Course project: Mars distributed storage system

Distributed Systems, 2024-2025

You have been enrolled by the European Space Agency (ESA) as a consultant for one of its most ambitious and technically demanding projects: the deployment of a distributed network of data collection stations on the surface of Mars. These stations are spread strategically across the Martian landscape to gather vital scientific data from various sources, including rovers and orbiting satellites. A fundamental requirement of this system is ensuring data availability and robustness. Given the extreme conditions on Mars, the risk of data loss is significant and must be minimized. To address this, data must be effectively replicated and shared among multiple stations. These stations act as nodes within a decentralized, peer-to-peer network, each equipped with a processing unit and capable of wireless communication via antenna arrays. Unlike a stable, cabled environment on Earth, the Martian context introduces considerable challenges. Harsh weather, hardware degradation, and fluctuating environmental conditions mean that stations can crash, disconnect, reboot, or join the network unpredictably. This dynamic and often unstable behavior creates a highly volatile environment for data storage and communication, making resilience and fault-tolerance critical system features.

Your mission is to develop a proof-of-concept for a peer-to-peer key-value storage system that meets these constraints. To demonstrate the system's functionality and resilience, you are also required to implement a main program file that simulates key operations. This script must showcase typical scenarios such as uploading and requesting data, handling node failures and recoveries, and executing basic network management commands.

To distribute the load effectively, nodes divide the data they store using a key-based partitioning scheme. Keys, represented as unsigned integers, are arranged in a circular space called a "ring", i.e., the largest key value wraps around to the smallest key value like minutes on analog clocks. Each data item with a key *K* is assigned to the first *N* nodes found when moving clockwise from *K* on the ring. Here, *N* is a configurable replication factor. For instance, with *N = 3* and nodes identified by 10, 20, 30, and 40, a data item with key 15 would be stored on nodes 20, 30, and 40. A data item with key 35 would be stored on nodes 40, 10, and 20. Every node in the system must be aware of all other nodes so it can independently determine which nodes are responsible for storing a particular data item. When nodes are added or removed, the system must automatically adjust and redistribute data to maintain balance and consistency.

The project that you are developing should support management services and create clients supporting data services. The data service consists of two commands: **update(key, value)** and **get(key)->value**. Any storage node in the network must be able to manage both requests, regardless of the key, by forwarding data to/from appropriate nodes. The node contacted by a client for a given user request is called the *coordinator*. The management service consists of a **join** operation that creates a new node in the storage network and a **leave** operation to remove a node.

🔁Replication

Replication is based on quorums. Therefore, to support consistent reads, it is necessary to associate internally the versions with every data item.
The system-wide parameters *W* and *R* specify the write and read quorums, respectively.
It is important to remember that $W \leq N$ and $R \leq N$. Moreover, if an operation cannot be completed within a given timeout *T*, the coordinator reports an error to the requesting client.
Read. When it receives a **get** command from a client, the coordinator requests the item from the *N* nodes responsible for it. To provide better availability, as soon as *R* replies arrive, the request coordinator sends the data item back to the requesting client.
Your approach needs to offer sequential consistency by taking advantage of version numbers. Consistent with the order of operations specified by each process (client), read and write operations must appear as if they were occurring in a certain global order.
Write. When a node receives an **update** request from the client, it first requests the currently stored version of the data item from the *N* nodes that have its replica. For better availability, the write coordinator reports success to the client as soon as it receives the first *W* replies required to support sequential consistency. It then sends the update to all *N* replicas, incrementing the version of the data item based on the *W* replies. Otherwise, on timeout, the coordinator informs the client of the failed attempt and stops processing the request. For simplicity, we assume that there are no failures within the request processing time.
Local storage. Every node should maintain a persistent storage containing key, version, and value for every data item the node is responsible for. You can simulate persistent storage in your implementation using a suitable data structure whose data does not get deleted upon crash failures.

