Bangladesh University of Engineering & Technology

1805004

COURSE

CSE 322

Computer Networks Sessional

Report On (NS2 Project):

EWPHCC - Exponentially Weighted Probabilistic Hybrid Congestion Control for TCP

Author:
Syed Jarullah Hisham
Roll: 1805004
CSE'18 Section A1

Supervisor:
Dr. Md. Shohrab Hossain
Professor
Department of Computer Science and Engineering
Bangladesh University of Engineering and Technology (BUET)

February 27, 2023

Exponentially Weighted Probabilistic Hybrid Congestion Control for TCP

Abstract

Nowadays, the majority of Internet traffic employs the TCP protocol for reliable transmission. However, the standard TCP's performance is inadequate in High Speed Networks (HSN), resulting in underutilization of the core gigabyte links. This issue is rooted in the conservative nature of TCP, particularly in its Additive Increase Multiplicative Decrease (AIMD) phase. In other words, TCP adopts a conservative strategy to prevent network congestion because it cannot accurately determine the network's congestion status. We propose that accurate congestion estimation in the network can resolve this issue by avoiding unnecessary conservatism. Therefore, this paper presents an algorithm that jointly considers packet loss and delay information and employs a probabilistic approach to estimate the congestion status accurately. To assess the performance of the proposed scheme, we conducted extensive simulations using the NS-2 environment. Simulation results show that the proposed algorithm in some cases perform better than the existing algorithm.

Consulted Paper:

PHCC TCP

Authors:

Shahram Jamali
Reza Fotohi
Mir Talebi

Network Topologies Under Simulation

- Wireless 802.11 Mobile Network
- Wired Network
- **Bonus**: Cross Transmission (Wired-Cum-Wireless)
- **Bonus**: Satellite Network

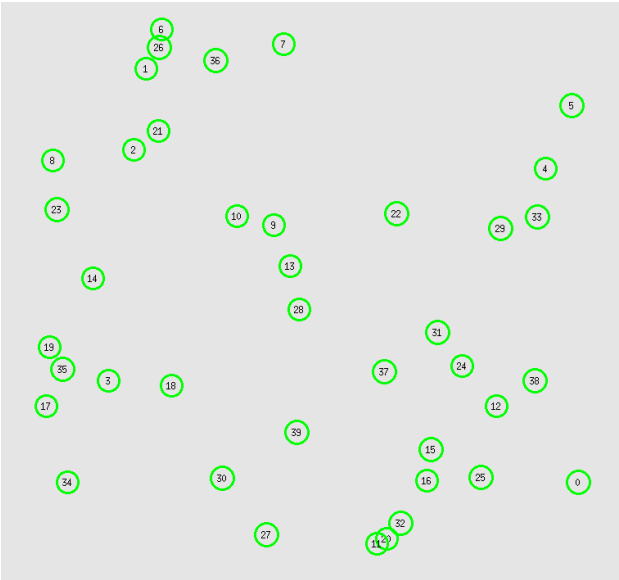


Figure 1: Wireless Random

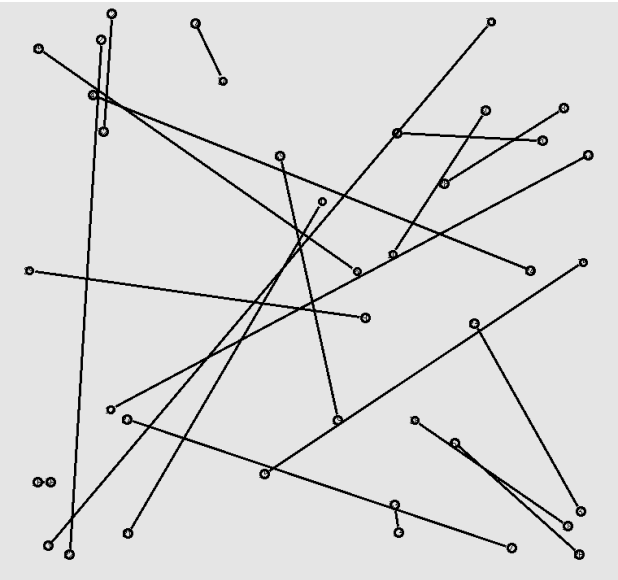


Figure 2: Wired Random

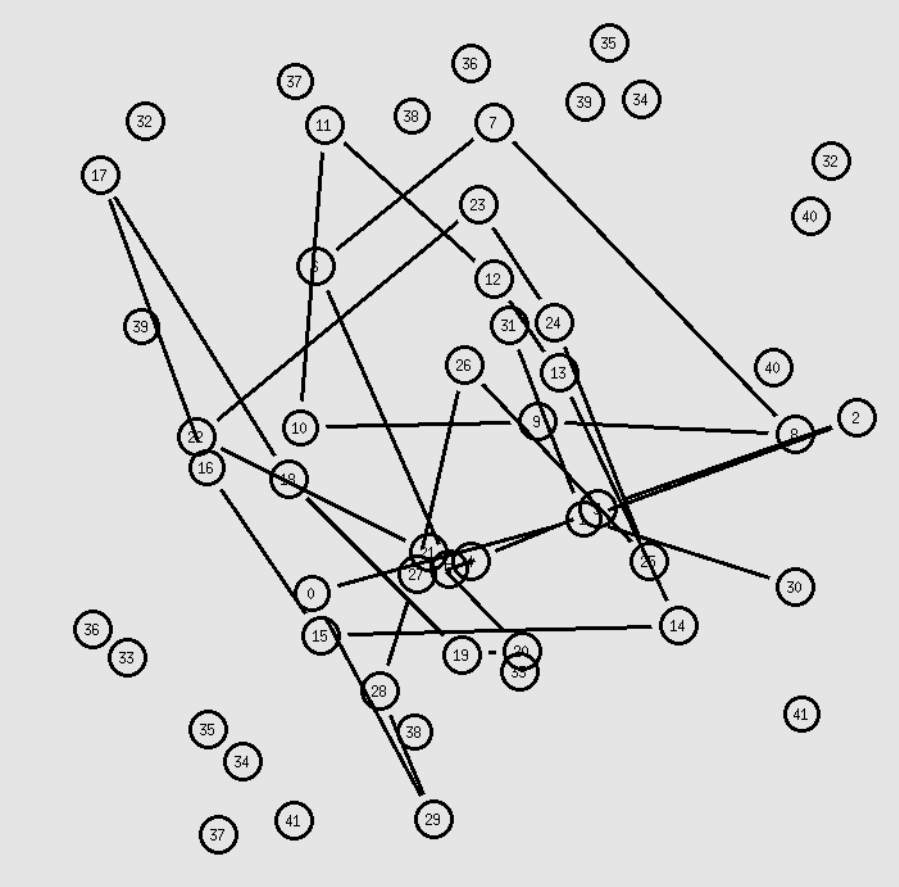


Figure 3: Wired-Cum-Wireless Random

Figure 4: Topology Examples

Parameters Under Variation

The simulation was conducted with the following parameters varied:

- Number of nodes: 20, 40, 60, 80, and 100 [for all 4 networks]
- Number of flows: 10, 20, 30, 40, and 50 [not for satellite]
- Number of packets per second: 100, 200, 300, 400, and 500 [not for satellite]
- Speed of nodes: 5 m/s, 10 m/s, 15 m/s, 20 m/s, and 25 m/s [Wireless and Wired-Cum-Wireless]
- Altitude(in m): 780, 840, 900, 960, 1020 [for only satellite]

The results were measured in terms of-

- Network throughput
- Average end-to-end delay
- Packet delivery ratio
- Packet drop ratio
- Energy Consumption per packet (for only wireless)
- Energy Consumption per byte (for only wireless)

The results were analyzed to evaluate the performance of the modified TCP-Vegas protocol under various network conditions.

Modifications made in the simulator

Architecture of the Probability Model:

In the first step, calculate delay-based probability and loss-based probability from delay and loss information, and in the second step, the window control strategy calculate from the exponential estimates of the delay and loss probability is realized by the joint congestion control component.

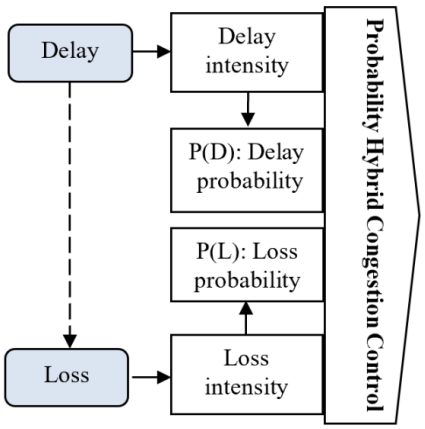


Figure 5: EWPHCC Architecture

Formulas Used:

$$WindowSize = \left(f\left(1-P(D),1-P(L)\right)\right)$$

$$P(D^{est}) = \Phi * P(D^{old}) + (1-\Phi) * rtt_{prev}$$

$$P\left(L^{est}\right) = \Phi * P\left(L^{old}\right) + (1-\Phi) * \frac{loss\_count}{elapse\_time}$$

$$W_i^{est}(D) = \left(1-P\left(D_i^{est}\right)\right).W\left(D_i^{tar}\right)$$

$$W_i^{est}(L) = \left(1-P\left(L_i^{est}\right)\right).W\left(L_i^{tar}\right)$$

$$W\left(D_i^{tar}\right) = \left(D_i * \frac{\alpha_i}{D_{queue}}\right)$$

$$W_i^{est}(D,L) = \sqrt{\frac{(W_i^{est}(D))^2 + (W_i^{est}(L))^2}{2}}$$

$$D_{\min}^{tr} = \left(\frac{ssthresh}{bw_{expect}}\right)$$

$$BFS = \left(\left(bw_{expect} - bw_{actual}\right) * D_{\min}\right)$$

$$D_{delay}^{tr} = \left(\left(seqno - lastack\right) * \left(\frac{Delay - D_{\min}}{BFS}\right)\right)$$

Figure 6: Equations used for this project

Phases of Congestion Control:

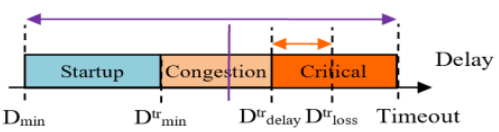


Figure 7: Three Phases of congestion control

Modified Codes:

at **VegasTcpAgent** of **tcp.h**

```
double v_impact_factor_;   // impact factor for EWMA method

double v_prev_rtt_;     // previous rtt value

// modified part for delay-based congestion control part
double v_prev_delay_prob_;  // previous mean value of delay probability obtained from rtt
double v_target_delay_window_;  // target delay window for EWMA method

// modified part for loss-based congestion control part
int v_loss_count_;          // loss count for EWMA method
double v_est_loss_prob_;    // loss probability for EWMA method
double v_target_loss_window_;   // target loss window for EWMA method
```

Figure 8: VegasTcpAgent modification

at **vegas-expire** method of **tcp-vegas.cc**

```
// modified part for loss
v_loss_count_++;

v_est_loss_prob_ = v_impact_factor_ * v_est_loss_prob_ +
                   (1.0 - v_impact_factor_) * double(v_loss_count_ / elapse);
// printf("est_loss = %lf\n", v_est_loss_prob_);

// v_target_loss_window_ = cwnd_;

// end of modified part for loss
```

Figure 9: vegas-expire modification

at **Recv** method of **tcp-vegas.cc**

- at count rtt > 0

```
if(v_cntRTT_ > 0) {
    rtt = v_sumRTT_ / v_cntRTT_;

    /* Modification Start */

    // estimate queue delay
    delay_i = (double) v_rtt_ * 2.0 / cwnd_;
    delay_queue = rtt - v_baseRTT_;
    delay_queue = delay_queue < 0 ? -delay_queue : delay_queue;
    // printf("delay_queue = %lf\n", delay_queue);

    /* Modification End */
```

Figure 10: when count rtt > 0

- at congestion avoidance

```
// threshold min value of delay
double min_threshold = ssthresh_ / expect;
// printf("min_threshold = %lf\n", min_threshold);

double delay = (double) 2.0 * v_rtt_ / cwnd_;
// printf("delay = %lf\n", delay);

double min_delay = (double) 2.0 * v_baseRTT_ /  cwnd_;
// printf("min_delay = %lf\n", min_delay);

double bfs_delay = (double) (expect - v_actual_) * min_delay;
// printf("bfs_delay = %lf\n", bfs_delay);

double loss = (double) v_rtt_ * v_loss_count_ /  cwnd_;
// printf("loss = %lf\n", loss*100.0);

double min_loss = (double) v_baseRTT_ * v_loss_count_ /  cwnd_;
// printf("min_loss = %lf\n", min_loss);

double bfs_loss = (double) (expect - v_actual_) * min_loss;
// printf("bfs_loss = %lf\n", bfs_loss);

int select_phase = (bfs_delay != 0 && bfs_loss != 0 && delay_queue != 0 &&
                    delay >= min_threshold) ? 1 : 0;
```

```
if(select_phase == 0) {
    // printf("normal\n");

    cwnd_ *= 2.0;
} else {

    // printf("hello\n");

    double threshold_delay = double(t_seqno_-last_ack_) * ((delay - min_delay) / bfs_delay);
    threshold_delay = threshold_delay < 0 ? -threshold_delay : threshold_delay;
    // printf("threshold_delay = %lf\n", threshold_delay);

    double threshold_loss = double(t_seqno_-last_ack_) * ((loss - min_loss) / bfs_loss);
    threshold_loss = threshold_loss < 0 ? -threshold_loss : threshold_loss;
    // printf("threshold_loss = %lf\n", threshold_loss);

    double max_threshold = threshold_delay > threshold_loss ? threshold_delay : threshold_loss;
    // printf("max_threshold = %lf\n", max_threshold);
```

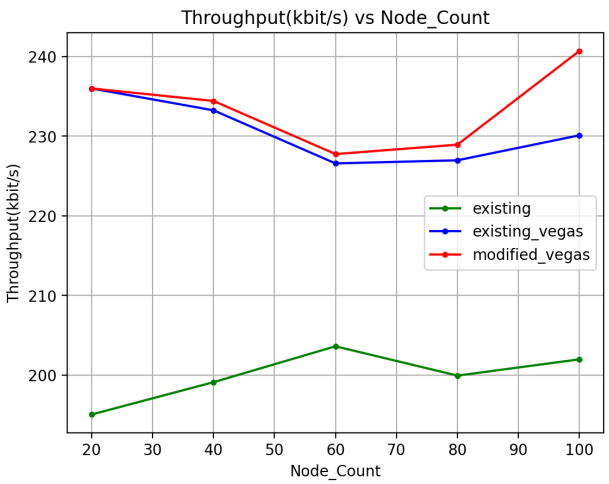
```
if(delay < max_threshold) {   // congestion control phase
    // printf("congestion control\n");

    // estimate delay probability using EWMA method
    double est_delay_prob = v_impact_factor_ * v_prev_delay_prob_ + (1.0 - v_impact_factor_) * v_prev_rtt_;
    // printf("est_delay = %lf\n", est_delay_prob);

    v_prev_delay_prob_ = est_delay_prob;
    v_prev_rtt_ = v_rtt_;

    // printf("window size = %lf\n", cwnd_.value());

    // calculate target window size
    double alpha = cwnd_ / v_rtt_;
    // printf("alpha = %lf\n", alpha);

    v_target_delay_window_ = (delay_i * alpha) / delay_queue;
    // printf("v_target_delay_window_ = %lf\n", v_target_delay_window_);

    double est_window_incr_delay = (1.0 - est_delay_prob) * v_target_delay_window_;
    est_window_incr_delay = est_window_incr_delay < 0 ? -est_window_incr_delay : est_window_incr_delay;

    // printf("prev_est_loss = %lf\n", v_est_loss_prob_);
    double est_window_incr_loss = (1.0 - v_est_loss_prob_) * v_target_loss_window_;
    est_window_incr_loss = est_window_incr_loss < 0 ? -est_window_incr_loss : est_window_incr_loss;

    double est_window_incr = sqrt((est_window_incr_delay * est_window_incr_delay + est_window_incr_loss * est_window_incr_loss) / 2.0);
    // printf("sqrt_avg = %lf\n", est_window_incr);
```

```
    double estimated_window = cwnd_ + est_window_incr;
    // printf("new window = %lf\n", estimated_window);

    // update window size
    cwnd_ = estimated_window;
    // v_newcwnd_ = cwnd_;

} else if(delay >= max_threshold && delay < v_timeout_) {   // critical phase
    // printf("critical\n");

    cwnd_ -= v_alpha_;
    if(cwnd_< 2.0) cwnd_ = 2.0;

} else {                    // timeout or loss phase
    // printf("timeout\n");
    cwnd_ /= 2.0;
}
```

