Автономная некоммерческая организация высшего образования «Университет Иннополис»

ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА (БАКАЛАВРСКАЯ РАБОТА) по направлению подготовки 09.03.01 - «ИНФОРМАТИКА И ВЫЧИСЛИТЕЛЬНАЯ ТЕХНИКА»

GRADUATION THESIS (BACHELOR'S GRADUATION THESIS) Field of Study 09.03.01 – «COMPUTER SCIENCE »

Направленность (профиль) образовательной программы «Информатика и вычислительная техника» Area of Specialization / Academic Program Title: «Computer Science»

Тема / Topic	**Performance evaluation of delayed acknowledgment scheme of TCP over Wi-Fi / Оценка эффективности работы механизма delayed acknowledgment в протоколе TCP через Wi-Fi соединения**

Работу выполнил / Thesis is executed by	**Авхадеев Альберт Фанисович / Avkhadeev Albert**	подпись / signature
Руководитель выпускной квалификационной работы / Supervisor of Graduation Thesis	**Златанов Никола / Zlatanov Nikola**	подпись / signature
Консультанты / Consultants	**Сейтназаров Шынназар / Shinnazar Seytnazarov**	подпись / signature

Иннополис, Innopolis, 2025

Contents

List of Tables

List of Figures

Abstract

This thesis compares the performance of delayed acknowledgment (DACK) mechanisms in the Transmission Control Protocol (TCP) over Wi-Fi networks, with a focus on comparing the default Linux implementation and an adaptive algorithm called TCP-AAD [1]. Traditional DACK strategies, originally designed for wired networks, often perform inefficiently in Wi-Fi environments due to variable latency and MAC-layer frame aggregation (FA).

TCP-AAD improves upon these limitations by dynamically adjusting acknowledgment timing based on inter-arrival times (IATs) of packets, enhancing responsiveness while reducing channel contention. The algorithm was implemented in the Linux kernel (v6.12.9) and evaluated through real-world experiments under both local (LAN) and wide-area network (WAN) conditions.

Results show that TCP-AAD achieves a consistent throughput improvement of approximately 9% compared to the default DACK mechanism in both LAN and WAN setups with all modern congestion control mechanisms (CCM) used. These findings demonstrate the potential of adaptive acknowledgment strategies to enhance TCP efficiency in modern wireless environments, while maintaining compatibility with existing systems.

Chapter 1

Introduction

Nowadays, the Transmission Control Protocol (TCP) is a core component of the network protocol suite, responsible for the reliable and ordered delivery of packets between endpoints [2]. The standard is used by many services, such as web browsing (HTTP/HTTPS), file transfer (FTP), and various other services operating at the application layer [3].

In parallel, research in computer science has introduced new methods of device communication, providing an alternative to traditional Ethernet cables. Radio waves, used in the IEEE 802.11 protocol (commonly known as the Wi-Fi standard), have eliminated the need for a physical medium, enabling mobile interaction between network devices [4]. However, this new technology has also introduced unexpected challenges due to its physical characteristics.

The initial design of TCP was based on the assumption of a reliable wired medium, where packet loss was typically interpreted as a sign of network congestion. In contrast, wireless connections may lose segments due to collisions or signal interference. These issues lead to performance degradation, most notably reflected in reduced throughput. Moreover, Wi-Fi networks operate over a shared medium, where each packet competes for access to the channel. Even small segments, such as acknowledgments (ACKs), consume the same amount of transmission resources as large data packets with payloads, resulting in channel over-utilization and an increased probability of collisions [5].

To address the mentioned challenges and reduce the number of ACKs, improvements to the delayed acknowledgment (DACK) strategy were developed. One such method is the adaptive acknowledgment delay algorithm (TCP-AAD), which is based on the frame aggregation (FA) technique introduced in later versions of the IEEE 802.11 protocol (n/ax) [1]. Although the proposed solution showed good improvements over the standard algorithm in a simulated network environment[1], it did not consider behavior in real world situations, where delayed acknowledgment has already been heuristically adjusted for wireless networks.

The goal of this article is to evaluate the performance of the default Linux TCP delayed acknowledgment algorithm, as implemented in kernel version 6.12.9, compared to TCP-AAD under real network conditions. The study focuses mainly on performance metrics such as throughput and retransmission behavior.

1.1 Contribution

This paper proposes the following contributions:

- An analysis of the delayed acknowledgment mechanism in the Linux kernel version 6.12.9, focusing on the flow of functions and the core logic behind the timer calculation.

- Reproduction and integration of the TCP-AAD algorithm into the existing Linux kernel for future evaluation.

[1]For testing purposes, the NS-3 simulator was used.

- Design and construction of a controlled experimental environment under real-world conditions.
- Performance evaluation of the adaptive acknowledgment delay algorithm compared to the default implementation in the Linux kernel.

1.2 Thesis structure

The work is organized into the following structure:

1. **Literature Review** – A review of research on delayed acknowledgment mechanisms, including abstract recommendations from Request for Comments (RFC) standards, academic strategies for optimizing behavior over wireless media and the intuition behind TCP-AAD (Chapter 2).
2. **Methodology** – Represents full explanation and implementation details for TCP-AAD, examines packet receiving flow in the Linux kernel with an emphasis on DACK, representation of the experimental environment along with the test cases chosen (Chapter 3).
3. **Implementation** – Provides a detailed explanation of the modifications made to the Linux kernel, including the integration of the TCP-AAD algorithm and changes to the network module in version 6.12.9 (Chapter 4).
4. **Results and Discussion** – Presents and interprets the performance comparison between TCP-AAD and the default DACK implementation. The chapter discusses the effectiveness of the proposed algorithm and highlights observed limitations and trade-offs (Chapter 5).

5. **Conclusion and Future Work** – Summarizes the findings and proposes potential directions for future research (Chapter 6).

Chapter 2

Literature review

The purpose of this chapter is to provide a comprehensive overview of the delayed acknowledgment mechanism—starting from its initial design requirements, through essential enhancements, to the final usable version implemented in modern operating systems, with a particular focus on Linux. In addition, special attention is given to several key aspects: the Request for Comments (RFC) standards, which serve as the main source for defining Internet protocols; research efforts aimed at addressing challenges introduced by wireless communication, including TCP-AAD; and the current implementation of the delayed acknowledgment mechanism in the Linux kernel.

2.1 RFC-Based Specification

2.1.1 Foundation rules

RFC 1122 defines delayed acknowledgment as a mechanism for improving transmission efficiency by reducing the number of acknowledgment packets sent by the receiver, significantly minimizing network contention.[1] According to the standard, an acknowledgment should be sent for at least every two full-sized segments received or when a 500-millisecond timeout occurs. However, if acknowledgments are delayed too long, this can negatively impact the accuracy of Round-Trip Time (RTT) estimation, significantly reducing overall TCP performance [6].

2.1.2 Future improvements

Subsequent research and updates to TCP standardization documents have clarified the interaction between acknowledgment behavior and congestion control, although none of these RFCs explicitly redefines the delayed acknowledgment mechanism. However, these later standards introduce important contextual constraints that any modern DACK implementation must take into account.

RFC 2581 introduced fundamental TCP congestion control techniques, including slow start and congestion avoidance, both of which rely on acknowledgment behavior. During slow start, the sender increases the congestion window (CWND) exponentially with each received ACK. Therefore, delaying acknowledgments can slow CWND growth—particularly in low-latency or bursty traffic scenarios where quick feedback is crucial [7].

RFC 5681, which updates RFC 2581, added the fast retransmit and fast recovery mechanisms. It emphasizes the importance of accurate round-trip time estimation for effective congestion control. If ACKs are delayed beyond reasonable bounds, they can distort RTT measurements and postpone loss detection, ultimately reducing efficiency during recovery [8].

