COSC473 Introduction to Computer Systems

Assignment Two

Task 1: Basic System

This system consists of two microbits as nodes (one sender and other receiver) and implements an OSI layer 2 (data link type) protocol. This is one of the 7 layers of OSI architecture model (see Figure 1). This layer is responsible for flow control and error checking of data send over the network. In addition, this layer consists of protocol which explain how the frames will be sent to the next device in the network. The protocol simulated in this task is called **Stop and Wait ARQ**.

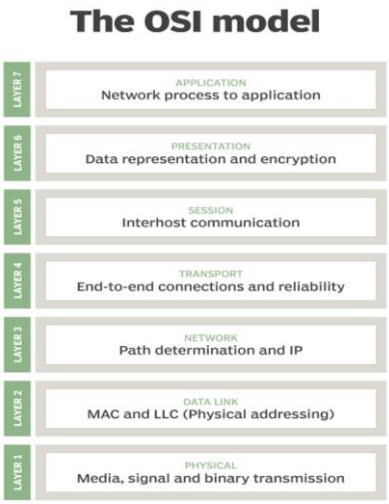

Figure 1: The OSI model (Rouse, 2014)

Stop and Wait ARQ is a protocol where the sender must stop after sending one frame and wait until it has received an acknowledge from the receiver. On the other side, whenever the receiver receives a packet from the sender, it must send an acknowledgement to the sender that the receiver has received a frame without any errors and ask for the next frame (GeeksforGeeks, 2017). See Figure 2 to see a visual representation of Stop and Wait ARQ.

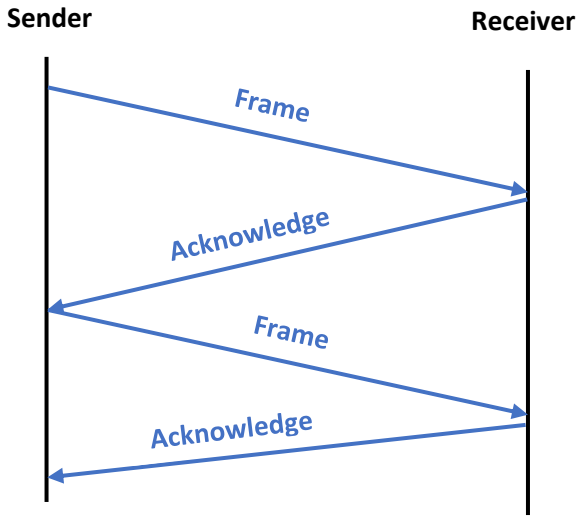

Figure 2: Stop and Wait protocol (GeeksforGeeks, 2017)

Clearly, it is evident that there are some issues with the current Stop and Wait protocol, it cannot handle if the frame or acknowledge is lost before reaching the destination, creating a deadlock. The solution to these issues is to a timeout mechanism, so that the sender resends the frame if it does not receive any acknowledges from the receiver in the given timeout period (Education 4u, 2018). For instance, if the timeout period is 2 seconds and the sender send a frame at t = 0 seconds. If the sender does not receive an acknowledge before t = 2 seconds, then the it will send the same frame again. This timeout mechanism considers for both a frame lost and an acknowledge lost as the receiver only sends an acknowledgment when it receives a correct frame. See Figure 3 to see a visual representation of Stop and Wait ARQ with timeout.

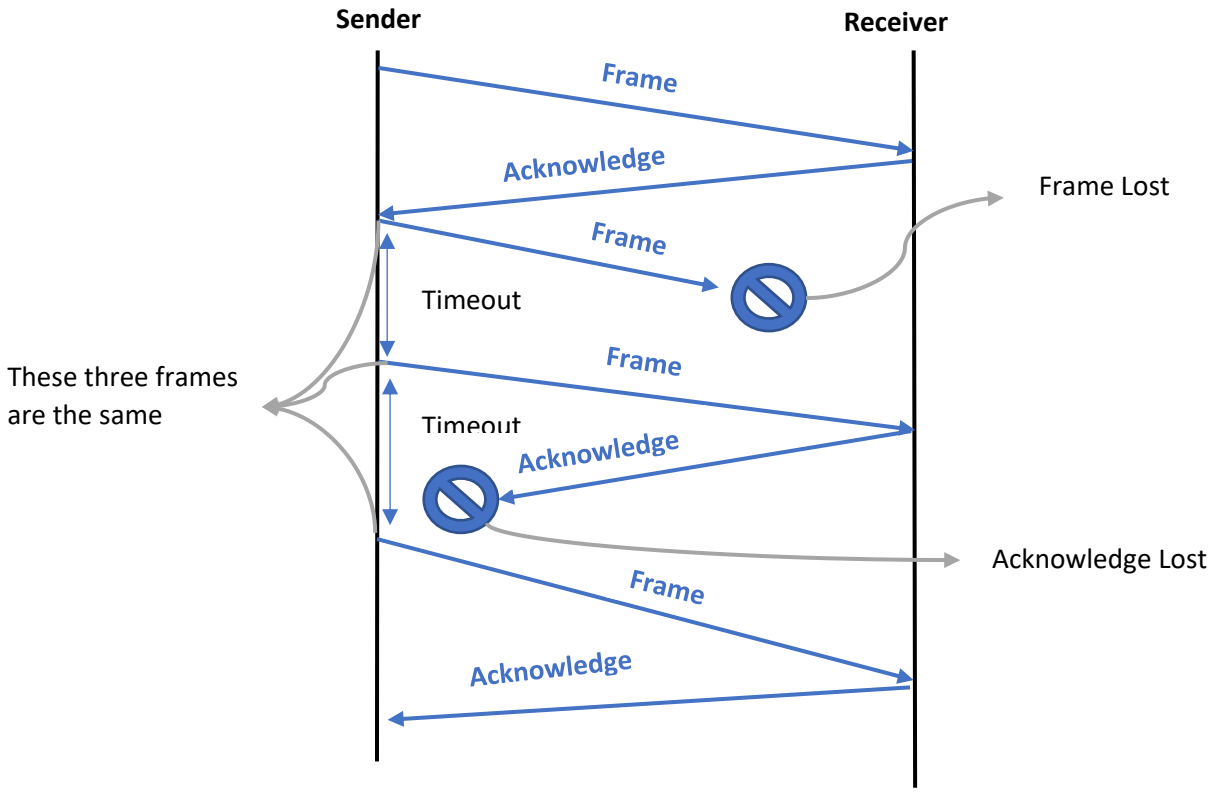

Figure 3: Stop and Wait protocol with timeout (GeeksforGeeks, 2017)

With the implementation of timeout, the protocol can send a frame again if needed. However, there is a possibility of either the frame or the acknowledge to be delayed and the microbits need to act accordingly. This is can done, by having sequence number of frames and acknowledgments. This is because if the frame is delayed, meaning the sender will not receive any acknowledgement and resends the frame after the timeout. The sequence number will allow the

receiver to identify a duplicate frame and allow the sender to identify a duplicate acknowledgment (Shashwat45, 2019). Since, the Stop and Wait protocol is sending only one frame at a time, it is ideal to have the sequence number of only one bit, i.e. 0 or 1. See Figure 4 to see a visual representation of Stop and Wait ARQ with timeout and sequence number.

