.h
    │   ├── GraphUtils.cpp
    │   ├── GraphUtils.h
    │   └── Server.cpp
    └── service/
        ├── GraphService.cpp
        ├── GraphService.h
        ├── GraphService_server.skeleton.cpp
        └── interface_types.h
```

- 'src/service/' : Directory containing auto generated files from Apache 'thrift' compiler.
- 'src/client/' : Directory containing all the client code & utility files.
- 'src/server' : Directory containing all the server code & utility files.
- Client.cpp : Code to instantiating a client.
- ClientUtils.* : Utility functions for displaying the menu and processing the input commands.
- Server.cpp : Code to instantiating a server.
- Graph.* : A class implementing all the interfaces to create the graph and algorithm for computing Minimum Spanning Tree weight.
- GraphUtils.* : Implementation of Union-Find data structure, used while computing MST.
- GraphServiceHandler.* : A service handler class derived from the RPC service files created by Apache 'thrift' compiler. Used by the server code to execute the RPC function calls.
- Run the command **'make clean; make'** from the parent directory to create the *client* and *server* executable binaries.

Building the *client* and *server* binaries

After updating the Makefile, build the client and server binaries as follows:

```
 20173071$ make clean; make
Cleaning the object files and binaries.
rm -f core ./bin/* ./obj/*

Generating interface files.
mkdir -p ./src/service
thrift -out ./src/service --gen cpp ./src/interface.thrift

clang++ -Wall -std=c++11 -I./ -I/usr/local/Cellar/thrift/0.14.0/include -c src/client/Client.cpp -o obj/Client.o
Compiled src/client/Client.cpp successfully.
clang++ -Wall -std=c++11 -I./ -I/usr/local/Cellar/thrift/0.14.0/include -c src/client/ClientUtils.cpp -o obj/ClientUtils.o
Compiled src/client/ClientUtils.cpp successfully.
clang++ -Wall -std=c++11 -I./ -I/usr/local/Cellar/thrift/0.14.0/include -c src/service/GraphService.cpp -o obj/GraphService.o
Compiled src/service/GraphService.cpp successfully.
clang++ -lthrift ./obj/Client.o ./obj/ClientUtils.o ./obj/GraphService.o -o ./bin/client
Linking client objects complete.
To start the "Client" run --> ./bin/client
```

```
clang++ -Wall -std=c++11 -I./ -I/usr/local/Cellar/thrift/0.14.0/include -c src/server/Server.cpp -o obj/Server.o
Compiled src/server/Server.cpp successfully.
clang++ -Wall -std=c++11 -I./ -I/usr/local/Cellar/thrift/0.14.0/include -c src/server/GraphServiceHandler.cpp -o obj/GraphServi
Compiled src/server/GraphServiceHandler.cpp successfully.
clang++ -Wall -std=c++11 -I./ -I/usr/local/Cellar/thrift/0.14.0/include -c src/server/Graph.cpp -o obj/Graph.o
Compiled src/server/Graph.cpp successfully.
clang++ -Wall -std=c++11 -I./ -I/usr/local/Cellar/thrift/0.14.0/include -c src/server/GraphUtils.cpp -o obj/GraphUtils.o
Compiled src/server/GraphUtils.cpp successfully.
clang++ -lthrift ./obj/Server.o ./obj/GraphServiceHandler.o ./obj/Graph.o ./obj/GraphUtils.o ./obj/GraphService.o -o ./bin/serv
Linking server objects complete.
To start the "Server" run --> ./bin/server

20173071$
```

Execution Run

- Instantiate ***./bin/server*** from one terminal
- Instantiate ***./bin/client*** from another terminal. Multiple client instantiations would require multiple terminals.
- Follow the menu in the client terminals to create the undirected graph and compute the MST weight.
- The log traces are categorized as below:

```
[INFO]  -> Informational traces
[RPCQ]  -> RPC Request traces
[RPCR]  -> RPC Response traces
[ERROR] -> Error traces
```