RFC 7323 introduces TCP timestamps to improve RTT accuracy, especially

[1]The paper presents a remote login case study in which delayed ACKs reduce the number of server-transmitted frames by a factor of three through packet consolidation.

in environments where delayed ACKs are used. Each data segment includes a timestamp, which is echoed back by the receiver in its acknowledgment. This allows the sender to estimate the RTT independently of ACK timing. To work well under such conditions, modern delayed acknowledgment implementations must be compatible with timestamp-based estimation and should avoid delay patterns that negatively impact TCP performance [9].

RFC 8312 defines CUBIC, a congestion control algorithm now used as the default in most Linux TCP implementations. Unlike earlier algorithms, which increased the CWND with every incoming ACK, CUBIC determines window growth based on the time elapsed since the last packet loss. This design helps CUBIC remain stable in networks with variable latency, including those with delayed or infrequent ACKs. However, if acknowledgments arrive in irregular bursts, CWND growth can become unstable, potentially causing unfair bandwidth distribution across multiple connections [10].

2.2 Delayed ACK Strategies in Research Literature

While the concept of delayed acknowledgment is well established in TCP standards, its practical behavior in wireless environments has been the subject of extensive research. Originally developed to reduce ACK traffic in stable, low-loss wired networks, DACK has shown performance limitations when applied to wireless systems like Wi-Fi, where variable delays and random losses are common.

Early work by Balakrishnan et al. [11] revealed that TCP mechanisms such as DACK significantly degrade throughput in wireless networks. This degradation is primarily due to delayed feedback interfering with the sender's congestion window growth, especially in environments with high packet error rates.

Altman and Jiménez [12] proposed dynamically adjusting the acknowledgment delay based on network conditions. Their findings showed that fixed ACK delay timers negatively affect multihop wireless flows, while adaptive mechanisms improve performance by reacting more quickly during bursty traffic or loss recovery.

Jiwei Chen et al. [13] introduced an enhanced delayed ACK mechanism for wireless networks. Their approach adjusts ACK timing dynamically based on observed packet arrival behavior, aiming to improve TCP responsiveness while minimizing delay. This strategy aligns with the goals of TCP-AAD, which seeks to balance reduced ACK overhead with timely congestion feedback.

In summary, these studies demonstrate that fixed delayed ACK strategies are poorly suited for modern wireless networks. This has led to the development of adaptive methods such as TCP-AAD, which aim to maintain performance under varying wireless conditions.

2.3 TCP-AAD algorithm

The main part of this research focuses on the implementation of the adaptive acknowledgment delay algorithm. Thus, it is important to understand the theoretical background and design rationale behind it. The core concept of TCP-AAD is based on the characteristics of modern Wi-Fi protocols, particularly the use of frame aggregation,[2] which combines multiple MAC-layer data units into a single frame to significantly reduce network overhead [14].

Detection of such aggregation patterns is achieved by analyzing timestamp differences between received TCP packets, a metric known as inter-arrival time

[2]The algorithm is based on the IEEE 802.11n protocol and its subsequent versions.

(IAT). A time gap between data segments that exceeds a heuristically defined threshold may indicate the start of a new aggregation frame. By sending an ACK just before the beginning of the next group of data packets, network contention can be reduced. According to simulation results, this method improved throughput by approximately 5% compared to the default scheme used in the ns-3 network simulator [1].

However, despite these promising results, the research did not include an analysis of DACK as implemented in modern operating systems. In the simulation environment the delayed acknowledgment behavior was modeled using simplified parameters, such as a fixed number of packets to wait for and a static timer assigned at the creation of the TCP socket, representing a simplified version of the OS-level network stack[3]. Therefore, it is essential to examine the actual implementation of the DACK mechanism in real-world systems.

[3]https://www.nsnam.org/docs/models/html/tcp.html

Chapter 3

Methodology

This chapter presents the methodology used to evaluate and enhance the performance of TCP delayed acknowledgment strategies under wireless conditions. The primary objective is to implement the TCP-AAD algorithm within the Linux kernel and compare its behavior against the default DACK mechanism through controlled and real-world experiments. The methodology follows two main components: the design and integration of the TCP-AAD algorithm and the configuration of an experimental testbed to observe performance under various network conditions.

The default Linux DACK mechanism relies on heuristics tuned for general scenarios, often falling short in dynamic wireless environments with variable latency and MAC-layer aggregation. To address this, the TCP-AAD algorithm adapts acknowledgment timing based on inter-arrival time (IAT) metric, aiming to optimize channel utilization and feedback efficiency. This chapter details the implementation of this logic at the kernel level and describes the tools, parameters, and environments used for empirical validation.

3.1 Design of TCP-AAD algorithm

The aggregation aware delay acknowledgment algorithm is designed to improve timing by detecting frame aggregation behavior in modern Wi-Fi networks.

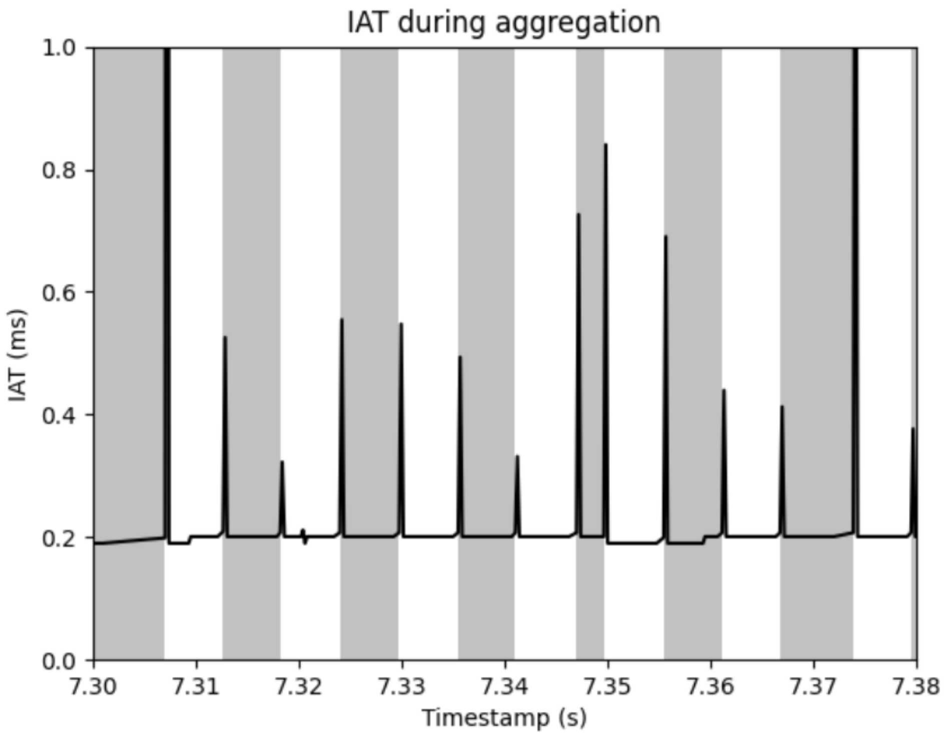

Fig. 3.1. Inter-arrival time (IAT) variation across aggregation frames, shown with alternating colors.

The Fig 3.1 represents IAT spikes that could be detected as aggregated frames. By using the intuition behind FA, the next formula was obtained:

$$T = (Iat_{min} \cdot \alpha + Iat_{curr} \cdot (1 - \alpha)) \cdot \beta \tag{3.1}$$

Here:

- Iat_{min} — the smallest inter-arrival time seen so far,
- Iat_{curr} — the inter-arrival time of the current packet,

- α — historical weight factor (default 0.75),
- β — tolerance parameter to avoid unnecessary ACKs (default 3),
- *T* — time-offset value after which an ACK should be sent (ATO analogy).

