



Module 8

Flows

c·rda



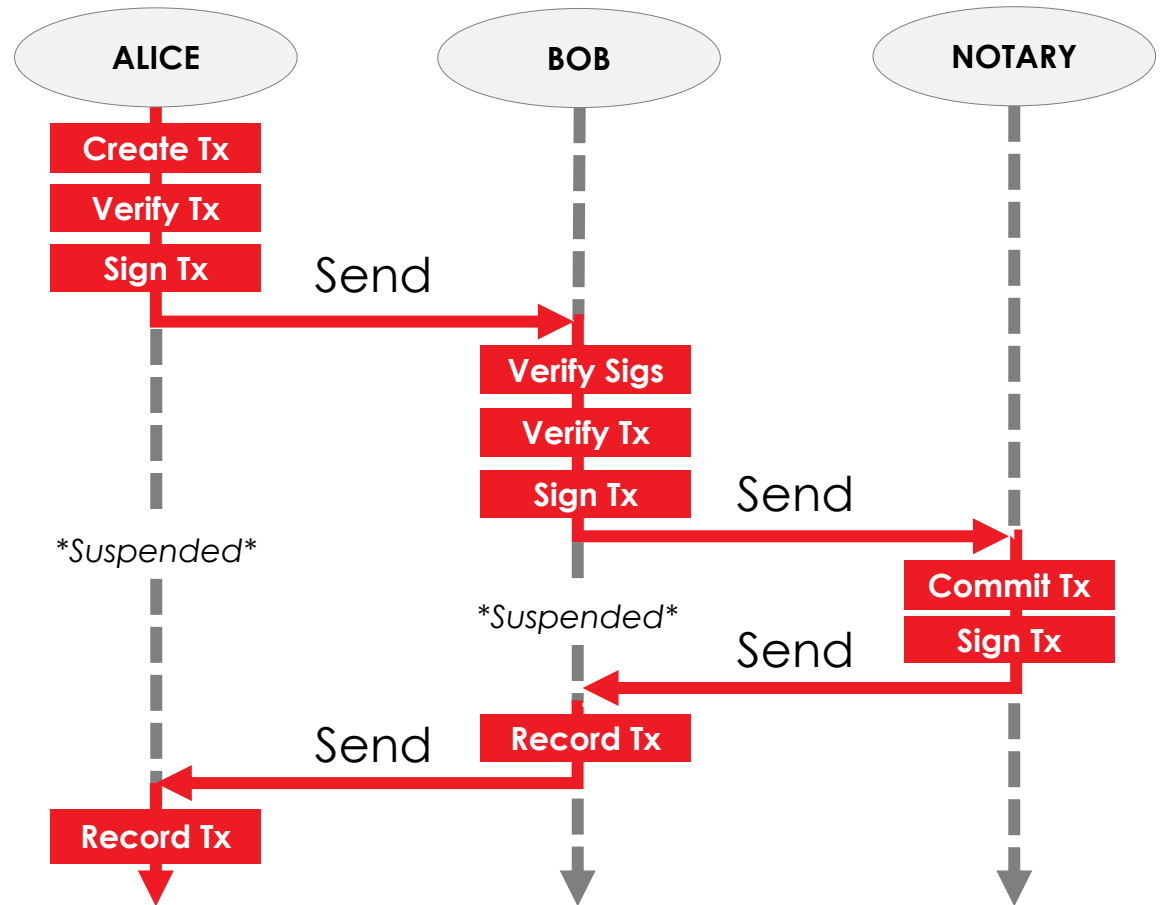
Learning outcomes

- Learn how flows allow the ledger update process to be automated
- Learn how flows are implemented in code
- Learn how to design your own flow

What is a flow?

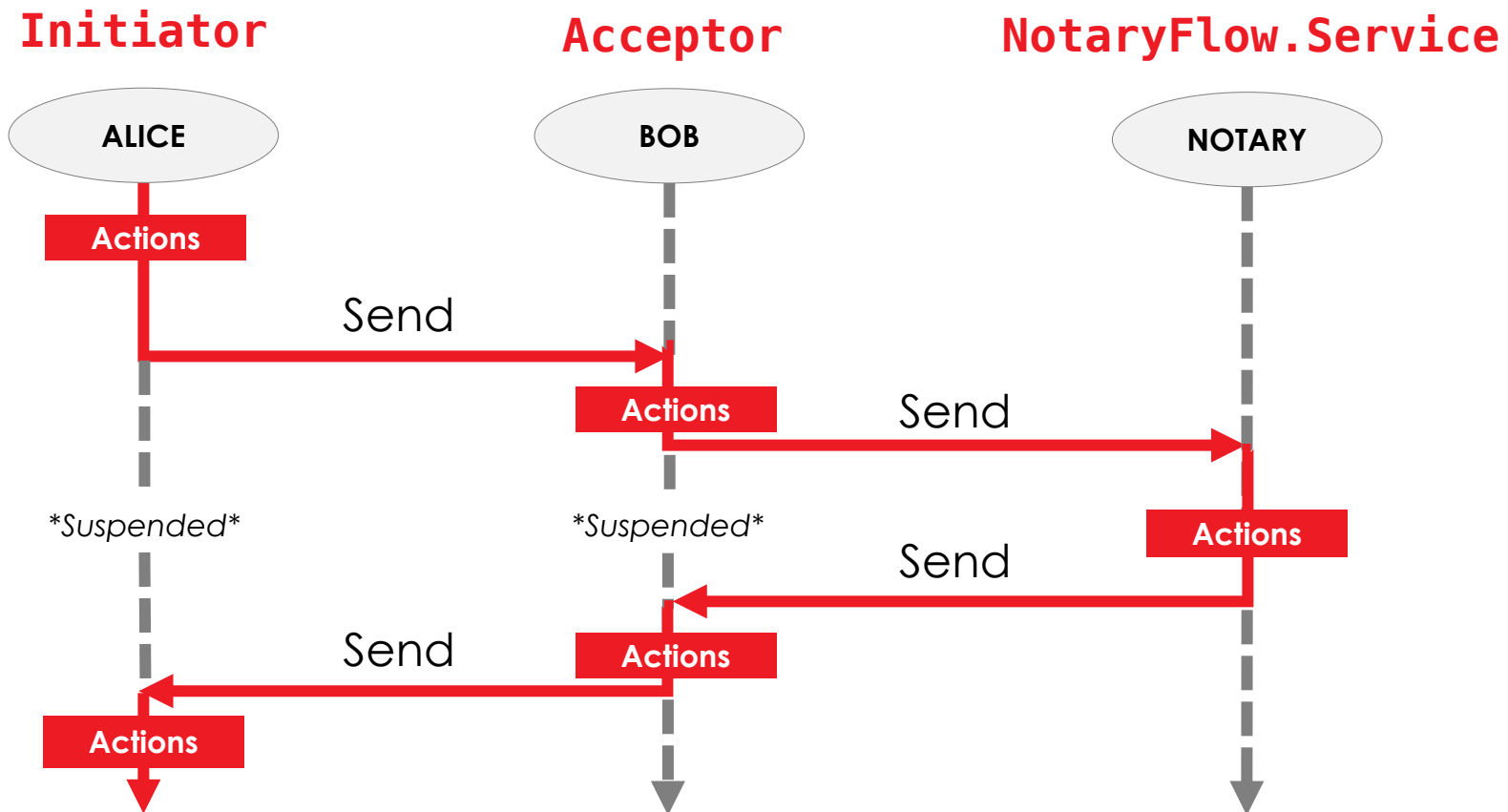
- Nodes communicate using structured sequences of messages called **flows**
- Flow steps are bilateral
- But any number of nodes can be involved in a single flow...
- ...and be informed of the flow's completion and resulting ledger updates

r3.



