Module 7

Transactions

corda

Learning outcomes

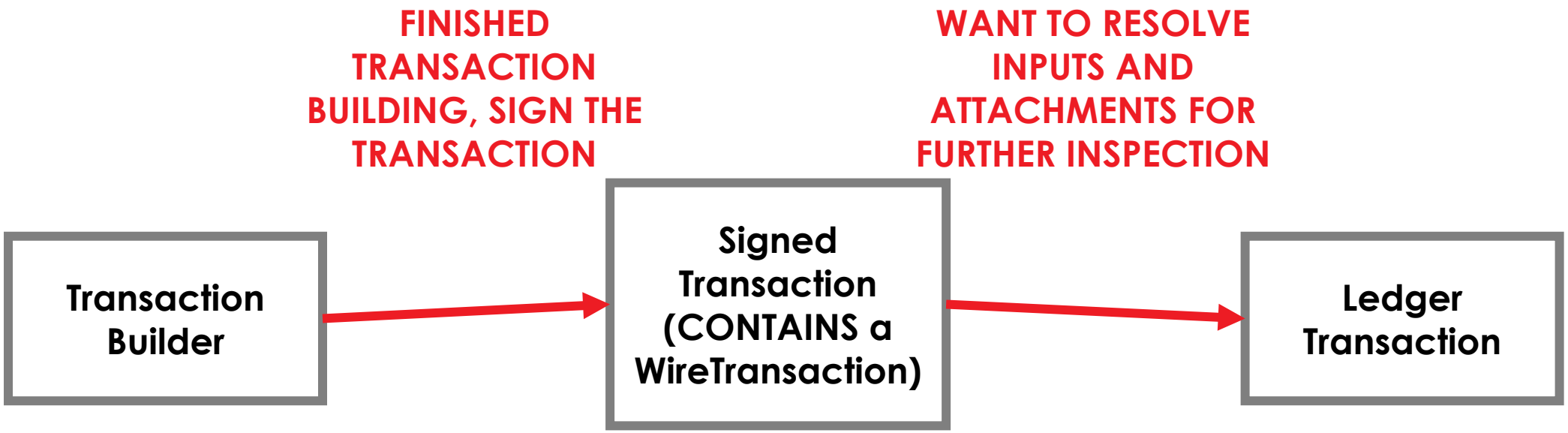
- Learn what states can a transaction occupy
- Learn how transactions are constructed
- Learn how we gather signatures over a transaction
- Learn how we verify a transaction

The transaction workflow

- To update the ledger, transactions go through the following stages:
  1. Building
  2. Initial node's verification and signature
  3. Other nodes' verification and signature
  4. Finalisation (notarisation and recording)

The transaction lifecycle

- During this process, the transaction will be represented by one of three classes:



TransactionBuilder

- `TransactionBuilder` is a **mutable** transaction class
- This is **unlike** the others which are all **immutable**!
- Peers can add any states, commands and other transaction components to the builder
- Once the `TransactionBuilder` is ready, we make it immutable by signing it to create a `SignedTransaction`

