2011-06-28

Experimental Performance Evaluation of ATP (Ad-hoc Transport Protocol) in a Wireless Mesh Network

Xingang Zhang
Brigham Young University - Provo

Experimental Performance Evaluation of ATP (Ad-hoc Transport Protocol) in a Wireless Mesh Network

Xingang Zhang

A thesis submitted to the faculty of
Brigham Young University
in partial fulfillment of the requirements for the degree of

Master of Science

Daniel M. Zappala, Chair
Kent E. Seamons
David W. Embley

Department of Computer Science

Brigham Young University

August 2011

ABSTRACT

Experimental Performance Evaluation of ATP (Ad-hoc Transport Protocol) in a Wireless Mesh Network

Xingang Zhang
Department of Computer Science, BYU
Master of Science

It is well known that TCP performs poorly in wireless mesh networks. There has been intensive research in this area, but most work uses simulation as the only evaluation method; however, it is not clear whether the performance gains seen with simulation will translate into benefits on real networks. To explore this issue, we have implemented ATP (Ad-hoc Transport Protocol), a transport protocol designed specifically for wireless ad hoc networks. We have chosen ATP because it uses a radically different design from TCP and because reported results claim significant improvement over TCP. We show how ATP must be modified in order to be implemented in existing open-source wireless drivers, and we perform a comprehensive performance evaluation on mesh testbeds under different operating conditions. Our results show that the performance of ATP is highly sensitive to protocol parameters, especially the epoch timeout value. To improve its performance we design an adaptive version that utilizes a self-adjustable feedback mechanism instead of a fixed parameter. A comprehensive measurement study demonstrates the advantages of our adaptive ATP under various operating conditions. For networks with high bit-rate, low quality links, our adaptive version of ATP demonstrates an average of more than 50% gain in goodput over the default ATP for a single flow case. With respect to fairness, the adaptive ATP generally outperforms the default ATP by an order of magnitude in most results.

Keywords: performance evaluation, ATP, wireless mesh network

ACKNOWLEDGMENTS

I could not express enough gratitude to my academic advisor: Dr. Daniel Zappala, for his consistent guidance and support during my study at BYU. I would also like to thank Lei Wang, Randy Buck, Evan Witt, and my other lab mates for their assistance with my experiments. Additionally, my sincere thanks go to my committee members for their reviews and insightful feedback. Finally, my heart goes to my dear wife and son for the joy they bring to my life.

Contents

Chapter 1

Introduction

Wireless mesh networks[1] provide an economical, yet flexible solution to the "last mile" problem of Internet connection, where the cost of laying fiber may be too expensive. However, TCP performs poorly, in terms of both throughput and fairness, in multi-hop wireless networks[2] [9, 10]. This is primarily due to the unique characteristics of wireless networks, including spatial reuse and interference constraints, which can be further exaggerated by the IEEE 802.11 MAC layer protocol that was initially intended for single-hop wireless communication[3]. Furthermore, packet loss caused by signal fading, route changes, or interference can be misinterpreted as a sign of congestion by TCP, and the subsequent rate reduction may lead to under-utilization of the wireless network.

Due to the wide deployment of TCP and its severe performance degradation in multi-hop wireless network, substantial effort has been made to improve the efficiency and fairness of the 802.11 MAC [3, 4, 11], provide better scheduling of flows [12, 25], to improve TCP performance [5, 7, 17, 27], and to create new transport protocols [16, 19, 20, 22]. Much of the earlier work in this area relies on mathematical modeling or packet-level simulation to validate the improvements made, with substantially less work validated with implementation and experiments. This can be attributed to the difficulty in modifying the network stack, which is normally built into the kernel of modern operating systems.

[1] A wireless network where nodes are mostly stationary and communicate using the ad hoc mode of the wireless driver rather than the access point mode.

[2] Any network where packets must travel over multiple wireless hops.

[3] For example, a wireless LAN, where a single access point provides an Internet connection to multiple stations.

However, despite this difficulty, experimental evaluations are indispensable since it's difficult to accurately model radio wave propagation with simulations [15], and simulation results based on simplified assumptions may differ significantly from experimental results [13, 15]. As a result, recent work has often recognized the need for implementation and experimental results [12, 13, 16, 25].

In this thesis, we add to the body of experimental work on wireless transport protocols by implementing and conducting a thorough performance evaluation of ATP (Adhoc Transport Protocol). ATP is a clean-slate design of a transport protocol for wireless networks, using rate-based congestion control and cross-layer feedback on MAC-layer packet delays [22]. We chose this protocol because relatively little work has been done to evaluate clean-slate and cross-layer designs in an experimental setting, and the original work on ATP included only packet-level simulations. Our approach differs from one previous implementation of ATP [25] in that we have implemented the entire protocol, rather than just the congestion-control algorithm. This enables us to study additional features, such as quick-start rate probing and epoch-based feedback, that have not previously been evaluated with experiments. Our work focuses on the *details* that go into developing a new transport protocol, whereas prior experimental work has typically examined overall performance or fairness [3, 12, 19].

Our implementation of ATP consists of three parts: driver-level delay averaging, per-hop delay collection, and an end-to-end user-level transport protocol. ATP's congestion control algorithm relies on measurement of the average transmission and queueing delay experienced by packets along the path used by a connection. We have modified an open-source wireless driver to collect this data. ATP's delay measurements must then be collected and inserted into packets as they are forwarded along a path. We have implemented a daemon that intercepts packets at each hop, reads the current delay measurement from the driver, and inserts this information into an ATP header. Finally, we have implemented the transport protocol itself on top of UDP using Python.

Our experimental results are obtained by running ATP in an indoor wireless mesh testbed located at BYU and comparing its performance to a TCP Tahoe implementation, also written in Python. By examining packet-level traces of ATP, we are able to identify several issues that cause it to perform poorly. First, the quick-start probe used to calculate an initial sending rate is extremely inaccurate, often resulting in an initial rate that is much too slow or much too fast. Then, two other default ATP parameters, the epoch timeout (1 second) and the rate increase factor (0.2), result in ATP adjusting its rate too slowly. Second, ATP is highly sensitive to the operating environment – the MAC layer transmission rate and link quality; this may explain why our results show worse performance than the original simulations. Making ATP more aggressive improves its performance somewhat, but it is difficult to find one set of parameters that works well in all conditions. We design an adaptive scheme for ATP that provides more prompt feedback as needed and also includes a better initial rate estimation. Our results show that these improvements to ATP allow it to provide better performance and fairness than TCP over paths of varying lengths. The advantages of our adaptive ATP are best demonstrated in the case of single flow multi-hop experiments with high bit-rate, low quality links. From our experimental results, adaptive ATP demonstrates an average of 64% goodput gain over the default ATP and almost 100% over TCP Tahoe. This performance gain is reduced when the links become saturated with more simultaneous flows. However, due to a flexi- ble and self-adaptive feedback mechanism, adaptive ATP generally performs an order of magnitude better in fairness than both default ATP and TCP Tahoe.

Chapter 2

Ad-hoc Transport Protocol (ATP)

ATP is a clean-slate design of a transport protocol for wireless ad-hoc networks, with an emphasis on support for mobility. The design can be divided into three separate functions, performed by intermediate nodes, the receiver, and the sender.

2.1 Intermediate Nodes

Intermediate nodes maintain an exponentially weighted moving average (EWMA) of the queuing delay (Q_t) and the transmission delay (T_t) for each packet:

$$\begin{aligned} Q_t &= \alpha * Q_t + (1 - \alpha) * Q_{sample}, \\ T_t &= \alpha * T_t + (1 - \alpha) * T_{sample}, \end{aligned}$$

where $\alpha = 0.75$. Each data packet carries the maximum delay it has encountered so far, D_{max} in an ATP header. A node calculates $D = Q_t + T_t$ and compares it to D_{max}, replacing the packet's value if $D > D_{max}$. Whenever the node observes an idle channel, then $D = \eta * (Q_T + T_t)$, where $\eta = 3$. This multiplier may increase to 5 for a path length of 5 or more hops [22].

2.2 ATP Receiver

For every received packet belonging to a flow, the receiver maintains an EWMA of the received D_{max} along the path:

$$D_{avg} = \beta * D_{avg} + (1 - \beta) * D_{max} \tag{2.1}$$

where $\beta = 0.85$. Whenever an epoch timer expires ($E = 1\,s$), the receiver sends a feedback packet to the sender with this D_{avg} and up to 20 SACK blocks, which helps the sender know which packets are missing.

2.3 ATP Sender

ATP senders implement a rate-based congestion control algorithm that operates in three phases: increase, decrease, and maintain. If the feedback rate ($1/D_{avg}$) from the receiver is (ϕ) times greater than the current rate ($\phi = 1.1$), the sender increases its sending rate linearly by $1/5$ ($\kappa = 0.2$) times the difference between the current rate and the feedback rate [22]. If the feedback rate is smaller than the current rate, the sender immediately lowers it sending rate to the feedback rate. Otherwise, the sender simply maintains its current sending rate.

To determine the initial sending rate, the ATP sender performs *quick-start* by sending a short probe packet along its path. The receiver returns the probe immediately, without any averaging applied to the maximum measured delay along the path. The ATP sender repeats quick-start whenever it misses three consecutive feedback packets from the receiver, effectively restarting the congestion control algorithm.

For clarity, we list the relevant ATP parameters, their meaning, and default values in Table 2.1. The default values are directly from the ATP paper [22] and are claimed to provide optimal performance from exhaustive simulations.

Parameter	Meaning	Default Value
E	epoch timeout	1 s
α	intermediate nodes averaging	0.75
β	receiver averaging	0.85
η	idle multiplier	3
ϕ	rate increase threshold	1.1
κ	rate increase factor	0.2

Table 2.1: List of Important ATP Parameters

The only part of ATP that we do not implement is link failure notification from intermediate nodes, similar to ELFN [27]. We omit this because we use static routing for our experiments, without testing any link failure scenarios.

Chapter 3

Implementation

The architecture of our ATP implementation consists of three major components: 1) *delay averaging*, 2) *delay collection*, and 3) *transport protocol*. Figure 3.1 illustrates how these components interact. Delay averaging is done by modifying the open source `ath5k` driver [1]. Delay collection is performed by writing an application for the WiFu toolkit, which is software that has recently been developed at BYU. The transport protocol, shown as ATP in the figure, is implemented on top of UDP in Python.

3.1 Delay Averaging

ATP requires each intermediate node to measure the real-time queuing and transmission delay for every packet transmitted and to maintain a moving average of each of these values. However, these measurements are not readily available in the `ath5k` driver we use in our experiments, because packets are dequeued and transmitted in hardware. This means that even if we can tell exactly when a packet is put into the transmit queue, it's

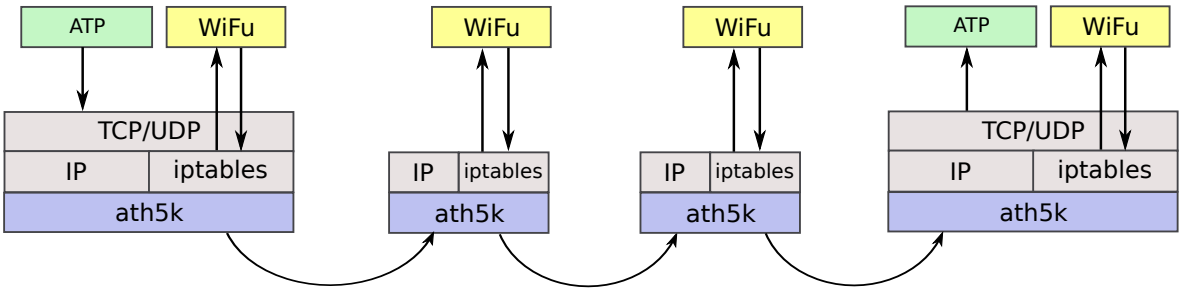


Figure 3.1: ATP Implementation Architecture

not possible for us to know when it's dequeued and the actual transmission starts. Thus to approximate ATP's delay measurements, we modified the `ath5k` driver to measure the sum of ($Q_{sample} + T_{sample}$). This is the total time from when a packet enters the transmit queue until an interrupt is received, signalling the packet was successfully transmitted. The driver maintains a moving average of this sum:

$$D = Q_t + T_t = \alpha * (Q_t + T_t) + (1 - \alpha) * (Q_{sample} + T_{sample}) \tag{3.1}$$

Mathematically, this is equivalent to keeping two separate averages of the delays and then summing them. The driver writes D to the */procfs* file system every time it changes.

One complication we encountered involves interrupt handling for transmitted packets in the `ath5k` driver. In the driver initialization phase, an interrupt mask is configured to determine which interrupts will be handled by the driver. Although this can be enabled to generate an interrupt for every single successful packet transmission (`TXOK`), this may cause a performance degradation due to high interrupt load. Thus, by default the driver is configured to only enable two interrupts: one for the end-of-line (`AR5K_INT_TXEOL`), which indicates this data frame is the last one in the transmit queue; and one for the transmit descriptor (`AR5k_INT_TXDESC`), which indicates that a group of frames were transmitted from the transmit queue. This default interrupt handling configuration will create an inaccuracy for our measurement of the total delay for each packet under heavy load. When the wireless network card is not busy transmitting data frame, the EOL interrupt can be used as a good approximation for transmission finish time for a packet because it may be the only packet in the transmit queue. However, if the wireless card is busy sending data frames, the descriptor interrupt only reflects the last transmitted packet's finish time, and no interrupt is sent for the other packets in the batch. Therefore, the trade-off between a more accurate measurement and better performance under load is inevitable. We choose the latter since this is the default configuration for `ath5k` driver and we want to evalu-

ate ATP performance in a more realistic environment. We will discuss the impact of this trade-off in our experimental results.

3.2 Delay Collection

We implement delay collection using the WiFu toolkit, which allows user-space applications to intercept and process IP packets as they are forwarded by the kernel. With WiFu, the application specifies a set of `iptables` rules to indicate which packets to intercept, and then reads these packets using the `netfilter` interface. Thus our delay collection daemon reads packets from the `netfilter` queue, and then compares D_{max} from the packet with the current delay measurement stored in */procfs*. If the local delay is larger, then the daemon replaces D_{max} with this node's current delay measurement. All packets are then sent back down to the kernel to continue the forwarding process.