Additionally, the logic includes:

- Discarding too small IATs (< 0.2ms)
- Resetting IAT_{min} with some period to adapt the formula to constant changes in network conditions
- Immediate acknowledgemnt in case of getting out-of-order packets

Here is the pseudo-code of ACK logic in TCP-AAD:

Listing 3.1: Pseudocode implementation of TCP-AAD

```
/* Initialization */
Iat_min = +inf;
Iat_curr = +inf;
maxDelayedSegments = 2;
delayedSegments = 0;
resetDelay = 1s;
lastResetTime = 0;
maxDelayTimeout = 0.5s;
alpha = 0.75;
beta = 1.5;

/* On new data packet arrival */
if (CurrentTime >= lastResetTime + resetDelay) {
    Iat_min = +inf;
    lastResetTime = CurrentTime;
}

Iat = CurrentTime - PreviousPacketTime;
```

```

if (Iat is not too small) {
    Iat_curr = Iat;
    Iat_min = min(Iat, Iat_min);
}

delayedSegments += 1;

if (Packet is out-of-order) {
    delayedSegments = 0;
    sendACK();
} else {
    cancelActiveTimeout();

    if (delayedSegments < maxDelayedSegments) {
        T = maxDelayTimeout;
    } else {
        T = (Iat_min * alpha + Iat_curr *
            (1 - alpha)) * beta;
        T = min(T, maxDelayTimeout);
    }

    scheduleTimeout(CurrentTime + T);
}

/* When timeout expires */
onTimeoutExpire() {
    delayedSegments = 0;
    sendACK();
}
```

3.2 Existing ACK delaying algorithm in Linux kernel

To establish a baseline for evaluating alternative acknowledgment strategies such as TCP-AAD, it is essential to understand how delayed acknowledgment is currently implemented in the Linux TCP stack. The Linux kernel includes a dynamic acknowledgment delay mechanism based on heuristics derived from RFC guidelines and runtime conditions.

This section outlines the main components of the Linux DACK implementation, focusing on how acknowledgment delays are calculated, scheduled, and triggered. The description is based on kernel version 6.12.9, which was used as the basis for the experimental comparison in this thesis.

One of the key characteristics of the Linux TCP delayed acknowledgment mechanism is its dynamic behavior. Rather than relying on a fixed delay timer, the Linux implementation adapts the acknowledgment timing based on several metrics such as packet arrival intervals, round-trip time, and socket state [15].

The key components of the delayed acknowledgment mechanism can be categorized into three main functions:

- Calculating *ATO* (time-offset) before sending an ACK - tcp_event_data_recv
- Setting delayed ACK timer - tcp_send_delayed_ack
- Timer callback on expiration - tcp_delack_timer_handler

1.Calculating *ATO* before sending an ACK (tcp_event_data_recv)

This function is called when the TCP socket receives a new in-order data segment. Core logic includes checking how much time has passed since the last received packet - inter-arrival time (i.e. iat_{curr}), and adjusting the time-offset accordingly. The aim is to avoid acknowledging every single packet while not introducing too much delay. Hence, the algorithm finds a trade-off between reliability and channel over utilization.

- If no time-offset is assigned (i.e., first packet received), quick ACK mode is activated and ATO is initialized to TCP_ATO_MIN (e.g., 40 jiffies[1]). This starts the delayed ACK engine.
- If the last packet was received within a very short interval (less than or equal to 20 jiffies), ATO is reduced using the formula:
  ATO = (ATO'[2] / 2) + (TCP_ATO_MIN / 2)

  For example, if ATO' = 100 jiffies, the new ATO becomes 70 jiffies. This allows faster ACKs for fast-arriving segments.
- If the last packet was received within a moderate time interval (i.e., greater than 20 jiffies but less than the last ATO), the ATO is updated to reflect the new inter-arrival time:
  ATO = (ATO' / 2) + iat_{curr}

  For example, if ATO' = 100 jiffies and iat_{curr} = 60 jiffies, then the new ATO becomes 110 jiffies. If this exceeds retransmission timeout (RTO), it is adjusted to RTO.
- If the last packet was received after a long delay (i.e., more than the retransmission timeout RTO, e.g., m > 300 jiffies), quick ACK mode is reactivated to quickly respond and resynchronize. The ATO is not changed in this case.

[1]Jiffy is relative value used in DACK timer that is based on CPU clocking. Nowadays, all modern CPU's have such equivalence: 1 jiffy = 1 ms. For simplicity, let us assume the following equivalence is true in our implementation.

[2]Previous captured value for ATO.

Here is the simplified representation of the function tcp_event_data_recv implementing such logic:

Listing 3.2: Calculate time-offset for receiving packet

```
Input: socket, tcp packet
Constants: min_timeout, RTO

now := current time

if ATO is not set then
    ATO := min_timeout  // typically 40ms
    enable quick ACK mode
else
    delta := now - last_recv

    if delta <= min_timeout / 2 then
        ATO := ATO / 2 + min_timeout / 2

    else if delta < ATO then
        ATO := ATO / 2 + delta
        if ATO > RTO then
            ATO := RTO

    else if delta > RTO then
        enable quick ACK mode
```

One important part of the Listing 3.2 is the ability to dynamically change the mode of ACK. This helps the system respond quickly during critical situations, such as connection start or long wait for the packet. The function also keeps track of receive window updates and round-trip time to make sure congestion and data flow are managed correctly. The implementation mentioned above provide the ability to dynamically change TCP delayed ACK timing, allowing the system to balance between responsiveness and network efficiency based on current conditions.

2. Setting delayed ACK timer (tcp_send_delayed_ack)

The function mainly responsible for scheduling a delayed acknowledgment by setting a timer after a calculated offset. The timeout computed earlier is adjusted based on connection-specific factors such as recent RTT measurements, ping-pong heuristics, or whether PUSH flags are set on received data.

Internally, the function verifies whether a delayed ACK timer is already pending. If not, it schedules a new timer based on the estimated acknowledgment timeout. If a timer exists, it only updates it, if the newly computed value have sooner time expiration. This ensures minimal delay while avoiding redundant rescheduling.

The timeout is dynamically influenced by the smoothed RTT estimate (srtt_us). In low-latency environments, the delay is shortened to preserve responsiveness. In cases like ping-pong interactions—where the socket alternates frequently between sending and receiving—aggressive ACK behavior is preferred to maintain low turnaround time.

Overall, listing 3.3 represents a key part of the Linux TCP stack's adaptive ACK mechanism. It enables the protocol to reduce overhead while ensuring timely feedback in interactive or time-sensitive contexts.

Listing 3.3: Set timer for calculated time-offset

```
Input: socket

Constants:
    delay_timeout_min
    delay_timeout_max

Variables:
    max_ato    // Right bound of dACK timeout
    ato        // Already calculated time-offset from socket
```

```

if ato > delay_timeout_min then
    if interactive_mode or pushed_data then
        max_ato := delay_timeout_max

    if srtt_us exists then
        rtt := max(srtt_us / 8, delay_timeout_min)
        if rtt < max_ato then
            max_ato := rtt

    ato := min(ato, max_ato)
end if

ato := min(ato, tcp_delack_max(socket))    // Clamp to safe upper bound
timeout := now + ato                                // Schedule ACK time

if ACK timer already pending then
    if timeout_pending <= now + ato / 4 then
        send_ack_now(sk)
        return
    if timeout < timeout_pending then
        timeout := timeout_pending

delack_timer := timeout
Schedule delack_timer
```

3. Timer callback on expiration (tcp_delack_timer_handler)

The function tcp_delack_timer_handler reflecting the final step in the delayed acknowledgment process. It is invoked when the ACK timer expires. The main purpose of the callback is to send an ACK back to the sender and update statistics with DACK count.

Listing 3.4: Handle routine when timer expires

```
Function: handle_delayed_ack_timer(connection)

if connection is in CLOSED or LISTEN state then
    exit handler

if compressed ACKs are pending then
    refresh timestamp
    send compressed SACK ACK
    exit handler

if ACK timer is not pending then
    exit handler

if current time < timer expiration then
    reschedule timer to existing expiration
    exit handler

clear ACK timer pending flag

if ACK is still scheduled then
    if not in ping-pong mode then
        double ATO (adaptive timeout), up to RTO max
    else
        exit ping-pong mode
        reset ATO to minimum value

    refresh timestamp
    send ACK packet
    increment delayed ACK statistics counter

end handler
```

3.2.1 Migrating to High-Resolution Delayed ACK Timer

As mentioned previously, default DACK mainly relies on jiffies timer that is milliseconds precision, but the design of TCP-AAD requires larger resolution to effectively distinguish between aggregated frames.

