Performance Evaluation of Delayed Acknowledgment in TCP over Wi-Fi

Author: Avkhadeev Albert
Supervisor: Dr. Zlatanov Nikola
Consultant: Dr. Seytnazarov Shinnazar

Innopolis University

June, 2025

What is TCP?

What is TCP?

- TCP = Transmission Control Protocol.
- Operates on the **transport layer** of the protocol stack.
- Guarantees: reliable delivery, in-order data flow.
- Used by HTTP (web), FTP and many other applications that require full reliability [1].

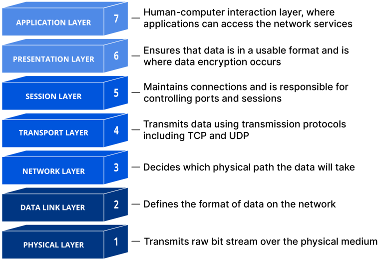

Figure 1: OSI model representation

ACK vs DACK

- Acknowledgment (ACK) - one of the main mechanisms of reliability in the protocol.
- Initial TCP implementation sent an ACK for every received packet.
- Later, delayed ACK (**DACK**) was introduced: receiver waits shortly before sending an ACK (commonly after receiving two full-sized segments or when timeout expires) [2].
- Delaying ACKs reduces overhead — fewer packets in the channel → mitigate channel overhead.
- But too much delay → throughput degradation.

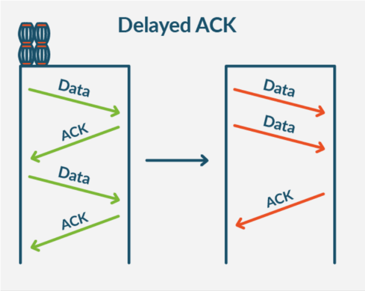

Figure 2: DACK in TCP

Motivation

Motivation

- Nowadays, IEEE 802.11 standard (also known as **Wi-Fi**) has rapidly growth tendency.
- However, Wi-Fi introduced new challenges for the TCP, including interference, delay and random packet loss, which reduced connection performance [3].
- Therefore, it is important to adjust key TCP components, such as DACK, to suit wireless conditions.

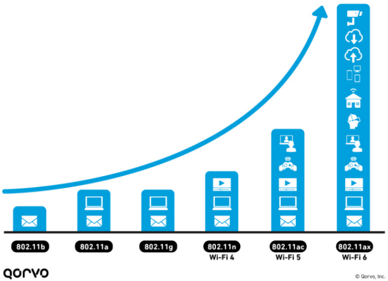

Figure 3: Trend of Wi-Fi integration

Problem Statement

Problem Statement

- Standard design of DACK works well in wired connections, but have negative influence on performance in wireless environments [4].
- One of the solution proposed is known as **a**ggregation-aware **A**CK **d**elaying algorithm (**TCP-AAD**) [5].
- However, new mechanism was tested in virtual environment like NS-3.
- Our study revealed that modern OSs such as Linux have another version of DACK that uses heuristic approach.
- It is important to compare TCP-AAD with existing mechanism in modern OS such as Linux.
- **Research Questions:**
  - How to integrate proposed solution in existing Linux Kernel?
  - Is TCP-AAD more efficient than standard Linux DACK in Wi-Fi environment?

Literature review

Brief Review of TCP Standards and Research

- RFC 1122: Introduced delayed ACKs to reduce network load [2].
- RFC 2581, RFC 5681, RFC 7323, RFC 8312: Explained congestion control mechanism (CCM) and adjustments that DACK implementation should notice during design [6]–[9].
- Altman and Jim´enez: Fixed ACK delay = bad on Wi-Fi [10].
- Chen et al and Al-Jubari et al: Development of dynamic DACK mechanism [11], [12].
- Zakirov et al: Aggregation-aware ACK delaying (TCP-AAD) - adaptive logic by considering frame aggregation of modern Wi-Fi standards [5].

TCP-AAD Algorithm

What is TCP-AAD?

- Based on property of modern Wi-Fi - **Frame Aggregation (FA)** [13]
- Uses **Inter-Arrival Time (IAT)** to detect packet bursts.

