Distributed system project

Mario Vallone
Matteo Carrara

How does the library work?

The library consists of three key components:

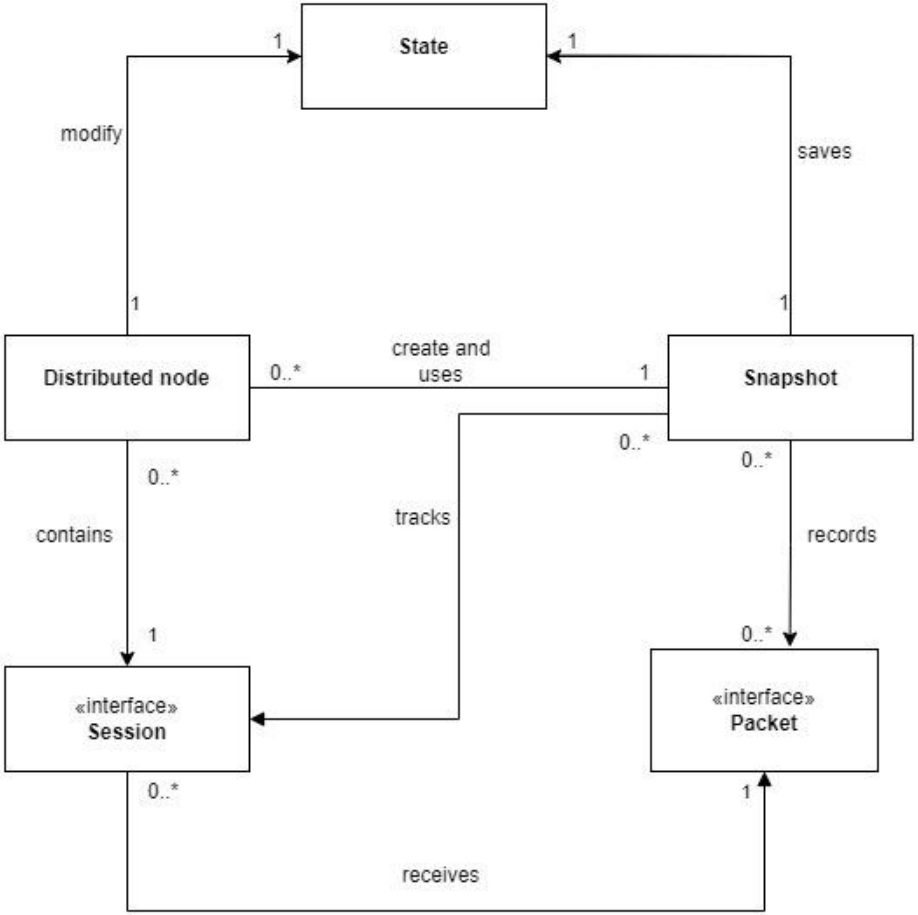
1. Distributed node: This class represents a node in the network and contains both the node's state and the logic required to implement the snapshot functionality.

2. State: The interface that represent the state of the application, it is serializable, and has a deep clone and a restore method.