Flow-logics

Each party in a flow runs its own **flow-logic**:



Flow-logics

- Flow-logics are subclasses of the **FlowLogic** class that describe the flow from a single node's point-of-view:

```
abstract class FlowLogic<out T>
```

- A flow-logic's actions are defined by overriding **FlowLogic.call**:

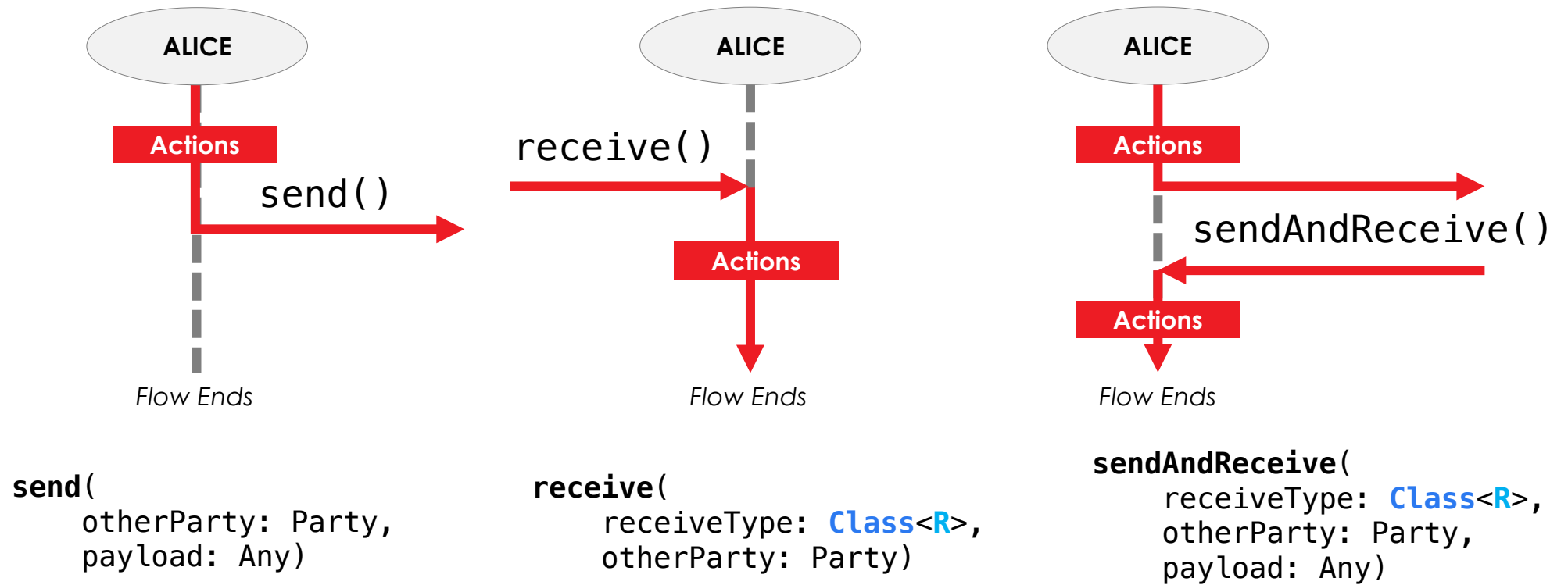
```
@Suspendable  
@Throws(FlowException::class)  
abstract fun call(): T
```

Services available to flows

- `call` can access the node's other services via the node's `ServiceHub`
- The services offered by the `ServiceHub` can be seen by inspecting `net.corda.core.node.ServiceHub`
- Some of the key services are:
 - `vaultService`, providing access to the node's vault
 - `identityService`, allowing node identities to be looked up
 - `myInfo`, allowing the node to provide info about itself

Inter-party communication

Flow-logics within a flow can communicate in three ways:



Registering flows

- Flows must be annotated to indicate how they can be used:
- **@InitiatingFlow** – flows that can be started directly
- **@StartableByRPC** – flows that can be started via RPC
- **@InitiatedBy(FlowLogic.class)** – flows that can only be kicked off via messages from other flows

Invoking response flow-logics

- When the counterparty node receives a message from a given flow, it responds by:
 1. Looking up the **FlowLogic** class that sent the message
 2. If the **FlowLogic** class is registered as part of an **@InitiatedBy** annotation, the node initiates the registered response **FlowLogic**
 3. Otherwise, the message is ignored

UntrustworthyData

- You cannot trust data received from other nodes – a counterparty could send you anything
- As a result, data received from other parties is wrapped in an **UntrustworthyData** instance:

```
class UntrustworthyData<out T>(  
    private val fromUntrustedWorld: T)
```

- The underlying data is accessed using **unwrap**:

```
inline fun <R> unwrap validator: (T) -> R = validator(data)
```

unwrap()

- **unwrap()** takes a lambda checking the data received from the counterparty
- Example:

```
unwrap { tx ->  
    // Check sigs and verify tx  
    val wireTx = tx.verifySignatures(myKey, notaryKey)  
    wireTx.toLedgerTransaction(serviceHub).verify()  
    tx  
}
```

Starting a flow

- Each node exposes a set of RPC (remote procedure call) operations to clients
- The **startFlowDynamic** RPC operation starts a flow-logic:

```
@RPCReturnsObservables  
fun <T : Any> startFlowDynamic(  
    logicType: Class<out FlowLogic<T>>,  
    vararg args: Any?)  
: FlowHandle<T>
```

- The flow-logic's **call** method is invoked, returning an observable that allows us to observe the flow's result



Flow session

- Interflow communications happen within a flow session
- Once created, flow session can and should be reused for all following communications
- Flow session can be initiated by invoking **initiateFlow(Party)**
- Flow session is injected through a constructor on the receiving side

ProgressTracker

- A flow-logic's progress is charted by overriding **FlowLogic.progressTracker**:

```
open val progressTracker: ProgressTracker? = null
```

- A **ProgressTracker** is constructed with a series of steps corresponding to the steps in the flow

```
class ProgressTracker(vararg steps: Step)
```

- And each **ProgressTracker.Step** is defined using a label:

```
open class Step(open val label: String)
```

- You advance through the progress-tracker steps using:

```
progressTracker.currentStep = myStep
```

ProgressTracker example

- An example **ProgressTracker** definition:

```
companion object {  
    object GENERATING_TX : Step("Generating transaction.")  
    object VERIFYING_TX  : Step("Verifying contract.")  
    ...  
    fun tracker() = ProgressTracker(  
        GENERATING_TX, VERIFYING_TX, ...)  
}  
  
override val progressTracker = tracker()
```

