
INDIAN INSTITUTE OF INFORMATION TECHNOLOGY
ALLAHABAD

8th Semester Course Project

# WSN-Project : Implementing Aggregation in CTP protocol in Contiki-OS

Submitted Under Guidance of -

Dr. Jagpreet Singh
**Assistant Professor,**
Department Of Information Technology,
Indian Institue of Information Technology, Allahabad

# Table Of Contents :-



# Submitted by :-


1. **PRABHAKAR KUMAR - IRM2017008**
2. **RITESH YADAV - IRM2017001**

# INTRODUCTION :-

Wireless Sensor Networks can be defined as spatially dispersed, decentralized and dedicated sensors that monitor and record the physical conditions of the environment, collect the relevant data and forward the collected data to a central location, or better called data sink where all data is to be stored ultimately.

It should be noted that not every sensor node can have a direct connection to the sink node, hence the transmission of data over the network happens by a chain mechanism, where data hops from one node to another, until it reaches the sink. The chain of nodes or better the path that has to be followed to get to the sink should appropriately be as short as possible, which is done by the Routing Protocols, that generate the data for the shortest path from a source to the sink.

Then one the major bottle-necks for WSN is that the sensor nodes are tiny devices and have a very limited battery life. So the design of routing algorithms and the transmission of data should be done in such a way that it minimizes the energy consumption of the nodes. Let's say we have a scenario where we have an event at any location, which is sensed by the sensor nodes around the event, then the nodes then need to pass the data towards the sink, and if we use a binary strategy that for each event, all the nodes that sense it forward it to the sink, then it will require a very large energy consumption. This is where we can apply aggregation of events at the sensor nodes, such that we can reduce the power consumption.

Now if we see forward to aggregation, then it could have been done in three different ways, one where aggregation is done on the basis of number of events or messages at a node, second where aggregation is done based upon a time frame, and the third where we can use both number of messages and time frame, and check the one that causes the exhaustion of resources first. Once let's say we have the number of messages that we can handle or we have reached the end of our time frame, then we can aggregate all the messages that we have accumulated, aggregate those messages, and send the aggregated message to the sink, instead of sending them one by one. This way we can significantly reduce the amount of energy consumed in transmission of the data over the network to the sink.

In the report ahead we have proposed usage of both types of aggregation methods, have mentioned the pros and cons of each one that we could think of, and have then explained how we used a hybrid approach to arrive at what we assume to be a better approach to aggregation. We then also have presented the results that we arrive at over several metrics, and compare over different scenarios.

# METHODOLOGY :-

In order to achieve our objective we have used the Contiki Operating System which comes with Cooja Simulator, where we can define our motes or the sensor nodes, define the specific codes that each of the nodes execute and also can run scripts to generate events, and see how the nodes respond to the events and how the data flows over the network.

We had been given two files as a part of this project, one with the Javascript code that is to be used to generate events and the other a C file which is the code that the sensor nodes would execute. Once we have opened a new simulation in Cooja, we should first load the script that is to generate the events, and that can be done by simply pasting the Javascript code to the Simulation Script Editor, that is present under the Tools tab, and once pasted then we just need to Activate the script. After this step we can be sure that once we start the simulation, the Javascript code would generate the events with its code.

Now after this we then need to create our sensor nodes, which can be created as Skymotes, where we overwrite the C file that was given to us, such that each node, runs the same script when simulated. Once we have configured the motes too, we can then start the simulation and all the outputs like which node received the event, which node sent the event message, what event message was sent and also when did the sink receive the message can be seen on the mote output tab of cooja.

If we give a brief description of the code files, what the Javascript code is doing is that first it gathers the location of each of the nodes, and sends a -

- message to each node telling it its own coordinates. Then it generates an event at a specific location, here we have taken it to be (20,20), and find all the nodes that are under the given radius of the location, and send the message of event to these recognized motes. It does so in a loop, such that there are four events to be generated with four different messages. But we have made slight changes in the Javascript code that each event is created after a delay of 3 second after the previous event, and the event does not stop after a given counter, but keeps on repeating.

```
GENERATE_MSG(3000, "sleep");
WAIT_UNTIL(msg == "sleep");
```

The above two lines enabled us to delay events one after the other by a given time of 3 second. Here it should be noted that the time by which each event waits after the previous event also becomes one the the Hyperparaters of the problem.

