
Wireless Communications

Final Project

# Practical Rate Adaptation for Very High Throughput WLANs

Anagnostopoulos Vasileios 03153
Georgitziki Garyfalia 03218
Gavouras Dimitrios 03145

June 28, 2024

# Contents

# 1 Introduction

Our project aims to compare the $L3S$ (Long-term Stability and Short-term Responsiveness) algorithm and the $Minstrel$ algorithm, two leading rate adaptation schemes for WLANs. The $L3S$ algorithm, designed for high throughput 802.11n MIMO WLANs, dynamically adjusts transmission rates based on real-time channel conditions to balance long-term stability and short-term responsiveness. Conversely, the $Minstrel$ algorithm, widely used in Linux-based systems, optimizes rate selection by utilizing acknowledgment feedback and periodic probing. $Minstrel$ continuously updates transmission rates based on the success and failure of previous transmissions to maintain the best possible throughput given current link conditions. Unlike the original implementation described in the paper `Practical Rate Adaptation for Very High Throughput WLANs` by `Arafet Ben Makhlouf` and `Mounir Hamdi`, which used two spatial streams, we utilize three spatial streams to enhance throughput and reliability, optimizing performance in demanding environments with higher data requirements.

We implemented the $L3S$ algorithm in the $Ath9k$ driver and evaluated its performance across various experimental scenarios. This assessment will help us determine the adaptability, efficiency, and reliability of the $L3S$ algorithm compared to the $Minstrel$ algorithm and our findings will provide insights into the strengths and weaknesses of each approach.

# 2 Rate Adaptation Algorithm & Minstrel

## 2.1 Rate Adaptation Algorithm

Rate adaptation algorithms are essential for optimizing WLAN performance by dynamically selecting the best data rate based on current channel conditions. These algorithms ensure efficient use of the available bandwidth and enhance the reliability of wireless communication. They are generally categorized into two main types: statistic-based and signal-based algorithms. Statistic-based algorithms, such as ARF (Automatic Rate Fallback), AARF (Adaptive ARF), SampleRate, and ONOE, rely on historical transmission success ratios to make rate adaptation decisions. They monitor past transmissions and adjust the data rate based on the observed success or failure of packet deliveries. Signal-based algorithms, such as CHARM, RBAR (Receiver-Based Auto Rate), OAR (Opportunistic Auto Rate), and Goodput Analysis, utilize real-time signal measurements to determine the best data rate. They measure signal strength, quality, and other channel characteristics to make more informed decisions. Some advanced rate adaptation algorithms use a hybrid approach, combining Received Signal Strength Indicator (RSSI) and transmission statistics to achieve enhanced performance. This allows for a more balanced and accurate rate selection by leveraging both historical data and real-time channel conditions. The 802.11n standard supports both open-loop and closed-loop rate adaptation schemes. Open-loop schemes use implicit acknowledgment (ACK) feedback to make rate adaptation decisions, while closed-loop schemes rely on explicit Channel State Information (CSI) feedback from the receiver to optimize rate selection.

## 2.2 Minstrel

The Minstrel algorithm is a practical rate selection method for 802.11 wireless networks, implemented in the Linux kernel and the ns-3 network simulator. It effectively addresses packet loss, rate adaptation, and interaction with higher layer protocols like TCP by relying solely on acknowledgment feedback. Minstrel maintains a table of acknowledgment probability estimates for each neighbor and rate, using an exponential weighted moving average to smooth out probability estimations.

The algorithm probes 10% of frames to gather information on unused rates, balancing the need for information with the overhead of probing. These probing rates are chosen in a round-robin manner from a static list to avoid the computational cost of random number generation. Minstrel ensures that no two consecutive frames are probes and skips probing rates with over 95% success probability. Designed to handle varying link conditions, Minstrel aims to achieve the best available throughput while maintaining high reliability, adapting quickly to changing conditions to find the optimal rates for performance.

# 3 Multi-rate retry mechanism

The multi-rate retry mechanism implemented in the $Ath9k$ driver for Atheros chipsets enhances reliability by transmitting lost packets at progressively lower rates. Each frame is assigned four retry series (r0/c0, r1/c1, r2/c2, r3/c3), with specific rates and maximum transmission attempts. If a packet fails at the initial rate, it is retried at the next lower rate until it is successfully transmitted or all retries are exhausted. This process ensures robust communication by adapting to varying channel conditions. The $Ath9k$ driver uses the ONOE rate adaptation algorithm, which periodically updates throughput based on the Packet Error Rate (PER) and sends a small fraction of packets at adjacent rates to optimize performance.

# 4 Implementation of L3S Algorithm

The $L3S$ (Long-term Stability and Short-term Responsiveness) algorithm is designed to improve rate adaptation in $MIMO$ WLANs. This algorithm monitors the binary $ACKs$ as implicit feedback and does not rely on explicit feedback from the receiver. The key components of L3S include maintaining both short-term and long-term statistics to dynamically adjust transmission rates. The short-term statistics handle transient variations, while the long-term statistics manage sustained changes in link conditions. By continuously monitoring transmission results and intelligently probing at new data rates, $L3S$ can quickly adapt to varying channel conditions, optimizing throughput. The algorithm is implemented within the $Ath9k$ driver for 802.11n devices, ensuring compatibility with existing standards and achieving significant performance improvements over traditional rate adaptation mechanisms. In the $Ath9k$ driver, we added or modified existing code in order to implement the new algorithm. The files we changed are the following:

- **rc80211_minstrel_ht.c**

- **rc80211_minstrel_ht.h**
- **rc80211_minstrel_ht_debugfs.c**

## 4.1 Short-term Statistics

Short-term statistics control the transmission rate in response to transient variations, enhancing system responsiveness. Wireless links vary unpredictably, with $ACK$ frames indicating rapid improvements or deteriorations in channel conditions. Before probing, the sender distinguishes between short-term and long-term variations. The multi-rate retry mechanism uses counters for $consecutive\_retries$, $consecutive\_successes$, and $consecutive\_failures$ to adjust probing periods dynamically. If 10 consecutive $ACKs$ are received, probing is delayed, indicating stable conditions. Missing 2 consecutive $ACKs$ triggers an immediate fallback to the previous rate, while missing 4 accelerates probing due to degrading channel quality.

### 4.1.1 Rate Statistics function

The **rate_statistics** function updates transmission statistics for rate adaptation. It tracks consecutive successes and failures, adjusts probing intervals, and modifies transmission rates based on the success or failure of packet acknowledgments to ensure optimal performance in varying wireless conditions. It is important to get the statistics quite often and for all rates that attempt to transmit ($r_1$, $r_2$ and $r_3$), so we decided to call the function **rate_statistics** in the **minstrel_ht_tx_status** because is called whenever the transmission of a packed is finished. Also, we commented out the logic of minstrel to update statistics and rates that was included in **minstrel_ht_tx_status** so as not to affect the functionality of L3S. The use of counters $consecutive\_retries$, $consecutive\_successes$, and $consecutive\_failures$ is implemented in the **rate_statistics** function that we created based on the $L3S$ pseudocode. More specific:

- **consecutive_retries**: we add the number of retransmissions for the current rate. If a packet is transmitted without retries, the count is set to zero.
- **consecutive_successes** and **consecutive_failures**:
  - If there are no retries ($consecutive\_retries[i] == 0$) and the function is called for the last successful rate we consider the packet as acknowledged so:
    - we increment $consecutive\_successes$ by one and we reset $consecutive\_failures$ to zero.
    - if $consecutive\_successes$ reaches 10 ($consecutive\_successes >= 10$), we change the $probe\_interval$ to $90ms$ (as discussed in section 4.1.4).
  - If there are retries ($consecutive\_retries[i] >= 1$) the packet is considered as missed $ACK$ so:
    - we reset $consecutive\_successes$ to zero and increment $consecutive\_failures$ by the number of retries.

- if $consecutive\_failures$ reaches 4 ($consecutive\_failures >= 4$), we change the $probe\_interval$ to $10ms$.
- if $consecutive\_failures$ reaches 2 ($consecutive\_failures >= 2$), we trigger the **recovery** function and we change the $probe\_interval$ to $30ms$.

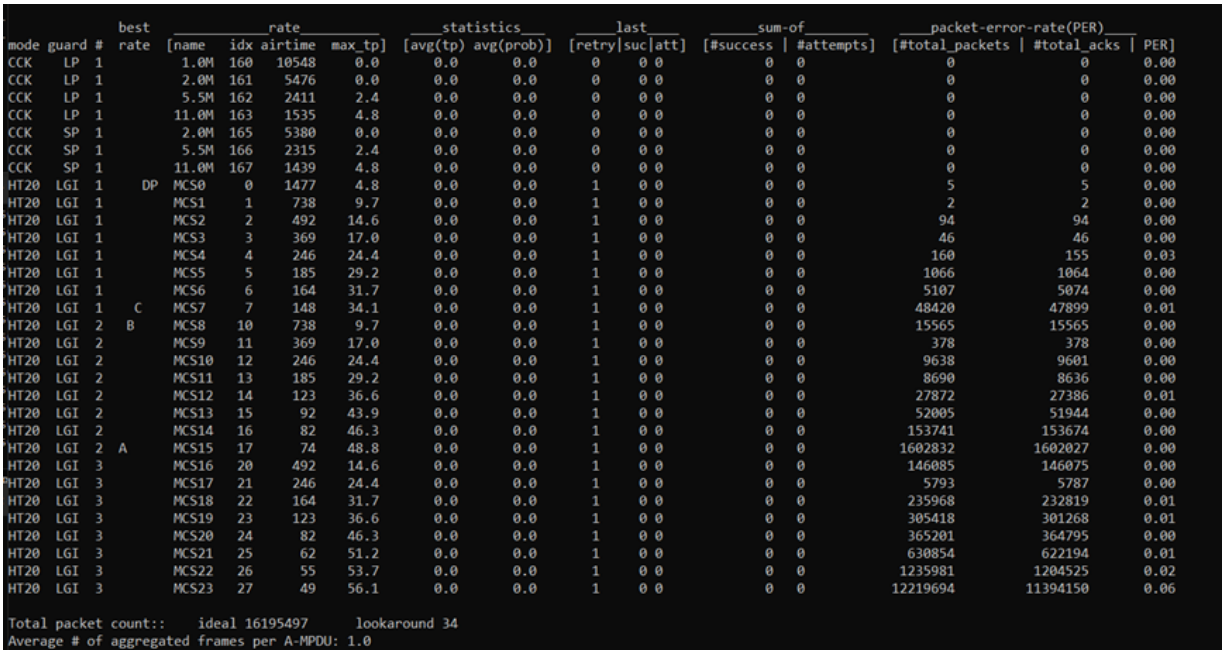
If this is the last successful rate:

- reset $consecutive\_failures$ and $consecutive\_retries$ for the next packet.
- update transmission statistics counters $total\_packets\_per\_mcs$ and $ack\_counter\_per\_mcs$ for each $MCS$.
- update the $PER(Packet - Error - Rate)$ statistic based on the number of packets transmitted and acknowledged in the current rate.
- call **get_rate** to determine the best transmission rate based on the updated statistics(is explained in the section 4.2.2).

