Dependable Real-time Systems (EDA423)
HWA1 Report

Edvin Mellberg [991110-3613],
Tianqi Wen [010907-0391],
Kavineshver Sivaraman Kathiresan [971110-3672]

Group DRTS 1
Chalmers University of Technology

February 2024

1 Introduction

The aim of this report is to demonstrate and argument for the design decisions made throughout Home Work Assignment 1 in the course EDA423. The software design and solutions to the specified questions for each problem will be explained each in a chapter of its own.

2 Design Overview (for problem 1-4)

The application as a whole consists of four objects: *MusicPlayer*, *TxRxObj*, *ToneObj* and *FailureHandler*. The *MusicPlayer* is the main object that controls the functionality of the whole application. This object stores the state of the application and which mode the board is in (Conductor/Musician). It also keeps track of the tempo, key and tone index. The *TxRxObj* handles inputs from the Serial and CAN ports and distributes these commands to the *MusicPlayer*. The *ToneObj* controls the tone generator and stores the state of the volume. The final object, the *FailureHandler*, enables the application to simulate faults F1 and F2 and also detects fault F3. A simplified illustration of the application is given in Figure 1.

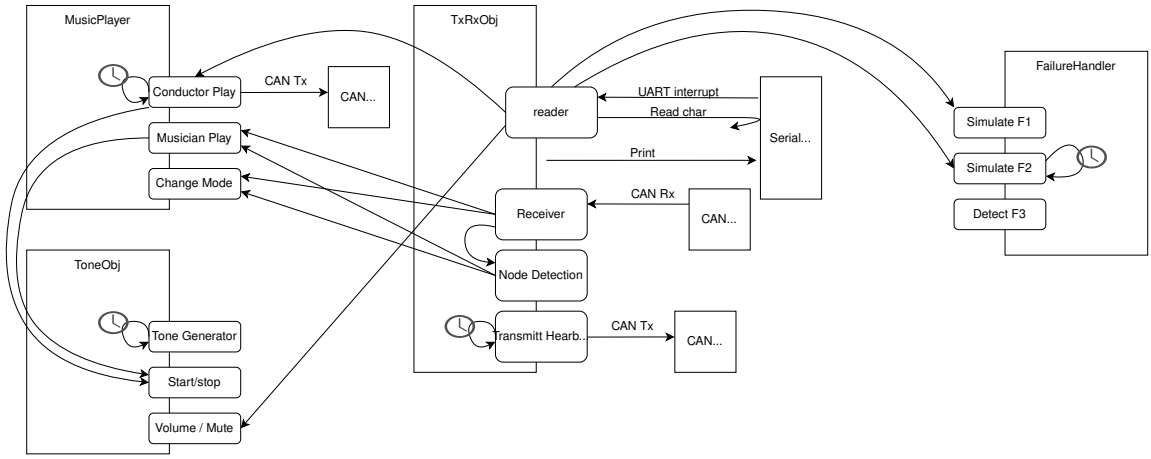

Figure 1: Access graph of the application and interactions between objects.

To be able to communicate with the other boards, a common communication protocol was developed together with the other groups within the Meta group. The resulting protocol is shown in Table 1.

Table 1: Communication protocol

Message	**MsgID**	**Length**	**Buff[0]**	**Buff[1]**	**Buff[2]**	**Buff[3]**	**Buff[4]**	**Buff[5]**
Heartbeat	1	3	Mode	Lead. Num.	Req. Conduct	-	-	-
Play	2	6	Receving Node	Tone Index	Tempo	Key	Vol	Lead. Num.
Update	3	5	Tempo	Key	Vol	Play State	Lead. Num.	-

This protocol consists of three different types of messages: *Heartbeat*, *Play*, and *Update*. The *Heartbeat* is transmitted by every node with a periodicity of 200ms. It contains information about the node, such as its current mode, the current leader number, and if it's currently requesting conductorship. The *Play* message is sent by the conductor to a specific node that should play the tone given by *Tone Index*. This message contains all the information needed to play a tone with the correct length, volume, and key. Finally, the *Update* message is sent by the conductor every time a new user input is given, to make sure that all nodes know the current state of the music player in case a musician needs to assume conductorship.