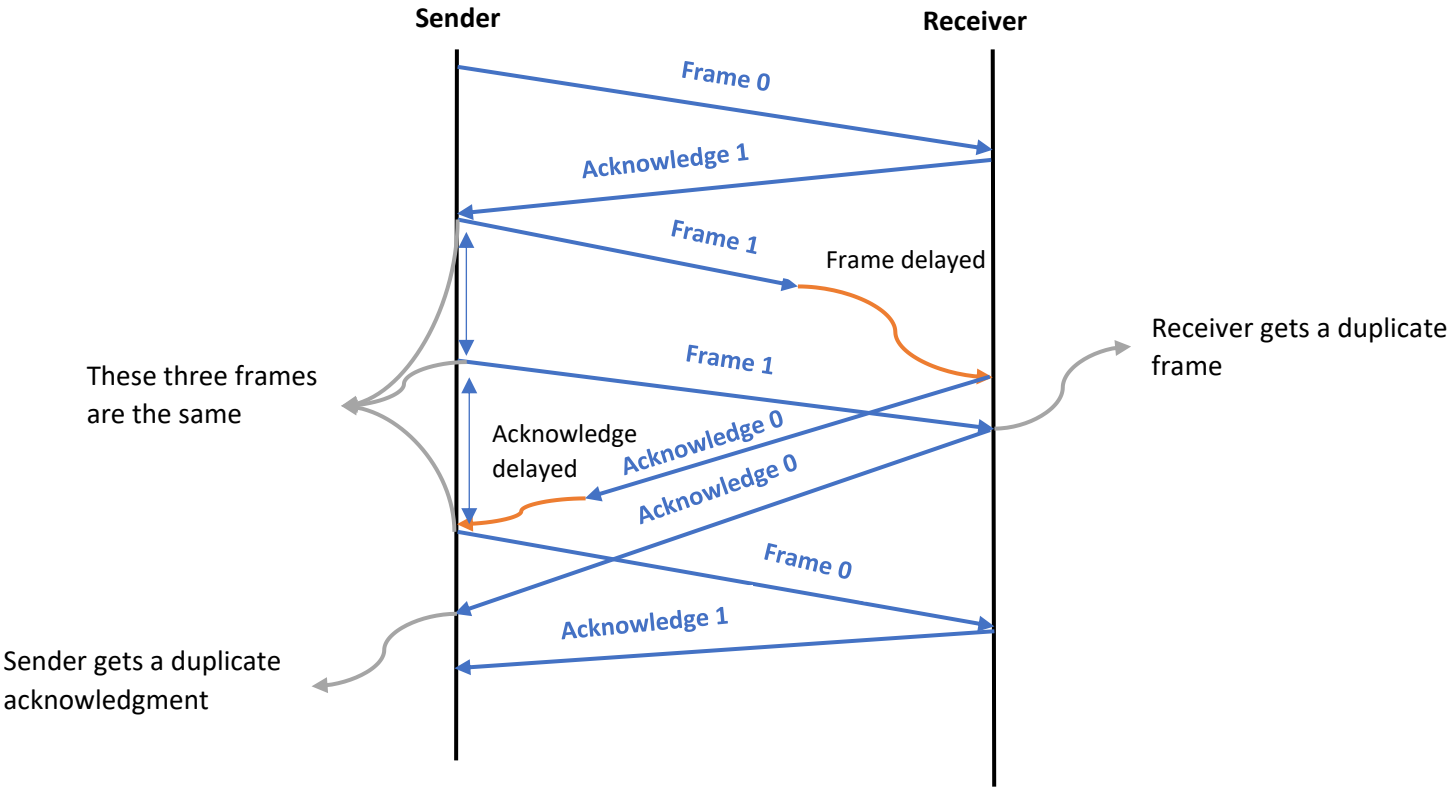

Figure 4: Stop and Wait protocol with timeout and sequence number (Shashwat45, 2019)

The receiver can identity a duplicate frame by checking the sequence number of the new frame and the previous frame. If the sequence number are the same, it is a duplicate frame and the receiver can ignore the contents of the frame. Similarly, the sender can identity a duplicate acknowledgment by checking the sequence number of the new acknowledgment and the previous acknowledgment. If the sequence number are the same, it is a duplicate acknowledgment and the sender can ignore acknowledgment.

Implementing Stop and Wait Protocol in microbits

Sender

As a sender, it is important to establish a connectivity test with a certain threshold to determine if the network connection is usable. For instance, let threshold be **0.9 (90%)**, and during the process, sender will calculate the frame loss rate. At any point in time during the transmission, if the frame loss rate is greater than or equal to the threshold, the send should determine the network is disconnected and will not send any more frames. The frame loss rate can be calculated using the formula:

$$frame\ loss\ rate = \frac{total\ frames\ lost}{total\ frames\ sent}$$

The sender will only stop sending more frames when the frame loss rate reaches the threshold or all the frames which are fragments of data are sent. This is can be done using a loop (see Figure 5):

```
#Variable
THRESHOLD = 0.9
frame_loss_rate = 0

frame_sent = 0
frame_lost = 0
frame_received = 0

#Data which is sent over the network. Each element in DATA is for one frame
DATA = ["1", "2", "3", "4", "5", "6", "7", "8", "9", "10", "11", "12", "13",
       "14", "15", "16", "17", "18", "19", "20"]

while True:  #Main loop

    #Some code here

    if frame_received == len(DATA): #Condition 1
        return

    frame_loss_rate = frame_lost / frame_sent #Condition 2
    if error_rate >= THRESHOLD:
        return
```

Figure 5: Code Snippet 1

The code above has the program's main while loop, that is the sender will continuously send frames and receive acknowledgments until the two conditions are meet. In addition, the counter variables ***frame_sent, frame_lost*** and ***frame_received*** are being updated somewhere in the loop.

