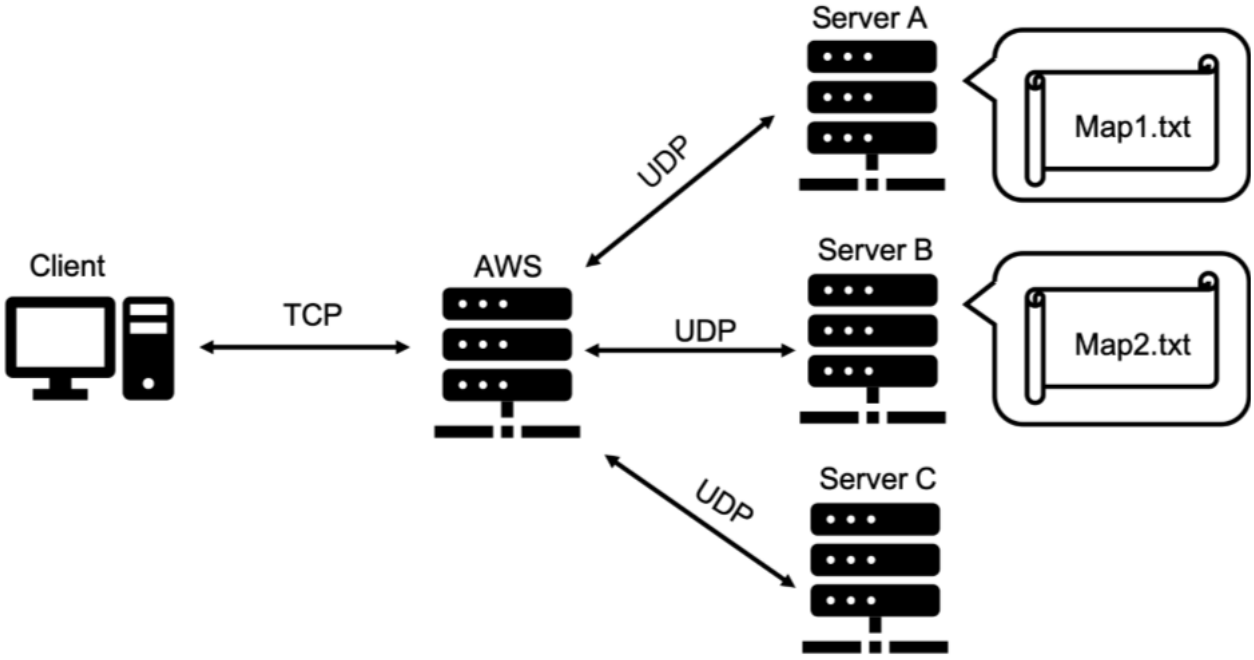
PROBLEM STATEMENT

Many network related applications require fast identification of the shortest path between a pair of nodes to optimize routing performance. Given a weighted graph $G(V,E)$ consisting of a set of vertices V and a set of edges E, we aim at finding the path in G connecting the source vertex v_1 and the destination vertex v_n, such that the total edge weight along the path is minimized. Dijkstra Algorithm is a procedure of finding the shortest path between a source and destination nodes.

In this project, we will implement a distributed system to compute the shortest path based on client's query. Suppose the system stores maps of a city, and the client would like to obtain the shortest path and the corresponding transmission delay between two points in the city. The figure below summarizes the system architecture. The distributed system consists of three computation nodes: a main server (AWS), connected to three backend servers (Server A, Server B and Server C). The backend server A and B has access to a file named map1.txt and map2.txt, respectively, storing the map information of the city. For simplicity, there is no overlap of map ID between map1.txt and map2.txt. The AWS server interfaces with the client to receive his query and to return the computed answer. Upon the request from the client, the AWS server will initiate the search work in backend Server A and Server B. After searching, AWS server will send the map result to Server C to calculate the shortest path and different types of delays (propagation delay, transmission delay and total delay). After calculation, server C will send back the result to AWS. Then AWS will get it back to the client. If there is no matched map ID, AWS will send corresponding messages to the client.



