CS4222 Project

GROUP: 6

NAMES:

Name	Student Number
Elvis Teo Chin Hao	A0218206M
Zhuang Jianning	A0214561M
Zhao Haoxiang	A0206128M
Wong Yun Rui Chris	A0217544E

CONSIDERATIONS FOR BONUS:

- Extra data analysis of packet reception intervals in Task 1, calculating things like mean and standard deviation and performing analysis with them, instead of only relying on CDF
- Many extra combinations of different settings tested for Task 1 with detailed analysis and comparison for each of them
- Detailed explanation of how our deterministic discovery protocol works with diagrams included
- Testing of stability of different parameters for our deterministic discovery protocol
- Testing of sensortag with obstacles in between to check if there is change in performance
- Testing of performance of deterministic discovery protocol using packet reception interval
- Possible applications of our Project in the real world. (Neighbour Discovery Protocol)

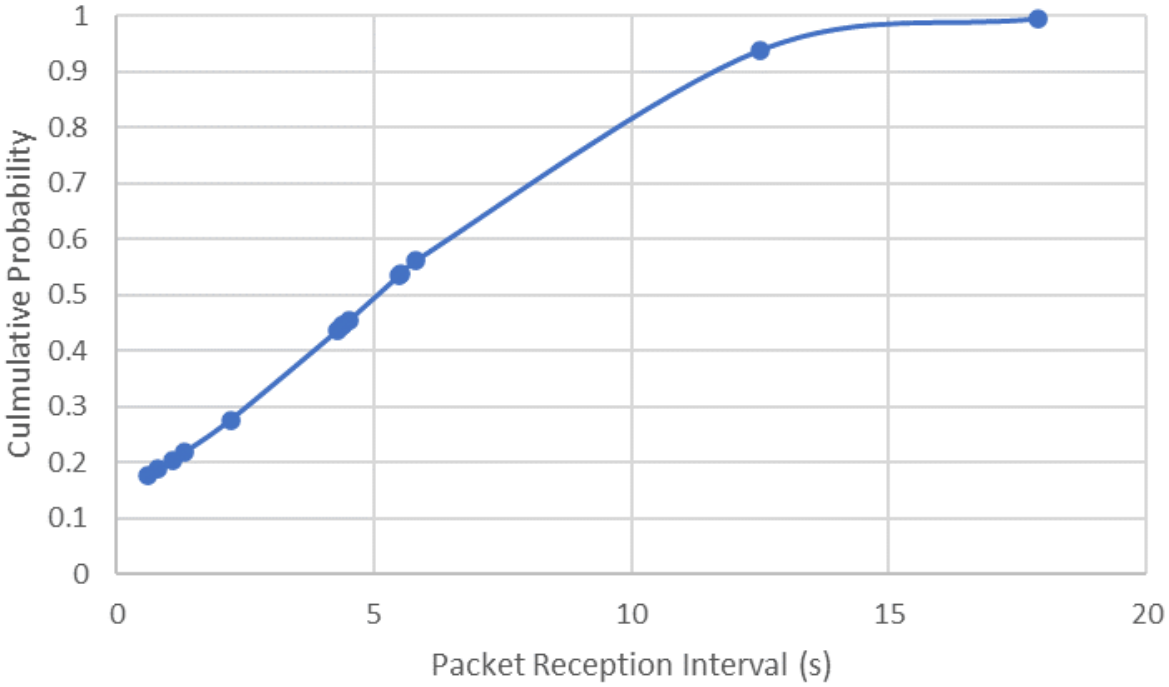
TASK 1:

Take 2 sensortags and run nbr_original to collect data

[1] Intervals of packet receptions on A from B for around 15 receptions

Mean Packet Interval Time (s): 5.06
Standard Deviation of Packet Interval Time (s): 4.79

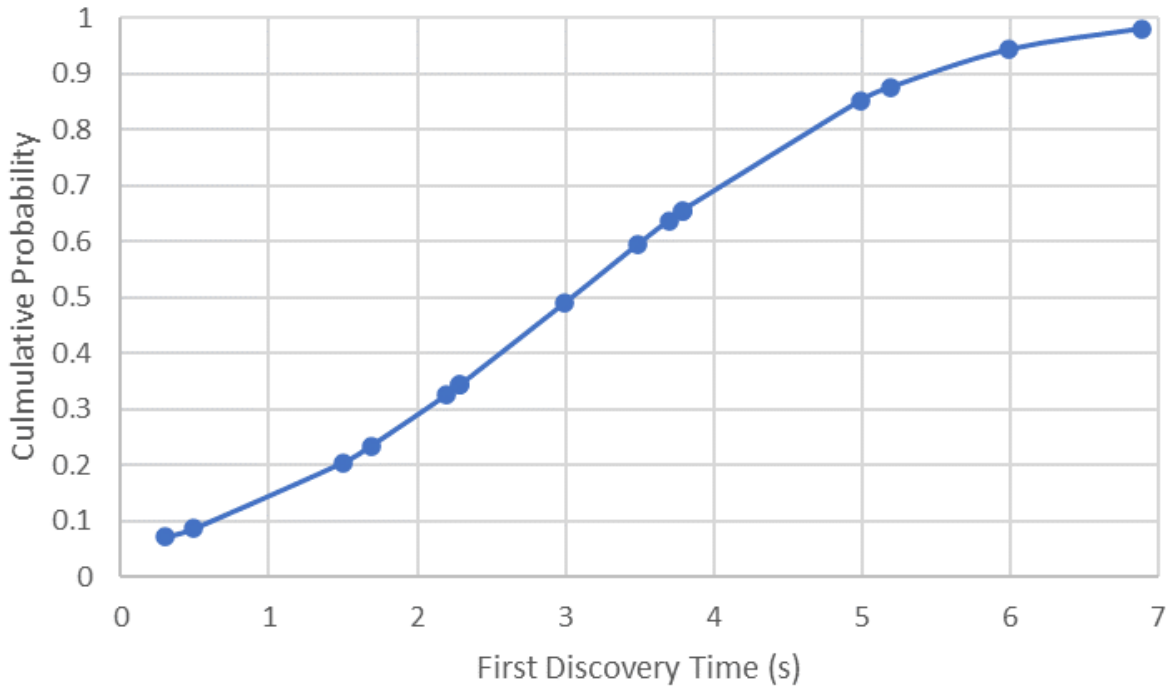
No.	Time of packet reception (s)	Interval between packet reception (s)
1	2.468	4.297
2	6.765	5.805
3	12.570	4.398
4	16.968	2.203
5	19.171	5.493
6	24.664	17.898
7	42.562	0.805
8	43.367	12.5
9	55.867	5.5
10	61.367	1.093
11	62.460	0.602
12	63.062	4.5
13	67.562	4.398
14	71.960	1.305
15	73.265	


Culmulative Probability
1
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
0
5
10
15
20
Packet Reception Interval (s)

[2] First discovery time that A hears from B. (Time 1st packet is received) - Reboot B 10 times

Mean First Discovery Time (s): 5.49
Standard Deviation of First Discovery Time (s): 3.42

No.	First Discovery Time	No.	First Discovery Time
1	6.789	11	4.195
2	6.39	12	2.695
3	4.9	13	11.992
4	4.492	14	4.89
5	4.695	15	4.695
6	15.695	16	5.195
7	3.093	17	7.789
8	4.289	18	6.289
9	0.296	19	3.695
10	1.39	20	6.289



[3] Other combinations of Settings

Parameters Chosen for Testing:

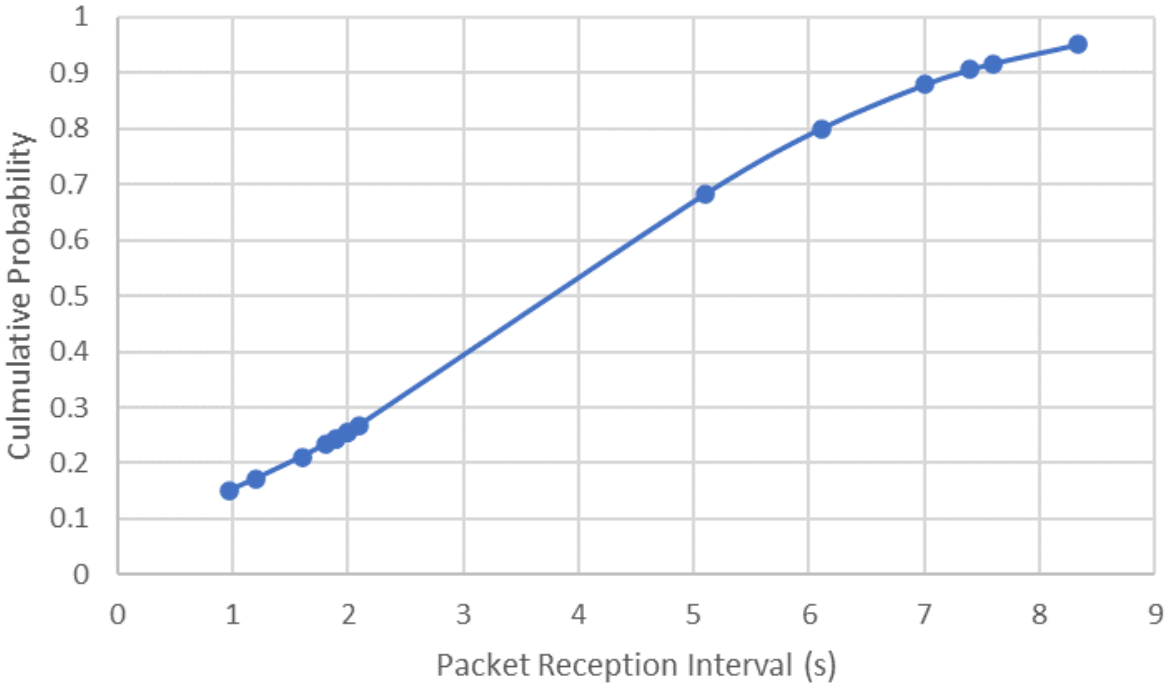
No.	Name	Wake Time (s)	Sleep Time (s)	Sleep Cycle	Ave Duty Cycle (%)
1	Original	0.1	0.1	9	10.00
2	Increased wake time	0.2	0.1	9	18.19
3	Increased sleep time	0.1	0.2	9	5.26
4	Lower sleep cycle	0.1	0.1	5	16.67
5	Lower wake and sleep time	0.05	0.05	9	10.00
6	Slightly higher sleep time	0.1	0.125	9	8.16