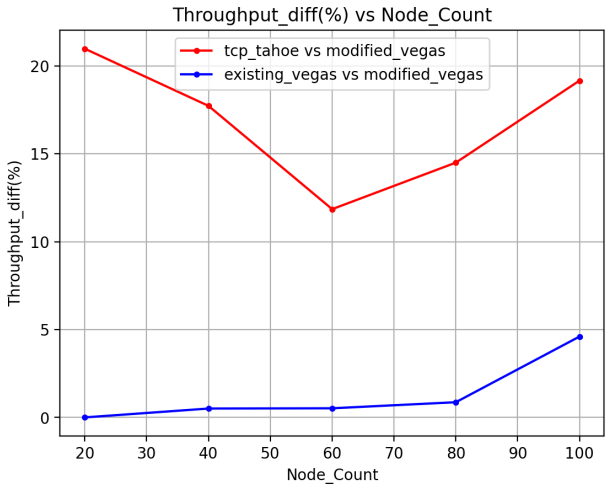
Figure 11: recv modified codes

Graphs(Wireless)

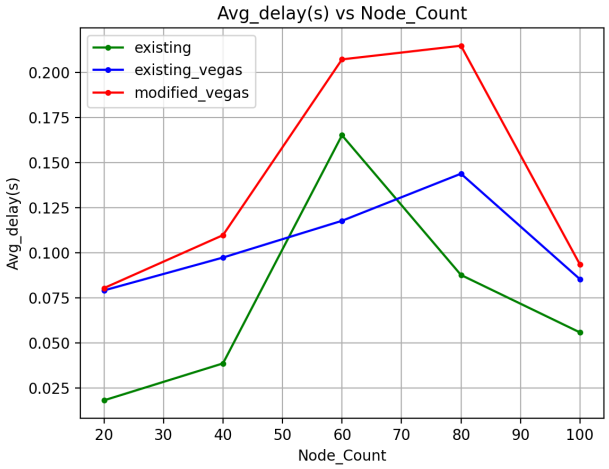
Varying number of nodes



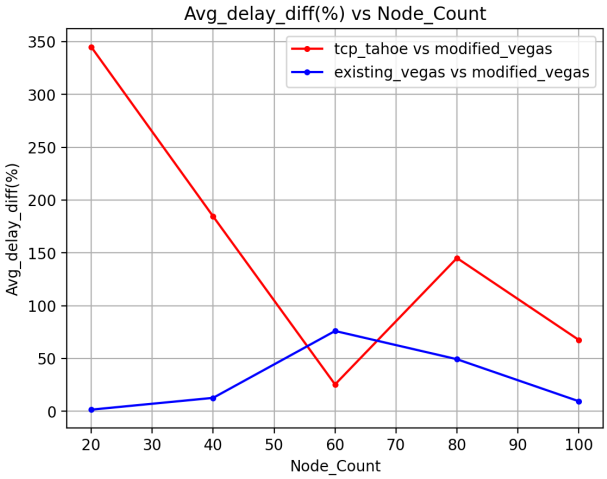
(a) throughput vs Node



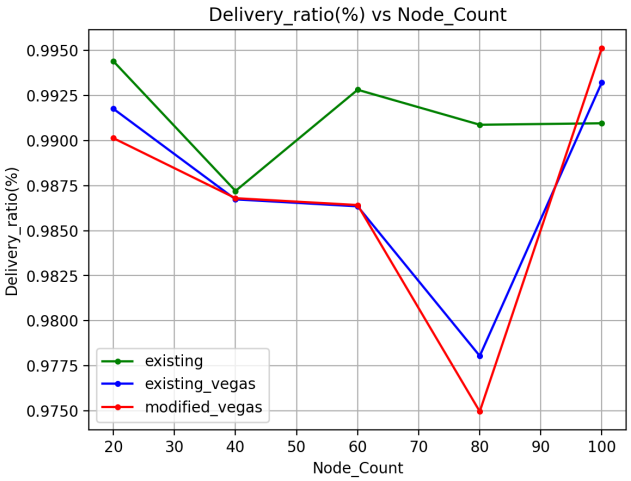
(b) throughput diff vs Node



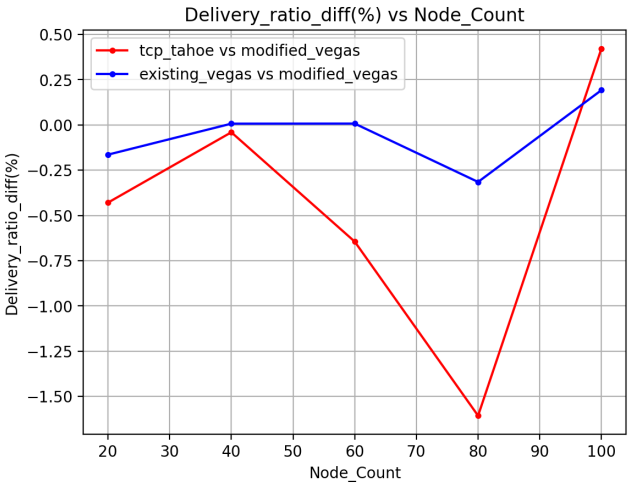
(c) avg delay vs Node



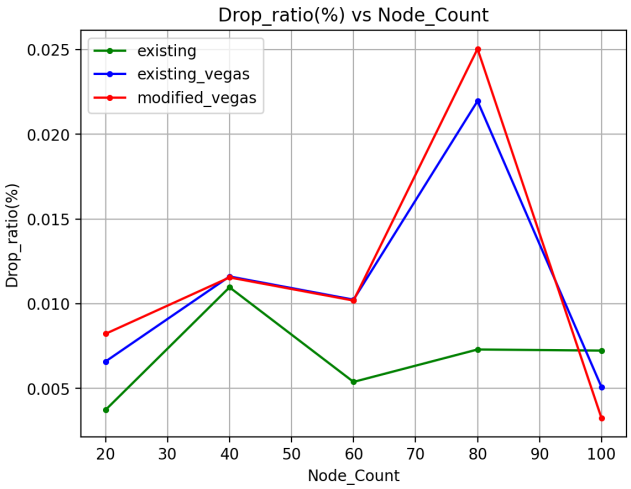
(d) avg delay diff vs Node



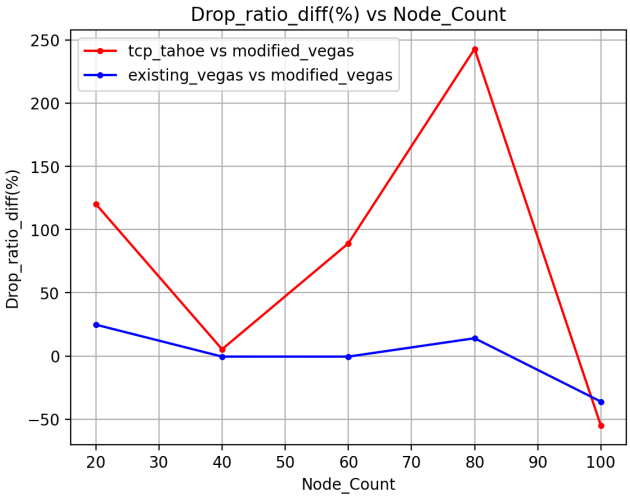
(e) delivery ratio vs Node



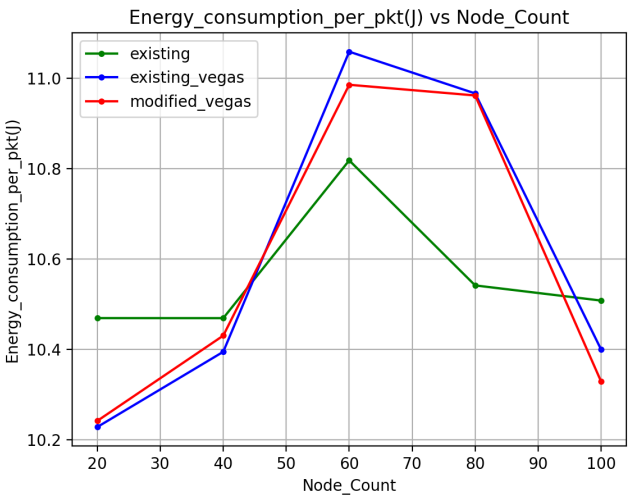
(f) delivery ratio diff vs Node



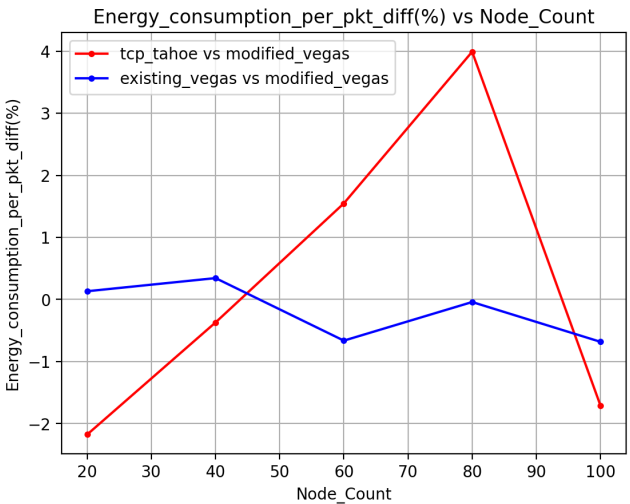
(g) drop ratio vs Node



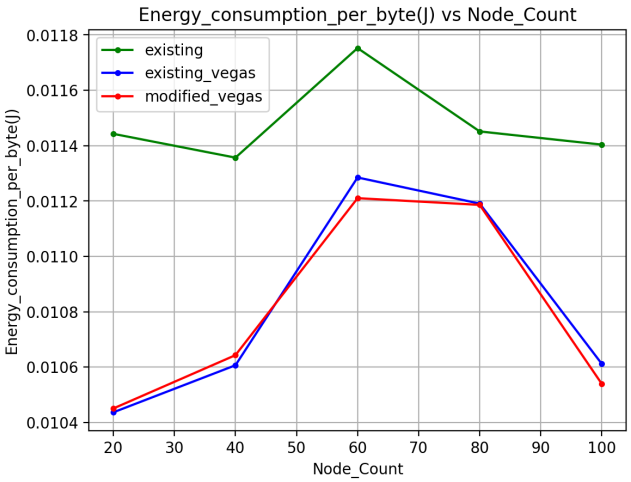
(h) drop ratio diff vs Node



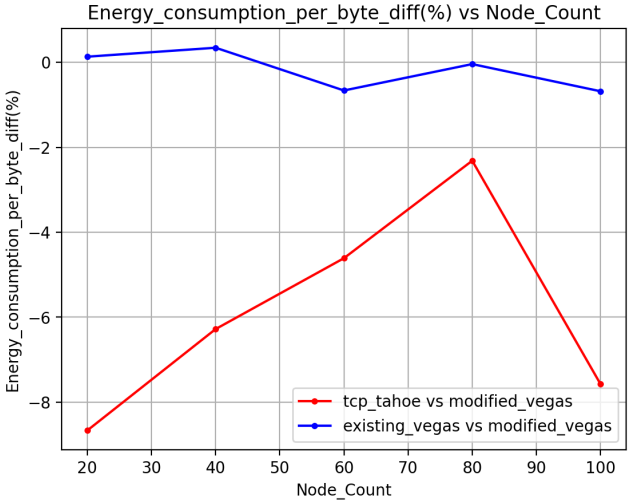
(i) energy pkt vs Node



(j) energy pkt diff vs Node



(k) energy byte vs node

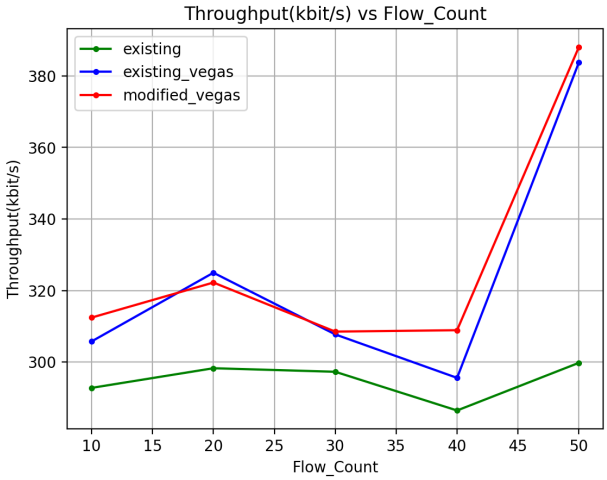


(l) energy byte diff vs node

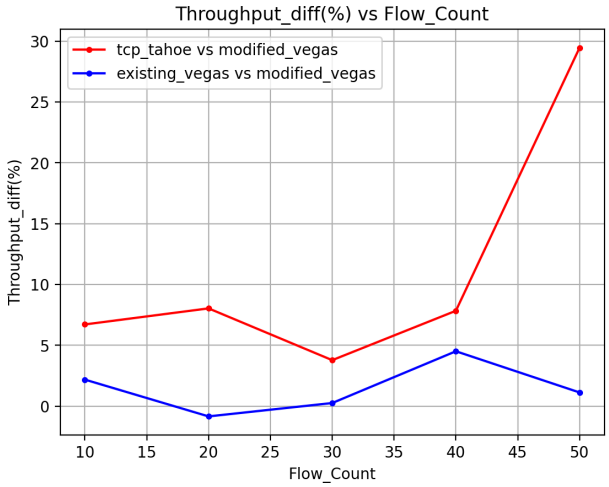
The observations we get from these graphs are -

- In the network throughput, the modified vegas algorithm works slightly better than existing algorithm(<=5%) and much better than the default tcp tahoe implementation(<=20%)
- In the average delay, the modified vegas performed worst among the three, the reason can be excessive and complex calculation of the congestion window which takes more time than the existing algorithm execution.
- In the delivery ratio, the modified vegas performed slightly worse than the existing ones.
- Energy consumption decreases for the modified vegas algorithm which is a good performance improvement than the existing ones.
- But one thing to note here that the graphs are not so much well formed even after experimenting with multiple variations. For e.g: for node count 80, delivery and drop ratio showed an unusual spike, this can be of various reasons. As randomized topology used, maybe in that case the topology introduced extra traffic in the network which is unusual.