Detailed computation and communication steps performed by the system is listed below:

1. [Communication] Client -> AWS: client sends the map ID, the source node in the map and the transmission file size (unit: bit) to AWS via TCP.
2. [Communication] AWS -> Server A: AWS forwards the map ID and source node to server A via UDP.
3. [Search] Server A searches for the map ID from map1.txt.
4. [Communication] Server A -> AWS: Server A sends the search result via UDP.
5. [Communication] AWS -> Server B: AWS forwards the map ID and source node to server B via UDP.
6. [Search] Server B searches for the map ID from map2.txt.
7. [Communication] Server B -> AWS: Server B sends the search result via UDP.
8. [Communication] AWS -> Server C: AWS sends the map information to server C via UDP.

9. [Calculation] Server C calculates the shortest path using Dijkstra Algorithm and propagation/transmission/total delay.
10. [Communication] Server C -> AWS: Server C sends the shortest path and delay values to AWS via UDP.
11. [Communication] AWS -> client: AWS sends to client the shortest path and delay results, and client prints the results.

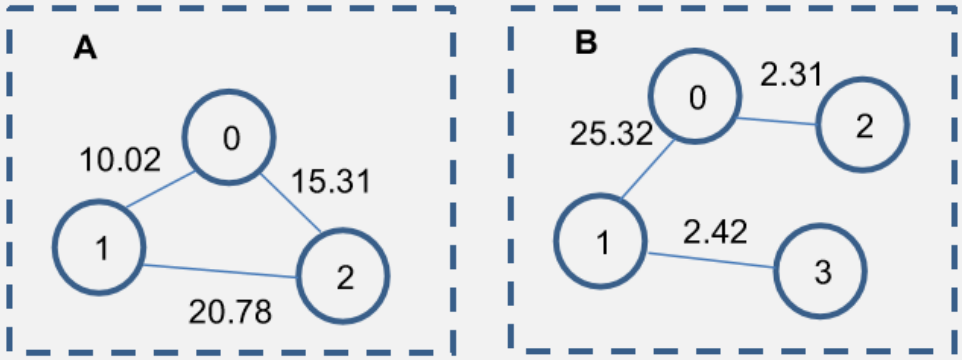
The map information of the city is stored in a file named map1.txt and map2.txt. The map1.txt and map2.txt files contain information of multiple maps (i.e. graphs), where each map can be considered as a community of the city. Within each map, the edge and vertex information are further specified, where an edge represents a communication link. We assume edges belonging to the same map have identical propagation speed and transmission speed.

The format of map1.txt or map2.txt is defined as follows:

```
<Map ID 1>
<Propagation speed>
<Transmission speed>
<Vertex index for one end of the edge> <Vertex index for the
other end> <Distance between the two vertices>
… (Specification for other edges)
<Map ID 2>
…
```

Example:

```
A
200000.00
8000000
0 1 10.02
0 2 15.31
1 2 20.78
B
150000.00
9089
0 1 25.32
0 2 2.31
1 3 2.42
…
```



Note:

1. For each map, the maximum number of vertices is 10

2. Vertices index is between 0 and 99

3. The maximum number of edges is 40

4. The graph is connected

5. We consider undirected, simple graphs:

a. There are no repeated edges or self-loops

b. An edge (p,q) in the graph means p and q are mutual neighbors

6. Datatype, Units, range:

a. Propagation speed: float, km/s, [10000.00,299792.00)

b. Transmission speed: int, KB/s, [1,1048576)

c. Distance: float, km, [1.00,10000.00)

d. Filesize: int, KB, [1,1048576)

Source Code Files:

The implementation includes the source code files described below, for each component of the system.

1. AWS: The server can be viewed as a much-simplified Amazon Web Service server. The name of this file is aws.c
2. Backend-Server A, B and C: The names of these files are serverA.c, serverB.c, serverC.c respectively
3. Client: The name of this file is client.c

DETAILED EXPLANATION

Phase 1:

All three server programs (AWS, Back-end Server A & B & C) boot up in this phase. While booting up, the servers must display a boot up message on the terminal. The format of the boot up message for each server is given in the onscreen message tables at the end of the document. As the boot up message indicates, each server must listen on the appropriate port for incoming packets/connections.

Once the server programs have booted up, the client program runs. The client displays a boot up message as indicated in the onscreen messages table. Note that the client code takes input arguments from the command line. The format for running the client code is:

```
./client <Map ID> <Source Vertex Index> <Destination Vertex Index> <File Size>
```

(Between two input arguments, there should be a space)

For example, if the client wants to calculate the end to end delay of each shortest path from source vertex 1 to vertex 3 in Map A, with file size of 1024 KB, then the command should be:

```
./client A 1 3 1024
```

After booting up, the client establishes TCP connections with AWS. After successfully establishing the connection, the client sends the input (map ID, source vertex index, destination vertex index and file size) to AWS. Once this is sent, the client should print a message in a specific format. This ends Phase 1 and we now proceed to Phase 2.