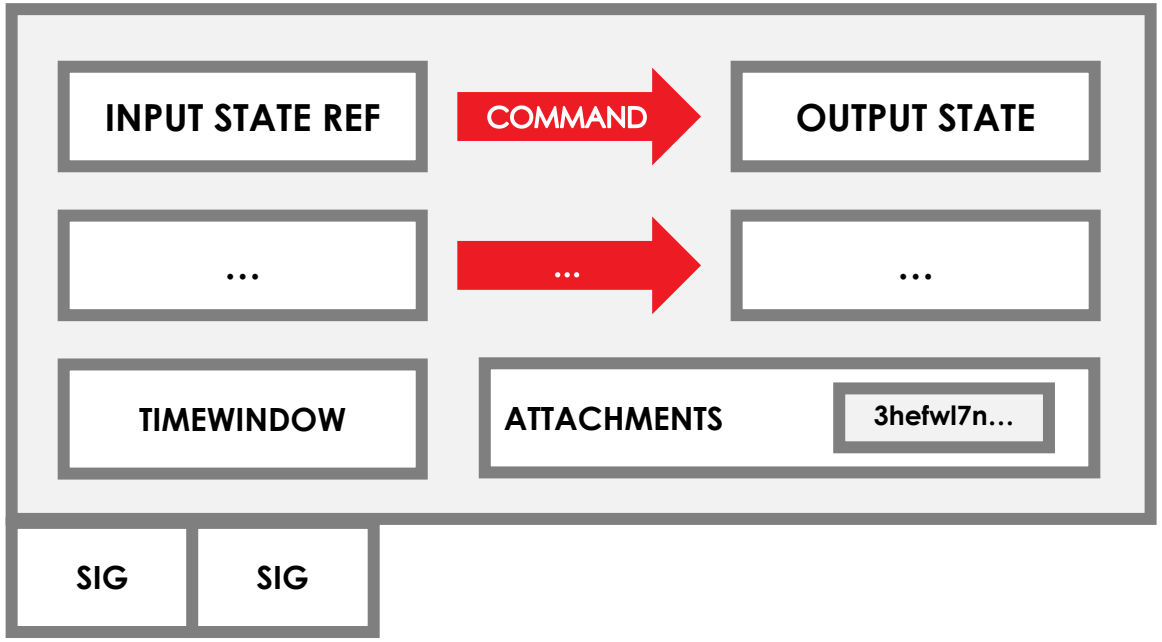
TransactionBuilder contents

The `TransactionBuilder` has the following key fields:

```
val inputs: MutableList<StateRef>
val outputs: MutableList<TransactionState<ContractState>>
val attachments: MutableList<SecureHash>
val commands: MutableList<Command>
var window: TimeWindow?
val signers: MutableSet<PublicKey>
var notary: Party?
```

TransactionBuilder contents

This corresponds to the transaction elements we saw in the transaction diagram in the Key Concepts section:



TransactionBuilder contents

`TransactionBuilder` fields:

- `inputs` is a list of input state references (`StateRef`s) to be consumed when the transaction is committed
- `outputs` is a list of output states of type `TransactionState`
- `attachments` is a list of attachment hashes

TransactionBuilder contents

- `commands` is a list of `Command`s
- `signers` is a list of the transaction's required signers
  - It corresponds to the union of the signers listed on each command
- `window` specifies the time-window during which the transaction must be notarised
- `Notary` is the identity of the notary who will notarise the transaction

StateRef

- Because Corda uses a UTXO model, transaction take as inputs the unconsumed outputs of previous transactions
- We reference these unconsumed outputs using **`StateRef`**s:

```
data class StateRef(val txhash: SecureHash, val index: Int)
```

- **`txhash`** is the hash of the transaction that created the unconsumed output state
- **`index`** is the index of the output state among the outputs of the transaction that created it

StateAndRef

A **StateAndRef** is a class that pairs a **StateRef** with the state it points to:

```
data class StateAndRef<out T : ContractState>(
    val state: TransactionState<T>,
    val ref: StateRef)
```

Commands

- **Command** is a class pairing a **CommandData** instance with a list of **PublicKey**s representing the required signers:

```
data class Command(val value: CommandData, val signers:
        List<PublicKey>) {
    init { require(signers.isNotEmpty()) }

    constructor(data: CommandData, key: PublicKey) :
        this(data, listOf(key))

    private fun commandDataToString() = /* Redacted. */
    override fun toString() = /* Redacted. */
}
```

Commands

- **`CommandData`** is a featureless marker interface that designates a class as a command. Commands can also include any other arbitrary data

- For commands without any additional data, you should use **`TypeOnlyCommandData`**:

```
abstract class TypeOnlyCommandData : CommandData {
    override fun equals(other: Any?) = other?.javaClass == javaClass
    override fun hashCode() = javaClass.name.hashCode()
}
```