#### 4.1.2 PER(Packet-Error-Rate) Statistics

- Calculate the index of the group for $MCS$ value(is explained in section 4.2.1):
- Update packet counters:
  - Increase the counter of total frames included in a packet, as if the packet is aggregated, one packet consists of more than one frame, otherwise one packet is assigned to one frame.
  - Increase the acknowledgment counter for all frames that were acknowledged in a packet.
- Calculate the PER statistics based on the index of the current rate:
  - If $total\_frames[index] \neq 0$:
    - $per[index] = \frac{100 \times (total\_frames[index] - acks[index])}{total\_frames[index]}$.

Also we changed the file **rc80211_minstrel_ht_debugfs.c** to show the per statistics for each rate in the table that is printed every 1 second. The figure below shows how the table changed and add one more column-section in it $packet - error - rate(PER)$ with $total\_packets = total\_frames$, $total\_acks$ and value of $PER$.



| mode | guard | # | best rate | rate [name | idx | airtime | max_tp] | statistics [avg(tp) | avg(prob)] | last [retry\|suc\|att] | sum-of [#success | #attempts] | packet-error-rate(PER) [#total_packets | #total_acks | PER] |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| CCK | LP | 1 | | 1.0M | 160 | 10548 | 0.0 | 0.0 | 0.0 | 0 0 0 | 0 0 | 0 | 0.00 |
| CCK | LP | 1 | | 2.0M | 161 | 5476 | 0.0 | 0.0 | 0.0 | 0 0 0 | 0 0 | 0 | 0.00 |
| CCK | LP | 1 | | 5.5M | 162 | 2411 | 2.4 | 0.0 | 0.0 | 0 0 0 | 0 0 | 0 | 0.00 |
| CCK | LP | 1 | | 11.0M | 163 | 1535 | 4.8 | 0.0 | 0.0 | 0 0 0 | 0 0 | 0 | 0.00 |
| CCK | SP | 1 | | 2.0M | 165 | 5380 | 0.0 | 0.0 | 0.0 | 0 0 0 | 0 0 | 0 | 0.00 |
| CCK | SP | 1 | | 5.5M | 166 | 2315 | 2.4 | 0.0 | 0.0 | 0 0 0 | 0 0 | 0 | 0.00 |
| CCK | SP | 1 | | 11.0M | 167 | 1439 | 4.8 | 0.0 | 0.0 | 0 0 0 | 0 0 | 0 | 0.00 |
| HT20 | LGI | 1 | DP | MCS0 | 0 | 1477 | 4.8 | 0.0 | 0.0 | 1 0 0 | 0 0 | 5 | 5 | 0.00 |
| HT20 | LGI | 1 | | MCS1 | 1 | 738 | 9.7 | 0.0 | 0.0 | 1 0 0 | 0 0 | 2 | 2 | 0.00 |
| HT20 | LGI | 1 | | MCS2 | 2 | 492 | 14.6 | 0.0 | 0.0 | 1 0 0 | 0 0 | 94 | 94 | 0.00 |
| HT20 | LGI | 1 | | MCS3 | 3 | 369 | 17.0 | 0.0 | 0.0 | 1 0 0 | 0 0 | 46 | 46 | 0.00 |
| HT20 | LGI | 1 | | MCS4 | 4 | 246 | 24.4 | 0.0 | 0.0 | 1 0 0 | 0 0 | 160 | 155 | 0.03 |
| HT20 | LGI | 1 | | MCS5 | 5 | 185 | 29.2 | 0.0 | 0.0 | 1 0 0 | 0 0 | 1066 | 1064 | 0.00 |
| HT20 | LGI | 1 | | MCS6 | 6 | 164 | 31.7 | 0.0 | 0.0 | 1 0 0 | 0 0 | 5107 | 5074 | 0.00 |
| HT20 | LGI | 1 | C | MCS7 | 7 | 148 | 34.1 | 0.0 | 0.0 | 1 0 0 | 0 0 | 48420 | 47899 | 0.01 |
| HT20 | LGI | 2 | B | MCS8 | 10 | 738 | 9.7 | 0.0 | 0.0 | 1 0 0 | 0 0 | 15565 | 15565 | 0.00 |
| HT20 | LGI | 2 | | MCS9 | 11 | 369 | 17.0 | 0.0 | 0.0 | 1 0 0 | 0 0 | 378 | 378 | 0.00 |
| HT20 | LGI | 2 | | MCS10 | 12 | 246 | 24.4 | 0.0 | 0.0 | 1 0 0 | 0 0 | 9638 | 9601 | 0.00 |
| HT20 | LGI | 2 | | MCS11 | 13 | 185 | 29.2 | 0.0 | 0.0 | 1 0 0 | 0 0 | 8690 | 8636 | 0.00 |
| HT20 | LGI | 2 | | MCS12 | 14 | 123 | 36.6 | 0.0 | 0.0 | 1 0 0 | 0 0 | 27872 | 27386 | 0.01 |
| HT20 | LGI | 2 | | MCS13 | 15 | 92 | 43.9 | 0.0 | 0.0 | 1 0 0 | 0 0 | 52005 | 51944 | 0.00 |
| HT20 | LGI | 2 | | MCS14 | 16 | 82 | 46.3 | 0.0 | 0.0 | 1 0 0 | 0 0 | 153741 | 153674 | 0.00 |
| HT20 | LGI | 2 | A | MCS15 | 17 | 74 | 48.8 | 0.0 | 0.0 | 1 0 0 | 0 0 | 1602832 | 1602027 | 0.00 |
| HT20 | LGI | 3 | | MCS16 | 20 | 492 | 14.6 | 0.0 | 0.0 | 1 0 0 | 0 0 | 146085 | 146075 | 0.00 |
| HT20 | LGI | 3 | | MCS17 | 21 | 246 | 24.4 | 0.0 | 0.0 | 1 0 0 | 0 0 | 5793 | 5787 | 0.00 |
| HT20 | LGI | 3 | | MCS18 | 22 | 164 | 31.7 | 0.0 | 0.0 | 1 0 0 | 0 0 | 235968 | 232819 | 0.01 |
| HT20 | LGI | 3 | | MCS19 | 23 | 123 | 36.6 | 0.0 | 0.0 | 1 0 0 | 0 0 | 305418 | 301268 | 0.01 |
| HT20 | LGI | 3 | | MCS20 | 24 | 82 | 46.3 | 0.0 | 0.0 | 1 0 0 | 0 0 | 365201 | 364795 | 0.00 |
| HT20 | LGI | 3 | | MCS21 | 25 | 62 | 51.2 | 0.0 | 0.0 | 1 0 0 | 0 0 | 630854 | 622194 | 0.01 |
| HT20 | LGI | 3 | | MCS22 | 26 | 55 | 53.7 | 0.0 | 0.0 | 1 0 0 | 0 0 | 1235981 | 1204525 | 0.02 |
| HT20 | LGI | 3 | | MCS23 | 27 | 49 | 56.1 | 0.0 | 0.0 | 1 0 0 | 0 0 | 12219694 | 11394150 | 0.06 |

Total packet count:: ideal 16195497 lookaround 34
Average # of aggregated frames per A-MPDU: 1.0

Figure 1: Table with info per rate.

### 4.1.3 Recovery

The **recovery** function is called inside the **rate_statistics** function as mentioned above in section 4.1.1.

- Loop through each rate $r_i$:
  - If the $index$ is $(0, 7]$ in a group:
    - Decrease the rate by $1$.
  - If the $index = 0$ in the $2^{nd}$ or $3^{rd}$ group:
    - Decrease the rate by $3$ to revert to the previous valid $MCS$ rate.
  - If the $index = 0$ in the $1^{st}$ group:
    - The rate remains the same in the $index = 0$.

### 4.1.4 Probing Mechanism

The probing mechanism is used to dynamically adjust transmission rates in response to changing wireless channel conditions, handling short-term variations and long-term trends to maintain optimal performance. To achieve probing the appropriate time, we set the variable $probe\_state = true$ in the **minstrel_ht_get_rate** which is called before the transmission of every frame. Our probing mechanism is implemented with two different ways. The first one is the **slow probing** mechanism, the second one is the **quick probing** and they are used to enable **Probe State** (as discussed in section 4.2.4).

- **Slow Probing**: In the slow probing mechanism, the goal is to reduce the times the algorithm does probing (enables $probe\_state$) so as to keep the rate more stable. To achieve this, whenever a change is taking place and it is time for the $probe\_interval$ to be set, we check if the current $probe\_interval$ is the same with the one that is going to be set. If the intervals are the same, we do not reset the $time\_of\_probing$ to current time (jiffies) as we suppose that the conditions of the channel did not changed significantly and we want our timer to continue counting. On the other hand, if the intervals are not the same, we suppose that the conditions of the channel changed significantly, so we need to probe to find a better rate for transmission and we reset $probe\_interval$ and $time\_of\_probing$. The probing is slow because we change the value of $probe\_interval$ quite often and in this way the timer may not finish until the next change of the $probe\_interval$ that resets the timer ($time\_of\_probing$).