Phase 2:

In the previous phase, the client receives the query parameters from the user and sends them to the AWS server over TCP socket connection. In phase 2, the AWS server will have to query server A and server B for the corresponding map, and forward the map data to server C for calculation if the map ID and vertex ID has been found.

The socket connection between AWS and server A, B, and C are established over UDP. Each of these servers and the AWS have its unique port number specified in Port Number Allocation section with the source and destination IP address as localhost/127.0.0.1/::1.

AWS, server A, B, and C are required to print out on screen messages after executing each action as described in the "On Screen Messages" section. These messages will help with grading if the process did not execute successfully. Missing some of the on-screen messages might result in misinterpretation that your process failed to complete. Please follow the exact format when printing the on-screen messages.

Phase 2 can be subdivided into 2 phases. Phase 2A executes map storage while 2B will take the results from phase 2A and calculate the shortest path, transmission, and propagation delays if the map exists. Table 1 describes the implementation in detail.

Table 1. Server Operations

Map Storage and Search	In this operation, you will read the data file (map1.txt and map2.txt) to their respective servers (serverA and serverB). The server will be ready for query look up from AWS and return the map data if found. AWS will send the map data only if the map ID and vertex ID exists.
Path Finding and Calculation	In this operation, you will find the path of minimum length from a given start vertex to all other vertices in the selected map, using Dijkstra algorithm, then compute the transmission delay (in s), the propagation delay (in s), and the end-to-end delay (in s) for transmitting a file of given size in the selected map.

Phase 2A:

Phase 2A starts when server A and server B boots up, each server will execute map lookup against its own database. Server A will read map1.txt while server B will read map2.txt. These servers will store the map data in a data structure and will send it to AWS if the queried map ID is found on its server. If the map ID or vertex ID is not found, the server simply notifies AWS and returns to standby.

Server A reads and stores all maps from map1.txt only while serverB reads and stores all maps from map2.txt only. The queried map ID might not exist in both servers, and might exist in only one of the servers, but not both. Refer to the "Download Sample Maps" section to obtain the map1.txt / map2.txt files.

Phase 2B:

Phase 2B starts when AWS has received all required data from the client and the servers A ,B. Depending on the lookup result of servers A and B, AWS will perform one of the two operations.

1. If the queried map ID exists in neither A nor B: in this case, server C has nothing to compute for the shortest path and delay. AWS will print out a message (see the "On Screen Messages" section) and will not have any interaction with server C.

2. If the queried map ID exists in one of servers, A or B: AWS will forward to server C:

    1). the graph information received from A or B, and

    2). the source and destination vertex indices and file size received from the client.

After server C receives the information from AWS, it computes the shortest path and delay (from the source to the destination vertex) using Dijkstra's algorithm (i.e., the "Path finding and calculation" operation in Table 1). Upon completing the computation, server C will print out the calculation results (see the "On Screen Messages" section). Finally, server C will send the results to AWS, and AWS will print out the received data (see the "On Screen Messages" section). This concludes all the operations of Phase 2B.

The goal of server C is to find the path from source to destination with shortest overall delay ($T_{trans} + T_{prop}$). Since for a given graph, all the links within it have the same transmission rate and we do not use store and forward transmission, the source-to-destination transmission delay, T_{trans}=$Filesize/transmission\ rate$, is fixed regardless of the route we choose. Therefore, the path with shortest overall delay is the same as the path with the shortest distance.

Phase 3:

At the end of Phase 2B, backend server C should have the calculation results ready. Those results should be sent back to AWS using UDP. When the AWS receives the calculation results, it needs to forward all the results to the client using TCP. The results should include minimum path length between the source and destination node and 3 delays to transfer the file to corresponding destination. The clients will print out a path and a table to display the response. The table should include 6 columns. One for source node index, one for destination node index, one for path length and the other three for delays. The results are rounded to 2nd decimal point.

