2012-04-06

# WiFu Transport: A User-level Protocol Framework

Randall Jay Buck
*Brigham Young University - Provo*

# WiFu Transport: A User-level Protocol Framework

Randall Jay Buck

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

Master of Science

Daniel Zappala, Chair
Sean Warnick
Robert P. Burton

Department of Computer Science

Brigham Young University

June 2012

# ABSTRACT

WiFu Transport: A User-level Protocol Framework

Randall Jay Buck
Department of Computer Science, BYU
Master of Science

It is well known that the transport layer protocol TCP has low throughput and is unfair in wireless mesh networks. Transport layer solutions for mesh networks have been primarily validated using simulations with simplified assumptions about the wireless network. The WiFu Transport framework complements simulator results by allowing developers to easily create and experiment with transport layer protocols on live networks. We provide a user-space solution that is flexible and promotes code reuse while maintaining high performance and scalability. To validate WiFu Transport we use it to build WiFu TCP, a decomposed Tahoe solution that preserves TCP semantics. Furthermore, we share other WiFu developers' experiences building several TCP variants as well as a hybrid protocol to demonstrate flexibility and code reuse. We demonstrate that WiFu Transport performs as well as the Linux kernel on 10 and 100 Mbps Ethernet connections and over a one-hop wireless connection. We also show that our WiFu TCP implementation is fair and that the framework also scales to support multiple threads.

Keywords: networking, wireless mesh, transport layer, protocol, framework, flexibility, code reuse, scalability, user-space networking, protocol decomposition

# ACKNOWLEDGMENTS

I would like to thank my sweet wife, Chia-Chen (Jenny) Buck for being with me during this great adventure. She has been and continues to be extremely supportive and patient, especially as I have spent many hours at school learning and growing. We have experienced the birth of our first son, Henry Austin Buck, while in graduate school. He remedies many frustrations each day when I come home to see his beaming smile and radiant joy.

Dr. Daniel Zappala is a great advisor, committee chair, and friend as we have explored new research areas together. He is always willing to meet to discuss questions and concerns that I have. His insight, help, and encouragement to take the next big step are invaluable. He has also provided many excellent comments while I have written this thesis.

I appreciate others who have willingly given of their time and energy to help me in my education and this thesis. My parents Michael Lon Buck and Kristine Deeben Buck are always supportive and loving. I thank my other graduate committee members Dr. Sean Warnick and Dr. Robert P. Burton for their willingness to help. I would also like to thank the entire BYU CS faculty, especially Dr. Ken Rodham and Dr. Charles Knutson.

Finally, the other students in the Internet Research Lab that I have had the privilege to get to know have helped with many questions and concerns. Scott Erickson helped early in the process, especially in finding good third-party libraries to incorporate into WiFu Transport. Special thanks goes to Rich Lee and Phil Lundrigan for their willingness to use WiFu Transport and build protocols. Travis Andelin, Justin Wu, Ryan Padilla, and Chase Johnson each added insight and help.

# Table of Contents

# List of Figures

# List of Tables

# Chapter 1

## Introduction

A wireless mesh network is a group of computers, phones, or other electronic devices formed into a "mesh." Each node in a mesh is responsible for routing to nearby nodes. Wireless mesh networks are important as they are a cost-effective and relatively fast solution to networking in situations where deploying infrastructure is infeasible because of cost, location, or transience. Some examples where wireless mesh networks are used include third-world countries, military battlefield scenarios, and urban EMS and utility networks. Furthermore, some nodes may be connected to the Internet to provide gateway service for the the rest of the wireless mesh.

It is well known that TCP performs poorly and is unfair when used in wireless mesh networks. These problems arise partly from TCP's assumptions about the underlying network and partly from the design of the IEEE 802.11 MAC. The primary reasons for poor performance are contention, interference, and MAC unfairness. First, contention occurs when nodes share the same wireless medium. There can be significant overhead induced when competing nodes try to determine whether the medium is free and back off when it is busy. The effect of contention is to significantly reduce the available rate over the path of the network, particularly as the length of the path increases. Second, interference occurs when at least one node does not correctly detect that another node is currently sending. Multiple nodes sending at the same time may cause one or both packets to be dropped. However, TCP assumes that any dropped packet is due to overflowing queues, and reduces its sending rate unnecessarily. Finally, the IEEE 802.11 MAC was not designed for wireless mesh networks and can lead to unfairness due to its back off algorithm. One-hop flows can entirely starve two-hop flows when they share the same first hop [39]. Furthermore, unfairness can also occur without starvation [17]. These three issues show clearly that there is a need to experiment with new, even radical, transport layer protocols for wireless mesh networks.