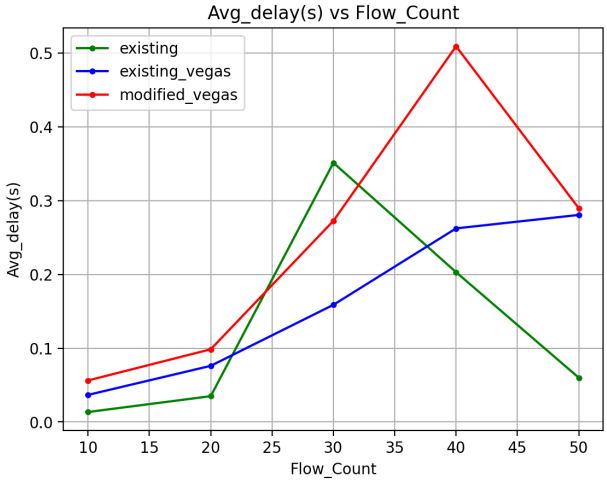
Varying number of flows



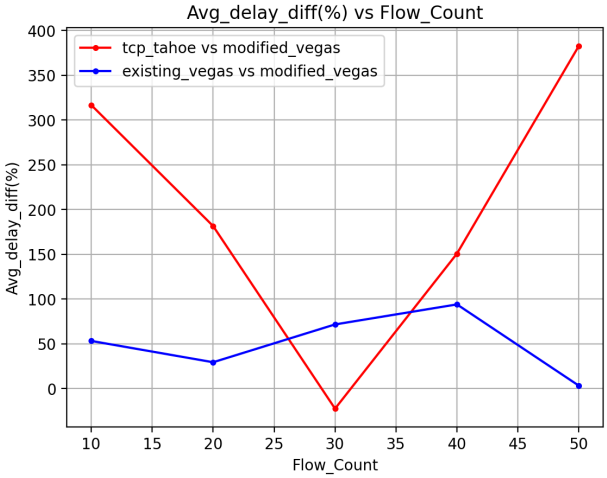
(m) throughput vs flow



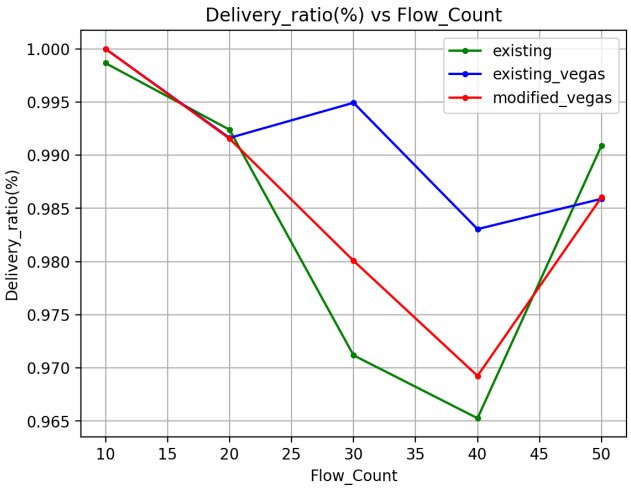
(n) throughput diff vs flow



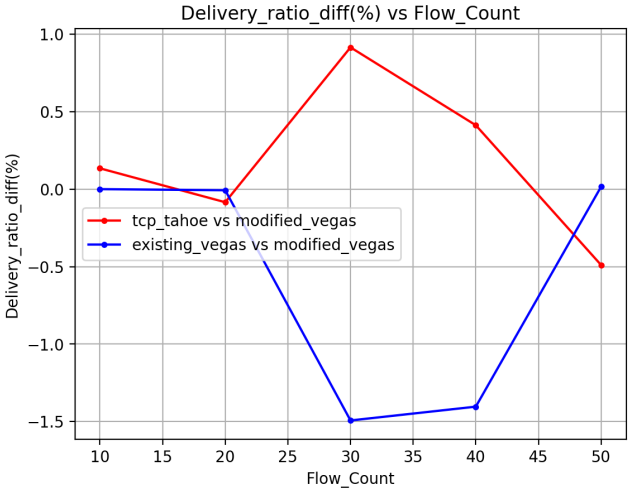
(o) avg delay vs flow



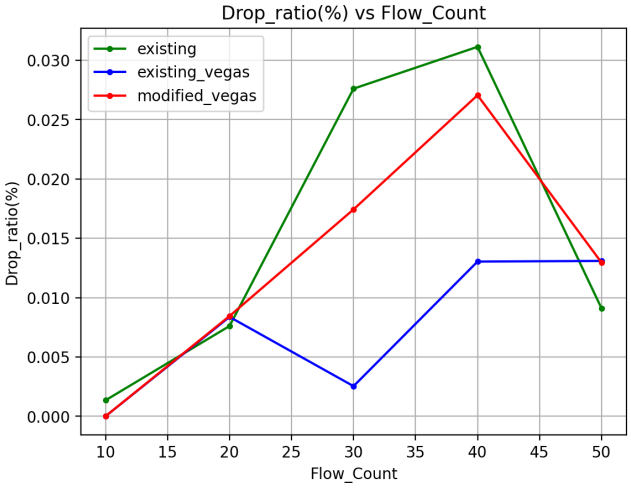
(p) avg delay diff vs flow



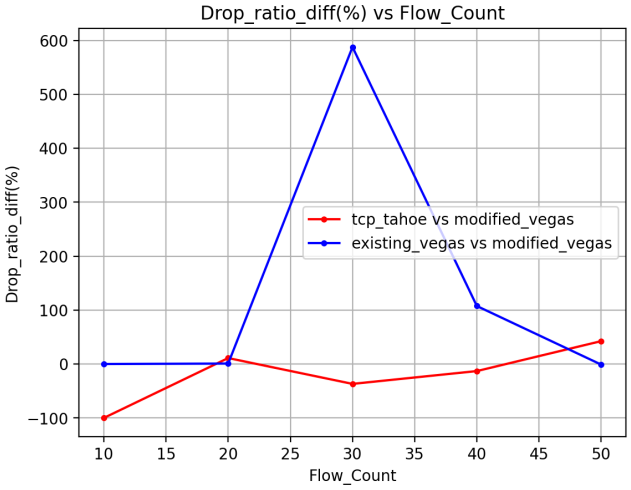
(q) delivery ratio vs flow



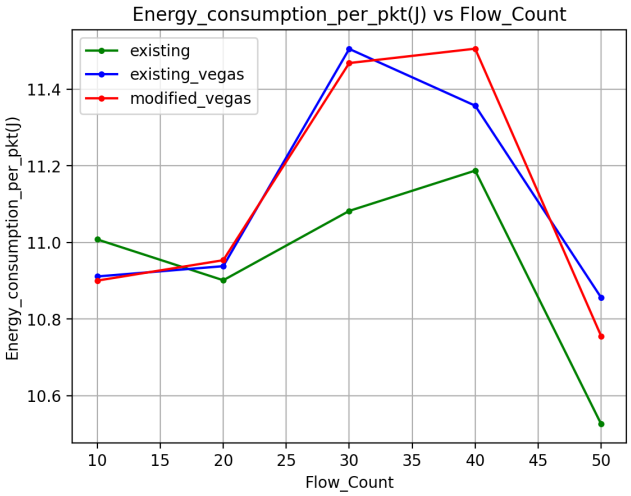
(r) delivery ratio diff vs flow



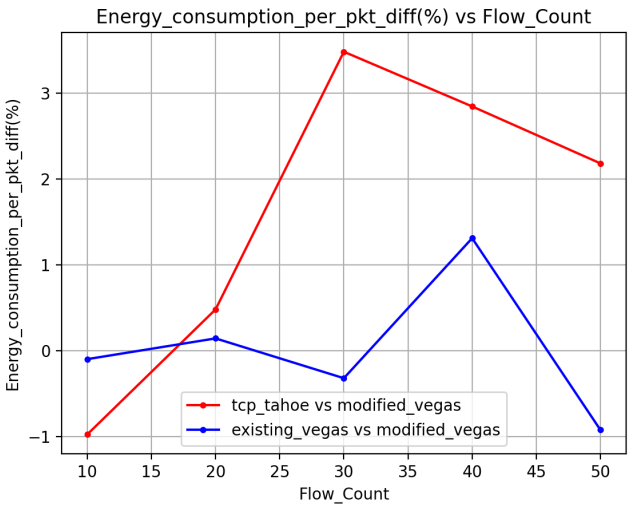
(s) drop ratio vs flow



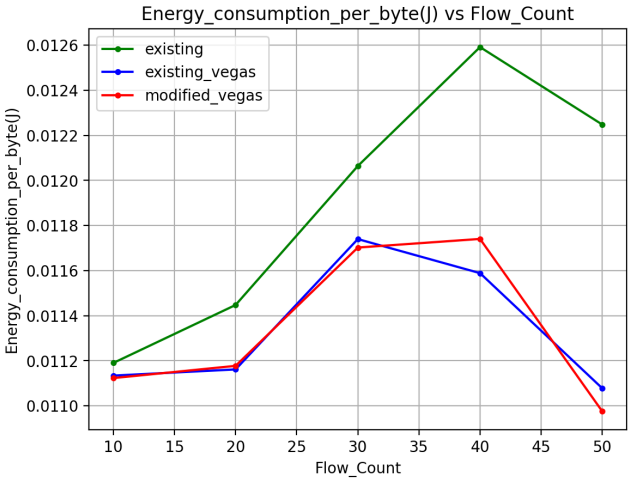
(t) drop ratio diff vs flow



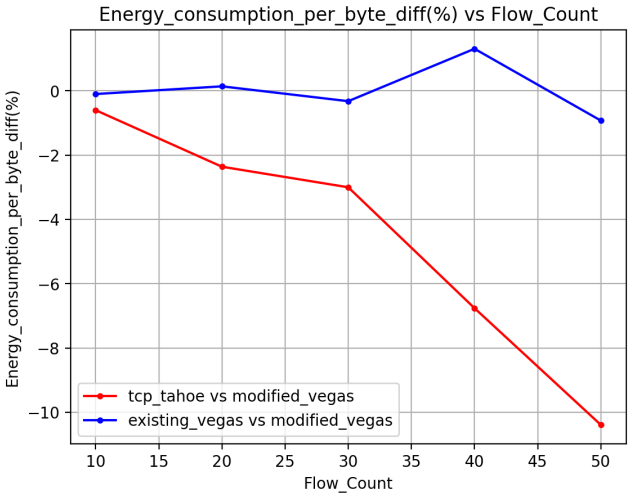
(u) energy pkt vs flow



(v) energy pkt diff vs flow



(w) energy byte vs flow

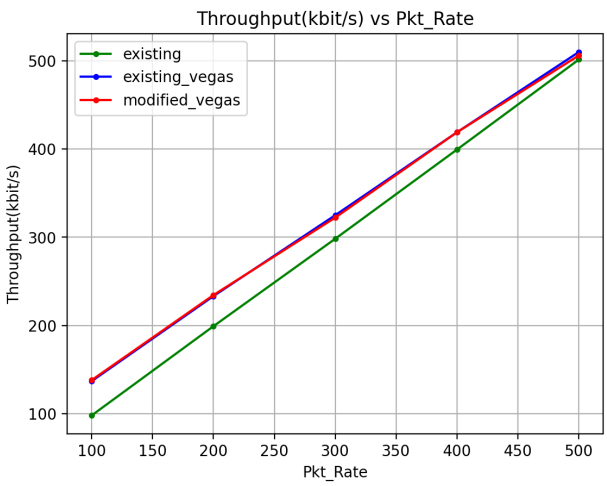


(x) energy byte diff vs flow

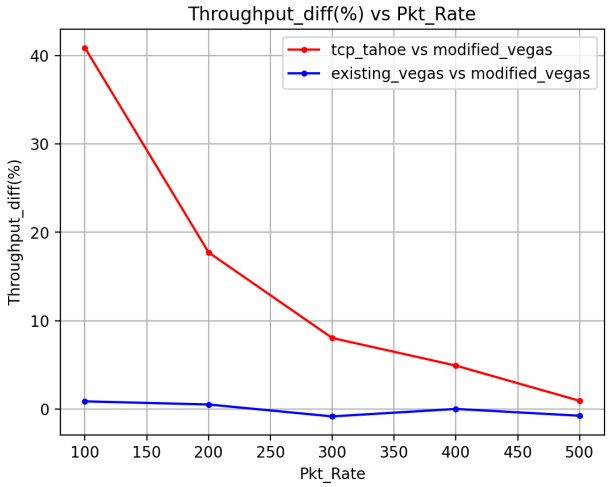
The observations we get from these graphs are -

- In the network throughput, the modified vegas algorithm works slightly better than existing algorithm($<$4%) and better than the default tcp tahoe implementation(on average 10%)
- In the average delay, the modified vegas performed worst among the three, the reason can be excessive and complex calculation of the congestion window which takes more time than the existing algorithm execution.
- In the delivery ratio, the modified vegas performed slightly better than the existing vegas which is opposite of the variation of number of nodes.
- Energy consumption per pkt decreases for the modified vegas algorithm which is a good performance improvement than the existing ones but per byte the performance goes worse.
- Again, the graphs are not so much well formed. For e.g: for node count 80, most of the graphs showed the spike. The proper reason cannot be discovered by excessive testing during the project.

