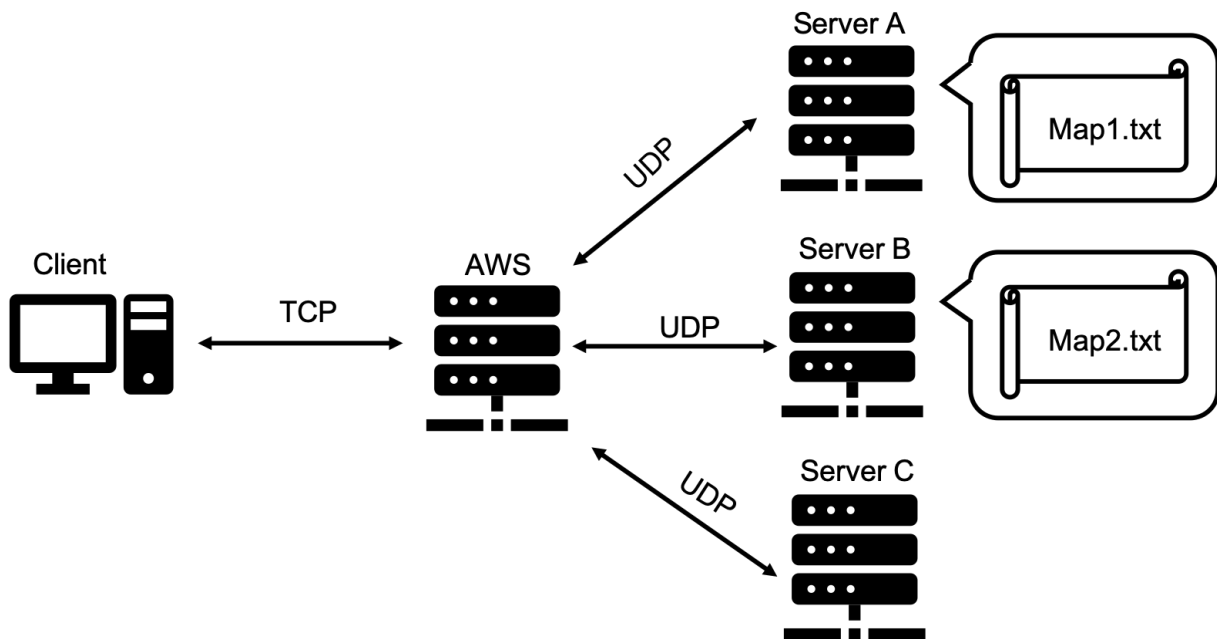


## PROBLEM STATEMENT

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. This algorithm will be discussed later in the semester.

In this project, you 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 and destination node in the map and the transmission file size (unit: KB) to AWS via TCP.

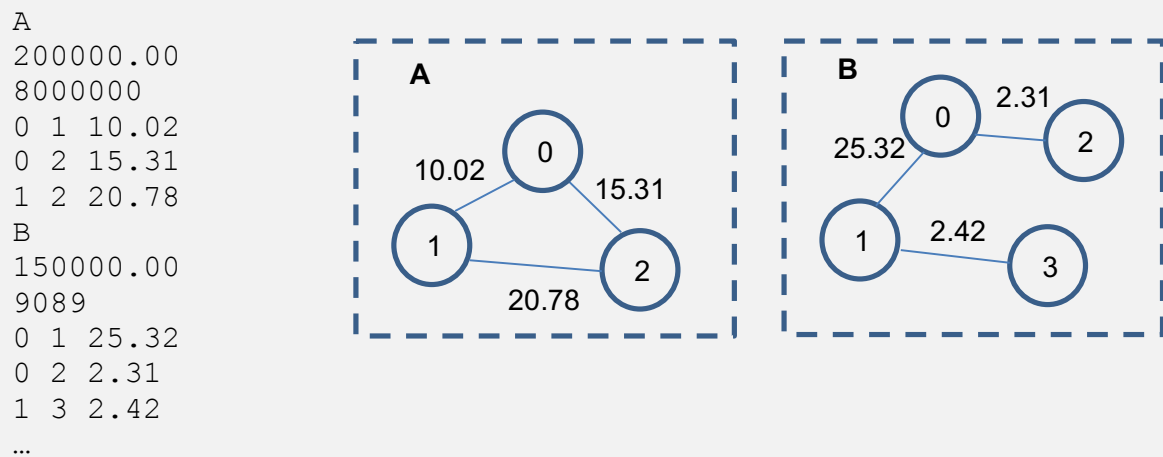
2. [Communication] AWS -> Server A: AWS forwards the map ID to server A via UDP.
3. [Search] Server A searches for the map ID from map1.txt.
4. [Communication] Server A -> AWS: Server A sends the search result via UDP.
5. [Communication] AWS -> Server B: AWS forwards the map ID to server B via UDP.
6. [Search] ServerB searches for the map ID from map2.txt.
7. [Communication] Server B -> AWS: Server B sends the search result via UDP.
8. [Communication] AWS -> ServerC: 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 final 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:



