**BRIEF PROGRAM DESCRIPTION**

**OVERVIEW**

This is a simple banking account system that permits users to transfer money from one of their accounts to the desiderated account. Each account corresponds to a node in a network following the *asynchronous network model* and communication between nodes is established by *TCP sockets.* In order to transfer money from an account (a node) to another two different distributed algorithms are used. The first one is the asynchronous version of *Bellman Ford* algorithm *()* and permits to calculate the *shortest path* between sender and receiver accounts. The second () one is a simple algorithm that transfers money from the sender account to the receiver along the path calculated by .

**NETWORK MODEL**

I start describing the network model adopted for this algorithm, that is called the *asynchronous network model*. The *asynchronous* network is a point-to-point (store-and-forward) communication network, described by an undirected communication graph (V, E) , where the set of nodes V represents processors of the network and the set of links E represents bidirectional noninterfering communication channels operating between them. No common memory is shared by the node’s processors, and each node has a distinct identity (ID). Each node processes messages received from its neighbors, thing that involves performing local computations and sending messages to its neighbors. All these actions are assumed to be performed in a negligible time compared to the time taken by a sent message to reach its destination. All the messages have a fixed length and may carry only a bounded amount of information. Each message sent by a node to its neighbor arrives within some finite but unpredictable time. When a node receives a message, it stores it at the end of a message queue following FIFO logic. Therefore when the node has done with the computation associated to the current message, it extracts the next message from the head of the queue and processes it. In this way messages can accumulate at a node. In addition, each edge is labeled with a positive integer representing the *cost of communication* (or *weight*) that may have a physical meaning, such as the inverse of bandwidth. When the node wants to communicate with the node , it sends a message along the path connecting to ; the *total cost* of the path is the sum of the costs of edges composing the path. The *shortest path* between and is the path with minimum *total cost* among all the paths joining the two nodes.

Communication between nodes is established by *TCP sockets*. If node wants to send message to node , it needs to instantiate a *client* TCP socket, whereas node , that it is supposed to receive messages from , has to instantiate a *server* TCP sockets. Therefore if a node has N neighbors it will instantiate N client sockets to send messages and N server sockets to receive messages. In order to manage this form of communication each node has to instantiate a *server thread* (*Handler* class in implementation) to handle incoming connections and messages. This thread establishes connections with neighbor nodes, reads the messages sent to it and puts the messages in the message queue. The *client thread* (represented by *Node* class itself in implementation), on the other hand, extracts a message from the head of message queue (if it is not empty) and processes it.

**Algorithm**

algorithm permits to compute all the shortest paths joining a specific node (*source* node) to all the other node in the network. Let me explain it with a practical example.

Consider the network represented by the following adjacency matrix:

Each row of the matrix contains the weights of the outgoing edges of a given node. For example the first row is related to the node indexed with . This node has outgoing edges having weights and leading to nodes and 5 respectively. Following the same reasoning for all the nodes, it is possible to draw the graph:

9

2

1

5

4

3

6

**Fig. 1:** Starting situation.

The black arrow indicates the *source* node. Node 0, triggered by a signal from the outer world, starts the algorithm sending *update* messages to all its neighbors. *Update* messages contain the *distance* of the sender node from the *source* node. In present situation node 0 sends *update* messages containing *distance* 0 to nodes 1, 5 and 2. In figures, arrows indicate the *update* messages and the number above the nodes, the current *distance*. At the beginning of the algorithm all the *distances*, except for *source* node (set to 0), are set to infinite.

