Flexible Random Early Detection (FXRED)

Fardin Anam Aungon - 1805087

Introduction

FXRED is a modification of RED algorithm which is used to control the congestion in the network. The main difference between RED and FXRED is that the latter uses a flexible threshold value instead of a fixed threshold value. The threshold value is calculated using the average queue size and the average packet size. The algorithm is described in the paper Flexible Random Early Detection Algorithm for Queue Management in Routers.

Overview of the Algorithm

In the proposed FXRED, the router's queue with finite capacity is divided into four segments (A, B, C, D) via threshold values min_{th}, Δ, max_{th} where $\Delta = \frac{1}{2}(min_{th} + max_{th})$.

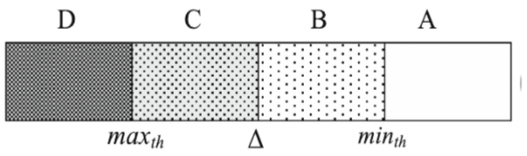

Figure 1: Queue Segments

Just like traditional RED and its other variants, in FXRED, average queue length avg is computed,

$$avg = (1 - w)avg' + wq(t)$$

There are some other parameters that are used in the algorithm:

- $\lambda(t)$: Total data arrival rate in the queue at time t.
- μ: Bandwidth of the bottleneck link.
- $\rho(t)$: $\rho(t) = \lambda(t)/\mu$

Based on traffic load, three states are defined:

$$State - 1 : \rho(t) < 1, State - 2 : \rho(t) \approx 1, State - 3 : \rho(t) > 1.$$

`Probability Function:`

$$p_d = \begin{cases} 0 & avgmin_{th}, \\ 2^{\lfloor k \rfloor}(\frac{avg - min_{th}}{max_{th} - min_{th}})^{\lfloor k \rfloor}.(1 - \epsilon) & min_{th} \leq avg\Delta, \\ 2\epsilon(\frac{avg - \Delta}{max_{th} - min_{th}}) + (1 - \epsilon) & \Delta \leq avgmax_{th}, \\ 1 & avg \geq max_{th} \end{cases}$$

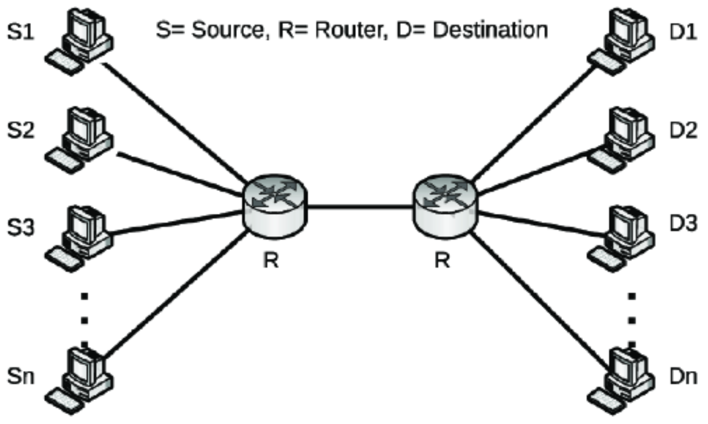

Figure 2: Dumbbell Shaped Topology

$$k = c^{\frac{1}{\gamma}}, c \geq 2, \epsilon = c^{-\gamma} \begin{cases} \gamma 1 & State - 1, \\ \gamma \approx 1 & State - 2, \\ \gamma \geq c & State - 3. \end{cases}$$

Network Topologies Under Simulation

1. Wired

A dumbbell shaped topology is used where all the source nodes are connected to a single router and similarly all the destination nodes are connected to a single router. Both the routers are connected to each other through a link. The topology is shown below.

2. Wireless

Nodes are randomly generated in a square area. For each simulation, a node is selected as source and a number of random nodes are selected as destination. Node configurations are as follows:

- `Node Movement`: Static
- `Wireless MAC Type`: IEEE 802.15.4
- `Routing Protocol`: DSDV
- `Agent`: TCP

- `Application`: FTP

An example of the topology is shown below.