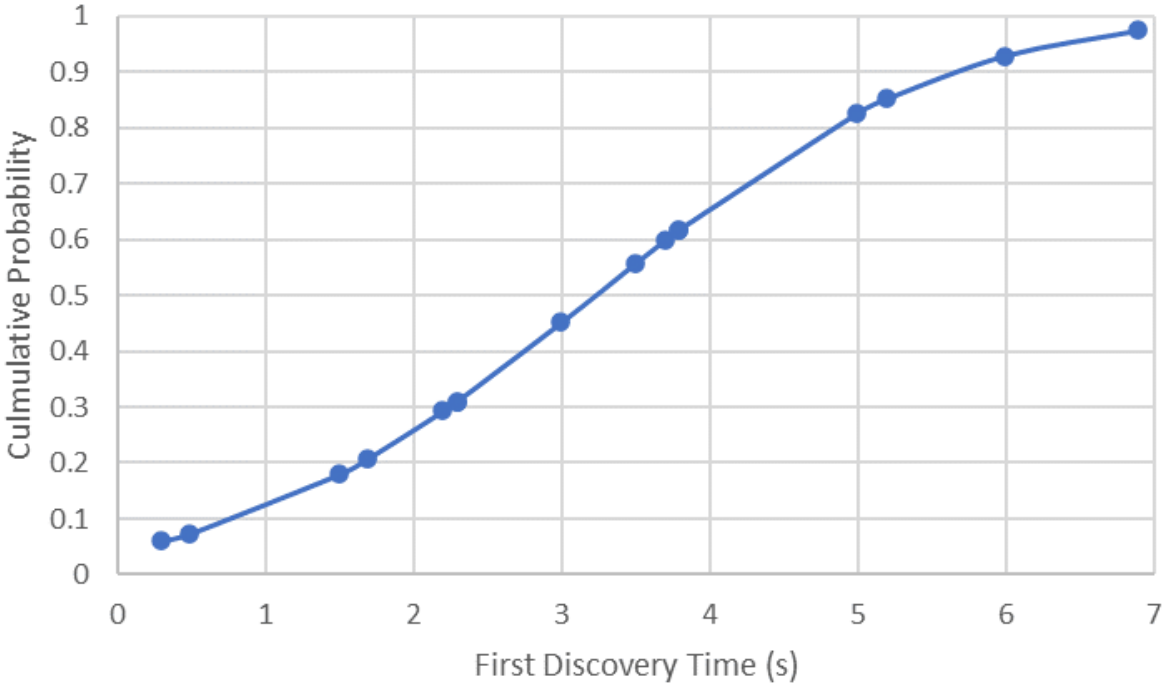
2.

No.	Name	Wake Time (s)	Sleep Time (s)	Sleep Cycle	Ave Duty Cycle (%)
2	Increased wake time	0.2	0.1	9	18.19

	Packet Interval Time (s)	First Discovery Time (s)
Mean	3.80	3.22
Standard Deviation	2.74	1.88

No.	Packet Reception Timing	Packet Reception Interval	First Discovery Time
1	1.796	7.594	0.296
2	9.390	7.000	5.195
3	16.390	1.602	0.492
4	17.992	1.898	4.992
5	19.890	7.399	3.695
6	27.289	2.101	1.496
7	29.390	2.000	3.789
8	31.390	5.102	3.492
9	36.492	1.898	3.789
10	38.390	2.000	2.289
11	40.390	0.969	2.289
12	41.359	8.330	1.69
13	49.689	1.803	6.89
14	51.492	1.195	2.195
15	52.687	6.111	2.992
16	58.798		5.992


Culmulative Probability
1
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
0
1
2
3
4
5
6
7
8
9
Packet Reception Interval (s)

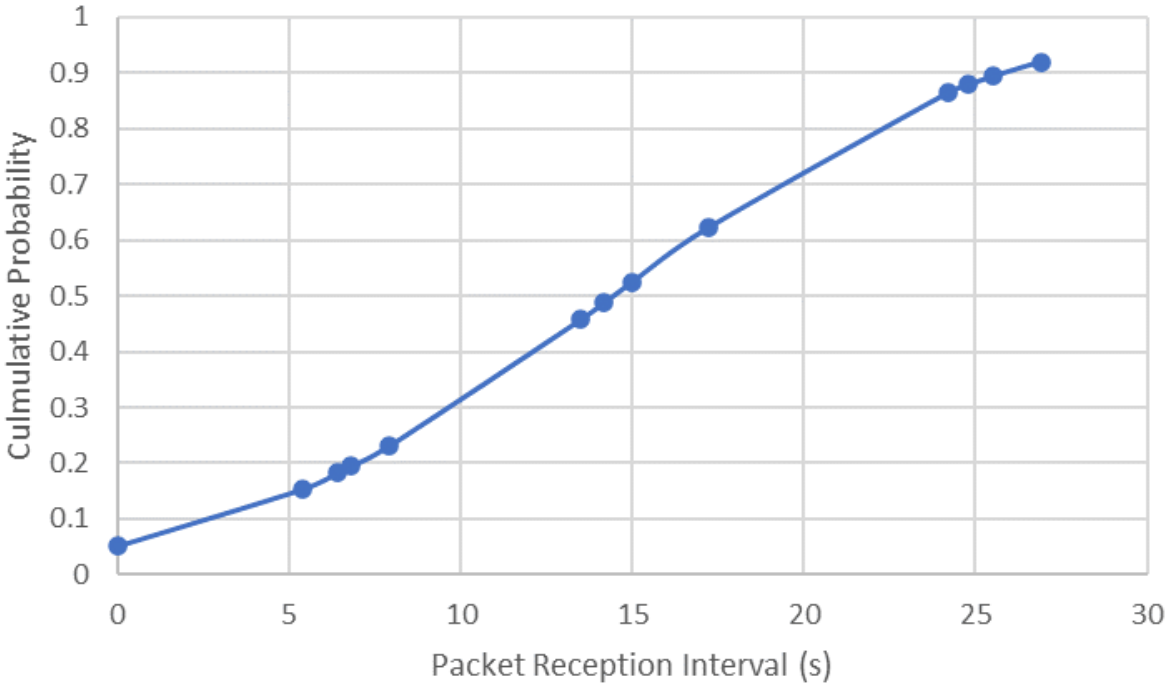

Culmulative Probability
1
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
0
1
2
3
4
5
6
7
First Discovery Time (s)

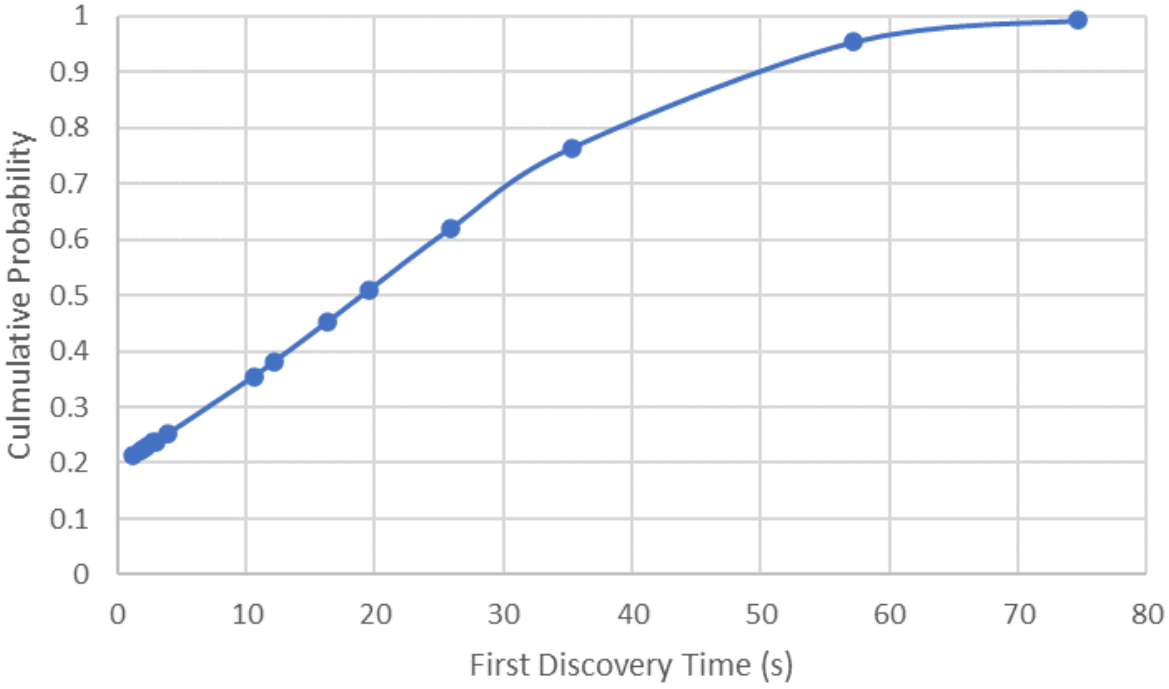
3.

