Building transactions

Introduction

Understanding and implementing transactions in Corda is key to building and implementing real world smart contracts. It is only through construction of valid Corda transactions containing appropriate data that nodes on the ledger can map real world business objects into a shared digital view of the data in the Corda ledger. More importantly as the developer of new smart contracts it is the code which determines what data is well formed and what data should be rejected as mistakes, or to prevent malicious activity. This document details some of the considerations and APIs used to when constructing transactions as part of a flow.

The Basic Lifecycle Of Transactions

Transactions in Corda are constructed in stages and contain a number of elements. In particular a transaction's core data structure is the `net.corda.core.transactions.WireTransaction`, which is usually manipulated via a `net.corda.core.contracts.General.TransactionBuilder` and contains:

1. A set of Input state references that will be consumed by the final accepted transaction.

2. A set of Output states to create/replace the consumed states and thus become the new latest versions of data on the ledger.

3. A set of `Attachment` items which can contain legal documents, contract code, or private encrypted sections as an extension beyond the native contract states.

4. A set of `Command` items which give a context to the type of ledger transition that is encoded in the transaction. Also each command has an associated set of signer keys, which will be required to sign the transaction.

5. A signers list, which is populated by the `TransactionBuilder` to be the union of the signers on the individual Command objects.

6. A notary identity to specify the Notary node which is tracking the state consumption. (If the input states are registered with different notary nodes the flow will have to insert additional `NotaryChange` transactions to migrate the states across to a consistent notary node, before being allowed to mutate any states.)

7. Optionally a timestamp that can used in the Notary to time bound the period in which the proposed transaction stays valid.

Typically, the `WireTransaction` should be regarded as a proposal and may need to be exchanged back and forth between parties before it can be fully populated. This is an immediate consequence of the Corda privacy model, which means that the input states are likely to be unknown to the other node.

Once the proposed data is fully populated the flow code should freeze the `WireTransaction` and form a `SignedTransaction`. This is key to the ledger agreement process, as once a flow has attached a node's signature it has stated that all details of the transaction are acceptable to it. A flow should take care not to attach signatures to intermediate data, which might be maliciously used to construct a different `SignedTransaction`. For instance in a foreign exchange scenario we shouldn't send a `SignedTransaction` with only our sell side populated as that could be used to take the money without the expected return of the other currency. Also, it is best practice for flows to receive back the `DigitalSignature.WithKey` of other parties rather than a full `SignedTransaction` objects, because otherwise we have to separately check that this is still the same `SignedTransaction` and not a malicious substitute.

The final stage of committing the transaction to the ledger is to notarise the `SignedTransaction`, distribute this to all appropriate parties and record the data into the ledger. These actions are best delegated to the `FinalityFlow`, rather than calling the individual steps manually. However, do note that the final broadcast to the other nodes is asynchronous, so care must be used in unit testing to correctly await the Vault updates.

Gathering Inputs

One of the first steps to forming a transaction is gathering the set of input references. This process will clearly vary according to the nature of the business process being captured by the smart contract and the parameterised details of the request. However, it will generally involve searching the Vault via the `VaultService` interface on the `ServiceHub` to locate the input states.

To give a few more specific details consider two simplified real world scenarios. First, a basic foreign exchange Cash transaction. This transaction needs to locate a set of funds to exchange. A flow modelling this is implemented in `FxTransactionBuildTutorial.kt`. Second, a simple business model in which parties manually accept, or reject each other's trade proposals which is implemented in `WorkflowTransactionBuildTutorial.kt`. To run and explore these examples using the IntelliJ IDE one can run/step the respective unit tests in `FxTransactionBuildTutorialTest.kt` and `WorkflowTransactionBuildTutorialTest.kt`, which drive the flows as part of a simulated in-memory network of nodes.

> **Note**
>
> Before creating the IntelliJ run configurations for these unit tests go to Run -> Edit Configurations -> Defaults -> JUnit, add `-javaagent:lib/quasar.jar -Dco.paralleluniverse.fibers.verifyInstrumentation` to the VM options, and set Working directory to `$PROJECT_DIR$` so that the `Quasar` instrumentation is correctly configured.