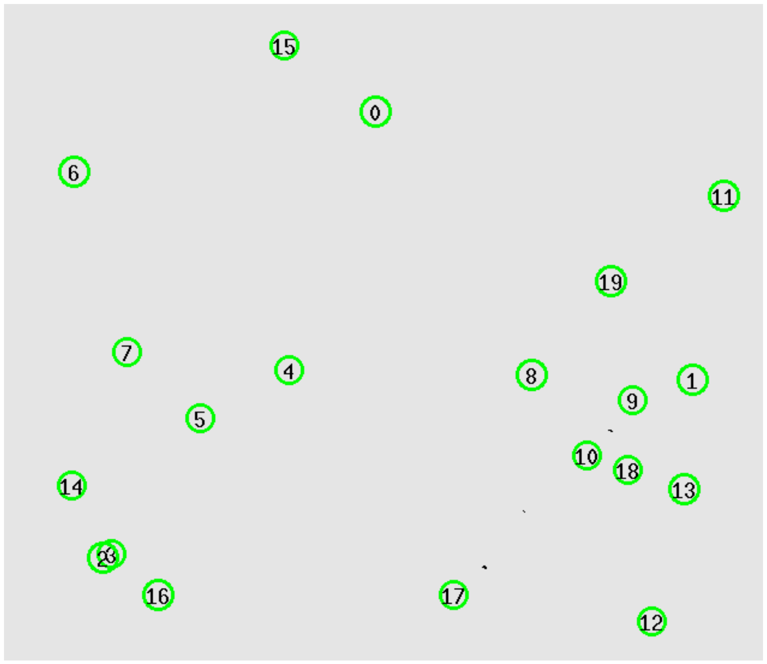

Figure 3: Caption

Parameters Under Variation

To study the effect of different parameters on the performance of the network, the following parameters are varied in the simulation:

Base Parameters:

- `Number of Nodes`: 20
- `Number of Flows`: 10
- `Packets Per Second`: 100
- `Coverage Area`: Tx_Range = 250

Keeping one of the base parameters constant, the other parameters are varied in each simulation as follows:

- `Number of Nodes`: 20, 40, 60, 80, 100

- `Number of Flows`: 10, 20, 30, 40, 50
- `Packets Per Second`: 100, 200, 300, 400, 500
- `Coverage Area`: Tx_Range, 2x Tx_Range, 3x Tx_Range, 4x Tx_Range, 5x Tx_Range

Modifications in NS-2

The following modifications are made in the NS-2 code to implement the FXRED algorithm:

Two files are changes in the `ns2.35/queue` directory:

`red.h`

- In `struct edp` the following two variables are added:

  ```
  /*
  * Parameters for FXRED
  */
  double c; /* FXRED: constant c which should be greater than 2 */
  int fxred; /* FXRED: enable/disable FXRED */
  ```

- To keep track of the number of bytes arrived since the last time a packet was dropped, the following variables are added in `struct edv`

  ```
  int count_bytes;  /* # of bytes since last drop */
    double count_start_time; /* time when the count is 0 */
  ```

  Notice that the `count` variable is already there to keep track of the number of packets arrived since the last time a packet was dropped.

- A new method in `REDQueue:Queue` is added to calculate data arrival rate:

  ```
  /*
  * Calculate data arrival rate
  */
  double getDataArrivalRate();
  ```

`red.cc`

Only the class `REDQueue` is modified. The following changes are made:

- To pass parameter from tcl file to the class, the following code is added in `REDQueue::REDQueue()`:

```
bind("c_", &edp_.c);
bind_bool("fxred_", &edp_.fxred);
```

- The two new parameters are initialized in `REDQueue::initparams()`:

  ```
  edp_.fxred = 0;
  edp_.c = 2.0;

  edv_.count = 0;
  edv_.count_bytes = 0;
  ```

- In `REDQueue::reset()` just below the line `edv_.count = 0;` the following line is added:

  ```
  edv_.count_bytes = 0;
  edv_.count_start_time = Scheduler::instance().clock();
  ```

  These two lines are added everywhere where `edv_.count = 0;` is present.

- The `REDQueue::calculate_p_new()` is modified to calculate the probability function of FXRED.

  ```
  double
  REDQueue::calculate_p_new(double v_ave, double th_max, int gentle,
  double v_a, double v_b, double v_c, double v_d, double max_p)
  {
    double p;

    if (edp_.fxred)
    {
      double miu = link_->bandwidth();
      double rho = getDataArrivalRate() / miu;
      double gamma = 1;
      double err = 0.01;
      double c = edp_.c;

      if (rho < 1.0 - err)
        gamma = 0.15;
      else if (rho > 1.0 + err)
        gamma = c;

      int k = pow(c, 1 / gamma);
      double epsilon = pow(c, -gamma);

      double th_min = edp_.th_min;
  ```

```
    double delta = (th_max + th_min) / 2;

    if (v_ave < th_min)
      p = 0.0;
    else if (th_min <= v_ave && v_ave < delta)
    {
      p = pow((v_ave - th_min) / delta, k);
      p *= 1 - epsilon;
    }
    else if (delta <= v_ave && v_ave < th_max)
    {
      p = 2 * epsilon * (v_ave - delta) / (th_max - th_min);
      p += 1 - epsilon;
    }
    else
      p = 1.0;
  }
  else
  {
    // default RED implementation
  }

  return p;
}
```

