FACULTY OF ENGINEERING

Incremental calculation of shortest path in dynamic graphs

Distributed Systems

Presented by
Ahmed Magdy (07)
Mohamed Gaber (56)
Hesham Medhat (71)
05/04/2020

Table of Contents

1 Problem Definition

The task is to answer the shortest path queries on a changing graph, as quickly as possible by applying the Client-Server Model to solve the graph shortest path problem using java RMI (Remote Method Invocation).
Client-Server model is a distributed application structure that partitions tasks or workloads between the providers of a resource or service, called **servers**, and service requesters, called **clients**. Often clients and servers communicate over a computer network on separate hardware, but both client and server may reside in the same system.

Moreover, another variant or method to enhance the response time should be implemented while reporting sufficient performance analysis.

2 Algorithms

We use BFS to compute the shortest path between 2 nodes in a graph.

```
        srcEdges.remove(dest);
        clearDP();
    }


    private int minEdgeBFS( int u, int v) {
        // Initialize distances as 0
        HashMap<Integer, Integer> distance = dpMap.getOrDefault(u, new HashMap<Integer, Integer>());
        distance.put(u, 0);

        // queue to do BFS.
        Queue<Integer> Q = new LinkedList<>(distance.keySet());

        while (!Q.isEmpty()) {
            int x = Q.peek();
            Q.poll();

            for (Integer dest : edges.getOrDefault(x, new HashSet<Integer>())) {
                if (distance.containsKey(dest))
                    continue;

                int newDist = distance.get(x) + 1;
                distance.put(dest, newDist);// update distance for i
                Q.add(dest);
            }
        }
        // Keep for the future
        memoResult(u, distance);
        return distance.getOrDefault(v, -1);
    }

```

As a performance improvement variation, we apply a memoization (Dynamic Programming) technique by storing the computed shortest paths and use them in the future to answer further queries if they are already calculated.
This method relies heavily on the fact that in practice the update operations to the graph should be a rare operation that doesn't happen often.

```
private Integer lookupDP(int source, int destination) {
    HashMap<Integer, Integer> resultMap = dpMap.get(source);
    return resultMap != null ? resultMap.getOrDefault(destination, null): null;
}

private void memoResult(int source, HashMap<Integer, Integer> result) {
    dpMap.put(source, result);
}
```

Check GitHub repo for the full code.

3 Implementation

We represented the original graph as :

HashMap<Integer, HashSet<Integer>> edges = new HashMap<Integer, HashSet<Integer>>();

There are three operations that can be applied to the original graph. The three operation types are as follows:

'Q'/query: This operation needs to be answered with the distance of the shortest (directed) path from the first node to the second node in the current graph. The answer should appear as output in the form of a single line containing the decimal ASCII representation of the integer distance between the two nodes, i.e., the number of edges on a shortest directed path between them. If there is no path between the nodes or if either of the nodes does not exist in the graph, the answer should be -1.

'**A'/add:** This operation requires you to modify your current graph by adding another edge from the first node in the operation to the second. As was the case during the input of the original graph input, if the edge already exists, the graph remains unchanged. If one (or both) of the specified endpoints of the new edge does not exist in the graph, it should be added.

'D'/delete: This operation requires you to modify your current graph by removing the edge from the first node in the operation to the second. If the specified edge does not exist in the graph, the graph should remain unchanged.

RMI

The **RMI** (Remote Method Invocation) is an API that provides a mechanism to create distributed application in java. The RMI allows an object to invoke methods on an object running in another JVM.

Structure:

GraphSolverRMI/server/GraphProcessingI.java (Interface: **extends java.rmi.Remote**)
/GraphProcessing.java (**extends UnicastRemoteObject implements GraphProcessingI**)
/Server.java (Server Launcher)
/client/Client.java (Client program)

/Start.java (Starts server , and clients as processes)

An RMI server registers its remote objects by using bind() or rebind() method of Naming class.

The clients obtain this reference by using lookup() method of Naming class.

SSH

To start a process on a remote machine, we should use the remote login facility **SSH** in Unix systems.

SSH is a security protocol that can be used for end point authentication, message integrity and confidentiality.

To use **SSH** we will need two machines: one to run the client program and another to run the RMI Registry and server program. Both machines should have J2SE SDK. The client machine should have SSH client and the server machine should have SSH daemon. For testing purposes, it would be good if we are able to close all ports other than the port 9000, where SSH daemon listens for incoming connections, on the server machine.