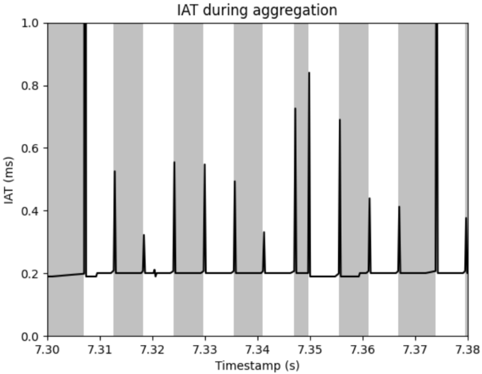

Figure 4: Behaviour of IAT for aggregate frames. Alternating background colors define start of new aggregate frame reception [5].

What is TCP-AAD?

- Based on property of modern Wi-Fi - **Frame Aggregation (FA)** [13]
- Uses **Inter-Arrival Time (IAT)** to detect packet bursts.
- Adjusts delay dynamically:

$$T = (\mathsf{IAT}_{\mathsf{min}} \cdot \alpha + \mathsf{IAT}_{\mathsf{curr}} \cdot (1 - \alpha)) \cdot \beta$$

What is TCP-AAD?

- Based on property of modern Wi-Fi - **Frame Aggregation (FA)** [13]
- Uses **Inter-Arrival Time (IAT)** to detect packet bursts.
- Adjusts delay dynamically:

$$T = (\mathrm{IAT}_{\mathrm{min}} \cdot \alpha + \mathrm{IAT}_{\mathrm{curr}} \cdot (1 - \alpha)) \cdot \beta$$

- Immediate ACK for out-of-order packets.
- Resetting $\mathrm{IAT}_{\mathrm{min}}$ each second for adapting to dynamic wireless network conditions such as mobility of the user
- Shows much superior throughput performance compared to standard and other existing schemes in literature [5]

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();
}
```

Implementation

Key concepts in Linux Kernel to consider

- Jiffies-based DACK timer (ms-level precision).
- Heuristics-based DACK timeout calculation.

Key concepts in Linux Kernel to consider

- Jiffies-based DACK timer (ms-level precision).
- Heuristics-based DACK timeout calculation.
- tcp_event_data_recv - calculating acknowledgment time-offset (ATO).
- tcp_send_delayed_ack - setting the timer.
- tcp_delack_timer_handler - callback function after timer expiration.

Key concepts in Linux Kernel to consider

- Jiffies-based DACK timer (ms-level precision).
- Heuristics-based DACK timeout calculation.
- tcp_event_data_recv - calculating acknowledgment time-offset (ATO).
- tcp_send_delayed_ack - setting the timer.
- tcp_delack_timer_handler - callback function after timer expiration.
- inet_connection_sock - main structure responsible for delayed acknowledgment.

From Idea to Kernel Code

Linux DACK

- Jiffy-based (ms-level precision) timer.
- Heuristics based ATO calculation.

TCP-AAD

- µs-resolution timer.
- Real-time IAT analysis.

What we have made

```diff
-        now = tcp_jiffies32;

-        if (!icsk->icsk_ack.ato) {
-                /* The _first_ data packet received, initialize
-                 * delayed ACK engine.
-                 */
-                tcp_incr_quickack(sk, TCP_MAX_QUICKACKS);
-                icsk->icsk_ack.ato = TCP_ATO_MIN;
         } else {
-                int m = now - icsk->icsk_ack.lrcvtime;
-
-                if (m <= TCP_ATO_MIN / 2) {
-                        /* The fastest case is the first. */
-                        icsk->icsk_ack.ato = (icsk->icsk_ack.ato >> 1) + TCP_ATO_MIN / 2;
-                } else if (m < icsk->icsk_ack.ato) {
-                        icsk->icsk_ack.ato = (icsk->icsk_ack.ato >> 1) + m;
-                        if (icsk->icsk_ack.ato > icsk->icsk_rto)
-                                icsk->icsk_ack.ato = icsk->icsk_rto;
-                } else if (m > icsk->icsk_rto) {
-                        /* Too long gap. Apparently sender failed to
-                         * restart window, so that we send ACKs quickly.
-                         */
-                        tcp_incr_quickack(sk, TCP_MAX_QUICKACKS);
-                }
         }
```

```diff
+        /* === Timestamping and ATO/IAT Logic === */
+        now = ktime_get_ns() / 1000ULL; // Convert to microseconds