- `REDQueue::modify_p()`:

```
double
REDQueue::modify_p(double p, int count, int count_bytes, int bytes,
  int mean_pktsize, int wait, int size)
{
  if (edp_.fxred)
  {
    p /= (1 - count * p);
  }
  else
  {
    // default RED implementation
  }
  return p;
}
```

- Finally, the new definition of the function `REDQueue::getDataArrivalRate()` is added:

```
/*
```

```
* Calculates and returns the data arrival rate in bytes/sec.
*/
double REDQueue::getDataArrivalRate()
{
  double elapsed_time = Scheduler::instance().clock() - edv_.count_start_time;
  if (elapsed_time == 0.0)
    return 0;

  double rate = edv_.count_bytes / elapsed_time;
  return rate;
}
```

Results with Graphs:

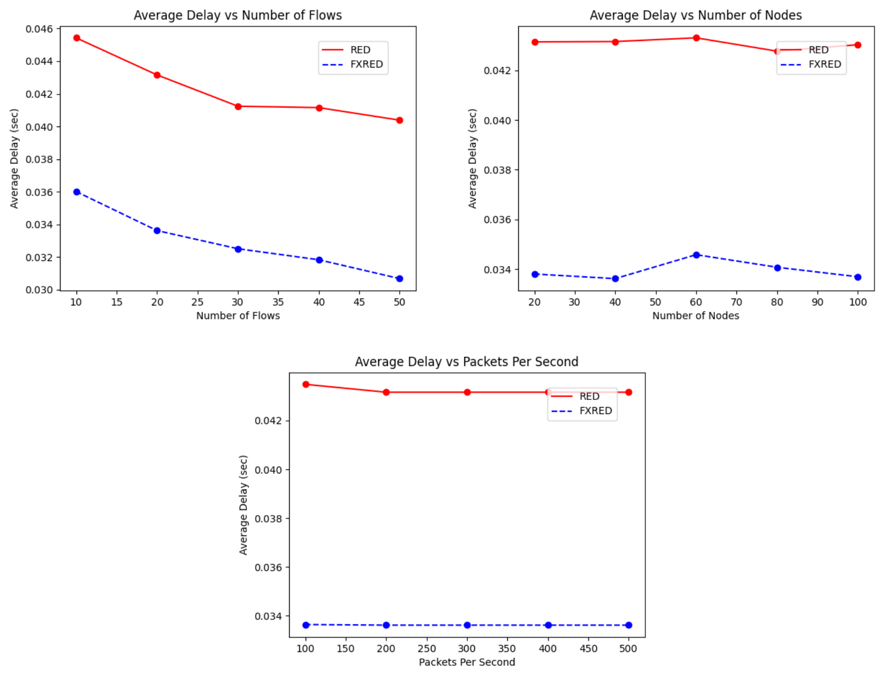

Figure 4: Average Delay in Wired Topology

Summary of Findings:

From the graphs found after analysing wired topology, we can see that the proposed algorithm works better in terms of end to end average delay but it has