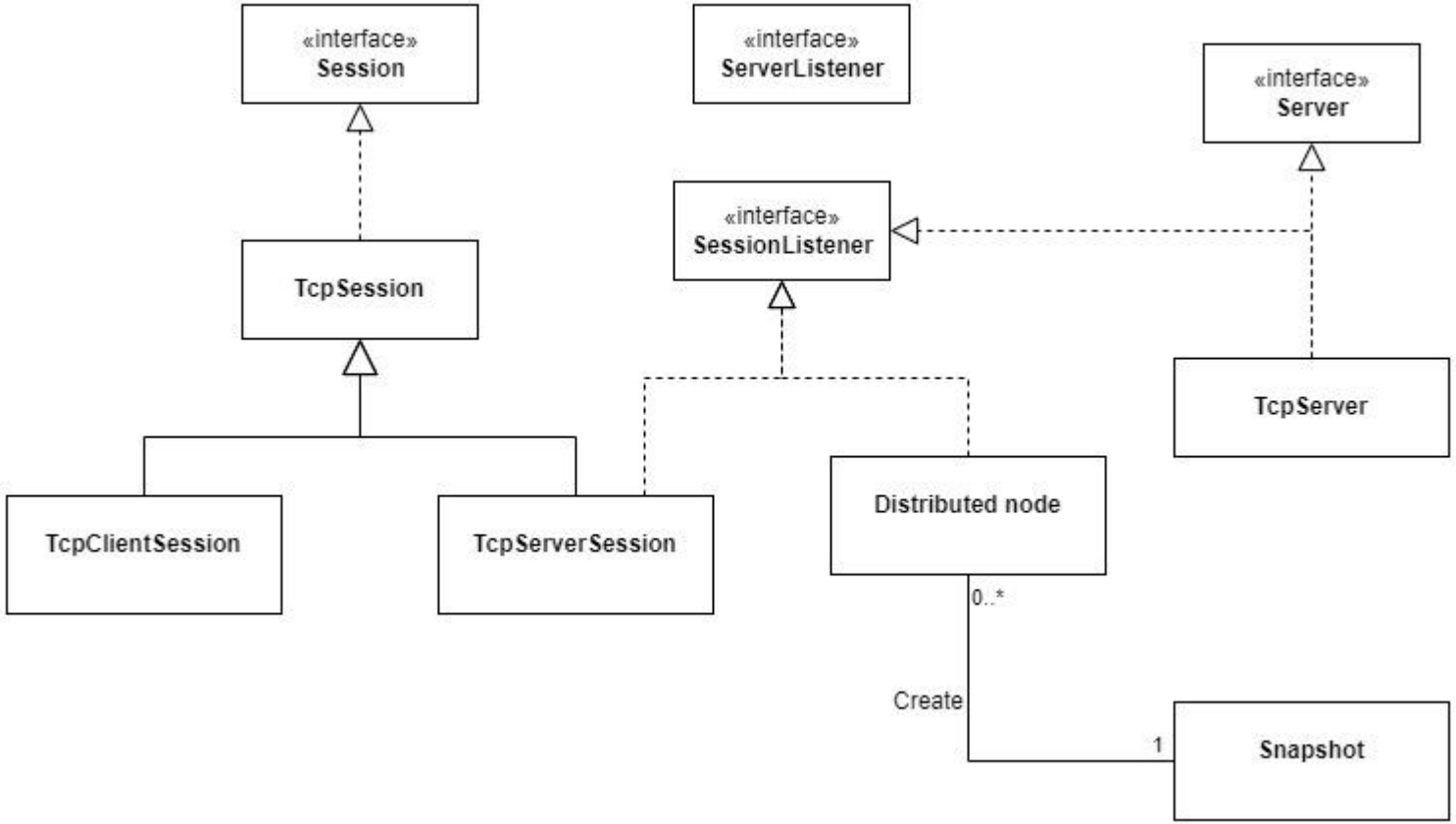
3. Snapshot: The snapshot component represents a complete state capture of the node. It includes all the necessary information to save the snapshot on disk and manage it effectively.

4. Session: A session represents one end of a link. Each session must have an unique ID, that must be equal on both ends of the link. Sessions allow to send and receive packets.

UML class diagram of some classes in the library



UML Class diagram of some classes in the library



Snapshot

A node starts a snapshot by generating an ID and sending a SnapshotMarkerPacket to all the neighbours, while also marking them as pending in the Snapshot object.

Upon receiving a snapshot marker packet, a node checks if it is a new snapshot and if so, it saves locally the snapshot info and broadcasts the message to all the neighbours excluding the link where it received the marker from, while also marking them as pending in the Snapshot object.
It also sends the ack back to the source, regardless if it already knew about the snapshot.

Upon receiving an ack response, the corresponding link is removed from the pending sessions.

Once the pending sessions become empty, the snapshot is completed and the listeners are notified.