No.	Name	Wake Time (s)	Sleep Time (s)	Sleep Cycle	Ave Duty Cycle (%)
3	Increased sleep time	0.1	0.2	9	5.26

	Packet Interval Time (s)	First Discovery Time (s)
Mean	19.44	18.06
Standard Deviation	8.87	21.52

No.	Packet Reception Timing	Packet Reception Interval	First Discovery Time
1	10.093	6.399	16.289
2	16.492	17.203	74.593
3	33.695	25.500	1.195
4	59.195	26.898	19.593
5	86.093	24.797	35.289
6	110.890	24.203	25.890
7	135.093	14.187	3.890
8	149.28	7.907	10.593
9	157.187	6.805	2.796
10	163.992	15.000	57.195
11	178.992	5.398	12.195
12	184.39	18.695	2.992
13	197.89	65.602	2.195
14	197.890		1.796


Culmulative Probability
1
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
0
5
10
15
20
25
30
Packet Reception Interval (s)

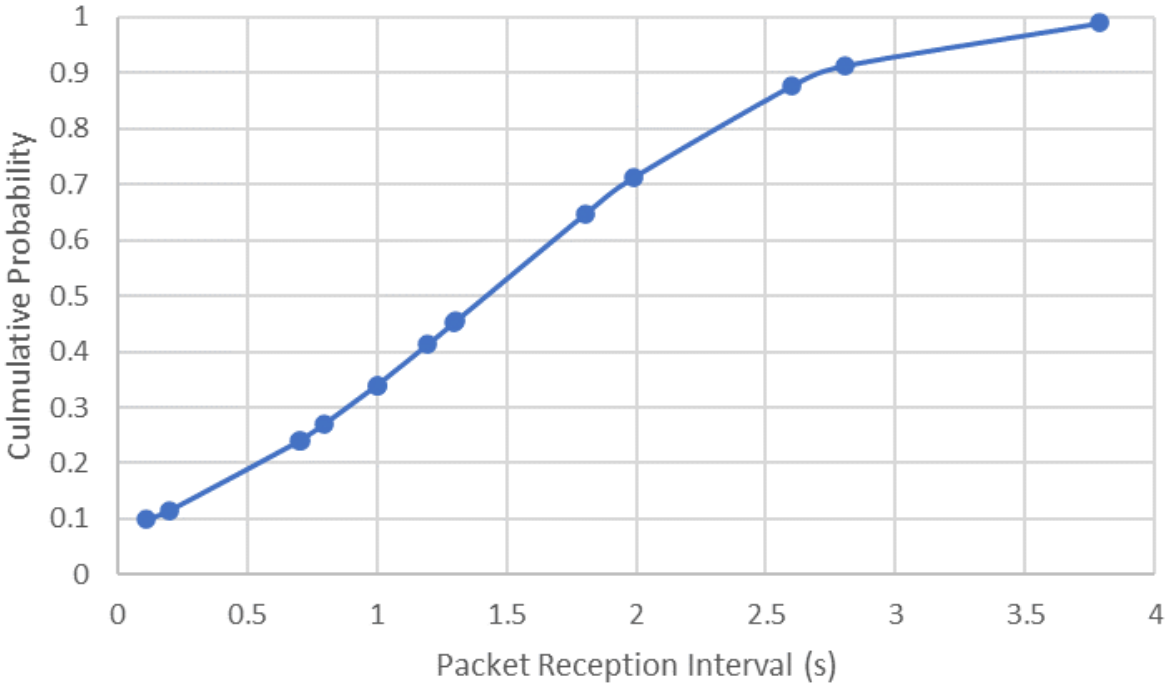

Culmulative Probability
1
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
0
10
20
30
40
50
60
70
80
First Discovery Time (s)

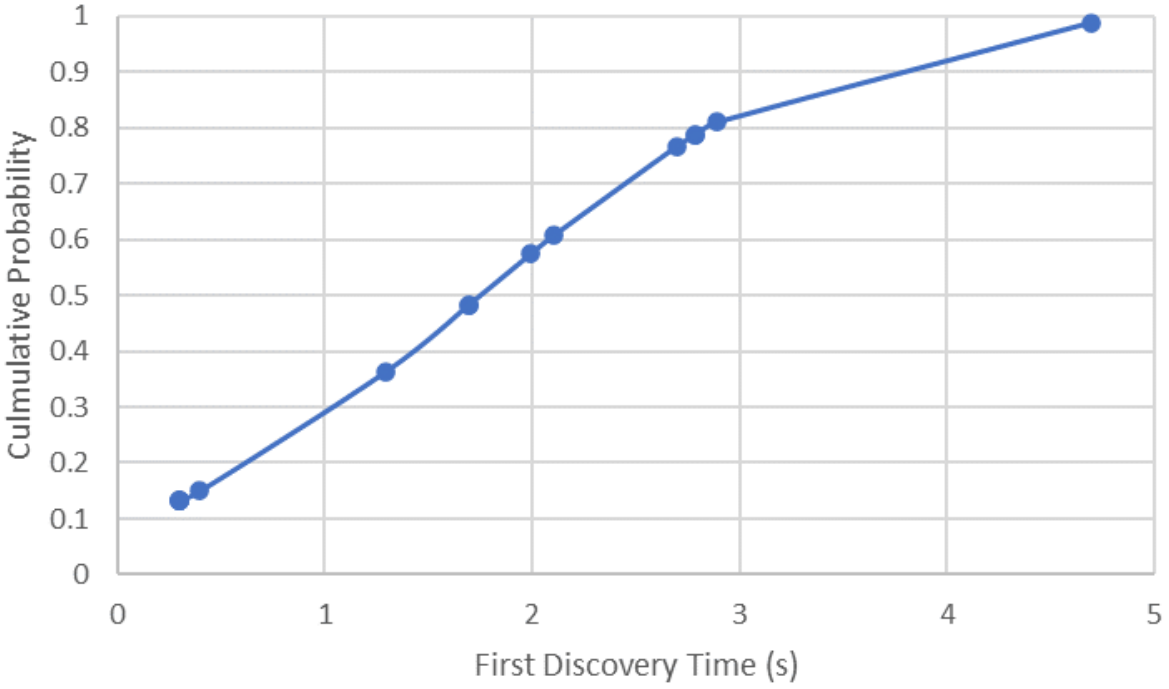
4.

No.	Name	Wake Time (s)	Sleep Time (s)	Sleep Cycle	Avg Duty Cycle (%)
4	Lower sleep cycle	0.1	0.1	5	16.67

	Packet Interval Time (s)	First Discovery Time (s)
Mean	1.42	1.75
Standard Deviation	1.02	1.30

No.	Packet Reception Timing	Packet Reception Interval	First Discovery Time
1	1.286	0.706	2.695
2	1.992	1.804	2.789
3	3.796	1.196	0.296
4	4.992	1.000	0.296
5	5.992	2.601	2.101
6	8.593	0.797	1.695
7	9.390	0.110	0.296
8	9.500	3.789	1.296
9	13.289	2.804	1.695
10	16.093	1.297	0.398
11	17.390	1.000	0.296
12	18.390	1.305	2.789
13	19.695	1.992	4.695
14	21.687	0.203	1.992
15	21.890	0.703	2.890
16	22.593		


Culmulative Probability
1
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
0
0.5
1
1.5
2
2.5
3
3.5
4
Packet Reception Interval (s)