The *Leader Number* is an important part of each message. This number indicates if the message is valid or not. Each time a new conductor is selected, it will increment the *Leader Number* and start transmitting *Play* messages. Other nodes will detect the incremented *Leader Number* and register that there has been a change in conductorship. If an old conductor would start transmitting, it would do so with an outdated *Leader Number* which enables all other nodes to discard the outdated messages. This allows for a very fault-tolerant system design.

When first starting the application, the board will initialize as a musician. The board will directly start transmitting *Heartbeat* messages to inform other nodes of its presence. It will also detect other nodes' *Heartbeats* and store the node ID in an array that keeps track of the currently active nodes. This allows each board to calculate its own rank in the network. The rank is based on node ID, and a lower node ID gives a higher priority. The rank is used to determine which node should become conductor if the conductor fails or if two nodes request conductorship simultaneously.

As a musician, the board will listen for *Play* messages containing the *Receiving Node* equal to its own node ID. If it receives such a message, it will play the corresponding tone given by the *Tone Index*. If a musician becomes conductor, it will assume the responsibility of transmitting *Play* messages for every note to be played. The conductor calculates the

Receiving Node by using the modulo operator with the current number of nodes in the network.

A more detailed description of how the design works will be given for each problem.

3 Problem 1

The main functionality of problem 1 was for each board to be able to detect the number of nodes in the network, and based on rank decide which of the tones in the melody to play.

To allow the boards to detect one another, we are using the *Heartbeat* message as shown in Table 1. This message will be sent periodically by each node. The board will detect the number of nodes in the network by registering each received *Heartbeat* message and storing the node's state in an array. The different states a node can have is: Unknown, Connected or Failed. All nodes are initialized as Unknown, and when the board receives a *Heartbeat* message, it will set the state of the transmitting node as Connected. After two sequences of *Heartbeat* messages, in other words every 400ms, another method called *Node Detection* will be scheduled which checks how many nodes are in the Connected state and thereby calculates the size of the network. All nodes that are still in the Unknown state at this time are set as Failed since no *Heartbeat* message was transmitted in the last 400ms. After this is done, the Connected nodes will be set back to state Unknown and the process will repeat. This allows the board to dynamically detect the number of nodes in the network (size) and if a node has failed. The periodicity of 400ms for running the method *Node Detection* was selected to give a stable behavior and omit false negatives (falsely assuming a node to have failed) but to also be fast enough to handle a failure within one second.

The other functionalities in problem 1 included the ability to change tempo, key, and volume, as well as muting a musician and enabling printing of the tempo or mute state. Since all of these functions are reactions to user input, we utilized the reactive nature of Tiny Timber to allow asynchronous tasks to be scheduled whenever the user input is given. In the serial reader method, tasks will be scheduled to update the state of the *MusicPlayer* and *ToneObj* objects as shown in Figure 1 depending on which key is pressed on the keyboard. For example, when it comes to printing the mute state of the musician, the *ToneObj* object stores the mute variable which has to be toggled for muting/unmuting, and the *TxRxObj* stores the state variables for enabling and disabling printing. When the user requests the musician to be muted and to activate printing, the application will schedule asynchronous tasks that toggle the above-mentioned state variables. Then, for printing periodically, a

printing task will be scheduled and then schedule itself with an offset equal to the printing periodicity. The rest of the functionalities in problem 1 are implemented in a similar fashion, with the reader function initiating tasks that update the related object state variables.

For this problem, we encountered quite many bugs. Even if the problem itself is not that difficult, since this was the first problem we implemented, we had a lot of initial bugs we needed to solve. Some bugs included not checking the return value of SYNC calls and directly sending the return value on CAN, which gave very strange behavior since the messages are sent as unsigned chars. We also had other issues with the CAN messages and the dynamic detection of the nodes. In the end, we feel that the resulting design works well and has a good structure.

