h e p i a
Haute école du paysage, d'ingénierie et d'architecture de Genève

Hes·SO Genève
Haute Ecole Spécialisée de Suisse occidentale

Lab 1 : Developing and deploying a broadcast algorithm for a blockchain system on a Cloud infrastructure

September 2020

Nabil Abdennadher

If you find a typo, no matter how small, please send an email to nabil.abdennadher@hesge.ch

We propose to design and implement a wave-based broadcast algorithm used to ensure communication in a blockchain system. The objective is not to implement a blockchain system itself, but to develop the communication layer used to send and receive transactions among the nodes that compose the blockchain system.

For the sake of simplicity, we assume that the blockchain system is composed of a set of identical "databases" which are stored on the nodes of the network. Each occurrence of the database is composed of transactions with this structure:

1. Sender (from)
2. Receiver (to)
3. Amount (in monetary units)

The blockchain concept is simple: each node that generates or creates a transaction stores it in its database, then sends it to the other nodes, which in turn store it in their database. In case of any tempering, a comparison between versions of the same transaction stored on the nodes allows the anomaly to be detected. A transaction is assumed to be valid if the same version of this transaction is stored in a majority of the nodes.

A node can check if a given transaction is valid. For this purpose, it can send a request to all nodes and ask them if the version it holds locally is the same as the one stored on these nodes.

The blockchain system we propose to develop supports two functions:

1. ***create_transaction (trans)***: a node generates a transaction "*trans*", stores it in its database and sends it to all nodes of the network. The sender node must wait until it receives a confirmation that the transaction *trans* was correctly received and stored by all the nodes.
2. **rate = vote (*trans*)**: A node sends a request to all nodes to check if the transaction *trans*, stored locally, is the same as the ones stored in the other nodes. The result (rate) is a percentage representing the transactions which are the same as the one stored locally.

Two additional functions will enable testing the blockchain system:

1. ***fake (authentic_trans, fake_trans)***: this function, executed locally, simulates a tempering attempt. It replaces an authentic transaction (*authentic_trans*) by a fake one (*fake_trans*).
2. ***list_of_trans = list (node)***: list all transactions on a given node. This function is executed locally.

The goal of this Lab is twofold:

1. design and develop a socket server implementing these four functions.
2. design and develop a client program that can access a given node and invoke one of these functions. For this, the client program needs to know the IP address and the port on which the socket server is listening.

We assume that each node *x* only knows the identifiers (addresses) of its neighbours and has a routing table that determines to which neighbour a message should be sent if the recipient is the node *y*. When starting, each node *x* must read two text files: *neighbour-x.txt* and *routing-x.txt*. The first (*resp.* second) file contains the neighbours (*resp.* the routing table) of the node x. The routing table is used when the node *x* receives a message which has to be sent to a node *y*. *routing-x[y]* contains the neighbouring node of *x* to which a message to node *y* must be routed.

Format of the *neighbour-x.txt* file

Example: We assume that nodes identifiers are represented by their IP.

```
129.194.184.101  // id of the first neighbour
129.194.184.110  // id of the second neighbour
...
129.194.184.111  // id of the nth neighbour
```

The number of lines in this table is equal to the number of neighbouring nodes of node *x*.

Format of the *routing-x.txt* file

The routing file (*routing-x.txt*) contains *n-1* lines where *n* is the number of nodes in the network. Each line corresponds to a node in the network and contains:

1. identifier of the concerned node
2. identifier of the neighbouring node (of node x) to which the message must be routed.

Example: We assume that nodes ids are represented by their IP.

```
129.194.184.130        129.194.184.110
129.194.184.140        129.194.184.101
...
```

In this example, if node *x* receives a message to route to node 129.194.184.130 (*resp.* 129.194.184.140) it forwards it to the neighbour node 129.194.184.110 (*resp.* 129.194.184.101).

For sake of compatibility, you must respect the format of these two files.

Work schedule:

Steps	Deliverable
Step 1 1. Design of the communication protocol. 2. Data structure.	A one-page document (pdf format) describing the communication protocol and the data structure to use. Deadline : see Cyberlearn
Step 2 Local deployment of the blockchain system. The whole system is deployed on one machine. All socket servers (nodes) are listening on the same IP but on different ports.	Two source codes (Python & Golang) + neighbour and routing files used to test your program (a network of minimum 10 nodes) Deadline : see Cyberlearn
Step 3 Cloud deployment	A source code + neighbour and routing files used to test your program (a network of minimum 10 nodes)

Students must work in pairs. One will develop the program in Python, the other will develop in Go.

Assessment criteria

1. The algorithm is deployed locally. In this case, each node uses a different port. This criterion is evaluated in step 2. To perform Step 2, the format of neighbour and routing files could be modified.
2. When needed, messages are sent in "parallel" (concurrence). This criterion is evaluated in step 2.
3. Each node correctly reads the "neighbour" and “routing” files as described above. Neighbour and routing file names must be as described above. This criterion is evaluated in step 2.
4. When receiving a message, the extraction of the sending node IP must be made through the functionalities offered by the sockets and not by including the IP of the sending node in the message itself. This criterion is evaluated in step 3.
5. The system is deployed on a minimum of 6 nodes (represented by 6 AWS Amazon instances). This criterion is evaluated in step 3.