To support the precision mentioned, the delayed ACK system should migrate from a jiffies-based timing mechanism to a high-resolution timer (hrtimer)[3], which operates in nanoseconds and enables more accurate control over ACK scheduling.

High-Resolution Timer Change

Here is short breakdown of changes made to satisfy our goals:

- Replace basic function for setup, init, reset and stop timer to appropriate listed in docs
- Updated the ACK scheduling logic to store timeouts in microseconds and convert them accordingly for hrtimer use.
- Registered a custom tcp_delack_hrtimer() callback function that replicates the functionality of tcp_delack_timer() on timer expiration.

In the next chapter we will explore in details migration process with code snippets.

3.3 Testing environment setup

After integrating new aggregation-aware ACK algorithm it is required to set up experimental environment. We will divide the section on several parts

[3]https://docs.kernel.org/timers/hrtimers.html

that are hardware configuration, software tools and evaluation metrics that will be considered during tests. The experimental setup consists of the following components:

- **Sender node**: Samsung NP300E5X laptop, running Ubuntu 22.04 LTS, connected to the router via a wired Ethernet interface. The system uses the default Linux kernel 6.12.9.
- **Receiver node**: HP Pavilion Gaming Laptop 13-ec1500ur, also running Ubuntu 22.04, with the patched Linux kernel incorporating the TCP-AAD algorithm.
- **Access Point**: TP-Link TL-WDR4300 (5GHz band) running OpenWRT firmware. Transmission rate is set to constant with using a 40 MHz channel width with Short Guard Interval (SGI) enabled and two spatial streams(2x2 MIMO) [16]. This configuration corresponds to Modulation and Coding Scheme (MCS) 13, which results in a fixed data rate of 240 Mbps[4].

3.3.1 Network topology

For testing purposes, we designed two network topologies. The first relies on communication between devices within the same LAN, allowing for a controlled environment under real physical conditions. The second topology involves a sender and receiver in the same area, but the traffic is routed through the WAN to a server located in Moscow (approximately 807 km away), which then forwards the traffic back to the receiver. This setup is intended to simulate real network behavior under uncontrolled conditions. Detailed topology diagrams are presented below.

[4]https://en.wikipedia.org/wiki/IEEE_802.11n-2009#Data_rates

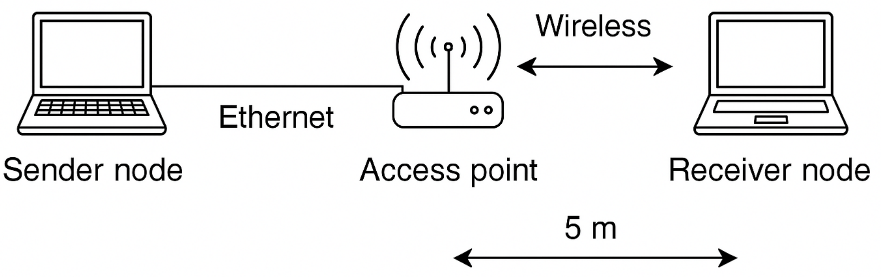

Fig. 3.2. Network topology of devices communicating in the same LAN

In the network layout presented in Fig. 3.2 the sender is connected via Ethernet for stable and consistent upstream delivery, while the receiver connects through the Wi-Fi at a fixed distance of 5 meters from the AP.

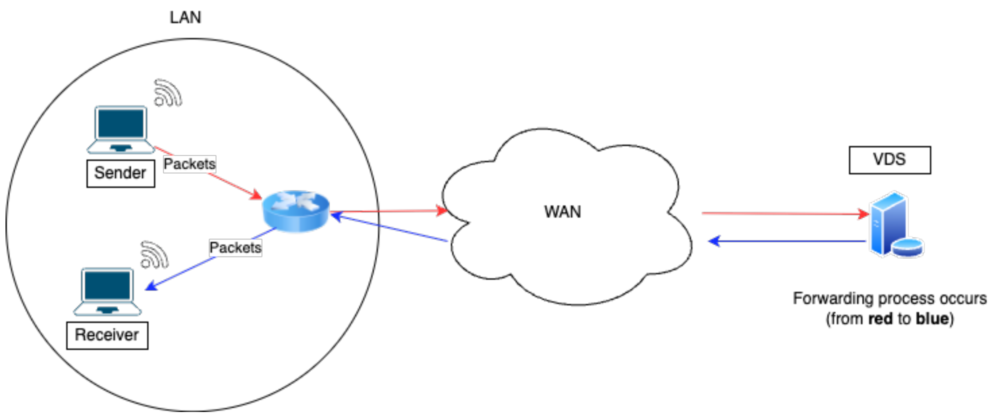

Fig. 3.3. Network topology of devices communicating throug the WAN

The network topology in Fig. 3.3 introduces packet flow that emulates long-path TCP behavior. Both the sender and receiver are connected to the same wireless LAN; however, packets do not travel directly between them. Instead, traffic initiated by the sender is routed through the wireless router to a remote Virtual Dedicated Server (VDS) over the WAN. The VDS is configured to forward these packets back into the LAN, where they are delivered to the receiver via the same Wi-Fi access point.

This topology forces all TCP traffic to traverse an external wide-area network segment. It enables evaluation of TCP performance (e.g., acknowledgment behavior) under conditions that more closely to Internet, like round-trip delays and routing complexity, despite the physical closeness of the two communicating hosts.

3.3.2 Software and Tools

The following open-source tools were used for traffic generation, logging, and analysis:

- iperf2.0 [17] – used to generate long-lived TCP streams between the sender and receiver.
- Wireshark [18] – packet capture and visualization tool used for analyzing TCP segment behavior and ACK traffic.
- Traffic Control (tc)[5] - utility to change network conditions by enforcing bandwidth limits through the Token Bucket Filter (TBF) and, when required, to simulate delay or loss using the netem module. This allowed controlled emulation of constrained wireless environments during the tests.

[5]https://man7.org/linux/man-pages/man8/tc.8.html

3.3.3 Evaluation Metrics

To compare TCP-AAD with the default DACK behavior, the following metrics were selected:

- Throughput (Mbps) – measuring the sustained transfer rate.
- Retransmissions – indicating loss sensitivity and congestion response.

Each test was executed with the following protocol configurations:

- Nagle's algorithm disabled because of conflict possibility with DACK [19].
- Selective Acknowledgment (SACK) disabled to ensure all ACK traffic (if possible) will be processed by delayed acknowledgment mechanism [20].
- Network offloading features (GRO, GSO, TSO) disabled to avoid taking the work of the algorithm by network interface card (NIC)[6]
- Fixed transmission rate explained in details in Section 3.3

3.3.4 Test case description

Each test run lasted for 5 minutes. To ensure statistical significance, each configuration (TCP-AAD and default DACK[7]) was executed 10 times under identical hardware and software conditions, with results averaged across runs. Additionally, both algorithms were tested under real (without any restriction used) and controlled network conditions using tc, including delay settings of 10 ms, 50 ms, and 100 ms, and bandwidth restrictions set to 10 Mbps, 50 Mbps, and 100 Mbps[8]. These

[6]https://tcpack.blogspot.com/2015/07/tcp-ack-generation-rate-simple-but-also.html

[7]Each test case was tested as separate kernel version built.

[8]Bandwith limits and delay restrictions were enabled only for topology described in figure 3.2.

variations allowed a comprehensive evaluation of performance across a range of realistic wireless scenarios.

