LTIMindtree

R3 Corda Hands-on Tutorial

Table of Contents

1 Overview

While regulated companies may start on public blockchains, they soon realize when they get to production, they require capabilities native to Corda such as privacy, security, scalability and ease-of-integration with existing systems[1].
R3's Corda is a scalable, permissioned peer-to-peer (P2P) distributed ledger technology (DLT) platform that enables the building of applications that foster and deliver digital trust between parties in regulated markets.

2 Difference between Corda Distributed Ledger Technology and Public Blockchain

- As Corda Distributed Ledger Technology is **Permissioned Blockchain**, that simply means each node in corda are known to each other so that they can easily share the data with relevant parties whereas in Public Blockchain, it is **Permissionless Blockchain platforms**, in which all data is shared with all parties.

- Corda DLT has **high security** because to participate in a CorDapp each entity must be granted access to do so and tied to a legal entity but in public blockchain, it is completely open so anyone can join as an anonymous actor on the network raising concerns about security.

- Corda DLT uses **Validity Consensus** and **Uniqueness Consensus** algorithms whereas public blockchain uses **Proof of Work** and **Proof of Stack** algorithm.

- Corda is **highly confidential**, as by design corda shares data only with the counterparties of a transaction but in public blockchain it validates transaction with public network this broadcasting sensitive information to all participants.

- Corda DLT is **highly Scalable** than Public Blockchain as in Corda P2P (Peer-to-Peer) architecture enables high level of network scalability and throughput but in public blockchain.

3 Tools for executing tutorial

Before we get started with the Corda tutorial, we will need the following tools.

- Gradle v7.4.2 https://gradle.org/next-steps/?version=7.4.2&format=bin
- Git
- IntelliJ
- Command-line
- Zulu JDK v8.64.0.19 https://www.azul.com/downloads/

As of the integrated development environment, we are going to use the IntelliJ. It will simplify our development.

4 Executing the Tutorial

To execute the tutorial, follow the steps mentioned in the following link- https://training.corda.net/getting-started/run-the-example-project/

5 The Corda Objects

Details about the Corda objects used in the tutorial are provided below:[2]

5.1 Nodes

- A node is a piece of software that participates in a Corda system or network and runs Cordapps.

5.2 States

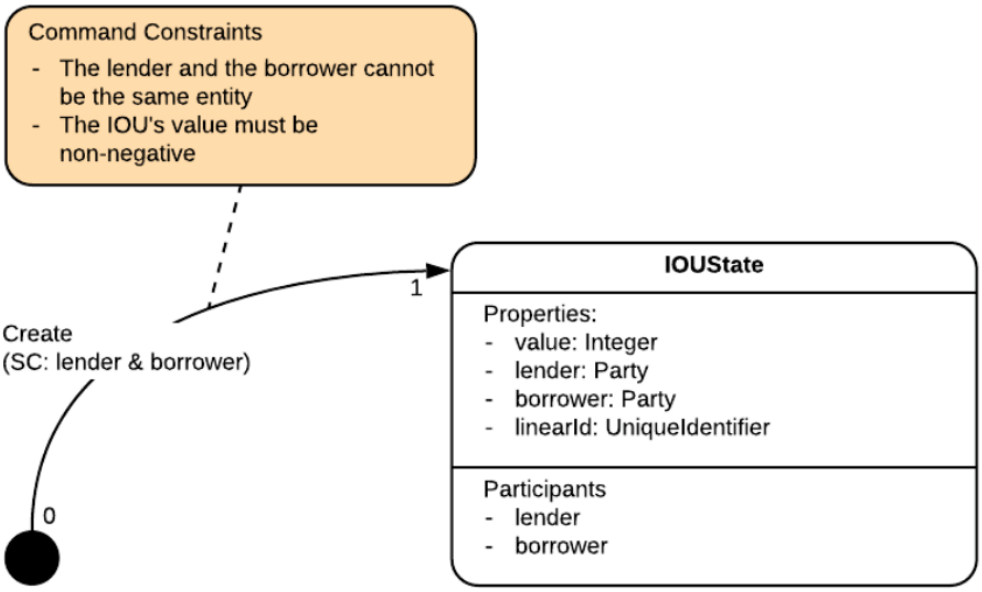
- States represent facts on the ledger.
- Facts evolve on the ledger when participants create new states and mark outdated states as historic.
- Each node has a vault where it stores the states it shares with other nodes.
- The aggregate of all states held by all nodes of the network is the distributed ledger state.
- There is no central ledger, and not all nodes know all states, so the overall ledger is subjective from the perspective of each participant.