2

6

3

1

4

5

9

(0)

0

(0)

(0)

**Fig. 2:** Node 0 sends its *update* messages.

When a node receives an *update* message, it sums the *distance* contained in the message with the weight of the edge connecting it to the *sender* node (the node that had sent the *update* message) and compares that quantity with the value assumed by its own *distance* variable. If the latter quantity is higher than the former, it updates its *distance* with the former quantity, stores the index of *sender* node in a variable named *parent* and sends *distance* to all its neighbors except for the *sender* node. Therefore, node 1 sends an *update* message containing *distance* 2 to node 1, node 5 sends an *update* message with *distance* 3 to node 4, and node 2 a message with *distance 6* to node 3 (Fig. 3). Now, two messages are addressed to node 4; let us say that the message from node 1 arrives first. In this case node 4 updates its *distance* to 11 (2 + 9) and send two messages containing that quantity to node 5 and 3. Contemporary, node 3 processes the message coming from node 2 and therefore corrects its *distance* to 11 (6 + 5) and sends it to node 4 (Fig. 4).

i

2

6

3

1

4

5

9

0

(6)

(2)

(3)

**Fig. 3:** Nodes 1,5, 2 process the *update* messages received from node 0.

The *update* messages towards nodes 3, 4 and 11 have no effect since the *distance* they carried is higher or equal the *distance* of their destination nodes. The message sent to node 4 eventually reaches its destination; node 4, as a consequence, adjusts its distance to 4 and send two *update* messages to nodes 1 and 3 (Fig. 5). The message sent to node 3 has no consequences, whereas the one that reaches node 3 makes it update its *distance to* 8 and send two messages to nodes 4 and 2 (Fig. 6). Finally, these latter messages reach their objectives but have no effect.

A problem with is that there is no way for a process to know when there are no further corrections for it to make. Thus, the algorithm so far presented is not technically a solution to the *shortest paths* problem, because it never produces the required *parent* outputs (as previously seen each node stores a variable named *parent* that is continuously updated and at the end of the algorithm should contain the index of the node included in the shortest path leading it to *source* node). The problem is resolved with a convergecast of messages back to the *source* node. This enables *source* node to learn when the system has reached a stable state and then broadcast a signal to all the processes to perform their *parent* outputs. This convergecast is a bit complicated, because a process may need to participate many times. Each time a process obtains a new *distance* estimate from neighbor and sends out corrections to all its other neighbors before sending an acknowledgement to . Bookkeeping is needed to keep the different set of acknowledgments separate. The details of this convergecast are in file Node.cpp.

2

6

3

1

5

9

2

0

6

11

11

(11)

(3)

3

4

(11)

(11)

11

**Fig. 4:** Node 4 processes the message from node 1 and node 3 processes the one fro node 6. The message from node 5 towards node 4 is not arrived at destination yet.

2

6

3

1

5

9

2

0

6

(4)

4

3

(4)

11

4

**Fig. 5:** Node 4 processes the *update* message from node 5.

2

6

3

1

5

9

0

3

2

4

6

8

4

(8)

(8)

**Fig. 6:** Node 3 processes the *update* message from node 4.

**Algorithm**

Once the shortest paths leading to a node have been calculated is possible to transfer money to the account represented by that node. The algorithm that carries out the task is very simple: the sending account sends a message across the path joining it to the receiving account. When the latter receives the message it sends back a message across the same path both to terminate the algorithm at a given node and to assure sender account that the transaction has been successful.

**Program workflow**

The workflow of the program is depicted below:

1. The user is asked to insert username and password to log in (passwords are contained in the file bin/banking\_system/passwords.txt).
2. A welcome message is displayed followed by a list of user’s banking accounts with the sum of the amounts present in the single accounts
3. The user is asked to insert the account number of the account from which she wants to transfer money and the amount.
4. The user is asked to insert the data about the receiving account: first name and family name of the owner and account number (data about all the accounts present in the system are contained in folder bin/banking\_system/Test/Input/Accounts\_data.
5. If the transaction is successful, the messages exchanged by the nodes involved in the two algorithms mentioned above are displayed followed by a message stating that the transaction is successful.
6. An updated list of user’s banking accounts is displayed with the sum of the amounts present in the single accounts.
7. The user is asked whether she wants to carry out another transaction. If the answer is positive the algorithm starts again at point 1.
8. If the answer is negative the algorithm terminate. A log file containing records about all the transactions performed is printed. That file also contains an updated list with the data of all the accounts present in the system (log files are contained in folder bin/banking\_system/Test/Output, sorted according to the adjacency matrix of the input graph).