Much work in the community has been done to find solutions to these and other wireless mesh network specific issues. For example, TCP with Adaptive Pacing suggests pacing packets according to the four-hop delay [13]. TCP FeW fractionally increments the TCP congestion window [31]. ELFN adds notification of link failures to the transport layer [19]. Delayed ACKs also has shown promising results in a simulator [4]. Non-TCP variants include ATP, which suggests using the queuing and transmission delays to inform a rate-based sending algorithm [42]. HxH uses credit-based congestion control and reverse acknowledgements [38].

These solutions have been validated primarily using *simulations*; *experimentation* on a wireless mesh network to validate each new protocol is necessary. Simulations are helpful in understanding protocol behavior because they are deterministic, can be run quickly in simulated time, and can more easily test protocols on large scale networks. However, the networking community is increasingly recognizing the need for experimental results, particularly for wireless mesh networks. As an example, radio signal propagation is complex. Most simulators that emulate radio signal propagation make significant simplifications that may compromise the validity of their results. Despite this need, experimental results with transport protocols are relatively rare due to the complexity and time required to develop in the operating system kernel.

We are in the process of solving this problem by developing WiFu, a toolkit that can be used to quickly create and compare transport layer protocols and their components. Several user-level networking solutions exist or are under development [14, 35, 9, 1, 12]. However, they do not provide the performance or flexibility that we require and their source code, in most cases, is not available. We would like a user-level solution that can run as fast as the kernel. In addition, we would like a solution that provides the flexibility to easily mix and match pieces of protocols, create hybrids, and be cross-layered. In short, we need to be able to quickly create novel transport protocols on a real network. After sufficient testing in user-space, we can then consider moving the solution to the kernel for further testing.

There are two main pieces of WiFu: WiFu Core and WiFu Transport. WiFu Core is used in the middle nodes of a transmission and WiFu Transport is used at the ends of the transmission. WiFu Transport contains implementations of transport layer protocols, while WiFu Core is used to intercept, modify, reorder, or drop packets. WiFu Core and WiFu Transport are meant to complement each other in order to provide complete control over the transport layer both at the ends and at intermediate hops. We need both to be able to implement and test cross-layer solutions. An optimal transport layer protocol for a wireless mesh network may violate the end-to-end principle [37]; we need a mechanism that allows us to violate this principle by putting the necessary functionality in both the ends and the middle.

In support of this thesis, we build and test the end-to-end portion of the WiFu toolkit: WiFu Transport. We provide a standard BSD socket API for applications and a modular, event-driven framework for developing novel transport protocols. The product development cycle utilizes an object-oriented design, software design patterns [24, 40], and agile development methods to ensure quality. We push for a good balance among competing goals of (1) performing as fast as the Linux kernel implementation over a one-hop wireless link, (2) provide flexibility to develop many types of transport protocols in user-space, (3) promote software reuse by WiFu developers, and (4) be scalable with respect to the number of active transport connections.

Our goals have been met through performance testing and experiences from developers using WiFu Transport. We show that WiFu Transport is able to achieve transfer speeds equal to the goodput of the kernel on a wireless connection. Furthermore, we are as fast as the kernel on a 100 Mbps Ethernet connection. This is very encouraging and suggests that we may be able to use WiFu Transport outside the target area of wireless networks. In addition, three developers have participated in building transport protocols using WiFu Transport in a relatively short amount of time. One of the developers has created a hybrid of TCP [33] and ATP [42] to illustrate flexibility and code reuse. The speed at which we were able to implement these protocols and test them on a network testbed is sufficient evidence that WiFu Transport is an excellent solution and meets our goals.