- The progress-tracker would then be updated using:

```
// Stage 1 – Generating a Transaction.  
progressTracker.currentStep = GENERATING_TX
```

ProgressTracker output

These progress-tracker steps will be shown in the terminal:

```

  _____
 /  _  _  _  \
/_  _  _  _  \
 \_  _  _  _  /
  \_  _  _  _ /
   \_  _  _  /
    \_  _  _ /
     \_  _  /
      \_  _ /
       \_  _/
        \_  /
         \_/

    "It's OK computer, I go to sleep after
    twenty minutes of inactivity too!"

--- DEVELOPER SNAPSHOT -----

Logs can be found in           : /Users/joeldudley/Desktop/cordapp-tutorial/kotlin-source/build/nodes/NodeA/logs
Database connection url is      : jdbc:h2:tcp://10.163.199.132:61491/node
Node listening on address       : localhost:10004
Loaded plugins                  : com.example.plugin.ExamplePlugin
Node started up and registered in 22.42 sec

[✓] Generating transaction based on new IOU.
[✓] Verifying contract constraints.
[✓] Signing transaction with our private key.
[✓] Sending proposed transaction to recipient for review.
[✓] Done

```


FlowException

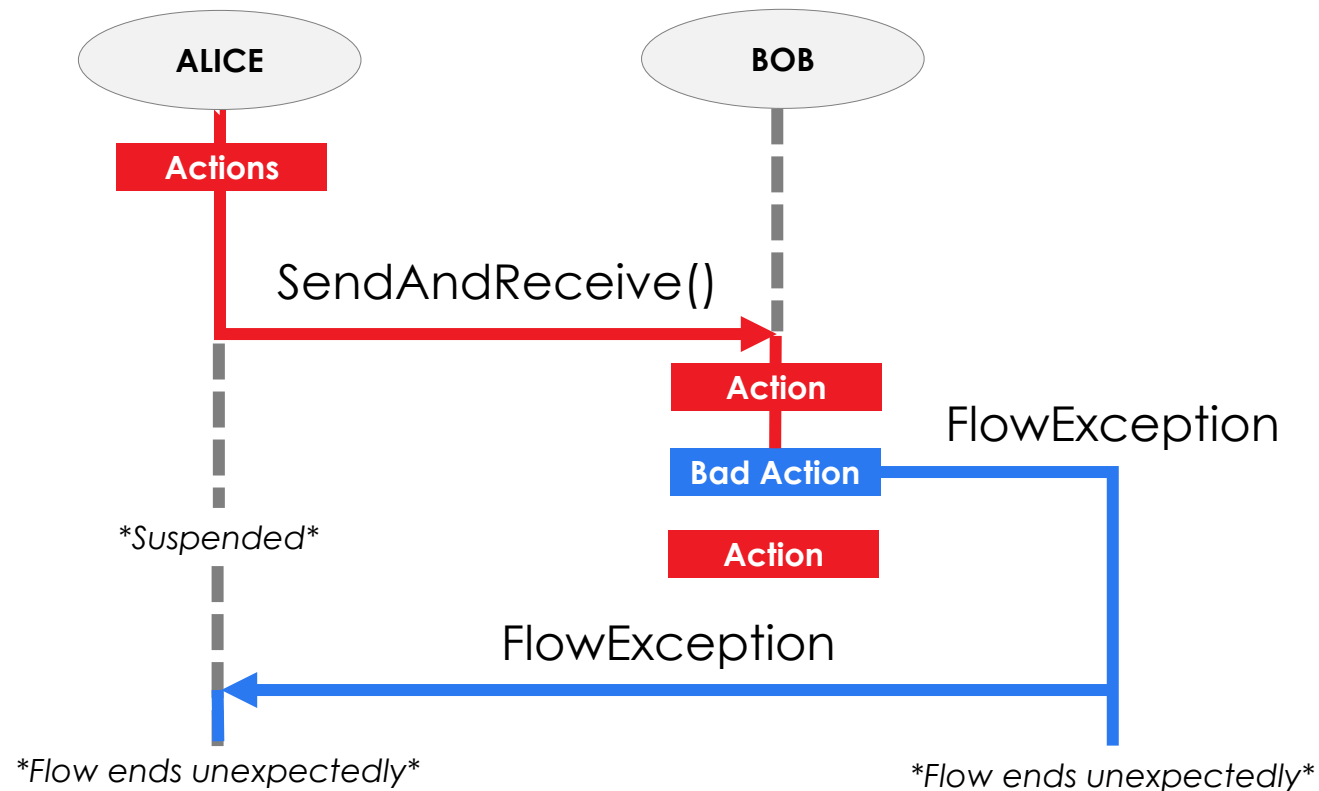
- Flows can throw a special kind of exception called **FlowException**:

```
open class FlowException @JvmOverloads constructor(  
    message: String? = null, cause: Throwable? = null)  
: Exception(message, cause)
```

- Unlike other exceptions, **FlowExceptions** are propagated to the flow's other active counterparties
- Parties can use **FlowExceptions** to let other parties know that they are not willing to proceed with the flow:
 - Signatures are not valid, or the transaction doesn't verify
 - The parameters of the deal are not as discussed
 - They are reneging on the deal...

FlowException diagram

FlowExceptions cause all participants' flows to end without executing any further steps:

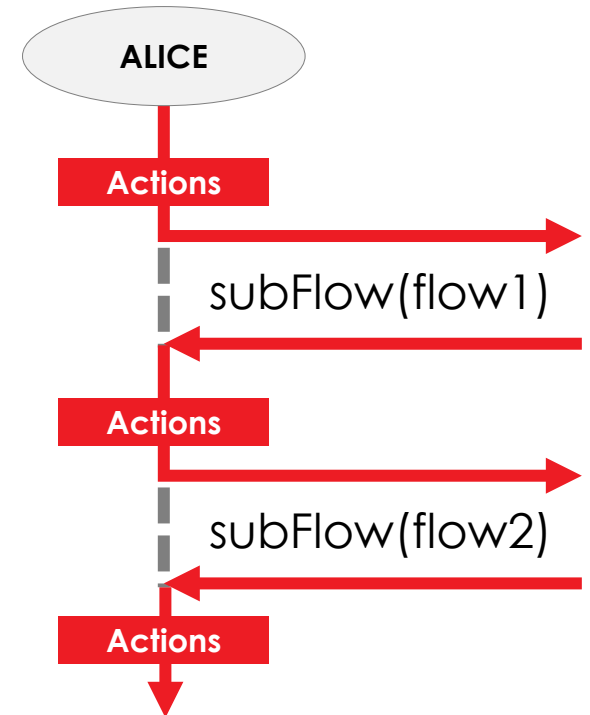


Subflows

- Within `call()`, a node can start additional *sub-flows*:

```
open fun <R> subFlow(  
    subLogic: FlowLogic<R>  
): R
```

- Once a sub-flow starts, the parent flow is suspended until the sub-flow returns
- Subflows are used to:
 - Abstract away common flow tasks
 - Allow more than two nodes to communicate within a given flow

