

Developing and deploying a broadcast algorithm for a blockchain management system on a Cloud infrastructure

Lab 1

September 21st, 2022

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 transactions. Each transaction is replicated on all the nodes of the network as an object storage. In other words, each node manages an object-based storage containing all the transactions. A transaction has this structure:

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

The blockchain concept is simple: each node that generates a transaction stores it in its own object-based storage, then sends it to the other nodes, which in turn store it in their object-based storage.

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 most 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 object-based storage and sends it to all nodes of the network. This function relies on the broadcast by wave distributed algorithm.
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. This function relies on the broadcast by wave with ACK distributed algorithm. However, some adjustments are needed to implement this function.

Two additional functions will enable testing the blockchain system (they are not part of it):

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 (the object storage related to the node). This function is executed locally.

There is no distributed processing in these two functions.

The goal of this Lab is twofold:

1. design and develop a socket server implementing the four functions presented above.
2. design and develop a client program that invokes these functions. For this, the client program needs to know the IP address and the port on which the socket server is listening.

Client and server programs are different and must not be merged.

We assume that each node x only knows the identifiers (addresses) of its neighbours. When starting, each node x must read a text file: *neighbour-x.yaml*. The file contains the neighbours of the node x .

Format of the *neighbour-x.yaml* file

Each node is identified by an identifier (*id*), the port on which it listens (*port*) and a set of neighbours (*neighbours*). Each neighbour is a node which is represented by an identifier and the address/port on which it is listening. In the example below, the neighbours are deployed on the local machine. The “*edge_weight*” field is not used in this lab.

```

1  id: 1
2  address: "127.0.0.1"
3  neighbours:
4    - id: 2
5      address: "127.0.0.2"
6      edge_weight: 7
7
8    - id: 8
9      address: "127.0.0.8"
10     edge_weight: 4

```

Please, respect the format of this file.

The work is done by groups of 5 students. The group is responsible for the design of the ***create_transaction*** and ***vote*** algorithms. Then, each student is responsible to program and implement the two algorithms in Go or Python and with one of the following Object-based Storage infrastructures: Google Compute Engine, Azure or Exoscale.

Work schedule:

Steps	Deliverable
Step 1 1. Design of the communication protocol. 2. Data structure.	A ½ page document (pdf format) describing the communication protocol and the data structure to use. This deliverable is shared by the whole group (no evaluation).
Step2 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.	D#1.1: local deployment A source code program (Python or Golang) + the neighbour file used to test your program (a network of minimum 6 nodes). The storage of the transactions takes place on the main memory. This deliverable is individual (per student).
Step 3 Development of the object storage module: creating buckets, writing/reading data (transaction ID, sender, receiver, Amount) to/from the bucket.	D#1.2 The object storage module is fully operational in one of the object-based storage (Google Compute Engine, Azure or Exoscale): the server can write and read data (transaction ID, sender, receiver, Amount) from the object storage. This deliverable is individual (per student).

Step 4 Cloud deployment	D#1.3 (cloud deployment supporting object storage functionalities) 1. The four functions of the socket server are fully operational. They store/read data in/from the object storage. 2. The server is deployed on a network composed of at least 6 nodes represented by Virtual Machines on Exoscale, Google Cloud or Microsoft Azure. 3. The client is fully operational This deliverable is individual (per student).

Assessment criteria

1. Client and server programs are different and must not be merged.
2. When needed, messages are sent in "parallel" (concurrency, multi-thread programming).
3. Each node must read the "neighbour" file as described above.
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.
5. The code must be optimised to run for large scale networks.
6. The system is deployed on a minimum of 6 nodes.

Criteria 1, 2, 3 and 4 are necessary for the validation of the lab.