Table 2: Group member contributions to problem 1

Name	**Contribution (%)**
Edvin	100
Tianqi	100
Kavineshver	100

4 Problem 2

Problem 2 aims to implement the functionality of musician boards to be able to claim conductorship. A musician board will claim conductorship by first requesting conductorship during some time. If no other request is detected from nodes with a lower node ID, it will go ahead and claim conductorship. The musician board claims conductorship by simply starting to transmit *Play* messages with an increased *Leader Number*.

When a musician board receives a keyboard command via *UART Interrupt* to apply for the role of conductor, it sets its state variable *reqConduct* to true and also transmits the request in the *Heartbeat* message. Then, a method named *claimConductor* will be scheduled with an offset of one second. In this method, the board will judge whether to claim conductorship based on the value of the state variable *reqConduct*. During the one-second waiting period, if the received *Heartbeats* message indicates that another node with a lower node ID also requests conductorship, our board will set its *reqConduct* variable to false. When one second has elapsed and the *claimConductor* task is executed, it will not claim conductorship since it will detect that the *reqConduct* has been set to false due to another request. However, if the *reqConduct* variable still is true, the board will go ahead and claim conductorship.

Therefore, when multiple musician boards apply to become the conductor within the waiting period, only the musician board with the lowest node ID will claim conductorship and increment the *Leader Number* in the transmitted messages. When the former conductor receives a message with an increased *Leader Number*, it will call a method to set its mode to musician. Additionally, to ensure that the new conductor resumes playing with the previous tempo, volume, and key, these values are always read by all musician boards when receiving a *Play* message, which allows the internal state variables of musician boards to remain updated. This enables a fast and smooth transition when switching conductors.

After addressing some minor bugs and initially implementing the basic functions, we encountered a problem: whenever we switch the conductor, there is sometimes a glitch or a short overlap in the music, which could be caused by several reasons. It might be due to the last *Tone Index* not being completely played before the switch, or possibly issues with the rate of *Heartbeats* message. Ultimately, we believe the root cause is likely due to the *Tone Index*. Thus, in the *claimConductor* method, we retrieve the current *Tone Index* and calculate the length of this particular tone based on the current tempo. By doing so, the board can calculate when the next note should begin to play and therefore only claim conductorship in between tones instead of in the middle of tones. This solved the issue and now the switching of conductorship sounds very good.

Table 3: Group member contributions to problem 2

Name	**Contribution (%)**
Edvin	100
Tianqi	100
Kavineshver	100

5 Problem 3

In order to add the ability to simulate silent failures of musician boards, there is a *FailureHandler* object with three different methods to handle the different failure modes as shown in Figure 1. Failure mode F1 is where a user manually activates the failure and also manually exits the failure mode. Failure mode F2 is when a user manually activates the failure, and then the board will automatically recover after a random interval of 5 to 10 seconds. Failure mode F3 occurs due to the physical disconnection of the CAN cable and will recover upon reinsertion of the CAN cable.

To simulate the silence failure, we use an empty receiver to replace the receiver method

with a CAN port. This will essentially mean that all received messages are discarded. Also when the board enters failure modes F1 and F2, the transmission of *Heartbeat* messages is disabled. For failure mode F3, since it is caused by the physical disconnection of the CAN connection line, there is no need for simulation. Instead, this mode is detected by monitoring the number of nodes in the network and if a board goes from detecting multiple nodes to suddenly being alone in the network, this probably is due to failure mode F3 and the board will set this failure mode as active.

Since silent failures cannot be declared directly through CAN messages, the *Node Detection* method is used to confirm the presence of members in the network and the status of each member. As explained for problem 1, the board will detect other nodes dynamically in the *Node Detection* method using the *Heartbeat* messages. This method updates the status of each board in the array of network nodes. When the conductor is requested to print the membership of the network, it will simply print the nodes present in the array and their state (Failed/Active).

