Concurrent and Distributed Systems (RNW)

This pseudocode, executing in process \( P_j \), employs buffering to impose ordering:

```plaintext
receive(M from Pi) { // Message M received from process Pi
    S = getSeq(M); // Extract sequence number S
    if (S == nextSeq(Sji)) { // If S is the next sequence number:
        deliver(M); // Deliver M to current process (Pj)
        Sji = flush(HBQ, Sji); // Deliver backlog from HBQ; update Sji
    } else holdback(HBQ, M); // Else: Hold back M for future delivery
}
```

(a) Explain what ordering model(s) this pseudocode implements. [2 marks]

(b) Write pseudocode (with comments) for the following functions, to be used on
the sender (\( P_i \)) or receiver (\( P_j \)), which accept \( M \) (a message), and \( S \) (a sequence
number):

- Receiver: `receive_reliably(M)` Reliably receive \( M \) from \( P_i \).
- Sender: `send_reliably(M)` Reliably send \( M \) to \( P_j \).
- Sender: `process_ack(S)` Handle a received ACK for \( S \) from \( P_j \).
- Sender: `timeout(S, M)` Process a timeout for \( S \) and \( M \).

As needed, employ the following additional utility functions:

- `drop(M)` Drop received \( M \) without delivering.
- `setSeq(M, S)` Set sequence number \( S \) on message \( M \).
- `transmit_msg(M)` Transmit message \( M \) to \( P_j \).
- `transmit_ack(S)` Transmit an ACK with sequence number \( S \) to \( P_i \).
- `sched_timeout(S, M)` Schedule timeout \( S, M \) to run in 5 ms.
- `cancel_timeout(S)` If scheduled, cancel timeout for \( S \).

(c) Define the `happens-before` relationship. [2 marks]

(d) The pseudocode above imposes ordering on pair-wise communications. Assuming
reordering but no message loss, write pseudocode (with comments) for the
following functions supporting `causal ordering` for group communications:

- Receiver: `receive_causally(M)` Causally receive from the group.
- Sender: `send_causally(M)` Causally send to the group.

As needed, employ the following additional utility functions:

- `getVec(M)` Retrieves the version vector from a message.
- `setVec(M, V)` Set vector \( V \) on message \( M \).
- `testVec(LV, RV)` Returns whether vector \( RV \) only differs from \( LV \) in
  that it has exactly one entry one greater than the corresponding entry in \( LV \).
- `updateVec(V)` Returns \( V \) with the local vector entry incremented.
- `transmit_group(M)` Transmits message \( M \) to the entire group.