3.3 Transport Protocol

We implement ATP's transport protocol functionality using Python, running on top of UDP. This includes connection establishment and teardown, reliability, and congestion control. During connection establishment, ATP uses quick-start to find the initial rate, and it begins transmitting packets at this speed. The TCP receiver averages delay measurements and sends a periodic ACK to the sender, once per epoch. After each received ACK, the sender will calculate a new rate and may choose to retransmit some missing packets.

Our use of Python has some drawbacks. It is well established that interpreted languages are usually slower than compiled languages. When building a transport protocol in Python, this means that fine-grained timers may be less accurate than in a C implementation, and that per-packet network I/O may be slower. In our experiments, these drawbacks become more obvious with multiple flows when multiple senders reside on

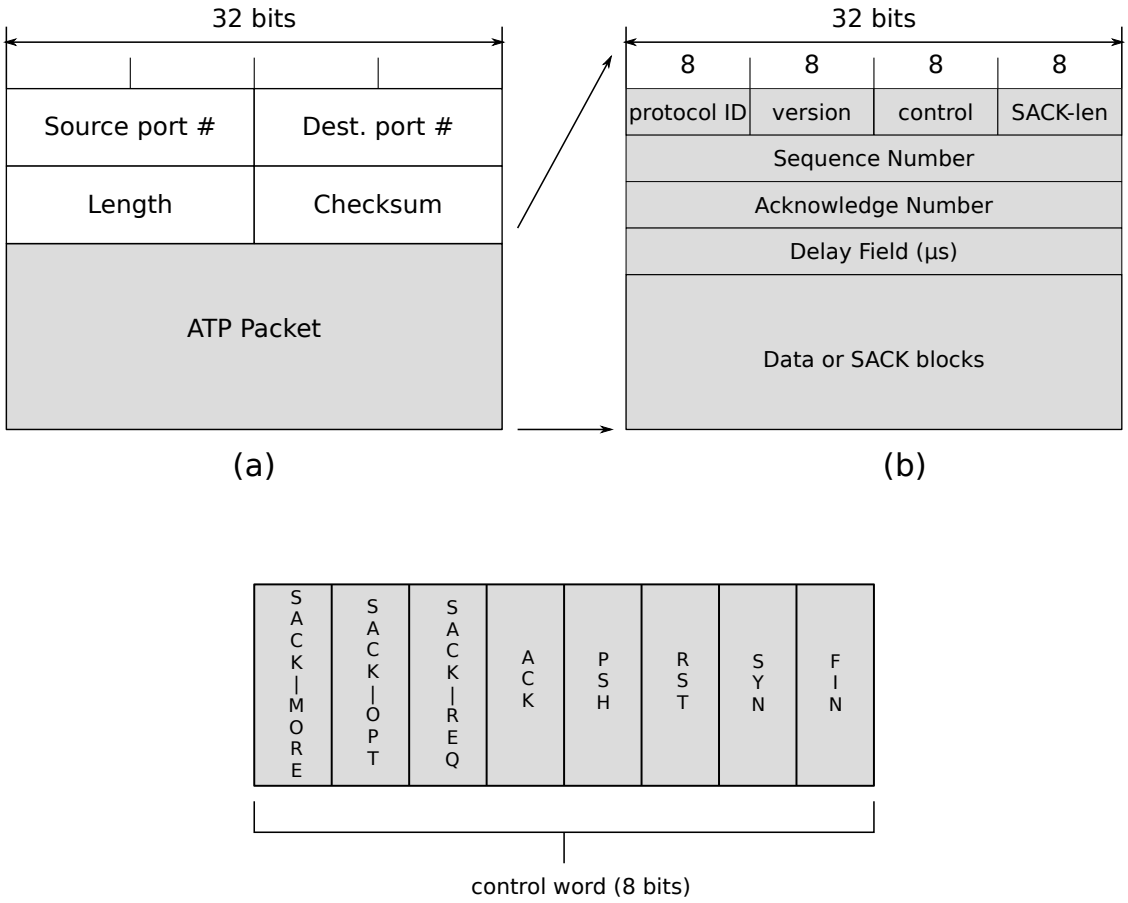


Figure 3.2: ATP Header Format

the same nodes and compete for the system resources. However, for most cases our code is efficient, and we are able to make comparisons between ATP in Python and TCP in Python. We are currently rewriting our code in C to make it more efficient, so that we can compare ATP to a kernel implementation of TCP.

Because previous ATP publications do not include a header format [21, 22], we design our own ATP header, which is shown in Figure 3.2. Figure 3.2(a) shows a UDP datagram with an ATP packet as the payload. Figure 3.2(b) shows the ATP header and body, including:

- *protocol ID*: a number specifying ATP or TCP,
- *version*: the ATP version number,

- *control*: control flags,
- *SACK len*: total length in bytes of the SACK[1] blocks,
- *sequence number*: sequence number in bytes, as with TCP,
- *acknowledgement number*: ACK number in bytes, as with TCP, and
- *delay*: maximum delay (D_{max}) seen by intermediate nodes along the path the packet traverses, and
- *data/SACK*: contains data for a packet going forward along a path, contains SACK blocks for an ACK.

Note that for simplicity ACK packets do not carry data.

Figure 3.2(c) shows the structure of control flags for ATP packet. Most of them have the same meaning as that of TCP except the first three: SACK-MORE, SACK-OPT, and SACK-REQ. Our implementation of ATP relies on SACK blocks to trigger retransmission, which means that the SACK functions essentially like a negative acknowledgement. When the sender is bursty or otherwise finishes sending a burst of data, the receiver cannot detect dropped packets from the end of the burst. Thus the receiver sets the SACK-MORE flag whenever it has not received any data for half of the epoch timeout period. When receiving an ACK with this flag set, the sender will retransmit anything that has not yet been acknowledged. The SACK-OPT flag is set by the receiver to indicate that the packet contains a SACK block rather than data. The SACK-REQ flag is set by the sender during quick-start to ask the receiver to immediately send a feedback packet, rather than waiting for an epoch to expire.

The reliability portion of ATP will resend packets that appear to be missing, based on the received SACK blocks in a feedback packet. Retransmitted packets always have priority over new packets. The sender expects a feedback packet once every $1.1 * E$ sec-

[1]Selective Ack, which only acknowledges missing packets.

onds; the additional 10% of waiting time is used to allow time for the feedback to travel from the receiver to the sender.

Chapter 4

Experiment Design and Setup

Experimental evaluation has several drawbacks. First, it can be hard to generalize experiments from one case to another, because the operating conditions (transmit power, placement of nodes, link quality, etc.) may be different in each deployment. Second, it can be difficult to achieve repeatable results, because there are so many factors that can affect performance. We have been careful to design our experiments to avoid these pitfalls as much as possible, by varying the operating conditions and by limiting variability from run to run as much as we can.

4.1 Experiment Design

We made an initial assessment of ATP performance using a set of simple experiments and found that the following variables influence performance and repeatability:

- *environmental variation*: Since our mesh nodes are distributed in TMCB, there are significant differences in human activities and wireless connectivity between different periods of day. To avoid this fluctuation in interference from other wireless sources, we use IEEE 802.11a rather than 802.11b, since there is no other traffic using this frequency in our building. In addition, we alternate the test sequence between different protocols or parameters in a random order to ensure that differences seen between versions are not due to an abrupt environment change.

- *MAC rate control*: By default, the `ath5k` driver uses a rate control algorithm called `minstrel` [2] to find the best transmit rate given current conditions. This affects repeatability when different experiments see different MAC rates. To eliminate this possibility, we turn off minstrel and use a fixed transmit rate for our experiments.
- *routing protocol*: We initially used OLSR [24] to compute routes for our experiments, but found that it would withdraw routes under heavy congestion, causing periods of time without any route between neighboring nodes. As a result, for most experiments we use static routing to ensure the routes are consistent across experiments.

4.2 Experiment Setup

The primary factor affecting transport protocol performance in a single radio wireless mesh network is path length. For this reason, our experiments primarily use a single path of varying length to evaluate ATP performance. Figure 4.1 illustrates the portion of our mesh testbed located on the second floor of our building, with a 6-hop chain from mesh9 $\rightarrow$ mesh6. To provide greater generality, we use a second 6-hop chain from mesh18 $\rightarrow$ mesh28 on the first floor of our building, as shown in Figure 4.2. For brevity, we name these configurations **Testbed A** and **Testbed B**. In our experiments, we activate only the nodes in the chain and turn the rest to a different frequency.

Each mesh node in these two configurations is a Dell desktop running Ubuntu Linux (kernel 2.6.32) with a 3Com 3CRDAG675B wireless card (Atheros AR5413 chip) that supports IEEE 802.11a/b/g. Each wireless card is loaded with our modified `ath5k` driver. In all our experiments, RTS/CTS is disabled and the MAC retransmission retry is set to *zero*. We configure each mesh node to operate on the 802.11a band to minimize uncontrolled wireless interference. In all our experiments, we transfer a 2MB text file from the sender to the receiver and record all packet events at both ends for analysis. We repeat each experiment at least 10 times to obtain a sufficient sample. Finally, we

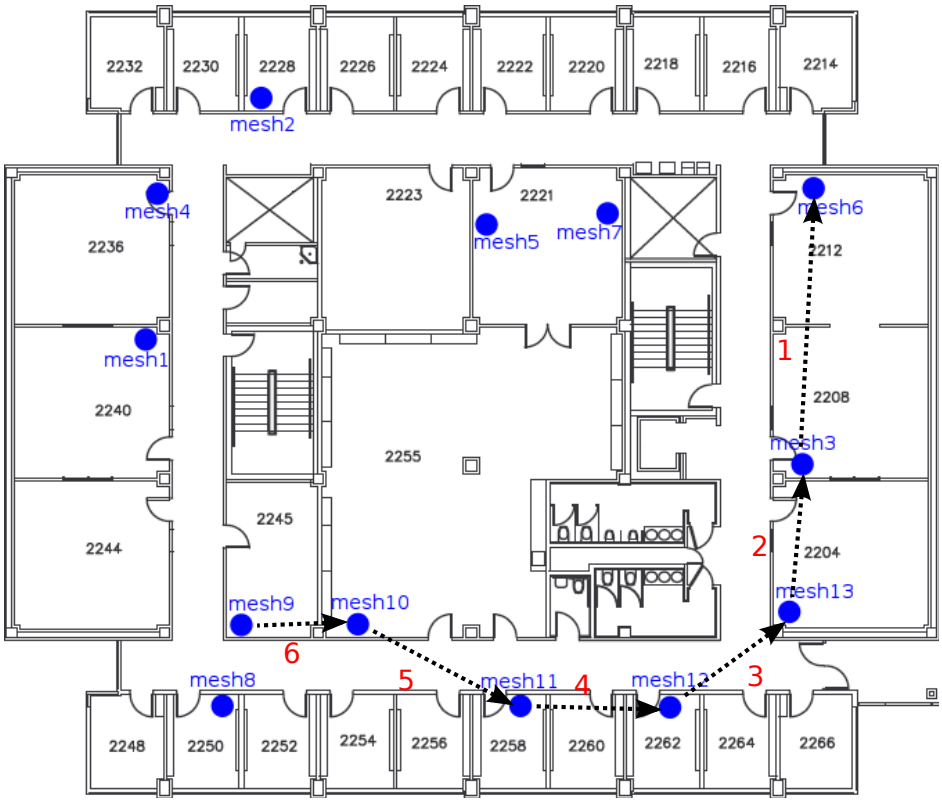


Figure 4.1: 2nd Floor Mesh Nodes and Path (Testbed A)

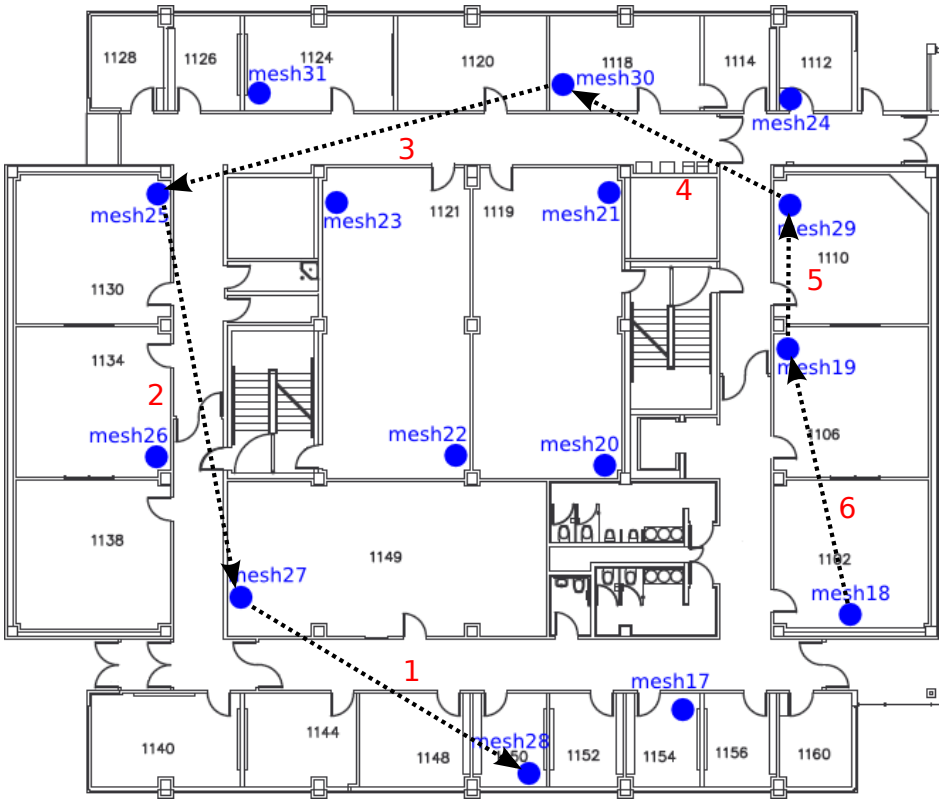


Figure 4.2: 1st Floor Mesh Nodes and Path (Testbed B)

explicitly change the combination of transmit power and transmit bitrate on the wireless card to emulate different link quality and operating conditions. Due to space limitations, we only present some of our results in the following sections.