In problem 3, we only handle the silence failures of musician boards, so when a musician board doesn't receive the *Play* messages from the conductor due to failure mode F1/F2/F3, the musician board will remain silent. On the other hand, the conductor node will always keep playing since we assume that the conductor never fails in this problem.

When a musician board exits silence failures and rejoins the network, it will start transmitting *Heartbeat* messages and wait for new *Play* messages. Therefore, the synchronization of a new musician board is entirely up to the current conductor, since the musician boards only listen to the *Play* messages and do not act on their own. When a musician joins back in the network, it will update its tempo, key, and volume state variables if these have changed during the failure period. Then the newly joined musician will simply wait to receive a *Play* message with a *Receiving Node* ID that matches its own ID, and then join in on the playing of the music. Therefore, it is easy to see that it is very much up to the conductor to make sure the synchronization of a new node goes smoothly.

When implementing the functionalities for problem 3, we faced almost no issues. This implementation step went very smoothly. However, when we tried to build on this part in problem 4, we started to get some issues and had to revise some parts of the design. These issues are explained in the section about problem 4.

Table 4: Group member contributions to problem 3

Name	Contribution (%)
Edvin	100
Tianqi	100
Kavineshver	100

6 Problem 4

The design of the software for problem 4 is built on the software for the previous problems, with some slight changes due to overlapping cases of problem 3 and problem 4. In problem 4, the conductor will now also be able to fail. In the case when there are only two boards left in the network and the communication between the boards is cut off, it is the musician that should continue to play in this problem, and the conductor should stay silent. This is the main difference compared to problem 3, where the conductor should always keep playing and the musician would be silenced in the same scenario.

The software design for problem 4 is very similar to problem 3, except for some added or updated functionality. The main differences are the conditions for entering failure mode F3, the scenarios for when the board should be silent or keep playing, and also the ability for a musician to claim conductorship. The failure modes F1, F2, and F3 have also been added when a board is in conductor mode as well.

When it comes to failure mode F3, in problem 4 it will be the conductor that leaves the network when there are only two nodes left. This means that we had to update the logic for entering this failure mode compared to problem 3. Now, each node will keep track of its mode (musician/conductor) at the time when there were two or more nodes left in the network. By doing this, each board can decide what to do when they suddenly become alone in the network. For example, if a board notices that it is alone in the network, but previously it was three or more nodes in the network, it means that it was the board itself that got disconnected and it should therefore stop playing. On the other hand, if it is alone in the network and the previous network size was two, then the board should either be silent if it is in conductor mode, or it should continue playing if it is in musician mode. The problem here is that the musician who now is alone will become conductor, and once the conductor is alone in the network it should be quiet since we can assume that it is the conductor who has failed in this problem. This is the reason why we need to keep track of the board's mode when there are two or more nodes left in the network; so that when the last musician becomes conductor and plays by itself, it will know that it has previously been a musician and that it should in fact keep playing.

Additionally, to solve the case of a conductor failing in the network, we need a new conductor to quickly be assigned so that the whole network can keep on playing the music. This is done based on the rank in the network. Each node will keep track of the current conductor node by tracking who is transmitting the *Play* messages and also the *Mode* of each node which is included in the *Heartbeat* message. As explained earlier, the rank is based on node ID and a lower ID gives a higher priority. This allows all the nodes to easily calculate who should become conductor if the current conductor fails, even without communicating with each other when the actual failure occurs. If the conductor fails, this will be noticed on the missing *Heartbeat* messages and the conductor node will be put into the Failed state in the board's array of network nodes. Then, each node will compare its own node ID with the other nodes that are still connected in the network. The board with the lowest node ID will directly start transmitting new *Play* messages from where the last conductor left off, with an increased *Leader Number*. All other nodes will detect the increased *Leader Number* and start listening to the newly elected conductor. Notice that there is no need for "requesting conductorship" here as is done when claiming conductorship by user input, since it should be clear to all nodes who the next conductor will be based on rank. When the old conductor joins the network again, it will detect the incremented *Leader Number* and become a musician. Even if an old conductor board would start transmitting *Play* messages when it joins back, these messages will be discarded by all the other boards since the *Leader Number* is outdated.