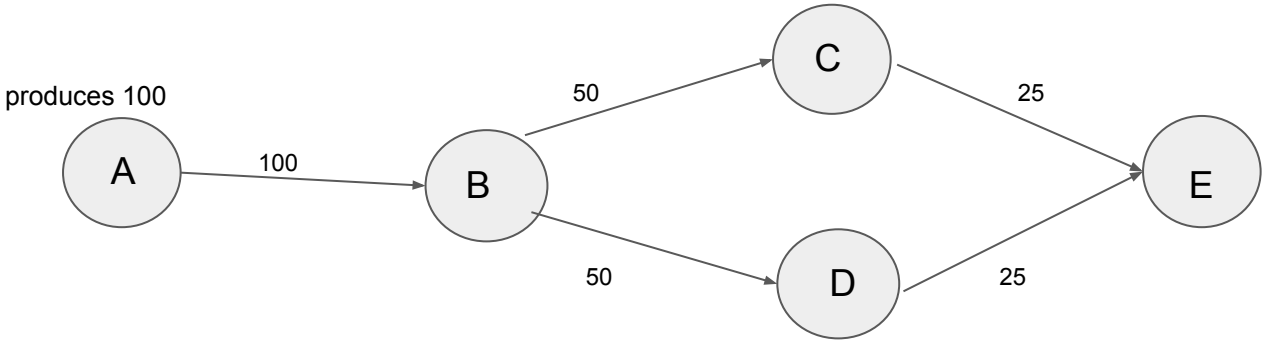
During the time in-between the marker and the ack, the node saves all the packets received on the link in the Snapshot object.

How to use the library

1. Create your own state extending State and implement the method restore.
2. Create your own DistributedNode, Server and ClientSession extending the one of the library.
3. Create your own packets for the application.
4. Add your listeners to the class of your interest implementing the logic of the application.

App to test the library

The developed application serves as a testing environment for the library and simulates a production chain. Each node in the network represents a machinery unit capable of receiving inputs from various channels. Once the machinery unit accumulates a sufficient amount of input materials based on its production chain's size, it initiates the production process and transmits the final product to the subsequent nodes, proportionally distributing it according to predetermined percentages.



How to deal with node failures (example application)?

In the event of a node failure, one or more nodes can experience a temporary disruption. However, when a failed node recovers and comes back online, it initiates a procedure to restore the most recent snapshot across the entire network. This process aims to bring the distributed system back to a consistent state.

To implement the recovery procedure, a node that intends to initiate a snapshot restoration sends a "RestoreSnapshot" message to all its sessions. Upon receiving acknowledgment (ack) from every session, the node proceeds to restart the last snapshot. If, during this procedure, the node receives another "RestoreSnapshot" message with the same identifier, it acknowledges the request by sending an ack through the corresponding channel, in order to deal with circular networks.

What happens to the other nodes in the network during this process?

When a node receives a "RestoreSnapshot" message, it saves the ID contained in the packet, which represents the ID of the ongoing restore procedure and the ID of the channel it received it from.

It then proceeds to send a RestoreSnapshot with the same ID to all its neighbours, except the one it received the RestoreSnapshot from, or if it has no other neighbours it sends the ack back to the initiator.

However, if the node receives another "RestoreSnapshot" message with the same ID, it sends an ack to the channel in order to deal with circular networks.

After receiving all the ACKS, it sends an back to the channel whose ID was saved for the current RestoreSnapshot procedure (called the initiator).

Testing

How the project has been tested?

The project has undergone testing in two ways: manual testing involving various network configurations and configuration parameters, and automated testing using JUnit with more specific test cases.

By conducting manual tests, different network setups and configuration parameters were explored to validate the project's functionality and performance under diverse scenarios. This approach allowed for a comprehensive assessment of the system's behavior and its ability to adapt to different configurations.

In addition, JUnit tests were employed to target specific functionalities and ensure their correctness. These automated tests provided a structured and repeatable validation process, allowing for thorough testing of individual components, edge cases, and expected behaviors.

Combining both manual and automated testing approaches provided a robust evaluation of the project, covering a wide range of scenarios and ensuring the reliability and quality of the implemented functionality.

Case tested through manual test

- Tested snapshot and snapshot restore in circular network
- Tested snapshot and snapshot restore in different network topologies
- Tested reconnection of a node
- Tested snapshot and snapshot restore after reconnection of one or more node

Case tested using junit

We used programmatic tests to test some conditions that are either hard to reproduce or very important to get right:

- Test1: Ensure that packets received in between a snapshot start packet and the ack are restored correctly
- Snapshots in a circular network: ensure that the snapshot completes when the network is circular
- Snapshot in a non circular network