TABLE 3.1
Test Matrix for TCP-AAD vs. Default DACK

Test ID	**Topology**	**Delay limit**	**Bandwidth**	**TCP Variant**	**Purpose**
T1	LAN	None	Unlimited	Default DACK	Baseline TCP comparison in
T2	LAN	None	Unlimited	TCP-AAD	unconstrained Wi-Fi
T3	LAN	10 ms	Unlimited	Default DACK	Evaluate performance under
T4	LAN	10 ms	Unlimited	TCP-AAD	light artificial delay
T5	LAN	50 ms	Unlimited	Default DACK	Test behavior with moderate
T6	LAN	50 ms	Unlimited	TCP-AAD	Wi-Fi delay
T7	LAN	100 ms	Unlimited	Default DACK	Emulate heavy jitter/delay in
T8	LAN	100 ms	Unlimited	TCP-AAD	wireless conditions
T9	LAN	None	10 Mbps	Default DACK	Simulate congested or legacy
T10	LAN	None	10 Mbps	TCP-AAD	wireless bandwidth
T11	LAN	None	50 Mbps	Default DACK	Evaluate throughput under
T12	LAN	None	50 Mbps	TCP-AAD	moderate bandwidth lim
T13	LAN	None	100 Mbps	Default DACK	Evaluate throughput under
T14	LAN	None	100 Mbps	TCP-AAD	high bandwidth
T15	WAN	Natural RTT	Unlimited	Default DACK	TCP performance in realistic
T16	WAN	Natural RTT	Unlimited	TCP-AAD	WAN environment

3.3.5 Small additions and logging

For better understanding packet flow and specific metrics during tests we will cover all important functions during packet flow in TCP receiving side that includes:

- tcp_rcv_established – Entry point for processing incoming data packets when the socket is in the ESTABLISHED state. We added logs here to mark packet boundaries, flow initiation, and data segment details.

- tcp_event_data_recv – Responsible for ATO calculation. Logging including inter-arrival time tracking, ATO updates, and logic branch decisions related to time-offset calculation.
- tcp_send_delayed_ack – Schedules the delayed ACK timer. We logged ATO values, timer conditions.
- tcp_send_ack – Sends the actual ACK packet. This function was logged to report when ACKs are triggered, what sequence number they acknowledge, and which flags were set.
- __tcp_ack_snd_check and tcp_ack_snd_check – Policy decision points that determine whether an ACK should be sent immediately or delayed. These were useful to observe the effect of quick ACK mode, out-of-order delivery, and PUSH flag handling.

Each log line was prefixed with a functional tag (e.g., *[ACK SCHED]*, *[DELAYED ACK]*, *[RECV PATH]*) and socket pointer, allowing post-processing scripts to correlate logs from different phases of the TCP packet receiving. The added logging was compiled only in debug mode and removed during performance measurements to prevent instrumentation overhead from affecting the results.

This level of logging enabled us to trace packet flows and confirm the intended behavior of the modified TCP-AAD implementation, particularly in how ACK timing adapts to aggregation bursts and idle recovery.

Chapter 4

Implementation

This chapter details the practical implementation of the TCP-AAD mechanism within the Linux kernel. Building upon the design principles outlined in the previous chapter, the implementation involved modifying the core TCP acknowledgment logic to replace the default delayed acknowledgment behavior with a dynamic strategy. The goal of the implementation was to ensure that acknowledgment timing better reflects the bursty nature of Wi-Fi traffic, thereby improving overall TCP performance in wireless environments.

To achieve this, several components of the Linux TCP stack were extended or rewritten, including the acknowledgment timeout calculation, the timer subsystem, and the relevant data structures within the kernel's TCP socket implementation. Additionally, a high-resolution timer was integrated to support microsecond-level precision. This chapter presents a step-by-step explanation of these modifications, their integration into the kernel build process, and the validation strategies used to ensure functional correctness.

4.1 Modification of time-offset calculation

The first step requires modification of time-offset logic calculation stored in the function tcp_event_data_recv. By analyzing logic of pseudo-code listed in 3.1 and modifying existing Linux code we are getting new representation of the core function:

Listing 4.1: TCP-AAD logic integrated in tcp_event_data_recv

```
static void tcp_event_data_recv(struct sock *sk, struct sk_buff *skb)
{
    struct inet_connection_sock *icsk = inet_csk(sk);
    struct tcp_sock *tp = tcp_sk(sk);
    u64 now = ktime_get_ns() / 1000; // current time in microseconds

    // Reset iat_min periodically to avoid stale estimations
    if (icsk->last_reset_time + 1000000ULL <= now) {
        icsk->iat_min = U64_MAX;
        icsk->last_reset_time = now;
    }

    // Measure current inter-arrival time
    unsigned long m = now - icsk->icsk_ack.lrcvtime;
    if (m > 200) {
        icsk->iat_curr = m;
        icsk->iat_min = min(m, icsk->iat_min);
    }

    // If packet is in-order, calculate adaptive ATO
    if (icsk->delayed_segs < 2) {
        icsk->icsk_ack.ato = 500000;
    } else {
        icsk->icsk_ack.ato = div_u64((icsk->iat_min * 75 + icsk->iat_curr *
   25) * 150, 10000);
        icsk->icsk_ack.ato = min(icsk->icsk_ack.ato, 500000UL);
    }
}
```

Listing 4.1 represents core features in the new calculation of time-offset. One of them is migration from the milliseconds to microseconds which essential for TCP-AAD algorithm that was discussed in the previous section.

4.2 Sending delayed ACK modification

After calculating ATO, the delayed ACK timer must be set. In the Linux kernel, this is handled by the tcp_send_delayed_ack function located in tcp_output.c. To implement the adaptive behavior proposed by TCP-AAD, a patch was applied to modify how the ACK timer is scheduled based on inter-arrival time estimation. The following listing shows changes made:

Listing 4.2: TCP-AAD high-resolution ACK timer scheduling

```
static void tcp_send_delayed_ack(struct sock *sk)
{
     struct inet_connection_sock *icsk = inet_csk(sk);
     unsigned long ato = icsk->icsk_ack.ato;                    // ATO in
    microseconds
     unsigned long timeout = ktime_get_ns() / 1000 + ato;       // Absolute
    timeout in microseconds

     icsk->icsk_ack.pending |= ICSK_ACK_SCHED | ICSK_ACK_TIMER;  // Mark
    delayed ACK as scheduled
     icsk->icsk_ack.timeout = timeout;                          // Store
    microsecond-based timeout

     hrtimer_start(&icsk->icsk_delack_timer,                    // Start high
    -resolution timer
                   timeout * 1000,                              // Convert
    microseconds to nanoseconds
```

```
                        HRTIMER_MODE_ABS_PINNED_SOFT);                  // Absolute,
    pinned, soft interrupt
}
```

Listing 4.2 illustrates the transition from the traditional jiffies-based delayed acknowledgment scheduling to a high-resolution timer mechanism, as required by the TCP-AAD algorithm. In the default Linux implementation, the acknowledgment timeout is calculated and bounded based on smoothed RTT estimates and socket behavior (e.g., ping-pong mode), and the resulting delay is applied in jiffies via sk_reset_timer().

In contrast, TCP-AAD requires precise timing, particularly when responding to aggregated traffic common in modern Wi-Fi environments. Therefore, the patched version computes the timeout in microseconds and starts the delayed ACK timer with hrtimer_start(). This allows the acknowledgment engine to operate with significantly higher resolution, aligning the delayed ACK behavior more accurately with dynamic traffic patterns.

Additionally, by removing complex heuristics like RTT bounding and mode-based max ATO adjustments, the new logic prioritizes responsiveness and simplicity. This change also ensures consistency with the microsecond-level ato value computed earlier in tcp_event_data_recv.

These modifications ensure that TCP-AAD's adaptive logic is supported at the timer layer, completing the integration of a high-resolution acknowledgment scheduling mechanism required for modern wireless scenarios.

4.3 High-resolution timers integrating

In the previous section, we concluded that high-resolution timers (hrtimers) are necessary for accurately scheduling delayed acknowledgments. In this section, we explore the complete migration from a traditional jiffies-based approach to high-precision timers within the Linux delayed acknowledgment mechanism. Firstly, the change occurring in the main field of the struct used for working with delayed acknowledgment strategy - inet_connection_sock, called icsk_delack_timer.