Overall, we are happy with the design of the software at this stage. There were some difficulties in combining the design due to the contradicting behavior in problem 3 and problem 4, but in the end, we were able to combine all functionalities into one software design by using macros to distinguish which of the problems we are currently running. This macro is called *PROBLEM4_ACTIVE* and will enable or disable the extra conditions and functionality explained above that only occur in problem 4. One difficult bug we faced was musicians who joined back into the network sometimes directly claimed conductorship even if there already was a conductor in the network. We solved this by adding a "recovering" period where the board that joins back after a failure starts transmitting and receiving *Heartbeat* messages for 600ms before it is allowed to claim conductorship. This made sure that the board detects any active nodes and if it in fact is alone in the network, it is allowed to claim conductorship.

Combining problem 3 and problem 4 was probably the hardest challenge of the whole lab, since we really wanted to have one unified software for all functionalities but it was very difficult to solve all possible cases in the same application. That's why we had to use the *PROBLEM4_ACTIVE* macro to distinguish between the different behaviors in problem 3 and problem 4 for the same scenarios. However, by facing these issues, it forced us

to iterate the design and in the end, we feel that the software quality was improved by having to solve these problems. Mainly we had to work on simplifying the design and the algorithms since it started to become overly complex, due to us just adding more and more ”special” conditions and not re-thinking the design. Therefore, when we eventually re-did the design we could improve it and also make it simpler and more coherent.

Table 5: Group member contributions to problem 4

Name	Contribution (%)
Edvin	100
Tianqi	100
Kavineshver	100

7 Problem 5

The objective of problem 5 is to design a regulator that controls the inter-arrival time between two subsequent CAN messages. Unlike the previous problem statements, this one is a non-collaborative part, where we don't require other boards. The board is set to loop-back mode, which enables it to receive the messages that it sends. The access graph of the implementation looks like this Figure 2. The object *Application* sends and receives CAN messages, it also gets the keyboard inputs and sends them to the *Regulator* where the data gets stored in a circular buffer and is sent out sequentially based on the set inter-arrival time.

The control flow of the program is as follows; on the reception of keyboard interrupt 'o', we send out a single CAN message with a message ID that increments for every new message sent. With 'b' as the keyboard interrupt, a new message gets transmitted with an updated message ID every 0.5 seconds. As the CAN loop-back mode is enabled, the message transmitted from CAN1 is received at CAN2 with the *receiver* function. The message ID is then pushed into the circular buffer. If the buffer is full then the received message ID is discarded i.e. not stored. When the empty buffer gets updated, then we call the *pull_buffer* method to empty the buffer. The data in the buffer gets pushed out only when the timer elapses the inter-arrival time. After a message gets pulled out we reset the timer. If the buffer is not empty after pulling out data, then the *pull_buffer* method schedules itself again with an offset equal to the inter-arrival time. In case the inter-arrival gets changed when the data gets printed out, for example, the initial inter-arrival was set to 1 sec and it gets changed to 5 sec when the buffer is not emptied, then we calculate the time