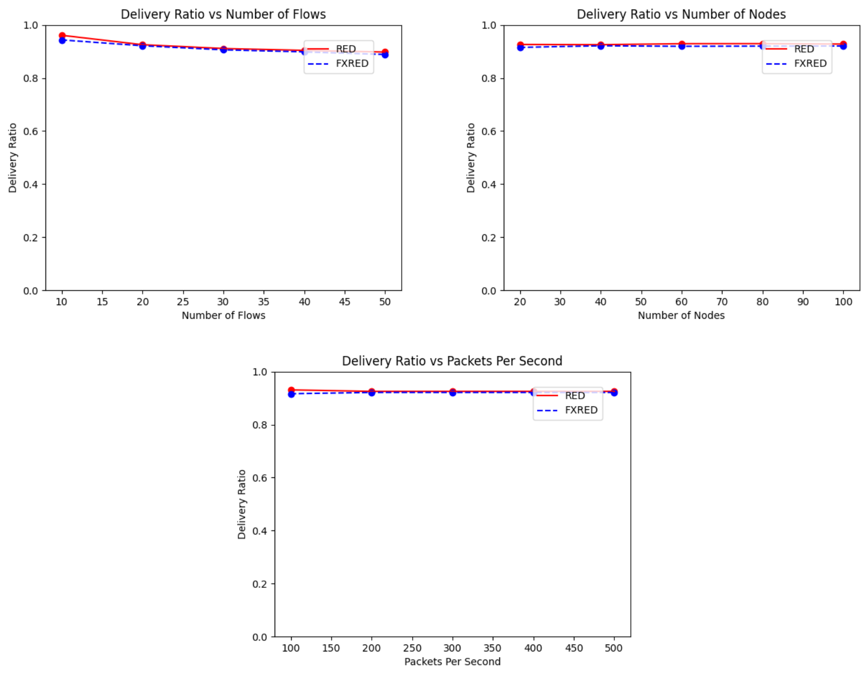

Figure 5: Delivery Ratio is Wired Topology

a higher drop ratio. This is because the proposed algorithm is more aggressive in dropping packets. The proposed algorithm also has a higher throughput than the default RED algorithm. In terms of delivery ratio, both algorithms have a similar performance.

On the other hand, wireless topology suggests quite a different result. The proposed algorithm has a higher end to end average delay than the default RED algorithm. Delivery ratio and throughput are also lower in FXRED. Drop ratio and energy consumption are slightly higher too.

Conclusion:

As we use RED algorithms to control congestion, the dumbbell shaped wired topology is more suitable for simulating the performance of RED algorithms. So, we can rely on the results found in wired topology to conclude that the proposed algorithm is slightly better than the default RED algorithm.