📦🔄Item repartitioning

When a node joins or leaves the network, repartitioning is required. Note that both operations are performed only upon external request. A crashed node should not be considered as leaving but only temporarily unavailable; therefore, there is no need to implement a crash detection mechanism or repartition the data when a node cannot be accessed.
Joining. The actor reference of one of the nodes that is currently operating, which serves as its bootstrapping peer, and the key of the newly created node are both included in the **join** request. First, the joining node contacts the bootstrapping peer to retrieve the current set of nodes constituting the network. Having received this information, the joining node should request data items it is responsible for from its clockwise neighbor, which holds all items it needs. It should then perform read operations to ensure that its items are up to date. After receiving the updated data items, the node can finally announce its presence to every node in the system and start serving requests coming from clients. The others should remove the data items assigned to a new node that they are no longer in charge of as soon as they become aware of it.
Leaving. It is possible to request a node to leave the network. In order to accomplish this, the leaving node notifies the others of its impending leaving and transfers its data items to the nodes that will be in charge of them once it leaves.
Recovery. When a crashed node is restarted, it requests a node listed in the recovery request for the current set of nodes rather than executing the join operation. It should

retrieve the items that are now under its control (because nodes left) and discard the ones that are no longer under its control (because other nodes joined while it was down).

📋🤔Other requirements and assumptions

- The project should be implemented in Akka, with nodes being Akka actors, written in Java programming language. In exceptional circumstances, the project may be implemented using alternative frameworks or languages; however, **this must first be discussed with the instructors.**
- The clients execute commands passed through messages, print the reply from the coordinator, and only then accept further commands. A client may perform multiple read and write operations.
- A node must be able to serve requests from different clients, and the network must support concurrent requests (possibly affecting the same item key).
- Nodes join and leave, crash and recover one at a time, and only when there are no ongoing operations. Operations might resume while one or more nodes are still in a crashed state.
- For the sake of this project, the network is assumed to be FIFO and reliable.
- Use (unsigned) integers for keys, and strings (without spaces) as data items. The keys are set by the user to simplify testing, though in real systems, random-like hash values are used.
- The replication parameters *N*, *R*, *W*, and *T* should be configurable at compile time.
- To emulate network propagation delays, you are requested to insert small random intervals between the unicast transmissions.
- *N* must always be less than or equal to the number of nodes.
- There must always be at least one node in the distributed storage system that is not in crash state.
- It is important to state all the additional assumptions in the report.

📑🖋️Project report

You are asked to provide a short document (typically 3-4 pages) in English explaining the main architectural choices. Your report should include a discussion on how your implementation satisfies consistency requirements (sequential consistency).
In the project presentation slides, you can find a list of relevant questions your document should answer.

📊✅Grading criteria

You are responsible for showing that your project works. The project will be evaluated for its technical content, i.e., algorithm correctness. Do not spend time implementing features other than the ones requested; focus on doing the core functionality, and doing it well.
A correct implementation of the whole requested functionality is worth 6 points. It is possible to submit programs implementing a subset of the requested features or systems requiring stronger assumptions. In these cases, lower marks will be awarded.
You are expected to implement this project with exactly one other student. However, the marks will be individual, based on your understanding of the program and the concepts used.

🎤👩‍🏫Presenting the project

- You MUST contact through e-mail the instructor gianpietro.picco@unitn.it AND the teaching assistant erik.nielsen@unitn.it, stefano.genetti@unitn.it well in advance, i.e., a couple of weeks before the presentation.
- You can present the project at any time, also outside of exam sessions. In the latter case, the mark will be frozen until you can enroll in the next exam session.
- The code must be properly formatted otherwise, it will not be accepted (e.g. follow style guidelines).
- Both the code and the report must be submitted in electronic format via email at least one day before the meeting. The report must be a single self-contained pdf/txt. It must be sent together with all code in a single tarball consisting of a single folder (named after your surname). For instance, if your surnames are Rossi and Russo, put your source files and the report in a directory called **RossiRusso**, compress it with "**tar-czvf RossiRusso.tgz RossiRusso**" and submit the resulting "**RossiRusso.tgz**".
- The project will be presented to the instructor and/or assistant, with no ESA personnel in attendance.
- **Plagiarism is not tolerated.** Do not hesitate to ask questions if you run into trouble with your implementation. We are here to help.