As it is known that the sender will send a frame with the same sequence number as the acknowledgment's sequence number. Initially, the sender does not have reference to any sequence number, so it is necessary to set up a base case by sending one frame before entering the while loop (see Figure 6). The frame is a **String object** formatted as "(sender's address):(receiver's Address) (sequence number) (data)". For example, **"SD:RC 0 1"**, where **SD** is the sender's address, **RC** is the receiver's address, **0** is the sequence number and **1** is the data

```
#Addresses
SENDER_ADDRESS = "SD"
RECEIVER_ADDRESS = "RC"
ADDRESS = SENDER_ADDRESS + ":" + RECEIVER_ADDRESS

#Variables
frame_sent = 0

prev_seq_no = None
current_seq_no = 0

#Data which is sent over the network. Each element in DATA is for one frame
DATA = ["1", "2", "3", "4", "5", "6", "7", "8", "9", "10", "11", "12", "13",
       "14", "15", "16", "17", "18", "19", "20"]

#Base Case
frame_sent += 1
prev_seq_no = current_seq_no
send_frame(ADDRESS, current_seq_no, DATA[0])

while True: #Main Loop

    #Some code here
```

Figure 6: Code Snippet 2

Through a logical point of view, the program should send a frame and then update any related variables such as incrementing the frame sent variable by 1. However, it is important to

reduce time as much as possible between the sender sending a frame and waiting to receive an acknowledgment. This is because the frames between the microbits transfer very fast and if by any chance the sender takes some time to update variables and is not looking for an acknowledgment. The sender will miss the acknowledgment and must resend the frame. Therefore, the program is designed is designed to update any related variables. This methodology is applied both in Task 1 and Task 2.

After sending a frame over the network, the sender must wait until either an acknowledgment is received with a different sequence number compared to the sequence number sent in a frame or the time out period is reached. This can be implemented using a while loop and keeping a record of the sequence number of the previous frame sent (see Figure 7).

```
def get_ACK(address, prev_seq_no, start_time):
    TIMEOUT = 2000

    while True:
        ACK = radio.receive()

        #Checks if the acknowledge is from the correct receiver
        if ACK and ACK.startswith(address):
            index = ACK.find("ACK") + 4
            new_seq_no = int(ACK[index:])

            #Checks if the new sequence number is different
            if new_seq_no != prev_seq_no:
                return new_seq_no

        #Returns None is the acknowledge is not received within some timeout
        if running_time() - start_time >= TIMEOUT:
            return None
```

Figure 7: Code snippet 3

The **get_ACK** function is called in the main while loop in Figure 6. This is done to improve readability and looks cleaner. This function returns either the sequence number of the acknowledge (the function will ignore if the acknowledgment is duplicate) from the correct receiver or returns a **None** if the there is a timeout. Returning a None implies that the sender has not received an acknowledgment.

Now, if the sender does receive a sequence number from the **get_ACK** function and there are more frames to send, the sender must send the next frame with the sequence number received from the acknowledgment. On the other hand, if the sender does receive a **None** value, it must resend the same frame which was sent some moments ago. This can be done using an if statement (see Figure 8).

```
current_seq_no = get_ACK(ADDRESS, current_seq_no, start_time)

if current_seq_no is not None:
      frame_received += 1
      prev_seq_no = current_seq_no
      if frame_received < len(DATA):
          frame_sent += 1
          start_time = running_time()
          send_frame(ADDRESS, current_seq_no, DATA[frame_received])
else:
      frame_lost += 1
      frame_sent += 1
      start_time = running_time()
      send_frame(ADDRESS, prev_seq_no, DATA[frame_received])
```

Figure 8: Code snippet 4

Receiver

Compare with the sender, receiver has less functionality to perform. Receiver must continuously try to receive frames from the correct sender and send an acknowledgment back to the sender with the next sequence number (see Figure 9). The receiver must send an acknowledgment even if it receives a duplicate frame because there is a possibility of acknowledgment getting lost.

The acknowledge sent back to the sender is in a **String object**, formatted as "(sender's address):(receiver's Address) ACK (sequence number)". For instance, **"SD:RC ACK 1"**, where **SD** is the sender's address, **RC** is the receiver's address and **1** is the sequence number.

```
#Address
SENDER_ADDRESS = "SD"
RECEIVER_ADDRESS = "RC"
ADDRESS = SENDER_ADDRESS + ":" + RECEIVER_ADDRESS

#Variables
frame_received = 0

prev_seq_no = None
incoming_seq_no = None
next_seq_no = None

while True:

    frame = radio.receive()
    if frame and frame.startswith(ADDRESS):

        incoming_seq_no = int(frame[6:7])

        #Checks if the frame is not duplicate
        if prev_seq_no == None or prev_seq_no != incoming_seq_no:
             frame_received += 1

    next_seq_no = (incoming_seq_no + 1) % 2
    prev_seq_no = incoming_seq_no
    send_ACK(ADDRESS, next_seq_no)
```

Figure 9: Code snippet 5

Since this task does not require the receiver to store the data sent from the sender, so the receiver's program only updates the counter variable frame received. Otherwise, if receiver needs to collect the data from the frame and store it somewhere only if the frame is not duplicate.

Regarding showing LEDs, one LED implies one frame. The receiver will only turn on a LED if it receives a frame from the correct sender and it is not a duplicate frame.

Task 2: Improved System

This task still focuses on the OSI layer 2 (data link type layer). However, this requires implementing much more robust and complex prototype than the Stop and Wait ARQ. The protocol used in this task is called **Go Back-N Sliding Window ARQ**. This protocol can be seen as an advance version of Stop and Wait ARQ as it has the same concept of timeout, sequence number and acknowledge.

Go Back-N is a protocol where the sender can send multiple frames at once before waiting for an acknowledgment from the receiver (Sam, 2019). The number of frames sent at once depends on the window size and the window size can be anything greater than one. For example, lets the window size is 4, so the sender sends 4 frames at once. On the other hand, the receiver's act similar as in Stop and Wait, but only sends acknowledgment back to the sender up until in the order of the frame received. For instance, the sender sends frame 0, 1, 2, and 3, but the receiver gets only gets frame 0, 1 and 3. So, the receiver will send an acknowledgment for frames 0 and 1 and not for frame 2 and 3. As soon as the sender receives an acknowledge from the correct receiver, the sender sends the frame in the sequence, which will be frames 4 and 5. Even though the receiver receives frames 4 and 5, it does not send any acknowledgment back. This will make the sender resend the frames from 2 to 5 again after the timeout period. See Figure 10 to see a visual representation of Go Back-N sliding window ARQ.