Then if we talk about the C code that is executed on the sensor nodes, then each node first recognised the sink to which it should send the data to. Here, the first node with ID 1 always acts as the sink node. Then it responds to each of the messages that it receives, as a message is notified by an incoming message event handler. In the barebone code that was given, upon receiving the message, each node by default routes the message towards the sink, without doing any aggregation.

Now we had two options that we can use to enable aggregation: first the number of messages and second a time frame. We first moved forward by applying aggregation by using the number of messages received as the parameter. Hence what we wanted to achieve is that once we have received lets say K number of messages, then only we will forward an aggregated form of all these individual K messages to the sink as one. This

seems a reasonable strategy for an application where the number of messages are large and evenly distributed over each node, but it may fail badly otherwise. Let's say that a node received some messages but then it does not attain the K threshold, then all the stored messages will never be sent to the sink, or even if the threshold is achieved after a long time, the overall waiting time would increase drastically.

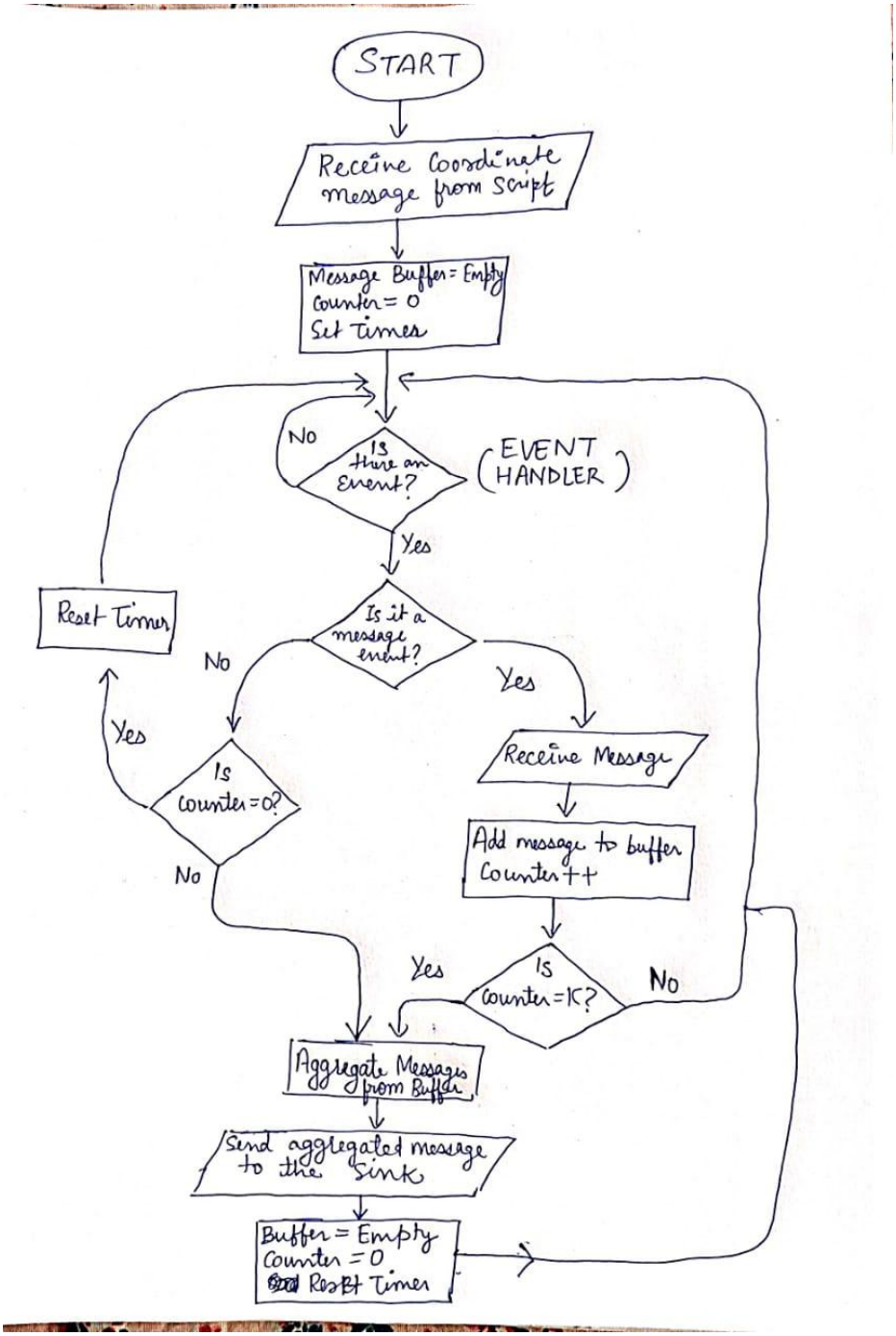
The other option was to use a time frame to aggregate the messages, in which the strategy would be like that we will wait for T time seconds at each node, and aggregate all the messages the node has received over this time period. This seems a pretty better approach, but this may also have a bottle-neck when there are nodes that receive so high a number of messages in the given time period that we can not aggregate them into one message and hence we would need to use multiple messages to send all the data to the sink.