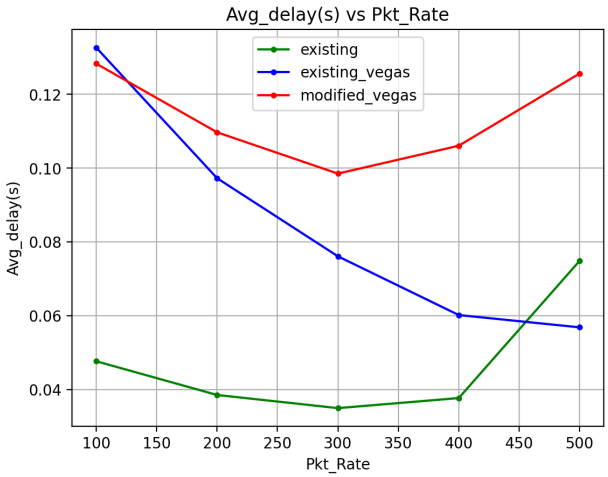
Varying Packet Rate



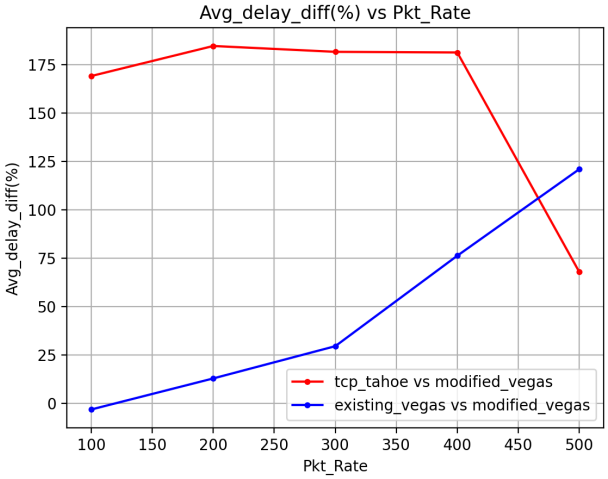
(y) throughput vs pkt rate



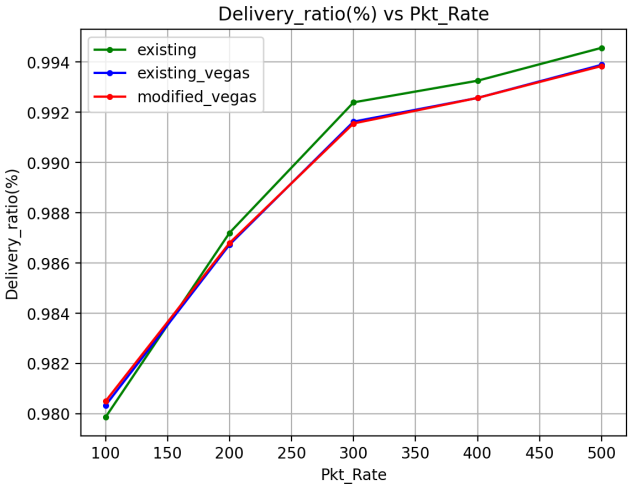
(z) throughput diff vs pkt rate



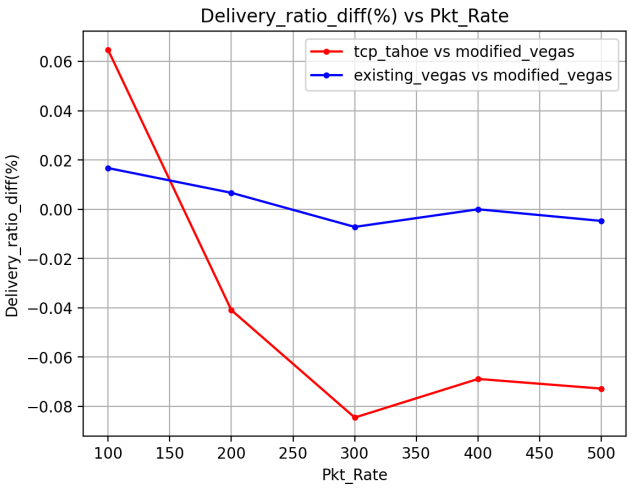
() avg delay vs pkt rate



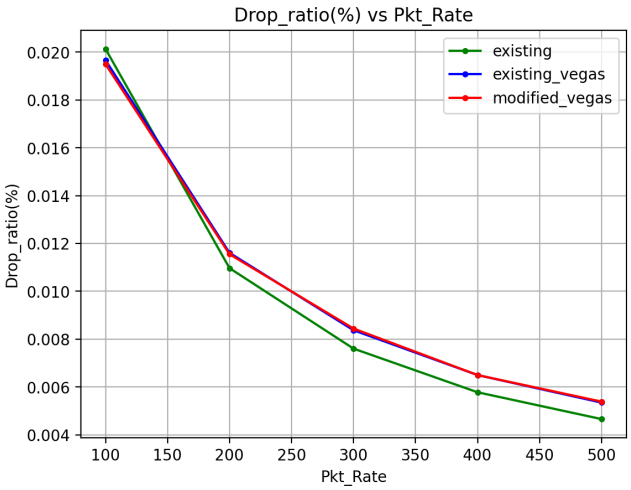
() avg delay diff vs pkt rate



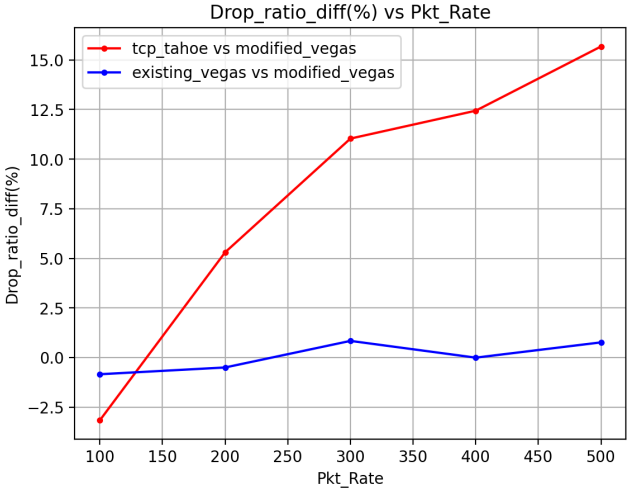
() delivery ratio vs pkt rate



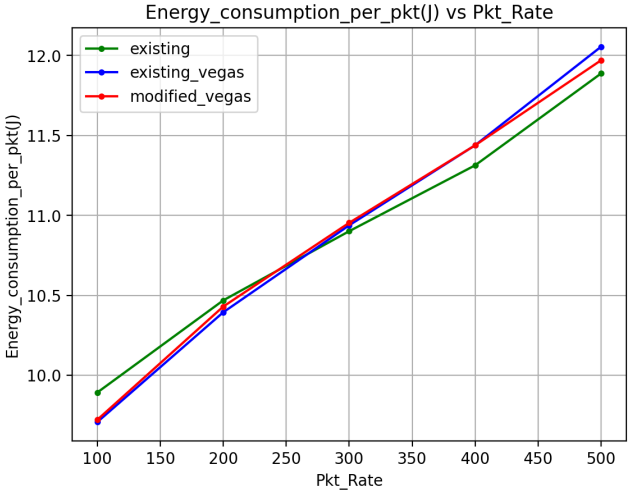
() delivery ratio diff vs pkt rate



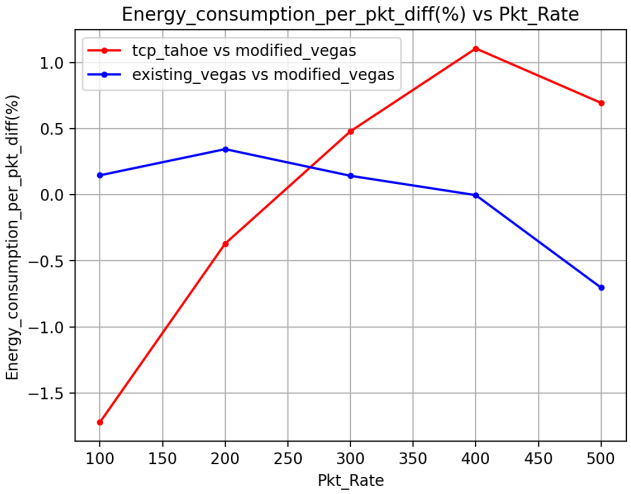
() drop ratio vs pkt rate



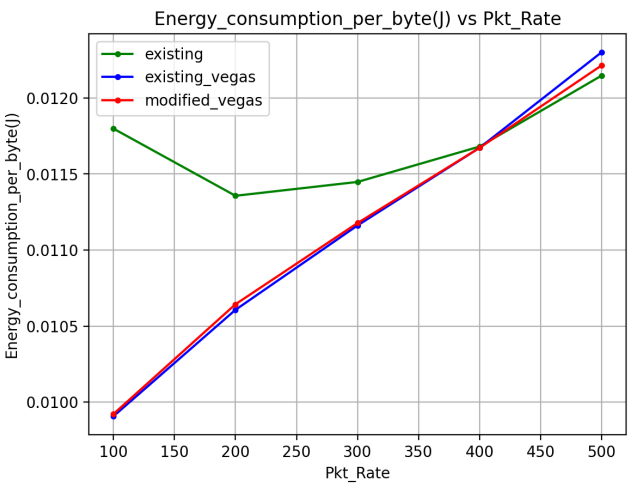
() drop ratio diff vs pkt rate



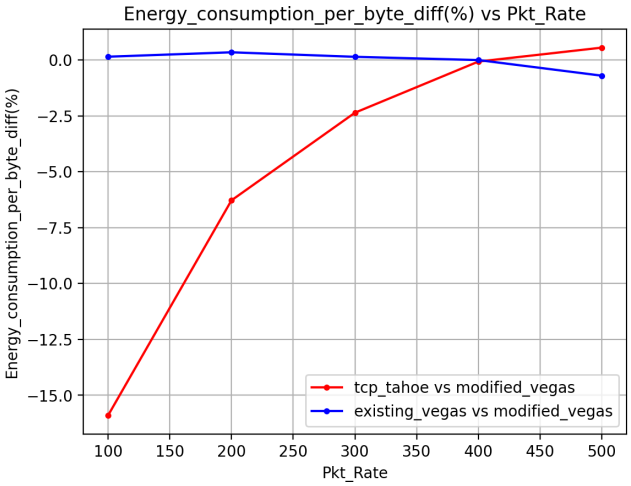
() energy pkt vs pkt rate



() energy pkt diff vs pkt rate



() energy byte vs pkt rate

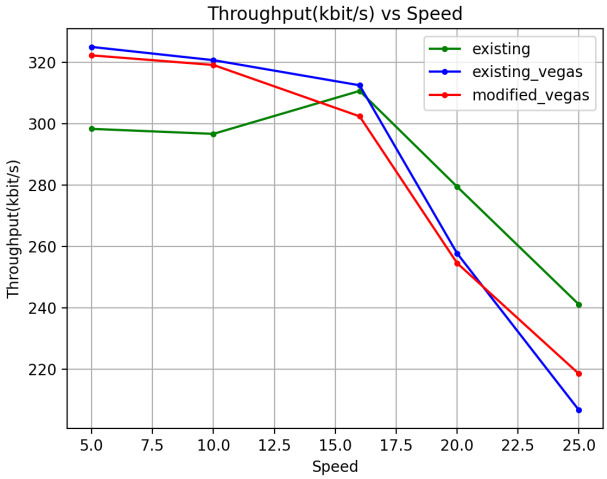


() energy byte diff vs pkt rate

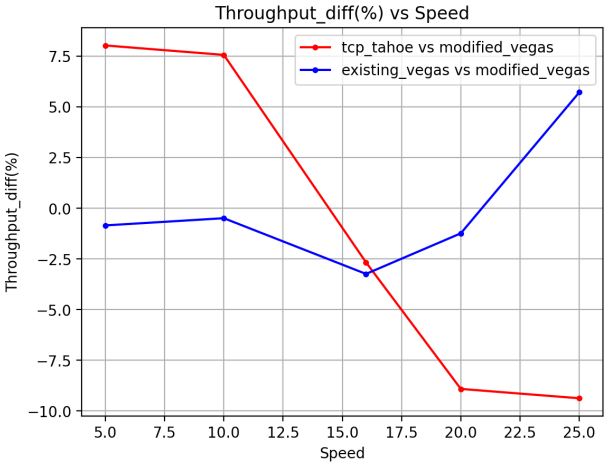
The observations we get from these graphs are -

- In the network throughput, the modified vegas algorithm works almost exactly same as the existing algorithm and slightly better than the default tcp tahoe implementation. So, pkt rate doesn't seem to have much effect on the existing vegas algorithm with respect to performance.
- In the average delay, the modified vegas performed worst among the three, the reason is described above in the other parts.
- In the delivery ratio and drop ratio, all the three performed almost same as others.
- Energy consumption per pkt decreases for the modified vegas algorithm which is a good performance improvement than the existing ones but per byte the performance goes worse.
- Overall, changing packet rate has not much effect on the performance of the three algorithms we compared. It may not have a significant impact on performance because TCP uses a congestion control algorithm that adjusts the sending rate dynamically based on network conditions to maintain network stability. Here these algorithms may have same kind of adjusting rates which lead to such kind of consistent situation.

