CSE 322

Network Simulator - 2
Project

SHRED
An Active Queue Management

Submitted By:
Sk. Sabit Bin Mosaddek
Department: CSE
Section: B2
Student ID: **S201805106**

Paper Link: https://core.ac.uk/download/pdf/47187389.pdf

Introduction:

SHRED is a variant of RED (Random Early Detection) which favors short-lived flows. Most of the improvements of the RED algorithm have focused on long-lived flow like FTP. But because of this, short-lived flows have to suffer more. RED even performs worse than droptail for web-only traffic. Short-lived flows have a small congestion window which makes them sensitive to packet drop.

In this project, we are going to implement SHort-lived flow friendly RED which modifies the Gentle RED algorithm. Gentle RED algorithm maintains a Minimum Threshold, a Maximum Threshold, a Maximum Probability of drop. When the queue length exceeds 2 * MAX_TH, the packet is dropped with probability 1. When the queue length is less than MIN_TH, the packet is never dropped. When the Length is between MIN_TH and MAX_TH, the packet is dropped with probability between 0 and MAX_P. Finally when queue length is between MAX_TH and MAX_TH * 2, the packet is dropped with probability between MAX_P and 1.

In SHRED, we change the value of MIN_TH and MAX_P to favor short lived flows. Here, to simplify things, we used the average packet size of a flow to change the MIN_TH and MAX_P.

$$Min\_th\_mod = Min\_th + (Max\_th - Min\_th) * \\ (1 - CurrenFlow\ Average\ Packet\ Size\ /\ Average\ Packet\ Size)$$

Similarly,

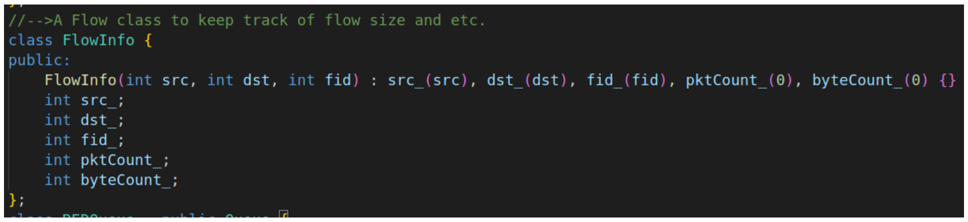
$$MaxP\_mod = MaxP * (Max\_th - Min\_th\_mod)\ /(Max\_th - Min\_th)$$

Topology Under Simulation:

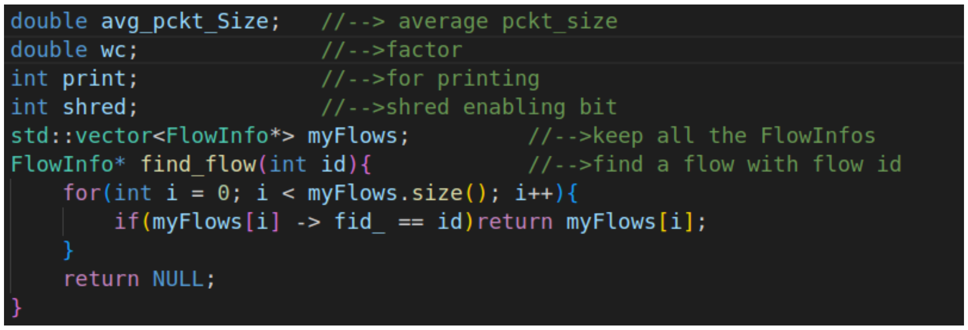
- Wireless 802.11 and 802.15.4
- Random placement of Node
- Random Source
- Random Sink
- Area size 500x500 for 802.11 and 250x250 for 802.15.4
- OmniAntenna
- UDP

Modifications

Added class and variables:

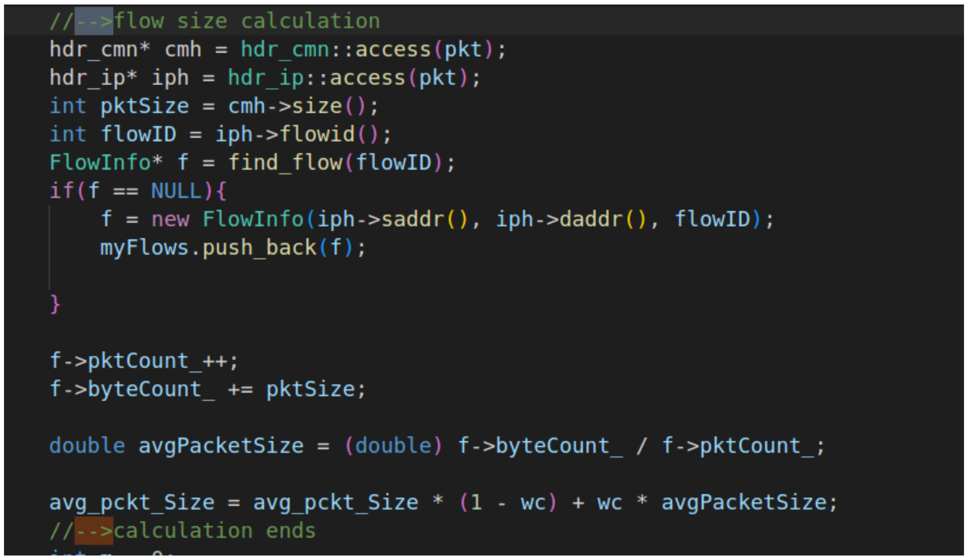


```
//-->A Flow class to keep track of flow size and etc.
class FlowInfo {
public:
    FlowInfo(int src, int dst, int fid) : src_(src), dst_(dst), fid_(fid), pktCount_(0), byteCount_(0) {}
    int src_;
    int dst_;
    int fid_;
    int pktCount_;
    int byteCount_;
};
```

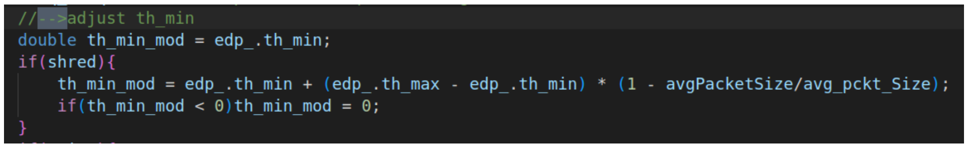


```
double avg_pckt_Size;   //--> average pckt_size
double wc;              //-->factor
int print;              //-->for printing
int shred;              //-->shred enabling bit
std::vector<FlowInfo*> myFlows;         //-->keep all the FlowInfos
FlowInfo* find_flow(int id){            //-->find a flow with flow id
    for(int i = 0; i < myFlows.size(); i++){
        if(myFlows[i] -> fid_ == id)return myFlows[i];
    }
    return NULL;
}
```

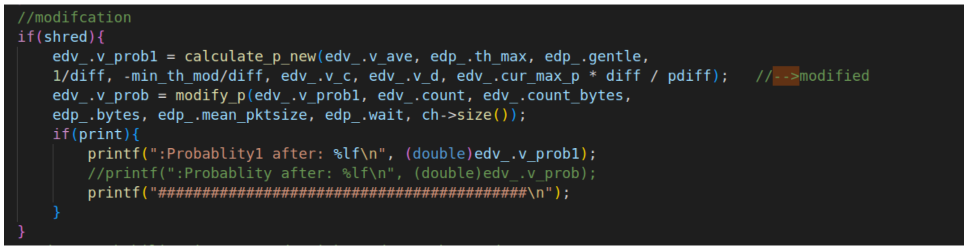
Calculations:



```
    //-->flow size calculation
    hdr_cmn* cmh = hdr_cmn::access(pkt);
    hdr_ip* iph = hdr_ip::access(pkt);
    int pktSize = cmh->size();
    int flowID = iph->flowid();
    FlowInfo* f = find_flow(flowID);
    if(f == NULL){
        f = new FlowInfo(iph->saddr(), iph->daddr(), flowID);
        myFlows.push_back(f);

    }

    f->pktCount_++;
    f->byteCount_ += pktSize;

    double avgPacketSize = (double) f->byteCount_ / f->pktCount_;

    avg_pckt_Size = avg_pckt_Size * (1 - wc) + wc * avgPacketSize;
    //-->calculation ends
```

