Writing flows

This article explains our approach to modelling business processes and the lower level network protocols that implement them. It explains how the platform's flow framework is used, and takes you through the code for a simple 2-party asset trading flow which is included in the source.

Introduction

Shared distributed ledgers are interesting because they allow many different, mutually distrusting parties to share a single source of truth about the ownership of assets. Digitally signed transactions are used to update that shared ledger, and transactions may alter many states simultaneously and atomically.

Blockchain systems such as Bitcoin support the idea of building up a finished, signed transaction by passing around partially signed invalid transactions outside of the main network, and by doing this you can implement *delivery versus payment* such that there is no chance of settlement failure, because the movement of cash and the traded asset are performed atomically by the same transaction. To perform such a trade involves a multi-step flow in which messages are passed back and forth privately between parties, checked, signed and so on.

Despite how useful these flows are, platforms such as Bitcoin and Ethereum do not assist the developer with the rather tricky task of actually building them. That is unfortunate. There are many awkward problems in their implementation that a good platform would take care of for you, problems like:

- Avoiding "callback hell" in which code that should ideally be sequential is turned into an unreadable mess due to the desire to avoid using up a thread for every flow instantiation.
- Surviving node shutdowns/restarts that may occur in the middle of the flow without complicating things. This implies that the state of the flow must be persisted to disk.
- Error handling.
- Message routing.
- Serialisation.
- Catching type errors, in which the developer gets temporarily confused and expects to receive/send one type of message when actually they need to receive/send another.
- Unit testing of the finished flow.

Actor frameworks can solve some of the above but they are often tightly bound to a particular messaging layer, and we would like to keep a clean separation. Additionally, they are typically not type safe, and don't make persistence or writing sequential code much easier.

To put these problems in perspective, the *payment channel protocol* in the bitcoinj library, which allows bitcoins to be temporarily moved off-chain and traded at high speed between two parties in private, consists of about 7000 lines of Java and took over a month of full time work to develop. Most of that code is concerned with the details of persistence, message passing, lifecycle management, error handling and callback management. Because the business logic is quite spread out the code can be difficult to read and debug.

As small contract-specific trading flows are a common occurrence in finance, we provide a framework for the construction of them that automatically handles many of the concerns outlined above.

Theory

A *continuation* is a suspended stack frame stored in a regular object that can be passed around, serialised, unserialised and resumed from where it was suspended. This concept is sometimes referred to as “fibers”. This may sound abstract but don’t worry, the examples below will make it clearer. The JVM does not natively support continuations, so we implement them using a library called Quasar which works through behind-the-scenes bytecode rewriting. You don’t have to know how this works to benefit from it, however.

We use continuations for the following reasons:

- It allows us to write code that is free of callbacks, that looks like ordinary sequential code.
- A suspended continuation takes far less memory than a suspended thread. It can be as low as a few hundred bytes. In contrast a suspended Java thread stack can easily be 1mb in size.
- It frees the developer from thinking (much) about persistence and serialisation.

A *state machine* is a piece of code that moves through various *states*. These are not the same as states in the data model (that represent facts about the world on the ledger), but rather indicate different stages in the progression of a multi-stage flow. Typically writing a state machine would require the use of a big switch statement and some explicit variables to keep track of where you’re up to. The use of continuations avoids this hassle.

A two party trading flow

We would like to implement the “hello world” of shared transaction building flows: a seller wishes to sell some *asset* (e.g. some commercial paper) in return for *cash*. The buyer wishes to purchase the asset using his cash. They want the trade to be atomic so neither side is exposed to the risk of settlement failure. We assume that the buyer and seller have found each other and arranged the details on some exchange, or over the counter. The details of how the trade is arranged isn’t covered in this article.