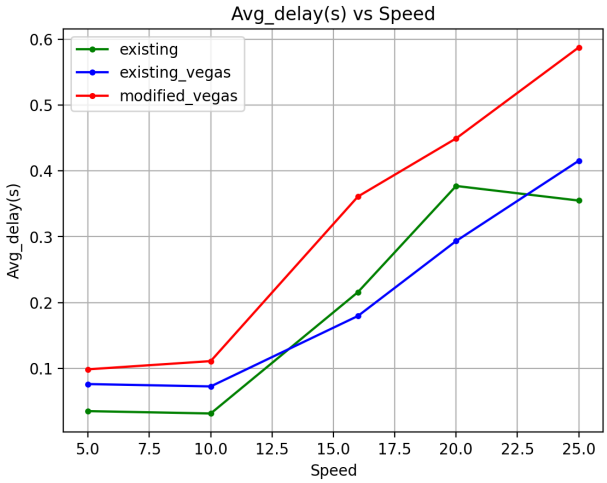
Varying Speed



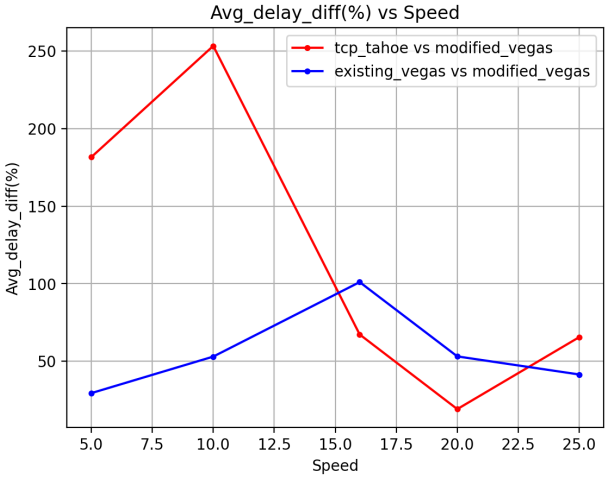
() throughput vs speed



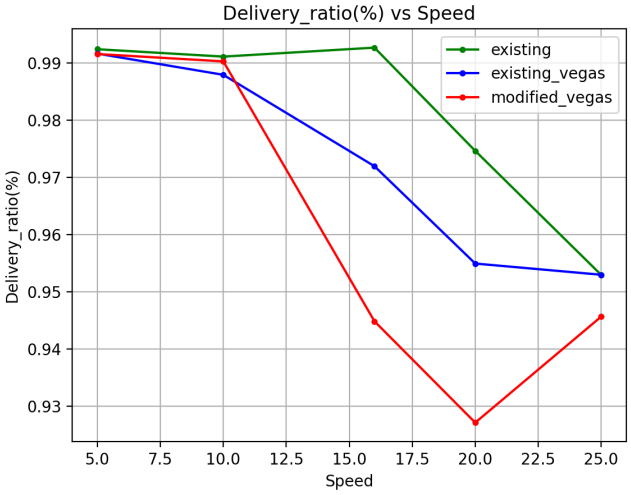
() throughput diff vs speed



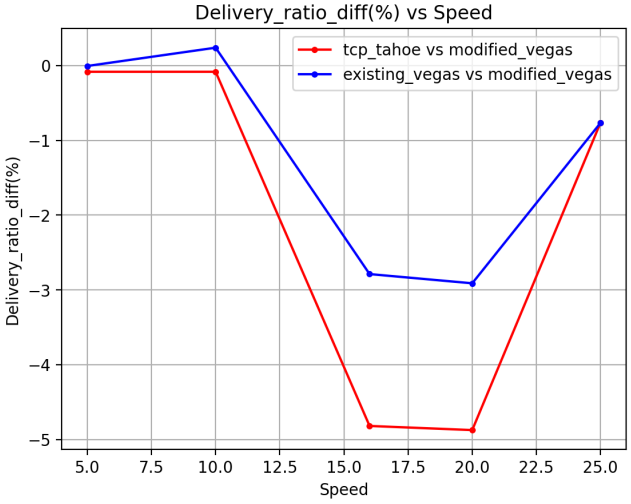
() avg delay vs speed



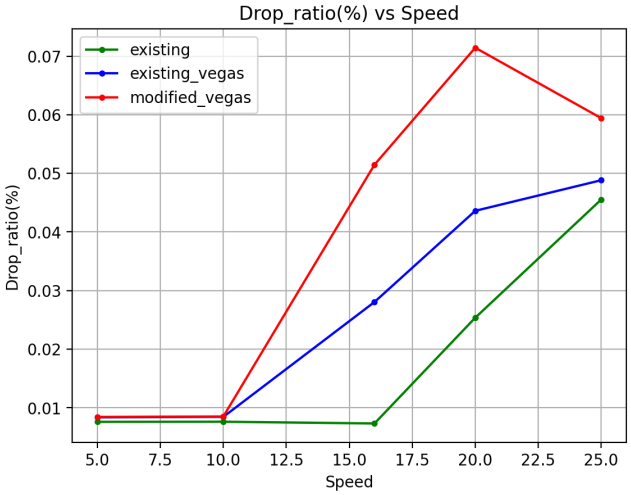
() avg delay diff vs speed



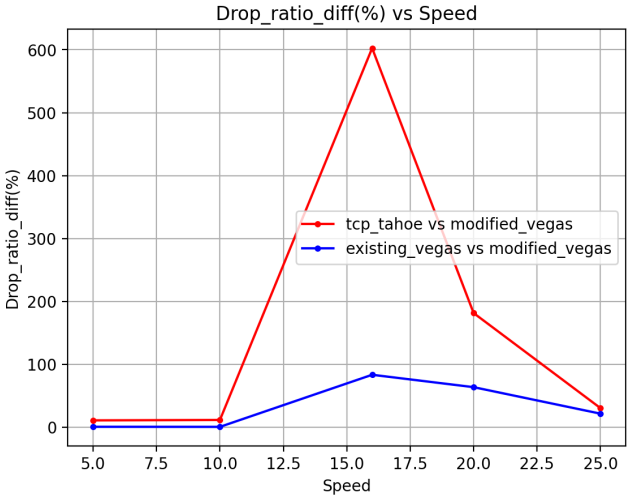
() delivery ratio vs speed



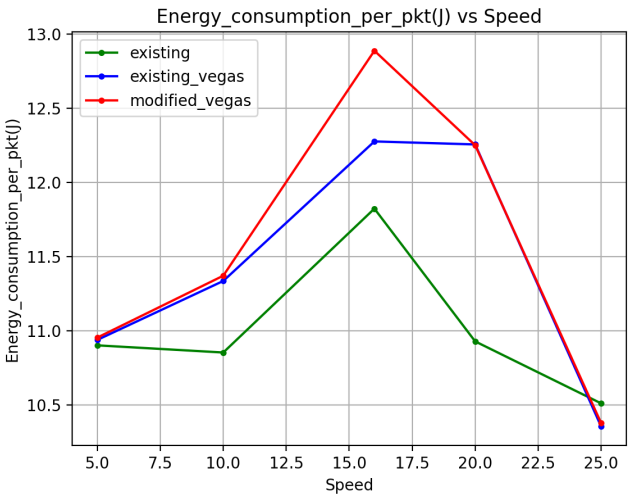
() delivery ratio diff vs speed



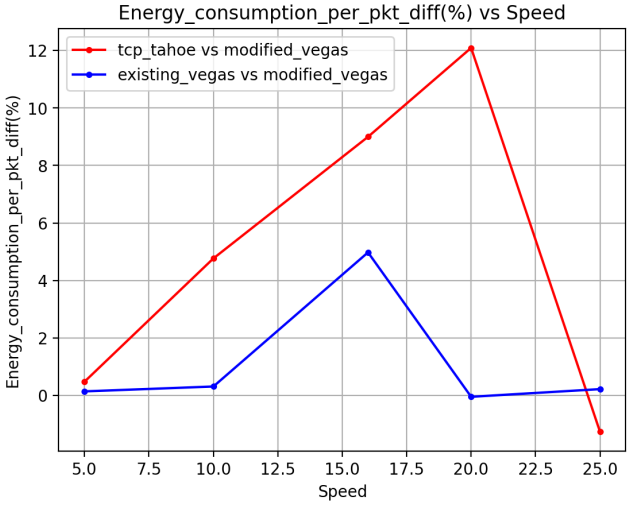
() drop ratio vs speed



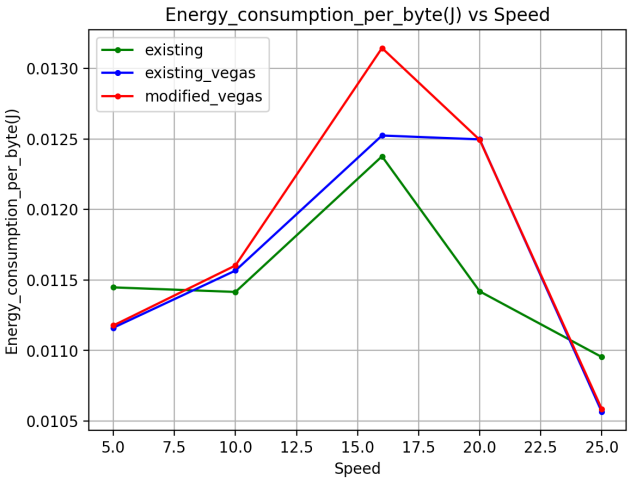
() drop ratio diff vs speed



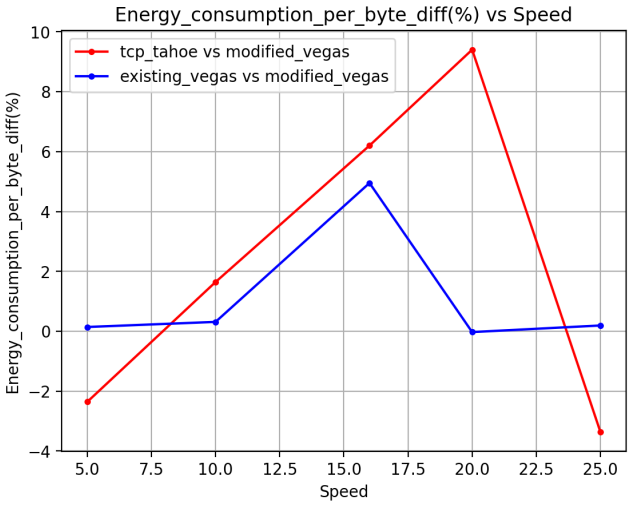
() energy pkt vs speed



() energy pkt diff vs speed



() energy byte vs speed



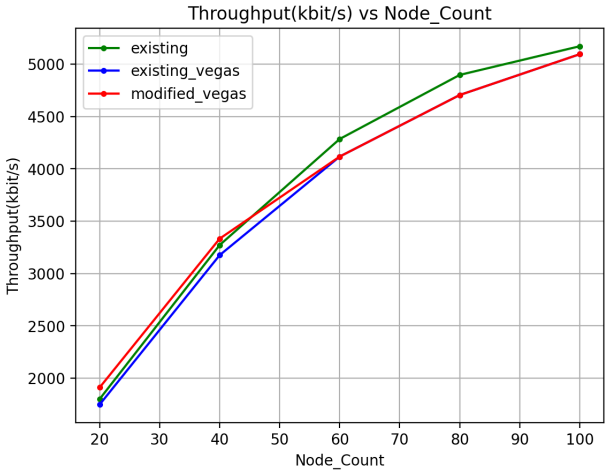
() energy byte diff vs speed

The observations we get from these graphs are -

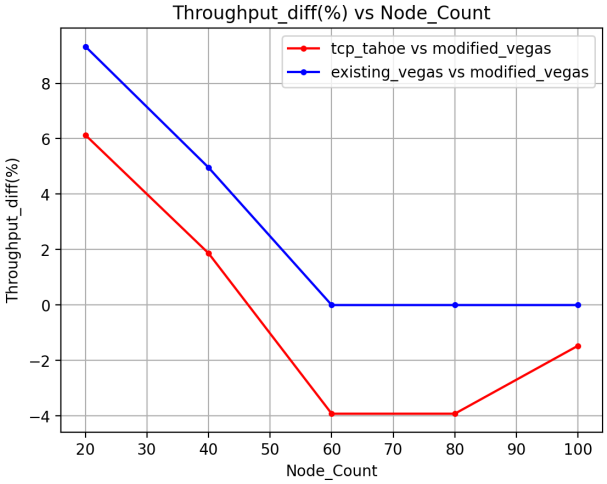
- In the network throughput, the modified vegas algorithm works variably as it doesnot follow any specific nature in the graph. In some cases, it works better whereas in other cases, works worse. But both not much.
- In the average delay, the modified vegas performed worst among the three, the reason is described above in the other parts.
- In the delivery ratio and drop ratio, the performance of modified vegas decreases though not so much significant change.
- Energy consumption also gives bad performance as modified vegas consumes more energy than others.
- Overall, changing speed doesnot give much effective and good performance for modified vegas algorithm.

Graphs(Wired)

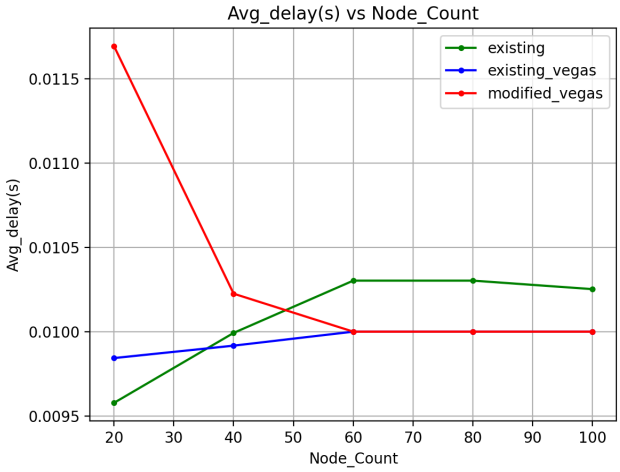
Varying number of nodes



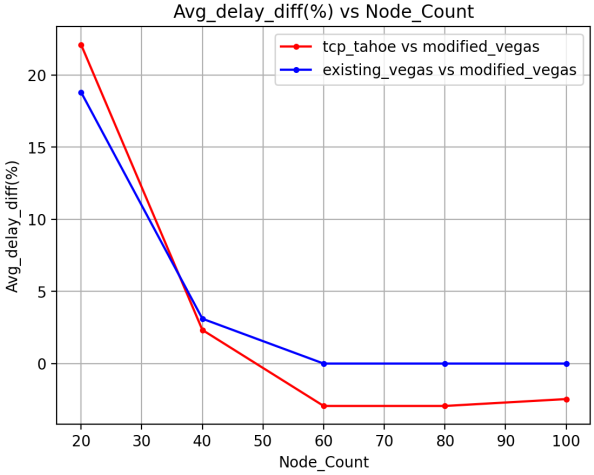
() throughput vs Node



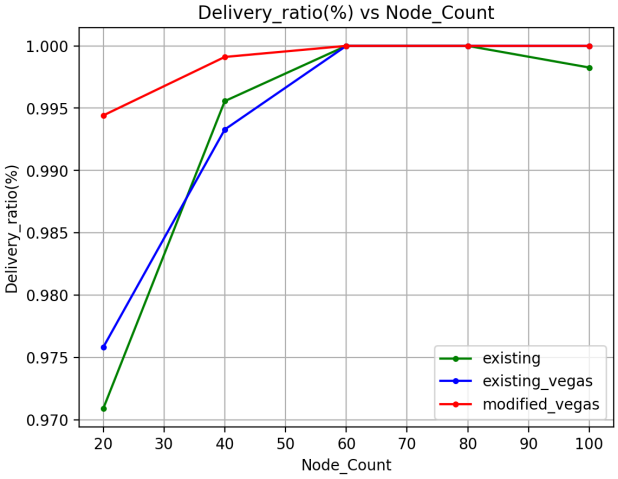
() throughput diff vs Node



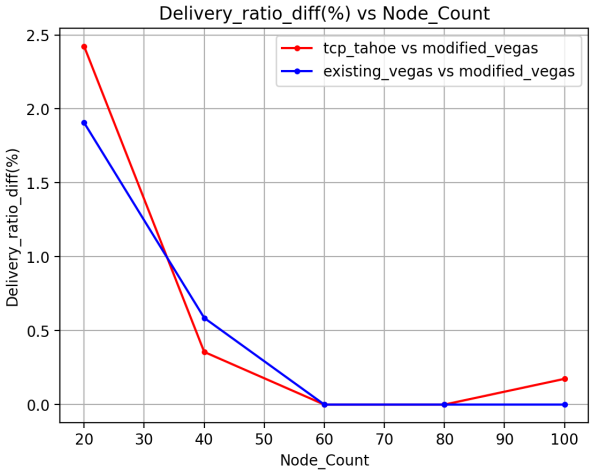
() avg delay vs Node



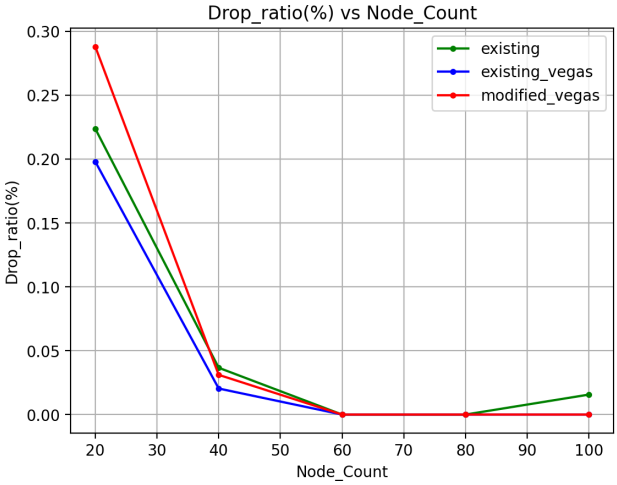
() avg delay diff vs Node



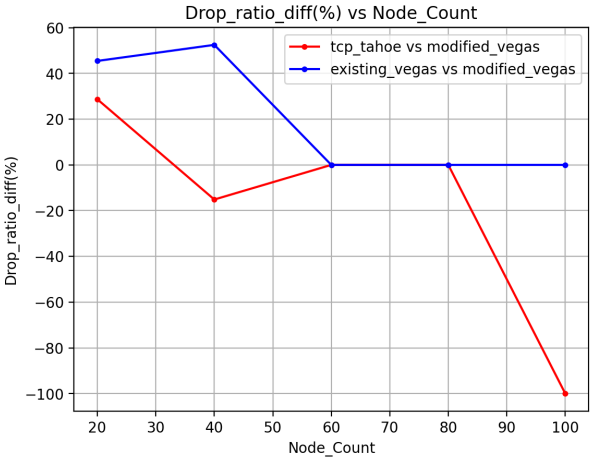
() delivery ratio vs Node



() delivery ratio diff vs Node



() drop ratio vs Node

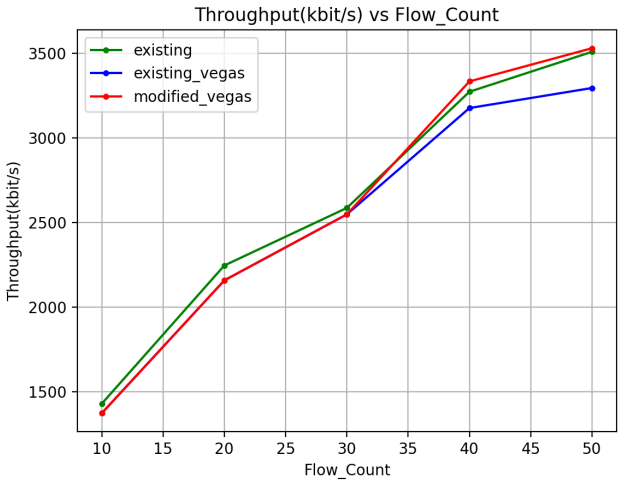


() drop ratio diff vs Node

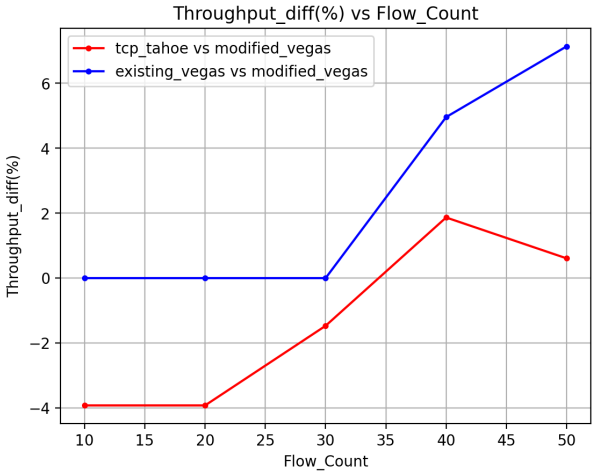
The observations we get from these graphs are -

- In wired topology, there has not been much variation in performances seen. For only specific cases- some variance discovered. For e.g: if the number of nodes is 20, the delivery ratio performs nearly 2.5% better than others whereas in case of average delay, that became worse.
- Overall, changing number of nodes in wired topology doesnot give significant impact on performance for modified vegas algorithm.

