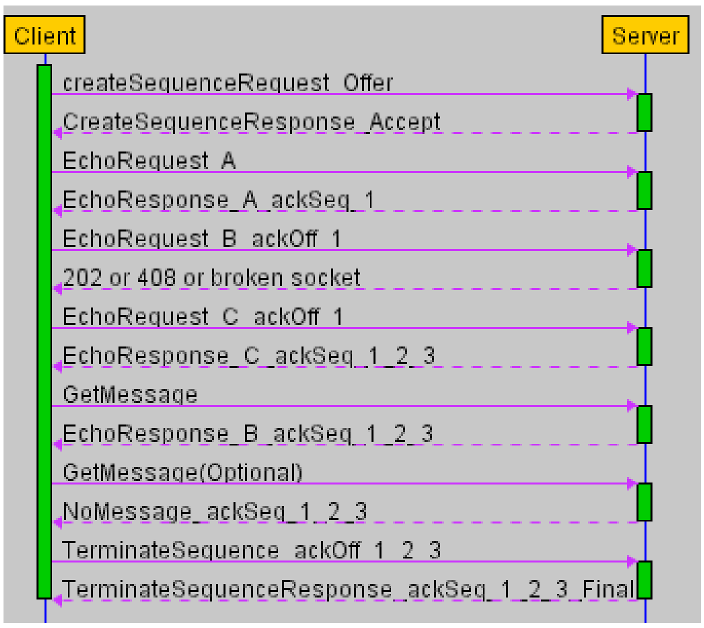
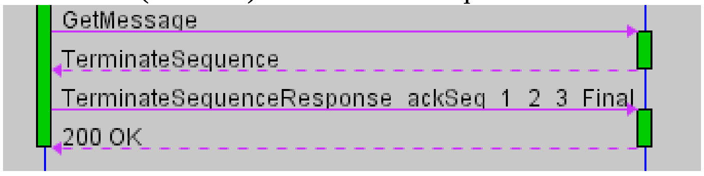

Client
Server
createSequenceRequest_Offer
CreateSequenceResponse_Accept
EchoRequest_A
EchoResponse_A_ackSeq_1
EchoRequest_B_ackOff_1
202 or 408 or broken socket
EchoRequest_C_ackOff_1
EchoResponse_C_ackSeq_1_2_3
GetMessage
EchoResponse_B_ackSeq_1_2_3
GetMessage(Optional)
NoMessage_ackSeq_1_2_3
TerminateSequence_ackOff_1_2_3
TerminateSequenceResponse_ackSeq_1_2_3_Final

1. CreateSequence and Offer
2. CSR and Accept
3. EchoRequest A
4. EchoResponse for A and ack for Request A.
5. EchoRequest B succeeds
6. but EchoResponseB is lost. The RM Agent doesn't know the status of either half of the interaction.
7. At this point we could "GetMessage" or we could carry on. After all, the RM agent may not be aware that EchoRequestB had a response. In this case the RM Agent sends: EchoRequestC
8. EchoResponse for C and also acks the outgoing messages 1-3
9. At some interval, or before closing up the sequence, the client's RM Agent should GetMessage. It knows to do this because there is an offered sequence with an anonymous endpoint.
10. The "waiting" EchoResponseB comes back
11. The RM agent might do another GetMessage, since the last one returned a message its possible there are still messages waiting.
12. NoMessage indicates that there are no messages waiting on the server side.
13. The client is now finished sending messages. So it can Terminate the outgoing sequence.
14. The TSR comes back

The offered sequence at this point is still potentially open. Both sides can independently clean up if they are aware of the MEPs. Alternatively, there may be situations where the Offered sequence lives beyond the life of the outgoing sequence (for example if the messages use the in-multi-out MEP). At some point the server will Terminate (or close) the Offered Sequence and the flow looks like this:



15. At some point the clients RM agent sends another GetMessage and in this case gets a TerminateSequence for the Offered sequence.
16. It asynchronously responds with the TSR.