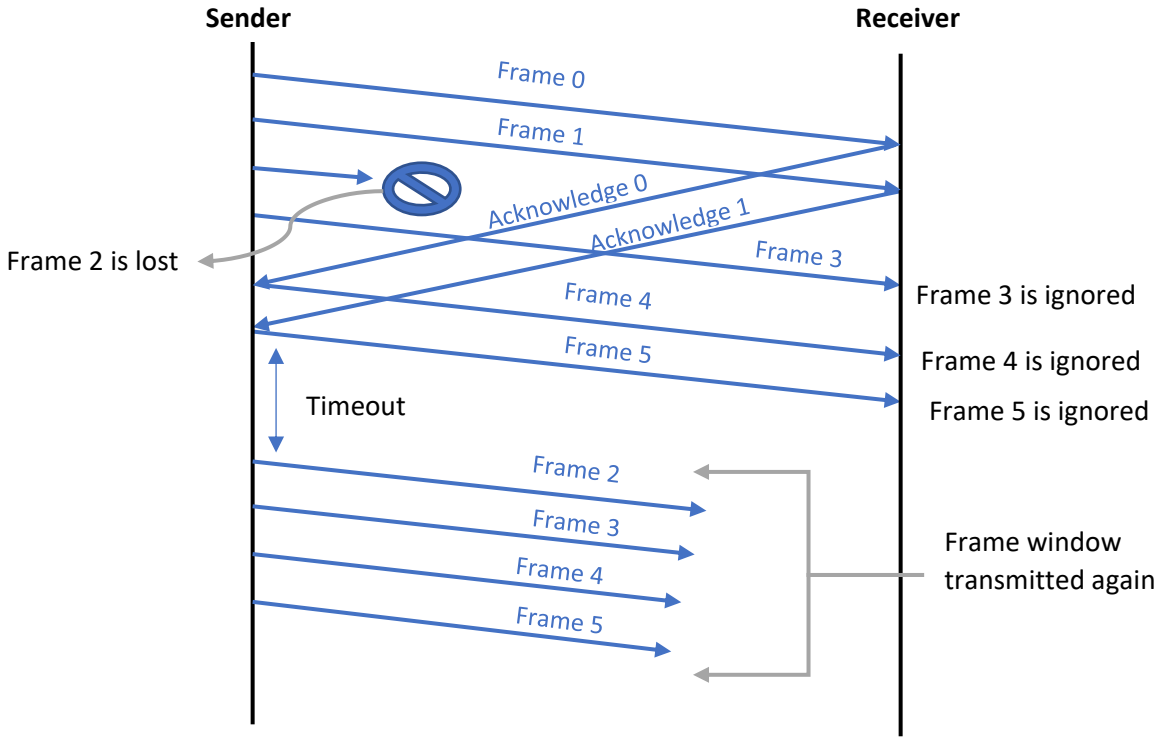

Figure 10: Go Back-N Sliding Window Protocol (GeeksForGeeks, 2017)

Unlike task 1, the use of the sequence numbers is slightly different. In Stop and Wait, the sequence numbers for both the frames and acknowledgments are using 1 bit, i.e. alternating 0 and 1 only. However, in Go Back-N, the size of sending window can be as long as possible, making it difficult to allocate specific number of bits to the sequence number. Therefore, the sequence number for frames the acknowledgments will start from 0 and continue sequentially till the number of frames.

Implementing Go Back-N Protocol in microbits

Sender

As this task requires the same condition for connectivity, the methodology to calculate the frame loss rate will be the same as task 1 and the sender will no more send any more frame once the frame loss rate reached the threshold.

Like Task 1, in this task, the sender will stop sending more frames if all frames in data is send or the frame loss rate reaches the threshold value. Hence, using a while loop and breaking out of it when the two conditions are met (see Figure 5). In addition, the format of the frames and the acknowledgments are the same as in Task 1.

As discussed before Task 1, the sender will send a new frame as soon as it gets an acknowledgment regarding the previous frame from the receiver. So, again it is important set up a base case before entering the while loop (see Figure 11). However, the base case will not send one frame, but the number of frames will be dependent on the sending window size. For example, if the sending window size is 4, the base case will send 4 frames at once.

```
#Address
SENDER_ADDRESS = "SD"
RECEIVER_ADDRESS = "RC"
ADDRESS = SENDER_ADDRESS + ":" + RECEIVER_ADDRESS

#Data which is sent over the network. Each element in DATA is for one frame
DATA = ["1", "2", "3", "4", "5", "6", "7", "8", "9", "10", "11", "12", "13",
        "14", "15", "16", "17", "18", "19", "20"]

#Variables
frame_sent = 0
window_start = 0
window_end = 3

#Base Case
frame_sent += (window_end - window_start) + 1
send_frames(ADDRESS, window_start, window_end, DATA)

while True: #Main Loop
      #Some Code here
```

Figure 11: Code snippet 6

From the code snippet above, there are two important variables for the sender's program, ***window_start*** and ***window_end***. The sending window in the program is represented as a list, ***DATA***. So, the **window_start** and **window_end** variables are starting and ending index of the sending window, respectively. This means if the **window_start** is 0 and **window_end** is 3, then the sending window frames are from index 0 to index 3 of ***DATA*** (from the first element to the fourth element of the ***DATA***). Thus, the total number of frames sent is:

$$frame\_sent = window\_end - window\_start + 1$$

After sending frames, the sender should wait for an acknowledgment until the timeout. The sender should also determine whether the acknowledgment is duplicate or not. This can be done by checking with the sequence number with the number of frames received. For example, if the sender has already received 3 acknowledgments and the sequence number for the incoming acknowledgment is also 3. The sender will determine this acknowledgment as not a duplicate as the it has received acknowledgments for frames 0, 1 and 2 and as the incoming acknowledgment's sequence number is 3, making it the fourth acknowledgment of the protocol because the sequence numbers starts from 0 and not from 1. Having a simple if statement while the sender is receiving and acknowledgment by comparing the frame received variable and new sequence number (see Figure 12).

```
def get_ACK(address, start_time, frame_received):
    TIMEOUT = 1500

    while True:
        ACK = radio.receive()

        if ACK and ACK.startswith(address): #Checking ACK from correct receiver
            index = ACK.find("ACK") + 4
            new_seq_no = int(ACK[index:])
            if frame_received == new_seq_no: #Checking for duplicate ACK
                return True

        if running_time() - start_time >= TIMEOUT:
            return False
```

Figure 12: Code snippet 7

The return values of **get_ACK()** function in Task 2 differs from the return value of **get_ACK()** fuction in Task 1. In this task, the function return either **True** or **False**, it will return true if sequence number matches with the frame received variable. Otherwise, it will return false only if there is a timeout.

One important thing to notice that the sender should not wait for all the acknowledgment, meaning if the senders sends multiple frames such as from frame 0 to frame 3. The sender must

only wait for an acknowledgment of frame 0. As soon as the sender receives an acknowledgment, it must send the next frame in the sequence which in this case is frame 4 and it should wait for an acknowledgment of frame 1. By receiving an acknowledgment of frame 1, the sender will send frame 5, and this will continue until all frames are send.

