Mercury Persistence Architecture

As described in the architecture document Mercury uses a state machine. Therefore its' state is primarily kept on three classes. RMSSequence (at RMS side), RMDSequence and InvokerBuffer (at RMD side). The main goal of the persistence module is to save states of these objects in a database so that the state can be recovered in a machine crash. In this regard persistence has two aspects.

1. Save the state while running the sequence
2. Restore the state after a crash

Here we have to think about the fact that most of the current server hardware fails rarely. Therefore we need to have an architecture which maximize the performance at a reliable state while being able to recover from a failure. Since this only updates the persistence state at runtime it add a minimal overhead.

Save the state while running the sequence

All the events which changes the state of above objects have an exclusive access to the object. So one particular event change the object state at one time. Therefore if we can save this change to persistence storage while updating the state we can keep the same state as in the object in the persistence storage. Here it can be seen that this persistence storage change should be atomic.

Lets take the createSequenceResponseReceived method to illustrate this.

```
public synchronized void createSequenceResponseReceived() throws PersistanceException {
      this.lastAccessedTime = System.currentTimeMillis();
      int currentState = state;
      switch (state) {
         case STATE_0100 : {
            // advance state to 1100
            state = STATE_1100;
            break;
         }
         case STATE_0101 : {
            state = STATE_1101;
            break;
         }
         case STATE_0110 : {
            state = STATE_1110;
            break;
         }
```

```
            case STATE_1110 : {
                break;
            }
            case STATE_1000 : {
                break;
            }
            case STATE_1100 : {
                break;
            }
            case STATE_1101 : {
                break;
            }
        }

        PersistanceManager persistanceManager = getPersistanceManager();
        if (persistanceManager != null){
            this.persistanceDto.setState(this.state);
            this.persistanceDto.setLastAccessedTime(this.lastAccessedTime);
            this.persistanceDto.setSequenceID(this.sequenceID);
            try {
                persistanceManager.update(this.persistanceDto);
            } catch (PersistanceException e) {
                // roll back the state
                this.persistanceDto.setState(currentState);
                this.state = currentState;
                this.sequenceID = null;
                throw new PersistanceException("Can not updated the RMS state for received create sequene
" +
                        "response message ",e);
            }
        }
    }
```

The persistence interface consists of a PersistanceManager and a set of data transfer objects to retrieve and update the state. PersistanceManager interface has methods corresponding to each state change. So these updates must happen as one atomic transaction.
Now lets see the method. First it changes the state while storing the previous state in currentState local variable. Then it updates the sate by calling the persistence interface. if something goes wrong persistence interface throws and exception and Mercury would roll back its state. In this way Mercury keeps the same memory state in the persistence storage.

Lets take the
public void update(RMSSequenceDto rmsSequenceDto) throws PersistanceException;
method implementation in HibernatePersistanceManager.

```
 public void update(RMSSequenceDto rmsSequenceDto) throws PersistanceException {
        Session session = null;
        try {
            session = sessionFactory.getCurrentSession();
            session.beginTransaction();
            session.update(rmsSequenceDto);
            session.getTransaction().commit();
        } catch (HibernateException e) {
            if (session.getTransaction().isActive()) {
                session.getTransaction().rollback();
            }
            throw new PersistanceException("Can not update the RMS Sequence object with state ==> " +
                    rmsSequenceDto.getState() + " toAddress " + rmsSequenceDto.getEndPointAddress(), e);
        }
    }
```

Here the transaction handling happens only at the Persistence implementation level and this has enable us to plug any persistence manager implementation easily.
Like wise for each and every event which changes state we have a similar way to update the state.

Restore a state after a crash

Here one of the problems is how to restart a sequence if the client crashes while sending a sequence of messages or after client has send messages to RMS but RMS has not been able to send all the messages to the RMD. The only solution is to adhere to the so called end to end argument .

Here it is assumed that it is the responsibly of the Application to check the RMSSequnce (corresponding persistence storage) table and resume the sequence. Here the sequence can be resumed as follows.

```
serviceClient.getOptions().setProperty(WSO2RMClientConstants.INTERNAL_KEY, "key1");
        serviceClient.getOptions().setProperty(WSO2RMClientConstants.RESUME_SEQUENCE,
Constants.VALUE_TRUE);
```

This resume message is a dummy message and once WS02RMOutHandler get this message and it restores the session from the persistence storage. At the server side when ever it gets an unknown sequence number it search in the persistence storage and restore the session.
See the table structure for more information.

Limitations

InOut operations can only be used with in memory model. Persistence inout operations can not be supported due to following reasons.

1. The callback objects can not be persisted.
2. At client side Axis2 keeps a map with the send MessageID and Operation context. Axis2 users this to dispatch the received message. This requires some Axis2 level persistence.
3. At the client side also when sending the Response back it requires the original operational context.