Our flow has two parties (B and S for buyer and seller) and will proceed as follows:

1. S sends a `StateAndRef` pointing to the state they want to sell to B, along with info about the price they require B to pay.
2. B sends to S a `SignedTransaction` that includes the state as input, B’s cash as input, the state with the new owner key as output, and any change cash as output. It contains a single signature from B but isn’t valid because it lacks a signature from S authorising movement of the asset.
3. S signs the transaction and sends it back to B.
4. B *finalises* the transaction by sending it to the notary who checks the transaction for validity, recording the transaction in B’s local vault, and then sending it on to S who also checks it and commits the transaction to S’s local vault.

You can find the implementation of this flow in the file `finance/src/main/kotlin/net/corda/flows/TwoPartyTradeFlow.kt`.

Assuming no malicious termination, they both end the flow being in possession of a valid, signed transaction that represents an atomic asset swap.

Note that it's the *seller* who initiates contact with the buyer, not vice-versa as you might imagine.

We start by defining two classes that will contain the flow definition. We also pick what data will be used by each side.

Note

The code samples in this tutorial are only available in Kotlin, but you can use any JVM language to write them and the approach is the same.

Kotlin

```kotlin
object TwoPartyTradeFlow {
    class UnacceptablePriceException(val givenPrice: Amount<Currency>) :
FlowException("Unacceptable price: $givenPrice")
    class AssetMismatchException(val expectedTypeName: String, val typeName: String) :
FlowException() {
        override fun toString() = "The submitted asset didn't match the expected type:
$expectedTypeName vs $typeName"
    }

    // This object is serialised to the network and is the first flow message the seller
sends to the buyer.
    @CordaSerializable
    data class SellerTradeInfo(
            val assetForSale: StateAndRef<OwnableState>,
            val price: Amount<Currency>,
            val sellerOwnerKey: PublicKey
    )

    open class Seller(val otherParty: Party,
                      val notaryNode: NodeInfo,
                      val assetToSell: StateAndRef<OwnableState>,
                      val price: Amount<Currency>,
                      val myKey: PublicKey,
                      override val progressTracker: ProgressTracker = Seller.tracker()) :
FlowLogic<SignedTransaction>() {
        @Suspendable
        override fun call(): SignedTransaction {
            TODO()
        }
    }

    open class Buyer(val otherParty: Party,
                     val notary: Party,
                     val acceptablePrice: Amount<Currency>,
                     val typeToBuy: Class<out OwnableState>) :
FlowLogic<SignedTransaction>() {
        @Suspendable
        override fun call(): SignedTransaction {
            TODO()
        }
    }
}
```

This code defines several classes nested inside the main `TwoPartyTradeFlow` singleton. Some of the classes are simply flow messages or exceptions. The other two represent the buyer and seller side of the flow.

Going through the data needed to become a seller, we have:

- `otherParty: Party` - the party with which you are trading.
- `notaryNode: NodeInfo` - the entry in the network map for the chosen notary. See "Notaries" for more information on notaries.
- `assetToSell: StateAndRef<OwnableState>` - a pointer to the ledger entry that represents the thing being sold.
- `price: Amount<Currency>` - the agreed on price that the asset is being sold for (without an issuer constraint).
- `myKey: PublicKey` - the PublicKey part of the node's internal KeyPair that controls the asset being sold.

The matching PrivateKey stored in the KeyManagementService will be used to sign the transaction.

And for the buyer:

- `acceptablePrice: Amount<Currency>` - the price that was agreed upon out of band. If the seller specifies a price less than or equal to this, then the trade will go ahead.
- `typeToBuy: Class<out OwnableState>` - the type of state that is being purchased. This is used to check that the sell side of the flow isn't trying to sell us the wrong thing, whether by accident or on purpose.

Alright, so using this flow shouldn't be too hard: in the simplest case we can just create a Buyer or Seller with the details of the trade, depending on who we are. We then have to start the flow in some way. Just calling the `call` function ourselves won't work: instead we need to ask the framework to start the flow for us. More on that in a moment.

Suspendable functions

The `call` function of the buyer/seller classes is marked with the `@Suspendable` annotation. What does this mean?