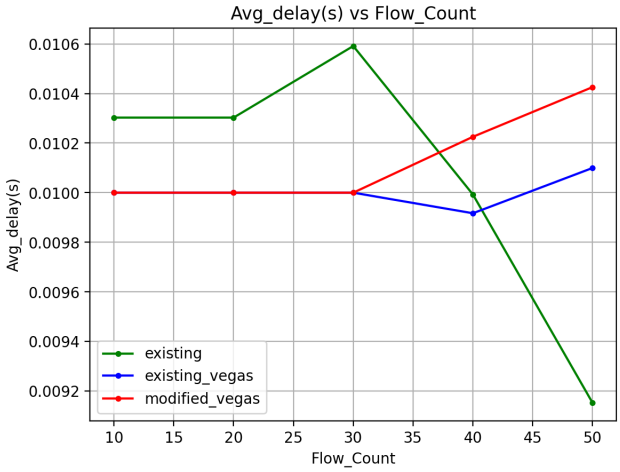
Varying number of flows



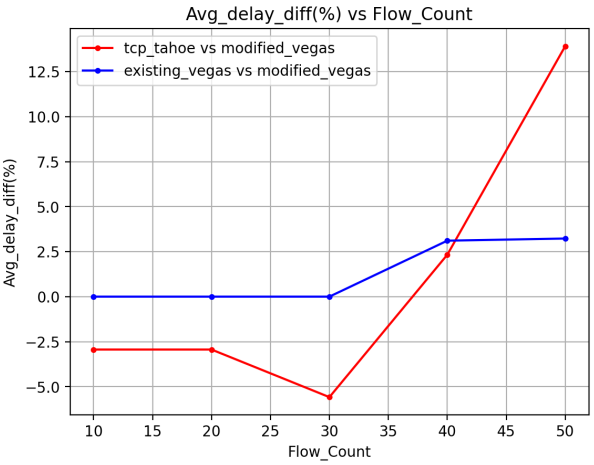
() throughput vs Flow



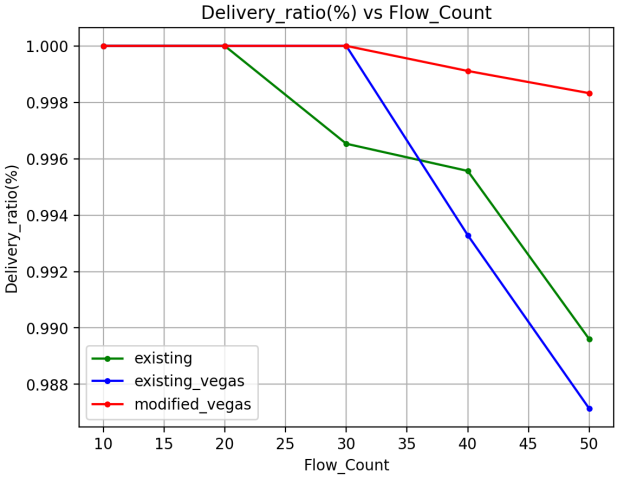
() throughput diff vs Flow



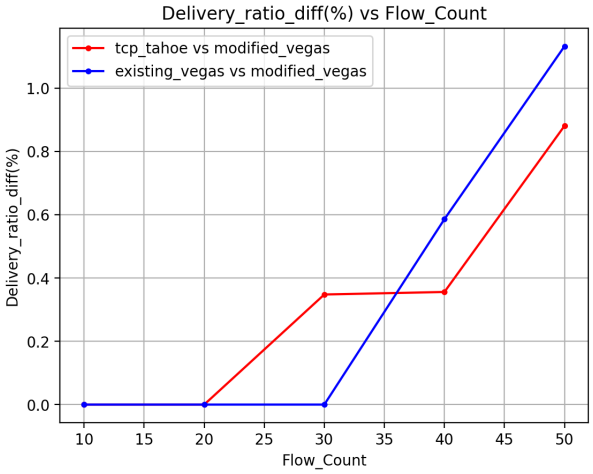
() avg delay vs Flow



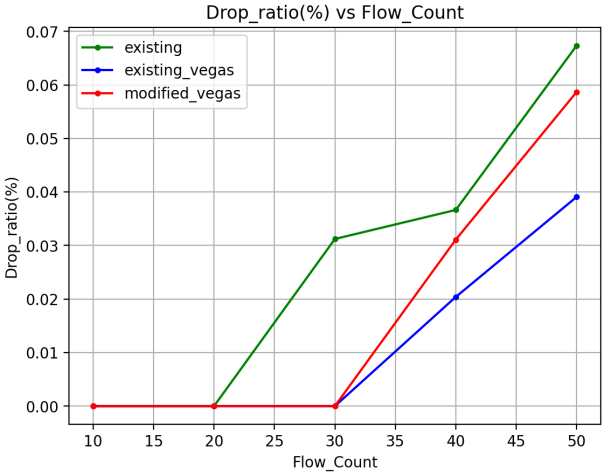
() avg delay diff vs Flow



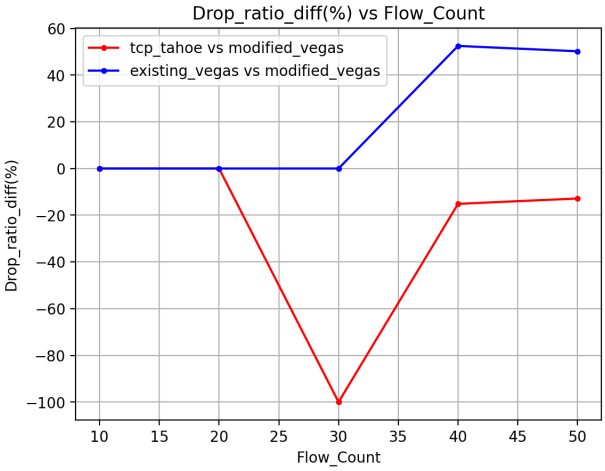
() delivery ratio vs Flow



() delivery ratio diff vs Flow



() drop ratio vs Flow

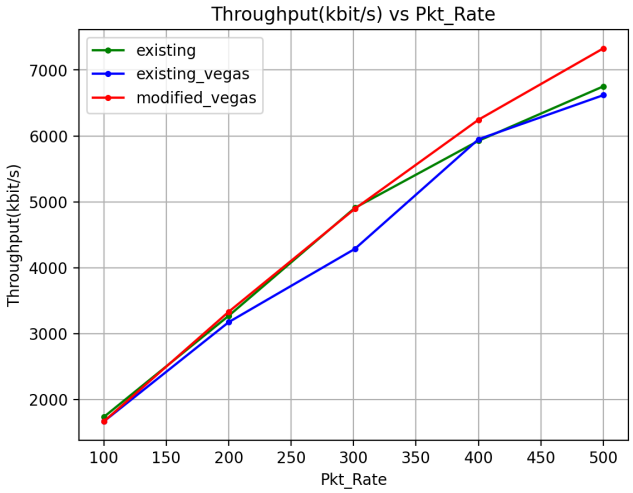


() drop ratio diff vs Flow

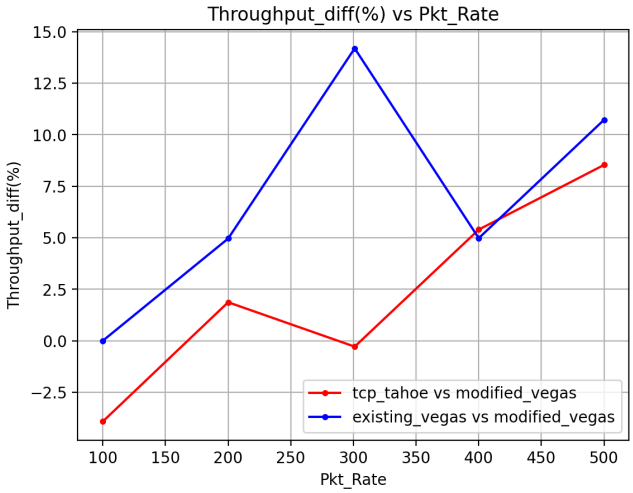
The observations we get from these graphs are -

- In wired topology, there has not been much variation in performances seen for delivery ratio, drop ratio and throughput.
- Surprisingly, average delay becomes better in modified vegas. Upto this point, this was the worst performance metric for modified vegas.
- Overall, changing number of flows too in wired topology doesnot give significant impact on performance for modified vegas algorithm.

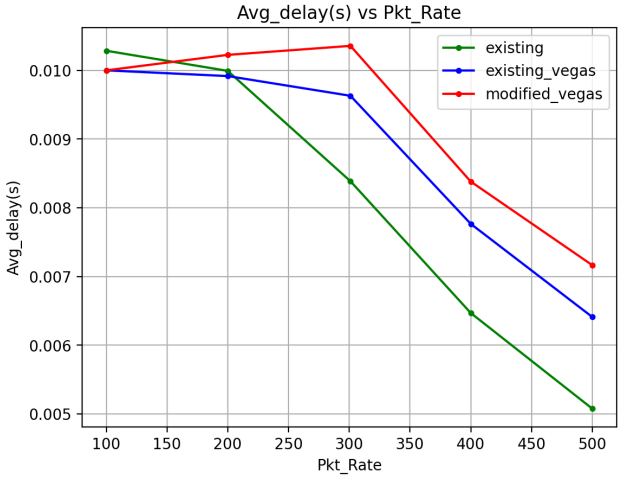
Varying Packet Rate



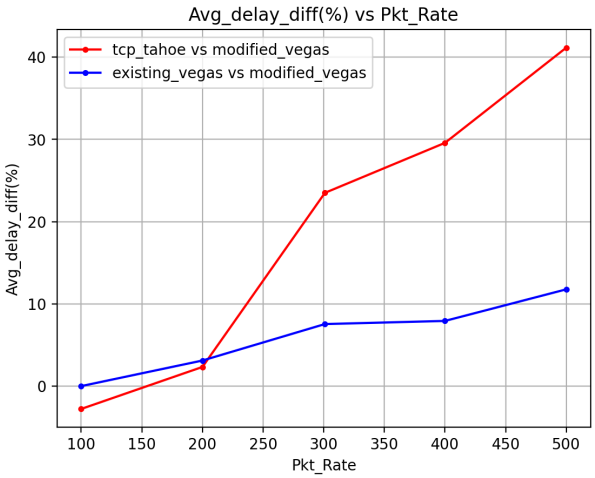
() throughput vs pkt Rate



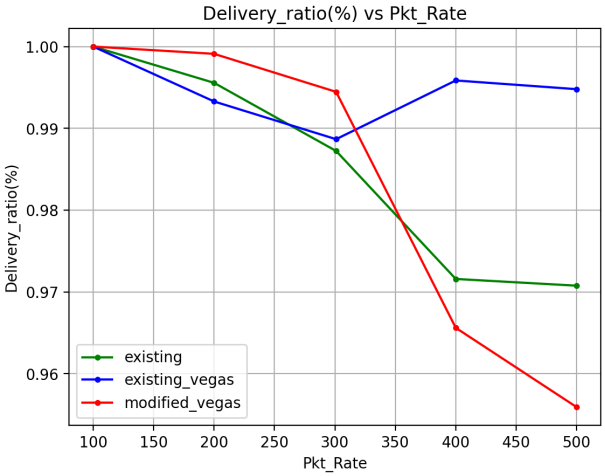
() throughput diff vs pkt rate



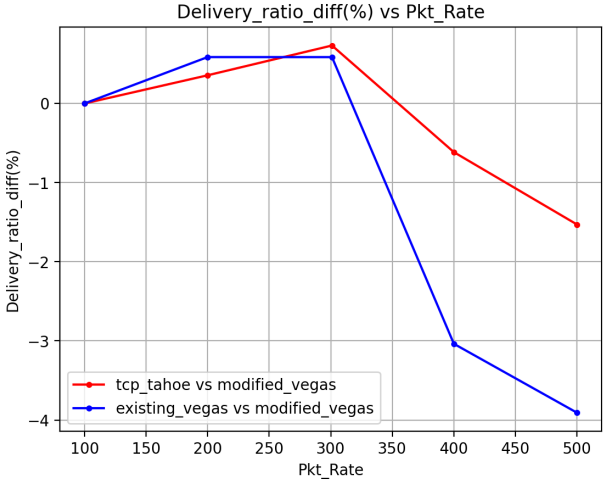
() avg delay vs pkt rate



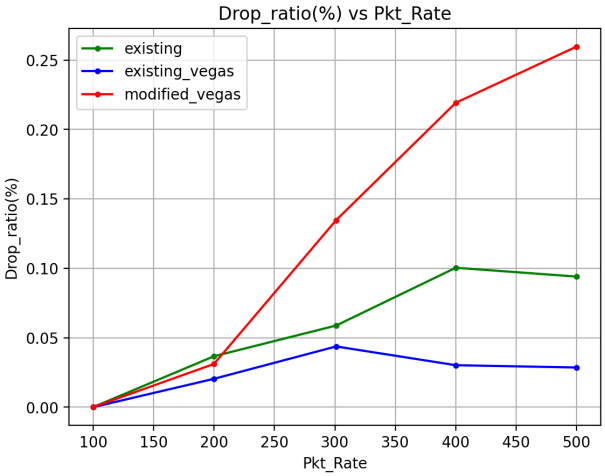
() avg delay diff vs pkt rate



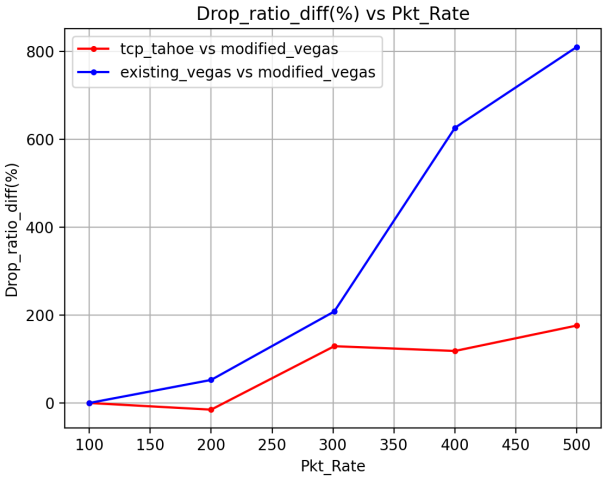
() delivery ratio vs pkt rate



() delivery ratio diff vs pkt rate



() drop ratio vs pkt rate



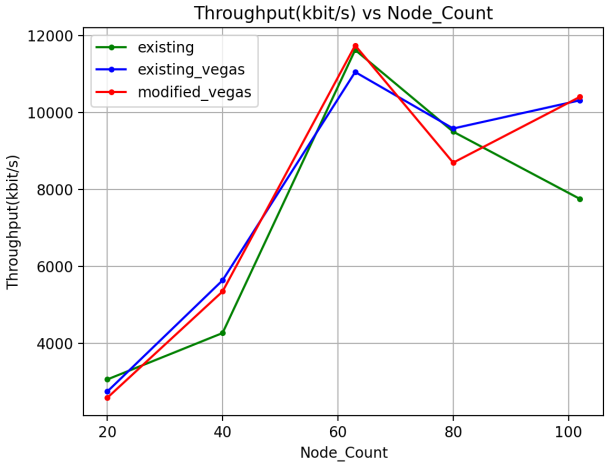
() drop ratio diff vs pkt rate

The observations we get from these graphs are -

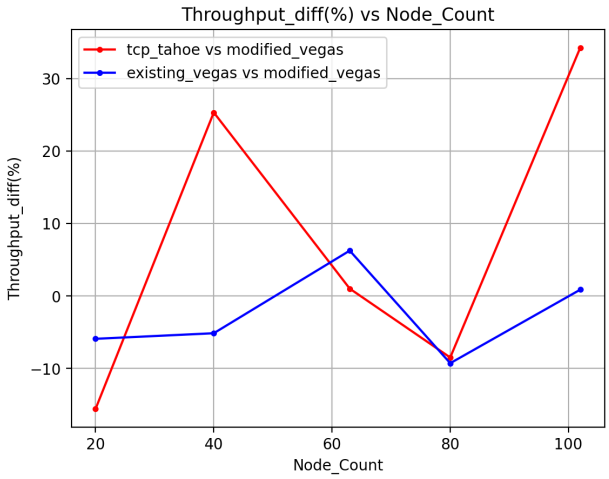
- In the case of Variation of packet rate in wired too, just like wireless, doesnot give significant impact on performance for modified vegas algorithm.

Bonus: Graphs(Wired-Cum-Wireless)

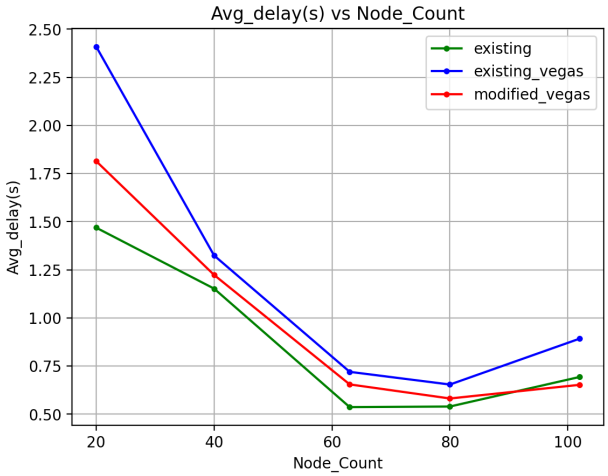
Varying Number of Nodes



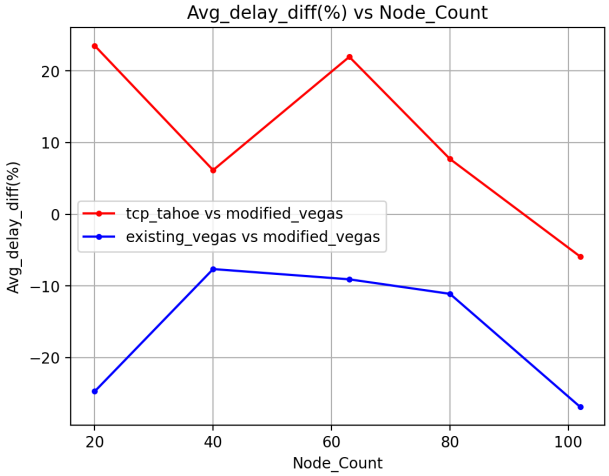
() throughput vs Node



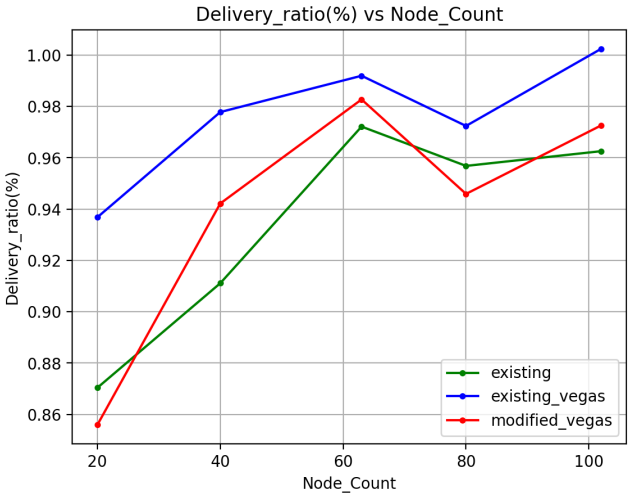
() throughput diff vs Node



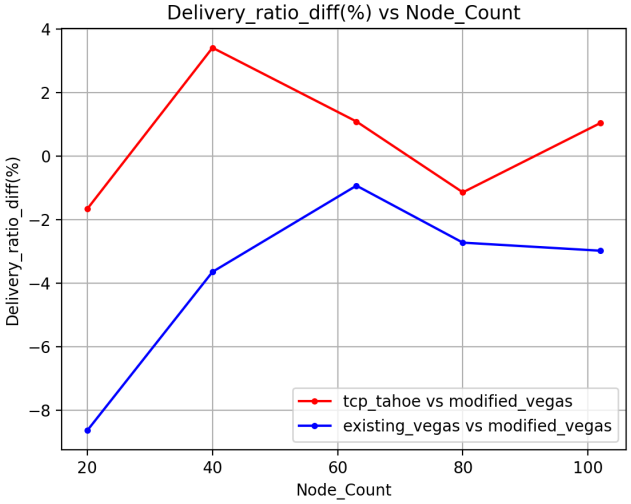
() avg delay vs Node



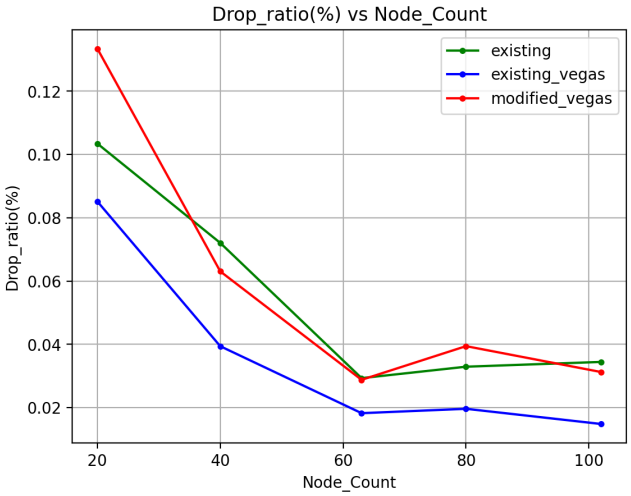
() avg delay diff vs Node



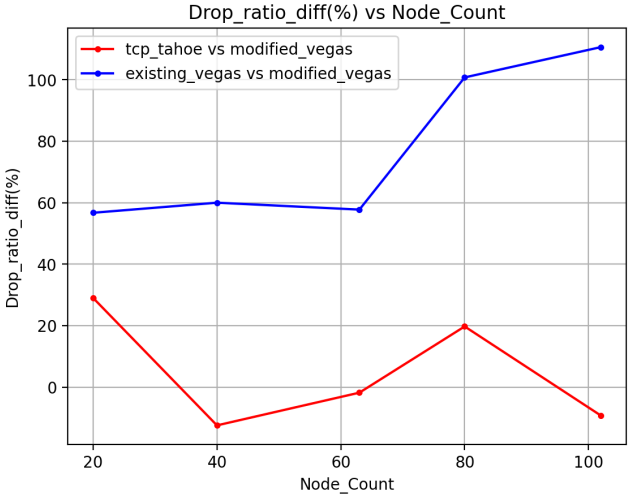
() delivery ratio vs Node



() delivery ratio diff vs Node



() drop ratio vs Node

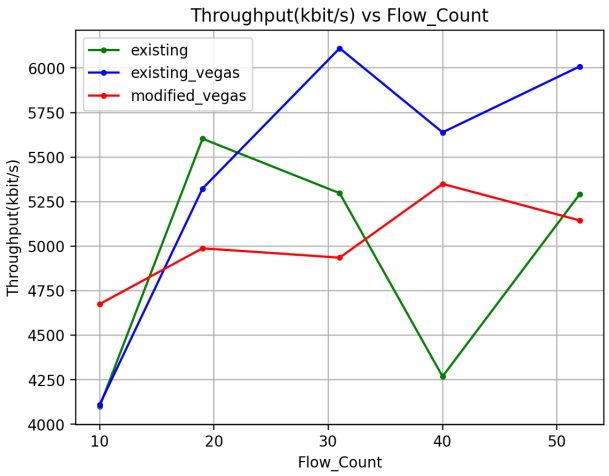


() drop ratio diff vs Node

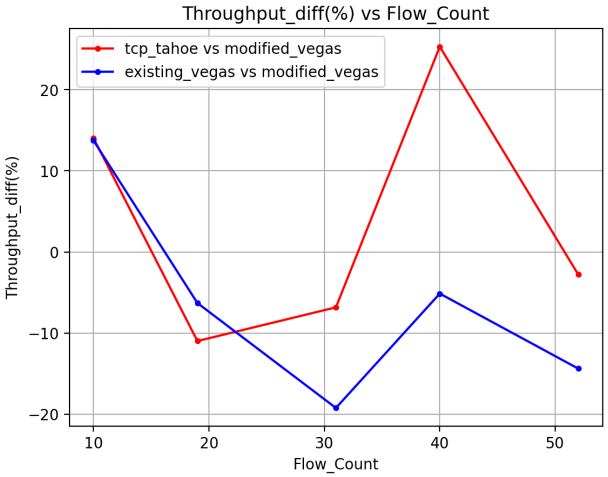
The observations we get from these graphs are -

- In wired-cum-wireless topology, there has not been much variation in performances seen for the variation of number of nodes.