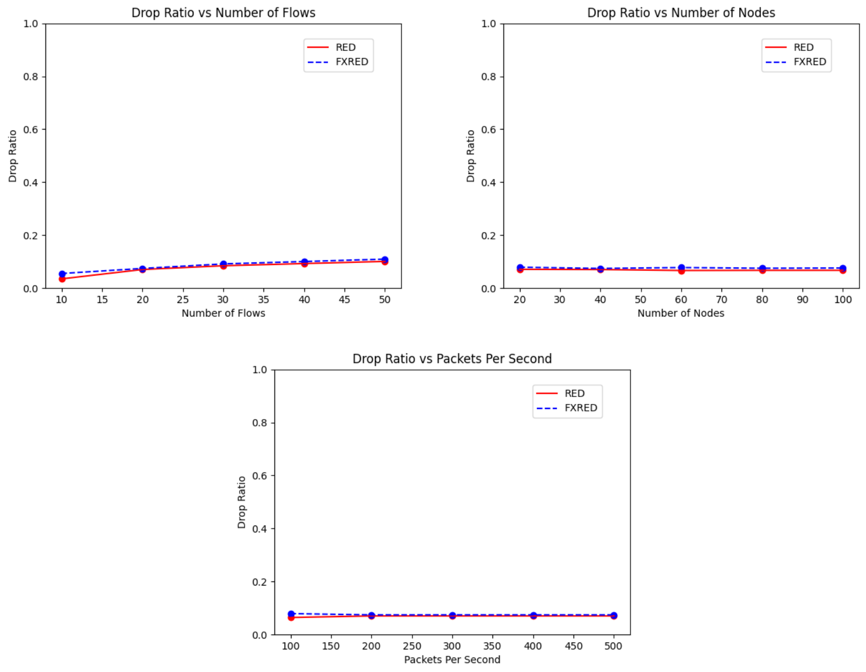

Figure 6: Drop Ratio vs Packets Per Second

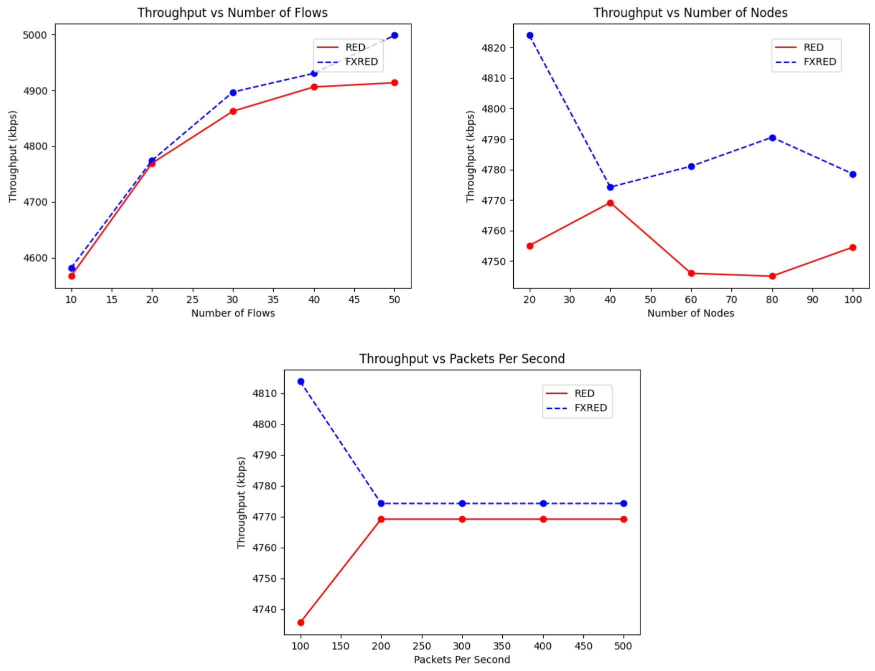

Figure 7: Throughput vs Packets Per Second

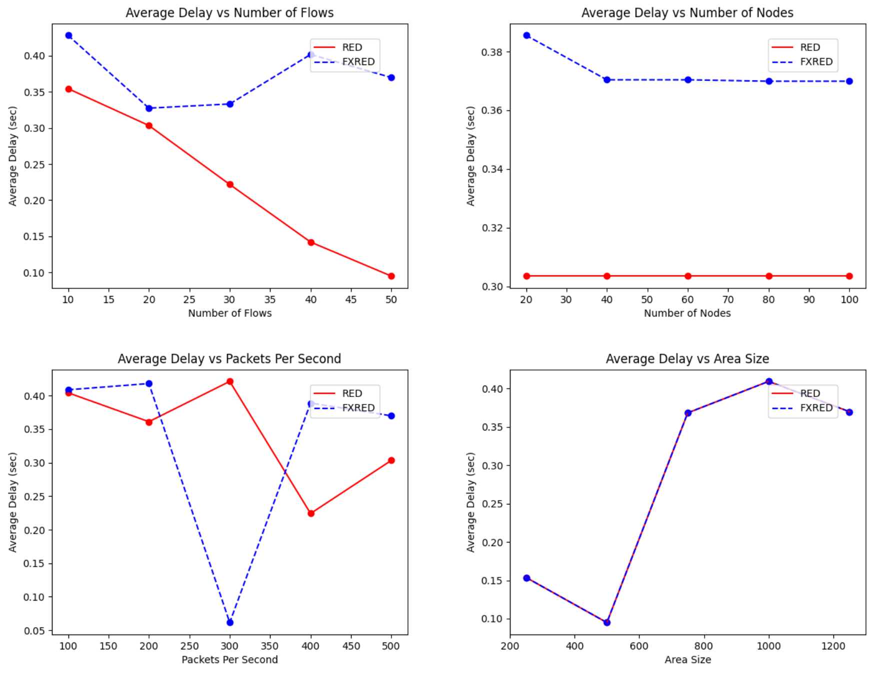

Figure 8: Average Delay in Wireless Topology

