

Question 1

Part 2

Assumption: “propagate the latest weather update” means that all connected clients will not only be notified of the latest weather, but **will also receive the latest weather**.

Let us define

- `isLatestSent` – whether the WCP has sent the update message to the CM.
`#define isLatestSent (cm_chan ? <USER_UPDATED_WEATHER, DUMMY_VAL, DUMMY_VAL>)`
- `are_all_new_successful` – whether all the connected clients reported success for using the new weather.

Then, the LTL property used is:

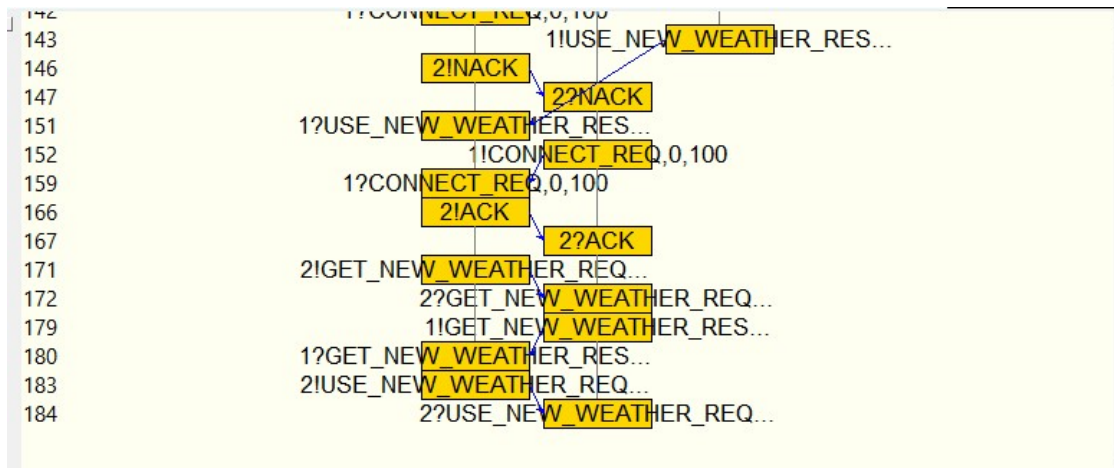
`[] (isLatestSent -> <> are_all_new_successful)`

It means:

Globally, if the Communication Manager is sent a weather update message, then eventually, all connected clients will successfully use the new weather (and send the corresponding message to the CM).

Part 3

The exact nature of the deadlock seems to depend on the implementation of the model. In my final implementation, states are written by the CM without using message passing, but can be read by clients and WCP. The resultant counter example encountered is:



The property that should hold is “each receive operation should eventually get a message from the corresponding send operation”. This does not hold for the CM.

In the MSC above, the CM sends the `USE_NEW_WEATHER` message to the client (step 183). The client receives the message (step 184) but it is in the `INITIALIZING` state. So, it does not process it, and line 88 of the code below is executed instead of line 95.

The CM is in the `POST_INITIALIZING` state and is waiting for a message from the client, but no response is sent from the client to the CM regarding the success status of using the new weather information (since the client did not process it). This results in a deadlock.

```
78  :: client_chan[id] ? req ->
79      if
80      /* Step A4a. B4a. Client response to Get New Weather info */
81      :: (client_status[id] == INITIALIZING || client_status[id] == UPDATING) -> {
82      if :: (req == GET_NEW_WEATHER_REQ) -> {
83          set_is_successful();
84          cm_chan ! GET_NEW_WEATHER_RESP, id, is_successful;
85      }
86
87      :: else ->
88      | printf("===Error: req=%d (should be 8- GET_NEW_WEATHER_REQ)\n", req);
89      fi
90  }
91
92  /* Step A5a. B5a. Client response to Use New Weather info */
93  :: (client_status[id] == POST_INITIALIZING || client_status[id] == POST_UPDATING) ->
94  if :: (req == USE_NEW_WEATHER_REQ) -> {
95      set_is_successful();
96      cm_chan ! USE_NEW_WEATHER_RESP, id, is_successful;
97  }
98
99  :: else
100  fi
```

Other Issues with the Protocol

1. In Step 4 of *Client Initialization*, if the client reports failure, the WCP is not re-enabled.
2. The key property mentioned in Part 2 does not hold. During the *Weather Update*, if a connected client reports failure to get the new weather or to use the new weather, they are asked to use the old weather or are disconnected. Hence, there can be scenarios in which the information from the weather update doesn't reach the connected clients.