As **get_ACK()** function returning either true or false and the sender sends either one or multiple frames at once. The sender can make a correlation as when the function returns true, the sender should one frame, and if the function returns false, the sender should send multiple frames (see Figure 13). This will make the sender send only one frame in the next sequential order if the sender gets a correct acknowledgment from the receiver or resend frames from the start of sending window till the end of the sending window. This implementation will be present in the main while loop because the sender will continuously send either one or multiple frames.

```
ACK = get_ACK(ADDRESS, start_time, frame_received)

if ACK:
   window_start += 1
   window_end += 1
   frame_received += 1
   if window_end < length:
      frame_sent += 1
      start_time = running_time()
      send_frame(ADDRESS, window_end, DATA)
   else:
      if window_end >= length:
         window_end = length - 1
      frame_lost += 1
      frame_sent += (window_end - window_start) + 1
      start_time = running_time()
      send_frames(ADDRESS, window_start, window_end, DATA)
```

Figure 13: Code snippet 8

From the above code snippet, the function **send_frame()** will only send one frame and the function **send_frames()** will send multiple frames from window_start to window_end.

Receiver

The receiver's job is to just receive any frames send from the correct sender and given an acknowledgment back to the sender. The program for the receiver in this task is the same as the program for the receiver in Task 1. However, the only difference is that the receiver in this task is that the sequence number used in acknowledgment if the same sequence number which is received from the current frame. For instance, if the receiver receives frame 0, the acknowledgment sent will have the sequence number 0, and if it receives frame 1, it will be acknowledgment 1, and so on. If the receiver does not get a frame with new sequence number, it will determine that the frame is duplicate and resend an acknowledgment back to the send.

LED usage for Task 1 and 2:

For both tasks, and for the sender and the receiver, one LED implies one frame. For the sender, if an acknowledgment is received from the receiver, meaning the frame has also received successfully. So, one of the LED will be turned on with full brightness and if the no acknowledgment is received by the sender before the timeout period, one of the LED will be turned on with low brightness. As the microbit only has 25 LEDs to work with, if the microbit sends more than 25 frames, the program will loop over to the start of the LEDs and while clearing the next two LEDs

Examples of using LEDs:

- When all 20 frames are sent successfully (implement for task 1 and 2)
    - For sender, see Figure 14
    - For receiver, see Figure 15
- Sending frames until the frame loss rate reaches threshold
    - For Task 1, see Figure 16 (threshold is 90%)
    - For Task 2, see Figure 17 (threshold is 30%)

Figure 14: Sender all 20 packets

Figure 15: Receiver all 20 packets

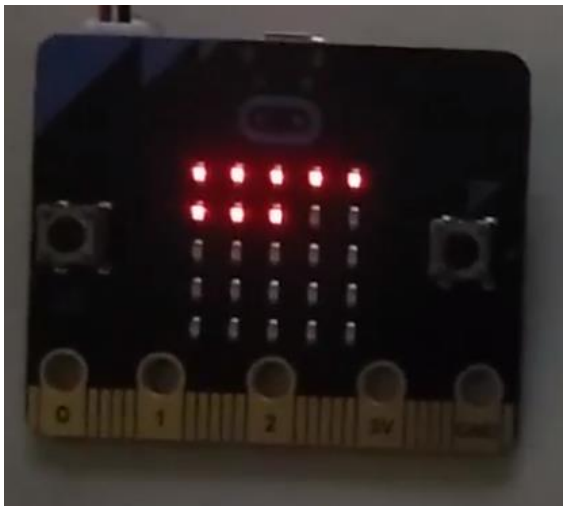

Figure 16: Sender frame lost (Task 1)

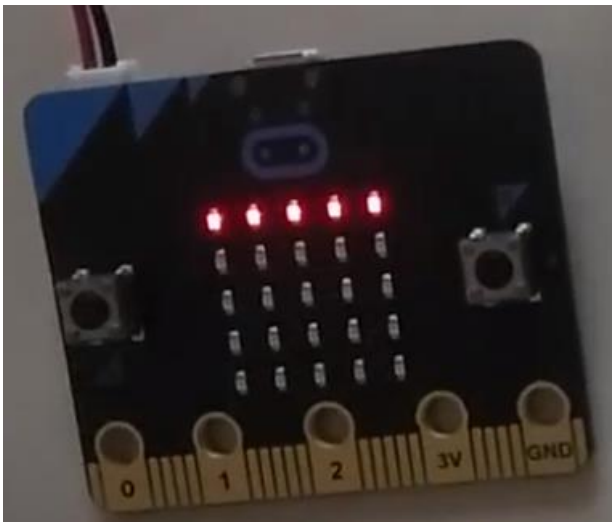

Figure 17: Sender frame lost (Task 2)

As the sender's program can end in two different ways: either all the frames are sent, or the frame loss rate reaches the threshold. The microbit will show a tick (see Figure 18) if all the frames are sent successfully or show a cross (see Figure 19) if the frame loss rate reaches the threshold. After a tick or a cross, it will also the number of frames (see Figure 20) lost during the protocol. For better understanding, refer to the video created.