Once we had explored the pros and cons of each of the methods, we then resolved to a hybrid method where we are taking care of both the number of messages as well as the time period that we are waiting for. In this we have two metrics, one K which defines the highest number of messages that we can aggregate into a single message, and T, which is the maximum time period that we can wait at a node. We keep on checking for the first condition that gets filled, like if either the number of messages reaches to K or the time period completes, in any of the two events, we aggregate the messages, and this time we can now be sure that the aggregated message would require only one message to be sent to the sink, and once sent we would empty the message buffer and also reset the timer.

We have submitted files corresponding to the first method as well as the hybrid method with this report, and a detailed explanation of how exactly we are doing aggregation in our algorithm can be found in the **Approach** section.

# APPROACH :-

Here we give a detailed description of the algorithm approach that we have used to aggregate the messages, and we would like to take help of a Flow Chart, as follows:



**NOTE**- In the above flow chart the parallelogram nodes that generally deonte any input or output denote any incoming or outgoing messages, while the other nodes represent their general proposes.

So this code runs independently on each of the nodes, and once the node has printed the coordinates that it has received from the Javascript script, the node then specifies three things:

- Message Buffer, that stores all the incoming messages, until they are forwarded towards the sink.
- Counter, which denotes the number of messages in the buffer.
- Timer, which iterates over a certain amount of time period and generates an event on timeout.

It then waits for the Event Handler to respond for an event. Here we have used two types of events:

- Event due to Incoming Message
- Event due to Timer Timeout.

In case of an event due to an incoming message, it first receives the message and adds the message to the buffer and increments the counter. Now if the counter counts to K, which means we have received the maximum number of messages that we can aggregate as one message then we move forward to the aggregation and sending step.

```
if(ev == serial_line_event_message && data != NULL)
```

This is the line where we check whether the event is that due to an incoming message, as the script sends a serial line message to a node in case of an event.

In the aggregation step, we concatenate each of the messages one after the other, in the order they were received, and then send it over to the sink.

Once the message is sent we clear the buffer, set the counter to 0, signifying that we don't have any more pending messages to be sent and also reset the timer.

In case of an event due to time out, we first need to check if we have any message that we can send, if not then we need to just reset the timer and listen for events. In case we have any message or say M number of messages, where M is surely to be less than K then we again proceed to the aggregation and sending step.

else if(ev == PROCESS_EVENT_TIMER)

This is the line with which we check whether the event that has triggered the event handler is due to the timer denoting a timeout or completion of the time frame.

Hence with the event handler listening to both types of event and responding differently to different type of recognized events, we are able to implement the hybrid approach that we had proposed in the previous section.

One major issue that I would like to bring up is that we encountered several errors while trying to use the default string concatenation function that is given by C compiler, and hence we had to implement our own concatenation function, such that it iterates over s string adding it to the buffer until we encounter a NULL character.

# RESULTS & COMPARISONS :-

It should be noted that in our proposed hybrid approach for aggregation, we now have three hyperparameters:

- Event Time, the time after which a new event is generated after the previous event.
- Waiting Time, the time till which each of the nodes waits until it decides whether it has any message to deliver to the sink.
- Maximum Buffer Size or K, which is the maximum number of messages that the node can aggregate into a single message.