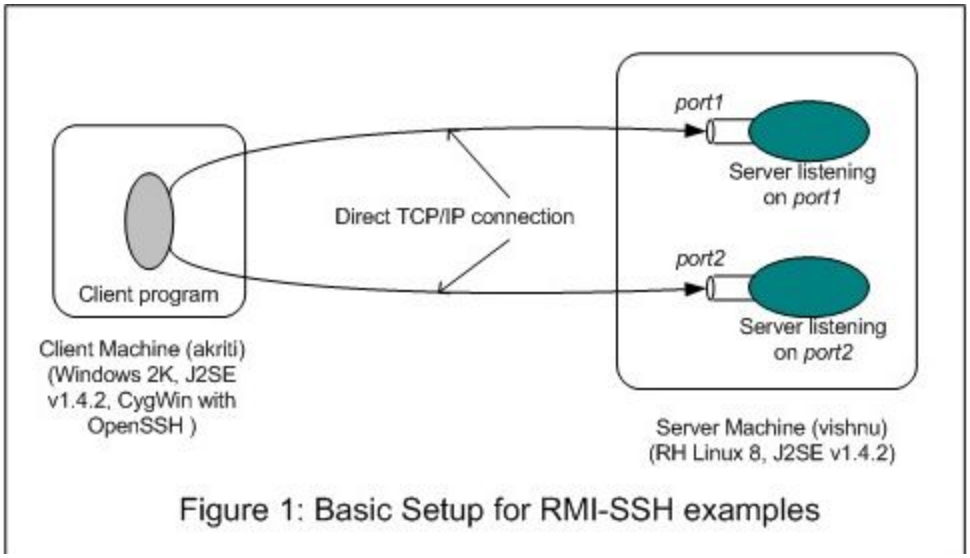


Figure 1: Basic Setup for RMI-SSH examples

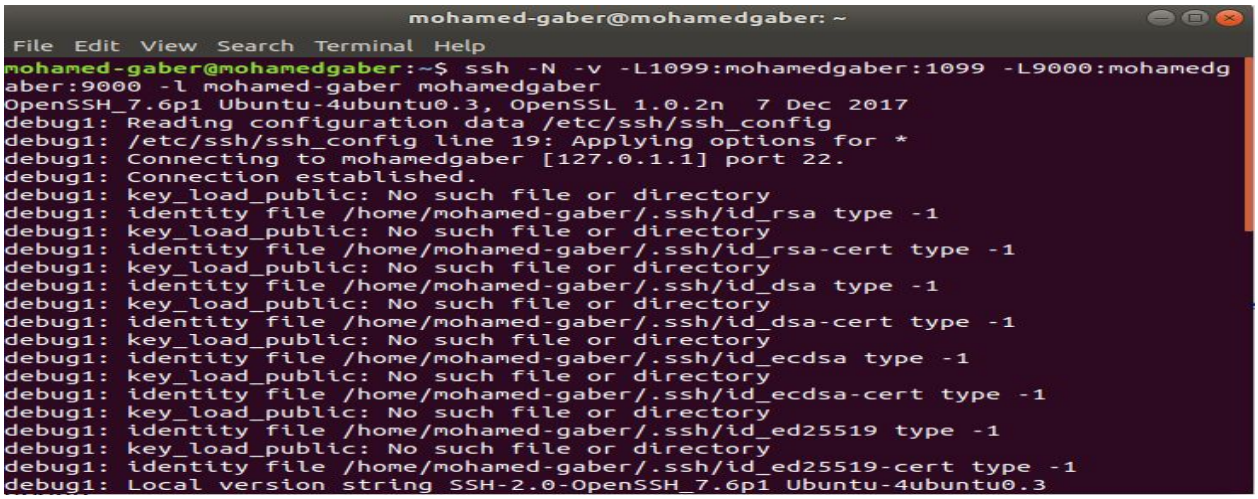
To create SSH tunnel for RMI calls, we need to forward two ports:

1. Port 1099 where **rmiregistry** waits for incoming connection.
2. The port at which the RMI server program waits for incoming connection. (port 9000)

We assigned a fixed port to the RMI server program by adding an extra constructor to the GraphProcessing.java .

```
public GraphProcessing(int port) throws RemoteException {
        super(port);
}
```

To set up the tunnel for ports 1099 (for RMI registry) and 9000 (for RMI server), issue the following command on the client machine:



Synchronization

Synchronization is achieved using a lock. A lock is used to secure the object data (edges) during a modifying operation such as calculating the shortest path or altering the graph by adding or removing edges.

```
if(operation == 'Q') {
    shortestPath = this.memo? lookupDP(execSrc, execDest): null;
    if(shortestPath == null) {
        lck.lock();
        shortestPath = query(execSrc,execDest);
        lck.unlock();
    }
    outputList.add(shortestPath);
} else if (operation == 'A') {
    lck.lock();
    addEdge(execSrc, execDest);
    lck.unlock();
} else if (operation == 'D') {
    lck.lock();
    deleteEdge(execSrc, execDest);
    lck.unlock();
} else {
    throw new UnsupportedOperationException("Unsupported query type: " + operation);
}
```

Logging

- Client and Server logs their logging on the format timestamp=<>, operation=op, src=<>, dest=<>, result=<>, latency=<>, where:

timestamp = The time of the request.
operation = “Q\A\D” for reporting single requests and “B” for the whole batch reporting.
src, dest are the source and destination nodes (only for client)
result = The shortest path between src and dest for Q operations.
latency = The time taken to receive a response from the server in ms.

- Server logs his logging on the format:

4 Results

Testing has some fixed criteria:

- 1000 nodes
- 10000 edges
- 10000 Shortest Path queries
- (0.2/2/20)% Add/Delete queries
- 5/10 clients
- BFS without Memoization vs BFS with Memoization

1. Testing with 5 Clients

a. Frequency vs Response Time

0.2% Add/Delete queries

	Without Memoization	With Memoization
No-Waiting	1.83	0.74
Random [0, 10] ms waiting	0.82	0.62
Random [0, 25] ms waiting	0.83	0.53
Random [0, 50] ms waiting	0.80	0.52

2% Add/Delete queries

	Without Memoization	With Memoization
No-Waiting	1.84	1.73
Random [0, 10] ms waiting	0.84	0.83
Random [0, 25] ms waiting	0.86	0.83
Random [0, 50] ms waiting	0.87	0.85

20% Add/Delete queries

	Without Memoization	With Memoization
No-Waiting	1.86	1.84
Random [0, 10] ms waiting	0.83	0.83
Random [0, 25] ms waiting	0.85	0.86
Random [0, 50] ms waiting	0.86	0.85

Increasing time between requests decreases the response time because of less congestion on the network.

b. Add/Delete percentage vs Response Time

No-Waiting

	Without Memoization	With Memoization
0.2%	1.83	0.74
2%	1.84	1.73
20%	1.86	1.84

Random [0, 10] ms waiting

	Without Memoization	With Memoization
0.2%	0.82	0.46
2%	0.84	0.83
20%	0.83	0.83

Random [0, 25] ms waiting

	Without Memoization	With Memoization
0.2%	0.83	0.53

2%	0.86	0.83
20%	0.85	0.86

Random [0, 50] ms waiting

	Without Memoization	With Memoization
0.2%	0.80	0.52
2%	0.87	0.85
20%	0.86	0.85

Increasing update queries increases the response time because doesn't affect the response time but reduces the effectiveness of the memoization

2. Testing with 10 Clients

a. Frequency vs Response Time

0.2% Add/Delete queries

	Without Memoization	With Memoization
No-Waiting	3.43	1.05
Random [0, 10] ms waiting	0.98	0.63
Random [0, 25] ms waiting	0.82	0.52
Random [0, 50] ms waiting	0.79	0.45

2% Add/Delete queries

	Without Memoization	With Memoization
No-Waiting	4.51	4.57
Random [0, 10] ms waiting	1.28	1.24
Random [0, 25] ms	1.04	1.03

waiting		
Random [0, 50] ms waiting	1.01	1.01

20% Add/Delete queries

	Without Memoization	With Memoization
No-Waiting	4.51	4.57
Random [0, 10] ms waiting	1.28	1.24
Random [0, 25] ms waiting	1.04	1.03
Random [0, 50] ms waiting	1.01	1.01

b. Add/Delete percentage vs Response Time

No-Waiting

	Without Memoization	With Memoization
0.2%	3.43	1.05
2%	3.86	3.63
20%	4.51	4.57

Random [0, 10] ms waiting

	Without Memoization	With Memoization
0.2%	0.98	0.43
2%	1.02	0.93
20%	1.28	1.24

Random [0, 25] ms waiting

	Without Memoization	With Memoization
0.2%	0.82	0.52
2%	0.88	0.85
20%	1.04	1.03

Random [0, 50] ms waiting

	Without Memoization	With Memoization
0.2%	0.79	0.45
2%	0.89	0.85
20%	1.01	1.01

In general, increasing the number of clients increases the response time because of more congestion on the network.