For the Cash transaction let's assume the cash resources are using the standard `CashState` in the `:financial` Gradle module. The Cash contract uses `FungibleAsset` states to model holdings of interchangeable assets and allow the split/merge and summing of states to meet a contractual obligation. We would normally use the `generateSpend` method on the `VaultService` to gather the required amount of cash into a `TransactionBuilder`, set the outputs and move command. However, to elucidate more clearly example flow code is shown here that will manually carry out the inputs queries using the lower level `VaultService`.

```
// This is equivalent to the VaultService.generateSpend
// Which is brought here to make the filtering logic more visible in the example
private fun gatherOurInputs(serviceHub: ServiceHub,
                            amountRequired: Amount<Issued<Currency>>,
                            notary: Party?): Pair<List<StateAndRef<Cash.State>>, Long> {
    // Collect cash type inputs
    val cashStates = serviceHub.vaultService.unconsumedStates<Cash.State>()
    // extract our identity for convenience
    val ourIdentity = serviceHub.myInfo.legalIdentity
    // Filter down to our own cash states with right currency and issuer
    val suitableCashStates = cashStates.filter {
        val state = it.state.data
        (state.owner == ourIdentity)
                && (state.amount.token == amountRequired.token)
    }
    require(!suitableCashStates.isEmpty()) { "Insufficient funds" }
    var remaining = amountRequired.quantity
    // We will need all of the inputs to be on the same notary.
    // For simplicity we just filter on the first notary encountered
    // A production quality flow would need to migrate notary if the
    // the amounts were not sufficient in any one notary
    val sourceNotary: Party = notary ?: suitableCashStates.first().state.notary

    val inputsList = mutableListOf<StateAndRef<Cash.State>>()
    // Iterate over filtered cash states to gather enough to pay
    for (cash in suitableCashStates.filter { it.state.notary == sourceNotary }) {
        inputsList += cash
        if (remaining <= cash.state.data.amount.quantity) {
            return Pair(inputsList, cash.state.data.amount.quantity - remaining)
        }
        remaining -= cash.state.data.amount.quantity
    }
    throw IllegalStateException("Insufficient funds")
}
```

As a foreign exchange transaction we expect an exchange of two currencies, so we will also require a set of input states from the other counterparty. However, the Corda privacy model means we do not know the other node's states. Our flow must therefore negotiate with the other node for them to carry out a similar query and populate the inputs (See the `ForeignExchangeFlow` for more details of the exchange). Having identified a set of Input `StateRef` items we can then create the output as discussed below.

For the trade approval flow we need to implement a simple workflow pattern. We start by recording the unconfirmed trade details in a state object implementing the `LinearState` interface. One field of this record is used to map the business workflow to an enumerated state. Initially the initiator creates a new state object which receives a new `UniqueIdentifier` in its `linearId` property and a starting workflow state of `NEW`. The `Contract.verify` method is written to allow the initiator to sign this initial transaction and send it to the other party. This pattern ensures that a permanent copy is recorded on both ledgers for audit purposes, but the state is prevented from being maliciously put in an approved state. The subsequent workflow steps then follow with transactions that consume the state as inputs on one side and output a new version with whatever state updates, or amendments match to the business process, the `linearId` being preserved across the changes. Attached `Command` objects help the verify method restrict changes to appropriate fields and signers at each step in the workflow. In this it is typical to have both parties sign the change transactions, but it can be valid to allow unilateral signing, if for instance one side could block a rejection. Commonly the manual initiator of these workflows will query the Vault for states of the right contract type and in the right workflow

state over the RPC interface. The RPC will then initiate the relevant flow using `StateRef`, or `linearId` values as parameters to the flow to identify the states being operated upon. Thus code to gather the latest input state would be:

```
// Helper method to locate the latest Vault version of a LinearState from a possibly out
of date StateRef
inline fun <reified T : LinearState> ServiceHub.latest(ref: StateRef): StateAndRef<T> {
    val linearHeads = vaultService.linearHeadsOfType<T>()
    val original = toStateAndRef<T>(ref)
    return linearHeads[original.state.data.linearId]!!
}
```

```
// Pull in the latest Vault version of the StateRef as a full StateAndRef
val latestRecord = serviceHub.latest<TradeApprovalContract.State>(ref)
```

Generating Commands

For the commands that will be added to the transaction, these will need to correctly reflect the task at hand. These must match because inside the `Contract.verify` method the command will be used to select the validation code path. The `Contract.verify` method will then restrict the allowed contents of the transaction to reflect this context. Typical restrictions might include that the input cash amount must equal the output cash amount, or that a workflow step is only allowed to change the status field. Sometimes, the command may capture some data too e.g. the foreign exchange rate, or the identity of one party, or the StateRef of the specific input that originates the command in a bulk operation. This data will be used to further aid the `Contract.verify`, because to ensure consistent, secure and reproducible behaviour in a distributed environment the `Contract.verify`, transaction is the only allowed to use the content of the transaction to decide validity.