Part 4

The deadlock can be fixed by putting the statements in an atomic block (see lines 217 to 221 below). Now, the client is guaranteed to be in the correct state (`POST_INITIALIZING`) to receive the `USE_NEW_WEATHER_REQ` message.

```

214 /* Step A4b. CM action to client response for Get New Weather info */
215 :: (cm_status == INITIALIZING && id == client_id && req == GET_NEW_WEATHER_RESP) ->
216 if :: (val == 1) ->
217     atomic {
218         client_chan[client_id] ! USE_NEW_WEATHER_REQ;
219         cm_status = POST_INITIALIZING;
220         client_status[client_id] = POST_INITIALIZING;
221     }
222
223 :: else ->
224     client_status[client_id] = DISCONNECTED;
225     cm_status = IDLE;
226     client_id = DUMMY_VAL;
227     // FIXME: Enable WCP?
228 fi

```

To prove that this fix is sufficient, use the “safety” option in “Verification” tool in iSpin. When we check for deadlocks, the output displays “No errors found”, while previously (in part 3) it would display “To replay the error-trail, goto Simulate/Replay and select “Run””.

The screenshot shows the iSpin verification tool interface. The top menu bar includes 'Edit/View', 'Simulate / Replay', 'Verification', 'Swarm Run', '<Help>', 'Save Session', 'Restore Session', and '<Quit>'. The 'Verification' tab is active, displaying a table with three columns: 'Safety', 'Storage Mode', and 'Search Mode'. The 'Safety' column has checkboxes for 'safety' (checked), 'Liveness', 'non-progress cycles', 'acceptance cycles', and 'enforce weak fairness constraint'. The 'Storage Mode' column has checkboxes for 'exhaustive' (checked), 'minimized automata (slow)', 'collapse compression', 'hash-compact', and 'bitstate/supertrace'. The 'Search Mode' column has checkboxes for 'depth-first search' (checked), 'partial order reduction', 'bounded context switching', 'breadth-first search', 'iterative search for short trail', 'partial order reduction', and 'report unreachable code'. Below the table are buttons for 'Run', 'Stop', 'Save Result in', and 'pan.out'. The bottom panel shows the results of the verification, including the number of states, transitions, and atomic steps, as well as memory usage statistics and a list of unreachable states.

initialization.pml

Spin Version 6.5.0 -- 1 July 2019 : iSpin Version 1.1.4 -- 27 November 2014

Safety	Storage Mode	Search Mode
<input checked="" type="checkbox"/> safety	<input checked="" type="checkbox"/> exhaustive	<input checked="" type="checkbox"/> depth-first search
<input checked="" type="checkbox"/> + invalid endstates (deadlock)	<input checked="" type="checkbox"/> + minimized automata (slow)	<input checked="" type="checkbox"/> + partial order reduction
<input checked="" type="checkbox"/> + assertion violations	<input type="checkbox"/> + collapse compression	<input type="checkbox"/> + bounded context switching
<input type="checkbox"/> + xi/xs assertions	<input type="checkbox"/> hash-compact <input type="checkbox"/> bitstate/supertrace	<input type="checkbox"/> with bound: 0
<input type="checkbox"/> Liveness	<input type="checkbox"/> Never Claims	<input type="checkbox"/> + iterative search for short trail
<input type="checkbox"/> non-progress cycles	<input type="checkbox"/> do not use a never claim or ltl property	<input type="checkbox"/> breadth-first search
<input type="checkbox"/> acceptance cycles	<input type="checkbox"/> use claim	<input checked="" type="checkbox"/> + partial order reduction
<input type="checkbox"/> enforce weak fairness constraint	claim name (opt):	<input checked="" type="checkbox"/> report unreachable code

Run Stop Save Result in pan.out

```

1 mtype = {
2   /*21*/ POST_REVERTING,
3   /*20*/ POST_UPDATING,
4   /*19*/ UPDATING,
5   /*18*/ PRE_UPDATING,
6
7   /*17*/ IDLE, /* CM and for connected clients */
8   /*16*/ PRE_INITIALIZING,
9   /*15*/ INITIALIZING,
10  /*14*/ POST_INITIALIZING,
11
12  /*13*/ DISCONNECTED,
13
14  /* WCP */
15  /*12*/ ENABLED,
16  /*11*/ DISABLED,
17
18  /*10*/ USER_UPDATED_WEATHER, /* WCP message */
19
20  /*9*/ CONNECT_REQ,
21  /*8*/ GET_NEW_WEATHER_REQ,
22  /*7*/ GET_NEW_WEATHER_RESP,
23  /*6*/ USE_NEW_WEATHER_REQ,
24  /*5*/ USE_NEW_WEATHER_RESP,
25  /*4*/ USE_OLD_WEATHER_REQ,
26  /*3*/ USE_OLD_WEATHER_RESP,
27
28  /*2*/ ACK,
29  /*1*/ NACK,
30 }

