Using a notary service

This tutorial describes how to assign a notary to a newly issued state, and how to get a transaction notarised by obtaining a signature of the required notary. It assumes some familiarity with *flows* and how to write them, as described in Writing flows.

Assigning a notary

The first step is to choose a notary and obtain its identity. Identities of all notaries on the network are kept by the Network Map Service. The network map cache exposes two methods for obtaining a notary:

```
/**
 * Gets a notary identity by the given name.
 */
fun getNotary(name: String): Party?

/**
 * Returns a notary identity advertised by any of the nodes on the network (chosen at
random)
 *
 * @param type Limits the result to notaries of the specified type (optional)
 */
fun getAnyNotary(type: ServiceType? = null): Party?
```

Currently notaries can only be differentiated by name and type, but in the future the network map service will be able to provide more metadata, such as location or legal identities of the nodes operating it.

Now, let's say we want to issue an asset and assign it to a notary named "Notary A". The first step is to obtain the notary identity – `Party`:

```
val ourNotary: Party = serviceHub.networkMapCache.getNotary("Central Bank Notary")
```

Then we initialise the transaction builder:

```
val builder: TransactionBuilder = TransactionType.General.Builder(notary = ourNotary)
```

For any output state we add to this transaction builder, `ourNotary` will be assigned as its notary. Next we create a state object and assign ourselves as the owner. For this example we'll use a `DummyContract.State`, which is a simple state that just maintains an integer and can change ownership.

```
val myIdentity = serviceHub.myInfo.legalIdentity
val state = DummyContract.SingleOwnerState(magicNumber = 42, owner =
myIdentity.owningKey)
```

Then we add the state as the transaction output along with the relevant command. The state will automatically be assigned to our previously specified "Notary A".

```
builder.addOutputState(state)
val createCommand = DummyContract.Commands.Create()
builder.addCommand(Command(createCommand, myIdentity))
```

We then sign the transaction, build and record it to our transaction storage:

```
val mySigningKey: PublicKey = serviceHub.legalIdentityKey
val issueTransaction = serviceHub.signInitialTransaction(issueTransaction, mySigningKey)
serviceHub.recordTransactions(issueTransaction)
```

The transaction is recorded and we now have a state (asset) in possession that we can transfer to someone else. Note that the issuing transaction does not need to be notarised, as it doesn't consume any input states.

Notarising a transaction

Following our example for the previous section, let's say we now want to transfer our issued state to Alice.

First we obtain a reference to the state, which will be the input to our "move" transaction:

```
val stateRef = StateRef(txhash = issueTransaction.id, index = 0)
```

Then we create a new state – a copy of our state but with the owner set to Alice. This is a bit more involved so we just use a helper that handles it for us. We also assume that we already have the `Party` for Alice, `aliceParty`.

```
val inputState = StateAndRef(sate, stateRef)
val moveTransactionBuilder = DummyContract.move(inputState, newOwner =
aliceParty.owningKey)
```

The `DummyContract.move()` method will a new transaction builder with our old state as the input, a new state with Alice as the owner, and a relevant contract command for "move".

Again we sign the transaction, and build it:

```
// We build it and add our default identity signature without checking if all signatures are present,
// Note we know that the notary signature is missing, so thie SignedTransaction is still partial.
val moveTransaction = serviceHub.signInitialTransaction(moveTransactionBuilder)
```

Next we need to obtain a signature from the notary for the transaction to be valid. Prior to signing, the notary will commit our old (input) state so it cannot be used again.

To manually obtain a signature from a notary we can run the `NotaryFlow.Client` flow. The flow will work out which notary needs to be called based on the input states (and the timestamp command, if it's present).

```
// The subFlow() helper is available within the context of a Flow
val notarySignature: DigitalSignature = subFlow(NotaryFlow.Client(moveTransaction))
```

Note

If our input state has already been consumed in another transaction, then `NotaryFlow` with throw a `NotaryException` containing the conflict details:

```
/** Specifies the consuming transaction for the conflicting input state */
data class Conflict(val stateHistory: Map<StateRef, ConsumingTx>)

/**
* Specifies the transaction id, the position of the consumed state in the inputs, and
* the caller identity requesting the commit
*/
data class ConsumingTx(val id: SecureHash, val inputIndex: Int, val requestingParty: Party)
```

Conflict handling and resolution is currently the responsibility of the flow author.

Note that instead of calling the notary directly, we would normally call `FinalityFlow` passing in the `SignedTransaction` (including signatures from the participants) and a list of participants to notify. The flow will request a notary signature if needed, record the notarised transaction, and then send a copy of the transaction to all participants for them to store. `FinalityFlow` delegates to `NotaryFlow.Client` followed by `BroadcastTransactionFlow` to do the actual work of notarising and broadcasting the transaction. For example:

```
subFlow(FinalityFlow(moveTransaction, setOf(aliceParty))
```