- **Quick Probing**: In the quick probing mechanism, we want to probe whenever the channel conditions are changing, so the $time\_of\_probing$ variable is reset when the **minstrel_ht_get_rate** function is called and only if the timer has finished. In this way, we manage to probe often and change rate quickly if it is not satisfactory.

The $probe\_interval$ values are varying. Each value of $probe\_interval$ indicates different channel conditions. To be more specific:

- **Probe interval 10ms**: indicates that more than 4 $consecutive\_failures$ happened and the channel quality is degrading so it is important to probe quickly.

- **Probe interval 20ms**: is set by the **minstrel_ht_update_rates** function.

- **Probe interval 30ms**: indicates that two consecutive ACKs were missed.

- **Probe interval 60ms**: is the initialization value of probe_interval to probe for the first time after a suitable time

- **Probe interval 90ms**: indicates that more than 10 $consecutive\_successes$ happened and the channel quality is very good so there is no need to probe yet.

## 4.2 Long-term Statistics

Long-term Statistics adjust the transmission rate for optimal throughput by analyzing packet error rates and achieved throughput over a defined packet window. The algorithm probes candidate rates intelligently to identify potential improvements. During probing, the long-term rate remains unchanged but is compared with the probing rates. If a probing rate performs better, it is adopted; otherwise, the sender reverts to the previous rate (as discussed in section 4.1.3). This continuous adjustment allows quick adaptation to channel variations using a multi-rate retry series mechanism.

The long-term transmission rate, tx_rate, for each station is adjusted based on the sender's state. The state transitions and functions are illustrated in the figure below. Specifically, a sender operates in two states: the Tx state and the Probe state. In each round, the transmitter alternates between these states, continuously updating the associated statistics. When the sender adjusts the long-term $MCS$, statistics for both periods are reset, marking the start of a new round.

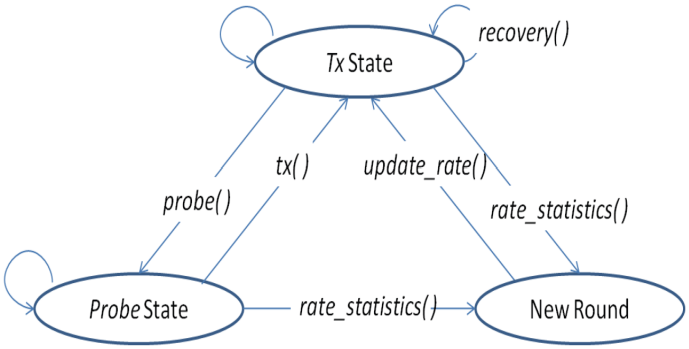



Figure 2: State transitions at the transmitter.

### 4.2.1 MCS groups

As we said above we implemented the code for 3 spatial streams so we have 3 different $MCS$ groups. In the $MCS$ groups of ath9k driver the indices 8 and 9 in each group are not utilized as we see below in the table so we don't choose them. The selected rate selected in the ath9k driver as the current rate of the sender is the idx variable assigned to $MCS$, for example if we have $MCS_{22}$ we have to change to index which is 26 in the table of $MCS$ rates($index = current\_rate.idx + (current\_rate.idx/8) * 2$). The values of $r_1$, $r_2$, and $r_3$ are the indices in the $MCS$ groups and not the $MCS$ values.

MCS Index - 802.11n and 802.11ac

802.11n 802.11ac

| HT MCS | VHT MCS | SS | Modulation | Coding | 20MHz No SGI | 20MHz SGI | 40MHz No SGI | 40MHz SGI | 80MHz No SGI | 80MHz SGI | 160MHz No SGI | 160MHz SGI |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 1 | BPSK | 1/2 | 6.5 | 7.2 | 13.5 | 15 | 29.3 | 32.5 | 58.5 | 65 |
| 1 | 1 | 1 | QPSK | 1/2 | 13 | 14.4 | 27 | 30 | 58.5 | 65 | 117 | 130 |
| 2 | 2 | 1 | QPSK | 3/4 | 19.5 | 21.7 | 40.5 | 45 | 87.8 | 97.5 | 175.5 | 195 |
| 3 | 3 | 1 | 16-QAM | 1/2 | 26 | 28.9 | 54 | 60 | 117 | 130 | 234 | 260 |
| 4 | 4 | 1 | 16-QAM | 3/4 | 39 | 43.3 | 81 | 90 | 175.5 | 195 | 351 | 390 |
| 5 | 5 | 1 | 64-QAM | 5/6 | 52 | 57.8 | 108 | 120 | 234 | 260 | 468 | 520 |
| 6 | 6 | 1 | 64-QAM | 3/4 | 58.5 | 65 | 121.5 | 135 | 263.3 | 292.5 | 526.5 | 585 |
| 7 | 7 | 1 | 64-QAM | 5/6 | 65 | 72.2 | 135 | 150 | 292.5 | 325 | 585 | 650 |
|  | 8 | 1 | 256-QAM | 3/4 | 78 | 86.7 | 162 | 180 | 351 | 390 | 702 | 780 |
|  | 9 | 1 | 256-QAM | 5/6 | n/a | n/a | 180 | 200 | 390 | 433.3 | 780 | 866.7 |
| 8 | 0 | 2 | BPSK | 1/2 | 13 | 14.4 | 27 | 30 | 58.5 | 65 | 117 | 130 |
| 9 | 1 | 2 | QPSK | 1/2 | 26 | 28.9 | 54 | 60 | 117 | 130 | 234 | 260 |
| 10 | 2 | 2 | QPSK | 3/4 | 39 | 43.3 | 81 | 90 | 175.5 | 195 | 351 | 390 |
| 11 | 3 | 2 | 16-QAM | 1/2 | 52 | 57.8 | 108 | 120 | 234 | 260 | 468 | 520 |
| 12 | 4 | 2 | 16-QAM | 3/4 | 78 | 86.7 | 162 | 180 | 351 | 390 | 702 | 780 |
| 13 | 5 | 2 | 64-QAM | 5/6 | 104 | 115.6 | 216 | 240 | 468 | 520 | 936 | 1040 |
| 14 | 6 | 2 | 64-QAM | 3/4 | 117 | 130.3 | 243 | 270 | 526.5 | 585 | 1053 | 1170 |
| 15 | 7 | 2 | 64-QAM | 5/6 | 130 | 144.4 | 270 | 300 | 585 | 650 | 1170 | 1300 |
|  | 8 | 2 | 256-QAM | 3/4 | 156 | 173.3 | 324 | 360 | 702 | 780 | 1404 | 1560 |
|  | 9 | 2 | 256-QAM | 5/6 | n/a | n/a | 360 | 400 | 780 | 866.7 | 1560 | 1733.3 |
| 16 | 0 | 3 | BPSK | 1/2 | 19.5 | 21.7 | 40.5 | 45 | 87.8 | 97.5 | 175.5 | 195 |
| 17 | 1 | 3 | QPSK | 1/2 | 39 | 43.3 | 81 | 90 | 175.5 | 195 | 351 | 390 |
| 18 | 2 | 3 | QPSK | 3/4 | 58.5 | 65 | 121.5 | 135 | 263.3 | 292.5 | 526.5 | 585 |
| 19 | 3 | 3 | 16-QAM | 1/2 | 78 | 86.7 | 162 | 180 | 351 | 390 | 702 | 780 |
| 20 | 4 | 3 | 16-QAM | 3/4 | 117 | 130 | 243 | 270 | 526.5 | 585 | 1053 | 1170 |
| 21 | 5 | 3 | 64-QAM | 5/6 | 156 | 173.3 | 324 | 360 | 702 | 780 | 1404 | 1560 |
| 22 | 6 | 3 | 64-QAM | 3/4 | 175.5 | 195 | 364.5 | 405 | n/a | n/a | 1579.5 | 1755 |
| 23 | 7 | 3 | 64-QAM | 5/6 | 195 | 216.7 | 405 | 450 | 877.5 | 975 | 1755 | 1950 |
|  | 8 | 3 | 256-QAM | 3/4 | 234 | 260 | 486 | 540 | 1053 | 1170 | 2106 | 2340 |
|  | 9 | 3 | 256-QAM | 5/6 | 260 | 288.9 | 540 | 600 | 1170 | 1300 | n/a | n/a |

Figure 3: $MCS$ groups with rates for $802.11n$ and $802.11ac$.

### 4.2.2 Get Rate function

In the **get_rate** function we call the **tx_state** or the **probe_state** function according the state in which the transmitter is located. Also, we change the rates and reset the short-term counters(inside the **minstrel_ht_update_rates** that is explained in section 4.2.3) when is called the probe state or if the state is changed from probe to tx state.

### 4.2.3 Update rates logic

Inside the **minstrel_ht_update_rates** function the counters are reset $consecutive\_successes$, $consecutive\_failures$ and $consecutive\_retries[\,]$, and if the rate is increased(group of current rate is increased related to group of previous rate or inside the same group the index is increased) the $probe\_interval$ changed to the value $20ms$.

#### 4.2.4 Tx State

In the multi-rate retry mechanism, $c_1$ is set to two, meaning the first and second attempts use the long-term rate during the transmission phase or the probe rate during the gauging phase. If consecutive failures occur, a rate reduction is likely, as these failures are more often caused by poor channel quality than by collisions. Similarly, $c_2$ and $c_3$ are also set to two retries, allowing for quick adaptation to short-term channel variations. The initial rate series, $r_0$, with a retry limit of four, is consistently used for sending control packets (such as $RTS$, $CTS$, and $ACK$) at the lower $MCS$ with a single antenna prior to data transmission. The $r_0$ for control packets is being handled by the ath9k driver so we don't need to adjust it in our implementation. In the **tx_state** function that is used inside the **get_rate** function we adjust the rates accordingly. We assume that the current rate is the last rate $r_1$, $r_2$ or $r_3$ that the latest packet is successfully transmitted and according to this current rate we make 3 decreasing consecutive rates.

