UNIVERSITAT ROVIRA I VIRGILI, TARRAGONA
Curs 2023/2024

Distributed storage systems and the CAP theorem

Distributed Systems, Assignment 2 (Version 1.1)

April 26, 2024

1 Assignment description

Distributed storage systems distribute data across a cluster of servers while giving users a single, unified service. They provide fault tolerance through data replication and scalability in the face of increasing client demands.

Depending on its implementation, distributed stores can be:

1. **Centralised**. They have one or more slave nodes connected to a master node. The master node is responsible for coordinating the whole system.
2. **Decentralised**. They have no master node, and coordination is done by consensus algorithms between their members.

You will develop **both (simplified) implementations** of a **distributed key-value storage system** using **python** and **`gRPC`**.

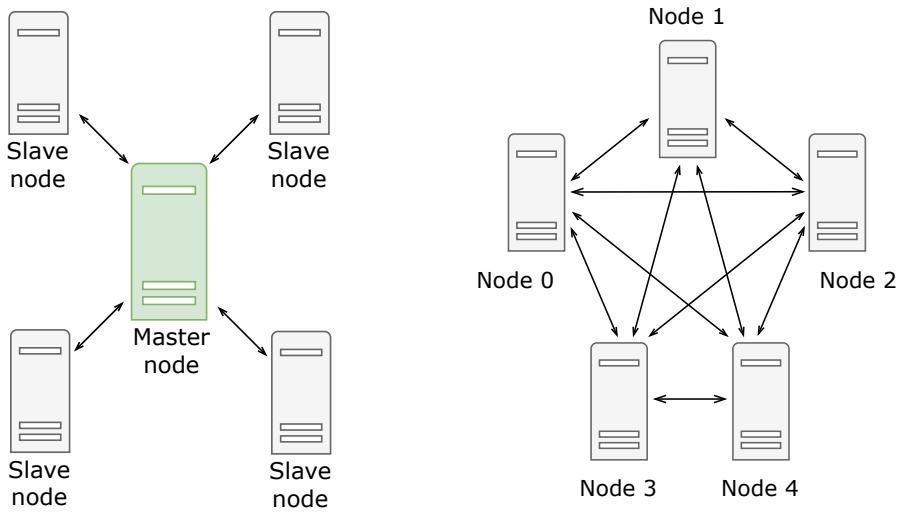

Figure 1: a) Centralised distributed system. b) Decentralised distributed system.

2 Implementation

2.1 Template and testing

You will be provided with:

- **`store.proto`**: A `.proto` file containing a client API definition. Your system will need to implement the gRPC calls contained in `store.proto`. You will have to extend the stub with the necessary system intra-communication requests.
- **`eval.py`**: Test and evaluation code for your implementations. Starts the underlying storage systems (assuming they follow the naming and implementation specifications described in the present document) and applies a set of test to them. Returns consistency and performance results of the system.

Both systems will have exactly 3 nodes, listening on ports [32770, 32771, 32772], each running in a separate process. All nodes must implement and serve the client API.

You will need to evaluate both systems separately, using the same **eval.py** file, and

compare the results.

Keys and values will be strings. All data (i.e. all keys and their corresponding values) will be replicated in all nodes.

2.2 Client API

Here we describe the operations defined in `store.proto` for the client API.

- `put(key: str, value: str)`: Saves the value into the specified key.
- `get(key: str): str`: Gets the value from the specified key.
- `slow_down(secs: int)`: The receiver node adds a `secs` seconds delay to all its gRPC responses. We will use this call in `eval.py` to simulate defective nodes and network partitions.
- `restore()`: The receiver node suppresses any delays, if it has previously received a `slow_down` request.

2.3 Centralised implementation

The centralised implementation **will run the master node on port 32770**. The rest of the nodes will be slave nodes.

You can assume that put (write) requests will only be directed to the master node, while get (read) requests can be hosted by any (master or slave) node.

Consistency throughout the system is ensured using a basic Two-Phase Commit (2PC) protocol (see slide 59 of "Consistency and Fault tolerance", in the Moodle page of the course). The master node will acknowledge **all** slave nodes on put operations via 2PC (Figure 3). Get operations do not imply synchronisation.

The 2PC protocol must be implemented with point-to-point gRPC requests between cluster nodes. This means that the gRPC server must be extended with the necessary requests (*e.g.* `canCommit` and `doCommit`).

You will need to set up the cluster using a **node discovery protocol** with point-to-point gRPC calls. Stored nodes will notify the master node at startup so that the master node can register them and include them in the cluster and 2PC protocol.

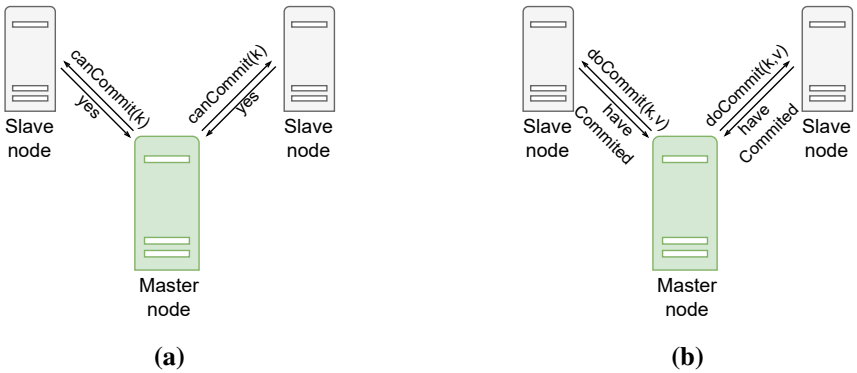

Figure 2: Two-Phase Commit protocol on a client `put(k, v)` request (all communication must be done with gRPC). The put request is directed the master node.
a) Voting phase. b) Completion according to outcome of vote.