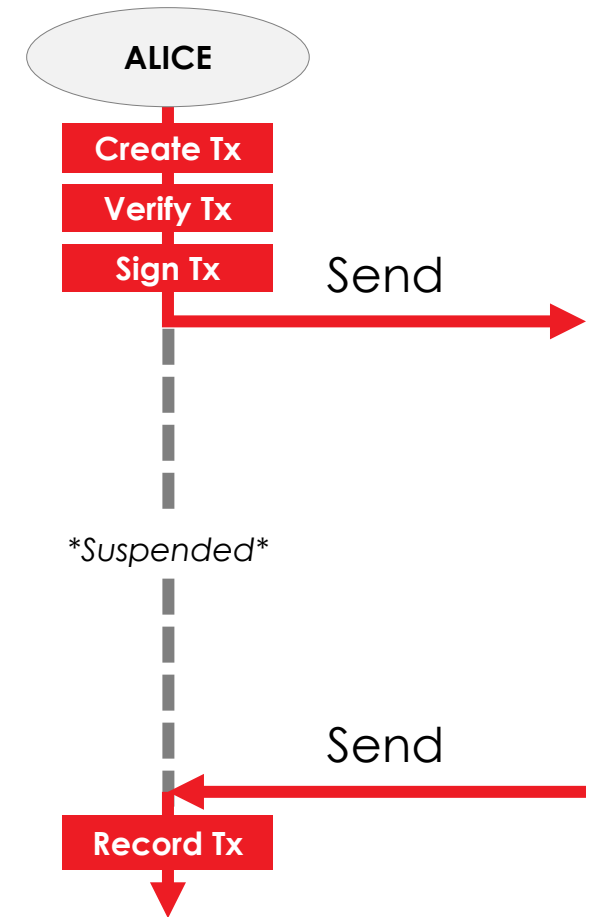


Built-in flows

- There are many useful built-in flows:
 - **FinalityFlow** notarizes and records a transaction
 - **SwapIdentitiesFlow** generates new identities with a party
 - **CollectSignaturesFlow** gathers all signatures on a transaction
 - **ResolveTransactionsFlow** walks the chain of transactions feeding into the current one, checking each for validity
 - **FetchDataFlow** retrieves data from peers
 - **FetchTransactionsFlow** retrieves transactions from peers
 - **NotaryChangeFlow** changes a transaction's notary
 - **TwoPartyDealFlow** a standard flow for creating two-party deals

Suspending flows

- Nodes need the ability to:
 - Handle millions of concurrent flows
 - Handle flows lasting for long periods of time
 - Survive restarts and upgrades
- To achieve this, flow-logics are regularly suspended and written to local storage:
 - When they are created
 - Whenever the network is called
 - While waiting on a counterparty
 - While waiting for human intervention



Writing a suspendable flow

- A flow-logic's **call** method is written in the normal blocking style:

```
override fun call(): SignedTransaction {  
    // Prepare a transaction...  
    flowSession.sendAndReceive(SignedTransaction,  
otherParty, partialTx)  
    // Handle the response...  
}
```

- To allow it to be suspended, **call** must be marked with an **@Suspendable** annotation
- Any functions that **call** invokes must also be marked **@Suspendable**

Suspending a flow

- At compile time, `call`'s bytecode is automatically instrumented to be suspendable using the Quasar library^[1]
- Once suspended, the flow is serialized using the Kryo framework^[2]
- Other networking issues such as session management, message receipts and delivery retries are also abstracted away



Flows in summary

- Flows allow nodes to communicate and agree on ledger updates
- Each flow is made up of two communicating, but untrusting, flow-logic subclasses
- Subflows are used to handle common tasks or create flows involving three or more parties
- Flows are suspended behind the scenes, allowing nodes to handle millions of concurrent flows

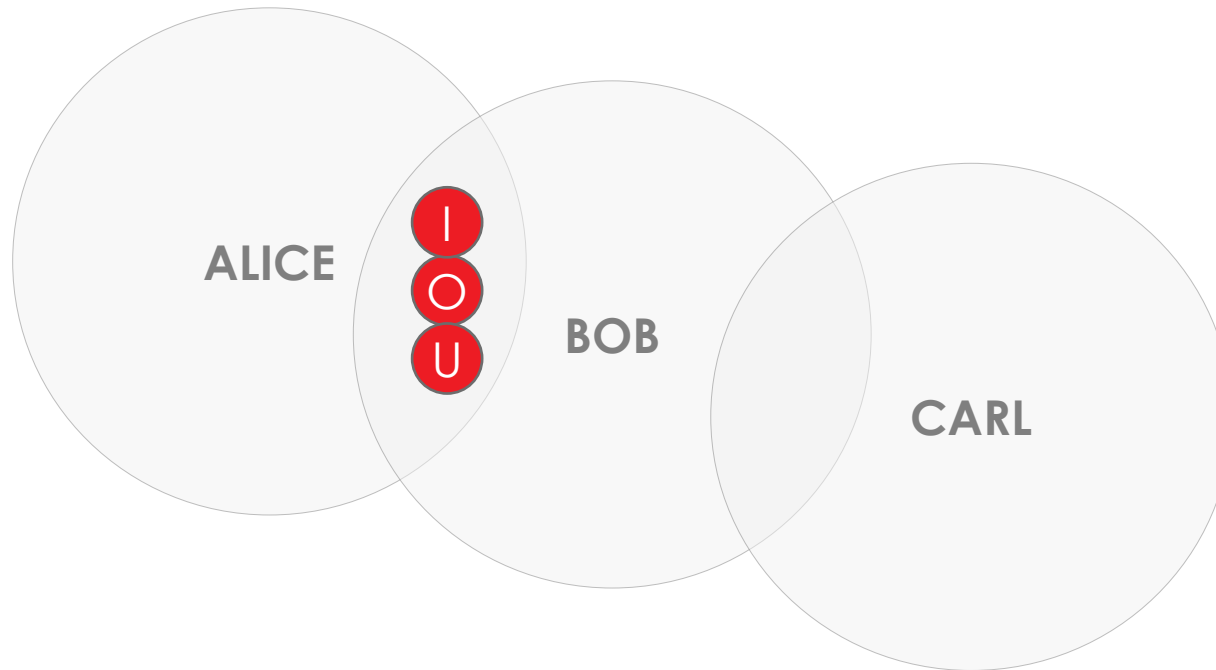


r3

Practical

The Goal of the IOU Flow

- Our flow must allow Alice and Bob to agree the creation of IOUs
- ...without Carl knowing about them



1. CorDapp Design

2. State

3. Contract

4. Flow

- Flow Tests
- Creating Signed TX
- Verifying TX
- Counterparty Sig.
- Finalizing TX
- ✓ Checkpoint

5. Network

6. API

Design Brainstorm



- What steps need to happen in a flow intended to create a new IOU?
- How should they be ordered?
- How should they be spread between participants?
 - Lender
 - Borrower
 - Notary

r3.