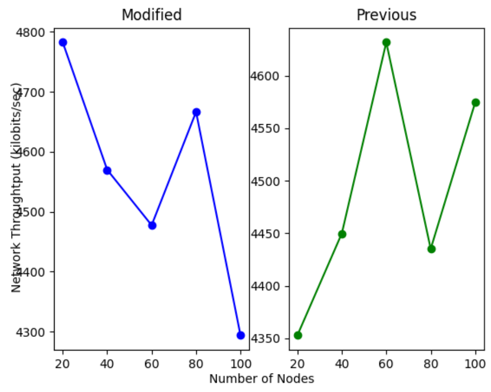


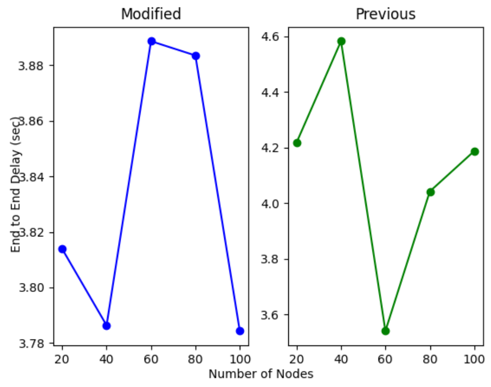
```
//-->adjust th_min
double th_min_mod = edp_.th_min;
if(shred){
    th_min_mod = edp_.th_min + (edp_.th_max - edp_.th_min) * (1 - avgPacketSize/avg_pckt_Size);
    if(th_min_mod < 0)th_min_mod = 0;
}
```

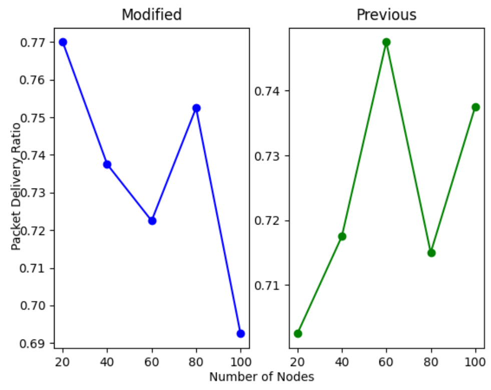


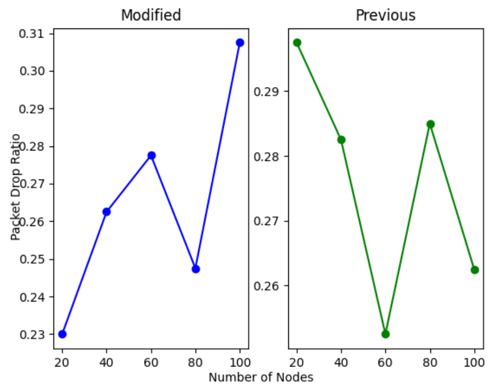
```
//modifcation
if(shred){
    edv_.v_prob1 = calculate_p_new(edv_.v_ave, edp_.th_max, edp_.gentle,
    1/diff, -min_th_mod/diff, edv_.v_c, edv_.v_d, edv_.cur_max_p * diff / pdiff);   //-->modified
    edv_.v_prob = modify_p(edv_.v_prob1, edv_.count, edv_.count_bytes,
    edp_.bytes, edp_.mean_pktsize, edp_.wait, ch->size());
    if(print){
        printf(":Probablity1 after: %lf\n", (double)edv_.v_prob1);
        //printf(":Probablity after: %lf\n", (double)edv_.v_prob);
        printf("##########################################\n");
    }
}
```