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)

We provide a sample map1.txt and map2.txt for you as a reference. To download the sample maps, please refer to the **DOWNLOAD SAMPLE MAPS** Section. We will use another map.txt for grading, so you are advised to prepare your own map files for testing purposes.

### Source Code Files

Your implementation should include 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. You must name your code file: **aws.c** or **aws.cc** or **aws.cpp** (all small letters). Also you must name the corresponding header file (if you have one; it is not mandatory) **aws.h** (all small letters).
2. **Backend-Server A, B and C**: You must use one of these names for this piece of code: **server#.c**

or **server#.cc** or **server#.cpp** (all small letters except for #). Also you must name the corresponding header file (if you have one; it is not mandatory) **server#.h** (all small letters, except for #). The “#” character must be replaced by the server identifier (i.e. A or B or C).

3. Client: The name for this piece of code must be **client.c** or **client.cc** or **client.cpp** (all small letters) and the header file (if you have one; it is not mandatory) must be called **client.h** (all small letters).

Note: Your compilation should generate separate executable files for each of the components listed above.

## **DETAILED EXPLANATION**

### **Phase 1 (10 points)**

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 (80 points)

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 in the event that 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.

You are not required to have exact named functions (map construction, path finding and calculation) in your code. These operations are named and divided to make the process clear.

## Phase 2A (40 points)

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, or 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 (40 points)

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.

Note:

You should decide on your own what information is required to be sent from AWS to server C. Similarly, it is your own choice what format / data structure to encode the AWS-to-server C and server C-to-AWS communication. We will grade based on your print out message only.

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,  $T_{trans} = \text{File size} / \text{transmission rate}$ , 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.

If the client’s query is valid, both server C and AWS should print out the calculation result so that we can give you partial credit if the server C-to-AWS or AWS-to-client communication fail. The on-screen message should indicate the vertex indices along the path, the total distance of the shortest path and the delays. Please format your output so the table is clear and readable.

### Phase 3 (10 points)

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.

Make sure you round the results of three delay time to the 2<sup>nd</sup> decimal point for display. Round the result **after** summing  $T_{trans}$  and  $T_{prop}$  along a path. Do not sum rounded  $T_{trans}$  and rounded  $T_{prop}$  as your total delay.

See the ON SCREEN MESSAGES table for an example output table.



## **DOWNLOAD SAMPLE MAPS**

Samples of map1.txt and map2.txt for this project are available online for download. The data in these map.txt files are generated randomly for each download and vary in size, but the structure and data type of the map.txt files are consistent. map1.txt and map2.txt are expected to be read and stored into serverA and serverB respectively.

Download a copy of map.txt files here:

<http://ee450project.us-west-1.elasticbeanstalk.com/>

map1.txt and map2.txt will be archived into a single maps.zip for download. Unzip and put the files in your program directory before using it.

There are no download limits, but both map1.txt and map2.txt should be used together. Do not mix map1.txt and map2.txt from different maps.zip downloads!

Similar map.txt files with different contents will be used for grading, but you are expected to read these files without errors.