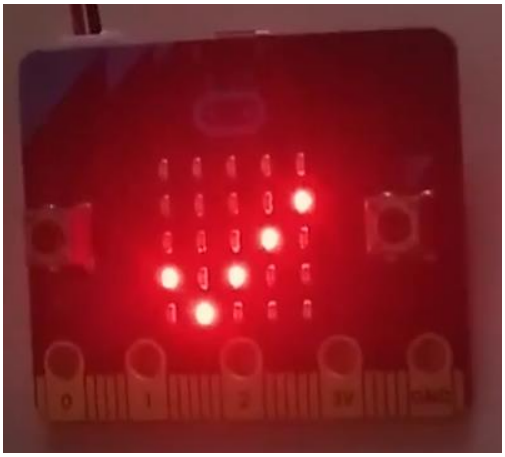

Figure 18: Sender tick

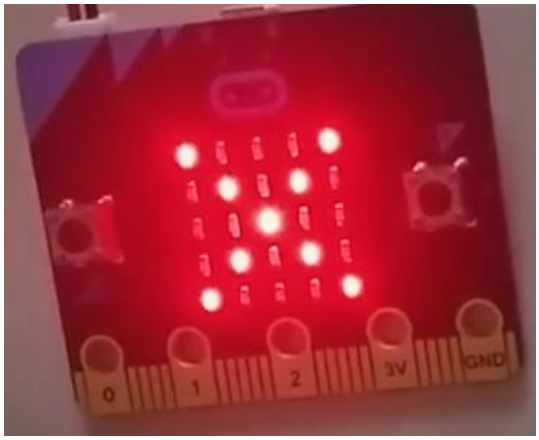

Figure 19: Sender cross

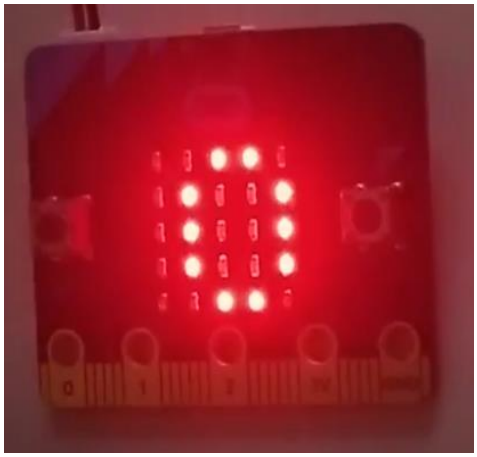

Figure 20: Sender frame lost

Please follow the step to run Tasks 1 and 2 on the microbits:

1. Download the sender's and receiver's microbits and wait till it gets downloaded
2. Press **Button A** on the receiver microbit
3. Press **Button A** on the sender microbit
4. Frames are now being transfer
5. If want to restart/rerun the protocol, press the **Reset Button** on both microbits and follow from step 2

Task 3: More Advance – Error Checking

Beside flow control, OSI layer 2's objective is to perform error checking on the frames being transferred over the network. This layer checks if the data received is whether a data error free or not. For the task, Even-parity SECDED error checking method has been implemented. SECDED code is one step advance version of Hamming error checking method; to understand SECDED code it is important to understand Hamming code. Hamming code involves inserting parity bits (parity bits are bits which does not represent the contents of the of data) between the bits of the data. The number of parity bits needed is calculated by the following equation:

$$2^p \geq m + p + 1$$

This is where **p** is the number of parity bits need and **m** is the size of the data in bits. For instance, if the data is 8 bits long, so the number of parity bits needed is 4 because $\mathbf{16 \geq 13.}$ So, the size of data sent over the network will be 12 bits long, not 8 bits long. After knowing the number of parity bits needed, the bit position of each parity bit needs to be calculated. To get the position of the parity bits, count the data bits from least significant bit (LSM) to the most significant bit (MSB) starting from 1, and the parity bits are at the position 1, 2, 4, 8, 18, 32, 64, etc. All powers of 2 (RMIT, n.d.). All other bits are data bits (see Figure 21).

Bit position	8	7	6	5	4	3	2	1
Data bits	D_8	D_7	D_6	D_5	D_4	D_3	D_2	D_1

Bit position	8	7	6	5	4	3	2	1
Parity bits	P_4				P_3		P_2	P_1

Bit position	12	11	10	9	8	7	6	5	4	3	2	1
Data/Parity bits	D_8	D_7	D_6	D_5	P_4	D_4	D_3	D_2	P_3	D_1	P_2	P_1

Figure 21: Data/Parity bits with bit position

The Hamming code works by using each check bit as a parity bit for a specific group of bits, as follows (RMIT, n.d.):

- **P_1** covers bits position 1, 3, 5, 7, 9, 11, 13, 15, etc.
    - Covers positions where there is a 1 in the first binary bit of the position number binary expression 000**1**, 001**1**, 010**1**, ...
    - Covers one bit and skips one bit
- **P_2** covers bits position 2, 3, 6, 7, 10, 11, 14, 15, 18, 19, etc.
    - Covers positions where there is 1 in the second binary bit of the position number binary expression: 00**1**0, 00**1**1, 01**1**0, 00**1**1, ...
    - Covers two bits and skips two bits
- **P_3** covers bits position 4, 5, 6, 7, 12, 13, 14, 15, etc.
    - Covers positions where there is a 1 in the third binary bit of the position number binary expression: 0**1**00, 0**1**01, 0**1**10, 0**1**11, ...
    - Covers four bits and skips four bits
- **P_4** covers bits position 8, 9, 10, 11, 12, 13, 14, 15, etc.
    - Covers positions where there is a 1 in the fourth binary bit of the position number expression: **1**000, **1**001, **1**010, **1**011, **1**100, **1**101, **1**110, **1**111
    - Covers eight bits and skips eight bits

When doing the parity checking, there is either Even-parity or Odd-parity. Even-parity means that the parity bit is reset to 0 if the number of 1s checked by that parity bit is even or set to 1 if the number of 1s checked by that parity bit is odd. Odd-parity means that the parity bit is reset to 0 of the number of 1s checked by the at parity bit is odd or set to 1 if the number of 1s checked by that parity bit is even. For example, lets take the data, 1010 (see Figure 22):

	D_4	D_3	D_2	P_3	D_1	P_2	P_1
Bit position	7	6	5	4	3	2	1
Data with parity bits	1	0	1		0		
P_1	1		1		0		0
P_2	1	0			0	1	
P_3	1	0	1	0			

Figure 22: Parity bit checking

Therefore, the fine hamming code for the data, 1010, is 1010010. Hamming codes are only able to identify and correct one-bit errors, but not able to identify if there are more than two-bit errors. Therefore, SECDED code has been introduced, it can identify if there is two-bit error. However, it is not able to correct as it cannot specify at which bit position the errors has occurred (RMIT, n.d.).

To use SECDED code, a new parity bit is added to the Hamming code. This parity bit is called P_0. This parity bit checks all the bit position including all other parity bits and data bits. For instance, the even-parity SECDED code for 1010 will be 10100101 as P_0 needs to be 1 to make the number of 1s even.

To check if the SECDED code has errors, it is necessary to check all the parity bits correct set or reset depending on the even and odd parity scheme. The following cases will imply if the code has errors:

- If both hamming parity bits and SECDED parity bit claims there is not error. This means that there is no error in the code
- If hamming parity bits claims there is an error and the SECDED parity bit claims there is not error. This means that there is at least two-bit error in the code and cannot be corrected
- If SECDED parity bits claim there is an error. This means that there is one-bit error and it can be corrected.

Implementing in microbits

Sender

The sender microbit can send three different type of messages, error free message, one-bit error message and two-bit error message. On the other side, the receiver receives any one of the types of message and will report if the message has no errors, one-bit error with correction or has more than two-bit errors.