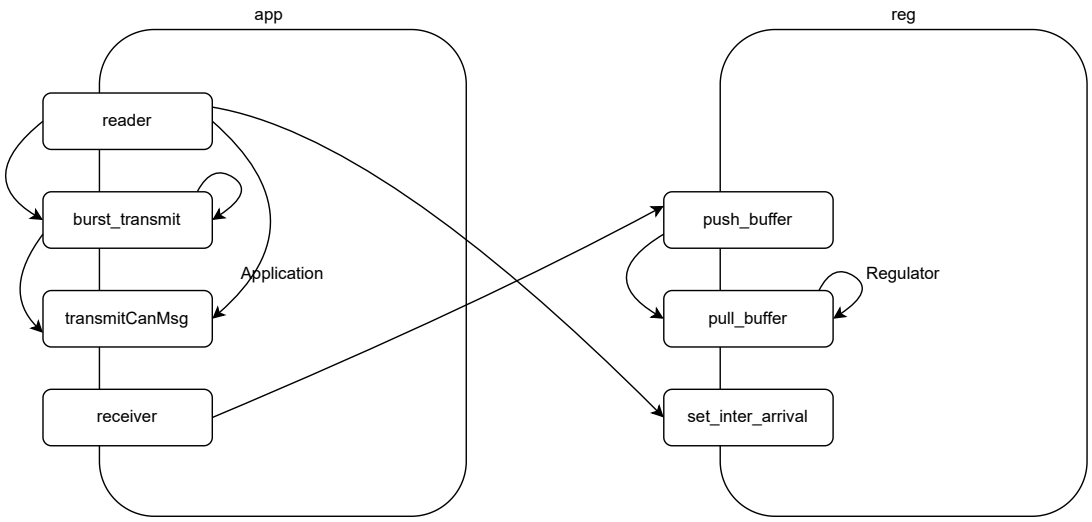

Figure 2: Access Graph

after which *pull_buffer* needs to be scheduled again. i.e. if inter-arrival time was changed from 1 sec to 5 sec, then *pull_buffer* gets scheduled again after 4 sec. We use two different timers, one to keep track of the time from the start of the program, and another to track the time at which the last message was pulled out of the buffer.

Delay calculation for the data buffered in regulator

The timing diagram of the design is depicted in Figure 3. Here the message is transmitted at an interval of every 0.5 seconds. If the buffer is not full, then the first message that's received at 0 seconds will be sent out at 0 seconds (in an ideal case without any delay or jitter). The second message received at 0.5 sec will be transmitted at the inter-arrival time, the third message received at 1 second will get transmitted at 2*inter-arrival time, and so on.
Let 'i' be the inter-arrival time and 'n' be the sequence number of messages.

*Delay of nth message = nth message * (inter-arrival time - transmission interval)*

If the transmission interval is 0.5 seconds then,

*Delay of nth message = n * (i-0.5)*

If the buffer is full then the received message is discarded.

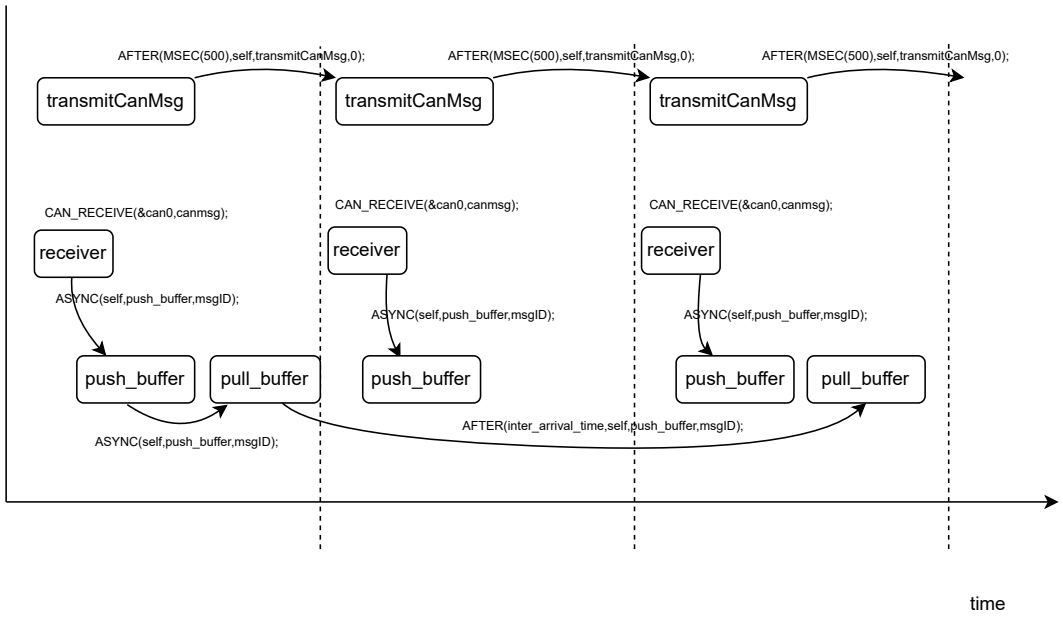

Figure 3: Timing Diagram

Space complexity of the regulator

typedef struct {
Object super;
uchar RxMsg[10];
uchar inter_arrival_time;
uchar head_index;
uchar tail_index;
}Regulator;