As mentioned above, our flow framework will at points suspend the code and serialise it to disk. For this to work, any methods on the call stack must have been pre-marked as `@Suspendable` so the bytecode rewriter knows to modify the underlying code to support this new feature. A flow is suspended when calling either `receive`, `send` or `sendAndReceive` which we will learn more about below. For now, just be aware that when one of these methods is invoked, all methods on the stack must have been marked. If you forget, then in the unit test environment you will get a useful error message telling you which methods you didn't mark. The fix is simple enough: just add the annotation and try again.

Note

Java 9 is likely to remove this pre-marking requirement completely.

Whitelisted classes with the Corda node

For security reasons, we do not want Corda nodes to be able to receive instances of any class on the classpath via messaging, since this has been exploited in other Java application containers in the past. Instead, we require that every class contained in messages is whitelisted. Some classes are whitelisted by default (see `DefaultWhitelist`), but others outside of that set need to be whitelisted either by using the annotation `@CordaSerializable` or via the plugin framework. See Object serialization. You can see above that the `SellerTradeInfo` has been annotated.

Starting your flow

The `StateMachineManager` is the class responsible for taking care of all running flows in a node. It knows how to register handlers with the messaging system (see "Networking and messaging") and iterate the right state machine when messages arrive. It provides the send/receive/sendAndReceive calls that let the code request network interaction and it will save/restore serialised versions of the fiber at the right times.

Flows can be invoked in several ways. For instance, they can be triggered by scheduled events (in which case they need to be annotated with `@SchedulableFlow`), see "Event scheduling" to learn more about this. They can also be triggered directly via the node's RPC API from your app code (in which case they need to be annotated with *StartableByRPC*). It's possible for a flow to be of both types.

You request a flow to be invoked by using the `CordaRPCOps.startFlowDynamic` method. This takes a Java reflection `Class` object that describes the flow class to use (in this case, either `Buyer` or `Seller`). It also takes a set of arguments to pass to the constructor. Because it's possible for flow invocations to be requested by untrusted code (e.g. a state that you have been sent), the types that can be passed into the flow are checked against a whitelist, which can be extended by apps themselves at load time. There are also a series of inlined Kotlin extension functions of the form `CordaRPCOps.startFlow` which help with invoking flows in a type safe manner.

The process of starting a flow returns a `FlowHandle` that you can use to observe the result, and which also contains a permanent identifier for the invoked flow in the form of the `StateMachineRunId`. Should you also wish to track the progress of your flow (see Progress tracking) then you can invoke your flow instead using `CordaRPCOps.startTrackedFlowDynamic` or any of its corresponding `CordaRPCOps.startTrackedFlow` extension functions. These will return a `FlowProgressHandle`, which is just like a `FlowHandle` except that it also contains an observable `progress` field.

Note

The developer *must* then either subscribe to this `progress` observable or invoke the `notUsed()` extension function for it. Otherwise the unused observable will waste resources back in the node.

Implementing the seller

Let's implement the `Seller.call` method. This will be run when the flow is invoked.

Kotlin

```
    @Suspendable
    override fun call(): SignedTransaction {
        progressTracker.currentStep = AWAITING_PROPOSAL
        // Make the first message we'll send to kick off the flow.
        val hello = SellerTradeInfo(assetToSell, price, myKey)
        // What we get back from the other side is a transaction that *might* be valid
and acceptable to us,
        // but we must check it out thoroughly before we sign!
        send(otherParty, hello)

        // Verify and sign the transaction.
        progressTracker.currentStep = VERIFYING_AND_SIGNING
        // DOCSTART 5
        val signTransactionFlow = object : SignTransactionFlow(otherParty,
VERIFYING_AND_SIGNING.childProgressTracker()) {
            override fun checkTransaction(stx: SignedTransaction) {
                if (stx.tx.outputs.map { it.data
}.sumCashBy(AnonymousParty(myKey)).withoutIssuer() != price)
                    throw FlowException("Transaction is not sending us the right amount
of cash")
            }
        }
        return subFlow(signTransactionFlow)
        // DOCEND 5
    }
```

We start by sending information about the asset we wish to sell to the buyer. We fill out the initial flow message with the trade info, and then call `send`. which takes two arguments:

- The party we wish to send the message to.
- The payload being sent.