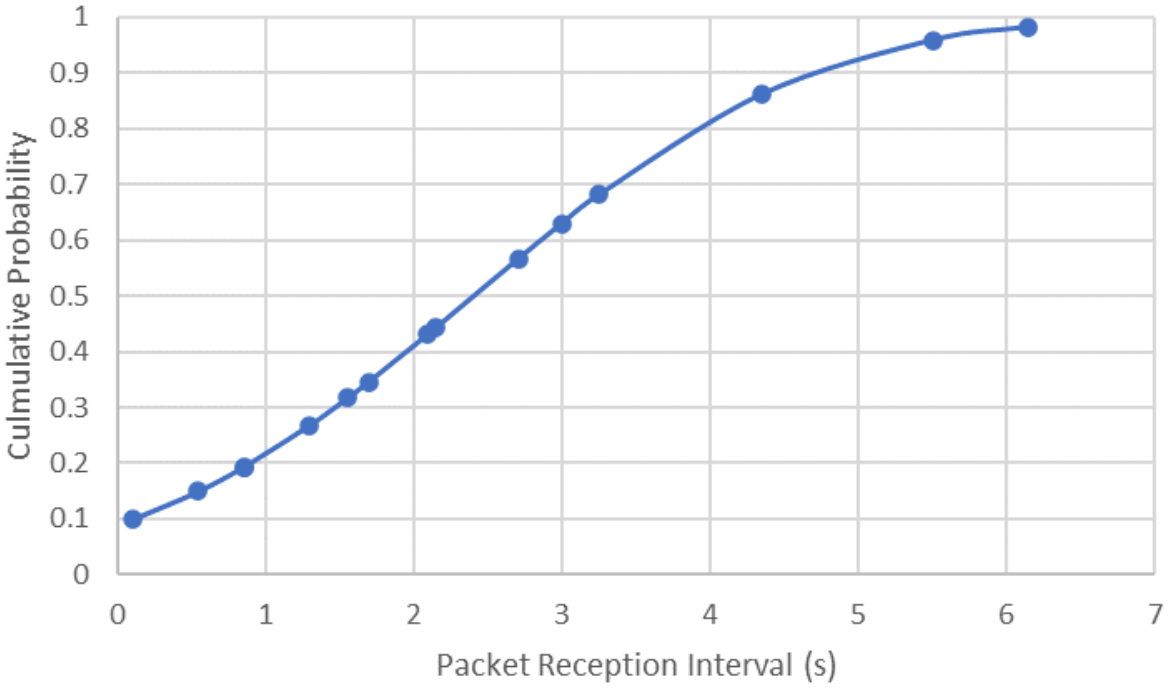

Culmulative Probability
1
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
0
1
2
3
4
5
First Discovery Time (s)

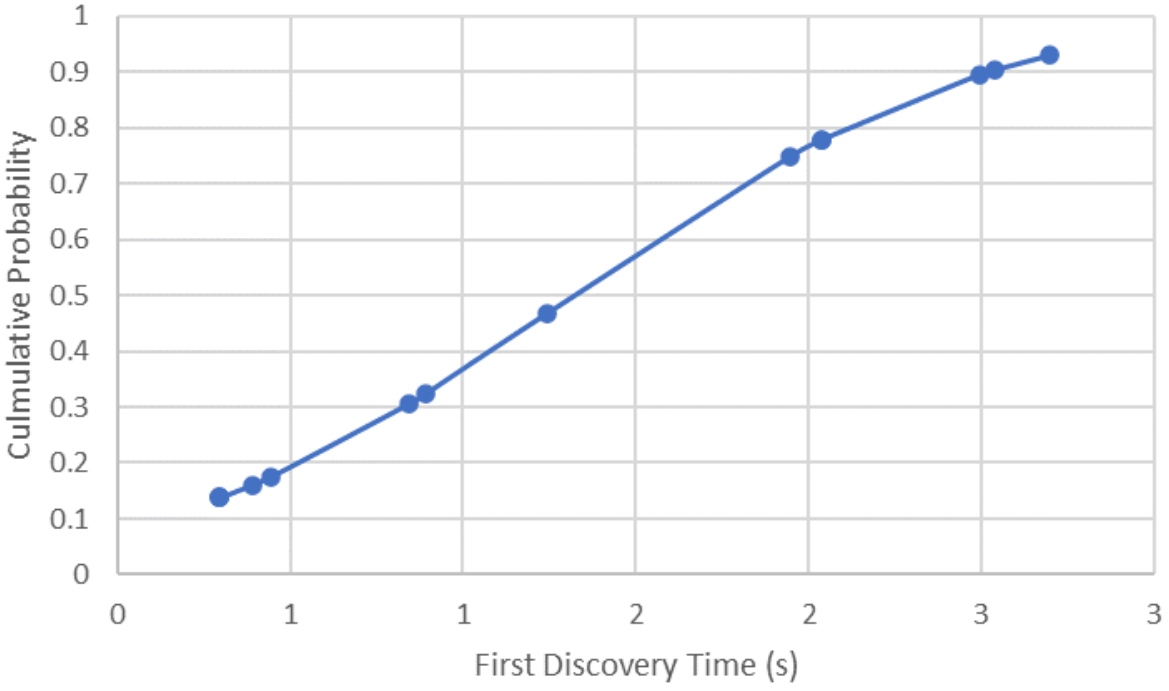
5.

No.	Name	Wake Time (s)	Sleep Time (s)	Sleep Cycle	Avg Duty Cycle (%)
5	Lower wake and sleep time	0.05	0.05	9	10.00

	Packet Interval Time (s)	First Discovery Time (s)
Mean	2.41	1.32
Standard Deviation	1.79	0.94

No.	Packet Reception Timing	Packet Reception Interval	First Discovery Time
1	3.492	0.851	2.695
2	4.343	2.149	0.390
3	6.492	0.101	0.296
4	6.593	2.094	2.539
5	8.687	0.852	1.242
6	9.539	1.554	2.492
7	11.093	1.696	1.945
8	12.789	2.703	2.039
9	15.492	1.297	2.039
10	16.789	0.546	0.296
11	17.335	5.500	0.445
12	22.835	3.000	0.890
13	25.835	6.149	0.843
14	31.984	3.250	0.296
15	35.234	4.351	
16	39.585		


Culmulative Probability
1
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
0
1
2
3
4
5
6
7
Packet Reception Interval (s)

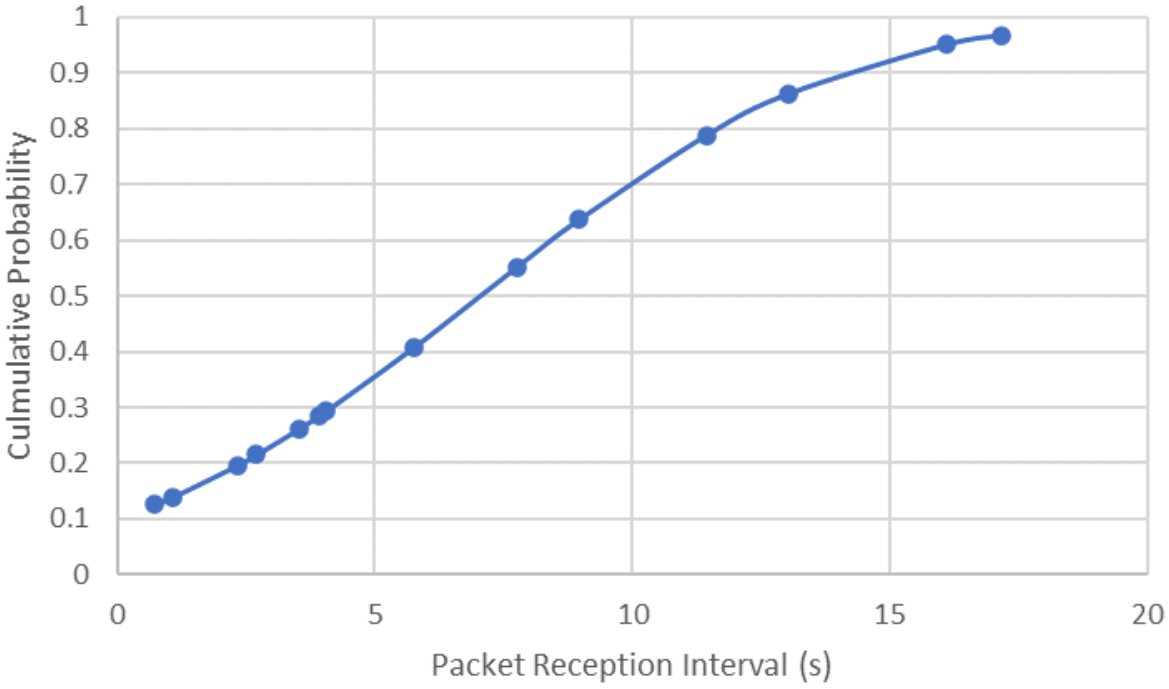

Culmulative Probability
1
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
0
1
1
2
2
3
3
First Discovery Time (s)