1. CorDapp Design

2. State

3. Contract

4. Flow

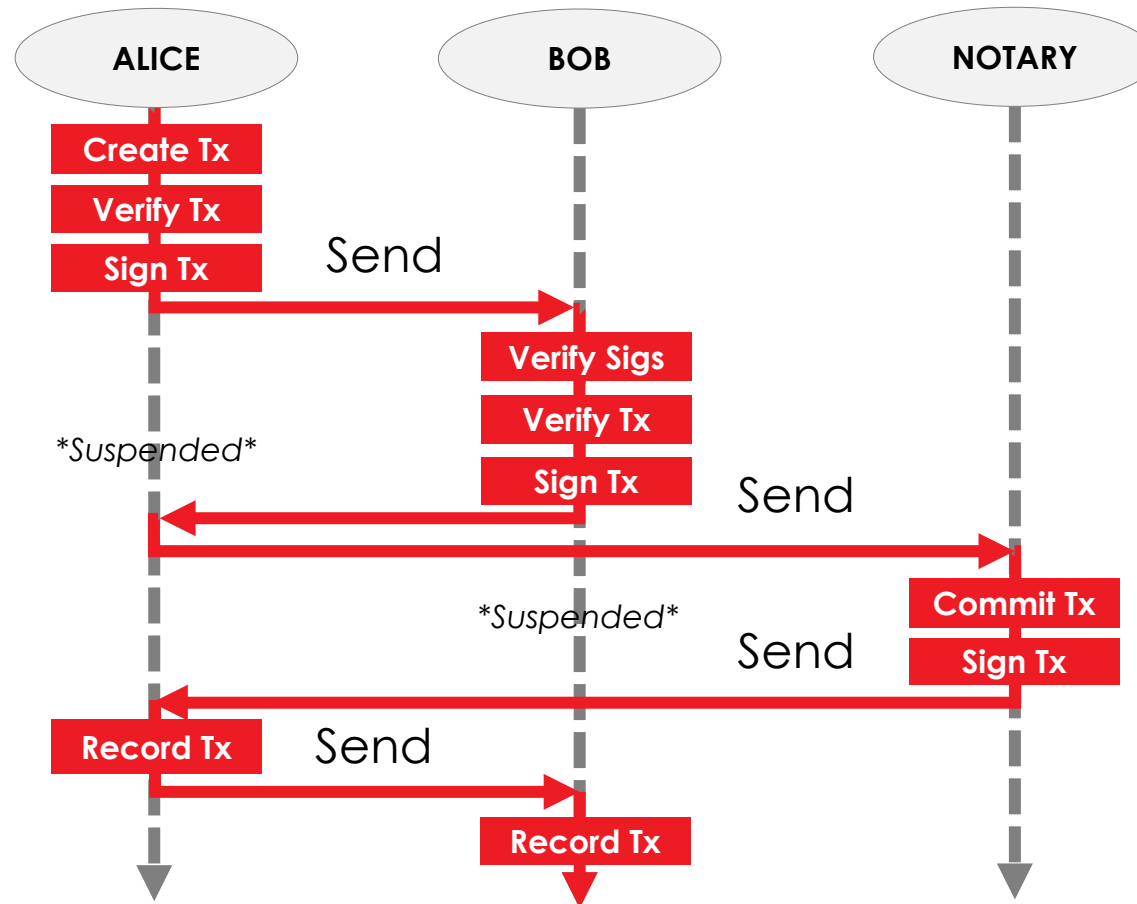
- Flow Tests
- Creating Signed TX
- Verifying TX
- Counterparty Sig.
- Finalizing TX
- ✓ Checkpoint

5. Network

6. API

The IOU Flow

- **IOUIssueFlow**, our IOU creation flow:



1. CorDapp Design

2. State

3. Contract

4. Flow

- Flow Tests
- Creating Signed TX
- Verifying TX
- Counterparty Sig.
- Finalizing TX
- ✓ Checkpoint

5. Network

6. API

Key Considerations

- **IOUIssueFlow** involves three parties:
 - The **Lender**
 - The **Borrower**
 - A notary
- **IOUIssueFlow** proceeds in three stages:
 1. The borrower party builds a new transaction and signs it
 2. The lender receives the partially-signed transaction, signs it, and sends it back to the borrower
 - This is handled by **SignTransactionFlow**
 3. The notary notarises the tx, and the other parties record it
 - This is handled by **FinalityFlow**

1. CorDapp Design

2. State

3. Contract

4. Flow

- Flow Tests
- Creating Signed TX
- Verifying TX
- Counterparty Sig.
- Finalizing TX
- ✓ Checkpoint

5. Network

6. API

Step 1 – Flow Tests

A low-angle, black and white photograph of a modern skyscraper with a complex, angular facade, viewed through a grid of small dots. The building's structure is composed of many sharp angles and lines, creating a sense of height and architectural complexity. The grid of dots is a light gray, semi-transparent overlay that covers the entire image, adding a technical or digital feel to the composition.

Flow Recap

- Our IOU issue flow is made up of two **FlowLogic** subclasses:
 - **IOUIssueFlow**
 - **IOUIssueFlowResponder**
- A node starts a flow by invoking its **call** method
- Flows are checkpointed at various points during their execution, so **call** must be marked **@Suspendable**

1. CorDapp Design

2. State

3. Contract

4. Flow

- Flow Tests
- Creating Signed TX
- Verifying TX
- Counterparty Sig.
- Finalizing TX
- ✓ Checkpoint

5. Network

6. API

The Flow Testbed

- Corda also provides a testing framework for flows
- The framework allows flows to be quickly tested using a mock network with mock nodes (with no spin-up time):

```
@Before
fun setup() {
    net = MockNetwork()
    val nodes = net.createSomeNodes(2)
    a = nodes.getPartyNodes[0]
    b = nodes.getPartyNodes[1]
    net.runNetwork(-1)
}
```

1. CorDapp Design

2. State

3. Contract

4. Flow

- Flow Tests
- Creating Signed TX
- Verifying TX
- Counterparty Sig.
- Finalizing TX
- ✓ Checkpoint

5. Network

6. API

Simulating Flows

- Flows are asynchronous, so instead of returning a value now, they return a **Future** – a placeholder for their eventual value
- Within a flow test, we obtain the **Future** representing the result of the flow's execution:

```
val flow = ExampleFlow.Initiator(state, b.info.legalIdentity)
```

```
val future = a.services.startFlow(flow).resultFuture
```

- We simulate the running of the network:

```
net.runNetwork()
```

- We can then retrieve the output of the flow:

```
future.get()
```



- Flow Tests
- Creating Signed TX
- Verifying TX
- Counterparty Sig.
- Finalizing TX
- ✓ Checkpoint

Writing Flow Tests

- We can test that a flow will throw an exception:

```
assertFailsWith<TransactionVerificationException> {  
    future.getOrThrow()  
}
```

- Or we can retrieve the flow's output and run assertions against it:

```
val signedTx = future.getOrThrow()  
assertEquals(signedTx.sigs.size(), 2)
```

1. CorDapp Design

2. State

3. Contract

4. Flow

- Flow Tests
- Creating Signed TX
- Verifying TX
- Counterparty Sig.
- Finalizing TX
- ✓ Checkpoint

5. Network

6. API



Our First Flow Test

- In our design, the **IOUIssueFlow** should return a **SignedTransaction**
- It does so by:
 1. Building a transaction
 2. Signing the transaction to convert it into a **SignedTransaction**
- We will test for this by requiring the party's signature on the **SignedTransaction** returned by **IOUIssueFlow**

