Mercury Architecture

By looking at an very abstract level, Mercury has the following main architectural components.

1. State handling model how to keep the sequence state at each side.
2. Thread handling model how to fork new threads and how threads being handled.
3. Persistence handling model how to persists the states at each side, and retrieval of them in a machine crash (This is given in a separate document).

State Handling model

Although Mercury supports inout operations, for simplicity we take an in only operation for this discussion. At inout operations for response path, Server side becomes the client and client side becomes the Server in a similar way.

There are two objects called RMSContext and RMDContext to handle the Mercury specific states at the RMS and RMD respectively. At the RMS, RMSSequence is used to keep the Mercury state. RMSContext keeps RMSSequence objects with three kind of keys with three maps.

1. Internal Sequence key map to identify the RMSSequence object for a particular epr and key combination specified by the user.
2. Message ID key map to keep the RMSSequence objects until the CSR message comes.
3. Sequence ID key map to keep the RMSSequence objects with session Ids.

At the RMD, RMDSequence is used to keep the Mercury state. RMDContext keeps RMDSequence objects with only sequence Id keys.

1. Session ID key map to keep the session states.

RMDSequence keep another object called the invoker buffer to handle in order delivery.

RMSSequence

RMS gets the messages from client. Then it buffers them and send to the RMD. After that, upon receiving the Acknowledgments it clears the buffer. In addition to normal messages it gets Last message and terminate signal from client. Thinking in this way shows four variables which affects the RMS state.

1. Sequence Started or not (SS) - Once receive the first application message from client RMS sends a CS message to RMD. Then it supposed to receive a CSR message and start the actual sequence. So this basically means whether it has got CSR or not.
2. Messages In the Buffer or not(MIB) Means whether it has send all the messages or not.
3. Last Message Received from client or not (LMR) Whether the last message is received or not. this is important since RMS can not terminate sequence without knowing the last message.

4. Terminate Message Received from client or not(TMR) Whether client has signaled to terminate. Here we use a dummy message to send this signal.

These four variables forms a possible 16 states for RMSSequene. Out of this there are only 7 possible states. Here are five possible external events to change its state.

1. Create Sequence Response Received (CRR) - receiving the create sequence response.
2. Acknowledgment Received for all messages (ACKR) This changes the state of the Message buffer
3. Last Message Received (LMR)
4. Application Message Received (AMR) last message also an application message. but threat it as a different one for simplicity.
5. Terminate Message Received (TMR)

By studying how above 7 states (and a special TERMINATE state) changes for these events, the state machine for RMS can be developed. See the images for more information.

RMDSequence

RMD gets the message from the network and passes them to the InvokerBuffer. As above thinking in this way would show following possible variables which determines the RMD sate

1. First Message Received or not (FMR) RMD starts the sequence by receiving a CS. Then it sends a CSR. But to ensure RMS received that message it has to wait until it gets the first message.
2. Last Message Received or not (LMR) this means whether it as a knowledge about LM or not.

Terminate Message Received would also seems to be a one. Since sequence is finish upon receiving it. That does not effect to the state.

These two variables forms 4 sates. But only 3 are actually possible. And also we add another two states COMPLETED, TERMINATED for simplicity.
Here are the possible events.

1. Application Message Received (AMR(SC)) Receiving an application message which completes the sequence.
2. Application Message Received (AMR(SNC))
3. Last Message Received (LMS(SC))
4. Last Message Received (LMS(SNC))
5. Terminate Message Received (TMR)

Here also state machine can be determine by studying how above five states changes with these events. See the images for more information.

InvokerBuffer

InvokerBuffer is used to keep the application messages, to ensure it in order delivery. Here are the possible variables which determines the state of the invoker buffer.

1. Last Message Received or not (LMR) Whether this sequence has got the last message or not.
2. Messages In the Buffer (MIB) or not Whether there are messages with already has received but has not send to application.
3. Terminate Message Received or not (TMR) Has got the terminate message or not

These three variables forms 8 possible states. Out of these 8 states 3 can be considered as one complete state. So this gives 6 possible states.

Here are the external events that may change the sate.

1. Last Message Received (LMR)
2. Terminate Message Received (TMR)
3. Application Message Received (AMR)
4. Send Messages to Application (SMA(SC)) Sending messages from the buffer to application.

Again sate machine can be developed by studying the state change with the events.
Now we know how these states are changes. Then we have to come up a method to execute actions in each states. Threading model describes this.

Threading model

WSRM 1.0 specification proposes a complete Asynchronous communication using separate channels to send requests and receive responses. So the Duplex model can be implemented with completely Asynchronous manner where RMD can start threads to send the messages. But Replay model suggests to use the back channel to receive the responses. This prohibits using server side threads. For in out operations, Axis2 kernel suggests to use the original thread, which is used to send the request to RM Handler to send the Response messages to Application. Therefore thread model is differ for Duplex mode (WS-RM 1.0 specification) and Anonymous mode (Replay model).

Duplex mode

Mercury in other words is a fault tolerant system. All events occurs in Asynchronous manner. Any message can come at anytime with any sequence. Therefore this design uses totally Asynchronous thread model. When receiving the events those threads update the state in each side. Then there is a separate thread (RMDSequenceWorker and RMDSequenceWorker) runs the possible actions in each

state. When sending a message these workers uses a separate thread called MessageWorker to improve the efficiency. Similarly InvokerWorker runs on the InvokBuffer.
This Asynchronous nature and keeping state in one object has completely avoid the

1. Possible Race Conditions - Race Conditions Occurs only if we have to synchronize threads.
2. Dead locks To happen dead locks we need to at least have two objects (which are possibly locked at runtime) referring to each other.

Anonymous mode

Server side

The main difference happens at the Server side where it can not initiate any threads and have to use the same thread to send the response. Therefore RMDSequenceWorker is not being used as in the Duplex mode. Same thread is used to send the response. This is bit a problem when handling the in out operations. For in out operations following steps used to send the response in the back channel.

1. First when the request comes it gives the request message to the RMDSequence and hence to the invoker buffer. Here we have to keep in mind that request should invoke the MessageReceiver (Service) in order manner.
2. Then the original thread waits on the corresponding RMSSequence (For annonymous inout operations RMS must make a sequence offer and offered sequence is known at RMD creation time) until it gets its' response. When application message is received to RMS it notifies those waiting threads and those threads send the response message.

Client Side

At the client side we have to consider the way Axis2 handles the same channel invocations. For same channel mode the transport only sets the input stream (for http) or creates the soap message (for smtp) and the thread which sent the message suppose to populate the response messageContex and call the AxisEngine.receive(). For inonly operations and the RM messages this is done at the Messageworker. For in out operations it is done in the following way

1. First the original thread put the message to the RMSSequence at the client side and waits until the response comes.
2. When the response comes Messageworker notifies this to original thread. Then the original thread is activated and it calls the AxisEngine.receive() method.
3. Then this message is given to the invoker buffer and again this threads wait until invoker buffer notifies this thread in order.
4. Finally the original thread returns and it either sends the message to the client or call the callback if it is an asynchronous invocation.