# Chapter 2

# Related Work

We outline related work in two sections: user-level networking stacks and protocol frameworks. User-level networking stacks move a portion or all of the traditional TCP/IP code from the kernel to the application layer. Protocol frameworks provide tools to aid in the development of protocols, generally in user-space.

## 2.1 User-level TCP Stacks

Alpine [14] is a user-level networking stack that uses the FreeBSD kernel TCP/IP stack and sockets. It was built to ease the burden of development, debug changes more easily, and be completely compatible with existing applications. Applications do not need to change their code to use Alpine. While Alpine is theoretically usable for non-TCP protocols, it appears that this would be difficult to do; there is no example given where a non-TCP protocol is developed and used with Alpine. Also, the architecture is not modularized; it is more of a pseudo-kernel solution rather than a complete user-space solution. Therefore, it adheres more to the current networking stack rather than providing flexible methods to extend functionality. Furthermore, Alpine uses busy-waiting with a one millisecond sleep while checking for packet reception in the Packet Capture (PCAP) library [22]. PCAP is known to be slow, and polling with sleep is clearly a poor design decision that inserts an unnecessary bottleneck.

Another user-level networking stack is Daytona [35]. While Daytona only supports TCP, it has some advantages that we wish to emulate. First, Daytona is not dependent on any particular network interface or the Linux kernel version. Second, it provides the same functionality as the kernel. Our proposed framework builds on this idea in that we want to be able to easily extend protocols and thus do not limit ourselves to only what is available in the kernel; we want many protocols, whether they are in the kernel or not, to be available in our framework. There are a few points of dissimilarity between Daytona and our proposed framework. First, Daytona is designed for TCP enhancements only. Our solution seeks to go beyond the scope of the Daytona project by supporting non-TCP transport layer protocols and to be able to compare them quickly with other protocols. Second, Daytona started by taking the Linux networking stack code and putting it in user-space. Using the Linux architecture directly may limit the flexibility needed to develop new transport protocols. Therefore, we need a framework that allows us to build transport layer protocols and modify existing ones. Finally, like Alpine, Daytona uses the PCAP library and may consequently suffer from poor performance.

More recently, Faulkner et al. [16], who rely heavily on the work of Jacobson [21], suggest that moving networking code out of the kernel will potentially increase performance due to the ability to utilize multiple cores. In fact, Jacobson says that "The end of the wire isn't the end of the net" [21] suggesting that there is more work to be done on the ends of the network. This is consistent with the end-to-end principle [37].

## 2.2 Protocol Frameworks

Click [25] is a framework for building software routers using elements as its building blocks. These elements are able to be combined to perform a certain routing function. Each element, by itself, provides a small function such as a queue, counter, or a classifier. However, the power of Click is achieved as these small, reusable, elements can be combined to perform a myriad of router functions. This notion of decomposing a router into several reusable elements is something we wish to achieve at the transport layer level. However, we cannot simply use or modify the Click modular router for a few reasons. First, Click's focus is on IP layer routers, not transport layer protocols. While the architecture is similar, our proposed framework and the Click framework are two different things. We need to be able to focus on reliability and congestion control, not routing. Second, Click does not provide a socket API; it passes packets to and from the transport layer rather than replacing the transport layer. Finally, Click's modules are too decomposed for our use. We wish to deal in terms of congestion control and reliability modules rather than queues, counters, and classifiers.

Another framework under development is FINS [1]. The goal of FINS is to move the networking stack to user-space so that new mid-stack or cross-layer protocols may be easily and quickly modified and created. This is complementary to our work; FINS moves most of the networking stack into user-space. WiFu Transport is built on top of the IP layer so that we can truly focus on building transport protocols, especially in a wireless setting. FINS currently has several shortcomings such as the use of the PCAP library, needing to rebuild the kernel once, and supporting only one network application at a time. Some of these shortcomings are known and future releases are supposed to fix them. It is possible that WiFu and FINS could be merged at a later date, but both projects are still new and being concurrently developed.

CTP uses microprotocols to build larger transport protocols [9]. CTP focuses on the needs of applications and uses microprotocols to create protocols that meet those needs. As part of the solution, CTP provides something similar to TCP but does not provide packet traces to validate the fidelity of the implementation. A big drawback of CTP is that it runs on top of UDP, incurring extra overhead.