Chapter 5

Experimental Results and Analysis

We compare the performance of ATP to our implementation of TCP Tahoe, which we also wrote in Python to provide a fair comparison. We first examine the performance of ATP regulating a single flow of traffic on Testbed A, so that we can examine the details of its performance. Using this experiment, we develop aggressive and adaptive versions of ATP, to overcome some of its shortcomings. We then examine the performance of these versions with multiple flows, and generalize our results to also consider Testbed B. We finish by examining the fairness of ATP in a common stack topology and the performance of ATP in randomized flows experiments.

In all of our results, the error bars represent the standard deviation of the measure. Also, we use goodput, which only counts the useful information bits received, instead of throughput which counts every bit received.

5.1 Single Flow

For our single flow experiments, we vary the length of the path over which ATP and TCP operate, from one to six hops, in Testbed A. We configure the nodes to use a bit rate of $24\ Mbps$ and a power of $10\ dBm$.

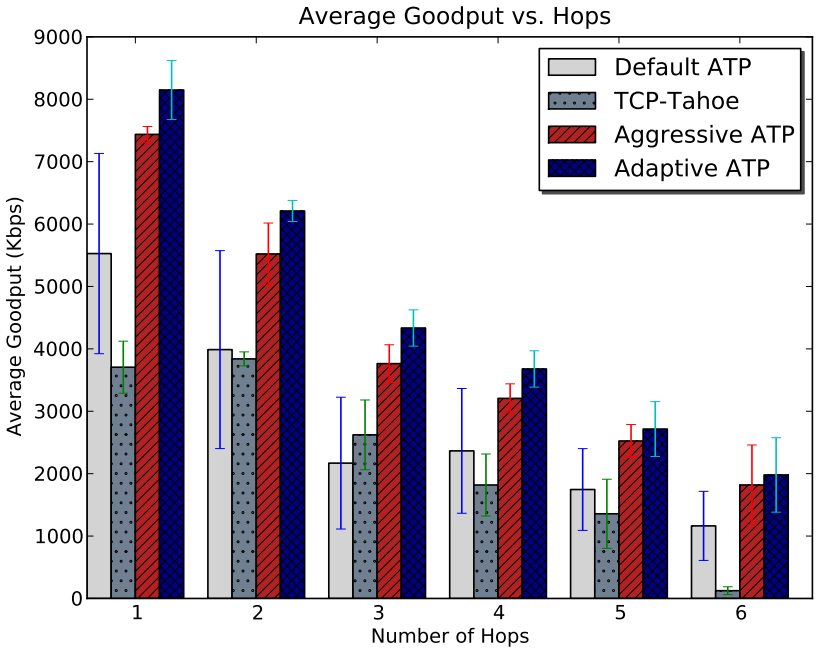


Figure 5.1: Goodput, ATP and our variants, single flow, Testbed A

5.1.1 Default ATP

We begin by running ATP with its default parameters, which are listed in Table 2.1. These are the parameters used by the simulations run by the designers of ATP [21, 22]. Figure 5.1 shows the average goodput versus path length. Generally, default ATP obtains better goodput than TCP, with significant gains for a single hop and for a six-hop path. ATP also demonstrates more variance in experiments than TCP Tahoe.

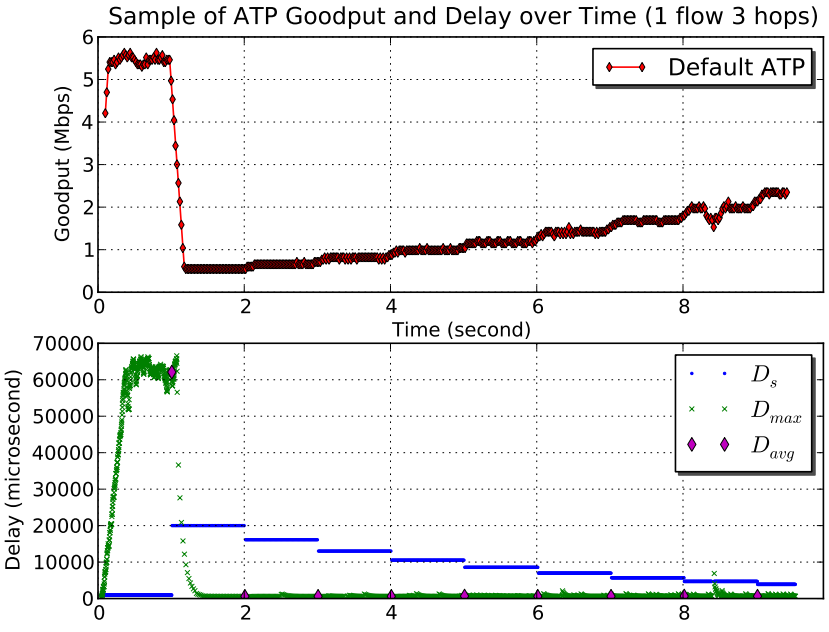
To investigate ATP performance in more detail, we plot the goodput and delay evolution over time from one sample 3-hop experiment in Figure 5.2(a). The upper half of the figure shows the instantaneous goodput of the connection over a 200 ms sliding time frame. The lower half shows a delay trace, with green points representing the D_{max} field of every incoming packet at the receiver, blue points marking the sending delay D_s chosen by the sender, and pink points marking the periodic feedback of D_{avg} to the sender every epoch. At the beginning of this trace, the sender transmits with a rate $(1/D_s)$ that is too fast, which congests the network and causes D_{max} to increase by more than 60 ms.

Because the receiver only sends feedback at the expiration of the epoch (1 second), the sender won't reduce its rate for a long period. When the sender receives the feedback from receiver, it overreacts by reducing to a sending rate of less than 1 Mbps, and then slowly increases its rate (represented by a steadily decreasing D_s) over multiple epochs due to a small rate increase factor (κ). Figure 5.2(b) shows a different trace where the initial sending rate is too slow and it takes multiple epochs to adjust the sending rate to take advantage of the link speed.

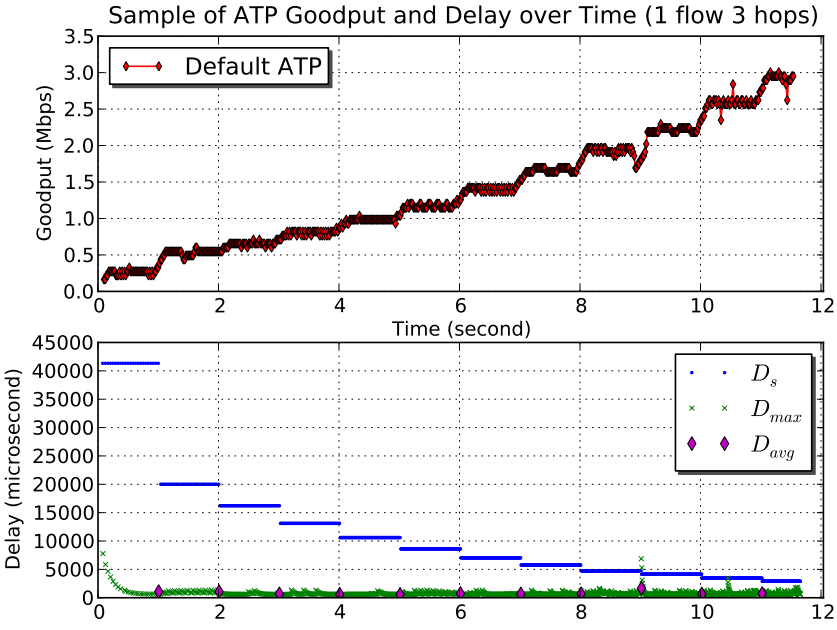
From the above analysis, we can see there are several obvious faults with the recommended ATP implementation: (a) quick-start may not obtain an appropriate sending rate with only one probe, (b) the epoch timeout of 1 second is too large for a high bitrate network, and (c) the rate increase factor κ of 0.2 may be too slow, particularly when combined with a long epoch period.

5.1.2 Aggressive ATP

To correct these problems, we explore using a more aggressive set of parameters for ATP by reducing the epoch timeout E to 0.04 second and increasing the rate increase factor κ to 0.5. Referring again to Figure 5.1, we can see that with more prompt feedback, aggressive ATP performs better than default ATP by an average of 47%. The lower half of Figure 5.3(a) shows how a faster rate adaption helps ATP adjust to a poor choice of an initial rate. However, we also observe that the instantaneous goodput is highly variable with these new settings. Furthermore, the more aggressive feedback does not necessarily mean more *prompt* feedback, which is best illustrated in Figure 5.3(b). This figure zooms in on the delay trace and adds an additional set of light blue points to indicate when the feedback from the receiver is actually received by the sender, denoted $D_{avg'}$. There is a significant delay for some of these feedback packets, indicating that they must contend for the channel on the reverse path; note a cluster of feedback packets received at 0.15

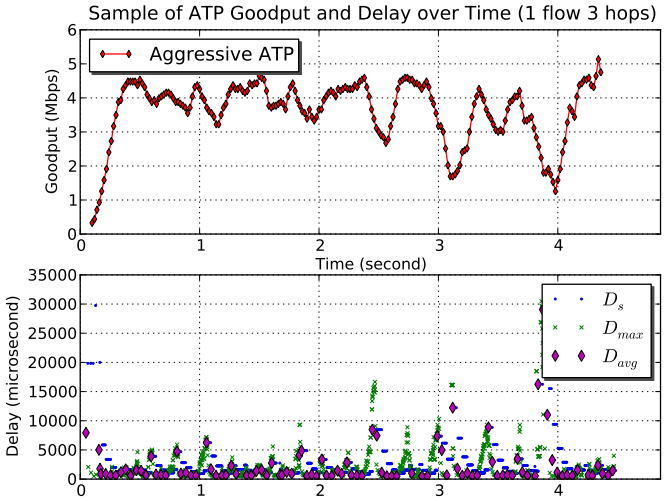


(a) Initial rate too fast

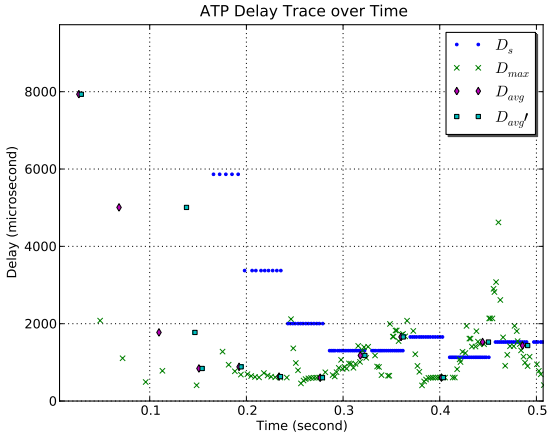


(b) Initial rate too slow

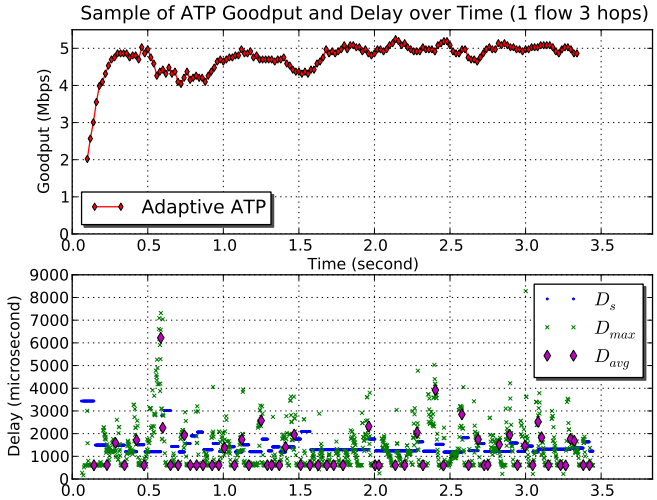
Figure 5.2: Goodput and delay traces, default ATP, single flow, Testbed A



(a) Aggressive ATP delay trace



(b) Aggressive ATP Zoomed delay trace



(c) Adaptive ATP delay trace

Figure 5.3: Goodput and delay traces, aggressive and adaptive ATP, single flow, testbed A

seconds in the figure. This clustering occurs because we do not use a priority queue for feedback packets.

We experimented with many different settings for E and κ, and the results shown here are the best we obtained on Testbed A. It is not guaranteed that these parameters will be best in all deployments and along all paths. The designers of ATP realized the potential problems with a fixed epoch timeout of 1 second and mention that it needs to be adapted for different environments [22]. However, no solution has been provided on how to adapt the epoch to fit different wireless paths.

5.1.3 Adaptive ATP

To find a better solution for this problem, we introduce an adaptive SACK feedback mechanism that can adjust the feedback delivery time according to different operating situations. Our adaptive version of ATP uses the same settings for E and κ as the aggressive ATP, but augments the receiver to send quicker feedback. For every outgoing packet from the sender, ATP inserts the current sending delay D_s in the ATP header as depicted in Figure 3.2. It then uses the following rules to send a feedback packet immediately, rather than waiting for an epoch:

- if the receiver detects two dropped packets,
- if there are 10 *consecutive* incoming packets with $D_{max} > D_s$ (this indicates the current sending rate is too fast and is causing some contention in network), and
- if there are 10 *consecutive* incoming packets with $D_{max} < (1 - \phi) * D_s$ (the sender is going too slowly).

We choose a threshold of 10 consecutive packets to avoid any temporal turbulence in the network that would induce false alarms and oscillation.

In addition, we add a number of hops field N_h to the ATP header. This field simply records the number of hops traversed from sender to receiver, similar to the IP TTL field.