2.4 Decentralised implementation

The decentralised implementation **can receive put and get operations in any of its nodes**. Both put and get operations are propagated using a weighted quorum-based protocol. In this way, put and get requests will not return to the client until a certain threshold of quorum size is reached.

Specifically, you will implement *Example 2* in slide 22 of "Replication", in the Moodle page of the course (ignoring latencies).

Node weights are assigned as follows

Node #	Port	Weight
0	32770	1
1	32771	2
2	32772	1

Quorum sizes are 2 for reads and 3 for writes. The system will use the quorum protocol to commit value updates between participating nodes once quorum is reached.

The quorum algorithm must be implemented using point-to-point gRPC requests. The node receiving a put request sends a quorum request to every other node in the cluster to begin the protocol. Post-quorum commit messages must also be implemented using gRPC. The gRPC server must be extended with the necessary requests for the quorum.

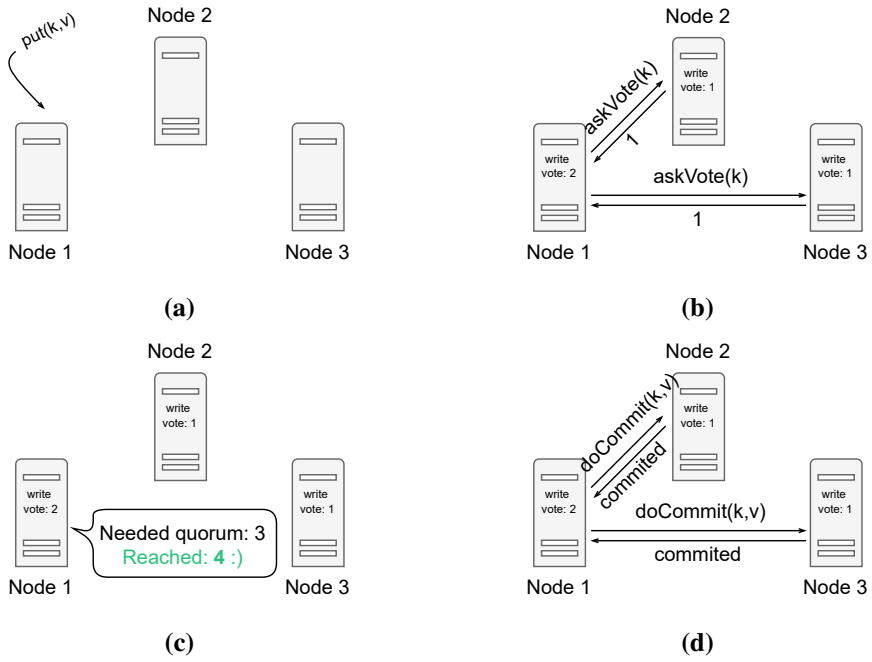

Figure 3: Quorum protocol on a client `put(k, v)` request (all communication must be done with gRPC). a) Client request. b) Vote gathering. c) Quorum verification. d) Commit of updates.

You will need to set up the cluster using a **node discovery protocol**. You can implement this according to your own criteria, with proper justification of the design in the documentation.

3 Delivery

3.1 Project requirements

- The system must be launched using only one script per version: `centralised.py` (to launch the centralised, master-slave server) and `decentralised.py` (to the decentralised, quorum-based server). Every other process and/or terminal must be launched automatically from inside the scripts.
- It is crucial that `eval.py` runs with no errors over both implementations.
- You are allowed to have configuration parameters (IPs and ports, for instance) hardcoded in a separate `config.yaml` file.
- UIs **must** be implemented directly in terminals with `ascii` code.
- Your system should permit the communication between computers with known IPs in a shared network.

3.2 Code delivery

You will publish your project in a public GitHub repository of your own. The repository must contain:

- All the code and auxiliary files necessary to run your system.
- A `README.md` with a brief description of the project and the steps to run it.

4 Documentation

You must provide a concise and coherent report for this task, comprising three sections:

1. Abstract. A brief summary (150 words) of your project. Tips for writing an abstract.

2. System design and discussion. Explain the major design decisions of your project and discuss on their appropriateness and limitations.
3. System comparison. Present the `eval.py` results for each implementation, compare them, and reason about their values.
4. Questions. Provide thoughtful responses to the following questions.
   4.1. **Q1** Justify the consistency level of your centralised implementation.
   4.2. **Q2** Characterize each of your implementations according to the CAP theorem.

5 Evaluation

- You will have to upload the documentation to the course's Moodle page, before its deadline. The documentation must include a link to the code's GitHub page.
- The task must be implemented and delivered in pairs, and the mark will be the same for both teammates. A group interview will be conducted to evaluate the task.
- You are free to implement the details not described in this document on your own criteria. Such decisions must be justified in the documentation.