Here I would first like to illustrate how we got the best ERT, worst ERT and the average ERT that we use to do comparisons of different values of the hyperparameters.

```
00:04.205   ID:2    I got the message as :-  0:Mess4
00:04.207   ID:2    Counter updated to - 1
00:04.208   ID:4    I got the message as :-  0:Mess4
00:04.210   ID:4    Counter updated to - 1
00:07.221   ID:2    I got the message as :-  1:Mess3
00:07.222   ID:2    Counter updated to - 2
00:07.224   ID:4    I got the message as :-  1:Mess3
00:07.225   ID:4    Counter updated to - 2
00:08.360   ID:8    New Timer Set
00:08.525   ID:2    Message Sent on Timer, Counter = 0
00:08.526   ID:2    New Timer Set
00:08.526   ID:2    Sending
00:08.529   ID:2    #L 1 1
00:08.538   ID:6    New Timer Set
00:08.637   ID:4    Message Sent on Timer, Counter = 0
00:08.638   ID:4    New Timer Set
00:08.639   ID:4    Sending
00:08.642   ID:4    #L 1 1
00:08.669   ID:1    New Timer Set
00:08.689   ID:7    New Timer Set
00:08.844   ID:10   New Timer Set
00:08.921   ID:1    Sink got message from 4.0, seqno 0, hops 1: len 16 '0:Mess4+1:Mess3'
00:08.996   ID:9    New Timer Set
00:09.002   ID:5    New Timer Set
00:09.048   ID:1    Sink got message from 2.0, seqno 0, hops 1: len 16 '0:Mess4+1:Mess3'
```

The above is a snippet from the output of one of the permutations of the hyperparameters. Here if we only consider the snippet, then Mess4 from Node 2 forms the case of worst ERT.

Time when Node 2 received Mess4 - 4.205 (T1)
Time when Sink received Mess4 - 9.048 (T2)

Hence Worst ERT = T2 - T1
= 4.843s

Again if we only consider the snippet, then Mess3 from Node 4 forms the case of best ERT. We can note the timestamp for each of the event of message being received and sent and hence calculate the Best ERT as:

Time when Node 4 received Mess3 - 7.224 (T1)
Time when Sink received Mess3 - 8.921 (T2)

Hence Best ERT = T2 - T1
= 1.697s

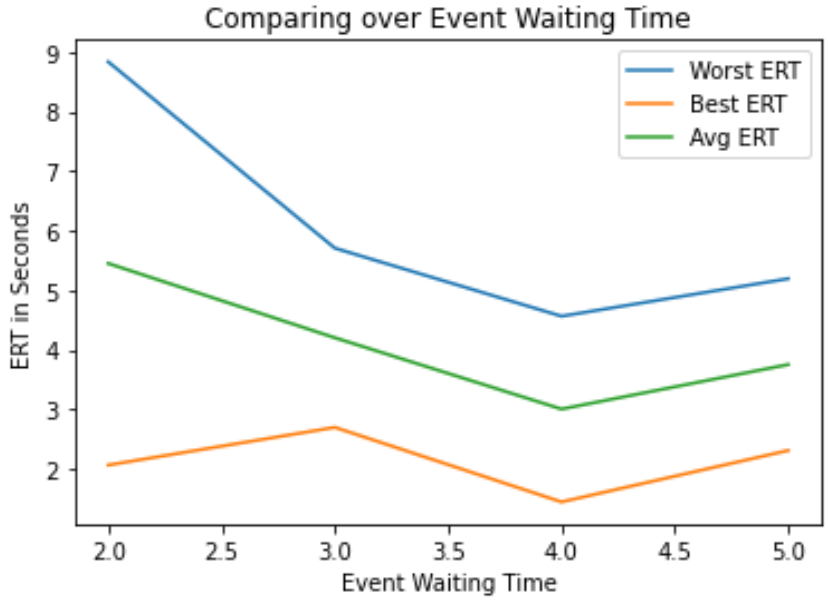
Now to calculate the average ERT for the above snippet we can just take the average of the ERT for all the messages from each of the node, and that would be the major parameter that we should be looking for, since Worst ERT and Best ERT may be misleading, like in the case that we forward an event message as soon as we get, in which case we may get a very short Best ERT but the number of messages being transmitted would increase.