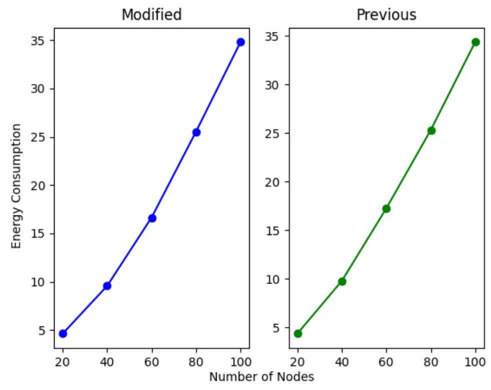
Analysis





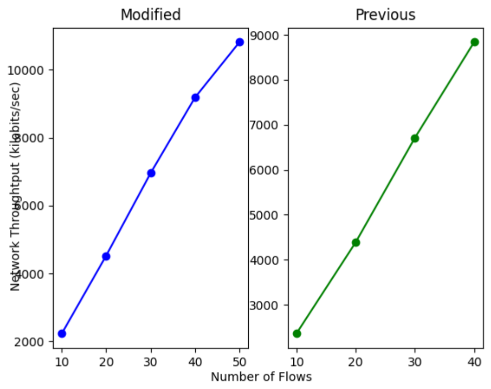


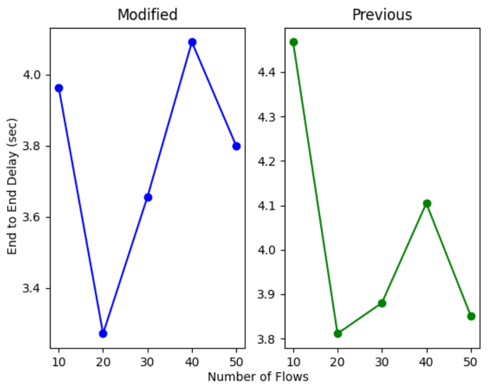


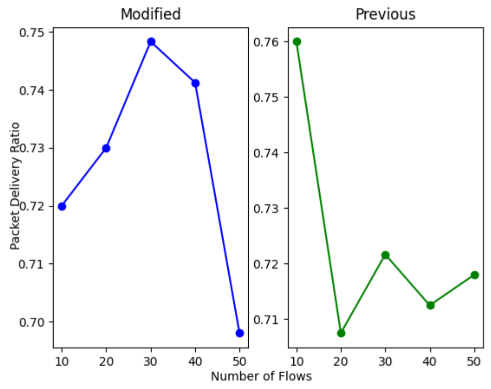


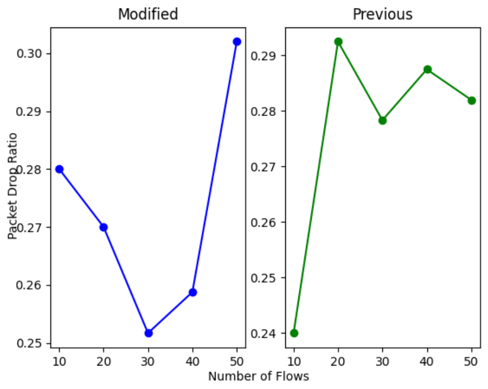
1. Varying Number of Nodes:

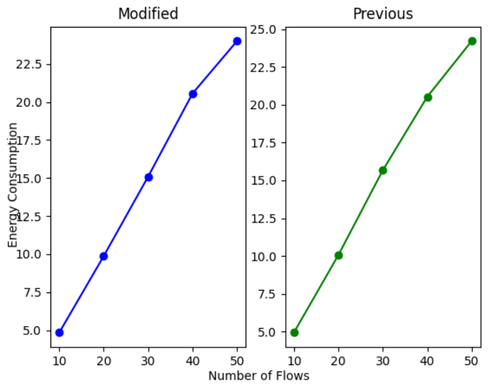
Here we can observe that our modified algorithm tends to work better when the number of nodes is low. As the number of nodes increases, congestion increases. Hence, long lived flows start to dominate and our algorithm works worse than the established RED algorithm when the number of nodes rises.