`send` will serialise the payload and send it to the other party automatically.

Next, we call a *subflow* called `SignTransactionFlow` (see Sub-flows). `SignTransactionFlow` automates the process of:

- Receiving a proposed trade transaction from the buyer, with the buyer's signature attached.
- Checking that the proposed transaction is valid.
- Calculating and attaching our own signature so that the transaction is now signed by both the buyer and the seller.
- Sending the transaction back to the buyer.

The transaction then needs to be finalized. This is the the process of sending the transaction to a notary to assert (with another signature) that the timestamp in the transaction (if any) is valid and there are no double spends. In this flow, finalization is handled by the buyer, so we just wait for the signed transaction to appear in our transaction storage. It will have the same ID as the one we started with but more signatures.

Implementing the buyer

OK, let's do the same for the buyer side:

```
    @Suspendable
    override fun call(): SignedTransaction {
        // Wait for a trade request to come in from the other party.
        progressTracker.currentStep = RECEIVING
        val tradeRequest = receiveAndValidateTradeRequest()

        // Put together a proposed transaction that performs the trade, and sign it.
        progressTracker.currentStep = SIGNING
        val (ptx, cashSigningPubKeys) = assembleSharedTX(tradeRequest)
        val partSignedTx = signWithOurKeys(cashSigningPubKeys, ptx)

        // Send the signed transaction to the seller, who must then sign it themselves
and commit
        // it to the ledger by sending it to the notary.
        progressTracker.currentStep = COLLECTING_SIGNATURES
        val twiceSignedTx = subFlow(CollectSignaturesFlow(partSignedTx,
COLLECTING_SIGNATURES.childProgressTracker()))

        // Notarise and record the transaction.
        progressTracker.currentStep = RECORDING
        return subFlow(FinalityFlow(twiceSignedTx, setOf(otherParty,
serviceHub.myInfo.legalIdentity))).single()
    }

    @Suspendable
    private fun receiveAndValidateTradeRequest(): SellerTradeInfo {
        val maybeTradeRequest = receive<SellerTradeInfo>(otherParty)

        progressTracker.currentStep = VERIFYING
        maybeTradeRequest.unwrap {
            // What is the seller trying to sell us?
            val asset = it.assetForSale.state.data
            val assetTypeName = asset.javaClass.name

            if (it.price > acceptablePrice)
                throw UnacceptablePriceException(it.price)
            if (!typeToBuy.isInstance(asset))
                throw AssetMismatchException(typeToBuy.name, assetTypeName)

            // Check that the state being sold to us is in a valid chain of transactions,
i.e. that the
            // seller has a valid chain of custody proving that they own the thing
they're selling.
            subFlow(ResolveTransactionsFlow(setOf(it.assetForSale.ref.txhash),
otherParty))

            return it
        }
    }

    private fun signWithOurKeys(cashSigningPubKeys: List<PublicKey>, ptx:
TransactionBuilder): SignedTransaction {
        // Now sign the transaction with whatever keys we need to move the cash.
        return serviceHub.signInitialTransaction(ptx, cashSigningPubKeys)
    }

    @Suspendable
    private fun assembleSharedTX(tradeRequest: SellerTradeInfo): Pair<TransactionBuilder,
List<PublicKey>> {
        val ptx = TransactionType.General.Builder(notary)

        // Add input and output states for the movement of cash, by using the Cash
contract to generate the states
        val (tx, cashSigningPubKeys) = serviceHub.vaultService.generateSpend(ptx,
tradeRequest.price, AnonymousParty(tradeRequest.sellerOwnerKey))

        // Add inputs/outputs/a command for the movement of the asset.
        tx.addInputState(tradeRequest.assetForSale)
        // Just pick some new public key for now. This won't be linked with our identity
in any way, which is what
        // we want for privacy reasons: the key is here ONLY to manage and control
ownership, it is not intended to
        // reveal who the owner actually is. The key management service is expected to
derive a unique key from some
        // initial seed in order to provide privacy protection.
        val freshKey = serviceHub.keyManagementService.freshKey()
        val (command, state) =
tradeRequest.assetForSale.state.data.withNewOwner(AnonymousParty(freshKey))
        tx.addOutputState(state, tradeRequest.assetForSale.state.notary)
        tx.addCommand(command, tradeRequest.assetForSale.state.data.owner.owningKey)

        // And add a request for a time-window: it may be that none of the contracts need
this!
        // But it can't hurt to have one.
        val currentTime = serviceHub.clock.instant()
        tx.addTimeWindow(currentTime, 30.seconds)
        return Pair(tx, cashSigningPubKeys)
    }
```