Listing 4.3: Changes in inet_connection_sock

```
- struct timer_list icsk_delack_timer;
+ struct hrtimer icsk_delack_timer;
```

After changing the struct of the timer it is required to change its initialization. It is performed by the function hrtimer_init with appropriate parameters:

Listing 4.4: Timer initialization changes

```
- timer_setup(&icsk->icsk_delack_timer, delack_handler, 0);
+ hrtimer_init(&icsk->icsk_delack_timer, CLOCK_MONOTONIC,
    HRTIMER_MODE_ABS_PINNED_SOFT);
    icsk->icsk_delack_timer.function = tcp_delack_hrtimer;
```

New type of initialization also requires explicitly defining timer expiration. By passing the flag HRTIMER_MODE_ABS_PINNED_SOFT we instruct the system to use callback (also specified explicitly) tcp_delack_hrtimer only when the timer will exceed some time T assigned as the argument to the function hrtimer_start[1]. Resetting the timer is performed automatically by restarting it through the start function.

New type of timer also requires callback function. Default implementation only check availability of the socket resources and after pass the work to routine

[1]Example is in listing 4.2.

that send ACK with additional checks. New callback only changed entry point while saving the same logic.

Listing 4.5: New callback for hrtimer

```
enum hrtimer_restart tcp_delack_hrtimer(struct hrtimer *timer)
{
    struct inet_connection_sock *icsk = container_of(timer, struct
   inet_connection_sock, icsk_delack_timer);
    struct sock *sk = &icsk->icsk_inet.sk;

    bh_lock_sock(sk);
    if (!sock_owned_by_user(sk)) {
        tcp_delack_timer_handler(sk);
    } else {
        __NET_INC_STATS(sock_net(sk), LINUX_MIB_DELAYEDACKLOCKED);
    }
    bh_unlock_sock(sk);

    return HRTIMER_NORESTART;
}
```

By performing all following changes we have completely migrated to high-resolution timers.

4.4 Additional changes

For the purpose of correct calculation inside the tcp_event_data_recv() we have added additional entities in inet_connection_sock structure.

Listing 4.6: New fields for working with TCP-AAD

```
    u64 iat_min;
    u64 iat_curr;
    u64 delayed_segs;
    u64 last_reset_time;
```

Chapter 5

Results and Discussion

The purpose of this chapter to consider and analyze results obtained after experiments, discuss limitations and possibility of the future work in the field.

5.1 Device communication under limits and delay

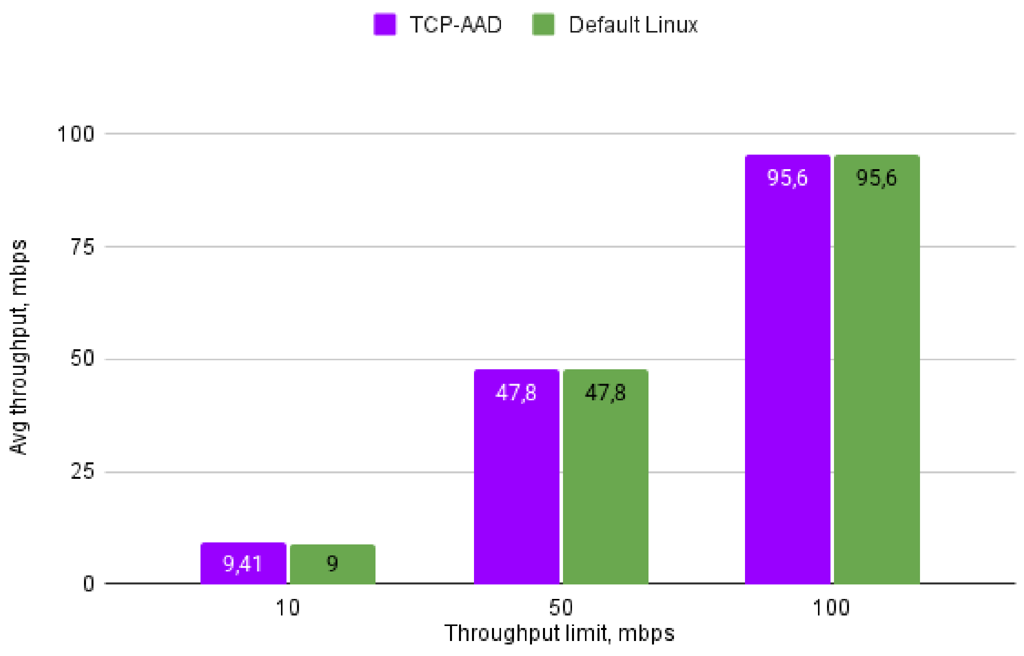

Fig. 5.1. Simulating bandwidth limit

The Fig 5.1 shows that artificial bandwidth not significantly affected on the performance of both algorithms that could be the reason of implementation peculiarities behind the bandwidth limitation.

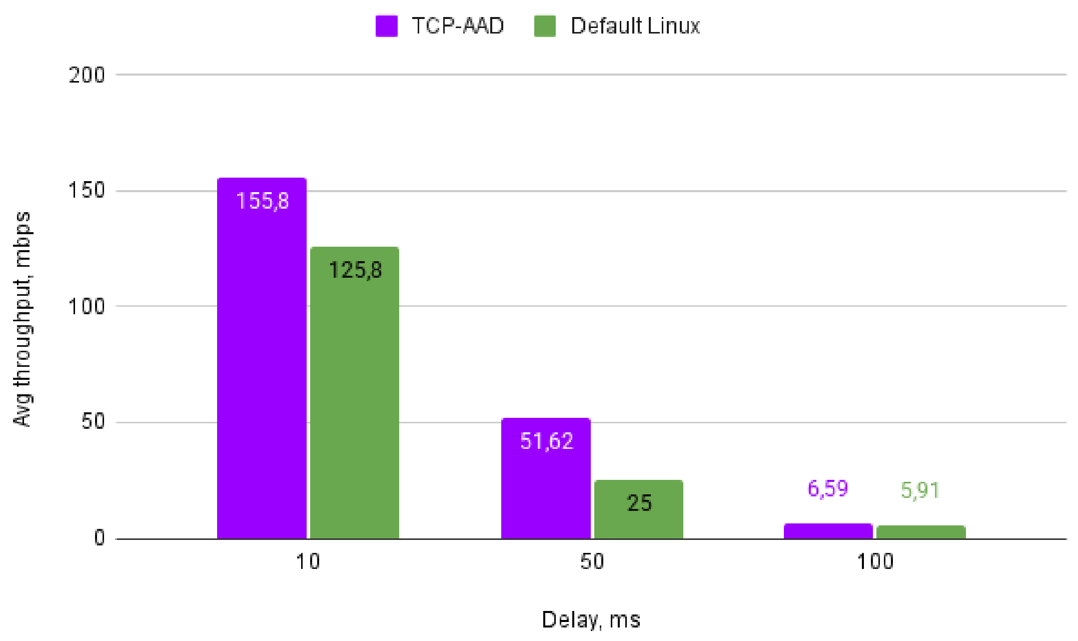

Fig. 5.2. Delays simulation

However, after applying delays to the client, we see the difference between TCP-AAD and the default DACK algorithm behavior on the receiver side. Especially, a delay of 50ms improves the performance when using TCP-AAD by approximately 106% compared to the default Linux DACK mechanism. Considerable difference in results could be the reason of different approaches used during implementation. While the default Linux kernel implementation is based on a heuristic approach with a lower bound of delay equivalent to 40ms, TCP-AAD tries to dynamically change the value based on network conditions in any situation. Due to this, TCP-AAD presents high responsiveness, which impacts CWND growth and throughput accordingly.

5.2 Device communication under different traffic flow

Next section describing throughput results with respect to different network traffic used. Although artificial limits could get abstract picture of algorithms behavior under critical conditions, it is important to consider effectiveness of the network under minimal controlled state.