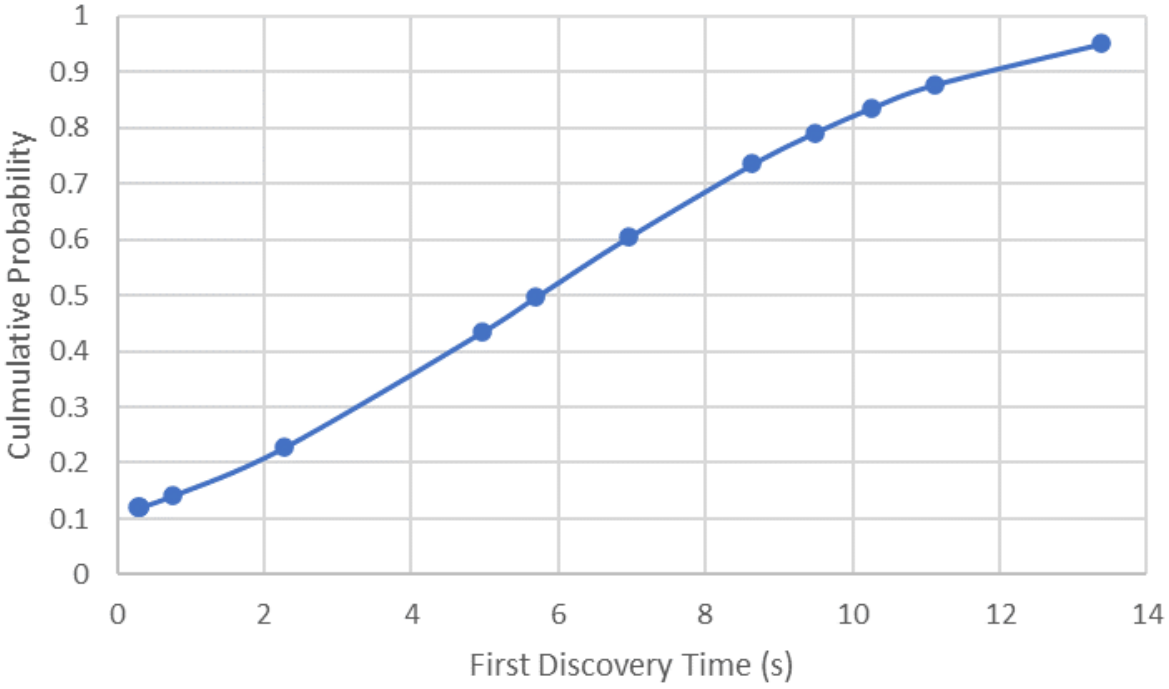
6.

No.	Name	Wake Time (s)	Sleep Time (s)	Sleep Cycle	Avg Duty Cycle (%)
6	Slightly higher sleep time	0.1	0.125	9	8.16

	Packet Interval Time (s)	First Discovery Time (s)
Mean	7.04	5.72
Standard Deviation	5.53	4.64

No.	Packet Reception Timing	Packet Reception Interval	First Discovery Time
1	2.992	0.726	0.296
2	3.718	3.524	2.265
3	7.242	3.929	10.265
4	11.171	11.446	0.765
5	22.617	17.179	8.640
6	39.796	4.047	4.968
7	43.843	5.750	0.286
8	49.593	16.102	9.492
9	65.695	13.047	13.390
10	78.742	1.078	6.945
11	79.820	2.328	11.117
12	82.148	8.969	5.695
13	91.117	2.703	0.296
14	93.820	7.750	
15	101.570		


Culmulative Probability
1
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
0
5
10
15
20
Packet Reception Interval (s)


Culmulative Probability
1
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
0
2
4
6
8
10
12
14
First Discovery Time (s)

Observations from the different settings (based on the original algorithm):

When testing out different settings, we wanted to see how changing the wake time, sleep time and average number of sleep cycles would affect the duty cycle, average packet reception interval and average first discovery time. To achieve this, we modify the 3 parameters one at a time, making them higher or lower to see how the change affects the results. Below is a summary of how our changes affect the packet reception interval and first discovery time.

Increasing Wake Time

In our 2nd set of parameters, we increased the wake time by 0.1s and left everything else untouched. Compared to the default parameters, this increases the duty cycle from 10% to 18%, almost doubling it. This means more energy is needed to be consumed, but in doing so, we observe that the mean and standard deviation of packet reception interval and mean first discovery time had all decreased. The mean packet interval time decreased from 5.06s to 3.80s, about 1.2s. However, the mean first discovery time had a notable larger decrease from 5.49s to 3.22s. From these results, we can deduce that increasing the wake time is a useful way to help the devices to find each other faster.

Increasing Sleep Time

In our 3rd set of parameters, we increased the sleep time by 0.1s, the same amount of time we increased the wake time in the 2nd set of parameters. We left everything else untouched (Wake Time at the original value of 0.1s, sleep cycle at the original value of 9). Compared to the default parameters, this decreases the duty cycle by approximately half from 10 to 5.26. However, this decrease in duty cycle may not be worth the significant increase in the mean and standard deviation of packet reception interval and mean first discovery time. They had both increased by about 4 times, with the means increasing from 5-6s to about 18-19s. The standard deviation is also much larger, with the first discovery time standard deviation increasing from 3.42s to the gigantic value of 21.5s. From these observations, we can deduce that merely increasing sleep time is not a good way to reduce power consumption and duty cycle as the decrease in performance is abysmal.

Lower Sleep Cycle

In our 4th set of parameters, we lowered the average number of sleep cycles from 9 to 5 to see how differently it affects the packet interval and first discovery times compared to changing the sleep time. Compared to the default

parameters, this increased the duty cycle from 10% to 16.67%, which is a slightly lower increase compared to the second set of parameters. The mean and standard deviation of packet reception interval and mean first discovery time had all decreased by a lot, with the means going from 5-6s to 1-2s, and standard deviations dropping to 1s. This is an even larger decrease compared to the 2nd set of parameters where we increased the wake time. As both 2nd and this set of parameters have about the same duty cycle, we can conclude that lowering the sleep cycle is a better way to get better packet transmission intervals and first discovery times compared to increasing the wake times.

Lower Wake and Sleep Time

In our 5th set of parameters, we lowered both the wake and sleep times from 0.1s to 0.05s to see whether the packet interval and first discovery times would be affected, even when both times are lowered and the duty cycle stays the same at 10% compared to the default set of parameters. From the results we can see that the mean times and standard deviations have both decreased significantly by more than half. From this we can conclude that lowering the wake and sleep times is an effective way to get better packet reception intervals and first discovery times, even though the duty cycle may be the same.

Slightly Higher Sleep Time

In our 6th set of parameters, we increased the sleep time but only slightly, from 0.1s to 0.125s, to see how different the results may be compared to the 2nd set of parameters where we doubled the sleep time from 0.1s to 0.2s. The duty cycle decreased slightly from 10% to 8%, compared to the first set of parameters. From the results, the mean packet reception interval has increased by almost double from about 5s to 9s. However, surprisingly the first discovery time has not increased much, only increasing by 0.23s from 5.49s to 5.72s. From this we can conclude that increasing the sleep time slightly is a good way to save power when only the first discovery timings are important. Compared to our 2nd set of parameters, the increase in interval and first discovery time follows an exponential trend when sleep time is increased linearly. Hence if sleep time is increased it should not be increased in excess, else it would lead to poor performance that will outweigh the decrease in duty cycle.

[4] Retry with NxN deterministic bound (Quorum-Based Protocol) - Run nbr_deterministic to collect data

You must include the following in the report (details instruction below in the document):
a) the algorithm you have implemented
b) the parameters chosen
c) the maximum two-way latency observed

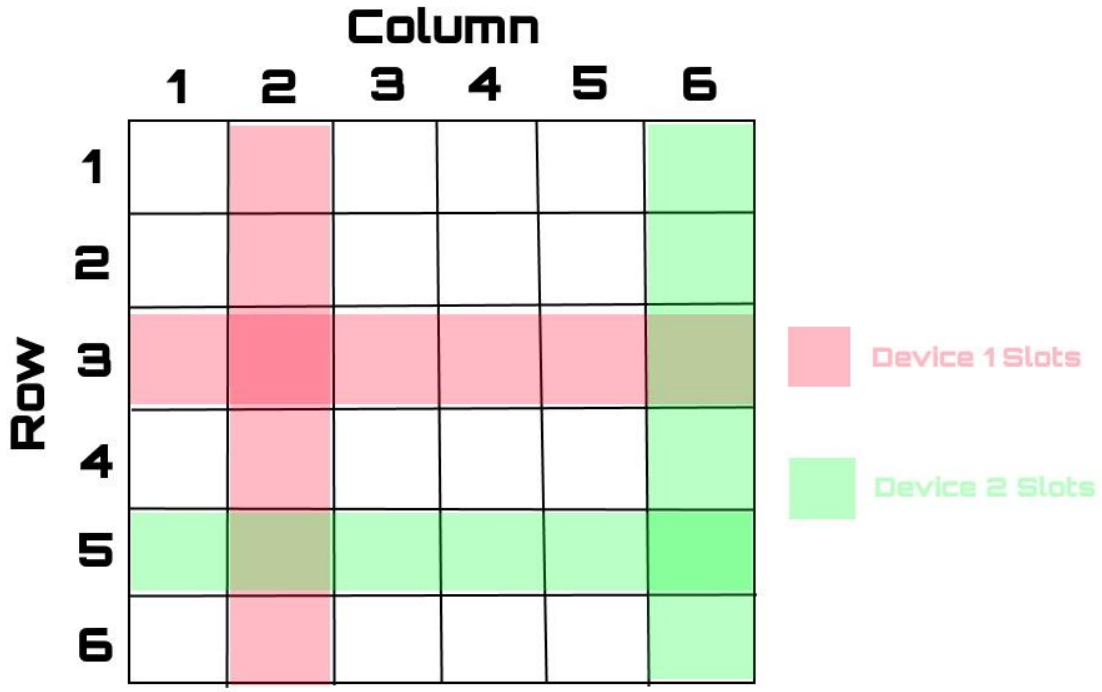
Important: Please note that the two way latency is the time that it takes for node A to hear from node B, and then for the node B to hear from A. Said another way, it is the time it takes for performing two-way neighbor discovery.