This code is longer but no more complicated. Here are some things to pay attention to:

1. We do some sanity checking on the proposed trade transaction received from the seller to ensure we're being offered what we expected to be offered.
2. We create a cash spend using `VaultService.generateSpend`. You can read the vault documentation to learn more about this.
3. We access the *service hub* as needed to access things that are transient and may change or be recreated whilst a flow is suspended, such as the wallet or the network map.
4. We call `CollectSignaturesFlow` as a subflow to send the unfinished, still-invalid transaction to the seller so they can sign it and send it back to us.
5. Last, we call `FinalityFlow` as a subflow to finalize the transaction.

As you can see, the flow logic is straightforward and does not contain any callbacks or network glue code, despite the fact that it takes minimal resources and can survive node restarts.

Flow sessions

It will be useful to describe how flows communicate with each other. A node may have many flows running at the same time, and perhaps communicating with the same counterparty node but for different purposes. Therefore flows need a way to segregate communication channels so that concurrent conversations between flows on the same set of nodes do not interfere with each other.

To achieve this the flow framework initiates a new flow session each time a flow starts communicating with a `Party` for the first time. A session is simply a pair of IDs, one for each side, to allow the node to route received messages to the correct flow. If the other side accepts the session request then subsequent sends and receives to that same `Party` will use the same session. A session ends when either flow ends, whether as expected or pre-maturely. If a flow ends pre-maturely then the other side will be notified of that and they will also end, as the whole point of flows is a known sequence of message transfers. Flows end pre-maturely due to exceptions, and as described above, if that exception is `FlowException` or a sub-type then it will propagate to the other side. Any other exception will not propagate.

Taking a step back, we mentioned that the other side has to accept the session request for there to be a communication channel. A node accepts a session request if it has registered the flow type (the fully-qualified class name) that is making the request - each session initiation includes the initiating flow type. The registration is done by a CorDapp which has made available the particular flow communication, using `PluginServiceHub.registerServiceFlow`. This method specifies a flow factory for generating the counter-flow to any given initiating flow. If this registration doesn't exist then no further communication takes place and the initiating flow ends with an exception.

Going back to our buyer and seller flows, we need a way to initiate communication between the two. This is typically done with one side started manually using the `startFlowDynamic` RPC and this initiates the counter-flow on the other side. In this case it doesn't matter which flow is the initiator and which is the initiated. If we choose the seller side as the initiator then the buyer side would need to register their flow, perhaps with something like:

Kotlin

```kotlin
class TwoPartyTradeFlowPlugin : CordaPluginRegistry() {
    override val servicePlugins = listOf(Function(TwoPartyTradeFlowService::Service))
}

object TwoPartyTradeFlowService {
    class Service(services: PluginServiceHub) {
        init {
            services.registerServiceFlow(TwoPartyTradeFlow.Seller::class.java) {
                TwoPartyTradeFlow.Buyer(
                    it,
                    notary = services.networkMapCache.notaryNodes[0].notaryIdentity,
                    acceptablePrice = TODO(),
                    typeToBuy = TODO())
            }
        }
    }
}
```

This is telling the buyer node to fire up an instance of `TwoPartyTradeFlow.Buyer` (the code in the lambda) when they receive a message from the initiating seller side of the flow (`TwoPartyTradeFlow.Seller::class.java`).