```

2695913 states, matched
6075778 transitions (= stored+matched)
321156 atomic steps
hash conflicts: 120576 (resolved)

Stats on memory usage (in Megabytes):
348.115 equivalent memory usage for states (stored*(State-vector + overhead))
284.196 actual memory usage for states (compression: 81.64%)
state-vector as stored = 76 byte + 12 byte overhead
64.000 memory used for hash table (-w24)
0.343 memory used for DFS stack (-m10000)
348.328 total actual memory usage

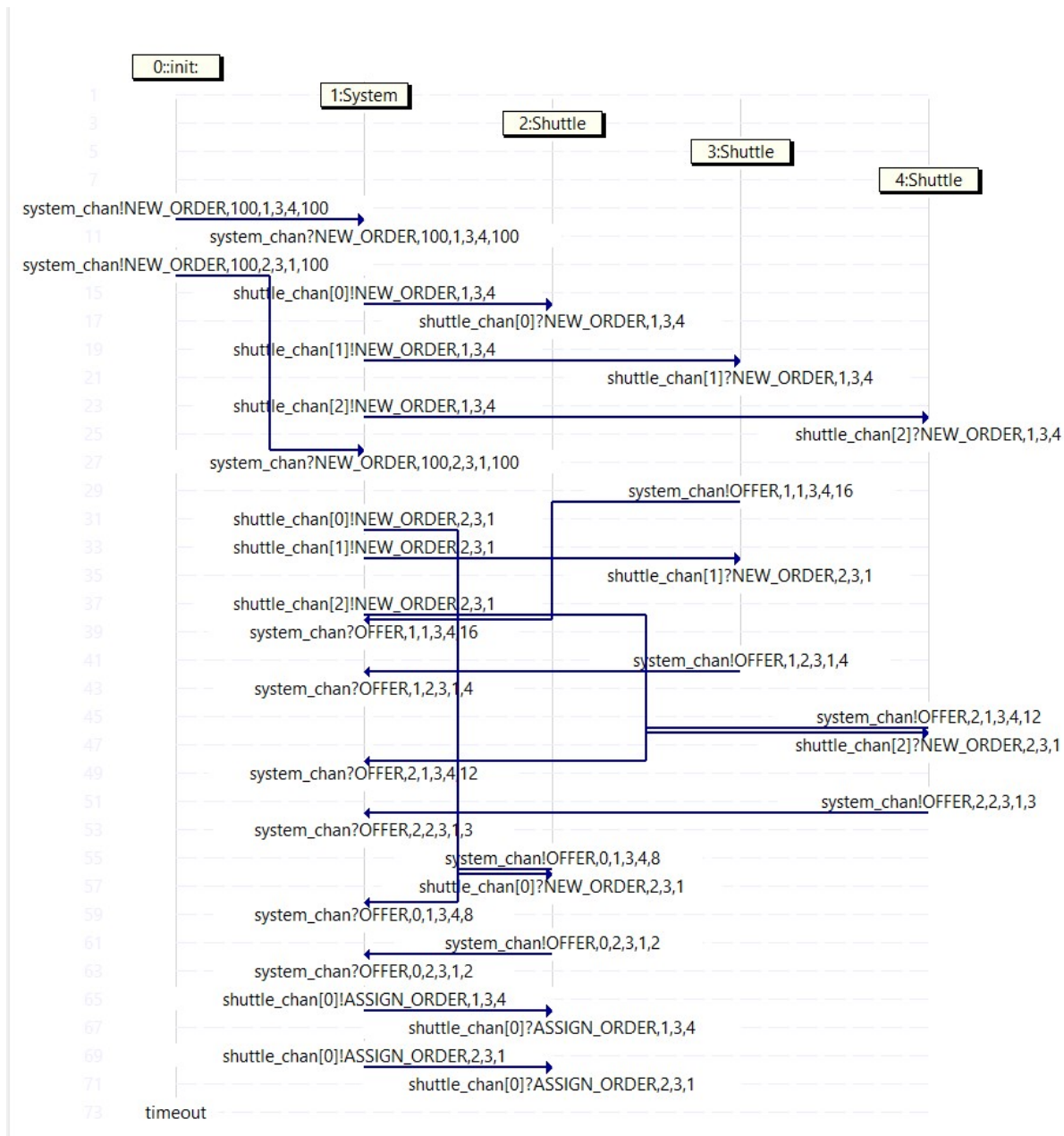
unreached in proctype Client
initialization.pml:112, state 74, "-end-"
(1 of 74 states)
unreached in proctype CM
initialization.pml:355, state 335, "-end-"
(1 of 335 states)
unreached in proctype WCP
initialization.pml:372, state 14, "-end-"
(1 of 14 states)
unreached in Init
(0 of 7 states)

