Sliding Window, based on posted SlidingWindowSender, SlidingWindowReceiver, and UnreliableChannel.
Let the tasks be
{ { send(m): m ∈ Messages } } for SlidingWindowSender
{ { receive(m) } : m ∈ Messages } for UnreliableChannel
{ { read(d) : d ∈ Int } , { {send(m) }: m ∈ Messages } for SlidingWindowReceiver

The liveness property is: If each message is dropped at most finitely many times by the UnreliableChannel, each message will eventually be delivered to the Reader, ie, in a fair execution, every write(i) is eventually followed by a read(i).

Discussion

We know that for a message to be delivered, there's a chain of events that has to happen: each message must get from the Writer to the SlidingWindowSender, from the SlidingWindowSender to the UnreliableChannel, from the UnreliableChannel to the SlidingWindowReceiver, and from the SlidingWindowReceiver to the Reader.

This requires showing that each automaton is either immediately enabled to send the message on once it receives it, or that it is eventually enabled to send it on. The former applies to writes from the Writer to the SlidingWindowReceiver, to receives from the UnreliableChannel to the SlidingWindowReceiver, and to reads from the SlidingWindowReceiver to the Reader.

In addition, we observe that the receive of a given message may not be enabled forever in the UnreliableChannel, since it can be dropped. So to get the message through the channel, we rely on two things: 1) it will be re-sent continually by the SlidingWindowSender until it has been acked; and 2) the UnreliableChannel drops each message at most finitely many times. However, the other actions mentioned above are enabled forever once they have been enabled.

Finally, we need to know that the SlidingWindowSender will eventually send each message that is written to it. This relies on it sending all messages in its window until they are acked (Claim 1, below) and also on it eventually receiving an ack for each message it has sent (Claim ?).

The following claims assume fair executions. An interesting thing to do, when studying for the midterm, is to consider each of these as a liveness condition for one automaton.

Claim 1 for SlidingWindowSender. For any write(i) with i≤lastAckReceived+window, the action send(m, B, A) occurs repeatedly until lastAckReceived≥ m.seq.

Proof: Following each write(i), some sendBuf[index] for
min(i,lastAckReceived+window)≥index ≥ lastAckReceived+1

is enabled until send(m[index], B, A) occurs. Fairness requires that eventually send(m[index], B, A) occurs and index is set to index+1, or else back to lastAckReceived+1 if

index> min(lastFrameWritten,lastAckReceived+window).

Inductively, we see that for every index with

min(index,lastAckReceived+window)≥index≥lastAckReceived+1,

eventually send(m[index], B, A) occurs. Furthermore, if lastAckReceived is never updated, each of these must occur infinitely often, i.e., index cycles from lastAckReceived+1 to lastAckReceived+window forever.

Later, we will show that for every i, eventually lastAckReceived+window≥i, so that send(m[i], B, A) must finally occur.

Claim 2 for SlidingWindowSender and UnreliableChannel. Following every send(m, B, A) action, there is eventually a receive(m, B, A).

Proof: Every message m sent through UnreliableChannel is either delivered to SlidingWindowReceiver (with an action receive(m), B, A)) or it is dropped. By assumption, each message is dropped at most finitely many times, but if a message is not delivered it will not be acked, and by Claim 1 above we see that any message that is not acked will be sent through UnreliableChannel infinitely often. So eventually, the action receive(m, B, A) will be enabled continuously until it occurs. Fairness requires that receive(m, B, A) eventually occurs.

Claim 3 for SlidingWindowReceiver. Following each receive(m, B, A) with m.data = i, there is a read(i).

Proof: The following relations hold between the state variables in SlidingWindowReceiver:

1) Any acknoweldgment sent has an ack value ≤ lastFrameReceived
2) lastAcceptableFrame = lastFrameRead+window.
3) lastFrameRead ≤ lastFrameReceived

1) and 2) are obvious from inspection of the code; 3) follows because lastFrameReceived is the same as the largest index in receiveBuf with a non-zero entry.

It follows from 1) that SlidingWindowSender.lastFrameAcknowledged ≤ SlidingWindowReceiver.lastFrameReceived and so SlidingWIndowSender doesn't send any frames with a sequence number larger than lastAcceptableFrame. As a result, every message received by SlidingWindowReceiver is either placed in the receiveBuf or its sequence number is smaller than lastFrameRead and therefore it can be safely ignored because the action read(i) has already occurred.

Claim 4 for SlidingWindowReceiver. If there is a subsequence of a fair execution of the form

receive(m_1, B, A), ..., receive(m_n, B, A)
such that {m_1.seq, ..., m_n.seq} = {1,...,n }, then lastFrameReceived≥n (and thus all actions send(m, A, B) where m.ack=m_i are enabled forever) and receiveBuf[m_i.seq] =m_i.

In other words, every received message is put in the receiveBuf and acked by SlidingWindowReceiver.

Proof: The "for" loop in receive guarantees that lastFrameReceived has the required property. Also, by the construction of the receive action, any m_i that goes into the receiveBuf goes into receiveBuf[m_i.seq] and for i≤lastFrameReceived, receiveBuf[i]~=empty, so that for m_i.seq≤lastFrameReceived, receiveBuf[m_i.seq] = m_i. send(m, A, B) is always enabled if m.ack≤lastFrameReceived (note change to tioa so that m.ack <= SlidingWindowReceiver.lastFrameReceived). So there will be infinitely many messages send(m, A, B) in any fair exection. (**If the precondition has an =, then liveness will not hold; why not?**).

Claim 5 for SlidingWindowReceiver and UnreliableChannel. Any action send(m, A, B) is followed eventually by an action receive(m,A, B)

Proof: Similar to Claim 2; instead of depending on sends from the window happening repeatedly, this depends on acks happening repeatedly.

Claim 6 for the entire system. For any write(i) in a fair execution, there is eventually a receive(m, B, A) with m.seq=i and also eventually (and even later) lastAckReceived = i.

Proof: This depends on the message being sent (Claim 1), received (Claim 2), acknowledged (Claim 4), and receipt of the acknowledgment (Claim 5).

By induction on lastAckReceived:
For the induction base use the above claims to establish that after write(1), eventually receive(m, B, A) with m.seq=1 must occur and subsequently send(m, A, B) with m.ack=1 occurs, and the corresponding receive, so that lastAckReceived takes the value 1.

The induction hypothesis is that following write(i), there is eventually an action receive(m, B, A) with m.seq=i and subsequently lastAckReceived eventually becomes i.

The induction step uses the above claims, as in the base case, to show that once lastAckReceived reaches i, either the action receive(m, B, A) with m.seq=i+1 has already occurred (if it happened out of order) or if not it will eventually occur. In either case, because **lastAckReceived≤lastFrameReceived** (state invariant!! – consider how to prove this using induction), it must be true that lastFrameReceived≥i+1 after the action receive(m, B, A) with m.seq=i+1. Thus the ack of i+1 is enabled, the ack of i+1 will be sent and received, and lastAckReceived will reach i+1.

Finally: Using claim 6, note every write(i) is followed eventually by a receive(m, B, A) with m.seq=i.