1. CorDapp Design

2. State

3. Contract

4. Flow

- Flow Tests
- Creating Signed TX
- Verifying TX
- Counterparty Sig.
- Finalizing TX
- ✓ Checkpoint

5. Network

6. API

VerifySignatures()

- Our test uses `SignedTransaction`'s `verifySignaturesExcept` method:

```
verifySignaturesExcept(vararg CompositeKey allowedToBeMissing)
```

- Be careful with `verifySignaturesExcept`:
 - Its parameter is the list of signatures that it SHOULDN'T verify
 - In our case, that's the signatures of the `Lender` and the `notary`, since we are only gathering a signature from the `Borrower` for now

1. CorDapp Design

2. State

3. Contract

4. Flow

- Flow Tests
- Creating Signed TX
- Verifying TX
- Counterparty Sig.
- Finalizing TX
- ✓ Checkpoint

5. Network

6. API

Running the Flow Tests

- Flows require the Quasar fiber framework
 - This allows them to be suspended and resumed
- If your tests do not run due to an “instrumentation error” then add the following VM options to the default JUnit test runConfiguration:

`-ea -javaagent:./lib/quasar.jar`

r3.

1. CorDapp Design

2. State

3. Contract

4. Flow

- Flow Tests
- Creating Signed TX
- Verifying TX
- Counterparty Sig.
- Finalizing TX
- ✓ Checkpoint

5. Network

6. API

Our First Flow Test - Implementation



r3.

Goal	Run the flow test for the borrower's signature
Where?	flows/IOUIssueFlowTests.kt
Steps	<ol style="list-style-type: none">1. Uncomment the <code>flowReturnsCorrectlyFormedPartiallySignedTransaction</code> test2. Run the test, using the Green arrow to the left of the test name3. The test should fail.4. Use the instructions provided above to write a flow to make the test pass – it WON'T, yet. We've got more work to do.
Key Docs	https://docs.corda.net/flow-testing.html

1. CorDapp Design

2. State

3. Contract

4. Flow

- Flow Tests
- Creating Signed TX
- Verifying TX
- Counterparty Sig.
- Finalizing TX
- ✓ Checkpoint

5. Network

6. API



Step 2 – Creating a Signed TX

Creating a SignedTransaction

- To create a valid **SignedTransaction**, we need to:
 1. Get a reference to a notary
 2. Create a command
 3. Create a **TransactionBuilder**
 4. Add the **Issue** command and **IOUState**
 5. Convert the **TransactionBuilder** into a **SignedTransaction** by signing the builder

1. CorDapp Design

2. State

3. Contract

4. Flow

- Flow Tests
- Creating Signed TX
- Verifying TX
- Counterparty Sig.
- Finalizing TX
- ✓ Checkpoint

5. Network

6. API

Choosing a Notary

- Let's start by choosing a notary
- Within `call`, you can use the `ServiceHub` to access the node's services, include the network map
- To retrieve a notary, we use:

```
serviceHub  
    .networkMapCache  
    .notaryIdentities.first()
```

1. CorDapp Design

2. State

3. Contract

4. Flow

- Flow Tests
- Creating Signed TX
- Verifying TX
- Counterparty Sig.
- Finalizing TX
- ✓ Checkpoint

5. Network

6. API

Choosing a Notary - Implementation



Goal	Retrieve the notary for the transaction
Where?	IOUIssueFlow.kt
Steps	1. Write the code to obtain a reference to the notary who will notarize the transaction
Key Docs	N/A

r3.

1. CorDapp Design

2. State

3. Contract

4. Flow

- Flow Tests
- Creating Signed TX
- Verifying TX
- Counterparty Sig.
- Finalizing TX
- ✓ Checkpoint

5. Network

6. API

Choosing a Notary - Solution



Goal	Retrieve the notary's identity in <code>call</code>
Steps	<ul style="list-style-type: none">• Get a reference to the <code>ServiceHub</code> object• Retrieve the list of notaries• Extract the first (and only) notary from the list• Retrieve its identity
Code	<pre>val notary = serviceHub .networkMapCache .notaryIdentities.first()</pre>

r3.

1. CorDapp Design

2. State

3. Contract

4. Flow

- Flow Tests
- Creating Signed TX
- Verifying TX
- Counterparty Sig.
- Finalizing TX
- ✓ Checkpoint

5. Network

6. API

Creating a Command

- Next, we need to create our transaction's **Command**
- **Commands** associate **CommandData** with a set of signers:

```
Command(  
    cd: CommandData,  
    requiredSigners: List<CompositeKey>  
);
```

- We are going to use the **Issue ComandData** class we defined earlier
- The lender and the borrower will be the signers

1. CorDapp Design

2. State

3. Contract

4. Flow

- Flow Tests
- Creating Signed TX
- Verifying TX
- Counterparty Sig.
- Finalizing TX
- ✓ Checkpoint

5. Network

6. API

Creating a Command - Implementation



Goal	Instantiate a Issue command
Where?	IOUIssueFlow.kt
Steps	1. Write the code to instantiate a Issue command
Key Docs	N/A

r3.

1. CorDapp Design

2. State

3. Contract

4. Flow

- Flow Tests
- Creating Signed TX
- Verifying TX
- Counterparty Sig.
- Finalizing TX
- ✓ Checkpoint

5. Network

6. API

Creating a Command - Solution



Goal	Create a Issue command instance in call
Steps	<ul style="list-style-type: none">Construct the Command object using:<ul style="list-style-type: none">The IOUContract.Issue CommandData subclassThe list of the IOUState's participants
Code	<pre>val issueCommand = Command(IOUContract.Commands.Issue(), state.participants.map { it.owningKey });</pre>

1. CorDapp Design

2. State

3. Contract

4. Flow

- Flow Tests
- Creating Signed TX
- Verifying TX
- Counterparty Sig.
- Finalizing TX
- ✓ Checkpoint

5. Network

6. API

Building the Transaction

- We create the transaction itself using the **TransactionBuilder** class
- **TransactionBuilder** is initialised with the following syntax:

```
TransactionBuilder(notary = notary)
```

- We add items to the **TransactionBuilder** using **withItems**, which takes a variable-length list of:
 - **StateAndRefs** (input state references)
 - **ContractStates** (output states)
 - **Commands**

1. CorDapp Design

2. State

3. Contract

4. Flow

- Flow Tests
- Creating Signed TX
- Verifying TX
- Counterparty Sig.
- Finalizing TX
- ✓ Checkpoint

5. Network

6. API

Building a Transaction - Implementation



Goal	Construct the transaction
Where?	IOUIssueFlow.kt
Steps	<ol style="list-style-type: none">1. Write the code to create a TransactionBuilder with:<ul style="list-style-type: none">• The selected notary• The Issue command• The IOU state
Key Docs	N/A

r3.

1. CorDapp Design

2. State

3. Contract

4. Flow

- Flow Tests
- Creating Signed TX
- Verifying TX
- Counterparty Sig.
- Finalizing TX
- ✓ Checkpoint

5. Network

6. API

Building the Transaction - Solution



Goal	Create a TransactionBuilder with the IOUState and Issue command
Steps	<ul style="list-style-type: none">• Create a new TransactionBuilder using the notary's identity• Add two items to the transaction:<ul style="list-style-type: none">• The TransactionState with IOUState and a reference to IOUContract• The Issue command
Code	<pre>val builder = TransactionBuilder(notary = notary) builder.withItems(StateAndContract(state, IOUContract.IOU_CONTRACT_ID), issueCommand)</pre>

r3.