## **PORT NUMBER ALLOCATION**

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

<b>Table 2. Static and Dynamic assignments for TCP and UDP ports</b>		
<b>Process</b>	<b>Dynamic Ports</b>	<b>Static Ports</b>
Backend-Server (A)	-	1 UDP, 30xxx
Backend-Server (B)	-	1 UDP, 31xxx
Backend-Server (C)	-	UDP, 32xxx
AWS	-	1 UDP, 33xxx 1 TCP with client, 34xxx
Client	1 TCP	-

**NOTE:** xxx is the last 3 digits of your USC ID. For example, if the last 3 digits of your USC ID are “319”, you should use the port: 30319 for the Backend-Server (A), etc.

Port number of all processes print port number of their own

### 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 TCP 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 TCP port number>

Table 7. <b>Client</b> on screen messages	
Event	On Screen Message
Booting Up:	The client is up and running
After sending query to AWS	The 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	<p>The client has received results from AWS:</p> <pre> ----- Source Destination      Min Length      Tt      Tp      Delay ----- 0          1          0.10          0.10  0.10  0.20 ----- Shortest path: &lt;src&gt; -- &lt;hop1&gt; -- &lt;hop2&gt; ... -- &lt;dest&gt; </pre>
After receiving output AWS, errors	<p>No map id &lt;mad id&gt; found</p> <p>or</p> <p>No vertex id &lt;vertex index&gt; found</p>

## ASSUMPTIONS

1. You have to start the processes in this order: **backend-server (A), backend-server (B), backend-server (C), AWS, and Client.**
2. The map1.txt and map2.txt files are created before your program starts.
3. If you need to have more code files than the ones that are mentioned here, please use meaningful names and all small letters and **mention them all in your README file.**
4. You are allowed to use code snippets from Beej's socket programming tutorial (Beej's guide to network programming) in your project. However, you need to mark the copied part in your code.
5. When you run your code, if you get the message "port already in use" or "address already in use", **please first check to see if you have a zombie process** (see following). If you do not have such zombie processes or if you still get this message after terminating all zombie processes, try changing the static UDP or TCP port number corresponding to this error message (all port numbers below 1024 are reserved and must not be used). If you have to change the port number, **please do mention it in your README file and provide reasons for it.**
6. You may create zombie processes while testing your codes, please make sure you kill them every time you want to run your code. To see a list of all zombie processes, try this command:

```
ps -aux | grep developer
```

Identify the zombie processes and their process number and kill them by typing at the command-line:

```
kill -9 <process number>
```

## REQUIREMENTS

1. Do not hardcode the TCP or UDP port numbers that are to be obtained dynamically. Refer to Table 3 to see which ports are statically defined and which ones are dynamically assigned. Use getsockname() function to retrieve the locally-bound port number wherever ports are assigned dynamically as shown below:

```

/*Retrieve      the locally-bound name of the specified
      socket and store it in the sockaddr structure*/
getsock_check=getsockname(TCP_Connect_Sock,(struct sockaddr
*)&my_addr, (socklen_t *)&addrlen);
//Error checking
if (getsock_check== -1) { perror("getsockname"); exit(1);
}

```

2. The host name must be hard-coded as **localhost (127.0.0.1)** in all codes.
3. Your client should terminate itself after all done. And the client can run multiple times to send requests. However, the backend servers and the AWS should keep running and be waiting for another request until the TAs terminate them by Ctrl+C. If they terminate before that, you will lose some points for it.
4. All the naming conventions and the on-screen messages must conform to the previously mentioned rules.
5. You are not allowed to pass any parameter or value or string or character as a command-line argument except while running the client in Phase 1.
6. All the on-screen messages must conform exactly to the project description. You should not add anymore on-screen messages. If you need to do so for the debugging purposes, you must comment out all of the extra messages before you submit your project.
7. Please do remember to close the socket and tear down the connection once you are done using that socket.

### Programming Platform and Environment

1. All your submitted code **MUST** work well on the provided virtual machine Ubuntu.