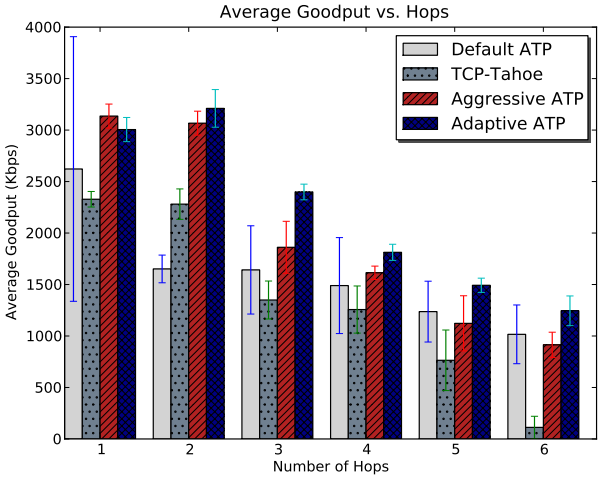
The sender can use this total number of hops to better adjust its idle multiplier (η) at the initiation stage of connection. We use $\eta = 3.0 + 0.5 * (h - 1)$, where h is the number of hops. The data collection portion of ATP updates this field as the packet is forwarded.

Referring again to Figure 5.1, we can see that in this same single flow experiment, adaptive ATP performs 64% better than default ATP. In addition, Figure 5.3(c) clearly shows a more smooth rate curve for adaptive ATP, which confirms our algorithm's resilience to small turbulence.

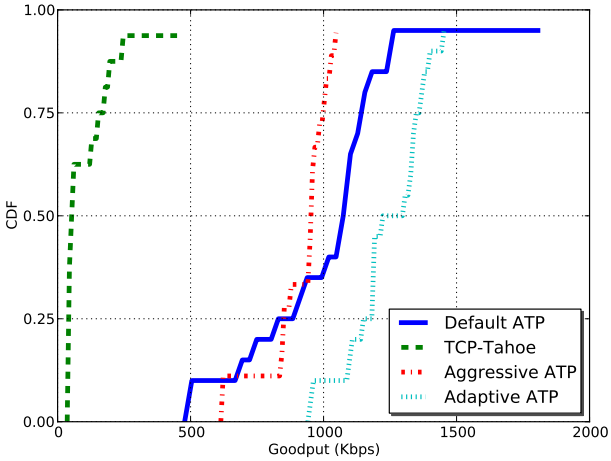
5.2 Multiple Flows

Fairness is another important evaluation criteria for ATP and our variants. We perform experiments with two and five simultaneous flows between the same sender and receiver pair in Testbed A. Figure 5.4 shows the results for two flows. Figure 5.4(a) shows that adaptive ATP generally achieves the best average goodput among the variants and TCP. Second, the performance advantage obtained with aggressive ATP over default ATP diminishes with increasing path length. Figure 5.4 (b) shows a CDF (Cumulative Distribution Function) of goodput for 6 hops, verifying the performance advantage of ATP over TCP Tahoe in a multihop mesh network. Finally, Figure 5.4(c) uses the same normalized standard deviation of goodput, which is introduced in [22] as an *unfairness index* to calibrate the fairness between the two flows. Smaller values indicate better fairness, with our modified versions of ATP performing much better than default ATP and TCP.

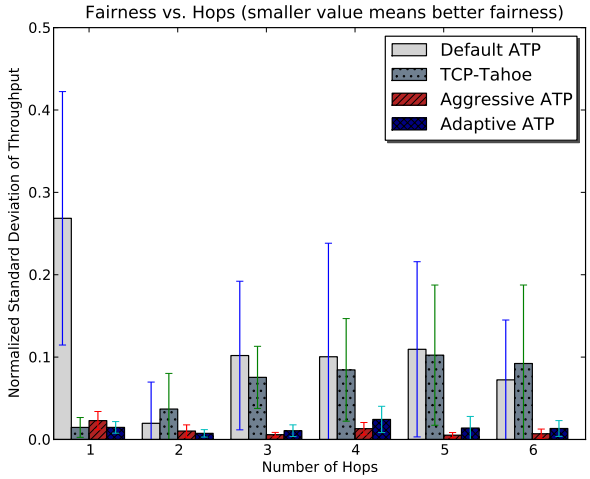
Figure 5.5 shows the results for five flows. Default ATP performs surprisingly well in this experiment for flows longer than one hop, and aggressive ATP fares poorly. This makes sense, because conservative rate adjustment should work better when the network is more congested. However, our adaptive version of ATP demonstrates remarkable consistency, and generally performs on a par with default ATP, with better fairness.



(a) Average Goodput vs. Hops

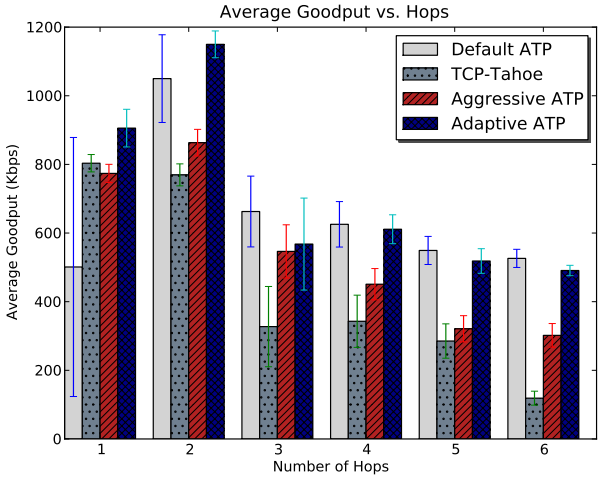


(b) CDF of Average Goodput for 6 hops

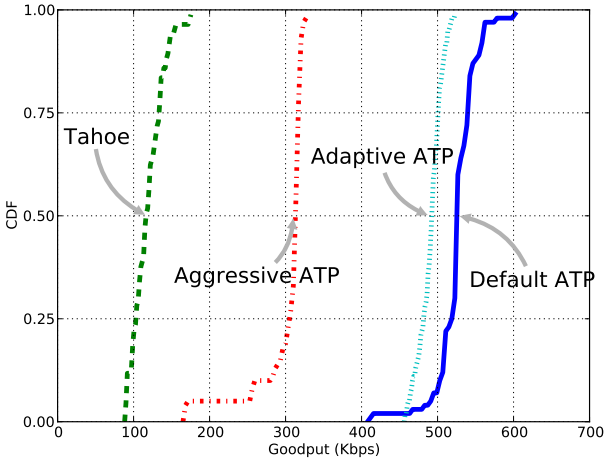


(c) Fairness

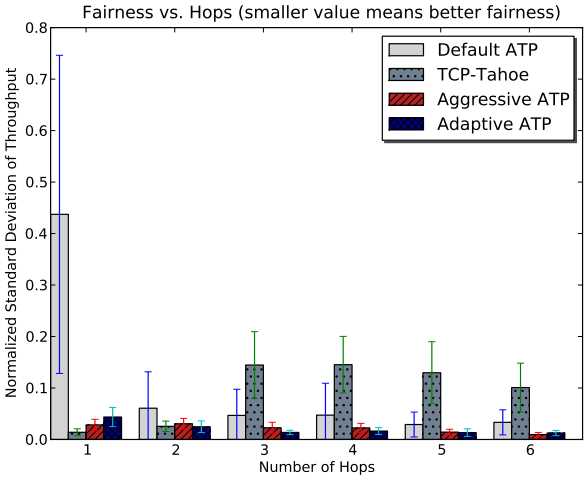
Figure 5.4: Two Flows, ATP and our variants, Testbed A



(a) Average Goodput vs. Hops

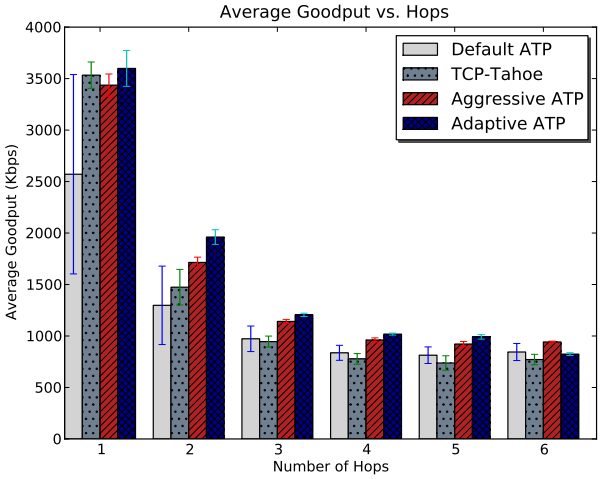


(b) CDF of Average Goodput for 6 hops

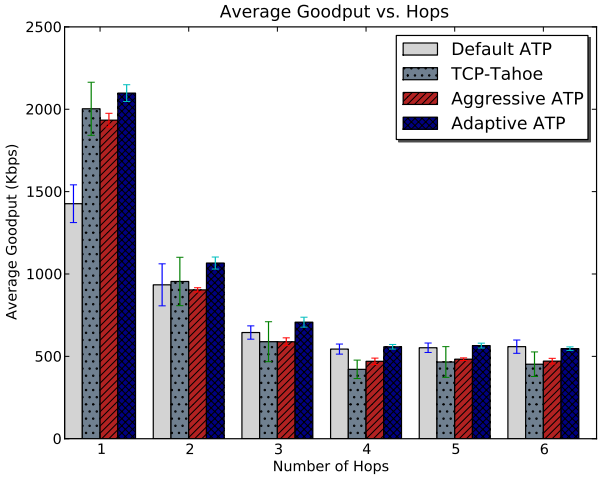


(c) Fairness

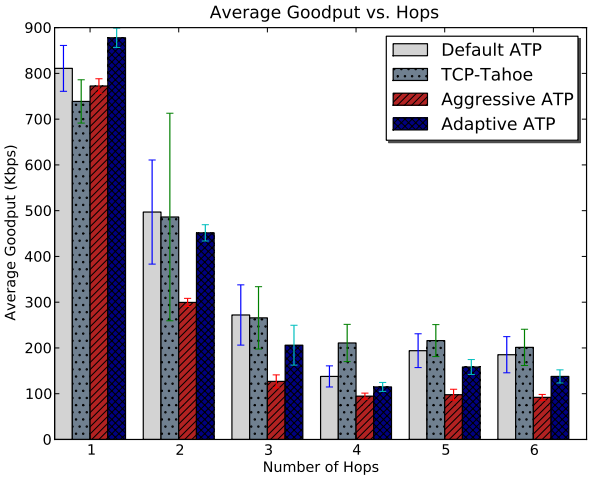
Figure 5.5: Five Flows, ATP and our variants, Testbed A



(a) Single Flow, Multihops



(b) Two Flows, Multihops



(c) Five Flows, Multihops

Figure 5.6: Goodput, ATP and our variants, Testbed B

5.3 Generality

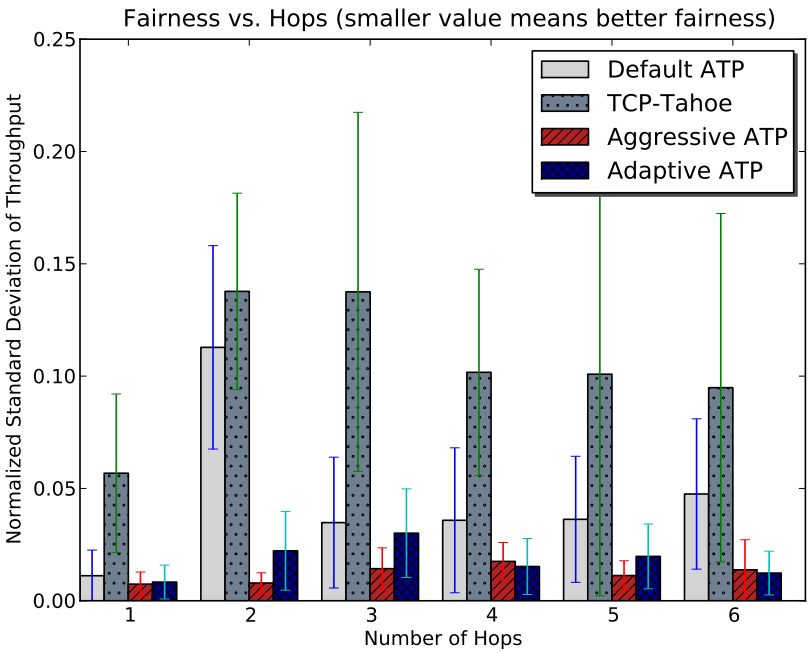
To demonstrate the generality of our results, we examine ATP's performance under a significantly different environment using Testbed B from Figure 4.2. We configure each mesh node to use a bit rate of $6\,Mbps$ and a power of $17\,dBm$. The increased transmission power and reduced bitrate, in addition to the sparser distribution of nodes, provides a more reliable, stable mesh with lower bandwidth. For brevity, we show only the goodput and fairness results with respect to the number of flows.

Figure 5.6 shows the goodput for the one, two, and five flow experiments. Due to the low bandwidth and better link quality, there is not much difference exhibited between different protocols in the one and two flow scenarios, and TCP's performance is much better. For five flows, TCP outperforms ATP and its variants for paths over three hops long. Aggressive ATP generally has the worst goodput among all the tested protocols in this scenario, which further confirms that the aggressive parameters setting may not be beneficial under all circumstances.