Minet [12] is a TCP/IP stack used primarily for coursework at Northwestern University. However, no code is available for use and the project looks abandoned.

The Internet Research Lab at Brigham Young University has built WiFu Core to provide application-level handling of packets at a router. The WiFu Core framework intercepts packets via the netfilter library [36] and Linux's `iptables`, brings them into user-space for processing, and re-injects them back into the kernel. Developers and researchers can easily create handlers that determine what packets are to be intercepted, when the packets are to be intercepted, and what to do with these packets once intercepted. The WiFu Core framework is to be used as a helper piece for end-to-end protocols. It is useful in determining what, if anything, should deviate from the standard end-to-end practice of networking.

As an example, we illustrate one way WiFu Core has been proven useful. The transport layer protocol ATP [42] sender requires knowing an average maximum delay (queuing delay plus transmission delay) for data packets among all nodes that it traverses. A developer in our lab has written a modified network driver that writes to the `proc` file system the delay for each packet at that node. WiFu Core inserts this information into each packet on the forward path. An earlier version of WiFu Transport consolidates it at the receiving node and sends back this feedback to the sending node. As WiFu Core intercepts acknowledgement packets, it simply reads the delay from the `proc` file system and inserts it into a special ATP header in the packet. After processing, the packet is injected back into the network for further processing.

# Chapter 3

## Goal-oriented Design Decisions

The WiFu Transport framework supports the rapid development of many transport protocols in user-space. These protocols might be pre-existing to be used for comparison (e.g. TCP), completely new or radical, or a hybrid of the two. In order to achieve this overarching goal, we detail four finer-grained goals that guide us in our solution. These goals may contend with one another. Gains in performance could mean losing some flexibility, code reuse, or scalability. We seek a balance among these goals and optimize where possible.

### 3.1 Performance

WiFu Transport should perform as fast as possible, while keeping in mind the other competing goals. More specifically, the protocols housed within our framework should be able to operate as fast as the Linux kernel over a one-hop wireless link. If the same protocol implemented in WiFu Transport can compete with the kernel's version, then a modification can be fairly evaluated in WiFu Transport, rather than requiring kernel development. Conversely, if our framework introduces unnecessary bottlenecks into the system, it might jeopardize protocol validation. Thus, we must ensure that the framework does not significantly impact protocol performance.

We have made several architectural and software engineering decisions to achieve this performance goal. First, we use C++ as our development language. C++ is a fast language that still provides an object-oriented approach. Second, we use an event-driven architecture built from the ground up, with object pools to re-use events where possible. Third, the use of raw sockets eliminates the overhead of developing protocols on top of an existing protocol such as UDP. Fourth, when receiving a packet from the network, we use the `epoll` system calls, not a library such as PCAP. Finally, our development process is ongoing. As we develop protocols and gain experience with WiFu Transport, we expect to learn how we can further improve the performance of WiFu. We continue to implement as many of these improvements as is feasible.

### 3.2 Flexibility

We define flexibility, for our purposes, as the types of protocols that our framework can support. Specifically, we would like to avoid being limited to variants of existing protocols such as TCP; we wish to be able to support as many types of protocols as we can. Wireless networking research has moved beyond considering only TCP variants to also including new, radical solutions that diverge from conventional solutions. These solutions include rate-based congestion control algorithms, cross-layer interactions, and modifications to how acknowledgements operate. In addition, any given protocol, validated via simulation, may not have the same results when tested on a real network. We intend to take principles and parts of promising solutions and aggregate them in search of a better protocol. Our framework should be flexible enough to support these known cases and adapt to new solutions.

We meet our goal of flexibility primarily by our choice of an event-driven architecture. Developers only need to handle events triggered by the socket API, the network, and protocol-specific requirements. Furthermore, events may be created or discarded as needed; there is complete control over event handling.