5.3 Transactions

- Transactions are what consume states and produce new states.
- They are atomic- Transactions either complete entirely or have no effect.
- There are not partially complete, “in-flight” transactions although they do have a lifecycle with various stages.

5.4 Contracts

- A transaction is contractually valid if all its input and output states are acceptable according to the contract.
- Contracts are written in Java or Kotlin.
- Contract execution is deterministic, and transaction acceptance is based on the transaction’s contents alone.



- Contract has a single method, **verify**, which takes a LedgerTransaction as input and returns nothing.
- This function is used to check whether a transaction proposal is valid, as follows:
  - We gather together the contracts of each of the transaction's input and output states
  - We call each contract's verify function, passing in the transaction as an input
  - The proposal is only valid if none of the verify calls throw an exception

5.4.1 LedgerTransaction Object

- The LedgerTransaction object passed into verify has the following properties: Inputs, Outputs, Commands, Attachments, Notary, TimeWindow.

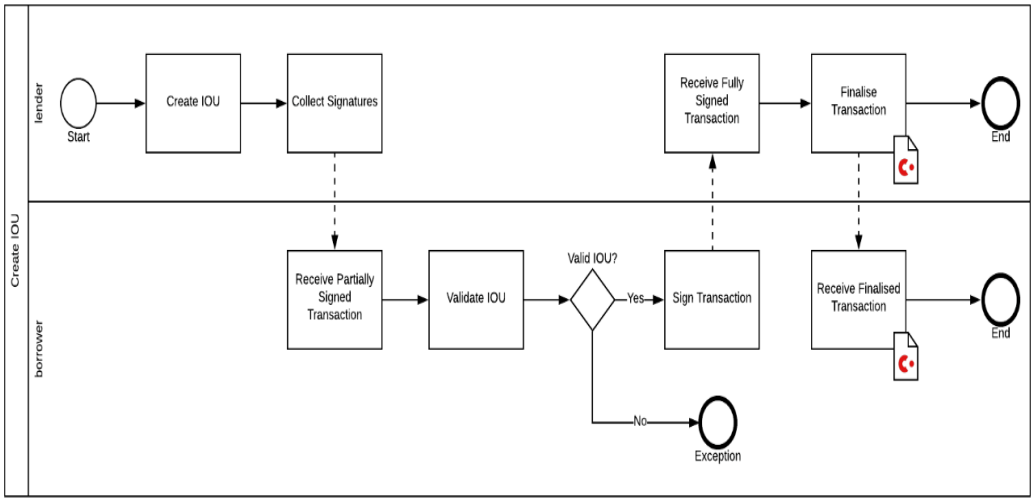
5.4.2 requireThat Function

- To impose a series of constraints, we can use requireThat.
- For each <String, Boolean> pair within requireThat, if the boolean condition is false, an IllegalArgumentException is thrown with the corresponding string as the exception message. In turn, this exception will cause the transaction to be rejected.

5.5 Flows

- Flows automate the process of agreeing ledger updates.
- Communication between nodes only occurs in the context of these flows and is point-to-point.
- Built- in flows are provided to automate common tasks.

To achieve this consensus, the flow goes through several steps:



- Flows have two sides[4]:
  - An Initiator side, that will initiate the request to update the ledger
  - A Responder side, that will respond to the request to update the ledger.

5.5.1 Initiator

In our flow, the Initiator flow class will be doing the majority of the work:

Part 1 - Build the transaction
- Choose a notary for the transaction
- Create a transaction builder
- Extract any input states from the vault and add them to the builder
- Create any output states and add them to the builder
- Add any commands, attachments and time-window to the builder