The snippet of code above is the data structure of the Regulator. The implemented circular buffer has a buffer capacity of 10, with *uchar* occupying 1 byte, the memory size of the regulator that is used to control the buffer can be represented as below.

Space complexity of Regulator = 3 + (Buffer_Size) Bytes

Time complexity of the regulator

Since the regulator uses a circular buffer for storing the received messages, the time complexity of operating this buffer is not very complex. Operations to handle the buffer only include moving the head and tail index as well as writing and reading to one position in the array at one time. All these operations give constant time complexity, or **O(1)** in big-O notation. If we had not used a circular buffer and instead shifted each message every time a new one was placed in the buffer, it would result in linear time operations. However, we choose to use the circular buffer due to its simplicity and lower time complexity.

Time complexity of Regulator = Constant time, ***O(1)***

Smallest possible inter-arrival time

For the current implementation with a buffer size of 10, the smallest possible inter-arrival time that can be set is **0** (in an ideal case). In practice, it is the time it takes the buffer to load and shoot out the data. If we also include delays and jitter, the minimum inter-arrival time might be around some microseconds. In our design, if the buffering capacity is increased to more than 32 with an inter-arrival time of 0, it may lead to a 'Panic! Empty pool' error. During this, the inter-arrival time has to be increased to the smallest inter-arrival time which would be around some microseconds.

Issues faced during problem 5

One of the main issues that we faced was the "PANIC!!! Empty pool" error, found the root cause of the problem to be ASYNC calling a function many times. In the previous version of the design, the *pull_buffer* method was called every time new data got pushed into the buffer, and each instance of *pull_buffer* method also called itself if the buffer was not empty, which leads to multiple instances of *pull_buffer* method running simultaneously. We arrested this issue by making sure only one instance of the method exists at a given time. It was made such that the *pull_buffer* function only gets called the first time a message is written to the empty buffer.

Table 6: Group member contributions to problem 5

Name	Contribution (%)
Edvin	100
Tianqi	100
Kavineshver	100

8 Conclusion

With problems 1 to 4 being the collaborative part, we experienced how important integration testing is. A board may work individually very well but can fail during integration with other boards. This means that there must be a clear understanding and consensus between groups for the whole system to work. Working this way in a separate team and then integrating the design with multiple other teams is not common in university, but was a really good learning experience for future work in the industry where we believe this way of working to be much more common.

We have explained some of the issues and bugs we faced for each of the problems. The biggest challenge overall was definitely to get a common understanding between all groups, since some scenarios in problem 3 and problem 4 were a bit tricky to understand. Once we had a common understanding within the meta group, the biggest implementation issue we faced in our design was the overlapping cases in problem 3 and problem 4. This overlap caused some logic that was developed for problem 4 to introduce bugs in the logic for problem 3 and vice versa, and it took some time and re-designing to be able to combine all cases in the same software version. We also tried to separate the software versions between the different problems, but this only increased the workload since if we found a bug we had to correct it in all the software versions. Therefore, we decided to try to combine problems 1-4 into one software version.

With problem 5 being the non-collaborative part, it was a bit more straightforward compared to previous problems. Here, we were able to implement a regulator that controls the rate at which the messages are delivered to the receiving task to avoid the occurrence of sporadic overruns.

Overall, the homework assignment as a whole was exciting although very challenging in some aspects. To summarize, we are content with our software designs and feel like we achieved a coherent and structured end product.