Sub-flows

Flows can be composed via nesting. Invoking a sub-flow looks similar to an ordinary function call:

Kotlin Java

```kotlin
@Suspendable
fun call() {
    val unnotarisedTransaction = ...
    subFlow(FinalityFlow(unnotarisedTransaction))
}
```

Let's take a look at the three subflows we invoke in this flow.

FinalityFlow

On the buyer side, we use `FinalityFlow` to finalise the transaction. It will:

- Send the transaction to the chosen notary and, if necessary, satisfy the notary that the transaction is valid.
- Record the transaction in the local vault, if it is relevant (i.e. involves the owner of the node).
- Send the fully signed transaction to the other participants for recording as well.

Warning

If the seller stops before sending the finalised transaction to the buyer, the seller is left with a valid transaction but the buyer isn't, so they can't spend the asset they just purchased! This sort of thing is not always a risk (as the seller may not gain anything from that sort of behaviour except a lawsuit), but if it is, a future version of the platform will allow you to ask the notary to send you the transaction as well, in case your counterparty does not. This is not the default because it reveals more private info to the notary.

We simply create the flow object via its constructor, and then pass it to the `subFlow` method which returns the result of the flow's execution directly. Behind the scenes all this is doing is wiring up progress tracking (discussed more below) and then running the object's `call` method. Because the sub-flow might suspend, we must mark the method that invokes it as suspendable.

Within FinalityFlow, we use a further sub-flow called `ResolveTransactionsFlow`. This is responsible for downloading and checking all the dependencies of a transaction, which in Corda are always retrievable from the party that sent you a transaction that uses them. This flow returns a list of `LedgerTransaction` objects.

Note

Transaction dependency resolution assumes that the peer you got the transaction from has all of the dependencies itself. It must do, otherwise it could not have convinced itself that the dependencies were themselves valid. It's important to realise that requesting only the transactions we require is a privacy leak, because if we don't download a transaction from the peer, they know we must have already seen it before. Fixing this privacy leak will come later.

CollectSignaturesFlow/SignTransactionFlow

We also invoke two other subflows:

- `CollectSignaturesFlow`, on the buyer side
- `SignTransactionFlow`, on the seller side

These flows communicate to gather all the required signatures for the proposed transaction. `CollectSignaturesFlow` will:

- Verify any signatures collected on the transaction so far
- Verify the transaction itself
- Send the transaction to the remaining required signers and receive back their signatures

- Verify the collected signatures

`SignTransactionFlow` responds by:

- Receiving the partially-signed transaction off the wire
- Verifying the existing signatures
- Resolving the transaction's dependencies
- Verifying the transaction itself
- Running any custom validation logic
- Sending their signature back to the buyer
- Waiting for the transaction to be recorded in their vault

We cannot instantiate `SignTransactionFlow` itself, as it's an abstract class. Instead, we need to subclass it and override `checkTransaction()` to add our own custom validation logic:

Kotlin

```
val signTransactionFlow = object : SignTransactionFlow(otherParty,
VERIFYING_AND_SIGNING.childProgressTracker()) {
    override fun checkTransaction(stx: SignedTransaction) {
        if (stx.tx.outputs.map { it.data
}.sumCashBy(AnonymousParty(myKey)).withoutIssuer() != price)
            throw FlowException("Transaction is not sending us the right amount of cash")
    }
}
return subFlow(signTransactionFlow)
```

In this case, our custom validation logic ensures that the amount of cash outputs in the transaction equals the price of the asset.

