Introduction to Computer Networking

Distance-Vector Computer Assignment

Roozbeh Sayadi - Amirali Monjar

Introduction

Distance-Vector is an iterative, asynchronous, and distributed algorithm. It is *iterative* because the algorithm continues until it sees no changes. It is asynchronous because the nodes do not need to require to operate in lockstep with each other. And it is distributed because each node does the calculations for itself and no centralized calculations are happening. It is used to find the minimum cost between all nodes in a graph, and it is based on the celebrated Bellman-Ford equation, which is:

$$d_x(y) = min_v\{c(x, v) + d_v(y)\}$$

Where the min_v in the equation is taken over all of x's neighbors.

Its use in networking is to build the forwarding table of routers, considering each router as a node in the network graph.

In the rest of the report, we'll show our implementation of this algorithm, written in Java.

How to run

You have to run Node0, Node1, Node2, and Node3 in this order. Also, keep in mind that to compile the codes, you'll need the Gson's jar file.

Code Explanation

Classes

Before we dive into the details, let's see what entities are defined in the project and what each of them represents.

Node

This class represents the common behaviors of routers.

Node0, Node1, Node2, Node3

As I said, the *Node* class only implements the common behaviors of routers. It also defines (but not implements) the specific behaviors that should be implemented in each router. Implementation of these functions is done in these 4 classes.

Packet

Represents a packet, containing source number, destination number, and packet contents (in this case, distance-vector).

DVReaderThread

A thread that helps a node read its inputs.

Additional tools used

Gson

Gson is a library for converting Java objects to JSON format as a string.

Classes Implementation

Now, the detailed implementation of each class.

Node

Attributes

This class has 5 attributes:

```
int index;
```

The number of the router.

```
Map<Integer, Integer> DV;
```

Distance-vector of the router. Key values are the number of a router, and its corresponding value is the calculated minimum cost to that router.

```
Map<Integer, Integer> directCost;
```

A mapping of the router number and direct link cost. If a link doesn't exist between two nodes, its cost will be ∞.

```
Map<Integer, Map<Integer, Integer>> neighborDV;
```

Keeps the distance-vector of the router's neighbors. As you saw earlier, distance-vector is a mapping from integers to integers, so this attribute should be a mapping of integers to distance-vectors. At first, all values of distance-vectors are ∞.

```
Map<Integer, Socket> sockets;
```

Maps neighbor routers' numbers to their sockets.

Methods

This class has the following methods:

```
Node( int index ) throws Exception {
    directCost = new HashMap<>();
    DV = new HashMap<>();
    neighborDV = new HashMap<>();
    sockets = new HashMap<>();
    this.index = index;

    rinit();
    for ( Map.Entry<Integer, Socket> e : sockets.entrySet() )
        new Thread( new DVReaderThread( this, e.getKey() ) ).start();
}
```

Constructor for the class. Gets the index of the router as its arguments.

The first four lines are for making the attributes ready. The 5th line sets the value of the index.

After these, rinit() is called. See its description to see what does it do and how.

When we reach this line, we know that the sockets are ready and connections are established, so for each neighbor, we run a thread to do some operations if they send data to us.

```
void rinit() throws Exception {
    initDirectCost();
    initDV();
    initNeighborDV();
    initSockets();
    sendDVToNeighbors();
    System.out.println( "DV After rinit in node " + this.index + ": " +
```

```
(new Gson()).toJson( this.DV ) );
}
```

This function does the initializations of the algorithm. As you can see, this function calls some other functions to do the initializations. We'll see what these functions do.

```
void rUpdate( Packet pkt ) throws Exception {
    boolean flag = false;
    synchronized ( this ) {
        this.neighborDV.replace( pkt.source, pkt.DV );
        for ( int i = 0; i < 4; i++ ) {
            int previousValue = this.DV.get( i );
            for ( Map.Entry<Integer, Map<Integer, Integer>> e :
neighborDV.entrySet() ) {
                int v = e.getKey();
                int temp = (int) Math.min( Integer.MAX_VALUE, (long)
directCost.getOrDefault( v, Integer.MAX_VALUE ) + e.getValue().get( i ) );
                this.DV.replace( i, Math.min( this.DV.get( i ), temp ) );
            }
            System.out.println( "DV After update in node " + this.index +
": " + (new Gson()).toJson( this.DV ) );
            if ( previousValue > this.DV.get( i ) )
                flag = true;
        }
    }
    if ( flag )
             sendDVToNeighbors();
}
```

This is a rather complicated function. Its purpose is to get a packet as an argument, replace the distance-vector of the respective neighbor, and update the router's distance-vector according to the changes.

First, we replace the old distance-vector of the neighbor with the new one.

The first loop is for checking all of the nodes and update the minimum cost to them if necessary. To achieve this, first, the current value of minimum cost is retrieved. Then we

look through all the distance-vectors of the neighbors and check if a better path exists to that neighbor.