1. CorDapp Design

2. State

3. Contract

4. Flow

- Flow Tests
- Creating Signed TX
- Verifying TX
- Counterparty Sig.
- Finalizing TX
- ✓ Checkpoint

5. Network

6. API

Signing a Transaction

- We now need to convert the **TransactionBuilder** into a **SignedTransaction**:
 - **TransactionBuilder** is a mutable transaction-in-construction
 - **SignedTransaction** is immutable because it has been signed
- You sign the builder using the ServiceHub, which has access to the node's keypair:

```
val signedTx = serviceHub.signInitialTransaction(builder)
```

1. CorDapp Design

2. State

3. Contract

4. Flow

- Flow Tests
- Creating Signed TX
- Verifying TX
- Counterparty Sig.
- Finalizing TX
- ✓ Checkpoint

5. Network

6. API

Flow Return Types

- Each flow-logic returns a value of a specific type from its **call** method
- To change a flow-logic's return type, you must:
 - Change the generic type of the FlowLogic it inherits from
 - Annotate **call** with the desired return type
- In our case, we need to modify our flow-logic to return a **SignedTransaction**

1. CorDapp Design

2. State

3. Contract

4. Flow

- Flow Tests
- Creating Signed TX
- Verifying TX
- Counterparty Sig.
- Finalizing TX
- ✓ Checkpoint

5. Network

6. API

Signing a Transaction - Implementation



Goal	Return the transaction signed by the Initiator
Where?	IOUIssueFlow.kt
Steps	<ol style="list-style-type: none">1. Write the code to create and return a SignedTransaction2. Run the flowReturnsCorrectlyFormedPartiallySignedTransaction test3. The test should finally pass!
Key Docs	N/A

r3.

1. CorDapp Design

2. State

3. Contract

4. Flow

- Flow Tests
- Creating Signed TX
- Verifying TX
- Counterparty Sig.
- Finalizing TX
- ✓ Checkpoint

5. Network

6. API

Signing a Transaction - Solution



Goal	Sign the in-construction transaction to generate a SignedTransaction
Steps	<ul style="list-style-type: none">Sign the TransactionBuilder to convert it into a SignedTransaction
Code	<pre>val ptx = serviceHub.signInitialTransaction(builder)</pre>

- Flow Tests
- Creating Signed TX
- Verifying TX
- Counterparty Sig.
- Finalizing TX
- ✓ Checkpoint



Step 3 – Verifying the TX

Verifying a Transaction

- Before sending the transaction to the counterparty, the **Initiator** also needs to verify it
- We want to verify our transaction **BEFORE** signing it, to ensure that we're not signing an invalid transaction
- We verify the **TransactionBuilder** by invoking its **verify** function
- **verify** will invoke the contract logic of every input and output, without checking required signatures

1. CorDapp Design

2. State

3. Contract

4. Flow

- Flow Tests
- Creating Signed TX
- Verifying TX
- Counterparty Sig.
- Finalizing TX
- ✓ Checkpoint

5. Network

6. API

Verifying a Transaction - Implementation



Goal	Make the Initiator verify the transaction
Where?	flow/IOUIssueFlow.kt, after creating the TransactionBuilder , but before signing it test/flows/IOUIssueFlowTests.kt
Steps	1. Add the transaction verification code to the IOUIssueFlow.call method
Key Docs	N/A

r3.

1. CorDapp Design

2. State

3. Contract

4. Flow

- Flow Tests
- Creating Signed TX
- Verifying TX
- Counterparty Sig.
- Finalizing TX
- ✓ Checkpoint

5. Network

6. API

Verifying a Transaction - Solution



Goal	Verify the SignedTransaction
Steps	<ul style="list-style-type: none">• Call the TransactionBuilder's verify method
Code	<pre>override fun call(): SignedTransaction { "" builder.verify() "" }</pre>

1. CorDapp Design

2. State

3. Contract

4. Flow

- Flow Tests
- Creating Signed TX
- Verifying TX
- Counterparty Sig.
- Finalizing TX
- ✓ Checkpoint

5. Network

6. API

A low-angle, black and white photograph of a modern skyscraper with a complex, geometric facade, viewed through a grid of small dots. The building's structure is composed of many sharp angles and lines, creating a sense of height and architectural complexity. The grid of dots is a light gray, semi-transparent overlay that covers the entire image, adding a technical or digital feel to the composition.

Step 4 – Counterparty Sig.

Flow Communication

- IOU creation also requires communication with the other party in the flow
- We can use the pre-built **SignTransactionFlow** to collect the signatures from the required counterparties for US
- It takes a **SignedTransaction** (already signed by the calling node) as a parameter and returns a **SignedTransaction** with all the specified counterparty signatures represented by a **FlowSession**
- This flow allows you to ignore the details of the back-and-forth communication required to collect all the signatures, thus simplifying your flow

r3.

1. CorDapp Design

2. State

3. Contract

4. Flow

- Flow Tests
- Creating Signed TX
- Verifying TX
- Counterparty Sig.
- Finalizing TX
- ✓ Checkpoint

5. Network

6. API

SignTransactionFlow

- **SignTransactionFlow(stx: SignedTransaction, sessions: Collection<FlowSession>)** does two things:
 - Verifies the transaction and our signature
 - Collects, adds and verifies all the signatures from required counterparties
 - An exception is thrown if any signatures are missing or invalid
- **SignTransactionFlow** is invoked using **subFlow(fl: FlowLogic)**

1. CorDapp Design

2. State

3. Contract

4. Flow

- Flow Tests
- Creating Signed TX
- Verifying TX
- Counterparty Sig.
- Finalizing TX
- ✓ Checkpoint

5. Network

6. API

SignTransactionFlow- Implementation



r3.

Goal	Send the SignedTransaction from the Initiator to the Acceptor and back
Where?	flow/IOUIssueFlow.kt test/flows/IOUIssueFlowTests.kt
Steps	<ol style="list-style-type: none">1. Uncomment the flowReturnsTransactionSignedByBothParties test2. Test should fail3. Modify IOUIssueFlow.kt to make the test pass
Key Docs	N/A

1. CorDapp Design

2. State

3. Contract

4. Flow

- Flow Tests
- Creating Signed TX
- Verifying TX
- Counterparty Sig.
- Finalizing TX
- ✓ Checkpoint

5. Network

6. API

SignTransactionFlow- Solution



Goal	Send the SignedTransaction to the other Party for signing
Steps	<ol style="list-style-type: none">1. Review the SignTransactionFlow2. Use the SignTransactionFlow, which takes a SignedTransation as a parameter and returns a SignedTransction
Code	<pre>val sessions = (state.participants - ourIdentity).map { initiateFlow(it) }.toSet() val stx = subFlow(CollectSignaturesFlow(ptx))</pre>