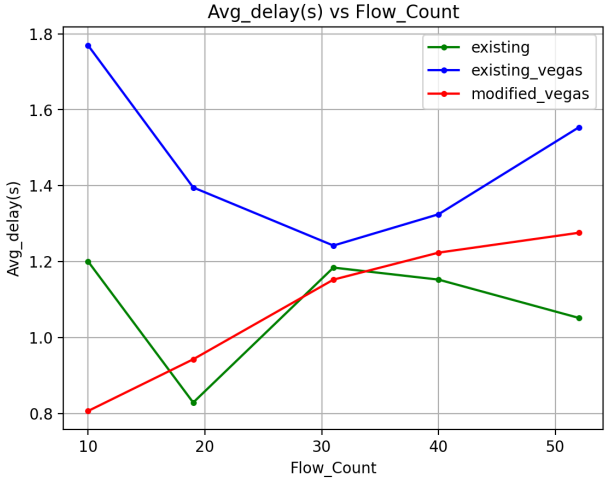
Varying Number of Flows



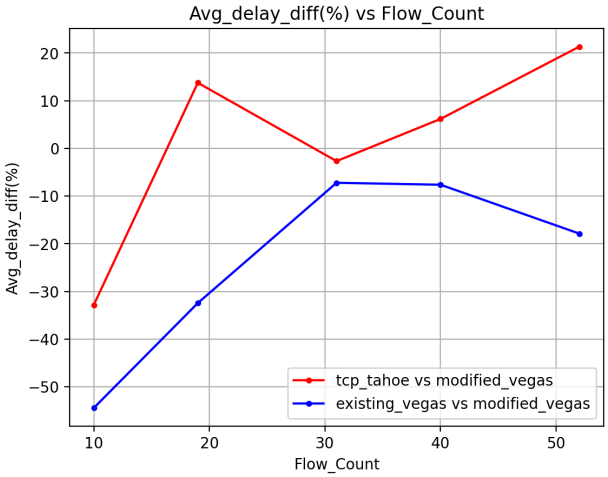
() throughput vs flow



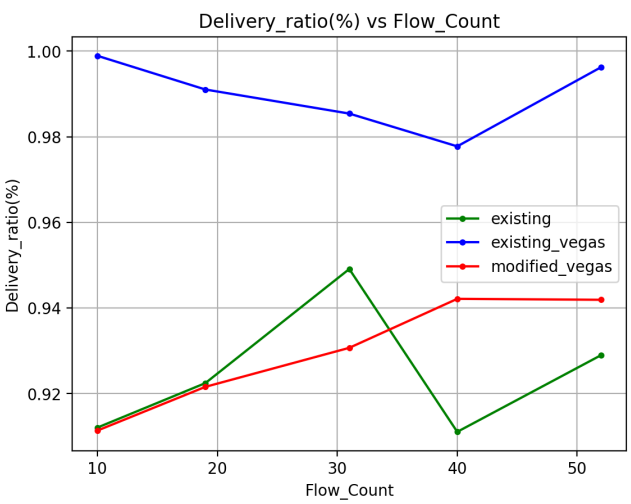
() throughput diff vs flow



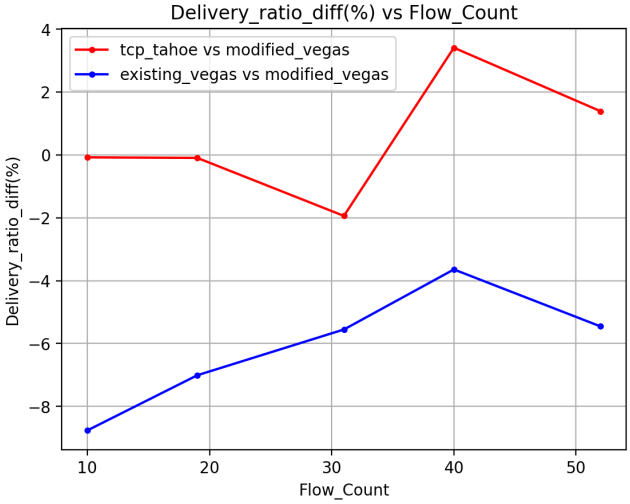
() avg delay vs flow



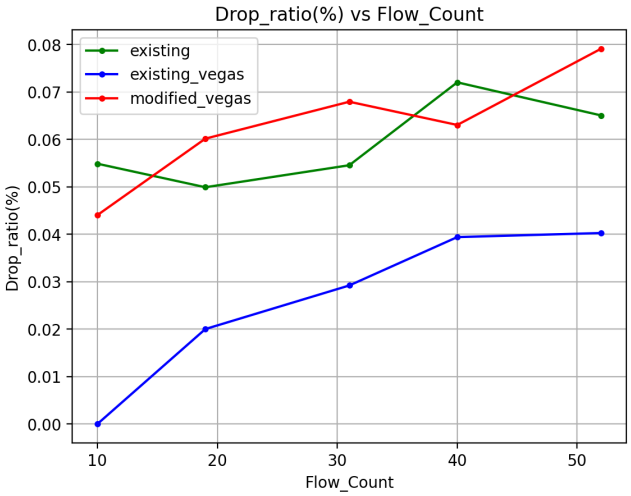
() avg delay diff vs flow



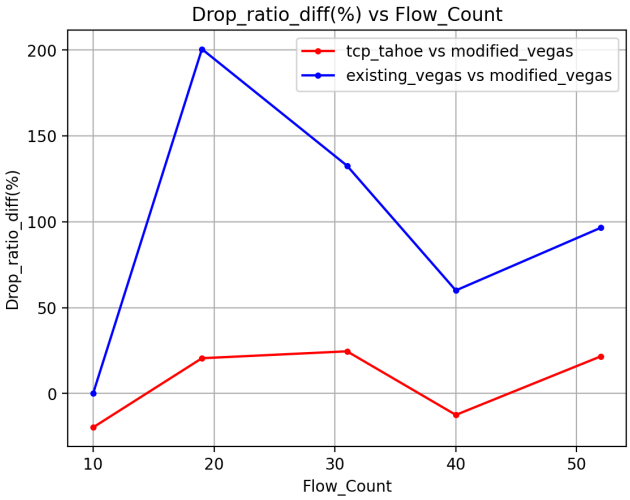
() delivery ratio vs flow



() delivery ratio diff vs flow



() drop ratio vs flow

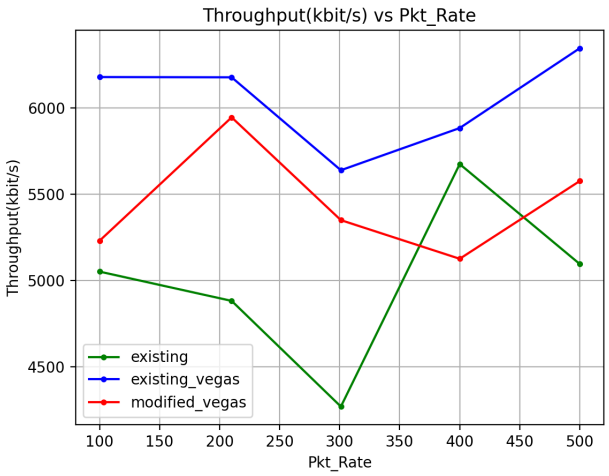


() drop ratio diff vs flow

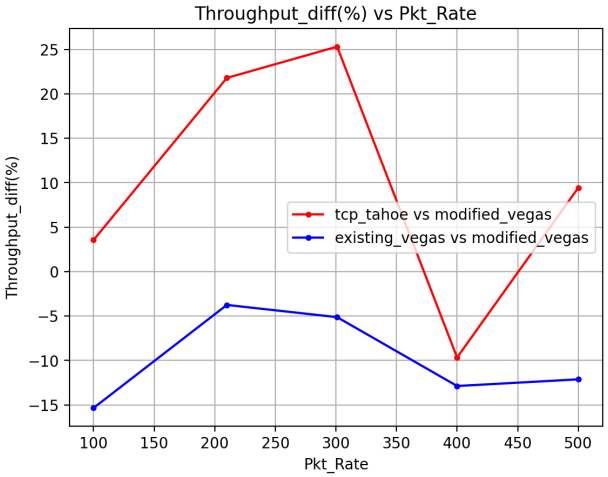
The observations we get from these graphs are -

- In wired-cum-wireless topology, number of flows became an important paramter as we see some significant change in graphs
- Throughput performance became worse for the modified vegas algorithm
- But the average delay decreases significantly for the modified vegas which is a good performance improvement
- But the delivery and drop performance significantly drops with respect to increment of flows.

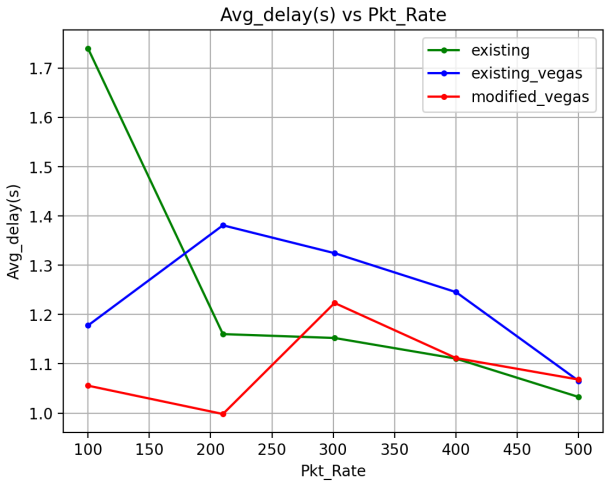
Varying Packet Rate



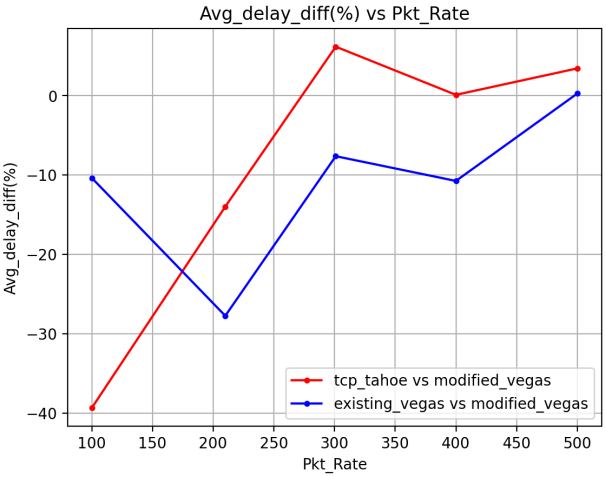
() throughput vs pkt rate



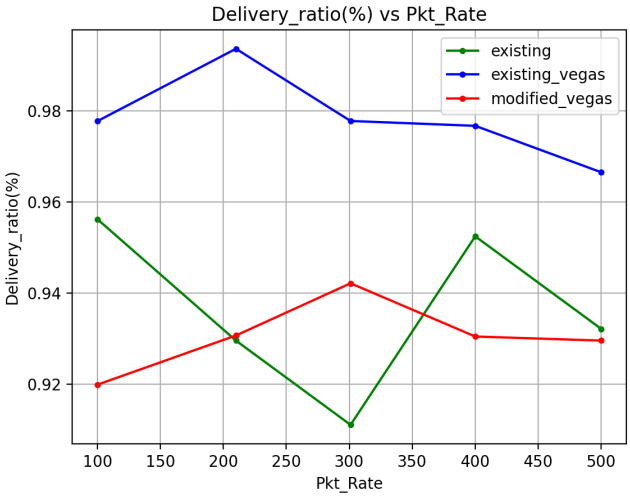
() throughput diff vs pkt rate



() avg delay vs pkt rate

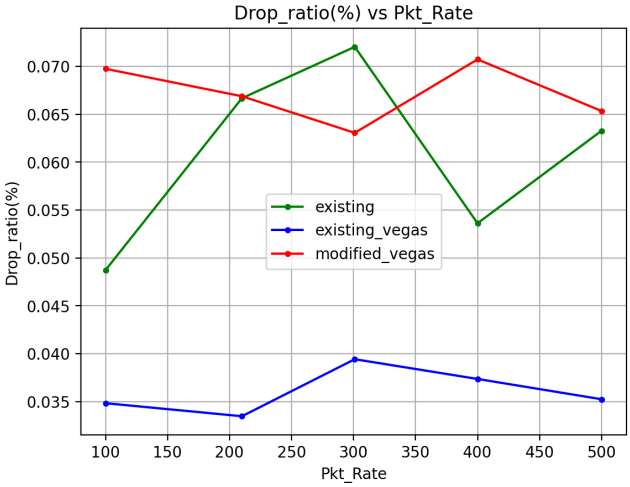


() avg delay diff vs pkt rate

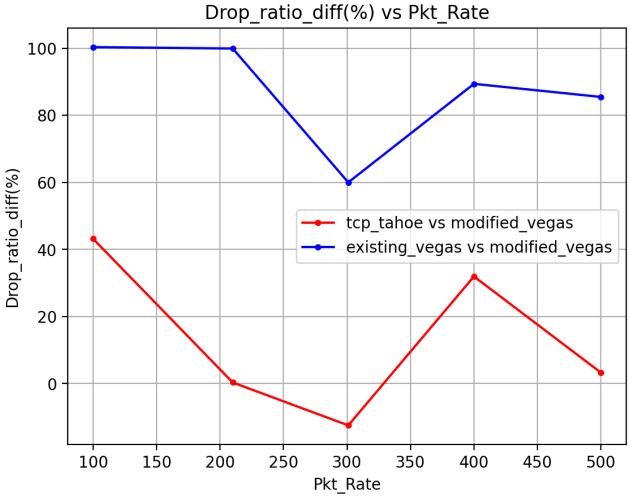


() delivery ratio vs pkt rate

() delivery ratio diff vs pkt rate



() drop ratio vs pkt rate

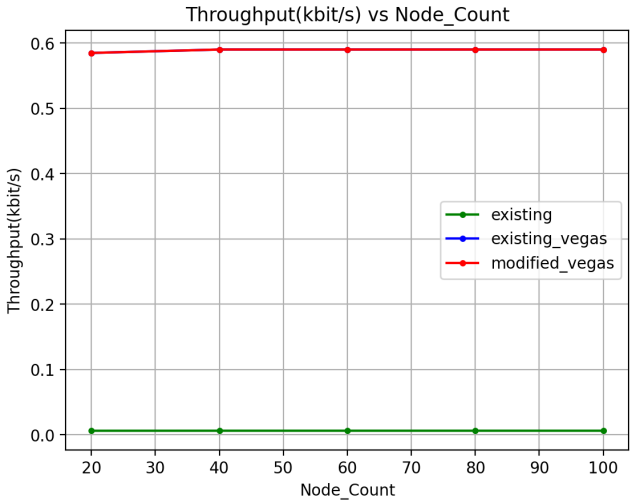


() drop ratio diff vs pkt rate

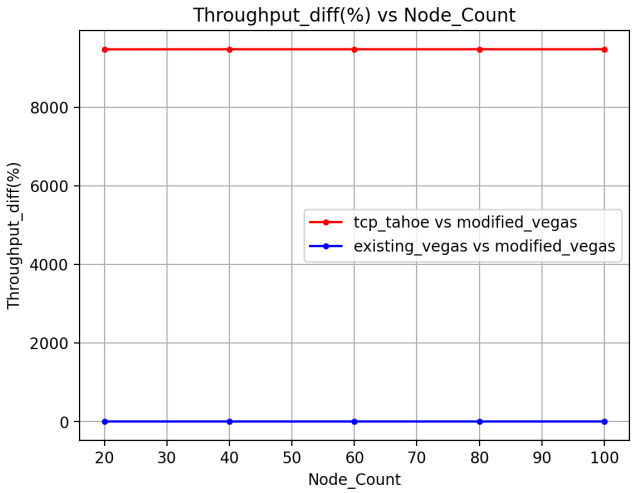
- In wired-cum-wireless topology unlike the other two networks, varying packet rates impact the performance greatly
- Throughput performance became worse for the modified vegas algorithm
- But the average delay decreases significantly for the modified vegas which is a good performance improvement
- But the delivery and drop performance became worse.
- Overall, varying packet rates in wired-cum-wirelss topology gives some significant impact on decreasing performance for modified vegas algorithm.