### 3.3 Code Reuse

Our work is indented for developers wanting to quickly implement transport protocols. Ideally, their primary concern should be to design, build, and/or test transport protocols and not menial implementation tasks such as building an event-handling infrastructure, timers, etc. The kernel provides many of these mechanisms that make the development process easier. One of the reasons that simulators are so popular is because they provide these needed tools for building protocols. Therefore, we create an environment in user-space, that provides these types of mechanisms. Furthermore, many protocols are similar in nature; it may require only minor modifications to a pre-existing protocol to exhibit new behavior and in essence become a new protocol. For example, TCP FeW [31] changes the way the congestion window is incremented. This is a minor change in the complex TCP protocol and code duplication should be avoided. We provide an environment that allows these small changes to be implemented and tested quickly.

We meet our goal of code reuse with the following choices. First, we use an object-oriented language, which allows us to utilize inheritance and polymorphism as needed. Small changes can be made by sub-classing and overriding relevant methods. Second, we use design patterns [24, 40] where applicable. Specifically, we utilize the state pattern as we create protocols. Third, we encourage protocol decomposition. For example, we have broken down TCP into three main components, each of which is independently a state machine and implemented with the state pattern: connection management, reliability, and congestion control. Each of these components is extensible, allowing us to override methods and have different protocol logic at varying levels of abstraction. Finally, we re-use the BSD socket API to access the protocols implemented within WiFu Transport. Developers familiar with this interface used by the kernel can re-use the same functions.

### 3.4 Scalability

Our final goal is scalability with respect to the number of concurrent connections supported by WiFu Transport. There are two main reasons for ensuring that our framework is scalable. First, fairness is a key metric when evaluating transport protocols. In a live network, transport protocols must deal with other network traffic. When experimenting with a protocol, we need to ensure we can test multiple flows on the same path, as well as flows on the same network but not necessarily on the same path. Second, the kernel can handle many processes, each with possibly many threads all accessing the networking stack. The kernel can also handle different types of transport protocols at the same time. WiFu Transport should support, as the kernel does, multiple heterogeneous sockets in different states on the same machine.

We have designed WiFu with this motivation in mind. We have separated WiFu Transport into two main pieces: a static library that contains a subset of the BSD socket API and a daemon that houses the transport protocols. The static library is thread-safe and supports most calls to the same socket API function at the same time. The daemon is event-driven and supports many protocols at the same time.

# Chapter 4

## WiFu Transport

In this chapter we detail the WiFu Transport architecture, which consists of a front-end static library and an event-driven back-end daemon, both of which are in user-space. In addition, we describe the building blocks of each piece and the communication methods we use between processes, modules, and components. A high-level diagram of the entire architecture is shown in Figure 4.1. WiFu Transport provides, in the front-end static library, a subset of the standard BSD socket interface and a translator that creates request messages, sends them to the back-end, and receives and parses the responses. The back-end provides the necessary functionality to facilitate building transport protocols, which includes a translator, event dispatcher, timeout manager, and network event handler.

Separating WiFu Transport into two main pieces is essential to mimic kernel behavior of the socket API. This allows developers to create standard applications that use WiFu Transport the same as if they were using the kernel. Existing applications should be able to be easily ported to WiFu Transport because it uses the same socket API. Like the kernel, WiFu Transport supports multiple applications each with possibly multiple threads.

Having WiFu Transport act like the kernel means it can support identical functionality. For example, we have considered implementing a congestion controller for groups of TCP connections, rather than on a per-flow basis. By using a daemon, WiFu Transport can support this style of control that requires aggregate information about sets of flows.