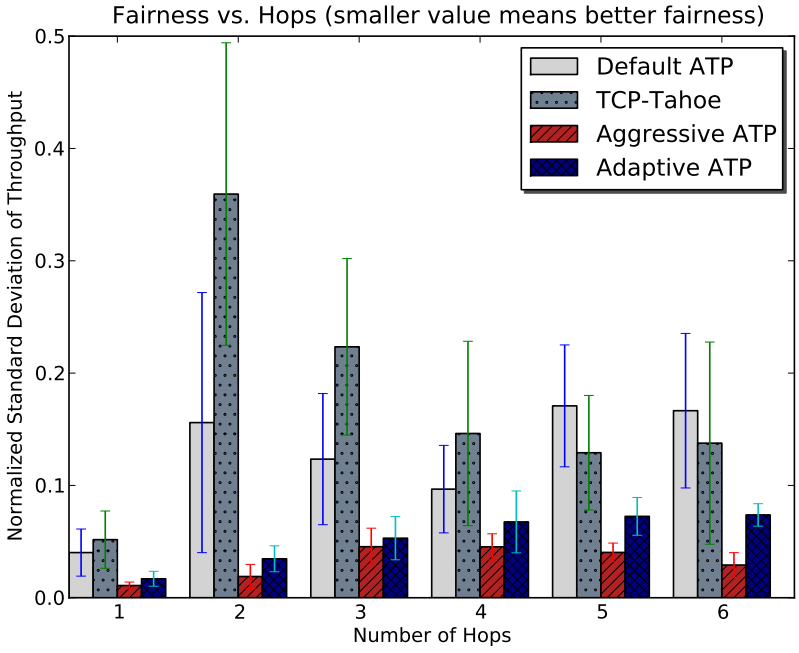
Figure 5.7 shows the fairness results for the same two and five flow experiments on Testbed B. It is clear that the goodput advantage of both TCP and default ATP shown in the previous figure are achieved at the sacrifice of fairness. TCP generally exhibits the worst fairness among all those tested. Meanwhile, aggressive ATP generally shows the best fairness over all, which suggests that a shorter epoch timeout contributes to better fairness. It is possible that adaptive ATP should also try to adjust the epoch time so that it is more fair in this situation.

5.4 Fairness with Stack Topology

Our fairness results so far are all for flows that share the same path during experiments. A common stack topology tests fairness when two flows on the edges of a network starve a flow in the middle due to the MAC layer unfairness [26]. The research community has ad-



(a) Two Flows, Multihops



(b) Five Flows, Multihops

Figure 5.7: Fairness, ATP and our variants, Testbed B

dressed this issue in numerous works [12, 19, 25, 26] with various approaches. To explore ATP's behavior in this situation, we construct a stack topology using our 1st floor nodes as shown in Figure 5.8. We run an experiment that starts flows 1, 2 and 3 sequentially, with a 3 second delay between each flow.

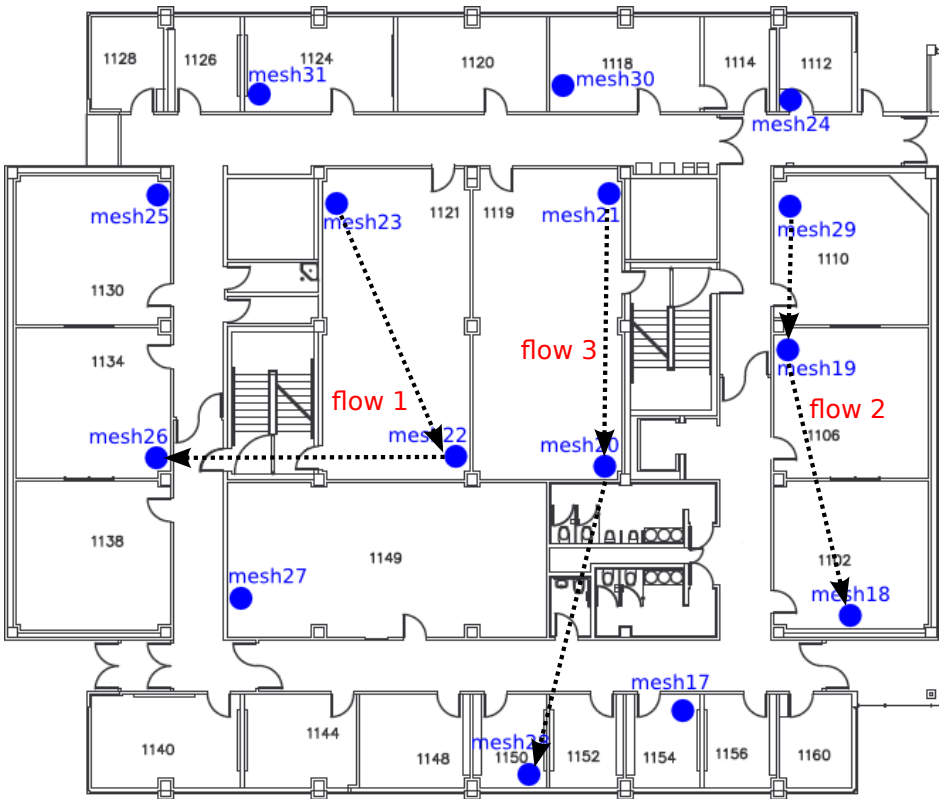
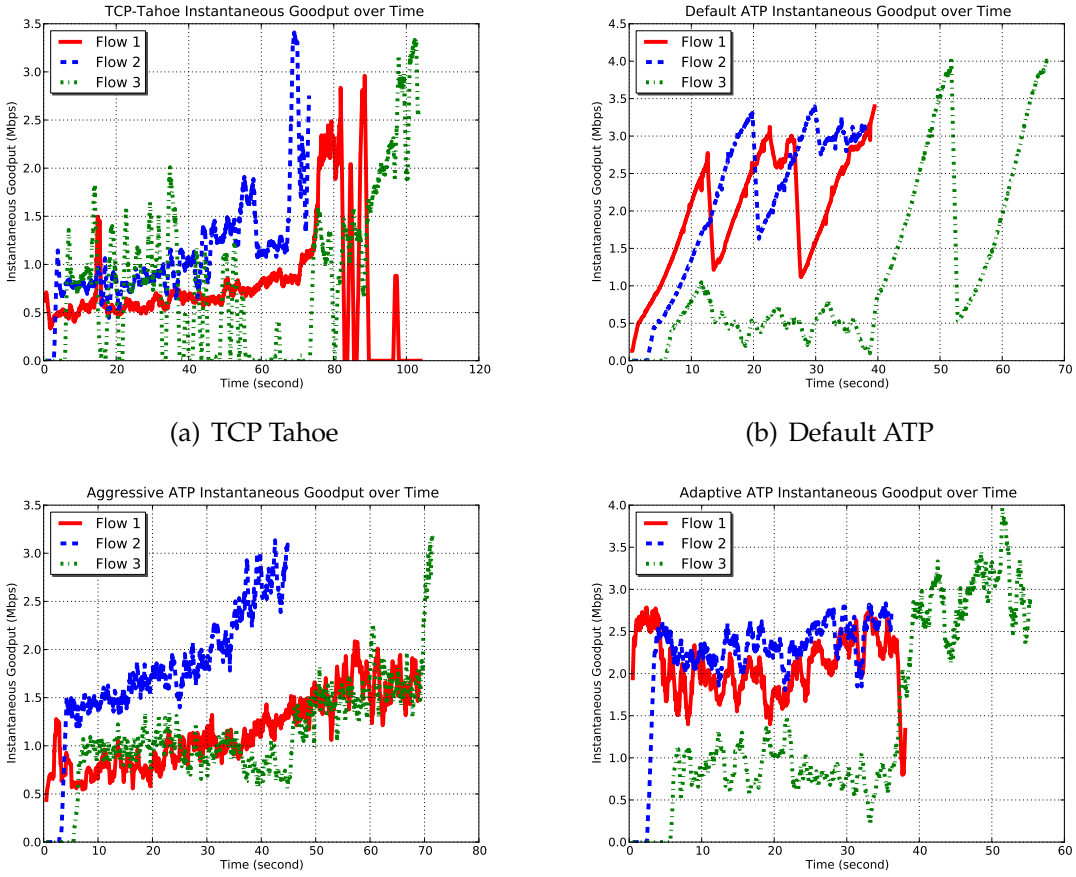


Figure 5.8: Stack Topology Constructed on 1st Floor Mesh

Figure 5.9 shows the instantaneous goodput of each flow for different protocols. Since our stack topology does not contain the exact spacing that is often used in simulations and mathematical models, we can not estimate the appropriate fair share for each flow. However, the difference among the protocols in how they handle this situation are enlightening. Tahoe periodically starves the flow in the middle (flow 3), which validates our topology configuration. Default ATP does not perform as badly, but does provide the middle flow with only 1/5 the throughput of the other two flows. The sawtooth pattern in this case is due to the combination of severe penalty for a missed feedback packet and ATP's conservative rate recovery. Both aggressive and adaptive ATP provide much better



(a) TCP Tahoe

(b) Default ATP

(c) Aggressive ATP

(d) Adaptive ATP

Figure 5.9: Goodput trace, TCP and ATP variants, Stack topology

goodput for the middle flow, with aggressive ATP taking more bandwidth away from flow 1. Table 5.1 lists the average goodput each flow obtains when all three flows are active with transmission. From the channel utilization perspective, both adaptive ATP and default ATP achieve better total goodput during the contention period, and adaptive ATP provides the best utility, as measured by the log of the goodput.

5.5 Randomized Flows

Finally, we perform a series of experiments that mimic the use of a mesh network. In these experiments, we randomly choose a set of flows in both the 1st and 2nd floors of our mesh network, varying the number of simultaneous flows to impose different loads.

Protocol	Flow 1 (kbps)	Flow 2 (kbps)	Flow 3 (kbps)	Aggreg. (kbps)	Utility
TCP Tahoe	703	1184	480	2368	8.60
Default ATP	2246	2483	546	5275	9.48
Aggressive ATP	966	1992	951	3909	9.26
Adaptive ATP	2085	2410	875	5369	9.64

Table 5.1: Goodput Share of Each Flow in Stack Experiment

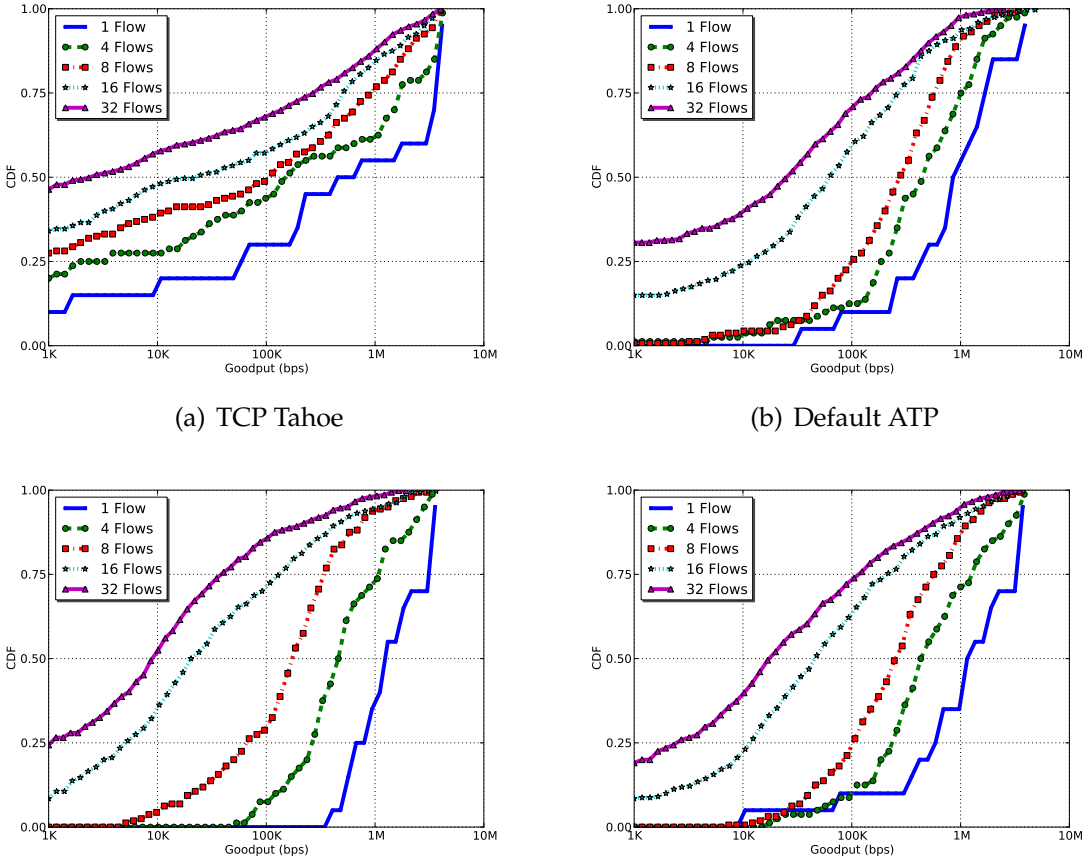
We examine two scenarios: *file transfer* and *streaming*. For file transfer, we simultaneously initiate a 1 MB file transfer between randomly pairs of nodes and then calculate the overall goodput. For streaming, we simultaneously initiate a backlogged TCP transfer between the randomly chosen pairs and terminate the transmission after 30 seconds.

In these experiments, we use the OSLR [24] routing protocol, rather than static routing, similar to how a mesh network would be operated. To obtain longer routes, we reduce the power to $2\ dBm$ and use a bit rate of $6\ Mbps$. This provides a maximum path length of 4 hops in our mesh. We repeat each experiment 10 times with different random flows.

5.5.1 File Transfer

Figure 5.10 shows the CDF of per-flow measured goodput for different number of simultaneous flows. We consider any flow with goodput lower than $1\ Kbps$ as starved. Clearly, TCP Tahoe has a greater proportion of starved flows. With 32 flows, the Tahoe starves nearly half of the total flows. All versions of ATP perform very well from the fairness perspective, with almost no starvation for up to 8 simultaneous flows. For 32 flows, ATP also has some starvation, with percentages from 20% to 30%, and adaptive ATP performs best among the variations.

