

Consensus and notaries

A notary is a service that provides transaction ordering and timestamping.


Notaries are expected to be composed of multiple mutually distrusting parties who use a standard consensus algorithm. Notaries are identified by and sign with [Composite Keys](#). Notaries accept transactions submitted to them for processing and either return a signature over the transaction, or a rejection error that states that a double spend attempt has occurred.

Corda has “pluggable” notary services to improve privacy, scalability, legal-system compatibility and algorithmic agility. The platform currently provides validating and non-validating notaries, and a distributed RAFT implementation.

Consensus model

The fundamental unit of consensus in Corda is the **state**. Consensus can be divided into two parts:

1. Consensus over state **validity** – parties can reach certainty that a transaction is accepted by the contracts pointed to by the input and output states, and has all the required signatures. This is achieved by parties independently running the same contract code and validation logic (as described in [data model](#))
2. Consensus over state **uniqueness** – parties can reach certainty the output states created in a transaction are the unique successors to the input states consumed by that transaction (in other words – an input state has not been previously consumed)

 Note


The current model is still a **work in progress** and everything described in this article can and is likely to change

Notary

A **notary** is an authority responsible for attesting that for a given transaction, it has not signed another transaction consuming any of the same input states. Every **state** has an appointed notary:

```
/**
 * A wrapper for [ContractState] containing additional platform-level state information.
 * This is the definitive state that is stored on the ledger and used in transaction
 * outputs
 */
data class TransactionState<out T : ContractState>(
    /** The custom contract state */
    val data: T,
    /** Identity of the notary that ensures the state is not used as an input to a
     * transaction more than once */
    val notary: Party) {
    ...
}
```

Transactions are signed by a notary to ensure their input states are **valid** (apart from issue transactions, containing no input states). Furthermore, when using a validating notary, a transaction is only valid if all its dependencies are also valid.

 Note

The notary is a logical concept and can itself be a distributed entity, potentially a cluster maintained by mutually distrusting parties

When the notary is requested to sign a transaction, it either signs it, attesting that the outputs are the **unique** successors of the inputs, or provides conflict information for any input state that has been consumed by another transaction it has already signed. In doing so, the notary provides the point of finality in the system. Until the notary signature is obtained, parties cannot be sure that an equally valid, but conflicting, transaction will not be regarded as confirmed. After the signature is obtained, the parties know that the inputs to this transaction have been uniquely consumed by this transaction. Hence, it is the point at which we can say finality has occurred.

Multiple notaries

More than one notary can exist in a network. This gives the following benefits:

- **Custom behaviour.** We can have both validating and privacy preserving Notaries – parties can make a choice based on their specific requirements.
- **Load balancing.** Spreading the transaction load over multiple notaries will allow higher transaction throughput in the platform overall
- **Low latency.** Latency could be minimised by choosing a notary physically closer the transacting parties

Changing notaries


A transaction should only be signed by a notary if all of its input states point to the same notary. In cases where a transaction involves states controlled by multiple notaries, the states first have to be repointed to the same notary. This is achieved by using a special type of transaction whose sole output state is identical to its sole input state except for its designated notary. Ensuring that all input states point to the same notary is the responsibility of each involved party (it is another condition for an output state of the transaction to be **valid**)

To change the notary for an input state, use the [NotaryChangeFlow](#) . For example:

```
@Suspendable
fun changeNotary(originalState: StateAndRef<ContractState>,
    newNotary: Party): StateAndRef<ContractState> {
    val flow = NotaryChangeFlow.Instigator(originalState, newNotary)
    return subFlow(flow)
}
```

The flow will:

1. Construct a transaction with the old state as the input and the new state as the output
2. Obtain signatures from all participants (a participant is any party that is able to consume this state in a valid transaction, as defined by the state itself)
3. Obtain the old notary signature
4. Record and distribute the final transaction to the participants so that everyone possesses the new state

 Note


Eventually, changing notaries will be handled automatically on demand.

Validation


One of the design decisions for a notary is whether or not to **validate** a transaction before accepting it.

If a transaction is not checked for validity, it opens the platform to “denial of state” attacks, where anyone can build an invalid transaction consuming someone else’s states and submit it to the notary to get the states “blocked”. However, if the transaction is validated, this requires the notary to be able to see the full contents of the transaction in question and its dependencies. This is an obvious privacy leak.

The platform is flexible and currently supports both validating and non-validating notary implementations – a party can select which one to use based on its own privacy requirements.

 Note

In the non-validating model, the “denial of state” attack is partially alleviated by requiring the calling party to authenticate and storing its identity for the request. The conflict information returned by the notary specifies the consuming transaction ID along with the identity of the party that had created the transaction. If the conflicting transaction is valid, the current one is aborted; if not, a dispute can be raised and the input states of the conflicting invalid transaction are “un-committed” (via a legal process).

 Note

At present, all notaries can see the entire contents of a submitted transaction. A future piece of work will enable the processing of [Transaction tear-offs](#), thus providing data hiding of sensitive information.

Timestamping

A notary can also act as a timestamping authority, verifying the transaction timestamp command.

For a timestamp to be meaningful, its implications must be binding on the party requesting it. A party can obtain a timestamp signature in order to prove that some event happened before, on, or after a particular point in time. However, if the party is not also compelled to commit to the associated transaction, it has a choice of whether or not to reveal this fact until some point in the future. As a result, we need to ensure that the notary either has to also sign the transaction within some time tolerance, or perform timestamping and notarisation at the same time, which is the chosen behaviour for this model.


There will never be exact clock synchronisation between the party creating the transaction and the notary. This is not only due to physics, network latencies, etc. but also because between inserting the command and getting the notary to sign there may be many other steps, like sending the transaction to other parties involved in the trade, or even requesting human sign-off. Thus the time observed by the notary may be quite different to the time observed by the party creating the transaction.

For this reason, times in transactions are specified as time windows, not absolute times. In a distributed system there can never be “true time”, only an approximation of it. Time windows can be open-ended (i.e. specify only one of “before” and “after”) or they can be fully bounded. If a time window needs to be converted to an absolute time (e.g. for display purposes), there is a utility method on `Timestamp` to calculate the mid point.

In this way, we express the idea that the true value of the fact “the current time” is actually unknowable. Even when both before and after times are included, the transaction could have occurred at any point between those two timestamps. Here, “occurrence” could mean the execution date, the value date, the trade date etc ... The notary doesn’t care what precise meaning the timestamp has to the contract.

By creating a range that can be either closed or open at one end, we allow all of the following facts to be modelled:

- This transaction occurred at some point after the given time (e.g. after a maturity event)
- This transaction occurred at any time before the given time (e.g. before a bankruptcy event)
- This transaction occurred at some point roughly around the given time (e.g. on a specific day)

 Note

It is assumed that the time feed for a notary is GPS/NaviStar time as defined by the atomic clocks at the US Naval Observatory. This time feed is extremely accurate and available globally for free.

Also see section 7 of the [Technical white paper](#) which covers this topic in significantly more depth.

HTTP Links of URL: <https://docs.corda.net/releases/release-M9.2/key-concepts-consensus-notaries.html>

Client IP: 192.99.56.22

<http://schema.org/Article>
<https://discourse.corda.net>
https://github.com/snide/sphinx_rtd_theme
<http://slack.corda.net>
<http://sphinx-doc.org/>
<https://readthedocs.org>
<https://www.google-analytics.com/analytics.js>