If any changes are made, the flag is set to true. At the end of the function, we check if the value of the flag is true. If so, we send the updated distance-vector to all of the neighbors.

```
void sendDVToNeighbors() throws IOException {
    Packet pkt = new Packet();
    pkt.source = this.index;
    for ( Map.Entry<Integer, Socket> i : sockets.entrySet() ) {
        pkt.destination = i.getKey();
        synchronized (this) {
            pkt.DV = DV;
            DataOutputStream temp = new DataOutputStream(
i.getValue().getOutputStream() );
            Gson gson = new Gson();
            temp.writeUTF(gson.toJson(pkt));
        }
    }
}
```

This function sends the router's distance-vector to all its neighbors.

First, the packet that's going to be sent is defined and its source is initialized.

Then the loop iterates through all the neighbors, sets the destination of the packet, and sends the packet through the stream of that neighbor.

```
void initDV() {
    for ( int i = 0; i < 4; i++ )
        DV.put( i, directCost.getOrDefault( i, Integer.MAX_VALUE ) );
}
```

This function initializes the distance-vector of the router for the first time. For each neighbor, it either puts the cost of the direct link or ∞ if no direct link is available.

```
protected abstract void initNeighborDV();
protected abstract void initSockets() throws IOException;
protected abstract void initDirectCost();
```

These abstract functions have no implementation. Their implementation differs based on the router they are running for, so each router has to implement this themself.

Node0

This class represents the router 0. It is extended from the Node class.

Attributes

It has no attributes as all possible attributes are common between all routers, therefor they are declared in the *Node* class.

Methods

```
private Node0() throws Exception {
    super( 0 );
}
```

It calls its upper constructor (constructor of the Node class) with its own index.

```
@Override
protected void initSockets() throws IOException {
    ServerSocket serverSocket = new ServerSocket( 3000 );
    sockets.put( 1, serverSocket.accept() );
    sockets.put( 2, serverSocket.accept() );
    sockets.put( 3, serverSocket.accept() );
}
```

This function sets up a serversocket for itself with port number 3000 and waits for other nodes to start and connect to it.

```
@Override
protected void initDirectCost() {
    directCost.put( 0, 0 );
```

```
    directCost.put( 1, 1 );
    directCost.put( 2, 3 );
    directCost.put( 3, 7 );
}
```

This function initializes the direct costs for the router's neighbors. These numbers are from the problem's graph.

```
@Override
protected void initNeighborDV() {
    Set<Integer> neighbors = new HashSet<>();
    neighbors.add( 1 ); neighbors.add( 2 ); neighbors.add( 3 );
    for ( Integer i : neighbors ) {
        neighborDV.put( i, new HashMap<>() );
        for (int j = 0; j < 4; j++) {
            neighborDV.get(i).put(j, Integer.MAX_VALUE);
        }
    }
}
```

This function initializes the distance-vector of the neighbors inside the router. At first, they are all filled with ∞.

```
public static void main(String[] args) throws Exception {
    new Node0();
}
```

This is the main function. It creates an instance of Node0.

Node1, Node2, Node3

These classes are very similar to the Node0 class, so we won't write their code here to keep the report short.

Packet

This class represents a packet.

Attributes

```
protected Map<Integer, Integer> DV;
protected int source;
protected int destination;
```

Distance-vector, index of the source router, and index of the destination router respectively.

Methods

This class has no methods. It is only used to transfer data between routers.

DVReaderThread

Each router has one object of this class for each of its sockets, to read data from them. As you can see from the name of the class, it is a thread and runs simultaneously with the main thread the of program.

Attributes

```
private Node node;
```

The node that created this thread.

```
private DataInputStream dis;
```

The input stream of the corresponding socket is used to read data.

Methods

```
public DVReaderThread( Node node, int socketIndex ) throws Exception {
    this.node = node;
    this.dis = new DataInputStream( node.sockets.get( socketIndex
).getInputStream() );
}
```

Constructor of the thread. Gets the node and index of the socket. Stores node for later use, and uses the socketIndex to get the considered socket of the node and store its input stream.

```
@Override
public void run() {
    Gson gson = new Gson();
    while ( true ) {
        String temp = null;
        try {
            temp = this.dis.readUTF();
        } catch (EOFException e) {
            System.out.println( "A socket is closed unexpectedly.
Exiting..." );
            System.exit( 1 );
        }catch (IOException e) {
            e.printStackTrace();
            continue;
        }
        Packet pkt = gson.fromJson( temp, Packet.class );
        try {
            node.rUpdate( pkt );
        } catch (Exception e) {
            System.out.println( "Exception in rUpdate" + e );
            System.exit( 1 );
        }
    }
}
```

This function is the function that runs simultaneously with the main thread of the program and has the duty of reading any inputs from the socket and pass them to the node to update its distance-vector.

Ignoring the exception handling parts, we have an infinite while loop that reads from the input stream, converts the input to a packet instance, and calls the rUpdate function for its Node with its new information.