We next consider the tradeoff of aggregate goodput and fairness for this experiment. For fairness, we use the log utility, which is the sum of the log of each flow's goodput, with a minor adjustement to avoid a singularity in the calculation. When taking



(a) TCP Tahoe

(b) Default ATP

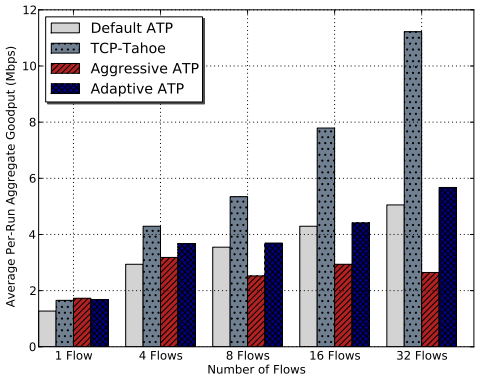
(c) Aggressive ATP

(d) Adaptive ATP

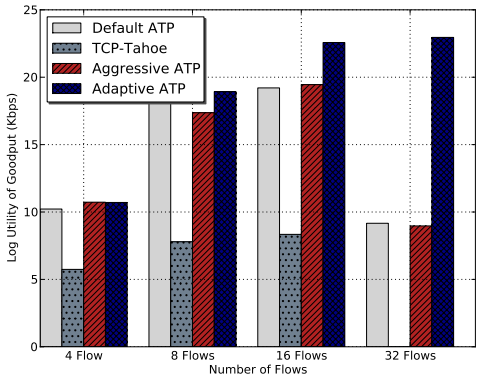
Figure 5.10: CDF of per-flow goodput, random file transfer

the log of a flow's goodput, any totally starved flow yields negative infinity. Thus for starved flows, we assign their goodput to one byte.

Figure 5.11 shows both the aggregate goodput and log utility for the file transfer experiments. Tahoe generally obtains the best aggregate goodput for the multiple flows runs. However, this performance is achieved at the cost of overall utility, which is the lowest of all protocols tested and is zero for 32 flows due to starvation of some flows. Adaptive ATP has both the best aggregate goodput among all ATP versions and also the highest utility.

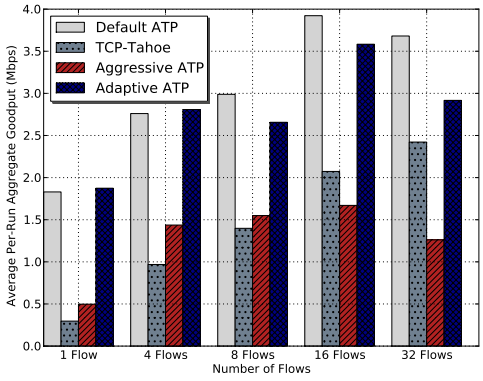


(a) Average aggregate goodput

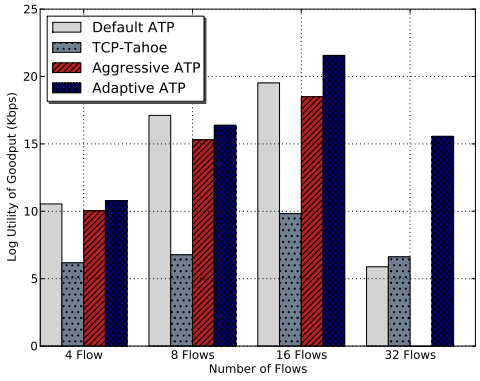


(b) Average log utility

Figure 5.11: Goodput and fairness, random file transfer



(a) Average aggregate goodput



(b) Average log utility

Figure 5.12: Goodput and fairness, random streaming

5.5.2 Streaming

The streaming experiments show drastically different goodput and utility results than the file transfer experiments. Figure 5.12(a) shows the aggregate goodput for the 30 second streaming experiments. Default ATP has much better performance here, and fairly good utility. This performance advantage can be attributed to the similar operating environment, i.e. low bitrate links, where default ATP has been optimized for in the original ATP paper [21]. Adaptive ATP performs competitively with the default settings in most cases, but has much better utility as the number of flows increases. The CDF of per-flow goodput is similar to the file transfer case, so we do not show them here.

Chapter 6

Related Work

There has been extensive research to improve the performance of transport protocols in multihop wireless networks in the last decade. However, relatively few fully implemented transport protocols are available, and experimental results are still uncommon. Two TCP alternatives that have been implemented are DiffQ [25] and Hop [16]. DiffQ applies the theoretical work of cross-layer optimization and develops a practical backlog-based MAC scheduling algorithm with router-assisted backpressure congestion control. This work includes an implementation of ATP, but only the congestion control portion of the protocol. Hop builds hop-by-hop transport protocol with in-network caching. It uses *blocks*, large segments of contiguous data, instead of packets, and uses transport-layer reliability to achieve both higher overall throughput and robustness with lossy wireless links. Their implementation is also done in user space running over UDP. Other recent experimental work focuses on the fairness of TCP in wireless networks, using protocols that allocate rates and work in conjunction with existing TCP protocols [12, 19].

Implementing network protocols at the user-level is not something new to the networking community [23] [6] [14]. The comparative advantages of user-level protocol implementation, such as the ease of prototyping and debugging are well established. Notable transport protocols include Alpine [8], which provides a framework for user-level network protocol development in FreeBSD, and Daytona [18], which implements a user-level TCP stack for Linux.

Chapter 7

Conclusion

In this thesis, we present an extensive performance evaluation of ATP in a wireless mesh testbed. We have examined the performance of ATP as it was designed, with regard to both goodput and fairness, and found that its original design does not perform as well as shown in earlier simulation results. Although it generally outperforms TCP Tahoe in terms of goodput, especially for the longer paths with lossy links, ATP's quick-start and fixed epoch feedback mechanism need improved designs. This is another piece of evidence that network protocols should be evaluated with both simulations and an implementation in order to properly validate their performance.

Our exploration of a revised design for ATP includes both using more aggressive parameters and creating a more adaptive feedback mechanism. A more aggressive ATP improves performance in some circumstances, but does not perform as well with multiple competing flows. Our adaptive design of ATP demonstrates a good mix of high goodput and fairness under most circumstances tested. In the particular case of high bit-rate, low quality links as shown in our testbed A, the goodput gain obtained by adaptive ATP over default ATP is on average of 64% for a single flow. This goodput gain is diminishing for the more saturated links and multiple flows scenarios. For low bit-rate, high-quality links, the advantages of adaptive ATP with respect to goodput are much less significant, since all transport protocols tested operate in a near ideal environment. However, in all situations our adaptive version of ATP generally has much better fairness – usually an order of magnitude better than default ATP and TCP Tahoe.

Due to unfairness at the MAC layer, ATP and our variants alone will not solve all the problems encountered by transport protocols in a wireless mesh network. However, we are encouraged that rate-based congestion control has been shown to be useful in a wireless setting, and that cross-layer feedback of delays encountered at the MAC layer can improve performance.

With the help of the WiFu toolkit, we plan to expand our experimental evaluations to include more transport protocols proposed by other researchers. In particular, those works that haven't been evaluated with implementations and experiments are of special interest to us.

Appendix A

Delay Calculation and Driver Modification

The ATP congestion control algorithm is built upon the real-time measurement of queuing delay (Q_t) and transmission delay (T_t) on each intermediate node. Queuing delay (Q_t) is the duration between when a packet is inserted into a transmission (TX) queue and when it is removed from the queue for transmitting, i.e. $Q_t = T_{dequeue} - T_{enqueue}$. Transmission delay ($T_t$) measures how long it takes to actually send the data packet, i.e. $T_t = T_{tx\_done} - T_{dequeue}$. Due to contention experienced by wireless medium access control, this transmission delay can not be calculated by simply dividing the packet size by the current transmission rate. Generally, queuing delay reflects the congestion among multiple flows that traverse the current node and transmission delay is influenced by the contention among the current node and its neighbors. In ATP, the exponential average (EWMA) of these two delays are considered to be indicators of congestion and contention and they are used to derive the appropriate transmission rate at the sender.

However, these two measurements are not readily available via the current open source Linux wireless driver, i.e. `ath5k`. Instead, we measure the total of queuing and transmission delay for each packet. This section provides background on the Linux kernel network stack and explains our modifications of the `ath5k` driver.

A.1 Architecture of Linux Network Stack

The architecture of Linux networking stack is directly inherited from the BSD stack, with well organized encapsulated interfaces. Figure A.1 illustrates a high-level overview of the stack. User-space applications, such as Firefox, FTP, etc., gain access to the kernel's networking stack via the **system call interface**. This interface de-multiplexes the call from user-space to a specific targeted socket through the `system_socketcall()`. Furthermore, the system call interface defines the network I/O as normal file operations, where socket read/write corresponds to file read/write to the socket file descriptor.

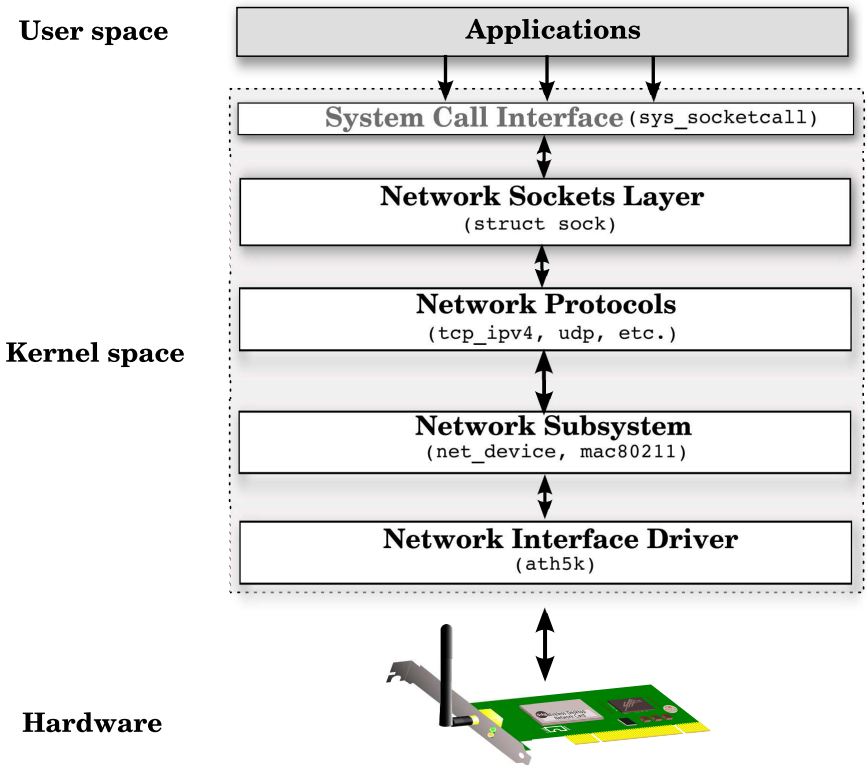


Figure A.1: Overview of Linux Network Stack

The **Network Sockets** layer provides support for different protocols, such as TCP, UDP, IP, etc., via the socket structure `sock` (defined in `linux/include/net/sock.h`). Basically, `sock` contains the state information of a connection, the particular protocol, and the operations that could be performed on the specific socket. The **Network Protocols**

layer defines a wide variety of networking protocols and initializes them as the system boots. It also maps the individual protocol to the corresponding module that supply the operations. One critical data structure *socket buffer* (`sk_buff`) stores data across the multiple layers of the protocol stack. The **Network Subsystem** layer connects network protocols to network interface drivers. The device drivers are registered to the kernel via the `net_device` structure. To transmit a packet (or `sk_buff`) to a network interface, this layer en-queues the packet using `dev_queue_xmit` and then calls `dev->hard_start_xmit` to initiate the transmission to the network device driver. Recently, a new application program interface (NAPI) was introduced into the kernel to provide support for the low-level network device drivers to operate with the high-level protocol stack. mac80211 is a special network subsystem that implements shared code for soft-MAC wireless devices. Finally, the **Network Interface Driver** manages operations on the physical network hardware. Most of the modern network device drivers are implemented as kernel modules, which can be flexibly loaded/unloaded as the device is inserted/pulled.

A.2 mac80211 Subsystem

The IEEE 802.11 specification defines common operations, such as: beacon, probe, associate, authenticate, etc., that should be available on any IEEE 802.11 compliant wireless device. The MLME (Media Access Control (MAC) Sublayer Management Entity) implements these operations in hardware or software for a wireless network device. Depending on where the MLME is implemented, the wireless network card can be classified as either Full-MAC (hardware) or Soft-MAC (software). Most modern wireless network cards are classified as Soft-MAC.

The mac80211 subsystem of the Linux kernel sits between the network interface driver and network protocols in the kernel network stack. It provides a framework for the Soft-MAC wireless device development and greatly reduces the effort required to develop wireless device drivers. Figure A.2 demonstrates the structure of mac80211 subsystem,

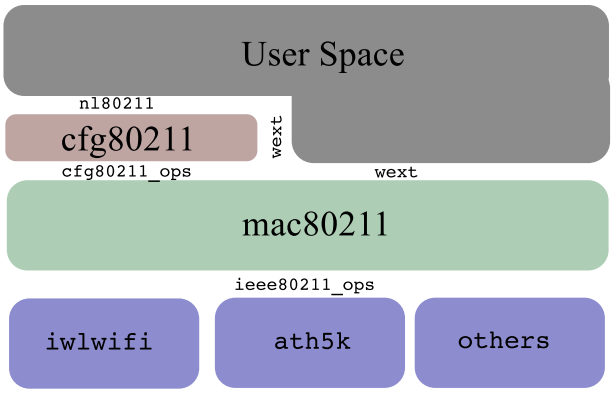


Figure A.2: Architecture of mac80211

where cfg80211 provides the configuration interface between user-space applications and 802.11 devices via mac80211. mac80211 implements the shared code of common MLME functionality for all Soft-MAC compatible 802.11 wireless devices. From the packet transmission perspective, mac80211 helps convert a packet from the network protocols layer to the IEEE 802.11 format, direct it to the master interface, initialize the transmit handlers, and create control information for transmission by the wireless hardware driver. In addition to the configuration hand-off through the cfg80211 module (via `cfg80211_ops`) and the TX/RX hand-off through individual wireless device drivers (via `ieee80211_ops`), there is another important functionality handled by mac8021 called *rate control*. This is a mac80211 subsystem that implements a variety of rate control algorithms, which an individual driver can select from. mac80211 is also informed of the actual transmission status, either success or fail, from the driver and adjusts the actual transmission rate for the subsequent packets.