Another essential requirement for commands is that the correct set of `CompositeKeys` are added to the Command on the builder, which will be used to form the set of required signers on the final validated transaction. These must correctly align with the expectations of the `Contract.verify` method, which should be written to defensively check this. In particular, it is expected that at minimum the owner of an asset would have to be signing to permission transfer of that asset. In addition, other signatories will often be required e.g. an Oracle identity for an Oracle command, or both parties when there is an exchange of assets.

Generating Outputs

Having located a set of `StateAndRefs` as the transaction inputs, the flow has to generate the output states. Typically, this is a simple call to the Kotlin `copy` method to modify the few fields that will transitioned in the transaction. The contract code may provide a `generateXXX` method to help with this process if the task is more complicated. With a workflow state a slightly modified copy state is usually sufficient, especially as it is expected that we wish to preserve the `linearId` between state revisions, so that Vault queries can find the latest revision.

For fungible contract states such as `Cash` it is common to distribute and split the total amount e.g. to produce a remaining balance output state for the original owner when breaking up a large amount input state. Remember that the result of a successful transaction is always to fully consume/spend the input states, so this is required to conserve the total cash. For example from the demo code:

```
// Gather our inputs. We would normally use VaultService.generateSpend
// to carry out the build in a single step. To be more explicit
// we will use query manually in the helper function below.
// Putting this into a non-suspendable function also prevents issues when
// the flow is suspended.
val (inputs, residual) = gatherOurInputs(serviceHub, sellAmount, request.notary)

// Build and an output state for the counterparty
val transferedFundsOutput = Cash.State(sellAmount, request.counterparty)

if (residual > 0L) {
    // Build an output state for the residual change back to us
    val residualAmount = Amount(residual, sellAmount.token)
    val residualOutput = Cash.State(residualAmount, serviceHub.myInfo.legalIdentity)
    return FxResponse(inputs, listOf(transferedFundsOutput, residualOutput))
} else {
    return FxResponse(inputs, listOf(transferedFundsOutput))
}
```

Building the WireTransaction

Having gathered all the ingredients for the transaction we now need to use a `TransactionBuilder` to construct the full `WireTransaction`. The initial `TransactionBuilder` should be created by calling the `TransactionType.General.Builder` method. (The other `TransactionBuilder` implementation is only used for the `NotaryChange` flow where `ContractStates` need moving to a different Notary.) At this point the Notary to associate with the states should be recorded. Then we keep adding inputs, outputs, commands and attachments to fill the transaction. Examples of this process are:

```
        // Modify the state field for new output. We use copy, to ensure no other
modifications.
        // It is especially important for a LinearState that the linearId is copied
across,
        // not accidentally assigned a new random id.
        val newState = latestRecord.state.data.copy(state = verdict)

        // We have to use the original notary for the new transaction
        val notary = latestRecord.state.notary

        // Get and populate the new TransactionBuilder
        // To destroy the old proposal state and replace with the new completion state.
        // Also add the Completed command with keys of all parties to signal the Tx
purpose
        // to the Contract verify method.
        val tx = TransactionType.
                General.
                Builder(notary).
                withItems(
                        latestRecord,
                        newState,
                        Command(TradeApprovalContract.Commands.Completed(),
                                listOf(serviceHub.myInfo.legalIdentity.owningKey,
latestRecord.state.data.source.owningKey)))
        tx.addTimeWindow(serviceHub.clock.instant(), Duration.ofSeconds(60))
        // We can sign this transaction immediately as we have already checked all the
fields and the decision
        // is ultimately a manual one from the caller.
        // As a SignedTransaction we can pass the data around certain that it cannot be
modified,
        // although we do require further signatures to complete the process.
        val selfSignedTx = serviceHub.signInitialTransaction(tx)
```

```
    private fun buildTradeProposal(ourStates: FxResponse, theirStates: FxResponse):
SignedTransaction {
        // This is the correct way to create a TransactionBuilder,
        // do not construct directly.
        // We also set the notary to match the input notary
        val builder =
TransactionType.General.Builder(ourStates.inputs.first().state.notary)

        // Add the move commands and key to indicate all the respective owners and need
to sign
        val ourSigners = ourStates.inputs.map { it.state.data.owner.owningKey }.toSet()
        val theirSigners = theirStates.inputs.map { it.state.data.owner.owningKey
}.toSet()
        builder.addCommand(Cash.Commands.Move(), (ourSigners + theirSigners).toList())

        // Build and add the inputs and outputs
        builder.withItems(*ourStates.inputs.toTypedArray())
        builder.withItems(*theirStates.inputs.toTypedArray())
        builder.withItems(*ourStates.outputs.toTypedArray())
        builder.withItems(*theirStates.outputs.toTypedArray())

        // We have already validated their response and trust our own data
        // so we can sign. Note the returned SignedTransaction is still not fully signed
        // and would not pass full verification yet.
        return serviceHub.signInitialTransaction(builder)
    }
```