- If the sender transmits with rate $r_1$:
  - Rates $r_1$, $r_2$, and $r_3$ remain the same.
- If the sender transmits with rate $r_2$:
  - Assign $r_1 = r_2$ and $r_2 = r_3$.
  - If $r_3$ is $index$ from $(0, 7]$ in a group, assign $r_3 = r_3 - 1$.
  - If $r_3$ is $index = 0$ in a group and not zero in the $group = 1$, assign $r_3 = r_3 - 3$.
- If the sender transmits with rate $r_3$:
  - Assign $r_1 = r_3$.
  - If $r_3$ is $index$ from $(0, 7]$ in a group:
    - Assign $r_2 = r_3 - 1$.
    - If $r_3 - 1$ is $index$ from $(0, 7]$ in a group, assign $r_3 = r_3 - 2$.
    - If $r_3 - 1$ is $index = 0$ and not zero in the $group = 1$, assign $r_3 = r_3 - 4$.
  - If $r_3$ is $index = 0$ and not zero in the $group = 1$, assign $r_2 = r_3 - 3$ and $r_3 = r_3 - 4$.
  - Otherwise keeps the same $r_2$ and $r_3$.

#### 4.2.5 Probe State

After a certain period, the sender initiates two successive probe series and enters the Probe state, without altering the tx_rate. It continues to monitor the probing results at the candidate rates. The first series probes, using the current spatial stream and stays in the same $MCS$ group, sending a constant fraction (10%) of the data at the two rates adjacent to the long-term rate. In the second series, the probing is based on the number of streams currently in use, which involves increasing or decreasing the $MCS$ group. Similarly, another

window of ten percent of the packets is sent at other candidate rates, which are always sorted in decreasing order. In the **probe_state** function we implement the two probe series that is used also in the **get_rate** function.

**First Probe Series**
The index that is mentioned below is referred to the index of the current rate.

- If the $index$ is in $(0, 7)$ in a group:
  - Assign $r_1 = idx + 1$, $r_2 = idx$, $r_3 = idx - 1$.
- If the $index = 7$ in a group:
  - Assign $r_1 = idx$, $r_2 = idx - 1$, $r_3 = idx - 2$.
- If the $index = 0$ in a group:
  - Assign $r_1 = idx + 2$, $r_2 = idx + 1$, $r_3 = idx$.
- Mark the first probe as complete by setting $first\_probe = true$ and set the $probe\_interval$ to $10ms$.

**Second Probe Series**
When the index of the current rate is in the middle group periodically we change the group either in the leftmost group or in the rightmost group.

- Reset the first probe indicator to $first\_probe = false$ and set the $probe\_interval$ to $60ms$.
- Check the current $MCS$ group and adjust probing direction:
  - If the group is the leftmost (indices $0$ to $7$):
    - Call the function to move to the right group.
  - If the group is the rightmost (indices $20$ to $27$):
    - Call the function to move to the left group.
  - If the group is the middle:
    - If it is time to move from the middle to the left group we call the function to move to the left group.
    - If it is time to move from the middle to the right group we call the function to move to the right group.
- The following functions manage the transitions between $MCS$ groups:
  - Function to move to the left $MCS$ group:
    - Set the rate $r_1 = idx$.
    - If the $index$ is in $[0, 7)$ in a group:

        - Assign $r_2 = idx - 9$.
        - Assign $r_3 = idx - 10$.
    - If the $index = 7$ in a group(-7 for $r_2$ because it is the immediately next $MCS$ index from -10 to avoid indices 8, 9):
        - Assign $r_2 = idx - 7$.
        - Assign $r_3 = idx - 10$.
- Function to move to the right $MCS$ group:
    - If the $index$ is in $(0, 7]$ in a group:
        - Assign $r_1 = idx + 10$.
        - Assign $r_2 = idx + 9$.
    - If the $index = 0$ in a group(+7 for $r_2$ because it is the immediately previous $MCS$ index from +10 to avoid indices 8, 9):
        - Assign $r_1 = idx + 10$.
        - Assign $r_2 = idx + 7$.
    - Set the rate $r_3 = idx$.

# 5 Performance Evaluation

## 5.1 Experiment Setup

The experiment was conducted using the NITLab indoor testbed, featuring a set of nodes configured to evaluate the performance of the Minstrel and L3S algorithms under different traffic conditions and interference scenarios. The setup consists of a main communication channel and two interfering channels to test the algorithms' adaptability and robustness in a controlled environment.

### 5.1.1 Nodes & Network Configuration

**Main Communication Channel**

- **Server Node**: Node081 (192.168.2.81)
- **Client Node**: Node085 (192.168.2.85)
- **Channel**: 5
- **Traffic Type**: UDP and TCP

**First Interfering Channel**

- **Server Node**: Node088 (192.168.2.88)
- **Client Node**: Node075 (192.168.2.75)
- **Channel**: 3
- **Traffic Type**: UDP

**Second Interfering Channel**

- **Server Node**: Node093 (192.168.2.93)
- **Client Node**: Node089 (192.168.2.89)
- **Channel**: 6
- **Traffic Type**: UDP

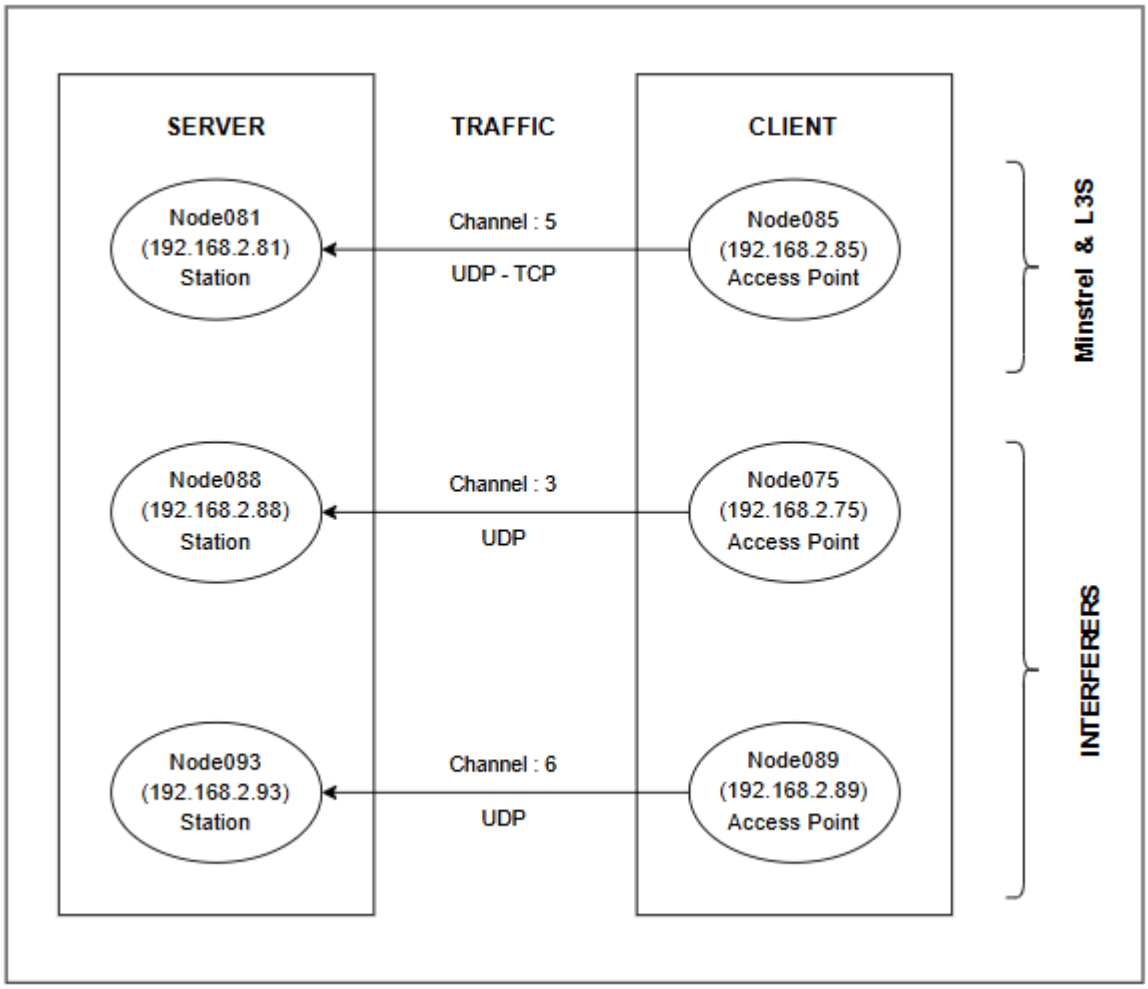



Figure 4: Topology of our experiment.

To accurately assess the performance of the Minstrel and L3S algorithms, we conducted three rounds of tests under different interference conditions: low, medium, and high interference. For each round, the parameters were kept consistent across all algorithms (Minstrel, L3S Quick, and L3S Slow). Each test was run three times, and the average results were taken to minimize the impact of any extraneous interference. A glimpse of the results will be shown with graphs later on, and all the results, including statistics from the Station and Access Points, will be included at the end of the analysis.

The interference levels were controlled by adjusting the parameters related to the noise and traffic in the network. The specific parameters for each interference level are as follows.