Algorithm implemented to ensure deterministic bound:

We have chosen to implement the quorum-based protocol to ensure that the two-way discovery can be done in a deterministic manner. This protocol determines when to transmit/receive based on a N x N table (N is a preset value), which contains N^2 slots. At the start of the protocol, it will randomly choose a single row and column from the table which determines all the slots that protocol should transmit/receive during. The protocol will then iterate through the N^2 slots of the table in a loop, waking and transmitting/receiving if the current slot falls within either the chosen row or column inside the N x N table.

How does the algorithm and these parameters ensure deterministic discovery in 10 seconds or less

When we have 2 devices implementing this algorithm, they each choose a row and column at random. Looking at an example of a 6x6 grid where Device 1 chooses row 3 and column 2, while Device 2 chooses row 5 and column 6.



In this case, we can see that Device 1 and Device 2 overlaps on slot 18 (Row 3, Column 6), and slot 26 (Row 5, Column 2). We can also imagine that the 2 devices choose different rows or columns by shifting the coloured rows/columns left or right/up or down, and we would see that there would always be at least 2 overlapping slots. There would be more overlapping slots if the devices chose the same rows or columns by chance. Hence it is guaranteed that the 2 devices would discover each other at least 2 times for each loop of the N x N grid. In order to make the discovery time deterministic within 10 seconds, we have to lower the duration of each slot such that the total time taken to loop through the N x N table is 10 seconds. This guarantees that the 2 devices would discover each other at least twice within 10 seconds.

This logic can be similarly extended to cover interaction between devices when there are more than 2 devices available, where each device would choose its own row and column to determine the slots it should transmit/receive in, which would overlap with the slots chosen by other devices in a deterministic way.

Parameters Chosen and Duty Cycle Comparisons

In order to choose the parameters we should use for our algorithm, we tested out different values of N for the N x N table. We have recorded the results in the table below. As we increase the value of N, we have to decrease the duration of each slot, in order to be able to keep the deterministic discovery time under 10 seconds.

N	10	15	20	25	30
Total slots	100	225	400	625	900
Active slots	19	29	39	49	59
Duty Cycle (%)	19.00	12.89	9.75	7.84	6.56
Duration per slot (ms)	100	40	25	16	10
Expected Delay (s)	2.770	2.407	2.630	2.603	2.324

We discovered that the more we increased the value of N, the lower the duty cycle and energy consumption. However, when the duration of each slot becomes shorter, this means that less data can be transferred during the slot, and shorter slot timings also makes the sensortag unstable.

We had tested up to N = 30, but the sensortag is extremely unstable by then and the algorithm randomly stopped running due to the extremely short slot duration. From our testing, we find that the limit for reducing the TIME_SLOT to is around 10ms with N = 30 not working most of the time. Hence we have sacrificed some power savings to make the sensortag more stable, by decreasing the value of N and increasing the slot duration.

After more testing, we found that N = 25 and TIME_SLOT = 16ms keeps a good balance of some stability and power savings. This means that N = 25 is the maximum N that we can run without the sensortag becoming extremely unstable.

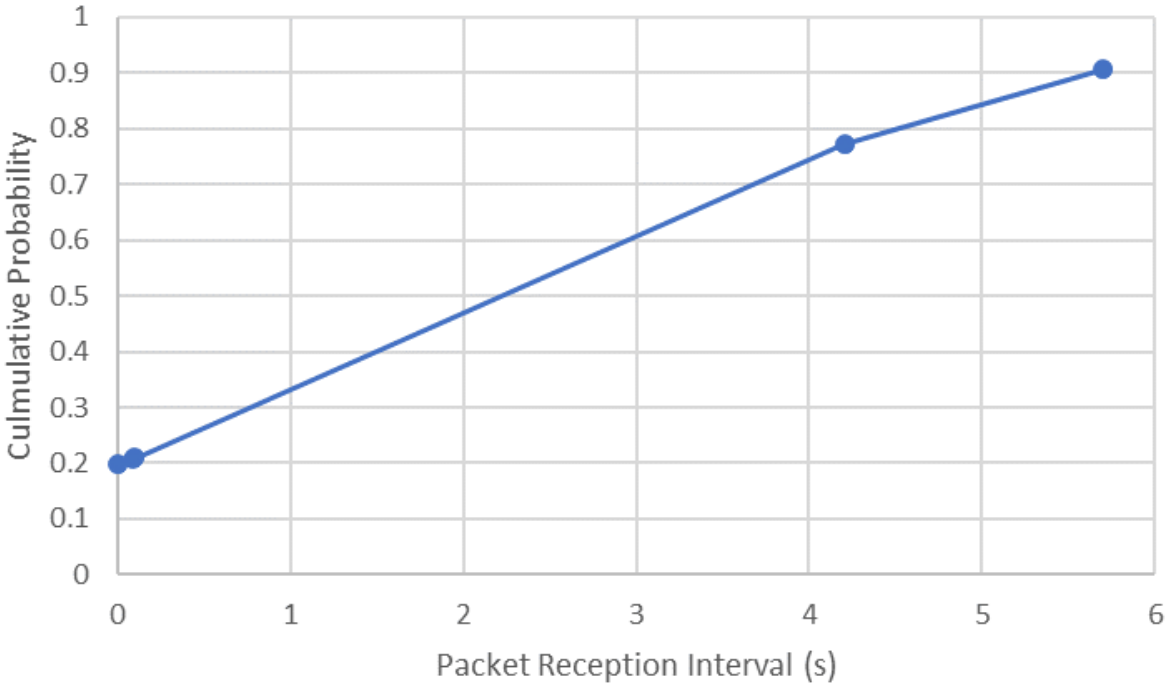
General Performance of our Deterministic Protocol

n = 10

	Packet Interval Time (s)
Mean	2.223
Standard Deviation	2.631

No.	Packet Reception Timing	Packet Reception Interval
1	35.000	0.093
2	35.093	5.696
3	40.789	0.000
4	40.789	4.203
5	44.992	0.101
6	45.093	5.696
7	50.789	0.000
8	50.789	4.203
9	54.992	0.101
10	55.093	

Observations:
The mean and standard deviation of the packet reception interval using the quorum-based protocol is noticeably lower than that default settings of the non-deterministic protocol (as well as lower than most other settings of the default protocol less when the sleep cycle is lowered).


Culmulative Probability
1
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
0
1
2
3
4
5
6
Packet Reception Interval (s)

Maximum 2 way latency that we expect (Time between A know B is there then B knowing A is there):

Snippet of collected data for when N = 25

No.	Node A Packet Timing	Node B Packet Timing
1A	59.960 (Send)	
1B		59.976 (Receive)
2A		60.059 (Send)
2B	60.075 (Receive)	

To calculate the maximum 2 way latency, we calculate the time between a packet being received by Node B, and the time that a packet is received by Node A later.

From this snippet of collected data, we find that the 2 way latency we observed is 0.099s (60.075s - 59.976s). However, this is only one of our observations.

The theoretical maximum 2 way latency that we can observe should be bounded by the slot duration. Hence the maximum for N = 25 should be 16ms, however the maximum 2 way latency observed is much larger, possibly due to imperfections in the hardware of the sensortag.

TASK 2:

CRITERIA:

- Detection accuracy. Accuracy includes detection of new nodes moving near, existing nodes moving away, and the times it takes to detect these events.
- Robustness. A node should be able to perform well with obstacle and potential collision (existence of additional node(s))
- Steps taken to reduce energy consumption
- Logic for transferring light sensor readings
- Creativity of solution

Neighbor Discovery Protocol Logic Implementation

As in TASK 1, we have chosen to implement the quorum-based protocol to ensure that the two-way discovery can be done in a deterministic manner. This protocol determines when to transmit/receive based on a N x N table (N is a preset value), which contains N^2 slots. At the start of the protocol, it will randomly choose a single row and column from the table which determines all the slots that protocol should transmit/receive during. The protocol will then iterate through the N^2 slots of the table in a loop, waking and transmitting/receiving if the current slot falls within either the chosen row or column inside the N x N table. For a more detailed explanation of how this algorithm works, please refer to Task 1. The code snippet for the deterministic discovery protocol is as follows:

```
while (1)
{

  if (r == row || c == col)
  {
    // radio on
    NETSTACK_RADIO.on();

    // send NUM_SEND number of neighbour discovery beacon packets
    for (i = 0; i < NUM_SEND; i++)
    {

      // Initialize the nullnet module with information of packet to be trasnmitted
      nullnet_buf = (uint8_t *)&light_sensor_packet; // data transmitted
      nullnet_len = sizeof(light_sensor_packet);     // length of data transmitted

      light_sensor_packet.seq++;

      curr_timestamp = clock_time();

      light_sensor_packet.timestamp = curr_timestamp;

      slot_num = r * N + c;

      printf("Send seq# %lu  @ slot# %d %8lu ticks   %3lu.%03lu\n", light_sensor_packet.seq, slot_num, curr_timestamp

      NETSTACK_NETWORK.output(&dest_addr); // Packet transmission

      // wait for WAKE_TIME before sending the next packet
      if (i != (NUM_SEND - 1))
      {

        rtimer_set(t, RTIMER_TIME(t) + TIME_SLOT, 1, (rtimer_callback_t)sender_scheduler, ptr);
        PT_YIELD(&pt);
      }
    }

    // radio off
    NETSTACK_RADIO.off();
  }

  else
  {

    rtimer_set(t, RTIMER_TIME(t) + TIME_SLOT, 1, (rtimer_callback_t)sender_scheduler, ptr);
    PT_YIELD(&pt);
  }
```

```
c = (c + 1) % N;
if (c == 0)
{
  r = (r + 1) % N;
}
```

After testing, we found that N = 25 and TIME_SLOT = 16ms from Task 1 is less stable for this application, possibly due to the short time slot. Hence after more testing we chose N = 15 and TIME_SLOT = 40ms instead, which keeps a good balance of some stability and reduced power usage.