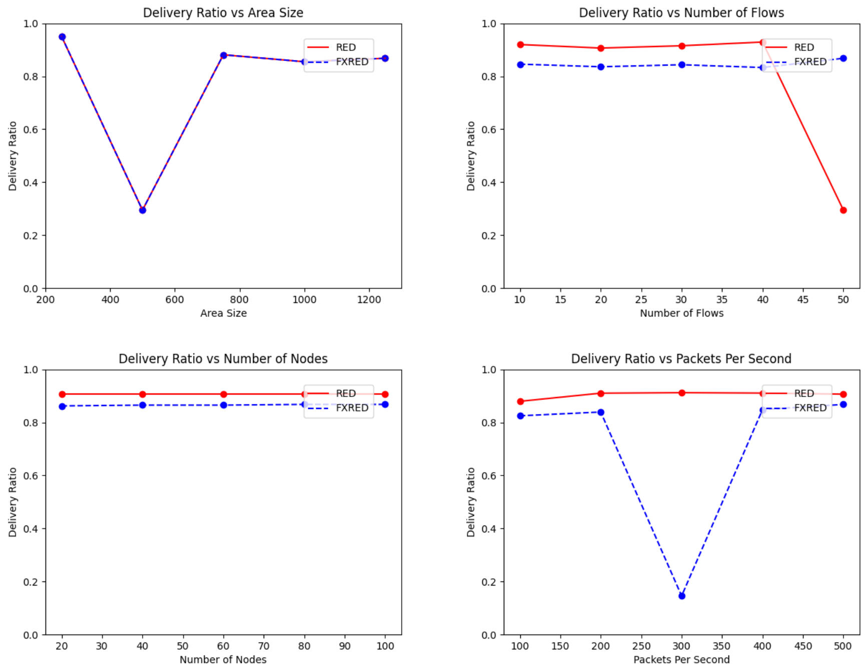

Figure 9: Delivery Ratio in Wireless Topology

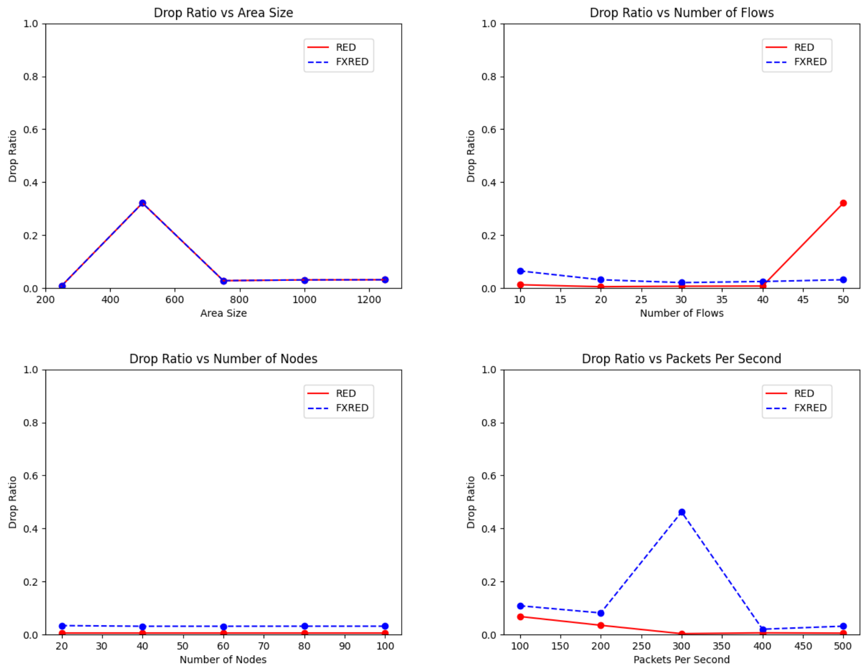

Figure 10: Drop Ratio in Wireless Topology

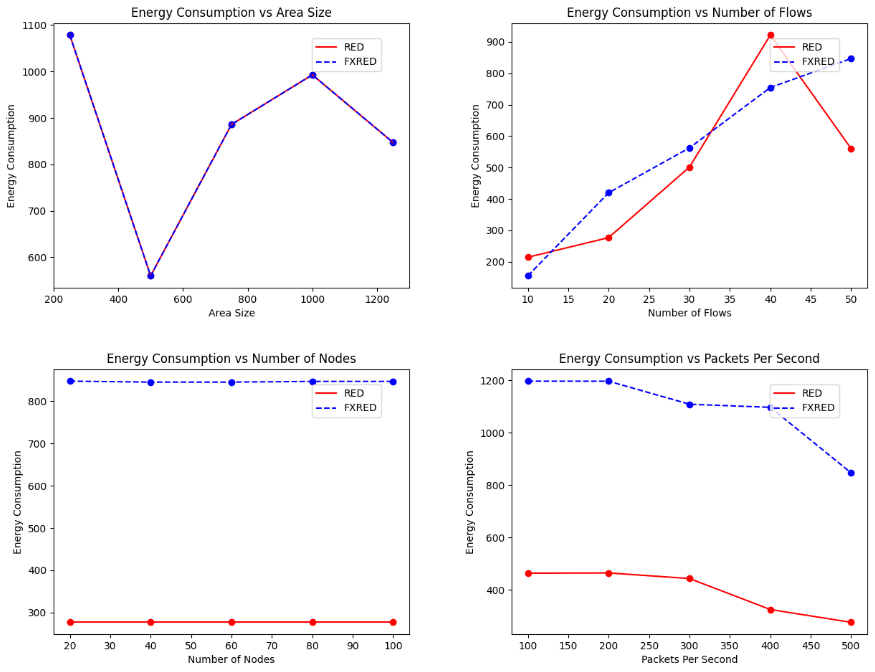

Figure 11: Energy Consumption in Wireless Topology