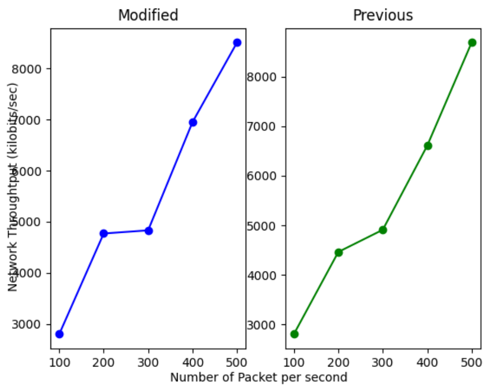


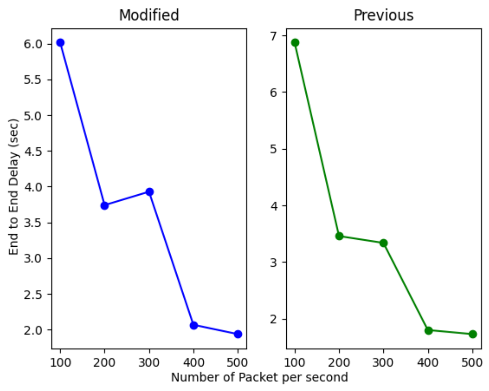


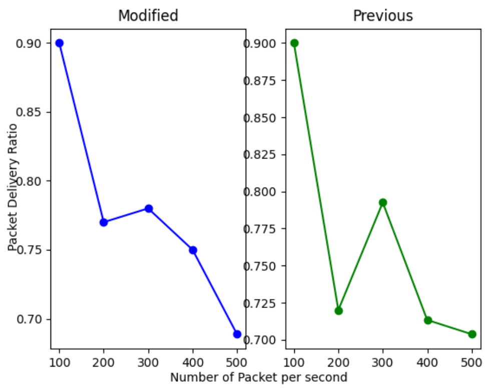


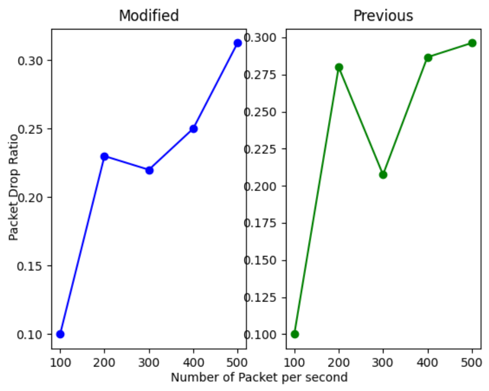


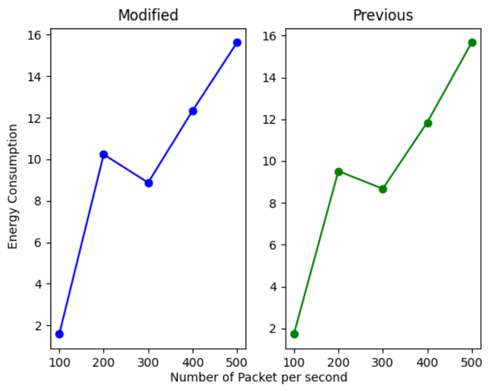
2. **Varying Number of Flows:** Now, our algorithm works well when the number of flows is neither too high, nor too low. If the number of flows is low, our algorithm might drop some long lived flows which were not necessary. For too much flow, traffic congestion allows long-lived flows to dominate which deteriorates our algorithms performance.