PORT NUMBER ALLOCATION

The ports to be used by the client and the servers are specified in the following table:

Process	Dynamic Ports	Static Ports
Backend-Server (A)	-	1 UDP, 30953
Backend-Server (B)	-	1 UDP, 31953
Backend-Server (C)	-	1 UDP, 32953
AWS	-	1 UDP, 33953 1 TCP with client, 34953
Client	1 TCP	

ON SCREEN MESSAGES

Table 3. Backend-Server A on screen messages

Event	On Screen Message
Booting up (Only while starting):	The Server A is up and running using UDP on port <server A port number>
For graph finding, upon receiving the input query:	The Server A has received input for finding graph of map <ID>
For graph finding, no graph found	The Server A does not have the required graph id <graph id>
For graph finding, no graph found after sending to AWS:	The Server A has sent "Graph not Found" to AWS
For graph finding, after sending to AWS:	The Server A has sent Graph to AWS

Table 4. Backend-Server B on screen messages	
Event	**On Screen Message**
Booting up (Only while starting):	The Server B is up and running using UDP on port <server B port number>
For graph finding, upon receiving the input query:	The Server B has received input for finding graph of map <ID>
For graph finding, no graph found	The Server B does not have the required graph id <graph id>
For graph finding, no graph found after sending to AWS:	The Server B has sent "Graph not Found" to AWS
For graph finding, after sending to AWS:	The Server B has sent Graph to AWS

Table 5. Backend-Server C on screen messages	
Event	**On Screen Message**
Booting up (Only while starting):	The Server C is up and running using UDP on port <server C port number>
For calculation, after receiving data from AWS:	The Server C has received data for calculation: * Propagation speed: <speed1> km/s; * Transmission speed <speed2> KB/s; * map ID: <ID>; * Source ID: <ID> Destination ID: <ID>;
After calculation:	The Server C has finished the calculation: Shortest path: <src> -- <hop1> -- <hop2> … -- <dest> Shortest distance: <dist> km Transmission delay: <delay1> s Propagation delay: <delay2> s
Sending the results to the AWS server:	The Server C has finished sending the output to AWS

Table 6. AWS on screen messages	
Event	**On Screen Message**
Booting up (only while starting):	The AWS is up and running.
Upon Receiving the input from the client:	The AWS has received map ID <map ID>, start vertex <vertex ID>, destination vertex <vertex ID> and file size <file size> from the client using TCP over port <AWS TCP port number>
After sending information to server A	The AWS has sent map ID to server A using UDP over port <AWS UDP port number>
After sending information to server B	The AWS has sent map ID to server B using UDP over port <AWS UDP port number>
After receiving results from server A or B	The AWS has received map information from server <A/B>
Check nodes in graph: src and dst in graph	The source and destination vertex are in the graph
Check node in graph: vertex not in graph	<Source/destination> vertex not found in the graph, sending error to client using TCP over port <AWS UDP port number>
After sending information to server C	The AWS has sent map, source ID, destination ID, propagation speed and transmission speed to server C using UDP over port <AWS UDP port number>
After receiving results from server C	The AWS has received results from server C: Shortest path: <src> -- <hop1> -- <hop2> … -- <dest> Shortest distance: <dist> km Transmission delay: <delay1> s Propagation delay: <delay2> s
After sending results to client	The AWS has sent calculated results to client using TCP over port <AWS UDP port number>

Table 7. Client on screen messages	
Event	**On Screen Message**
Booting Up:	`he client is up and running`
After sending query to AWS	`he client has sent query to AWS using TCP: start vertex <vertex index>; destination vertex <vertex index>, map <map ID>; file size <file size>`

After receiving output from AWS	`he client has received results from AWS:`
	`----------`
	`ource Destination Min Length Tt Tp Delay`
	`----------`
	`0 1 0.10 0.10 0.10 0.20`
	`----------`
	`hortest path: <src> -- <hop1> -- <hop2> … -- <dest>`

After receiving output AWS, errors	`o map id <mad id> found`
	
	`r`
	
	`o vertex id <vertex index> found`

About the Makefile:

Makefile should support following functions:

Compiles **all** your files and creates executables	`make all`
Runs server A	`make serverA`
Runs server B	`make serverB`
Runs server C	`Make serverC`
Runs AWS	`make aws`
Query the AWS	`./client <Map ID> <Source Vertex Index> <Destination Vertex Index> <File Size>`

The processes start in this order: **backend-server (A), backend-server (B), backend-server (C), AWS, and Client.**