### Low Interference

- **Parameters**: (see `params_low.json`)

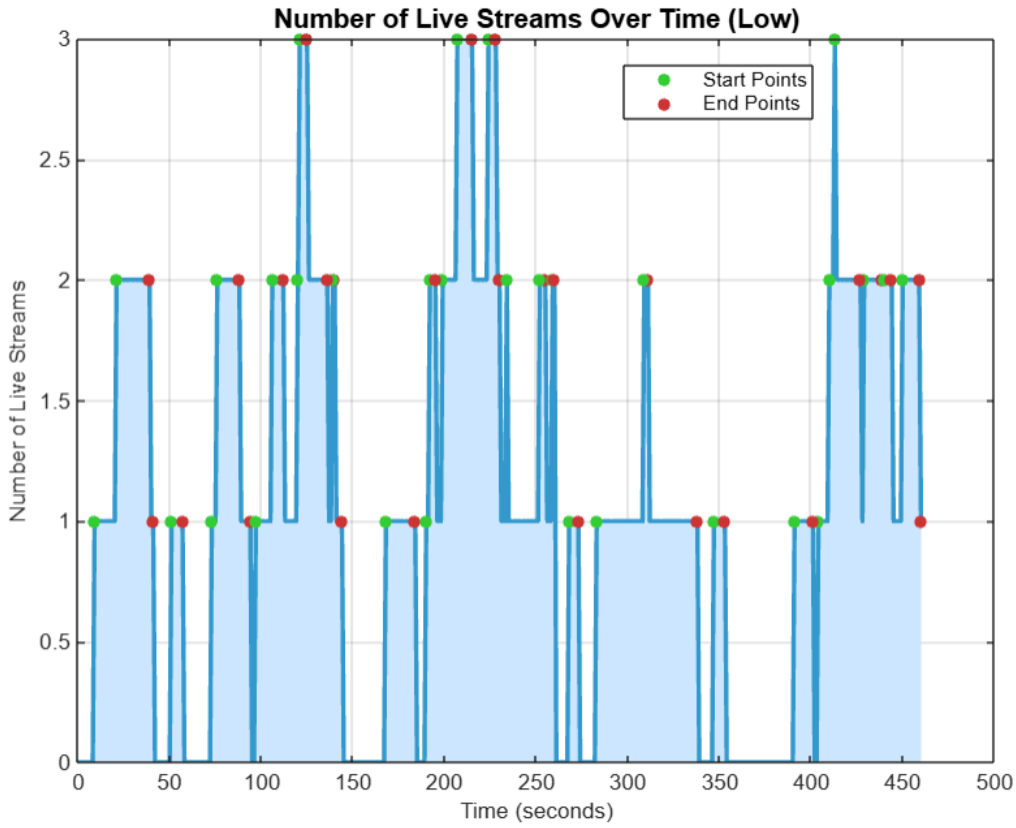



Figure 5: Streams for low interference per second.

**Medium Interference**

- **Parameters**: (see `params_med.json`)

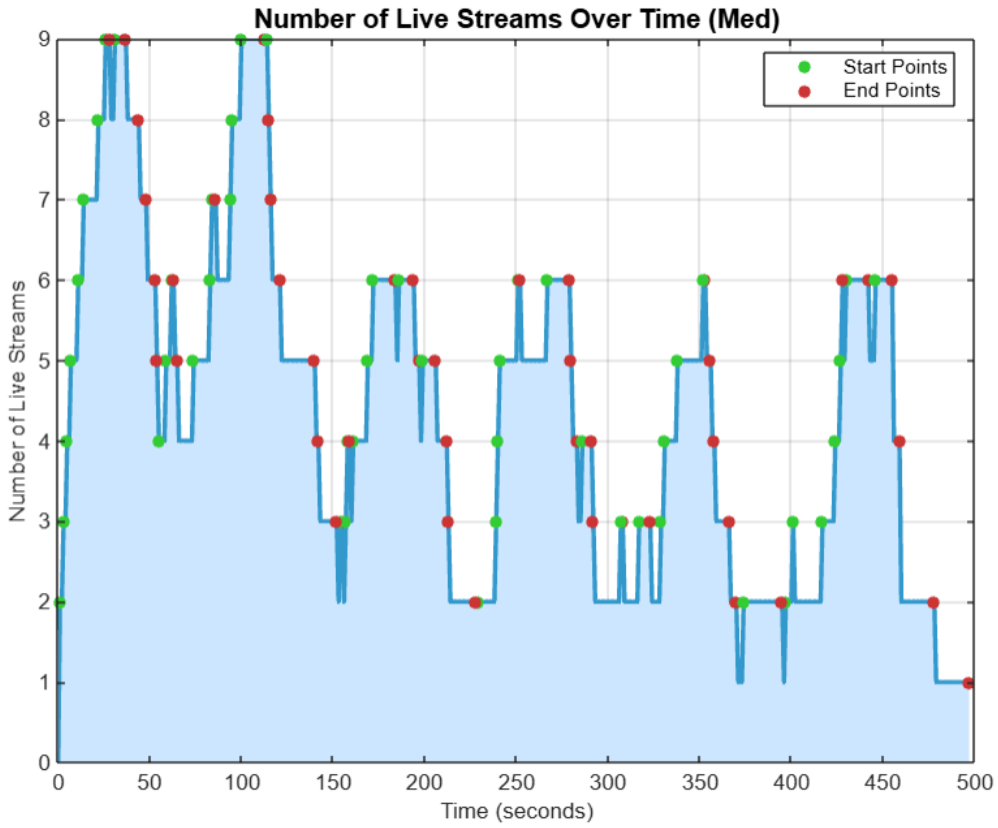



Figure 6: Streams for medium interference per second.

**High Interference**

- **Parameters**: (see `params_high.json`)

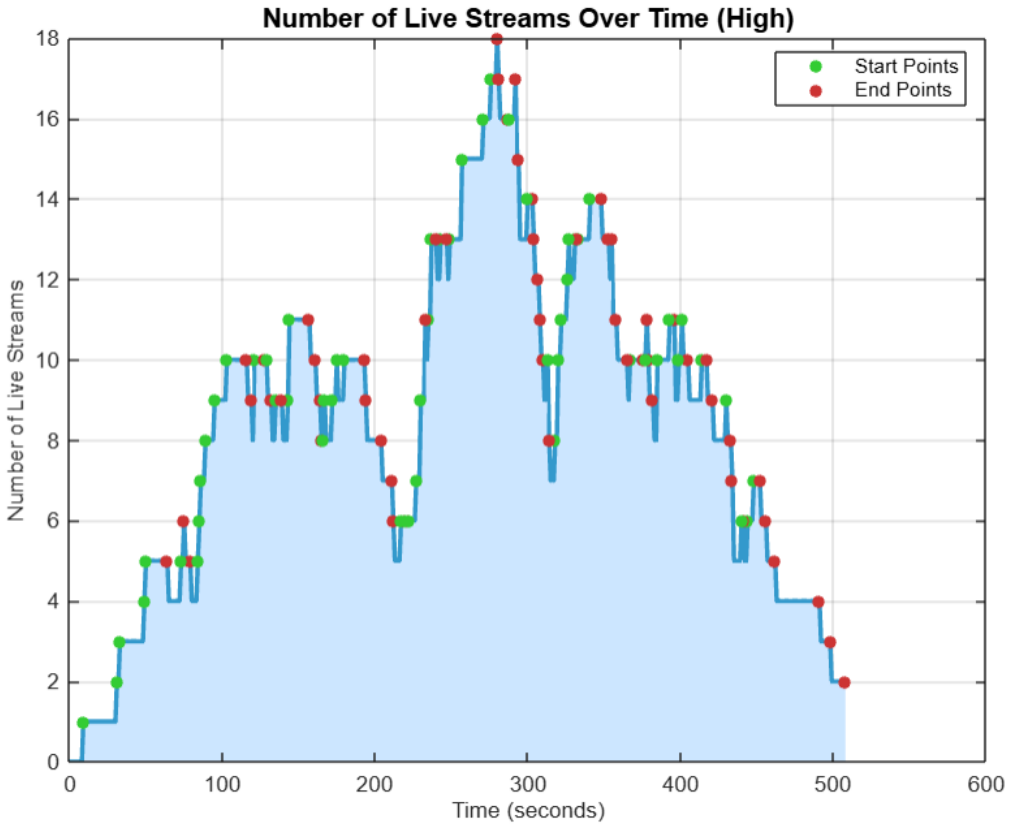



Figure 7: Streams for high interference per second.

All the interference parameters were generated randomly using a python script (also added at the end of the analysis) but were kept the same for algorithms to keep the experiment as objective as possible.

The following sections provide a detailed explanation of how the experiment was conducted, including the setup and execution phases, followed by an analysis of the results obtained.

1. **Setup**: Nodes were configured as per the network topology. The server and client nodes for the main communication channel and the interfering channels were set up with the specified IP addresses and traffic types.

2. **Interference Levels**: For each round, the interference levels were set according to the parameters in the provided JSON files. These parameters controlled the intensity and frequency of the noise introduced into the network.

3. **Traffic Generation**: Continuous traffic was generated using tools like *iperf* for UDP traffic (Interferers were sending only UDP traffic, UDP-TCP traffic were only present on the main channel as shown in the relevant photo). This ensured a consistent and measurable load on the network, allowing for accurate performance evaluation.

4. **Data Collection**: Throughput, latency, and packet loss metrics were collected during each test run. These metrics were averaged over three runs to account for any variability.

## 5.2 Results Analysis

**TCP (Transmission Control Protocol)**

TCP is a connection-oriented protocol that ensures reliable delivery of data packets between devices. It establishes a connection before data transfer, and packets are acknowledged upon receipt. If a packet is lost or corrupted, TCP retransmits it, ensuring the data arrives intact and in order. This reliability makes TCP suitable for applications where data integrity is crucial, such as web browsing, email, and file transfers. However, TCP's overhead in maintaining the connection and ensuring reliability can result in higher latency and reduced throughput under certain conditions, especially in networks with high interference or varying link quality.

**UDP (User Datagram Protocol)**

UDP is a connectionless protocol that sends data packets without establishing a connection and without guaranteeing delivery, order, or integrity. It is faster and more efficient for applications where speed is more critical than reliability, such as streaming audio/video, online gaming, and live broadcasts. Due to the lack of acknowledgment and retransmission, UDP has lower latency and can achieve higher throughput in optimal conditions. However, it is more susceptible to packet loss and errors, which can degrade performance in networks with high interference.

### 5.2.1 Low Interference

**UDP Throughput**

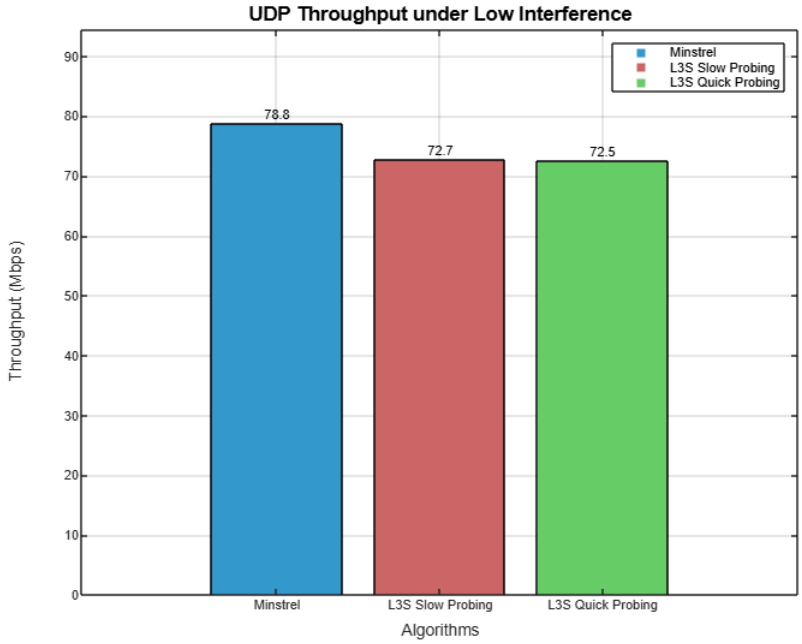



Figure 8: Overall average of achieved UDP throughput with low interference.