A.3 Ath5k Driver

`ath5k` is a complete open source Linux driver for Atheros wireless cards. It is derived from MadWifi, a Linux wireless driver that uses a proprietary, closed-source HAL, and OpenHAL, an effort to replace HAL with open source code. It's still under heavy devel-

opment but considered to be stable enough to ship as the default wireless driver for many Linux distributions. In `ath5k`, the driver directly calls the hardware functions instead of intermediate HAL or OpenHAL layer.

The `ath5k` driver has a number of source code files that are relevant to our project. First, the hardware registers are defined inside `reg.h` under `/wireless/ath/ath5k/`. The initialization and PHY control is managed by `initvals.c` and `phy.c`. The major mac80211 and PCI interface is defined in `base.h` and implemented in `base.c`. Finally, the data structures of the `ath5k` driver are defined inside `ath5k.h`.

Since packet transmitting and receiving is an asynchronous process, `ath5k` utilizes an interrupt mechanism to coordinate transmission and reception between the hardware and the kernel. Basically, at the driver initialization phase, an interrupt handler and tasklet is registered for each major event, such as TX and RX. When a new packet is received/transmitted by the hardware, an interrupt is raised and the tasklet is scheduled at a later time to process the TX/RX descriptor. After successfully processing the interrupt, the tasklet will inform the mac80211 subsystem about the TX/RX status, which is used for rate control or other control information generation.

A.4 Important Data Structures

In order to trace a packet's movement inside the Linux kernel, we need to understand how a packet is represented and manipulated across the network stack. There are two critical data structures, *socket buffer* (`skb`) and *network device* (`net_device`), that make the Linux network implementation both efficient and flexible. A socket buffer (`skb`) represents a packet during its lifetime inside the kernel. The `net_device` provides an abstraction for the network adapter and a uniform interface for higher protocol instances.

Figure A.3 illustrates the basic structure of the `skb`. This doubly linked list consists of all the buffers used by the network layers. It's used to keep the status of each packet with a block of memory attached. For example, `next` and `prev` are pointers to the adjacent

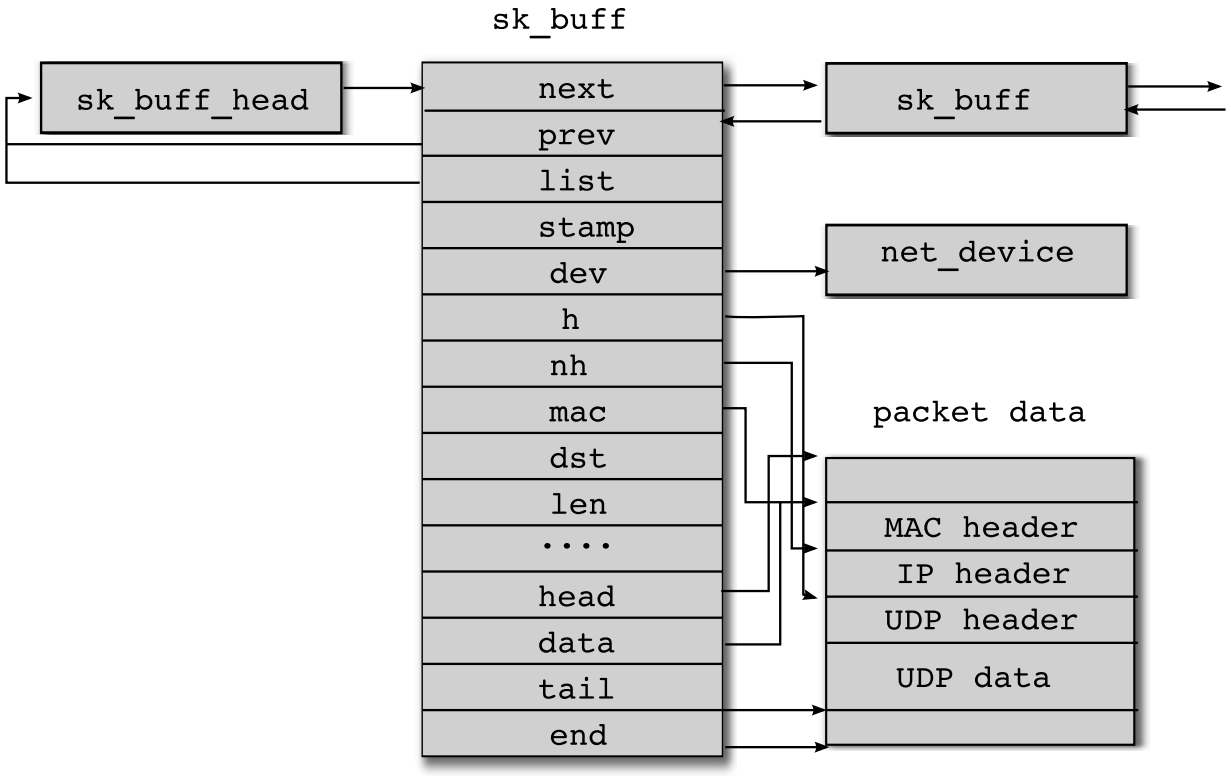


Figure A.3: Structure of socket buffers

socket buffers and concatenate `skb` into queues; `dev` is a reference to the designated network devices this packet is intended for or received from; `stamp` specifies the arrival time of this packet in Linux kernel; `h`, `nh`, and `mac` are pointers to packet headers of the transport layer, network layer, and the link layer; `head` and `end` point to the range of memory allocated for the packet data; `data` points to the currently valid protocol data unit for the specific layer processing the packet.

The life cycle of a socket buffer can be summarized as following. When a new packet is received or some data is ready to transmit from an application, a new socket buffer is created and memory is allocated for it. As the packet travels across different layers in kernel network stack, operations are performed on the data using `skb` pointers. After it's processed and delivered, the corresponding `skb` is destructed and memory is freed.

For the ath5k driver, there is a wrapper structure, ath5k_buff, around skb with a link to a virtual descriptor and the physical address of socket buffer data. Figure A.4 demonstrates the structure of ath5k_buff, where a similar doubly-linked list is used. desc is a pointer to the virtual descriptor address and skb is pointing to the socket buffer we described above. daddr and skbaddr are physical addresses of the descriptor and socket buffer data. Corresponding to the role of skb for a packet, ath5k_buff identifies a specific packet inside the wireless device driver.

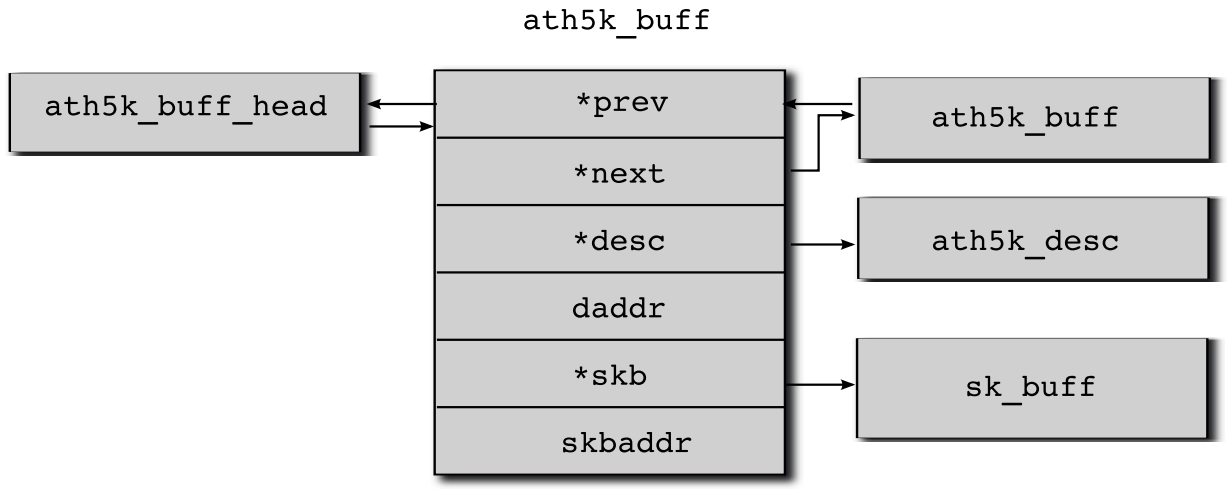


Figure A.4: Structure of ath5k buffers

A.5 Packet Transmission Path in ath5k Driver

Since we are most interested in measuring the transmission delay and queuing delay of a packet, we will only focus on the discussion of packet transmission path. Figure A.5 shows an overview of the functional flow related to packet transmission. At the bottom is the wireless network card, where registers are used to control the actual sending and receiving of the IEEE 802.11 data frame. On the top is the mac80211 subsystem, which implements MLME, provides rate control, and converts a packet into the IEEE 802.11 format. Because our wireless network cards are PCI type, the PCI device driver (ath5k_pci_driver) performs the common initialization job (ath5k_pci_probe) during ma-

chine boot time. Also, it helps "attach" the hardware interface to the mac80211 interface (`ath5k_attach`) so that driver can talk directly to hardware. One of the critical tasks finished during this attach call is the tasklet initialization, where tasklets for TX/RX are initialized to handle interrupts from the wireless network card.

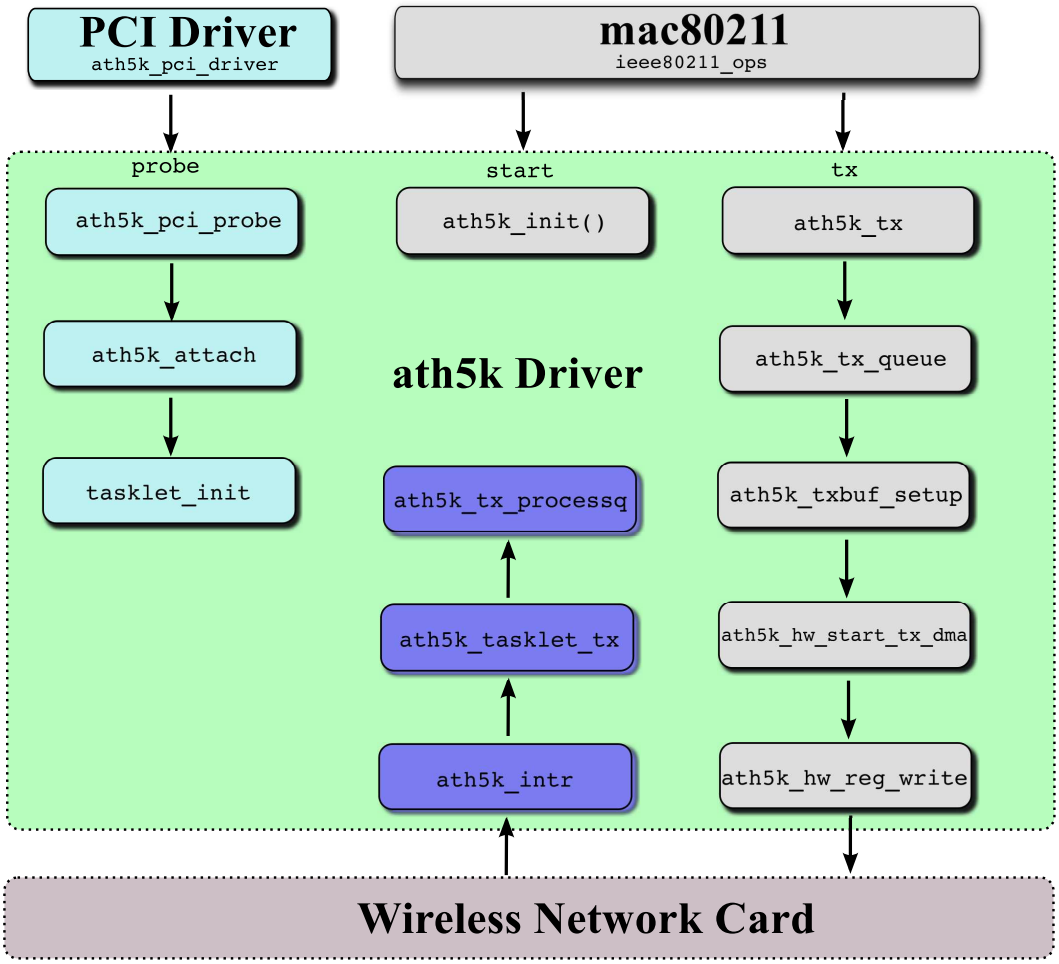


Figure A.5: Functional flow of packet transmission in `ath5k`

The mac80211 subsystem interacts with `ath5k` driver through `ieee80211_ops`. Two of the most important calls are *start* and *tx*, which are directed to the `ath5k_init` and `ath5k_tx` functions in the driver, respectively. `ath5k_init` basically resets the hardware and configures the interrupt masks for interrupt handling during TX/RX. When a packet is passed along from the mac80211, `ath5k_tx_queue` will initialize the transmission buffer and map it to the hardware queue. `ath5k_txbuf_setup` then pushes the packet into the TX

queue if there's still room left, otherwise, it drops the data packet. `ath5k_hw_start_tx_dma` starts the DMA transmit for a specific TX queue in hardware and `ath5k_hw_reg_write` actually writes the data frame into the wireless network card's registers and makes it ready to send. After the data frame is successfully transmitted, the wireless network card will inform the driver via an interrupt, where the interrupt handler, `ath5k_intr`, puts it into the TX tasklet, `ath5k_tasklet_tx`, and schedules a later time to process the TX descriptor by `ath5k_tx_processq`.