| | | |
|---|---|---|
| 00:08.921 | ID:1 | Sink got message from 4.0, seqno 0, hops 1: len 16 '0:Mess4+1:Mess3' |
| 00:08.996 | ID:9 | New Timer Set |
| 00:09.002 | ID:5 | New Timer Set |
| 00:09.048 | ID:1 | Sink got message from 2.0, seqno 0, hops 1: len 16 '0:Mess4+1:Mess3' |
| 00:09.187 | ID:3 | New Timer Set |
| 00:16.423 | ID:1 | Sink got message from 4.0, seqno 1, hops 1: len 24 '2:Mess2+3:Mess1+4:Mess4' |
| 00:16.523 | ID:2 | New Timer Set |
| 00:16.672 | ID:1 | Sink got message from 2.0, seqno 1, hops 1: len 24 '2:Mess2+3:Mess1+4:Mess4' |
| 00:16.675 | ID:1 | New Timer Set |

It should also be noted that the number of messages that are aggregated into one towards the sink may also vary, like in the first two messages received by the sink, the algorithm aggregated only two messages as it encountered a timeout, while in the last two messages received by sink has three messages being aggregated into one, as it must have been caused by buffer being full.

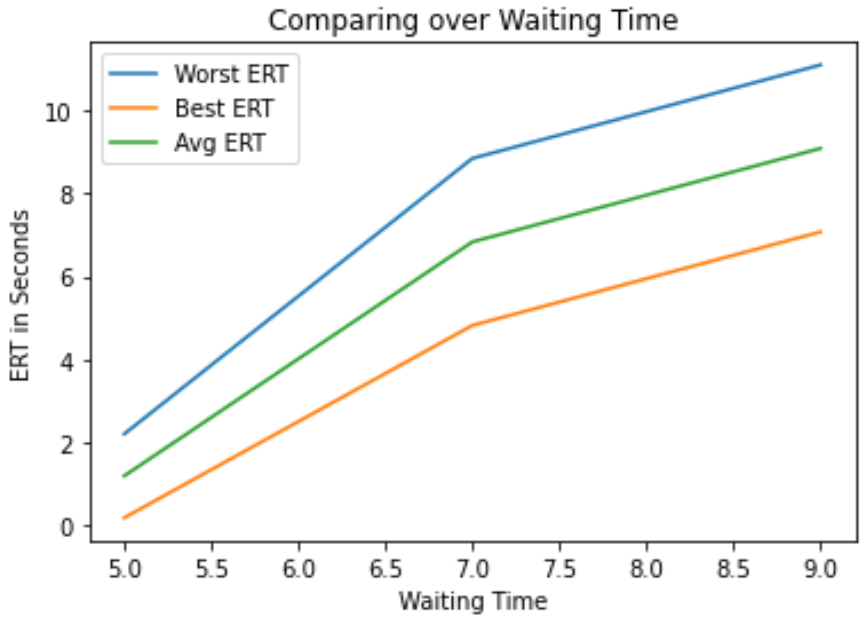
The following is the table where we have varied the event waiting time from 2s to 5s, keeping the waiting time as 7s and the message buffer capacity as 3.

| Event Waiting Time | Best ERT | Worst ERT | Average ERT |
|---|---|---|---|
| 2s | 2.067 | 8.837 | 5.452 |
| 3s | 2.700 | 5.708 | 4.204 |
| 4s | 1.488 | 4.566 | 3.007 |
| 5s | 2.312 | 5.196 | 0.754 |



The above graph shows a pictorial representation of the data generated from varying the event waiting time, being created by the python program being simulated over the files that store the mote outputs for each combination.
The following is the table where we have varied the waiting time from 5s to 9s, keeping the event waiting time as 2s and the message buffer capacity as 3. The waiting time to recall is the maximum time period that the node waits aggregating the incoming messages before forwarding them to the sink, provided the message buffer still has enough memory to accommodate any more messages.