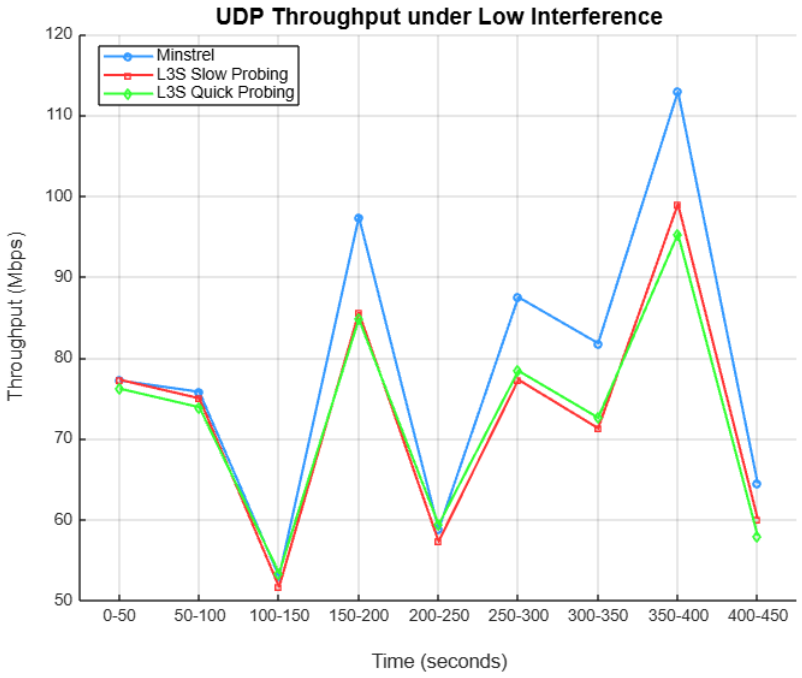



Figure 9: Average of achieved UDP throughput every 50 seconds with low interference.

**First Observation**: In low interference conditions, Minstrel achieves the highest UDP throughput, slightly outperforming both L3S Slow and L3S Quick Probing. This indicates

Minstrel's efficiency in stable environments with minimal interference.

### TCP Throughput

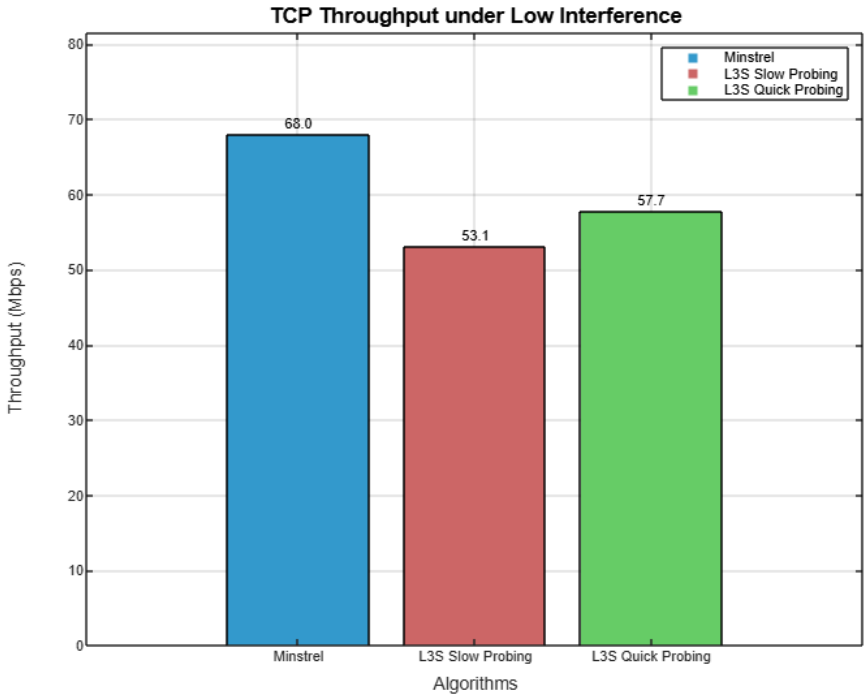



Figure 10: Overall average of achieved TCP throughput with low interference.

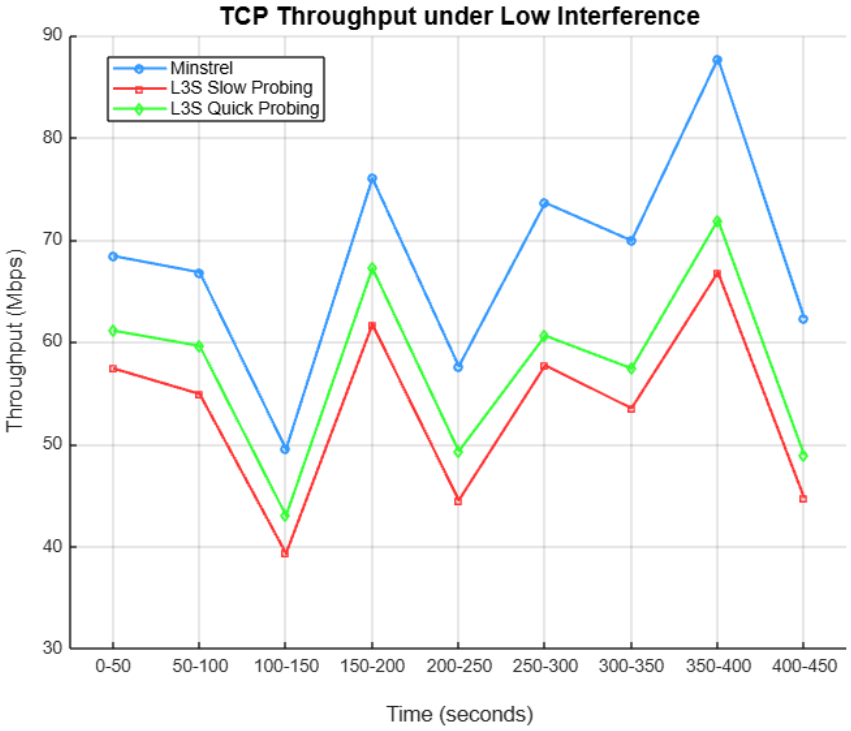



Figure 11: Average of achieved TCP throughput every 50 seconds with low interference.

**First Observation**: In low interference conditions, Minstrel achieves the highest TCP throughput, significantly outperforming both L3S Slow and L3S Quick Probing. This suggests Minstrel's robustness in maintaining high throughput with TCP's overhead.

### 5.2.2 Medium Interference

**UDP Throughput**

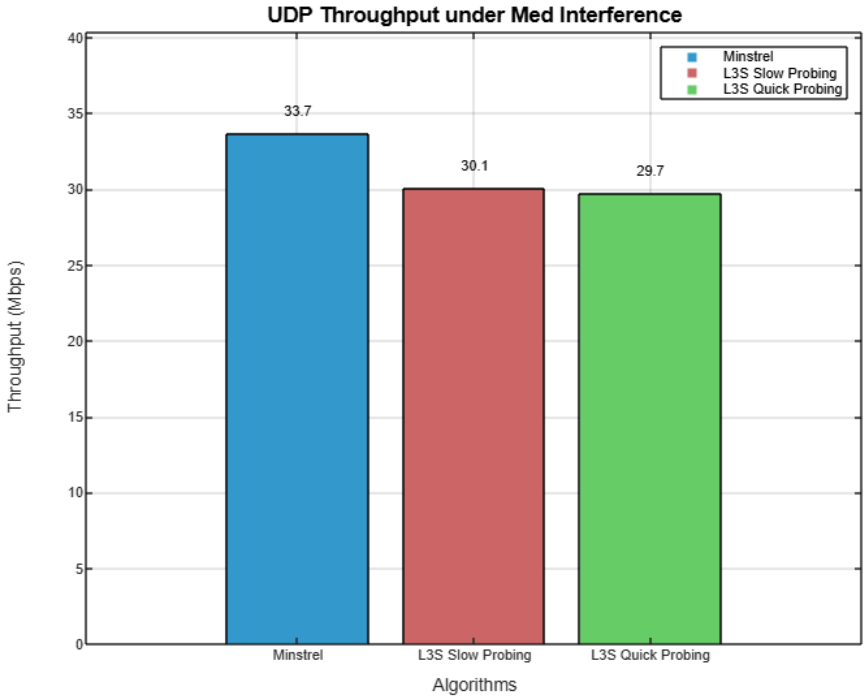



Figure 12: Overall average of achieved UDP throughput with medium interference.