Persisting flows

If you look at the code for `FinalityFlow`, `CollectSignaturesFlow` and `SignTransactionFlow`, you'll see calls to both `receive` and `sendAndReceive`. Once either of these methods is called, the `call` method will be suspended into a continuation and saved to persistent storage. If the node crashes or is restarted, the flow will effectively continue as if nothing had happened. Your code may remain blocked inside such a call for seconds, minutes, hours or even days in the case of a flow that needs human interaction!

Note

There are a couple of rules you need to bear in mind when writing a class that will be used as a continuation. The first is that anything on the stack when the function is suspended will be stored into the heap and kept alive by the garbage collector. So try to avoid keeping enormous data structures alive unless you really have to. You can always use private methods to keep the stack uncluttered with temporary variables, or to avoid objects that Kryo is not able to serialise correctly.

The second is that as well as being kept on the heap, objects reachable from the stack will be serialised. The state of the function call may be resurrected much later! Kryo doesn't require objects be marked as serialisable, but even so, doing things like creating threads from inside these calls would be a bad idea. They should only contain business logic and only do I/O via the methods exposed by the flow framework.

It's OK to keep references around to many large internal node services though: these will be serialised using a special token that's recognised by the platform, and wired up to the right instance when the continuation is loaded off disk again.

`receive` and `sendAndReceive` return a simple wrapper class, `UntrustworthyData<T>`, which is just a marker class that reminds us that the data came from a potentially malicious external source and may have been tampered with or be unexpected in other ways. It doesn't add any functionality, but acts as a reminder to "scrub" the data before use.

Exception handling

Flows can throw exceptions to prematurely terminate their execution. The flow framework gives special treatment to `FlowException` and its subtypes. These exceptions are treated as error responses of the flow and are propagated to all counterparties it is communicating with. The receiving flows will throw the same exception the next time they do a `receive` or `sendAndReceive` and thus end the flow session. If the receiver was invoked via `subFlow` (details below) then the exception can be caught there enabling re-invocation of the sub-flow.

If the exception thrown by the erroring flow is not a `FlowException` it will still terminate but will not propagate to the other counterparties. Instead they will be informed the flow has terminated and will themselves be terminated with a generic exception.

Note

A future version will extend this to give the node administrator more control on what to do with such erroring

flows.

Throwing a `FlowException` enables a flow to reject a piece of data it has received back to the sender. This is typically done in the `unwrap` method of the received `UntrustworthyData`. In the above example the seller checks the price and throws `FlowException` if it's invalid. It's then up to the buyer to either try again with a better price or give up.

Progress tracking

Not shown in the code snippets above is the usage of the `ProgressTracker` API. Progress tracking exports information from a flow about where it's got up to in such a way that observers can render it in a useful manner to humans who may need to be informed. It may be rendered via an API, in a GUI, onto a terminal window, etc.

A `ProgressTracker` is constructed with a series of `Step` objects, where each step is an object representing a stage in a piece of work. It is therefore typical to use singletons that subclass `Step`, which may be defined easily in one line when using Kotlin. Typical steps might be "Waiting for response from peer", "Waiting for signature to be approved", "Downloading and verifying data" etc.

A flow might declare some steps with code inside the flow class like this:

Kotlin Java

```
    object RECEIVING : ProgressTracker.Step("Waiting for seller trading info")
    object VERIFYING : ProgressTracker.Step("Verifying seller assets")
    object SIGNING : ProgressTracker.Step("Generating and signing transaction proposal")
    object COLLECTING_SIGNATURES : ProgressTracker.Step("Collecting signatures from other
parties") {
        override fun childProgressTracker() = CollectSignaturesFlow.tracker()
    }
    object RECORDING : ProgressTracker.Step("Recording completed transaction") {
        // TODO: Currently triggers a race condition on Team City. See
https://github.com/corda/corda/issues/733.
        // override fun childProgressTracker() = FinalityFlow.tracker()
    }

    override val progressTracker = ProgressTracker(RECEIVING, VERIFYING, SIGNING,
COLLECTING_SIGNATURES, RECORDING)
```

Each step exposes a label. By default labels are fixed, but by subclassing `RelabelableStep` you can make a step that can update its label on the fly. That's useful for steps that want to expose non-structured progress information like the current file being downloaded. By defining your own step types, you can export progress in a way that's both human readable and machine readable.