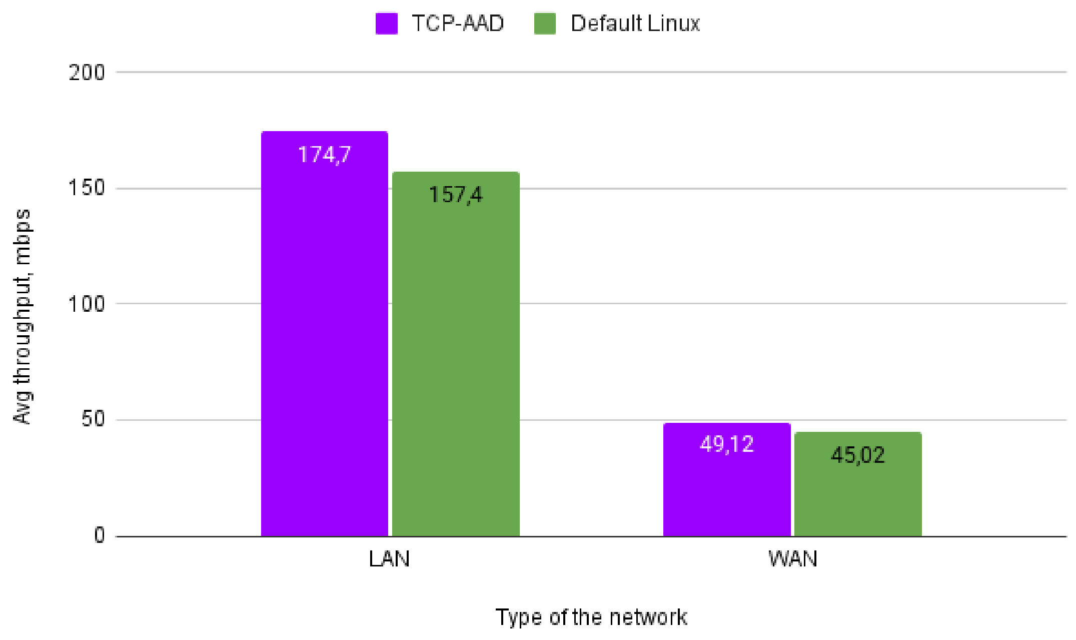

Fig. 5.3. Comparison of algorithms between different network traffic

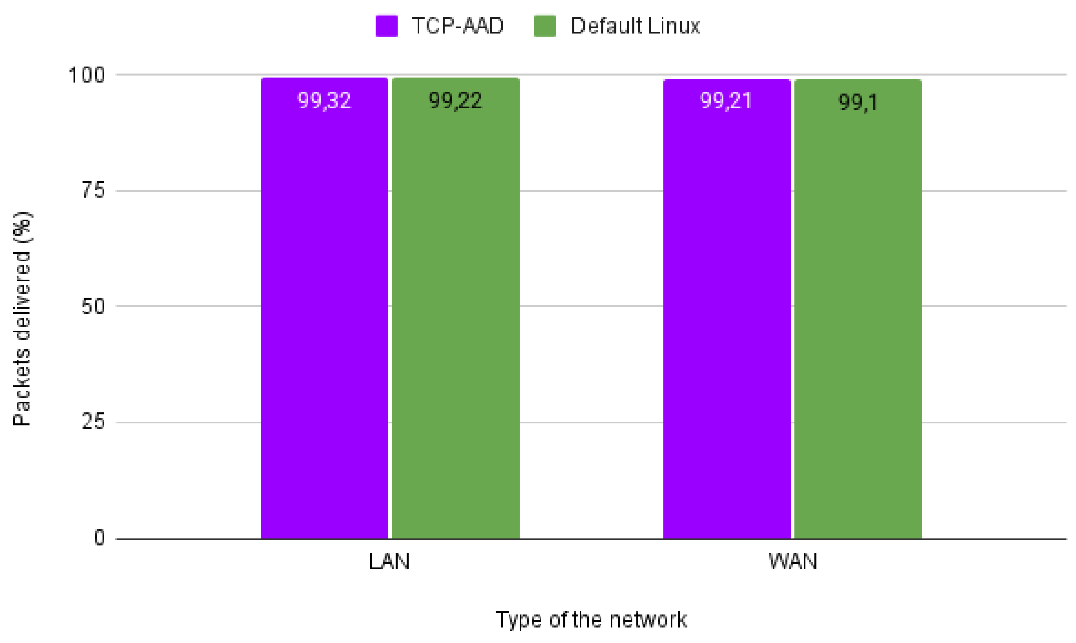

Fig. 5.4. Packet deliverance rate from the network type

Results obtained in Fig 5.3 represent approximately equivalent percentage differences between the two algorithms, with a consistent performance advantage of TCP-AAD. This trend is observable in both LAN and WAN scenarios, suggesting that the adaptive acknowledgment logic in TCP-AAD is effective across varying network environments. While the absolute throughput values differ due to variations in topology and latency, the relative improvement offered by TCP-AAD remains stable.

The Fig 5.4 represents no degradation in packet loss that confirm stable transmission by using TCP-AAD.

5.3 Comparison between different congestion control mechanisms

In order to evaluate the difference between algorithms we have also tested throughput of the algorithms across different congestion control mechanisms:

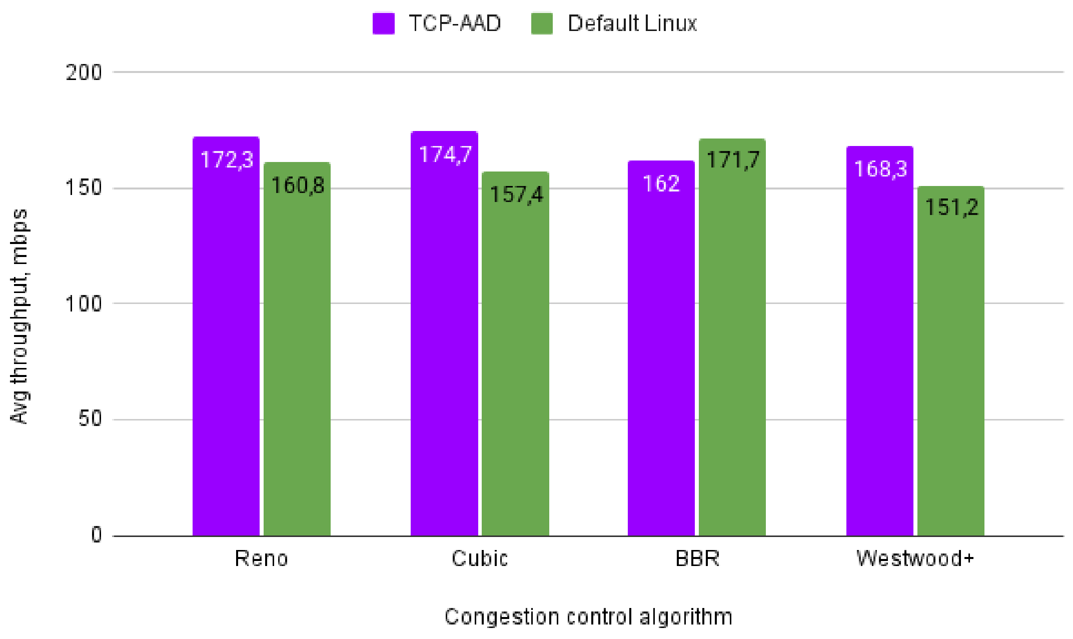

Fig. 5.5. Throughput rate from different congestion control algorithms

Figure 5.5 shows that TCP-AAD improves throughput when used with CUBIC, Reno, and Westwood+. These congestion control algorithms rely on ACK feedback to adjust their sending rate, so the faster and more adaptive ACKs from TCP-AAD help the connection grow faster and stay more stable.

However, the figure also shows a small performance degradation when using TCP-AAD with BBR. This is likely because BBR works differently—it does not rely on ACK timing but instead uses its own model to estimate bandwidth. The changes in ACK behavior from TCP-AAD may interfere with BBR's way of managing data flow.

