Modelling Corda Flows and State for use in Workbench

Workbench presents a simple view of a DLT, which can be summarised as

* A Finite State machine view, where there are
  + explicit functions for the possible actions
  + actions may trigger state changes
  + a finite set of states, and the system can only ever be in one state at a time
  + a set of rules that govern which actions are available based on the current state
* A “flat” data model
  + A contract state is in effect a simple list of name value pairs

Corda has a more flexible and general-purpose model of States, Contract and Flows and there may need to a bridging later between a native CorDapp app and Workbench. The design approach is to implement this layer as a set of Corda flows and in most cases this layer should be quite simple to construct. We DON’Trecommendcompromising the design of the CorDapp – Corda provides a richer model as our experience has shown this is necessary for many real-world scenarios.

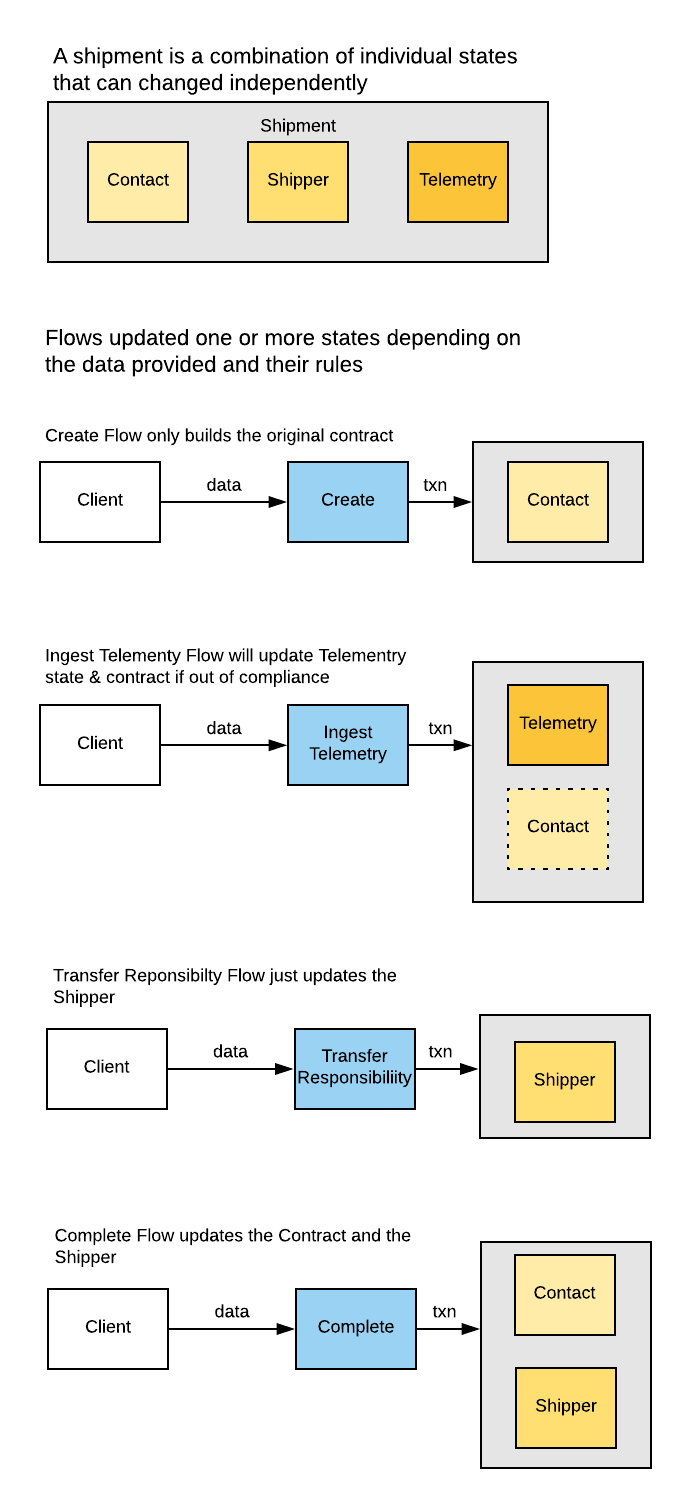
As an example, we will walk through the Refrigerated Transport application implement in a more Corda native style. In this case, the single state is split into three

1. The “Contract” with the owner, temperate / humidity ranges and status
2. The “Shipper”, which records transfer between shipper
3. The “Telemetry” state which just records sensor readings.

This is important when building a distributed app over a permissioned blockchain. Some reason are:

* without breaking out Telemetry as a discreet state, the provider of sensor data would need to see the whole contract just to sign their reading
* in Corda the smart contract need to verify the proposed state change is valid in the context of the proposes update. Larger states make this more difficult to reason about, for example in a single state an unscrupulous provider of sensor data could just update the contract parameters to keep the reading in range. Of course this fraudulent action is now evident in the ledger history, but that may not be of help.
* Other application may make use of this data. Obvious example would be to provide the history of sensor reading to other apps that deal with insurance claims, regulators and secondary markets for goods that still saleable at a lower quality. This is clearly harder if the sensor data is mixed in other private data.

The diagram below explains (original at <https://www.lucidchart.com/invitations/accept/94ca35cf-6dab-4986-a54b-fdd0fb8cbf45>)



So how does this map to the Workbench model?

The first two points about Finite State machines:

* explicit functions for the possible actions
* actions may trigger state changes

These should fall quite naturally out the design of the Corda flows. There can be one per action and they construct transaction as required. The only Workbench specific requirement is to return a TxnResult rather than the more usual SignedTransaction.

The third point about Finite State machines:

* a finite set of states, and the system can only ever be in one state at a time

This state is easily available from within Corda but there is no way for Workbench to use it without a convention. I suggest that a single Corda flow is added that returns the current state (against an enum). This is annotated with @WorkbenchContractState.

Workbench also needs to know which node to query for this state and to be the “authoritative” state (as there is no concept of global state in Corda). Assuming the this is node that initiated the transaction feels reasonable.

The fourth point about Finite State machines

* a set of rules that govern which actions are available based on the current state

This is a simple rule to implement, but again there needs to be a convention. I suggest a single Corda flow that returns a list of the available flows based on the current state. This is annotated with @WorkbenchContractActions

And finally, the point about the flat data model. This is an example of a classic Façade pattern (<https://en.wikipedia.org/wiki/Facade_pattern>) whereby some more complex internal state is simplified for external access, usually either for efficiency (keep the queries close to the data) or to simplify an external API . I propose a single additional flow that returns a “flat” model by querying for each of the states and combining in a single data structure. So, in this case we look for the active “Shipper”, “Contact” and “Telemetry”, and combine into a single structure. This is annotated with @WorkbenchState annotation.