Time-windows

- A transaction can include a **`TimeWindow`**:

```
data class TimeWindow(
    val fromTime: Instant?, val untilTime: Instant?)
```

- The notary will only sign the transaction if it processes it within the window set out by the **`TimeWindow`**
- If the time window is invalid, notary throws a **`NotaryException`**

Windows vs. points-in-time

- Time-windows are windows, rather than points in time, because there is no "true" time in a distributed system
- **`midpoint`** is provided for when an absolute time is needed:

```
val midpoint: Instant get() =
    after!! +
    Duration.between(after, before!!).dividedBy(2)
```

- The window can optionally be open at one end

Valid and invalid time-windows

After = 1,400,000,000
Before = 1,500,000,000

After = 1,400,000,000
Before = `null`

After = `null`
Before = 1,500,000,000

Using a TransactionBuilder

We create a **`TransactionBuilder`** as follows:

```
val notary = serviceHub.networkMapCache.notaryIdentities.first()
val txBuilder = TransactionBuilder(notary = notary)
```

Items are added to the transaction using **`withItems`**, which takes any number of transaction components of any type:

```
val state = Cash.StateAndContract(amount, owner)
val command = Command(Cash.Commands.Issue(), state.participants)
txBuilder.withItems(state, command) // Builder modified in-place
```

Additional TransactionBuilder methods

- `TransactionBuilder` also has many other methods to add transaction components individually - refer to the API docs for more information
- You can check that the transaction you are building is valid using `TransactionBuilder.verify()`, which will test the transaction against each input and output state's contract

SignedTransaction

- Once you are happy with the **`TransactionBuilder`**, you must finalise it by applying your signature
- You sign the builder via the **`ServiceHub`** (we'll discuss the **`ServiceHub`** in the module on flows):

```
ServiceHub.signInitialTransaction(builder: TransactionBuilder)
```

- This returns an immutable **`SignedTransaction`** for which we can start gathering the remaining required signatures

Verifying a SignedTransaction

- A `SignedTransaction` pairs a transaction with its signatures
- You can verify the transaction itself using:

```
SignedTransaction.verify()
```

- And you can verify the transaction's signatures using:

```
stx.checkSignaturesAreValid() // Are existing sigs valid?
stx.checkRequiredSignatures() // Are all required sigs present and valid?
stx.checkSignaturesExcept()   // Are all required sigs, except those
                                 listed, present and valid?
```

Signing a SignedTransaction

There are two ways to add a signature to an existing `SignedTransaction`:

- Using the ServiceHub to add the signature directly:

```
val stx2 = ServiceHub.addSignature(stx)
```

- Generating the signature and adding it manually:

```
val sig = ServiceHub.createSignature(stx)
val stx2 = stx.withAdditionalSignature(sig)
```

LedgerTransaction

A `LedgerTransaction` is derived from a `SignedTransaction`. It is the result of doing the following:

- Downloading and storing the transaction's dependencies and attachments
- Resolving the input `StateRef`s into `StateAndRef`s
- Performing lookups on the `Command`s to identify anonymous parties, and converting `Command`s into `AuthenticatedObject`s

Purpose of a LedgerTransaction

The advantage of a `LedgerTransaction` is that it allows us to perform additional validation over the transaction:

- We can inspect the contents of the input states
- We can assert conditions based on the inputs vs. the outputs (e.g. the amounts of input and output cash are the same)
- We can inspect the attachments

Where does transaction building happen?

- Transactions are built, signed and verified **inside flows**
- We will look at the flow API next…

Creating a transaction summary

- Create a transaction builder
- Gather and add any required inputs, outputs and other components
- Verify and sign the builder
- Gather additional signatures, with the counterparties verifying it and its existing signatures as required

Once this process is complete, we have a valid transaction ready to be notarised