+        if (icsk->icsk_ack.last_reset_time + 1000000ULL <= now) {
+                pr_info("[DATA RECV EVENT] 1 second elapsed - resetting iat_min\n");
+                icsk->icsk_ack.iat_min = U32_MAX;
+                icsk->icsk_ack.last_reset_time = now;
+        }
+
+        unsigned long m = now - icsk->icsk_ack.lrcvtime;
+        pr_info("[DATA RECV EVENT] Inter-arrival time (IAT): %lu us\n", m);
+
+        // TODO: threshold for values that are less than 0.2ms (heuristics approach calc). You can substitute by const or var.
+        if (m > 200UL) {
+                pr_info("[DATA RECV EVENT] Valid IAT - updating iat_min if smaller\n");
+                icsk->icsk_ack.iat_curr = m;
+                icsk->icsk_ack.iat_min = min(m, icsk->icsk_ack.iat_min);
+                pr_info("[DATA RECV EVENT] Updated iat_min: %lu us\n", icsk->icsk_ack.iat_min);
+        }
+
+        if (icsk->icsk_ack.delayed_segs < 2) {
+                pr_info("[DATA RECV EVENT] Few delayed segments - setting fixed ATO = 500000 us\n");
+                icsk->icsk_ack.ato = 500000UL; // TODO: could be moved to some constant as it is widely used in the program
         } else {
+                icsk->icsk_ack.ato = div_u32((icsk->icsk_ack.iat_min * 75UL + icsk->icsk_ack.iat_curr * 25UL) * 150UL, 10000UL); // simplified formula obtained by Zakirov
+                icsk->icsk_ack.ato = min(icsk->icsk_ack.ato, 500000UL);
+                pr_info("[DATA RECV EVENT] Adjusted ATO based on IATs: %lu us\n", icsk->icsk_ack.ato);
         }
+
```

Figure 4: Change in ATO calculation

What we have made

```diff
@@ -87,10 +87,10 @@ struct inet_connection_sock {
 	struct inet_bind2_bucket  *icsk_bind2_hash;
 	unsigned long             icsk_timeout;
 	struct timer_list         icsk_retransmit_timer;
-	struct timer_list         icsk_delack_timer;
+	struct hrtimer    icsk_delack_timer;
 	__u32                     icsk_rto;
 	__u32                     icsk_rto_min;
-	__u32                     icsk_delack_max;
+	__u32                    icsk_delack_max;
 	__u32                     icsk_pmtu_cookie;
 	const struct tcp_congestion_ops *icsk_ca_ops;
 	const struct inet_connection_sock_af_ops *icsk_af_ops;
@@ -113,14 +113,19 @@ struct inet_connection_sock {
 		__u8              quick;        /* Scheduled number of quick acks       */
 		__u8              pingpong;     /* The session is interactive           */
 		__u8              retry;        /* Number of attempts                   */
-		#define ATO_BITS 8
-		__u32             ato:ATO_BITS,  /* Predicted tick of soft clock         */
-		                  lrcv_flowlabel:20, /* last received ipv6 flowlabel    */
+		__u32             ato;   /* Predicted tick of soft clock         */
+	__u32    lrcv_flowlabel:28, /* last received ipv6 flowlabel      */
 		                  unused:4;
-		unsigned long     timeout;      /* Currently scheduled timeout          */
+		unsigned long long timeout;      /* Currently scheduled timeout in microsecs */
 		__u32             lrcvtime;     /* timestamp of last received data packet */
 		__u16             last_seg_size; /* Size of last incoming segment       */
 		__u16             rcv_mss;      /* MSS used for delayed ACK decisions   */
+
+		/* Fields responsible for TCP-AAD algorithm */
+		__u32 iat_min; /* Minimum in the iat between packets for last-reset-time interval */
+		__u32 iat_curr; /* Current IAT for calculation of ATO */
+		__u16 delayed_segs /* Number of delayed segments during transmission */;
+		__u64 last_reset_time /* Last time when iat_min was reset */;
 	} icsk_ack;
 	struct {
 		/* Range of MTUs to search */
```

Figure 5: Socket changes

What we have made