This algorithm was also shown in TASK 1 to have optimal power consumption.

Proximity Detection Logic Implementation

To implement proximity detection, where DETECT indicates that nodes are in close proximity within 3 meters for 15 seconds or more and ABSENT indicates a node in proximity has moved away for 30 seconds or more, we used an array of node data structures to keep track of the node_id, time_detected and time_absent.

```
typedef struct
{
  unsigned long node_id;
  unsigned long time_detected;
  unsigned long time_absent;

} node;

static volatile node node_list[5];
```

```
int rssi = (signed short)packetbuf_attr(PACKETBUF_ATTR_RSSI);

unsigned long curr_time = clock_time();

if (rssi < THREE_METER_THRESHOLD)
{
  printf("AWAY\n");
  for (int i = 0; i < 5; i++)
  {
    if (node_list[i].node_id == received_packet_data.src_id)
    {
      node_list[i].time_detected = 0;
      printf("CURRENT TIME %lu TIME ABSENT %lu\n", curr_time / CLOCK_SECOND, node_list[i].time_absent / CLOCK_SECOND);
      if (node_list[i].time_absent == 0)
      {
        node_list[i].time_absent = curr_time;
      }
      else
      {
        if ((curr_time / CLOCK_SECOND - node_list[i].time_absent / CLOCK_SECOND) >= 30)
        {
          printf("%lu ABSENT %lu\n", node_list[i].time_absent / CLOCK_SECOND, node_list[i].node_id);
          node_list[i].node_id = 0;
          node_list[i].time_detected = 0;
          node_list[i].time_absent = 0;
        }
      }
      break;
    }
  }
}
```

```
else
{
  printf("NEAR\n");
  bool new_node = true;
  for (int i = 0; i < 5; i++)
  {
    if (node_list[i].node_id == received_packet_data.src_id)
    {
      new_node = false;
      node_list[i].time_absent = 0;
      if (node_list[i].time_detected == 0)
      {
        node_list[i].time_detected = curr_time;
      }
      else
      {
        if ((curr_time / CLOCK_SECOND - node_list[i].time_detected / CLOCK_SECOND) >= 15)
        {
          printf("%lu DETECT %lu\n", node_list[i].time_detected / CLOCK_SECOND, node_list[i].node_id);

          node_list[i].time_detected = curr_time;
        }
      }
      break;
    }
  }
  if (new_node)
  {
    printf("NEW NODE\n");
    for (int i = 0; i < 5; i++)
    {
      if (node_list[i].node_id == 0)
      {
        node_list[i].node_id = received_packet_data.src_id;
        node_list[i].time_absent = 0;
        node_list[i].time_detected = curr_time;
        break;
      }
```

As shown in the code snippets above, whenever we receive a neighbor discovery packet, we first check if the RSSI value is more negative than the THREE_METER_THRESHOLD set. If it is, the node discovered is not in proximity. If a node in the node_list matches the src_id, its time_detected is reset to 0. If time_absent was 0, the node just transitioned from in proximity to away and its time_absent is set to curr_time. Else, time_absent is subtracted from curr_time to check if the node has been away for more than 30s. If it has been away for more than 30s, ABSENT is printed along with the node_id and time it was absent. The node is also removed from the node_list.

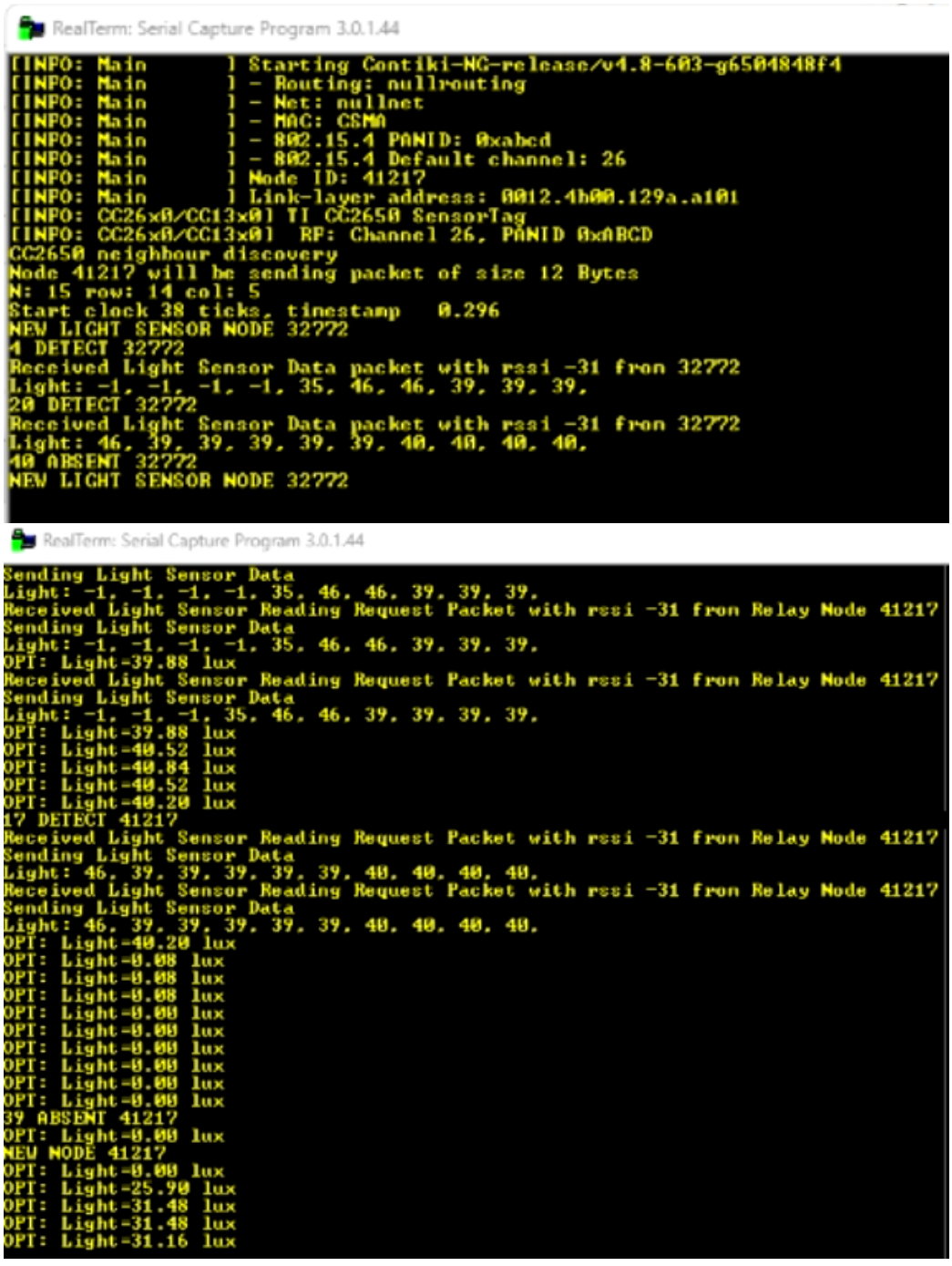
If the node discovered is within 3 meters, and the src_id is not found in the node_list, NEW NODE is printed and the node is added to the node_list with time_detected set to curr_time. If a node in node_list matches src_id, its time_absent is reset to 0. If time_absent was 0, the node just transitioned from away to in proximity and its time_detected is set to curr_time. Else,