1. CorDapp Design

2. State

3. Contract

4. Flow

- Flow Tests
- Creating Signed TX
- Verifying TX
- Counterparty Sig.
- Finalizing TX
- ✓ Checkpoint

5. Network

6. API



Step 5 – Finalizing the TX

Transaction Finalization Test

- The valid, signed IOU transaction now needs to be notarized and recorded in every participant's vault
- We will test this by ensuring that the state returned by the Initiator's flow matches the state recorded in both nodes' vaults
- We can access the transactions in a node's vault using the following syntax:

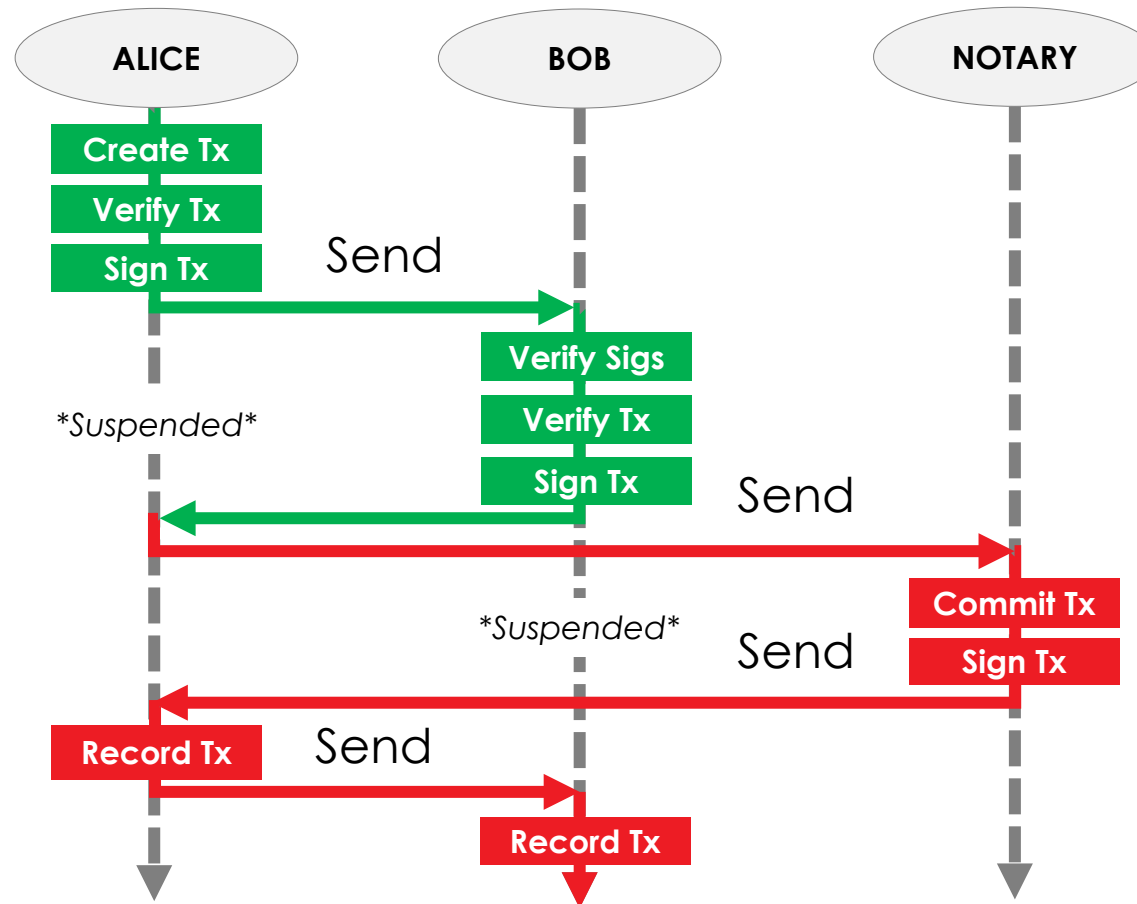
```
nodeName
  .services
  .validatedTransactions
  .getTransaction(id: SecureHash)
```



- Flow Tests
- Creating Signed TX
- Verifying TX
- Counterparty Sig.
- Finalizing TX
- ✓ Checkpoint

Transaction Finalization

- The finalization process involves another 7 steps!



1. CorDapp Design

2. State

3. Contract

4. Flow

- Flow Tests
- Creating Signed TX
- Verifying TX
- Counterparty Sig.
- Finalizing TX
- ✓ Checkpoint

5. Network

6. API

FinalityFlow

- Fortunately, we can use Corda's built-in **FinalityFlow** here to finalize the transactions
- **FinalityFlow(stx: SignedTransaction, participants: Set<Party>)** does two things:
 - It notarizes a transaction
 - It records the transaction in every participant's vault
- **FinalityFlow** is invoked from the **Acceptor's FlowLogic** using **subFlow(fl: FlowLogic)**



1. CorDapp Design

2. State

3. Contract

4. Flow

- Flow Tests
- Creating Signed TX
- Verifying TX
- Counterparty Sig.
- Finalizing TX
- ✓ Checkpoint

5. Network

6. API

Finalizing a Transaction - Implementation



Goal	Notarize and record the transaction in every participant's vault
Where?	flow/IOUIssueFlow.kt test/flows/IOUIssueFlowTests.kt
Steps	<ol style="list-style-type: none">1. Uncomment the following tests: flowRecordsTheSameTransactionInBothPartyVaults2. The tests should fail3. Modify IOUIssueFlow.kt to make the tests pass
Key Docs	N/A

r3.

1. CorDapp Design

2. State

3. Contract

4. Flow

- Flow Tests
- Creating Signed TX
- Verifying TX
- Counterparty Sig.
- Finalizing TX
- ✓ Checkpoint

5. Network

6. API

Finalizing a Transaction - Solution



Goal	Notarize the transaction and record it in the participants' vaults
Steps	<ul style="list-style-type: none">• Invoke the FinalityFlow• Return the resulting fully-signed SignedTransaction
Code	<pre>val ftx = subFlow(FinalityFlow(stx)) return ftx</pre>



1. CorDapp Design

2. State

3. Contract

4. Flow

- Flow Tests
- Creating Signed TX
- Verifying TX
- Counterparty Sig.
- Finalizing TX
- ✓ Checkpoint

5. Network

6. API

Over to you!

There are two more flows to complete!

Follow the instructions in
IOUTransferFlow and **IOUTSettleFlow**

r3.

1. CorDapp Design

2. State

3. Contract

4. Flow

- Flow Tests
- Creating Signed TX
- Verifying TX
- Counterparty Sig.
- Finalizing TX
- ✓ Checkpoint

5. Network

6. API

A low-angle, black and white photograph of a modern skyscraper with a complex, geometric facade, viewed through a grid of small dots. The building's structure is composed of many sharp angles and lines, creating a sense of height and architectural complexity. The grid of dots is a light gray, semi-transparent overlay that covers the entire image, adding a technical or digital feel to the composition.

Checkpoint – Progress So Far



Our progress so far

- We now have a flow allowing parties to issue and agree on new IOUs
- But we have no way of interacting with our nodes to get them to run these flows
- We will solve this by adding a REST API to our nodes
- But first, we're going to examine how we deploy and run the nodes that will be executing our CorDapp