Name: Chan Tze Yang, Aloysius
Matric Number: A0134202B
Email Address: aloysius.chan@u.nus.edu

Question 1

Part 1

Each station contains a boolean busy which determines if the station is currently busy. In this example, if the station is communicating with another station, then its status will be busy, and conversely, if the station is not communicating with another station, then its status will be not busy. Therefore, we can formalise the required property for the stations as:
$G(busy[0] \Rightarrow F\ !busy[0])\ \&\&\ G(busy[1] \Rightarrow F\ !busy[1])$

Part 2

The issues were derived by tracing the output from a random simulation with the seed of 123.
Issues:

1. There was a potential deadlock caused by lines 13 to 15. Assume that station 1 sent a start signal to station 0. In the original implementation, it is possible that the station 0 receives the start signal but is unable to set its status to busy as lines 14 and 15 are not part of the critical section. Therefore, it is then possible for station 0 to attempt to send a start signal to station 1, since it is not busy. This then causes a deadlock as station 0 waits for station 1 to acknowledge the sent start signal, and station 1 waits for station 0 to also acknowledge the sent start signal.

   To solve this, lines 13 to lines 15 should be covered in a critical section (ie "atomic" block) so that any process that receives a start signal can immediately set itself to busy and acknowledge this start signal.
2. Similarly, there was two potential deadlocks caused by lines 22 to 23. Firstly, it is possible that both station 0 and 1 are not busy. Since line 23 was not part of the critical section as line 22, it is possible for station 0 to become busy, but not send out the start signal immediately. Thereafter, station 1 also becomes busy and also send out a start signal to station 0, resulting in a deadlock. To solve this, lines 22 to 23 should be placed in the same critical section so that any process that wants to send out a start signal should immediately send it out after becoming busy.

   Secondly, assuming that lines 13 to 15 and lines 22 to 23 have already been placed in their respective critical sections, it is still possible for another deadlock to occur. Assume that both station 0 and 1 are not busy. Station 0 sends out a start signal to station 1. However, since station 1 is not yet busy, it can either receive the start signal that station 0 sent out or attempt to send out a start signal to station 0. Hence, we have to prevent station 1 from being able to send out a start signal to station 0 in this case. For this problem, I noted that start signals can only be sent out if both the sending and receiving stations are not busy. Therefore, a new condition is added in line 22.
3. There was a potential deadlock caused by lines 25 to 28. Assuming that both stations have successfully connected to each other and are sending data between each other. Without the presence of a limit in this code section, it is entirely possible that the sender sends infinite data to the receiver. Therefore, I limited the sender to only send a maximum of 999999 data per run, before requiring it to send a stop signal.

After making these amendments, I tested the model with the LTL property defined in 1, and there were not error trails emitted by the model. This verified that the model satisfies the required property.

Question 2

Part 1
Refer to code

Part 2
In my code, I assumed that the node has a buffer size of 8, although this deadlock will occur for any finite buffer size larger than 3. Using one example to showcase the deadlock, since each of the incoming streams independently sends red, green and blue messages to the node, it is possible that the node receives x red signals and (8 – x) green signals. Upon receiving these signals, the node's buffer becomes filled, and is unable to receive any more messages from the incoming streams. However, to send a reassembled message to the outgoing stream, the node required one of each type of message. Hence, at this moment, the node waits for a blue signal that is unable to enter the node, causing the reassembly deadlock.

Part 3
Since the program is supposed to reassemble red, green and blue messages, we can define the atomic properties as:

1. p: at least one red sent to node
2. q: at least one green sent to node
3. r: at least one blue sent to node
4. s: node outputs a reassembled message

Therefore, the LTL property is $G\ (p \;\&\&\; q \;\&\&\; r \Rightarrow F\ s)$. Converted to English, this will mean that if at least one red, green and blue have been sent to the node, then we should expect the node to eventually output a reassembled message.

In this case, using SPIN, we showed that this property is violated. The example trail showed that at least one of each type of messages was sent to the node. However, since the node's buffer was full, it was unable to store at least one of each type of message, preventing it from outputting a reassembled message. Hence, the atomic proposition s cannot be eventually achieved.

Question 3
Part 1
Refer to code.

Part 2
Assume that we only have two clients. For the LTL property, we define the atomic propositions as:

1. p: WCP is enabled
2. q: The WCP sent out at least one update signal to the communication manager
3. $r1$: Client 1 is connected
4. $r2$: Client 2 is connected
5. $s1$: Client 1 uses the latest weather update
6. $s2$: Client 2 uses the latest weather update

Hence, the LTL property is: $G(!p \Rightarrow Fp) \;\&\&\; G(q \Rightarrow F((r1 \Rightarrow s1) \;\&\&\; (r2 \Rightarrow s2)))$. Converted to English, this means that if the WCP is disabled, it will eventually be enabled, and if the WCP sent out at least one signal to update the weather, then eventually, every connected client will use this weather update.

Part 3
The deadlock can be seen from the error trail provided by SPIN. The issue comes from the fact that the WCP can be disabled forever.

For example, a disconnected client may attempt to connect to the communication manager. Thereafter, the communication manager disables the WCP. The client then goes through the normal initialisation process, but fails to get the new weather. In this case, the client will be disconnected, but the WCP does not get reenabled. Hence, this forms a loop where the client can continuously attempt to connect to the client, and keep failing at getting the initial weather. This prevents the WCP from ever getting reenabled, causing a deadlock. To solve this, we have to ensure that the WCP gets reenabled if the client fails to get the initial weather.

Another example is when the CM receives an update signal from the WCP. Thereafter, the WCP gets disabled. A disconnected client then bombards the CM with connecting signals, causing the CM to be continuously reply to this client, and be unable to continue with its execution. This also prevents the WCP from being enabled again, causing a deadlock. To solve this, we need to ensure that the client has knowledge about the CM's status and only send a signal when the CM is idle.

Part 4
Refer to code.