pan: elapsed time 1.7 seconds
No errors found -- did you verify all claims?

Question 2

Part 2

Using SPIN and running `spin -M railway.pml` gives this Message Sequence Chart that shows one sequence of events that leads to the desired results.



Sequence of events

Assignment of orders:

- Management System receives a **NEW_ORDER** with **start=1, dest=3, size=4** (order 1).
- Management System sends order 1 to the 3 shuttles using the **shuttle_chan[idx]** buffer.
- Management System receives a **NEW_ORDER** with **start=2, dest=3, size=1** (order 2).
- Shuttle 2 sends an offer for order 1 to Management System with **payment=16**.
- Management System sends order 2 to the 3 shuttles using the **shuttle_chan[idx]** buffer.
- Shuttle 2 sends an offer for order 2 to Management System with **payment=4**.
- Shuttle 3 sends an offer for order 1 to Management System with **payment=12**.
- Shuttle 3 sends an offer for order 2 to Management System with **payment=3**.

- Shuttle 1 sends an offer for order 1 to Management System with `payment=8`.
- Shuttle 1 sends an offer for order 2 to Management System with `payment=2`.
- Management System sends an `ASSIGN_ORDER` message to Shuttle 1 for order 1.
- Management System sends an `ASSIGN_ORDER` message to Shuttle 1 for order 2.

Processing of orders by Shuttle 1:

- Shuttle 1 processes the orders one at a time, starting from the earliest one.
- It chooses the initial direction to travel in based on the distance of the route to the start station.
- It then travels in the chosen direction, while checking that the track is used by only one shuttle.
- Once the start station (Station 1 and then Station 2) is reached, it loads the passengers.
It chooses the direction to travel in based on the distance of the route to the destination station.
- It then travels in the chosen direction, while checking that the track is used by only one shuttle.
- Once the destination station (Station 3 and then Station 3) is reached, it unloads the passengers.

At the end of the execution, the shuttles are:

- Shuttle 1: Stationary at Station 3 with no load
- Shuttle 2: Stationary at Station 1 with no load
- Shuttle 3: Stationary at Station 2 with no load

This can be seen from the Data Window in iSpin after the run terminates. For example, the screenshot below shows the values of the variables in the process corresponding to Shuttle 1 (note the values of `curr_station`, `load` and `status`) after running a simulation with `seed = 123` on iSpin. Similar results are observed for Shuttle 2 and Shuttle 3.

```
:init:(0):o1.start = 1
:init:(0):o2.dest = 3
:init:(0):o2.size = 1
:init:(0):o2.start = 2
Shuttle(2):are_passengers_loaded = 0
Shuttle(2):can_travel = 1
Shuttle(2):capacity = 5
Shuttle(2):charge = 2
Shuttle(2):curr_order = 2
Shuttle(2):curr_station = 3
Shuttle(2):distance = 1
Shuttle(2):i = 3
Shuttle(2):id = 0
Shuttle(2):idx = 1
Shuttle(2):j = 1
Shuttle(2):load = 0
Shuttle(2):msg = ASSIGN_ORDER
Shuttle(2):num_orders = 2
Shuttle(2):o.dest = 3
Shuttle(2):o.size = 1
Shuttle(2):o.start = 2
Shuttle(2):orders[0].dest = 3
Shuttle(2):orders[0].size = 4
Shuttle(2):orders[0].start = 1
Shuttle(2):orders[1].dest = 3
Shuttle(2):orders[1].size = 1
Shuttle(2):orders[1].start = 2
Shuttle(2):payment = 2
Shuttle(2):start_station = 1
Shuttle(2):status = STATIONARY
Shuttle(3):are_passengers_loaded = 0
Shuttle(3):can_travel = 1
Shuttle(3):capacity = 8
```