To calculate the number of Hamming parity bits to use in done by the following code, where the variable name **length** is the length of the data in bits. The parity bits are being increased until the condition is satisfied (see Figure 23).

```
def calculate_number_of_parity_bits(length):
    total_parity_bits = 0
    while True:
        if (2 ** total_parity_bits) >= (length + total_parity_bits + 1):
            break
        total_parity_bits += 1
    return total_parity_bits
```

Figure 23: Code snippet 9

As the position of the parities are all on the power of 2s, 1, 2, 4, 8, etc., the sender's program will insert a placeholder for the parity bits between the data bits and will store the whole code into a list, making it easier to travers through the data (see Figure 24). The following code inserts a placeholder for the parity bits at 2 to the powers position, otherwise the data bit is added.

```
def insert_parity_bits(length, parity_bits, data):
    current_parity = 0
    data_index = 0
    data_with_parity = ['0'] #Place holder for parity 0

    for parity in range(1, length + parity_bits + 1):

        if parity == (2 ** current_parity):
            data_with_parity.append('-1') #Place holder for all parity bit
            current_parity += 1
        else:
            data_with_parity.append(data[data_index])
            data_index += 1

    return data_with_parity
```

Figure 24: Code snippet 10

After knowing the amount parity bits needed and their position, the next step is to calculate the what are the values of each parity bit. So, for each parity bit, starting from P_1, it needs to check some bits and skip some bits, and this depends on the parity number. For instance, P_2 checks two bits (consider this as one segment) and skips two bits. So, the sender needs to loop through every segment in the data and in each segment, the sender needs to loop through every bit (see Figure 25). While looping through the bits, the sender must keep count of the occurrence of the 1s to either set or reset parity bit to make an even-parity error checking scheme.

```
def set_hamming_code(total_parity_bits, data):

    #Looping as many times as the the number of parity bits starting from P1
    for parity in range(total_parity_bits):
        occurence_of_1 = 0

        #Looping through every segment of the data
        for data_index in range((2 ** parity), len(data), 2 ** (parity + 1)):

            #Looping through every bit in the segment.
            for bits_to_check in range(data_index, data_index + (2 ** parity)):
                if bits_to_check < len(data) and data[bits_to_check] == '1':
                    occurence_of_1 += 1

        #Set to 1, it the occurence of 1s are odd
        if occurence_of_1 % 2 != 0:
            data[(2**parity)] = '1'
    return data
```

Figure 25: Code snippet 11

After performing Hamming code, the sender will perform SECDED code. For this, the sender will go through all the bits and count the occurrence of 1s in the data and it will set 1 if the number of 1s are odd (see Figure 26).

```
def set_secded_code(data):
    occurence_of_1 = 0

    #Loop through all the bits
    for bit in data:
        if bit == '1':
            occurence_of_1 += 1

    #Set to 1, it the occurence of 1s are odd
    if occurence_of_1 % 2 != 0:
        data[0] = '1'

    return data
```

Figure 26: Code snippet 12

At this point SECDED code has been implemented. Moreover, when the button a of the sender's microbit is pressed, a random bit in the data is flipped. This is because to simulate error checking with some errors.

Receiver

Like the sender, the receiver will need loop through that data received from the sender. That is, the receiver must loop through all the segments of the data and loop through all the bits in each segment. However, the receiver does not know what the number of occurrences of 1s will be while it is still counting for the number of 1s. For this reason, the receiver's program will how two parallel list, **lst1** and **lst2,** lst1 will become the real list if the occurrence if 1s are even or lst2 will become the real list as the occurrence of 1s are odd (see Figure 27).

```
def check_hamming_code(lst, parity, data):
    occurence_of_1 = 0

    #list if the parity bit is correct
    lst1 = lst[:]

    #list if the parity bit is not correct
    lst2 = lst[:]

    #Looping through every segment of the data
    for data_index in range((2 ** parity), len(data), 2 ** (parity + 1)):

        #Looping through every bit in the segment.
        for bits_to_check in range(data_index, data_index+(2 ** parity)):
            if bits_to_check >= len(data):
                break
            if data[bits_to_check] == '1':
                occurence_of_1 += 1
            lst1[bits_to_check] = None
            if lst2[bits_to_check] is not None:
                lst2[bits_to_check] += 1

    if occurence_of_1 % 2 == 0:
        return lst1
    else:
        return lst2
```

Figure 27: Code snippet 13

The above checks for each hamming parity bit and that function is called as many times as the number of hamming parity bits. After checking parity bits, the receiver will check for the SECDED parity bit by iterating through all the bits in the data and count the number of 1s. After

checking for the SECDED parity bit, the receiver will determine if data is either error free, one-bit correctable error or more than that (see Figure 28).

```
    highest = -1
    pos = None
    highest_count = 0
    for parity in range(len(lst)):
        if lst[parity] is not None and lst[parity] > highest:
            high = lst[parity]
            pos = parity
            highest_count = 0
        elif lst[parity] == highest:
            highest_count = None

    #2 bit or more bit error
    if highest_count is None:
        return
    #Error free
    elif pos is None:
        return
    #2 bit or more bit error
    elif lst[0] is None:
        return
    #1 bit correcable error
    else:
        message = list(data)
        message[pos] = str(((int(message[pos])) + 1) % 2)
        message = "".join(message)
        return
```

Figure 28: Code snippet 14

Examples of using LEDs:

The sender microbit will show a right arrow whenever it sends message to the receiver (see Figure 29). The receiver microbit will show a tick if the message is error free (see Figure 30), a surprise image (see Figure 31) if it is one-bit error or a confused image (see Figure 32) if there is two or more errors

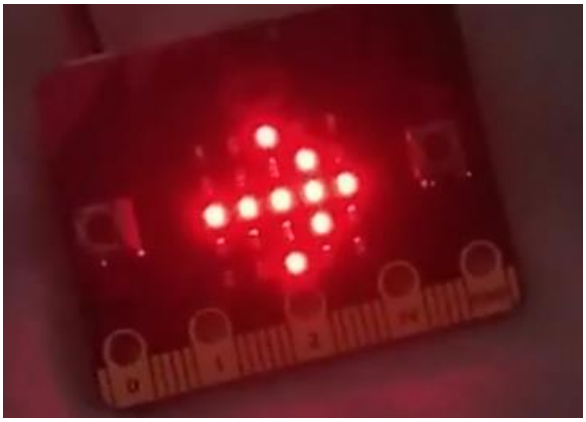

Figure 29: Sender right arrow

Figure 30: Receiver error free message

Figure 31: Receiver 1-bit error message

Figure 32: Receiver 2 or more bit error

Please follow the step to run Tasks 3 on the microbits:

1. Download the sender's and receiver's microbits and wait till it gets downloaded
2. Press **Button B** on the sender microbit (press 0 times for error free message, press 1 time for 1-bit error message or press 2 times for two or more-bit error message)
3. Press **Button A** on the sender microbit to send the message
4. Repeat steps 2 and 3 for rerun

Task 4: Most Advance – ARP

This task involves implementing Address Resolution Protocol (ARP), where it is used resolve for the Internet Protocol (IP) addresses to Media Access Control (MAC) addresses. MAC address is also known as the device's physical address (GeeksForGeeks, n.d.). With this information, the sender can only identify which network the receiver belongs to, but cannot identify the physical device in the network (PowerCert Animated Videos, 2018).This protocol comes in play whenever a computer (sender) wants to communicate with another computer (receiver), it needs to have the receivers MAC address. For example, the sender wants to communicate with the receiver but only knows the receiver's IP (2.2.2.2) address. Initially, the sender will investigate its internal ARP cache (table) for the receiver's MAC address (PowerCert Animated Videos, 2018). If the sender does not find a MAC address, it will send a broadcast message requesting for the device's MAC address if that device's IP address 2.2.2.2.

