Assignment 2

Due date: 13/1/22

In this assignment we will build an asynchronous single-threaded peer-2-peer doc-sharing system, written in JavaScript running on Node.js. The system is composed of a set of *clients,* one connected to each other. The clients modify locally a *replica of a shared string*, and send the *update operation* with a *time-stamp* to all other clients. Whenever a client gets an update operation with a time-stamp from another client, he updates its replica accordingly (by a simple *CRDT* algorithm described later). At the end, all clients should hold a same replica of the shared string.

The client

The client program is initialized with a unique id, a port number, a replica of the shared string (same for all clients), and a list of the clients in the system (defined by their id, host and port), and a list of string operations.

The client should first connect to the clients, according to the following scheme:

- The client connects via socket to clients with greater id.
- The client generates an http server (as shown in class, here at page 88) in order to accept connections from clients with lower id (if exists).
- Each of these sockets should be defined with a callback function for handling received data. The callback for this application should:
    - Update the client vector time-stamp (as shown in class, here at slide 13)
    - Apply the merge algorithm according to the received update operation.
- The client set its initial vector time-stamp (as shown in class, here at slide 13)
- From this point on the client run the following loop
    - For each operation
        - Add a task to the event loop, which:
            - Apply one string modification, as defined in the unput file
            - Send the update to all clients
        - Wait some time
- When all given local string modifications are done, the client sends a special 'goodbye' message to all clients.
- When 'goodbye' messages are received from all other clients, the client prints its replica and exit.

The merge algorithm

As shown in class, the operations of operation-based CRDT must be commutative. Since text operations are not commutative, we assign a total-ordered timestamp for each message: <vector timestamp, client id>. In this case, the client can apply the received update operations on its replica, according to their order. The problem is that the client may receive later update operation of other clients with earlier timestamp (causality violations). In order to handle this, we use the following algorithm:

- After applying an update (including the client's own operations), the client should store the operation with its timestamp and the updated string.

- Whenever the client receives an update operation: in case the timestamp of the operation is smaller than the last updated operation, it should re-apply the updated operations which are greater than the given new updated operation, on the string of that earlier point.
- In case the earlier stored updated operation is followed by updated operations of all clients with greater timestamps, it can/should be cleaned from the list of stored updated operations.

Logging

- When a client receives an updated operation, it should print:

Client <client Id> received an update operation <operation, timestamp> from client <sender id>

- During the merge algorithm, the client should print:

Client <client Id> started merging, from <starting timestamp> time stamp, on <starting string>

For each update operation: operation <updated operation, timestamp>, string: <updated string>

Client <client Id> ended merging with string <updated string>, on timestamp <ending timestamp>

- When client removes an update operation from his storage (due received operations with greater timestamps from all clients) it should print:

Client <client id> removed operation <operation, timestamp> from storage

- When the client finishes the local string modification, defined in its input file, it should print:

Client <client Id> finished his local string modifications

- Before client exit, it should print: Client <client Id> is exiting

Input

The input file of each client is composed of:

<client id>
<client port>
<initial string (same for all clients)>
\n
<other client id> <other client host> <other client port>
…
\n
<updated-operation>
…

Where <updated-operation> is defined by type ('insert' or 'delete') and index ('insert' without index denote character adding)

An example of input file is provided in the assignment's folder.

You can assume that the input is correct.

Note: the proposed update algorithm can be inefficient in case the client sends update messages to all other clients after <u>each local update</u>. Run your system in two modes:
- Update notifications after <u>every</u> local update.
- Update notifications after <u>10</u> local updates.

You can use any library of JS, in any way you wish. Remember that each client runs with a single thread, so I/O accessing should be asynchronous.

Your work will be checked frontally.

Good luck!