In general, the results show that TCP-AAD works well with traditional congestion control methods, but not all algorithms benefit equally. The effect depends on how each algorithm uses ACKs to control its behavior.

5.4 Effectiveness Comparison

To evaluate the most effective algorithm, we compared throughput performance under real network conditions with CUBIC congestion control mechanism as the widely used method on modern systems. Throughput is used as the primary metric, as it reflects the data transfer efficiency. Additionally, percentage improvement is calculated to quantify the gain of TCP-AAD over the default DACK algorithm.

TABLE 5.1
Performance improvement/degradation of TCP-AAD

Test case	**Default (Mbps)**	**TCP-AAD (Mbps)**	**Improvement (%)**
LAN. No restrictions	157,4	174,4	9,75%
LAN. Limit = 10mbps	9	9,41	4,36%
LAN. Limit = 50mbps	47,8	47,8	0%
LAN. Limit = 100mbps	95,6	95,6	0%
LAN. Delay = 10ms	125,8	155,8	23,85%
LAN. Delay = 50ms	25	51,62	106,48%
LAN. Delay = 100ms	5,91	6,59	11,51%
WAN. No restrictions	45,02	49,12	9,11%

As shown in Table 5.1, TCP-AAD achieves approximately 9.75% higher throughput than the default DACK implementation in the LAN environment, and

9.11% in the WAN scenario. This improvement highlights the benefits of using an adaptive acknowledgment strategy, especially under variable wireless network conditions. Moreover, the results prove the hypothesis defined by Zakirov [1], which suggested that responsiveness to aggregation dynamics and inter-arrival time variation can significantly enhance TCP efficiency over Wi-Fi.

5.5 Real-World Implications

These results suggest that TCP-AAD is particularly well-suited for modern wireless networks, such as Wi-Fi, where variable latency, jitter, and MAC-layer aggregation are common. In such settings, TCP-AAD can offer appropriate throughput in the most congestion control mechanisms. This makes it a viable candidate for improving transport-layer.

5.6 Limitations

Despite the results, this study has limitations. The experiments were conducted in a controlled testbed using a specific hardware setup and kernel version (Linux 6.12.9). The use of iperf means only bulk TCP flows were tested; short-lived or bursty connections were not considered. Additionally, the evaluation did not measure RTT variance or fairness to competing flows, which could provide further insight into protocol behavior.

Furthermore, while TCP-AAD showed improvements with several congestion control algorithms, its performance with BBR was less effective. This may be due to the fact that BBR does not rely on ACK timing for congestion window control. Instead, it uses model-based bandwidth and RTT estimates. Thus, the interaction between ACK delay strategies and modern congestion control schemes like BBR requires further study.

5.7 Recommendations for Future Work

Future research could investigate the behavior of TCP-AAD under fluctuating signal quality. Additional evaluation metrics, such as fairness and power consumption, may provide a deeper understanding of the trade-offs introduced by integrating the algorithm into the Linux kernel. Moreover, further research is needed in the area of BBR-like congestion control mechanisms to address the limitations of algorithms that rely heavily on RTT-based behavior.

Chapter 6

Conclusion

This thesis overview the performance of TCP-AAD in comparison with the default Linux DACK implementation, specifically in the context of wireless network environments.

Through kernel-level integration and systematic testing using tools like `iperf` and `tc`, we showed that TCP-AAD achieves measurable throughput improvements in both LAN and WAN conditions, particularly under increased delay. These insights demonstrate that adaptive feedback mechanisms can significantly improve TCP responsiveness.

While the focus of this work was throughput, future efforts could include broader metrics such as power consumption, and fairness. Moreover, evaluating TCP-AAD with mobile clients could provide deeper insights into its robustness.

In conclusion, TCP-AAD offers a promising direction for improving TCP efficiency in wireless networks, combining backward compatibility with better adaptation to network behavior.

Bibliography cited

[1] D. T. Zakirov, “Optimization of tcp protocol performance over wireless links,” English, Bachelor’s thesis, Innopolis University, Innopolis, Russia, 2024.

[2] *Transmission Control Protocol*, RFC 793, Sep. 1981. DOI: 10.17487/RFC0793. [Online]. Available: https://www.rfc-editor.org/info/rfc793.

[3] K. R. Fall and W. R. Stevens, *TCP/IP Illustrated: The Protocols, Volume 1*. Addison-Wesley, 2011.

[4] “Ieee standard for wireless lan medium access control (mac) and physical layer (phy) specifications,” *IEEE Std 802.11-1997*, pp. 1–445, 1997. DOI: 10.1109/IEEESTD.1997.85951.

[5] G. Xylomenos, G. C. Polyzos, P. Mahonen, and M. Saaranen, “Tcp performance issues over wireless links,” *IEEE communications magazine*, vol. 39, no. 4, pp. 52–58, 2002.

[6] R. T. Braden, *Requirements for Internet Hosts - Communication Layers*, RFC 1122, Oct. 1989. DOI: 10.17487/RFC1122. [Online]. Available: https://www.rfc-editor.org/info/rfc1122.

[7] D. V. Paxson, M. Allman, and W. R. Stevens, *TCP Congestion Control*, RFC 2581, Apr. 1999. DOI: 10.17487/RFC2581. [Online]. Available: https://www.rfc-editor.org/info/rfc2581.

[8] E. Blanton, D. V. Paxson, and M. Allman, *TCP Congestion Control*, RFC 5681, Sep. 2009. DOI: 10.17487/RFC5681. [Online]. Available: https://www.rfc-editor.org/info/rfc5681.

[9] D. Borman, R. T. Braden, V. Jacobson, and R. Scheffenegger, *TCP Extensions for High Performance*, RFC 7323, Sep. 2014. DOI: 10.17487/RFC7323. [Online]. Available: https://www.rfc-editor.org/info/rfc7323.

[10] I. Rhee, L. Xu, S. Ha, A. Zimmermann, L. Eggert, and R. Scheffenegger, *CUBIC for Fast Long-Distance Networks*, RFC 8312, Feb. 2018. DOI: 10.17487/RFC8312. [Online]. Available: https://www.rfc-editor.org/info/rfc8312.

[11] H. Balakrishnan, V. N. Padmanabhan, S. Seshan, and R. H. Katz, "A comparison of mechanisms for improving tcp performance over wireless links," *IEEE/ACM transactions on networking*, vol. 5, no. 6, pp. 756–769, 1997.

[12] E. Altman and T. Jiménez, "Novel delayed ack techniques for improving tcp performance in multihop wireless networks," in *IFIP international conference on personal wireless communications*, Springer, 2003, pp. 237–250.

[13] J. Chen, M. Gerla, Y. Z. Lee, and M. Sanadidi, "Tcp with delayed ack for wireless networks," *Ad Hoc Networks*, vol. 6, no. 7, pp. 1098–1116, 2008.

[14] Y. Xiao, "Ieee 802.11 n: Enhancements for higher throughput in wireless lans," *IEEE Wireless Communications*, vol. 12, no. 6, pp. 82–91, 2005.

[15] A. Jaakkola, "Implementation of transmission control protocol in linux," in *Proceedings of Seminar on Network Protocols in Operating Systems*, 2013, p. 10.

[16] M. Gast, *802.11 wireless networks: the definitive guide*. " O'Reilly Media, Inc.", 2005.

[17] ESnet, *Iperf - the ultimate speed test tool for tcp, udp and sctp*, 2025. [Online]. Available: https://iperf.fr.

[18] Wireshark Foundation, *Wireshark - network protocol analyzer*, 2025. [Online]. Available: https://www.wireshark.org.

[19] J. Nagle, *Congestion control in ip/tcp internetworks*, 1984. [Online]. Available: https://www.rfc-editor.org/rfc/rfc896.

[20] M. Mathis, J. Mahdavi, S. Floyd, and A. Romanow, *Tcp selective acknowledgment options*, 1996. [Online]. Available: https://www.rfc-editor.org/rfc/rfc2018.