A.6 Driver Modification and Kernel Data Export

The goal of our driver modification is *measuring the exponential averaging of queuing (Q_t) and transmission delay (T_t) and exporting them to user-space*. However, there are two details in this process that hinder our exact measurement of queuing and transmission delay. The first issue is the detailed de-queue behavior of wireless network card. Even though the driver has control over the packet enqueue process, the dequeue process is controlled by a QCU (Queue Control Unit), which is built into the hardware. This means that even if we can tell exactly when a packet is put into the TX queue ($T_{enqueue}$) without modifying the firmware, it's impossible for us to know when it's de-queued ($T_{dequeue}$) and the actual transmission starts. This points to one of the inherited limitations of using simulation as the performance evaluation method for wireless transport protocols. It's quite easy to "measure" the queuing and transmission delay of a packet in the simulation. In reality, due to constraints from a hardware implementation, it's very difficult, if not totally impossible, for us to measure the exact de-queue time. However, there's a compromise solution to this issue. ATP maintains an exponential average of both transmission and queuing delay per node and then sums it as the exponential average of total delay. Instead, we can measure the combined total delay ($D_Q + D_T$), since we can get the transmission finish time (T_{tx_done}) by recording the TX interrupt time. The difference between this transmission finish time and a packet's enqueue time is the sum of transmission and queuing

delay:

$$\begin{aligned} D_T + D_Q &= (T_{tx\_done} - T_{dequeue}) + (T_{dequeue} - T_{enqueue}) \\ &= T_{tx\_done} - T_{enqueue} \end{aligned} \tag{A.1}$$

Then, we can maintain an exponential average of this total delay per node. Theoretically, this approach is the same as the one proposed by ATP.

The second issue is the interrupt handling for transmission. In the driver initialization phase (`ath5k_init`), an interrupt mask is configured to determine which interrupts will be handled by the driver. Even though we can enable it to process the interrupt for every single successful packet transmission (`TXOK`), overall performance will suffer due to the interrupt overload. Thus, by default, `ath5k` driver is configured to only enable two interrupts: one for the end-of-line (`AR5K_INT_TXEOL`), which indicates this data frame is the last of current TX queue; and one for TX descriptor (`AR5k_INT_TXDESC`), which indicates the TX queue is getting deep and a bunch of data frames is being transmitted. This default interrupt handling configuration imposes a potential problem to our total delay measurement since we rely on the interrupt as the TX finish time. When the wireless network card is not busy transmitting data, the EOL interrupt can be used as a good approximation for the TX finish time for a packet because it may be the only packet in the TX queue. However, if the wireless card is busy sending data frames, the descriptor interrupt only reflects the last transmitted packet's finish time, not for the rest of packets sent over air in a batch, especially the first one in the TX queue. Therefore, the trade-off between a more accurate measurement and better performance under load is inevitable. We choose the latter since that's the default configuration for `ath5k` driver and we want to evaluate ATP performance in a more realistic environment.

The `ath5k` driver modification consists of two parts: 1) time-stamping every outgoing data frame; and 2) exporting the data from kernel to user-space. For the first part,

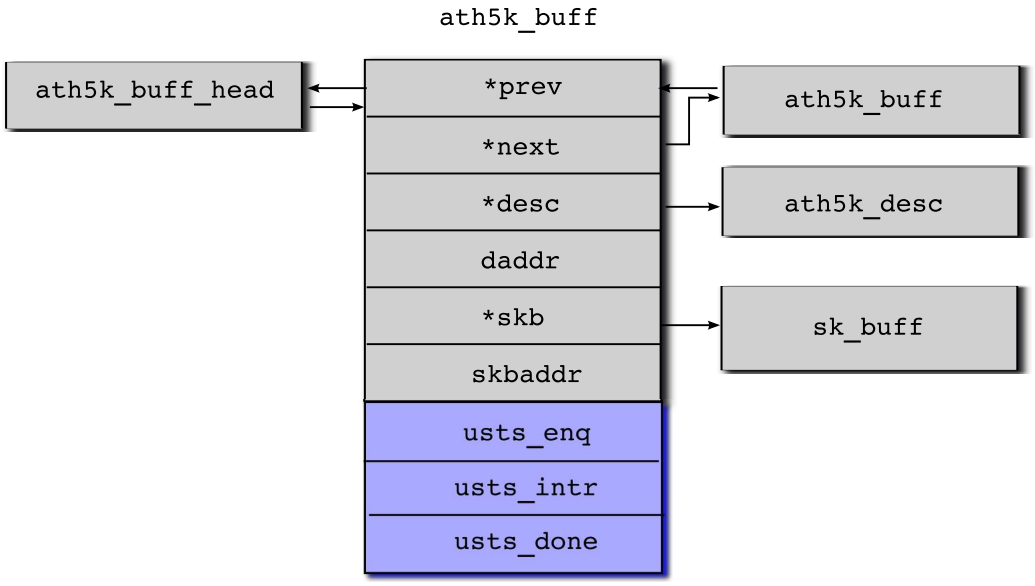


Figure A.6: Modified `ath5k_buff` data structure for data frame time-stamping

we can simply add three `timeval` structures to `ath5k_buff`, which represents a data frame inside `ath5k` driver. Figure A.6 illustrates the new, modified `ath5k_buff` structure, where `usts_en`, `usts_intr`, and `usts_done`, are used to record the enqueue, interrupt, and interrupt processing time, respectively.

In the Linux device driver, `jiffies` is normally used to manage the time interval. However, due to its low-resolution (millisecond precision), we use `do_gettimeofday` to achieve resolution of 1 microsecond. The rest of this time-stamping is straight forward, we can simply update the value of those three time structures of modified `ath5k_buff` at the proper place of the TX functional flow, as depicted in Figure A.5. For example, we update the `usts_enq` value inside `ath5k_txbuff_setup`, where a data frame is en-queued, and record `usts_intr` and `usts_done` at `ath5k_intr` and `ath5k_tx_processq`, for the interrupt and interrupt processing time, respectively. We apply an α value of 0.75 for the exponential averaging of the total delay (refer to Eqn. (2.1)).

Finally, to export the kernel data to user space, we have three choices: *procfs* (*/proc/* filesystem), *sysctl* (*/proc/sys/* directory), and *sysfs* (*/sys/* filesystem). The first and the third are virtual filesystems mounted at machine boot time. For simplicity, we choose *procfs*

because we only want to read the exponential average of the total delay exported from the `ath5k` driver. We create a sub-directory, */proc/ath5k/*, under */proc/* filesystem, to contain all the exported kernel information, where */proc/ath5k/Da* is the running average of the total delay for this node and */proc/ath5k/sample* is one sample delay for the last transmitted data frame. Applications can simply read the value out like a regular file or monitor the variation with a `cat` or `less` command under the Linux shell.

References

[1] Atheros Linux wireless drivers. URL `http://linuxwireless.org/en/users/Drivers/ath5k`.

[2] Minstrel rate control algorithm. URL `http://wireless.kernel.org/en/developers/Documentation/mac80211/RateControl/minstrel`.

[3] Adel Aziz, David Starobinski, Patrick Thiran, and Alaeddine El Fawal. EZ-Flow: removing turbulence in IEEE 802.11 wireless mesh networks without message passing. In *CoNEXT '09: Proceedings of the 5th International Conference on Emerging Networking Experiments and Technologies*, pages 73–84, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-636-6.

[4] B. Bensaou and Zuyuan Fang. A fair MAC protocol for IEEE 802.11-based ad hoc networks: Design and implementation. *IEEE Transactions on Wireless Communications*, 6(8):2934–2941, August 2007. ISSN 1536-1276.

[5] R. de Oliveira and T. Braun. A dynamic adaptive acknowledgment strategy for TCP over multihop wireless networks. In *INFOCOM '05: Proceedings of 24th Annual Joint Conference of the IEEE Computer and Communications Societies*, volume 3, pages 1863–1874, March 2005.

[6] Aled Edwards and Steve Muir. Experiences implementing a high performance TCP in user-space. *SIGCOMM Computer Communication Review*, 25(4):196–205, 1995. ISSN 0146-4833.

[7] Sherif M. Elrakabawy, Alexander Klemm, and Christoph Lindemann. TCP with adaptive pacing for multihop wireless networks. In *MobiHoc '05: Proceedings of the 6th ACM International Symposium on Mobile Ad Hoc Networking and Computing*, pages 288–299, New York, NY, USA, 2005. ACM Press. ISBN 1595930043.

[8] David Ely, Stefan Savage, and David Wetherall. Alpine: a user-level infrastructure for network protocol development. In *USITS'01: Proceedings of the 3rd Conference on USENIX Symposium on Internet Technologies and Systems*, pages 15–15, Berkeley, CA, USA, 2001. USENIX Association.

[9] Z. Fu, P. Zerfos, H. Luo, S. Lu, L. Zhang, and M. Gerla. The impact of multihop wireless channel on TCP throughput and loss. In *INFOCOM '03: Twenty-Second Annual Joint Conference of the IEEE Computer and Communications Societies*, volume 3, pages 1744–1753, 2003.

[10] Violeta Gambiroza, Bahareh Sadeghi, and Edward W. Knightly. End-to-end performance and fairness in multihop wireless backhaul networks. In *MobiCom '04: Proceedings of the 10th Annual International Conference on Mobile Computing and Networking*, MobiCom '04, pages 287–301, New York, NY, USA, 2004. ACM. ISBN 1-58113-868-7.

[11] Martin Heusse, Franck Rousseau, Romaric Guillier, and Andrzej Duda. Idle sense: an optimal access method for high throughput and fairness in rate diverse wireless lans. In *SIGCOMM '05: Proceedings of the 2005 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, pages 121–132, New York, NY, USA, 2005. ACM. ISBN 1-59593-009-4.

[12] Ki-Young Jang, Konstantinos Psounis, and Ramesh Govindan. Simple yet efficient, transparent airtime allocation for TCP in wireless mesh networks. In *Co-NEXT '10: Proceedings of the 6th International Conference*, pages 28:1–28:12, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0448-1.

[13] Wolfgang Kiess and Martin Mauve. A survey on real-world implementations of mobile ad-hoc networks. *Ad Hoc Network*, 5(3):324–339, April 2007. ISSN 1570-8705.

[14] Eddie Kohler and Eddie Kohler. The click modular router. *ACM Transactions on Computer Systems*, 18:263–297, 2000.

[15] David Kotz, Calvin Newport, Robert S. Gray, Jason Liu, Yougu Yuan, and Chip Elliott. Experimental evaluation of wireless simulation assumptions. In *MSWiM '04: Proceedings of the 7th ACM International Symposium on Modeling, Analysis and Simulation of Wireless and Mobile Systems*, pages 78–82, New York, NY, USA, 2004. ACM Press. ISBN 1581139535.

[16] Ming Li, Devesh Agrawal, Deepak Ganesan, and Arun Venkataramani. Block-switched networks: a new paradigm for wireless transport. In *NSDI '09: Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation*, pages 423–436, Berkeley, CA, USA, 2009. USENIX Association.

[17] Kitae Nahm, Ahmed Helmy, and Jay C. C. Kuo. TCP over multihop 802.11 networks: issues and performance enhancement. In *MobiHoc '05: Proceedings of the 6th ACM*

International Symposium on Mobile Ad Hoc Networking and Computing, pages 277–287, New York, NY, USA, 2005. ACM Press. ISBN 1595930043.

[18] Prashant Pradhan, Srikanth Kandula, Wen Xu, Anees Shaikh, and Erich Nahum. Daytona: A user-level TCP stack. URL `http://nms.lcs.mit.edu/~kandula/data/daytona.pdf`.

[19] Sumit Rangwala, Apoorva Jindal, Ki-Young Jang, Konstantinos Psounis, and Ramesh Govindan. Understanding congestion control in multi-hop wireless mesh networks. In *MobiCom '08: Proceedings of the 14th ACM International Conference on Mobile Computing and Networking*, MobiCom '08, pages 291–302, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-096-8.

[20] Daniel Scofield, Lei Wang, and Daniel Zappala. HxH: A hop-by-hop transport protocol for multi-hop wireless networks. In *WICON '08: Proceedings of the 4th Annual International Conference on Wireless Internet*, pages 16:1–16:9, ICST, Brussels, Belgium, Belgium, 2008. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering). ISBN 978-963-9799-36-3.

[21] K. Sundaresan, V. Anantharaman, Hung-Yun Hsieh, and A. R. Sivakumar. ATP: a reliable transport protocol for ad-hoc networks. In *MobiHoc '03: Proceedings of the 4th ACM International Symposium on Mobile Ad Hoc Networking and Computing*, pages 64–75, New York, NY, USA, 2003. ACM. ISBN 1-58113-684-6.

[22] K. Sundaresan, V. Anantharaman, Hung-Yun Hsieh, and A. R. Sivakumar. ATP: a reliable transport protocol for ad hoc networks. *IEEE Transactions on Mobile Computing*, 4(6):588–603, 2005.

[23] Ramohan A. Thekkath, Thu D. Nguyen, Evelyn Moy, Edward D. Lazowska, and Senior Member. Implementing network protocols at user level. In *IEEE/ACM Transactions on Networking*, pages 64–73, 1993.

[24] A. Tnnesen, T. Lopatic, H. Gredler, B. Petrovitsch, A. Kaplan, and S.-O. Tcke et al. olsrd - an adhoc wireless mesh routing daemon. URL `http://www.olsr.org`.

[25] A. Warrier, S. Janakiraman, Sangtae Ha, and I. Rhee. Diffq: Practical differential backlog congestion control for wireless networks. In *INFOCOM '09: Twenty-Eight Annual Joint Conference of the IEEE Computer and Communications Societies.*, pages 262–270, April 2009.

[26] Kaixin Xu, Mario Gerla, Lantao Qi, and Yantai Shu. Enhancing TCP fairness in ad hoc wireless networks using neighborhood RED. In *MobiCom '03: Proceedings of the 9th Annual International Conference on Mobile Computing and Networking*, pages 16–28, New York, NY, USA, 2003. ACM. ISBN 1-58113-753-2.

[27] Xin Yu. Improving TCP performance over mobile ad hoc networks by exploiting cross-layer information awareness. In *MobiCom '04: Proceedings of the 10th Annual International Conference on Mobile Computing and Networking*, pages 231–244, New York, NY, USA, 2004. ACM Press. ISBN 1581138687.