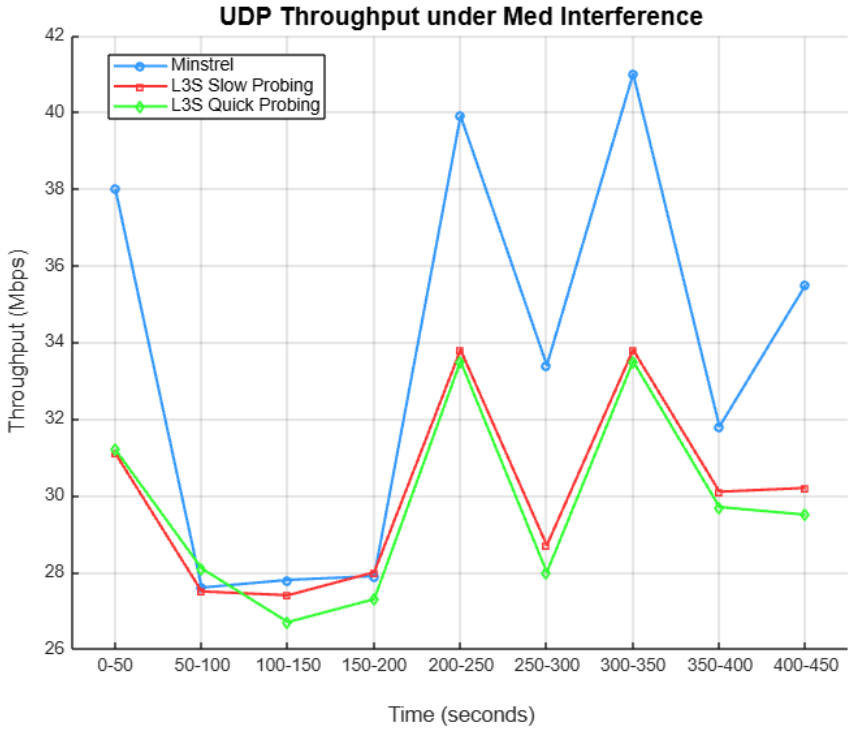



Figure 13: Average of achieved UDP throughput every 50 seconds with medium interference.

**First Observation**: Under medium interference, the throughput decreases for all algorithms. Minstrel maintains a higher throughput compared to both L3S variants, though the gap narrows, suggesting L3S's potential in moderately dynamic environments.

### TCP Throughput

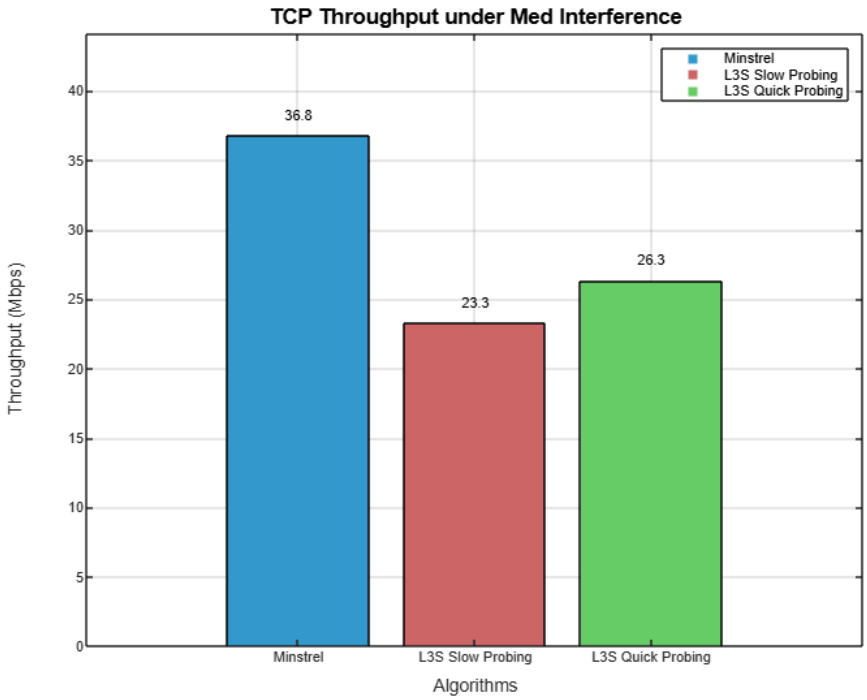



Figure 14: Overall average of achieved TCP throughput with medium interference.

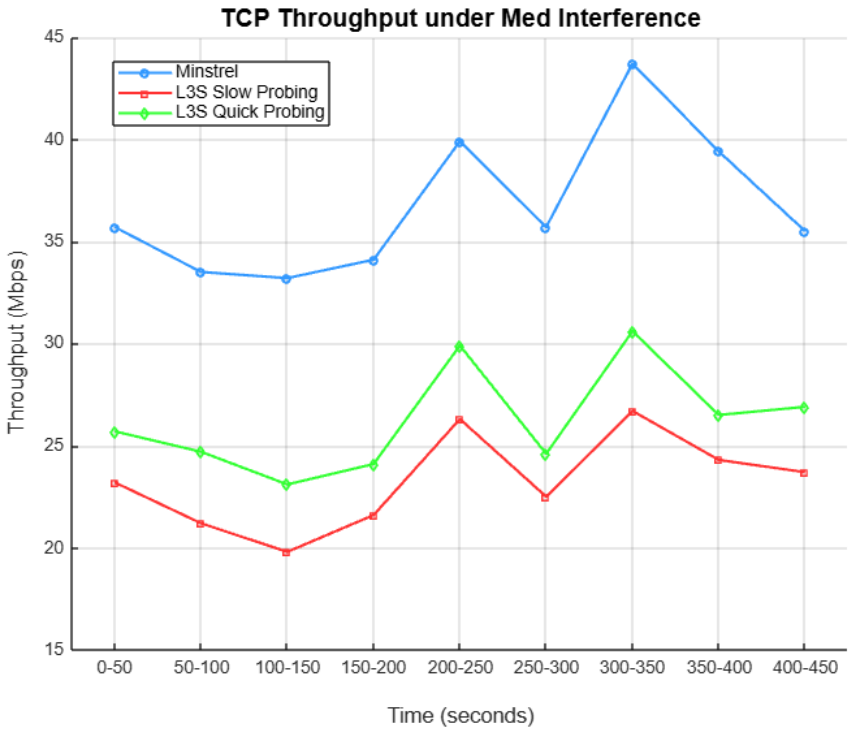



Figure 15: Average of achieved TCP throughput every 50 seconds with medium interference.

**First Observation**: With medium interference, Minstrel still leads in TCP throughput, but the performance gap with L3S Quick Probing narrows, highlighting its capability to adapt in moderately challenging environments.

### 5.2.3 High Interference

**UDP Throughput**

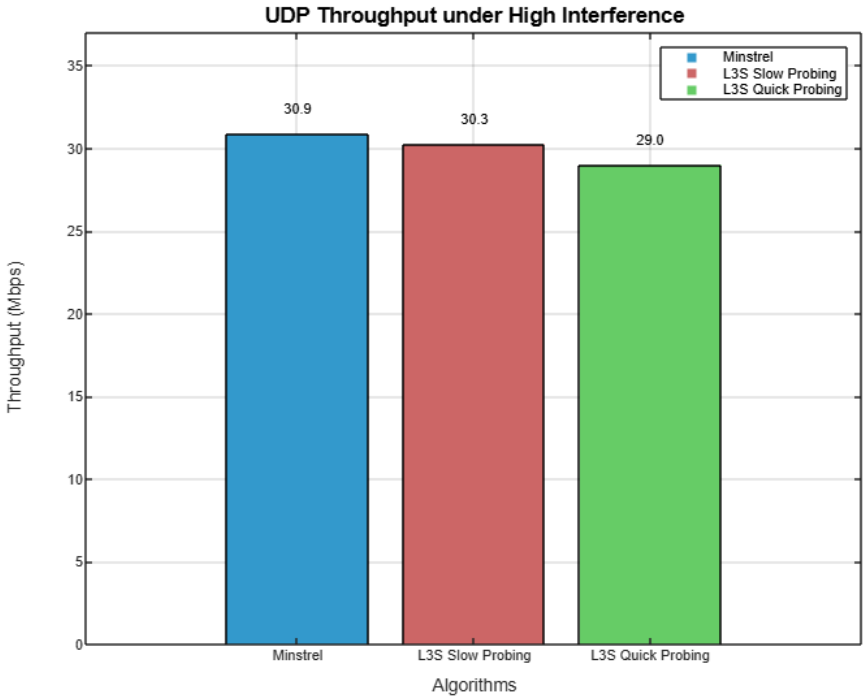



Figure 16: Overall average of achieved UDP throughput with high interference.

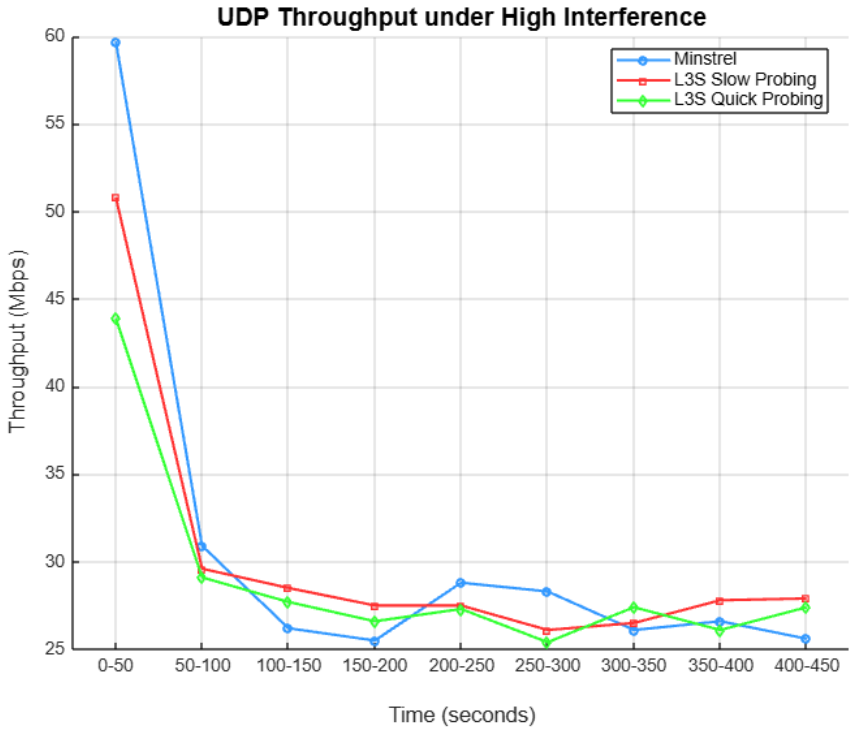



Figure 17: Average of achieved UDP throughput every 50 seconds with high interference.