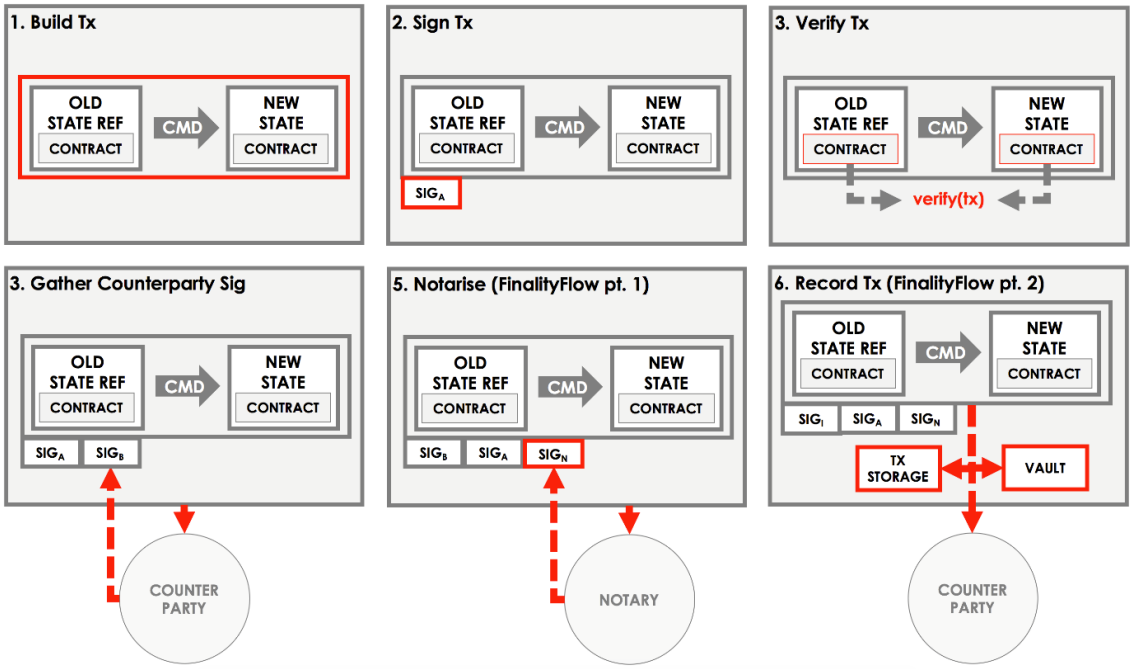
Part 2 - Sign the transaction
- Sign the transaction builder
- Convert the builder to a signed transaction

Part 3 - Verify the transaction
- Verify the transaction by running its contracts

Part 4 - Gather the counterparty's signature
- Send the transaction to the counterparty
- Wait to receive back the counterparty's signature
- Add the counterparty's signature to the transaction
- Verify the transaction's signatures

Part 5 - Finalize the transaction
- Send the transaction to the notary
- Wait to receive back the notarised transaction
- Record the transaction locally
- Store any relevant states in the vault
- Send the transaction to the counterparty for recording

We can visualize the work performed by initiator as follows:



5.5.2 Responder

To respond to these actions, the responder takes the following steps:

Part 1 - Sign the transaction

- Receive the transaction from the counterparty
- Verify the transaction's existing signatures
- Verify the transaction by running its contracts
- Generate a signature over the transaction
- Send the signature back to the counterparty

Part 2 - Record the transaction

- Receive the notarised transaction from the counterparty
- Record the transaction locally

Store any relevant states in the vault

5.6 Consensus

Consensus is the process by which all parties achieve certainty about the shared states. Corda applies two types of consensuses:

- **Validity Consensus** – It requires contractual validity of the transaction and all its dependencies. This is checked by each required signer before they sign the transaction.
- **Uniqueness Consensus** – It prevents double-spends. This is only checked by a notary service.

5.7 The Parties

- The Party class represents an entity on the network, which is typically identified by a legal name and public key that it can sign transactions under.

5.8 Notary Services

- The notary service prevents “double-spends”.
- The notary also acts as the time-stamping authority. If a transaction includes a time window, it can only be notarized during that window.
- Notary clusters may optionally also validate transactions, in which case they are called “validating” notaries, as opposed to “non-validating”.
- A network can have several notary clusters, all running different consensus algorithms.

6 References

1. https://corda.net/why-corda/
2. https://training.corda.net/corda-fundamentals/concepts/
3. https://training.corda.net/getting-started/run-the-example-project/
4. https://docs.r3.com/en/platform/corda/4.6/open-source/api-flows.html