time_detected is subtracted from curr_time to check if the node has been in proximity for more than 15s. If it has been in proximity for more than 15s, DETECT is printed along with the node_id and time it was first detected. The time_detected is then reset back to curr_time so DETECT will be printed if it is still in proximity 15s later. **(This means DETECT will keep printing every 15 seconds while the node is still in proximity)**

```
if (c == 0)
{ // check absent every 10/N seconds when discovering
  curr_timestamp = clock_time();

  for (int i = 0; i < 5; i++)
  {
    if (node_list[i].node_id != 0)
    {

      if (node_list[i].time_absent != 0)
      {
        printf("CURRENT TIME %lu TIME ABSENT %lu\n", curr_timestamp / CLOCK_SECOND, node_list[i].time_absent / CLOCK_SECOND);
        if ((curr_timestamp / CLOCK_SECOND - node_list[i].time_absent / CLOCK_SECOND) >= 30)
        {
          printf("%lu ABSENT %lu\n", node_list[i].time_absent / CLOCK_SECOND, node_list[i].node_id);
          node_list[i].node_id = 0;
          node_list[i].time_detected = 0;
          node_list[i].time_absent = 0;
        }
      }

      if (node_list[i].time_detected != 0)
      {
        printf("CURRENT TIME %lu TIME LAST DETECTED %lu\n", curr_timestamp / CLOCK_SECOND, node_list[i].time_detected / CLOCK_SECOND);
        if ((curr_timestamp / CLOCK_SECOND - node_list[i].time_detected / CLOCK_SECOND) >= 30)
        {
          printf("%lu ABSENT %lu\n", node_list[i].time_detected / CLOCK_SECOND, node_list[i].node_id);
          node_list[i].node_id = 0;
          node_list[i].time_detected = 0;
          node_list[i].time_absent = 0;
        }
      }
    }
  }
}
```

In cases where the node moves away such that neighbor discovery packets are no longer received, an additional layer of check is implemented for ABSENT. At intervals of 10/N seconds within neighbor discovery protocol, for nodes with time_absent and time_detected != 0, it is checked against current time to determine if the node has not received any discovery packets for more than 30s, if so, ABSENT is printed and the node is removed from the node_list.

Evaluation of Implemented System



From our repeated extensive testing, it appears that the proximity detection logic is working well, with the nodes getting detected as in proximity and absent in acceptable time. We had also tested with obstacles such as a thin glass panel in between and interference by additional nodes and it is still working well most of the time.

Light Sensing and Transmitting Logic Implementation

The light sensor readings are taken every 3 seconds by the sender/light sensor SensorTag so that an array of 10 readings can be filled in 30s (as mentioned in SLACK thread). A separate process was used to sample the light sensor and populate the array. The latest reading is populated from the back so that light sensor readings are printed in the order “Light: Reading 1, Reading 2, …. , Reading 10”.

```
static void
get_light_reading()
{
  int value;
  int current_lux;

  value = opt_3001_sensor.value(0);
  if (value != CC26XX_SENSOR_READING_ERROR)
  {
    printf("OPT: Light=%d.%02d lux\n", value / 100, value % 100);
    current_lux = value / 100;
    for (int i = 0; i < 9; i++)
    {
      light_sensor_readings[i] = light_sensor_readings[i + 1];
    }
    light_sensor_readings[9] = current_lux;
  }

  init_opt_reading();
}

static void
init_opt_reading(void)
{
  SENSORS_ACTIVATE(opt_3001_sensor);
}

PROCESS_THREAD(light_sensor_process, ev, data)
{

  PROCESS_BEGIN();
  init_opt_reading();

  while (1)
  {

    etimer_set(&timer_etimer, CLOCK_SECOND * 3); // sample every 3 seconds
    PROCESS_WAIT_EVENT_UNTIL(ev == PROCESS_EVENT_TIMER);
    get_light_reading();
  }

  PROCESS_END();
}
```

When a relay SensorTag detects that it is in close proximity to the SensorTag with the light sensor (within 3 meters) for a time period more than 15s, it starts the transfer process for light readings by setting the flag wait_for_light_sensor_readings. When the relay SensorTag receives the light_sensor_readings, the last 10 readings are printed out and the flag is cleared.

```
  else if (rssi >= THREE_METER_THRESHOLD)
  {
    printf("NEAR\n");
    if (light_sensor_node.node_id == 0)
    {
      printf("NEW LIGHT SENSOR NODE\n");
      light_sensor_node.node_id = received_packet_data.src_id;
      linkaddr_copy(&light_sensor_node_addr, src);
    }
    light_sensor_node.time_absent = 0;
    if (light_sensor_node.time_detected == 0)
    {
      light_sensor_node.time_detected = curr_time;
    }
    else
    {
      if ((curr_time / CLOCK_SECOND - light_sensor_node.time_detected / CLOCK_SECOND) >= 15)
      {
        printf("%lu DETECT %lu\n", light_sensor_node.time_detected / CLOCK_SECOND, light_sensor_node.node_id);
        light_sensor_node.time_detected = curr_time; // DETECT in another 15s if still in proximity
        wait_for_light_sensor_readings = 1;          // set flag
      }
    }
  }
}

else if (len == sizeof(light_sensor_readings_packet))
{

  static light_sensor_readings_packet_struct received_packet_data;

  // Copy the content of packet into the data structure
  memcpy(&received_packet_data, data, len);

  // Print the details of the received packet
  printf("Received Light Sensor Data packet with rssi %d from %ld\n", (signed int)packetbuf_attr(PACKETBUF_ATTR_RSS
  printf("Light: %d, %d, %d, %d, %d, %d, %d, %d, %d, %d,\n", received_packet_data.light_sensor_readings[0], receive

  wait_for_light_sensor_readings = 0; // clear flag
}
```

Within the sender_scheduler of the relay node, if the wait_for_light_sensor_readings flag is set, instead of the discovery protocol, the radio is kept ON and request packets are sent at intervals of TIME_SLOT until the flag is cleared. The **request packet** is only sent to the light sensor node using the **light_sensor_node_addr** saved when it first discovered the NEW LIGHT SENSOR NODE.

```
if (wait_for_light_sensor_readings)
{
  // radio on
  NETSTACK_RADIO.on();

  nullnet_buf = (uint8_t *)&request_packet; // data transmitted
  nullnet_len = sizeof(request_packet);     // length of data transmitted
  request_packet.src_id = node_id;
  request_packet.timestamp = clock_time();

  while (wait_for_light_sensor_readings)
  {
    printf("Sending Request for Light Sensor Readings\n");
    NETSTACK_NETWORK.output(&light_sensor_node_addr); // Packet transmission
    rtimer_set(t, RTIMER_TIME(t) + TIME_SLOT, 1, (rtimer_callback_t)sender_scheduler, ptr);
    PT_YIELD(&pt);
  }

  // radio off
  NETSTACK_RADIO.off();
}
```

When the light sensor node receives the request packet, it then responds to the **corresponding relay node** by sending the requested array using received **src** as the sending address.

```
else if (len == sizeof(request_packet))
{

  static request_packet_struct received_packet_data;

  // Copy the content of packet into the data structure
  memcpy(&received_packet_data, data, len);

  // Print the details of the received packet
  printf("Received Light Sensor Reading Request Packet with rssi %d from Relay Node %ld\n", (signed int

  // Initialize the nullnet module with information of packet to be trasnmitted
  nullnet_buf = (uint8_t *)&light_sensor_readings_packet; // data transmitted
  nullnet_len = sizeof(light_sensor_readings_packet);     // length of data transmitted

  light_sensor_readings_packet.src_id = node_id;
  light_sensor_readings_packet.timestamp = clock_time();
  for (int j = 0; j < 10; j++)
  {
    light_sensor_readings_packet.light_sensor_readings[j] = light_sensor_readings[j];
  }

  printf("Sending Light Sensor Data\n");
  printf("Light: %d, %d, %d, %d, %d, %d, %d, %d, %d, %d,\n", light_sensor_readings[0], light_sensor_rea

  NETSTACK_NETWORK.output(src); // Only send to node that requested
}
```

NOTES:

In order to be able to detect the distance of 3m, we had to choose an appropriate RSSI value for this. For our choice of RSSI value, we chose the RSSI value of -50 based on our experiments for Assignment 3A. This RSSI value is chosen for the 3m threshold and may be different across different sensortags, since we cannot control how sensitive the hardware of different sensortags are.

To run our code, please follow the README in the source-code to run the files. Inside the extra nbr folder, we have collated the code and the data that we had collected for most of the different tests that we had ran for this project.

Possible Applications of our Project (Neighbour Discovery Protocols)

- Trace Together (Contact Tracing Applications/Devices)
    - Neighbour discovery protocols are required for trace together to work, for different trace together devices to know which other devices it had been near. Trace Together devices need to use a protocol that has a very small duty cycle in order to save the most power and make them last as long as possible.
- Apple Air Tags
    - Air Tags and other similar tracking devices require the use of neighbour discovery protocols in order for them to be trackable. Apple Air Tags and other Apple devices use neighbour discovery protocols in order to discover each other, so that the other Apple device can inform the cloud about that Air Tag’s current whereabouts so that the whole system would work.
- Autonomous Vehicles
    - Neighbour discovery protocols allow automatic driving vehicles to discover each other when they come into proximity. This helps the 2 autonomous vehicles to coordinate their movement by communicating with each other so that they have a lower chance of a collision.

END OF REPORT

Thank you for reading through our report for the project. Please do take our project into consideration for Bonus based on the considerations raised on the first page of this report.