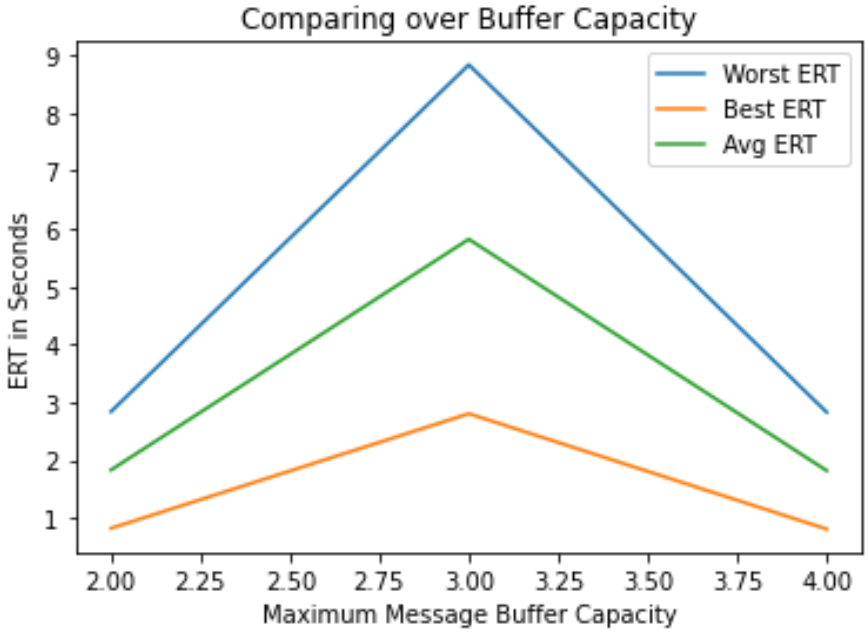
| Waiting Time | Best ERT | Worst ERT | Average ERT |
|---|---|---|---|
| 5 | 0.193 | 2.208 | 1.200 |
| 7 | 4.821 | 8.837 | 6.829 |
| 9 | 7.070 | 11.086 | 9.078 |



The above graph shows a pictorial representation of the data generated from varying the waiting time, being created by the python program being simulated over the files that store the mote outputs for each combination.

The following is the table where we have varied the maximum message buffer size from 2 to 4, keeping the event waiting time as 2s and the node waiting time as 7s. The maximum message buffer capacity to recall is the maximum number of messages that a node can accommodate to make a single message, without requiring multiple aggregated messages.

| Buffer Capacity | Best ERT | Worst ERT | Average ERT |
|---|---|---|---|
| 2 | 0.826 | 2.842 | 1.834 |
| 3 | 2.806 | 8.837 | 5.821 |
| 4 | 0.812 | 2.835 | 1.822 |




The above graph shows a pictorial representation of the data generated from varying the maximum message buffer capacity keeping the other two parameters as constant, being created by the python program being simulated over the files that store the mote outputs for each combination.

# CONCLUSION :-

We realized the need of aggregation of messages at nodes, when they intend to transfer data to the sink node, and hence we explored two different metrics for aggregation, the count of message and time period. We first explored the pros and cons of each of the metrics and then also came up with a hybrid approach to make use of both of the metric so that we have an optimal solution in any case.

Upon implementation and simulation of the proposed approach onto a sample network, we then saved the mote outputs to obtain the Best ERT, worst ERT and the average ERT metric for each combination of hyperparameters. We used these metrics to do our analysis and arrived at the following conclusions:

1. Increasing the Event Waiting Time generally resulted in decrease in ERT, in all the best, worst and average cases. This maybe because there will be more timeout initiated transfer to the sink than buffer overflow. In this case we can say that the message capacity utilized will reduce as we increase the Event Waiting time, as less and less messages will be aggregated.
2. Increasing the Waiting Time for nodes,resulted in an increase in ERT, all in best, average and worst case, but this increase would result in an increase of the message capacity, as more and more individual messages will be aggregated into one.
3. We tried to find a trend between ERT and the Maximum message buffer capacity, by iterating over three values but were unable to come to a conclusion. But as per intuition, it seems that buffer capacity very much depends on the size of each message that is to be transmitted, if we choose a very large buffer capacity, then we may need to divide the aggregate message into multiple messages, which is undesirable. If we keep the buffer capacity too low then it would cause too many capacity full initiated events, as capacity being unity is the case of receive and forward at the same time.

# FUTURE WORKS :-

We had applied all these aggregation methods over the initial node that senses an event only, but it could possibly applied over intermediate nodes of the network too, since that will result in further better utilisation of the resources and energy. We here were unable to do so because that would have needed us to make changes in the Network Layer algorithms rather than making changes in the codes for motes.

This also would have taken into account our second objective that we were asked to achieve as aggregation at intermediate nodes could have been applied using delays and count of messages too.