Completing the SignedTransaction

Having created an initial `WireTransaction` and converted this to an initial `SignedTransaction` the process of verifying and forming a full `SignedTransaction` begins and then completes with the notarisation. In practice this is a relatively stereotypical process, because assuming the `WireTransaction` is correctly constructed the verification should be immediate. However, it is also important to recheck the business details of any data received back from an external node, because a malicious party could always modify the contents before returning the transaction. Each remote flow should therefore check as much as possible of the initial `SignedTransaction` inside the `unwrap` of the receive before agreeing to sign. Any issues should immediately throw an exception to abort the flow. Similarly the originator, should always apply any new signatures to its original proposal to ensure the contents of the transaction has not been altered by the remote parties.

The typical code therefore checks the received `SignedTransaction` using the `verifySignatures` method, but excluding itself, the notary and any other parties yet to apply their signature. The contents of the `WireTransaction` inside the `SignedTransaction` should be fully verified further by expanding with `toLedgerTransaction` and calling `verify`. Further context specific and business checks should then be made, because the `Contract.verify` is not allowed to access external context. For example the flow may need to check that the parties are the right ones, or that the `Command` present on the transaction is as expected for this specific flow. An example of this from the demo code is:

```
        // First we receive the verdict transaction signed by their single key
        val completeTx = receive<SignedTransaction>(source).unwrap {
            // Check the transaction is signed apart from our own key and the notary
            val wtx = it.verifySignatures(serviceHub.myInfo.legalIdentity.owningKey,
it.tx.notary!!.owningKey)
            // Check the transaction data is correctly formed
            wtx.toLedgerTransaction(serviceHub).verify()
            // Confirm that this is the expected type of transaction
            require(wtx.commands.single().value is
TradeApprovalContract.Commands.Completed) {
                "Transaction must represent a workflow completion"
            }
            // Check the context dependent parts of the transaction as the
            // Contract verify method must not use serviceHub queries.
            val state = wtx.outRef<TradeApprovalContract.State>(0)
            require(state.state.data.source == serviceHub.myInfo.legalIdentity) {
                "Proposal not one of our original proposals"
            }
            require(state.state.data.counterparty == source) {
                "Proposal not for sent from correct source"
            }
            it
        }
```

After verification the remote flow will return its signature to the originator. The originator should apply that signature to the starting `SignedTransaction` and recheck the signatures match.

Committing the Transaction

Once all the party signatures are applied to the SignedTransaction the final step is notarisation. This involves calling `NotaryFlow.Client` to confirm the transaction, consume the inputs and return its confirming signature. Then the flow should ensure that all nodes end with all

signatures and that they call `ServiceHub.recordTransactions`. The code for this is standardised in the `FinalityFlow`, or more explicitly an example is:

```
        // Notarise and distribute the completed transaction.
        subFlow(FinalityFlow(allPartySignedTx, setOf(latestRecord.state.data.source,
latestRecord.state.data.counterparty)))
```

Partially Visible Transactions

The discussion so far has assumed that the parties need full visibility of the transaction to sign. However, there may be situations where each party needs to store private data for audit purposes, or for evidence to a regulator, but does not wish to share that with the other trading partner. The tear-off/Merkle tree support in Corda allows flows to send portions of the full transaction to restrict visibility to remote parties. To do this one can use the `WireTransaction.buildFilteredTransaction` extension method to produce a `FilteredTransaction`. The elements of the `SignedTransaction` which we wish to be hide will be replaced with their secure hash. The overall transaction txid is still provable from the `FilteredTransaction` preventing change of the private data, but we do not expose that data to the other node directly. A full example of this can be found in the `NodeInterestRates` Oracle code from the `irs-demo` project which interacts with the `RatesFixFlow` flow. Also, refer to the merkle-trees documentation.