Progress trackers are hierarchical. Each step can be the parent for another tracker. By altering the `ProgressTracker.childrenFor` map, a tree of steps can be created. It's allowed to alter the hierarchy at runtime, on the fly, and the progress renderers will adapt to that properly. This can be helpful when you don't fully know ahead of time what steps will be required. If you *do* know what is required, configuring as much of the hierarchy ahead of time is a good idea, as that will help the users see what is coming up. You can pre-configure steps by overriding the `Step` class like this:

Kotlin Java

```
        object VERIFYING_AND_SIGNING : ProgressTracker.Step("Verifying and signing
transaction proposal") {
            override fun childProgressTracker() = SignTransactionFlow.tracker()
        }
```

Every tracker has not only the steps given to it at construction time, but also the singleton `ProgressTracker.UNSTARTED` step and the `ProgressTracker.DONE` step. Once a tracker has become `DONE` its position may not be modified again (because e.g. the UI may have been removed/cleaned up), but until that point, the position can be set to any arbitrary set both forwards and backwards. Steps may be skipped, repeated, etc. Note that rolling the current step backwards will delete any progress trackers that are children of the steps being reversed, on the assumption that those subtasks will have to be repeated.

Trackers provide an Rx observable which streams changes to the hierarchy. The top level observable exposes all the events generated by its children as well. The changes are represented by objects indicating whether the change is one of position (i.e. progress), structure (i.e. new subtasks being added/removed) or some other aspect of rendering (i.e. a step has changed in some way and is requesting a re-render).

The flow framework is somewhat integrated with this API. Each `FlowLogic` may optionally provide a tracker by overriding the `flowTracker` property (`getFlowTracker` method in Java). If the `FlowLogic.subFlow` method is used, then the tracker of the sub-flow will be made a child of the current step in the parent flow automatically, if the parent is using tracking in the first place. The framework will also automatically set the current step to `DONE` for you, when the flow is finished.

Because a flow may sometimes wish to configure the children in its progress hierarchy *before* the sub-flow is constructed, for sub-flows that always follow the same outline regardless of their parameters it's conventional to define a companion object/static method (for Kotlin/Java respectively) that constructs a tracker, and then allow the sub-flow to have the tracker it will use be passed in as a parameter. This allows all trackers to be built and linked ahead of time.

In future, the progress tracking framework will become a vital part of how exceptions, errors, and other faults are surfaced to human operators for investigation and resolution.

Versioning

Fibers involve persisting object-serialised stack frames to disk. Although we may do some R&D into in-place upgrades in future, for now the upgrade process for flows is simple: you duplicate the code and rename it so it has a new set of class names. Old versions of the flow can then drain out of the system whilst new versions are initiated. When enough time has passed that no old versions are still waiting for anything to happen, the previous copy of the code can be deleted.

Whilst kind of ugly, this is a very simple approach that should suffice for now.

Warning

Flows are not meant to live for months or years, and by implication they are not meant to implement entire deal lifecycles. For instance, implementing the entire life cycle of an interest rate swap as a single flow - whilst technically possible - would not be a good idea. The platform provides a job scheduler tool that can invoke flows for this reason (see "Event scheduling")

Future features

The flow framework is a key part of the platform and will be extended in major ways in future. Here are some of the features we have planned:

- Exception management, with a "flow hospital" tool to manually provide solutions to unavoidable problems (e.g. the other side doesn't know the trade)
- Being able to interact with people, either via some sort of external ticketing system, or email, or a custom UI. For example to implement human transaction authorisations.
- A standard library of flows that can be easily sub-classed by local developers in order to integrate internal reporting logic, or anything else that might be required as part of a communications lifecycle.