**First Observation**: In high interference conditions, the throughput further decreases. Minstrel shows slightly higher performance than L3S Slow Probing, with L3S Quick Probing trailing closely behind. The performance difference is minimal, indicating that L3S's adaptive features help maintain comparable performance under severe interference.

### TCP Throughput

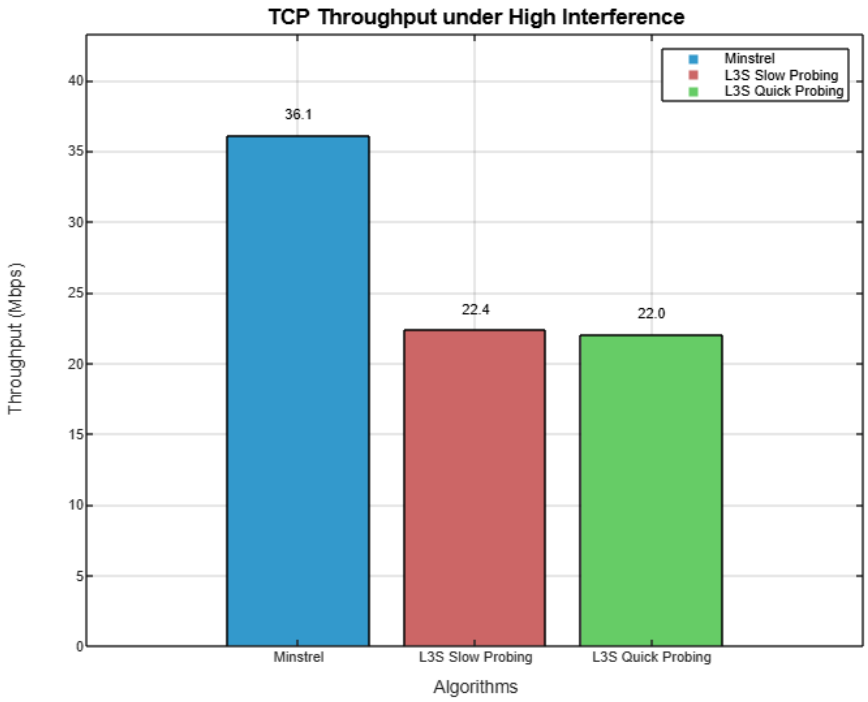



Figure 18: Overall average of achieved TCP throughput with high interference.

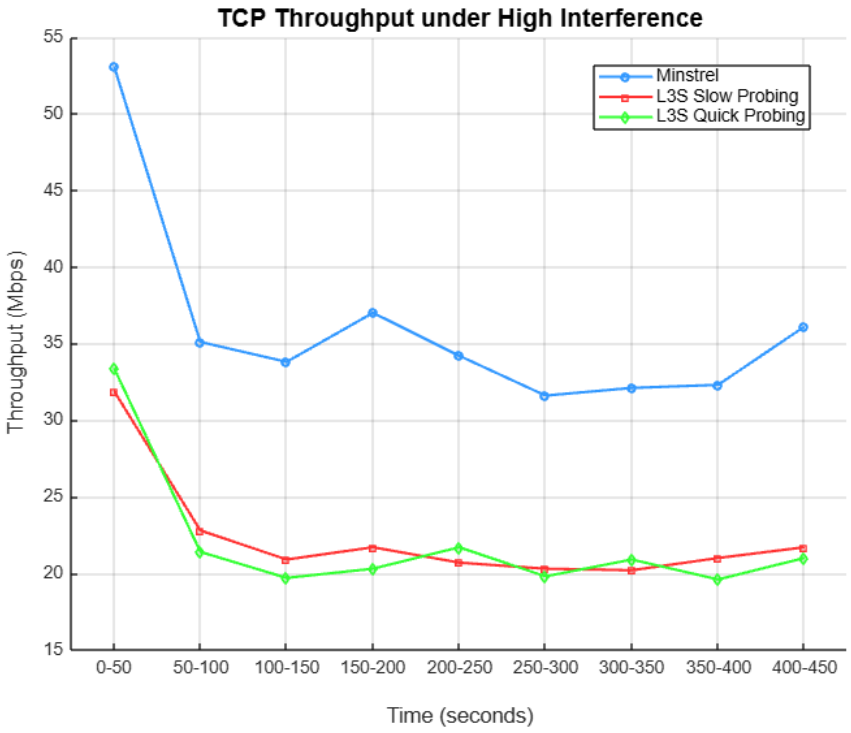



Figure 19: Average of achieved TCP throughput every 50 seconds with high interference.

**First Observation**: Under high interference conditions, Minstrel maintains higher TCP throughput compared to L3S Slow and L3S Quick Probing. However, the performance differences are less pronounced, indicating that L3S's adaptive mechanisms help it to perform reasonably well under severe interference.

## 5.3 Analysis & Interpretation

### 5.3.1 Throughput Differences

- **UDP vs. TCP**: UDP generally achieves higher throughput than TCP across all levels of interference. This is expected because UDP has lower overhead and does not perform retransmissions, unlike TCP, which ensures reliable delivery through acknowledgments and retransmissions.

- **Overall**: Minstrel achieves higher throughput than L3S in both UDP and TCP scenarios, particularly in low and medium interference conditions. This indicates Minstrel's efficiency in environments with stable or moderate interference.

### 5.3.2 Algorithm Performance

- **Minstrel**: Minstrel's performance indicates its efficiency in stable environments with low interference. It consistently outperforms L3S in both TCP and UDP traf-

fic. This superiority is more pronounced in scenarios with lower interference levels, where Minstrel can fully leverage its rate selection mechanism.

- **L3S Quick Probing vs. L3S Slow Probing**: L3S Quick performs better than L3S Slow, demonstrating that faster adaptation to changing network conditions helps maintain higher throughput. Despite this, both L3S variants still trail behind Minstrel, particularly in lower interference scenarios.

### 5.3.3 Interference Impact

- **Throughput Decrease**: Throughput decreases for all algorithms as interference levels rise, reflecting the challenges in maintaining high performance under noisy and congested conditions. The increased packet collisions and retransmissions under high interference significantly impact TCP performance due to its overhead.
- **Resilience of L3S**: The performance gap between Minstrel and L3S narrows as interference increases, suggesting that L3S's design may offer better resilience to higher interference, even if it doesn't surpass Minstrel in overall throughput. L3S's adaptive features help it maintain comparable performance under severe interference conditions.

## 5.4 Detailed Analysis of Graphs

### 5.4.1 UDP Throughput Analysis

1. **Low Interference (UDP)**
    - **Minstrel**: Achieves the highest throughput (~80-100 Mbps), with noticeable peaks and troughs indicating rapid adaptation to optimal rates.
    - **L3S Slow Probing and Quick Probing**: Both perform similarly (~70-90 Mbps), but consistently lower than Minstrel. L3S Quick Probing shows slightly more stability than L3S Slow Probing.
2. **Medium Interference (UDP)**
    - **Minstrel**: Fluctuates significantly but generally maintains higher throughput (~32-38 Mbps) compared to L3S.
    - **L3S Slow Probing and Quick Probing**: Both have lower throughput (~28-32 Mbps), with occasional spikes. L3S Quick Probing generally follows L3S Slow Probing closely, showing similar performance trends.
3. **High Interference (UDP)**
    - **Minstrel**: Starts with the highest throughput (~58 Mbps) but drops significantly, stabilizing around 28-30 Mbps.

- **L3S Slow Probing and Quick Probing**: Both start lower (~48 Mbps and ~42 Mbps, respectively) and stabilize around 25-30 Mbps. The performance of both L3S variants is very close, with L3S Slow Probing slightly outperforming L3S Quick Probing.

### 5.4.2 TCP Throughput Analysis

1. **Low Interference (TCP)**
   - **Minstrel**: Consistently achieves higher throughput (~60-70 Mbps) with peaks indicating efficient rate adaptation.
   - **L3S Slow Probing and Quick Probing**: Perform similarly (~50-60 Mbps), but with more pronounced fluctuations compared to Minstrel. L3S Quick Probing shows a slight edge over L3S Slow Probing, particularly in the mid-test intervals.
2. **Medium Interference (TCP)**
   - **Minstrel**: Shows more stable performance (~30-38 Mbps), maintaining higher throughput than L3S throughout the test.
   - **L3S Slow Probing and Quick Probing**: Both exhibit lower throughput (~20-28 Mbps) with L3S Quick Probing generally performing better than L3S Slow Probing, especially after the initial 50 seconds.
3. **High Interference (TCP)**
   - **Minstrel**: Begins with high throughput (~52 Mbps) and stabilizes around 30-35 Mbps. It maintains a consistent advantage over L3S variants.
   - **L3S Slow Probing and Quick Probing**: Both start lower (~25 Mbps) and stabilize around 20-25 Mbps. L3S Quick Probing slightly outperforms L3S Slow Probing in the latter half of the test.

## 5.5 Summary

From the experiments that took place above, we conclude that Minstrel is better in many cases, especially with TCP traffic on the main channel, but appearing unstable with more rate changes. On the other hand, L3S appears more stable in both implementations. To discuss the difference we are seeing on the graphs between the two types of traffic , its important to mention that in TCP traffic, the congestion avoidance mechanism influences positively the packet loss witch comes into conflict with the results we saw while running with UDP traffic.

# 6 Conclusion

The results indicate that while Minstrel generally outperforms L3S in terms of throughput, especially in low and medium interference conditions, L3S Quick Probing shows promise in maintaining competitive performance under high interference. This suggests that L3S could be more suitable for environments with highly dynamic conditions, where rapid adaptation is crucial. The differences in throughput between TCP and UDP further highlight the impact of protocol overhead and reliability mechanisms on performance, with UDP's lower overhead allowing for higher throughput, while TCP's reliability mechanisms result in lower but more consistent throughput. The performance trends observed in the graphs reinforce these conclusions, demonstrating the strengths and weaknesses of each algorithm under varying network conditions.

# References


[1] Practical Rate Adaptation for Very High Throughput WLANs Arafet Ben Makhlouf, Mounir Hamdi, 2013, IEEE

[2] Rate Adaptation for 802.11 Wireless Networks: Minstrel Andrew McGregor, Derek Smithies

[3] MCS Index - 802.11n and 802.11ac