On the other side, the receiver gets a request and sends its MAC address back to the sender. When the sender receiver the MAC address from the receiver, it will update its internal APR cache by linking the receiver's IP address to the receiver's MAC address. Now, the sender and the receiver can communicate. This is why ARP cache makes the network more efficient by storing IP address to MAC address associations (PowerCert Animated Videos, 2018). There are two types of ARP entries, **static** and **dynamic**, static entries must be stored by typing it manually via the command line utility. Whereas, dynamic entries are stored automatically whenever by requesting for the device's mac address. However, compared to static entries, dynamic entries are not permanent; they are removed periodically (PowerCert Animated Videos, 2018).

Implementing ARP in microbits

Sender

When the sender's program starts, it will first investigate its ARP cache. If the sender finds in its cache, it will display the receiver's IP and MAC addresses. If the sender does not find a MAC addressed associated with the receiver's IP address, it will broadcast a request for the MAC addresses in the network (see Figure 33).

```
def start(ARP_cache):
    sender_IP_address = "1.1.1.1"
    receiver_IP_address = "2.2.2.2"

    while True:

        #Looks in the ARP cache, if the MAC Address is not there, then it will
          request and add in the table or look it up in the table and will
          display it.
        if ARP_cache.get(receiver_IP_address) is None:

            #broadcasts a request and waits till it gets the MAC address.
            MAC = get_MAC(sender_IP_address, receiver_IP_address)
            ARP_cache[receiver_IP_address] = [MAC, "dynamic"]
        else:
            display.scroll(receiver_IP_address)
            display.scroll(ARP_cache.get(receiver_IP_address)[0])
            return ARP_cache
```

Figure 33: Code snippet 15

The ARP cache is structured as a dictionary, where the key is the receiver's IP address and the value is a list containing the MAC address and the entry type. In addition to this, the sender's program reformats the MAC address given by the user to reflect the format used in the real world. In real world, a MAC address is 8 digit long hexadecimal number. However, the unique for microbits are only 6 digit long hexadecimal number.

Receiver

In the receiver's program, the receiver is constantly listening for the requestion from the sender and as the is receives a requestion, it will return it's MAC address back to the sender. As the microbits do have a MAC address, an alternative unique number system is been used. With some research, each microbits have a unique **serial ID**. However, it is not purely unique but to find a microbit with same serial ID, one needs to collect around 100,000 to 250,000 microbits (Marklund, 2019). To get the microbit's serial ID, a special function is called (see Figure 34), this function is Karl Marklund's work.

```
#This function return the unique serial ID of the microbit
def get_serial_number(type=hex):
    NRF_FICR_BASE = 0x10000000
    DEVICEID_INDEX = 25

    @micropython.asm_thumb
    def reg_read(r0):
        ldr(r0, [r0, 0])
    return type(reg_read(NRF_FICR_BASE + (DEVICEID_INDEX*4)))
```

Figure 34: Code snippet 16

Please follow the step to run Tasks 4 on the microbits:

1. Download the sender's and receiver's microbits and wait till it gets downloaded
2. Press **Button A** on the receiver microbit
3. Press **Button A** on the sender microbit
4. If want to restart/rerun the protocol, press the **Reset Button** on both microbits and follow from step 2

LINK TO THE VIDEO

https://rmiteduau-my.sharepoint.com/:v:/g/personal/s3825891_student_rmit_edu_au/ES3YvM9AnvFAkUj6M-QAIH4BHlq_1GfZ3thjJBdm3dapdw?email=ying.zhang3%40rmit.edu.au&e=FfvOdd

References

Education 4u, 2018. *stop and wait arq | data communication | Bhanu priya.* [Online]
Available at: https://www.youtube.com/watch?v=KPWjG45TDTM
[Accessed 17 5 2020].

GeeksForGeeks, 2017. *Sliding Window Protocol | Set 2 (Receiver Side).* [Online]
Available at: https://www.geeksforgeeks.org/sliding-window-protocol-set-2-receiver-side/
[Accessed 22 5 2020].

GeeksforGeeks, 2017. *Stop and Wait ARQ.* [Online]
Available at: https://www.geeksforgeeks.org/stop-and-wait-arq/
[Accessed 16 5 2020].

GeeksForGeeks, n.d. *Introduction of MAC Address in Computer Network.* [Online]
Available at: https://www.geeksforgeeks.org/introduction-of-mac-address-in-computer-network/
[Accessed 28 5 2020].

Marklund, K., 2019. *How to read the device serial number.* [Online]
Available at: https://support.microbit.org/support/solutions/articles/19000070728-how-to-read-the-device-serial-number
[Accessed 28 5 2020].

PowerCert Animated Videos, 2018. *ARP Explained - Address Resolution Protocol.* [Online]
Available at: https://www.youtube.com/watch?v=cn8Zxh9bPio
[Accessed 28 5 2020].

RMIT, n.d. *Chapter 5: Memory & Hamming/SECDED Codes.* [Online]
Available at:
https://www.dlsweb.rmit.edu.au/set/Courses/Content/CSIT/oua/cpt160/2014sp4/chapter/05/CodingSchemes.html
[Accessed 25 5 2020].

Rouse, M., 2014. *OSI model (Open Systems Interconnection).* [Online]
Available at: https://searchnetworking.techtarget.com/definition/OSI
[Accessed 16 5 2020].

Sam, S., 2019. *Sliding Window Protocol.* [Online]
Available at: https://www.tutorialspoint.com/sliding-window-protocol
[Accessed 20 5 2020].

Shashwat45, 2019. *Flow Control - STOP & WAIT and STOP & WAIT ARQ Protocol.* [Online]
Available at: https://www.studytonight.com/post/flow-control-stop-wait-and-stop-wait-arq-protocol
[Accessed 18 5 2020].