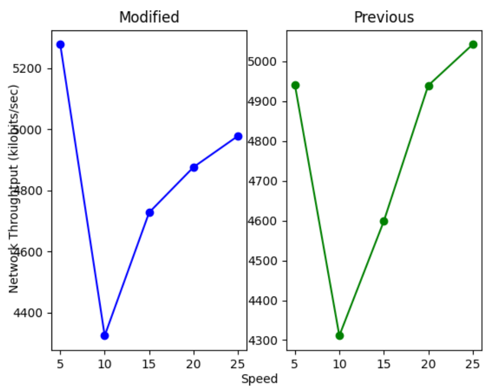


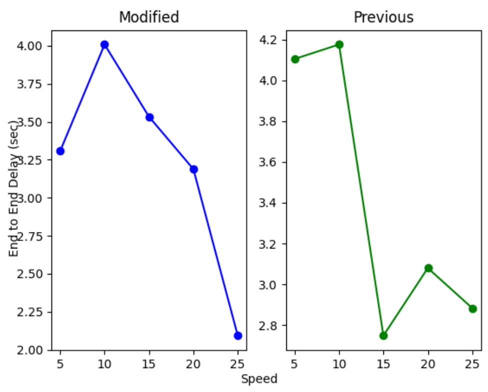


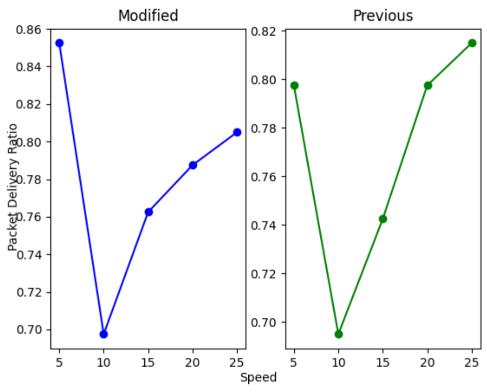


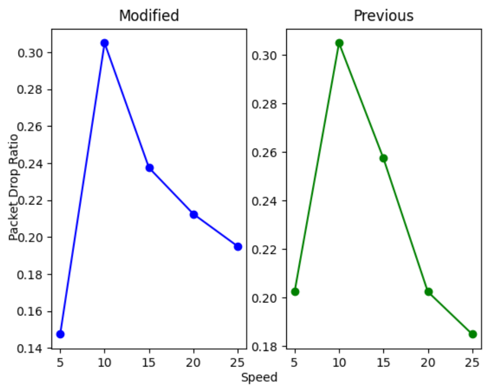


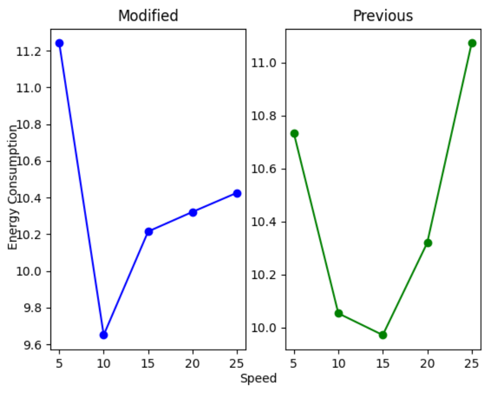
3. **Varying Number of Packets per second :** As the number of packets sent per second increases, our algorithm doesn't improve much but it doesn't work worse. Sometimes it performs slightly better when the packet rate is low. Packet delivery ratio and drop ratio is better than the existing algorithm. If the packet rate remains the same then our algorithm should not affect much as the average number of packets per flow will be similar.











4. **Varying Speed of Nodes:** Speed of nodes also doesn't affect our algorithm significantly. At low speed, the modified algorithm seems to perform slightly better but it consumes more energy. But most of the time the existing algorithm performs better as it should be. Existing algorithm is designed to perform well in general situations. Our algorithm will work better in specific scenarios. Thus, our algorithm does not improve much when we vary node speed.

Now, we can analyze exactly how and where our algorithm has introduced improvement. The Graph below illustrates our algorithm's goal.

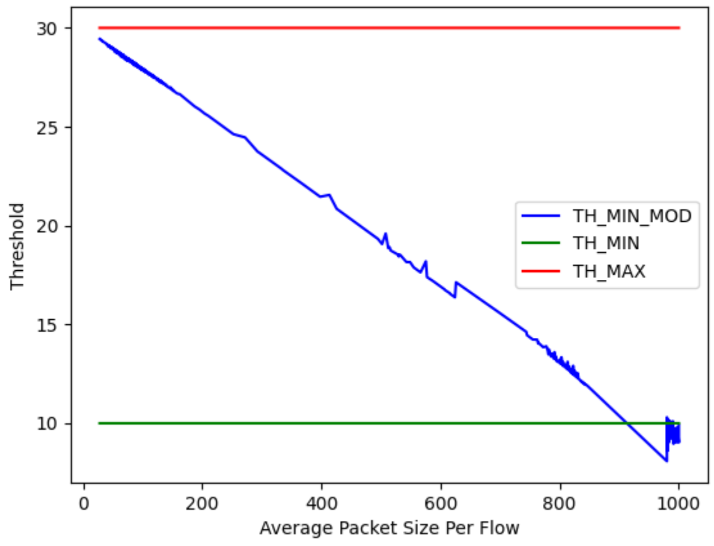


Fig: Thresholds for MAC 802.11

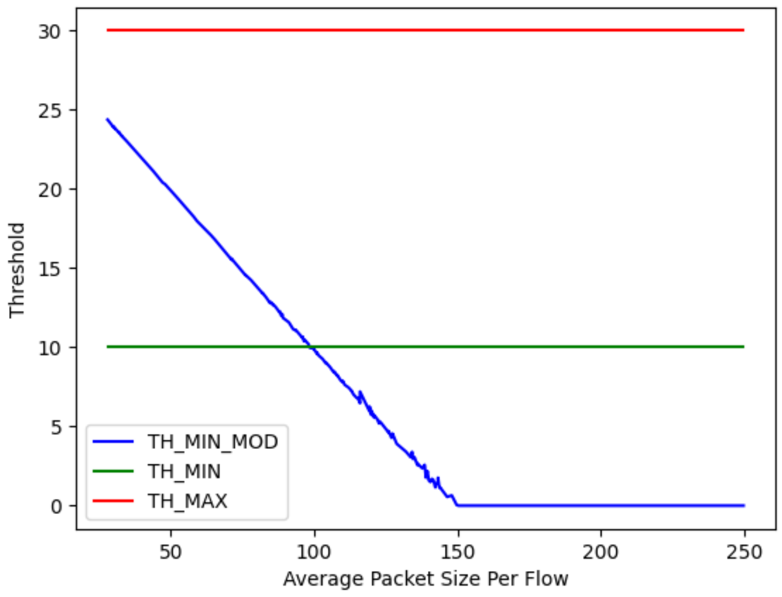


Fig: Thresholds for MAC 802.15.4

Conclusion

Analyzing all the graphs, we can conclude that-

- SHRED performs similar but sometimes better than existing RED algorithm
- Variation in speed or packet rate doesn’t affect the algorithm much.
- If the traffic congestion increases, SHRED starts to perform a bit worse.