```diff
-/**
- * tcp_delack_timer() - The TCP delayed ACK timeout handler
- * @t:  Pointer to the timer. (gets casted to struct sock *)
- *
- * This function gets (indirectly) called when the kernel timer for a TCP packet
- * of this socket expires. Calls tcp_delack_timer_handler() to do the actual work.
- *
- * Returns: Nothing (void)
- */
-static void tcp_delack_timer(struct timer_list *t)
+static enum hrtimer_restart tcp_delack_hrtimer(struct hrtimer *timer)
 {
-	struct inet_connection_sock *icsk =
-			from_timer(icsk, t, icsk_delack_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)) {
+		pr_info("[DELAYED CALLBACK] Owned by a user, processing");
 		tcp_delack_timer_handler(sk);
 	} else {
 		__NET_INC_STATS(sock_net(sk), LINUX_MIB_DELAYEDACKLOCKED);
-		/* deleguate our work to tcp_release_cb() */
-		if (!test_and_set_bit(TCP_DELACK_TIMER_DEFERRED, &sk->sk_tsq_flags))
-			sock_hold(sk);
 	}
 	bh_unlock_sock(sk);
-	sock_put(sk);
+
+	return HRTIMER_NORESTART;
 }
```

Figure 6: Timer callback change

Experiment Setup

Experiment Setup Summary

- Controlled Wi-Fi environment (LAN + WAN tests)
- Tools: iperf2, tc (delay/bw emulation), Wireshark
- 16 scenarios: delays (10-100 ms), bandwidths (10-100 Mbps)
- Default congestion control algorithm is CUBIC

Topology for experiments used

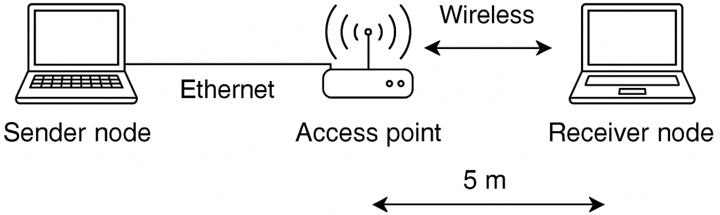

Figure 7: LAN topology

Topology for experiments used

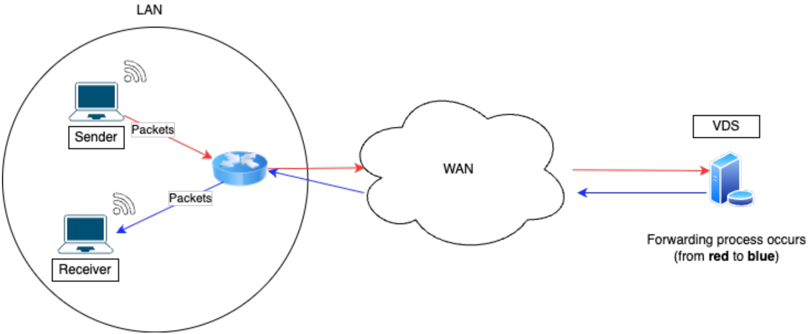

Figure 8: WAN topology

Results

LAN topology: delay in wired link

- We introduced various delay in wired link using `tc` tool
- AP's transmission rate in wireless link was fixed to a reliable one
- **Result:** TCP-AAD has superior throughput at different delays

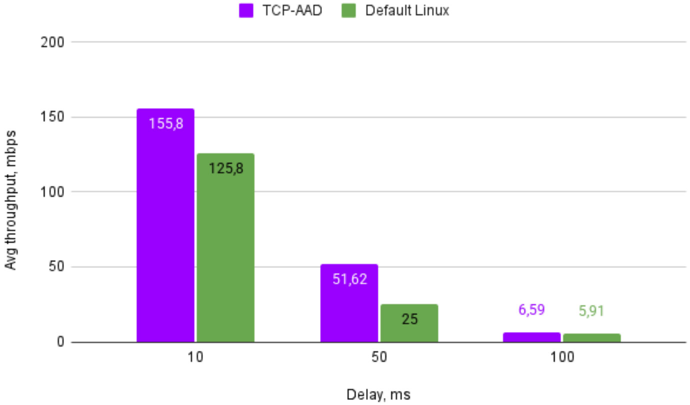

Figure 9: Throughput vs. delay in LAN topology

LAN topology: congestion control algorithms (CCAs)

- We set different CCAs at wired sender and no delay at wired link
- Other settings are the same as in the previous slide
- **Result:** All CCAs except BBR perform better with TCP-AAD

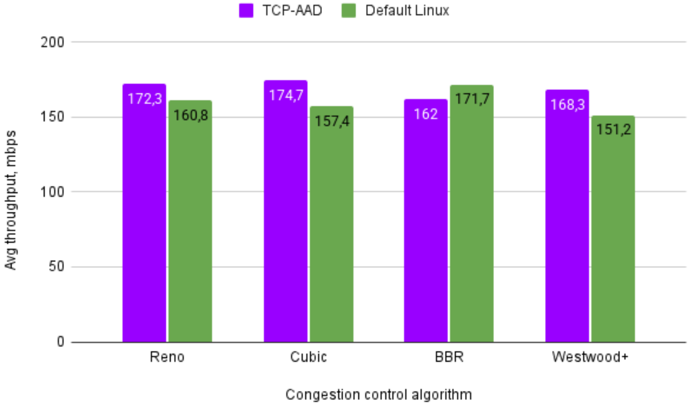

Figure 10: Throughput vs. CCAs

LAN vs. WAN