Bonus: Graphs(Satellite)

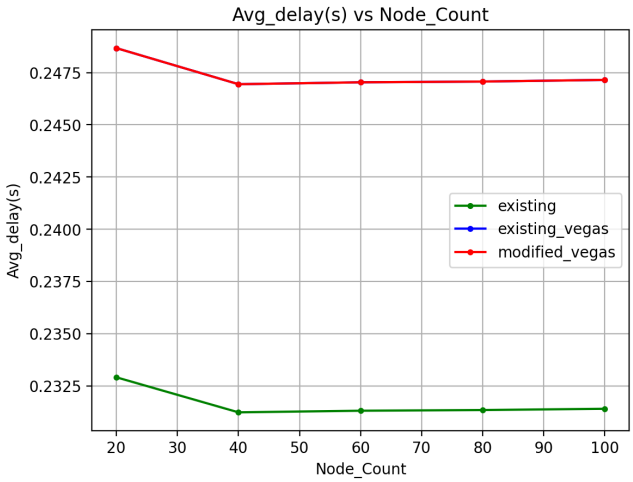
Varying Number of Nodes



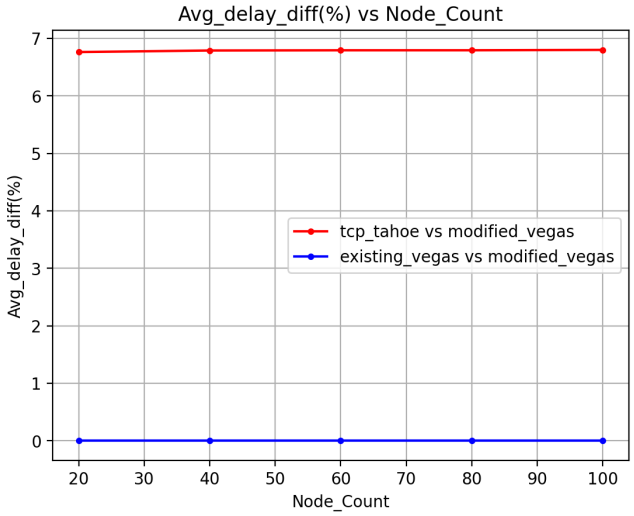
() throughput vs Node



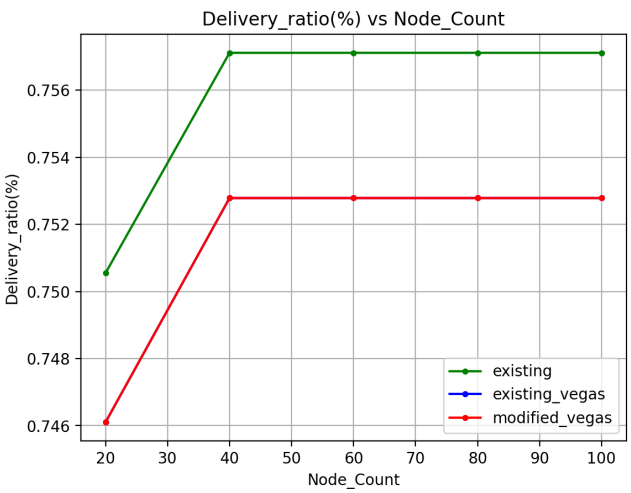
() throughput diff vs Node



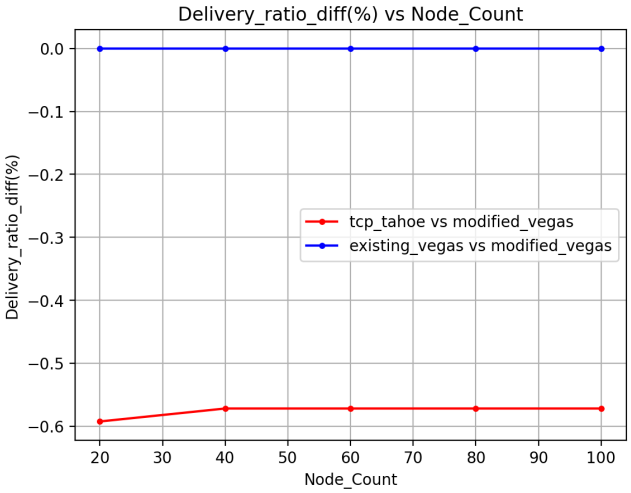
() avg delay vs Node



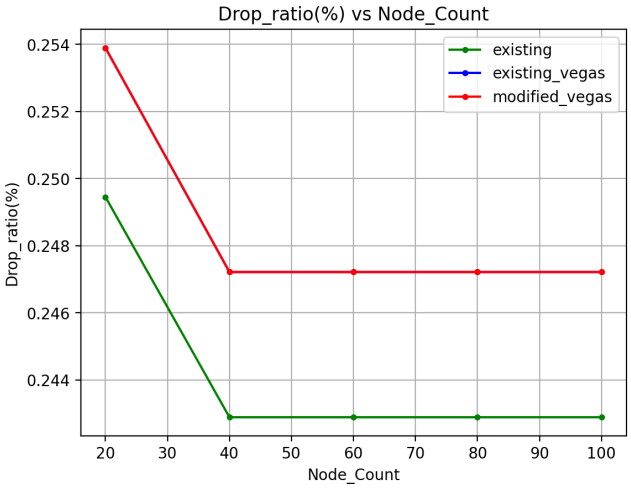
() avg delay diff vs Node



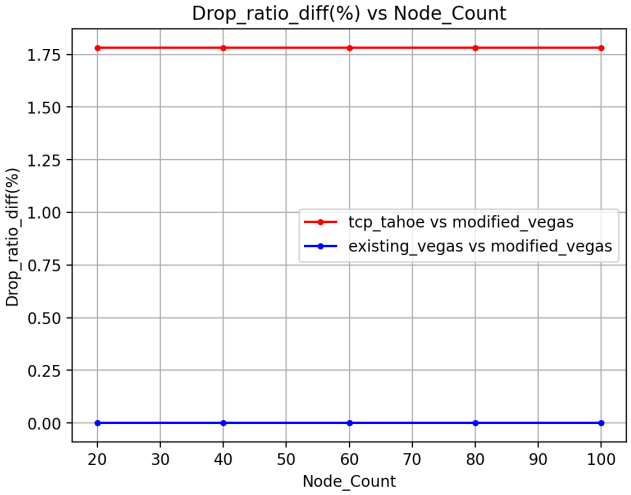
() delivery ratio vs Node



() delivery ratio diff vs Node

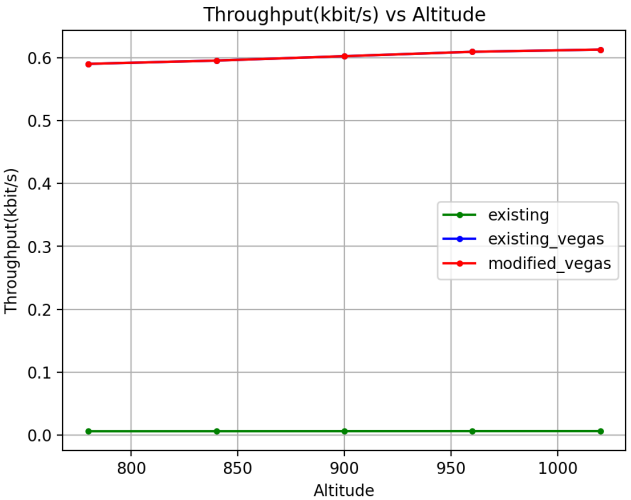


() drop ratio vs Node

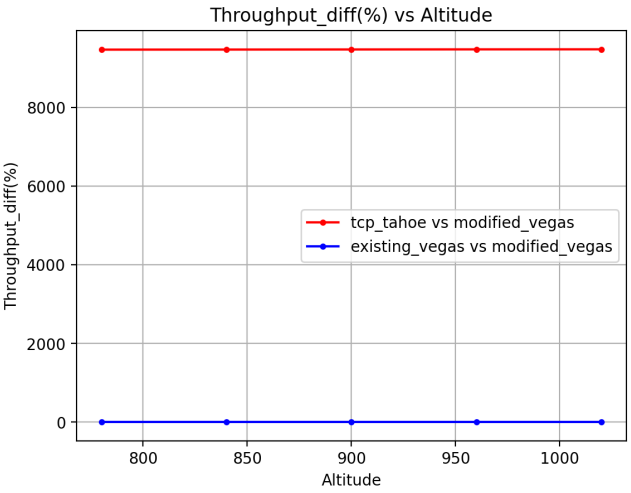


() drop ratio diff vs Node

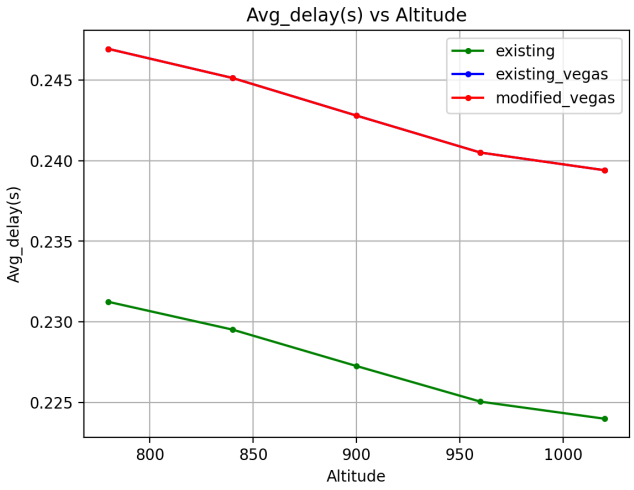
Varying Altitude



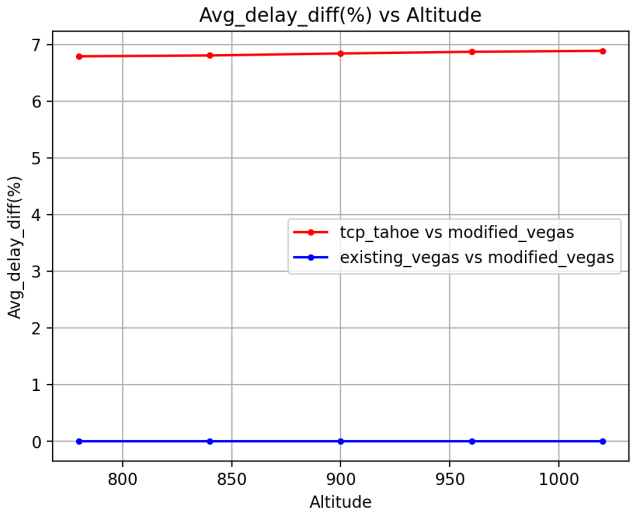
() throughput vs Altitude



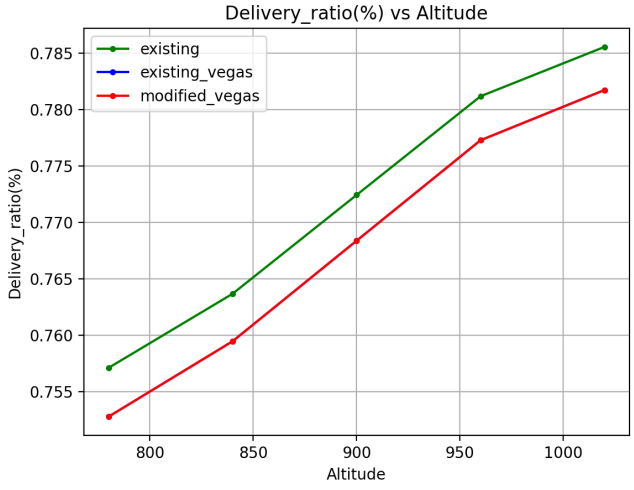
() throughput diff vs Altitude



() avg delay vs Altitude

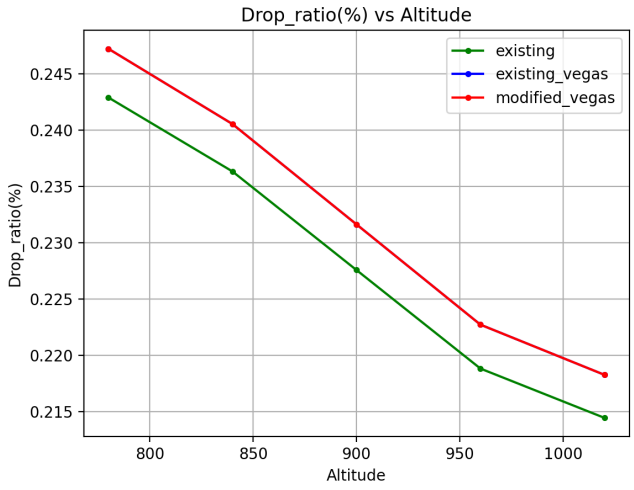


() avg delay diff vs Altitude

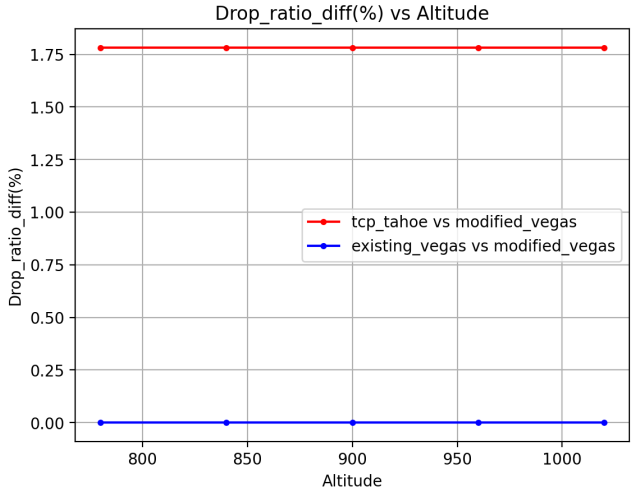


() delivery ratio vs Altitude

() delivery ratio diff vs Altitude



() drop ratio vs Altitude



() drop ratio diff vs Altitude

Summary Findings

Based on the results we get from the graphs, we can make some observations-

- Overall performance for the throughput was significantly increases with the modified vegas algorithm which is the main point of success of our project.
- Average delay became higher than the existing algorithms which had not gone in our way as we failed to capitalize the performance increment in this metric.
- Delivery ratio and drop ratio had not much effect in our experiment. In most cases the changes were not that impactful at all
- Energy consumption sometimes became better for the modified mechanism.
- This project was intended to give a performance improvement in all types of networks but as we see, except the wireless topology, the impact on other networks was not satisfactory. Reason behind this can be -
  - The change in equations of the project. As we developed the exponentially weighted mechanism instead of bayesian networks.
  - Our topology selection might be better. Also we could verify the performance into a predictable network with no randomization introduced
  - In the paper, they only took throughput metric in their observation. In that case, we came out successful. But the other metric like end to end delay, energy consumption were untouched.

On the whole, even though not all the cases were successful in this project, the overall change and modification in the performance were quite satisfactory in the context of the project. The algorithm gives success in network throughput improvement whereas badly affects the end to end delay.