- No restriction in terms of delay
- **Result:** TCP-AAD improves throughput in both topologies

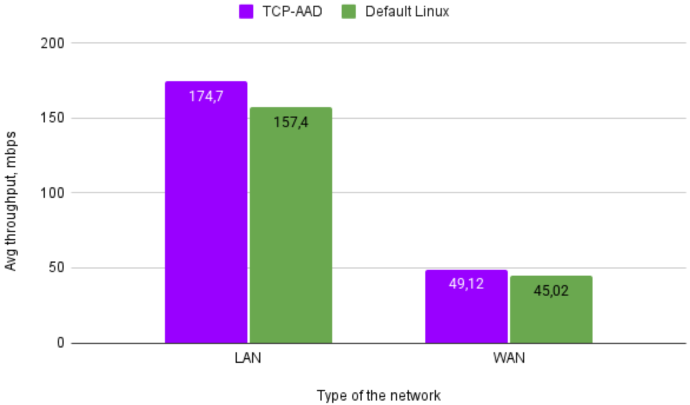

Figure 11: Throughput vs. topology

LAN topology: fixed bandwidth in wired link

- We set different bandwidth in wired link using `tc` tool
- **Result:** No significant difference when bandwidth is smaller than achievable throughput

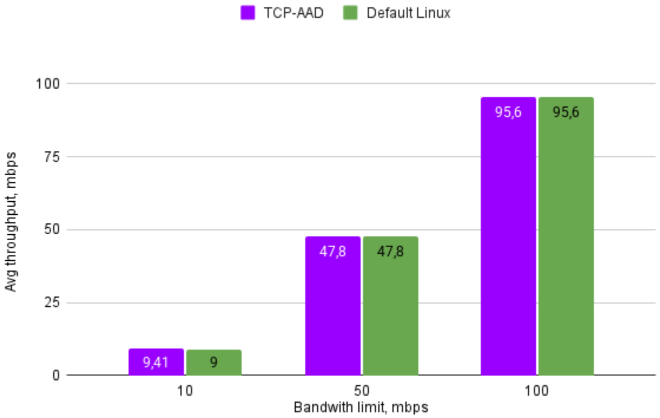

Figure 12: Throughput vs. bandwidth

Before vs After: Summary Table

Condition	**Linux DACK**	**TCP-AAD**	**Gain**
LAN (50ms delay)	25 Mbps	51.6 Mbps	+106%
LAN (10 Mbps bandwith limit)	9 Mbps	9.4 Mbps	+4.4%
LAN (No restrictions)	157.4 Mbps	174.7 Mbps	+11%
WAN (No restrictions)	45 Mbps	49.1 Mbps	+9.1%

Key Takeaways

- TCP-AAD is more responsive to traffic bursts.
- No reliability degradation.
- Boosts the throughput in both LAN and WAN topologies.
- Fully compatible with Linux kernel and tested live.
- Changes in format of Pull Request are available by the link: https://github.com/TatarinAlba/TCP-AAD-Linux/pull/2

Future Work

- Evaluate on mobile / short-lived flows
- Study fairness, CPU usage.
- Improve support for BBR congestion control
- Upstream proposal to Linux kernel (long term)

Thank You!

Questions?

References I

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

[2] 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.

[3] 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.

[4] 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.

References II

[5] D. T. Zakirov, "Optimization of TCP protocol performance over wireless links," English, Bachelor's thesis, Innopolis University, Innopolis, Russia, 2024.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] A. M. Al-Jubari, M. Othman, B. Mohd Ali, and N. A. W. Abdul Hamid, "An adaptive delayed acknowledgment strategy to improve TCP performance in multi-hop wireless networks," *Wireless Personal Communications*, vol. 69, no. 1, pp. 307–333, Mar. 2012, ISSN: 1572-834X. DOI: 10.1007/s11277-012-0575-9. [Online]. Available: http://dx.doi.org/10.1007/s11277-012-0575-9.

[13] "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.