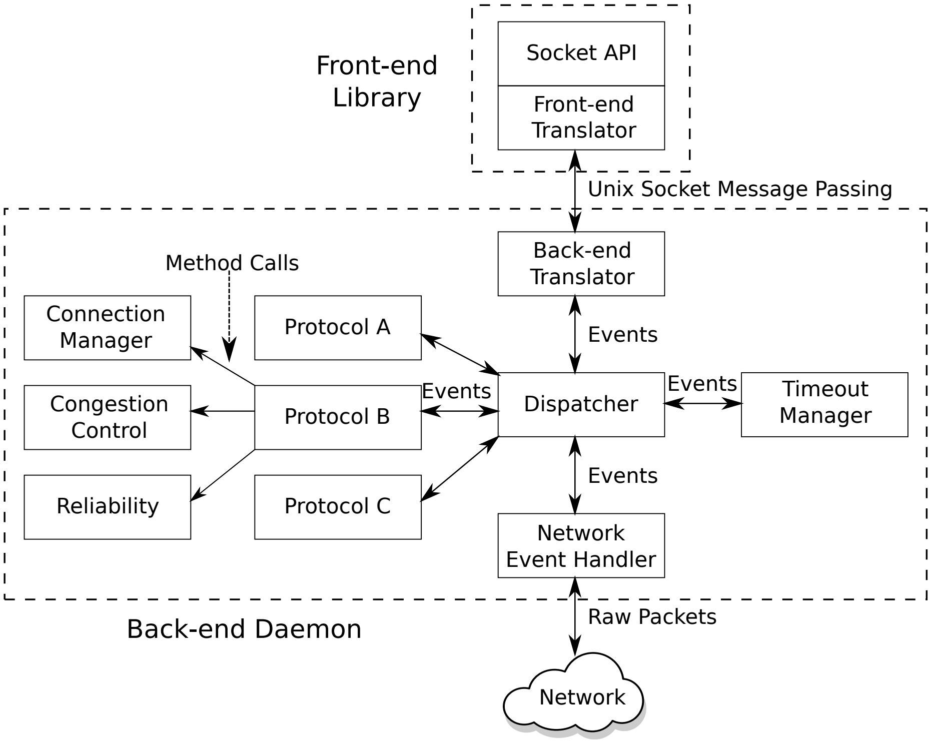


Figure 4.1: WiFu Transport Architecture

## 4.1 Front-end Library

The front-end portion of WiFu Transport replicates a subset of the standard BSD socket interface. It currently provides the following functions: `socket()`, `bind()`, `listen()`, `accept()`, `connect()`, `send()`, `sendto()`, `recv()`, `recvfrom()`, `getsockopt()`, `setsockopt()`, and `close()`. In our implementation, each function call is prepended with "wifu_" to avoid compilation errors with the kernel-provided socket API. Nevertheless, Ely et al. [14] were able to avoid recompilation of existing applications that use their networking stack by dynamically linking their library first; it is possible we will be able to do likewise in the future. Furthermore, we provide a wrapper test application that can use either the kernel interface or the WiFu interface. Using the same function definitions that already exist allows applications to easily switch between the standard Linux kernel implementation of a socket and our implementation. Through this application, we can compare application code fairly using either the WiFu or kernel socket API.

The WiFu Transport front-end is an *interface*. The *implementation* of the interface occurs in the back-end and may be protocol dependent. Our implementation of the API is currently limited to essential functionality, namely setting up the socket, setting up the connection, sending and receiving data, and tearing down the connection and socket. Additional support of the socket API calls, such as the many flags and options for each function, may be added later as needed.

When developers implement new transport protocols, it is often necessary to experiment with many protocol-specific parameters and values. For example TCP with Adaptive Pacing [13] uses an alpha value and history size in its algorithm. Rather than using a configuration file, we extend the `getsockopt()` and `setsockopt()` functions to pass protocol-specific data to the back-end implementation of the protocol. These options are automatically stored in a socket-specific object for use by the protocol.

This library should be able to support multiple applications, each with possibly numerous threads, as the Linux kernel does. Accordingly, the front-end library is thread-safe.

## 4.2 Message Passing

Interprocess communication between the front-end and the back-end is done via Unix sockets. The front-end library processes each API function call, creates a request message containing necessary parameter data from the BSD socket API call, sends it over a Unix socket to the back-end daemon, and then waits for a response. Upon receiving the function call response from the back-end, the front-end translates the message and fills in a return value, `errno` (if applicable), and any arguments used to pass data to the application such as a receive buffer.

Figure 4.2 shows both the request and response message formats. The top row represents request messages from the front-end to the back-end, while the bottom row depicts the responses. Columns in the figure move from most generic (left) to most concrete (right). All messages contain a type that correlates with the socket API function (e.g. `send()`, `recv()`, etc.), message length, and the file descriptor (socket). Furthermore, all request messages

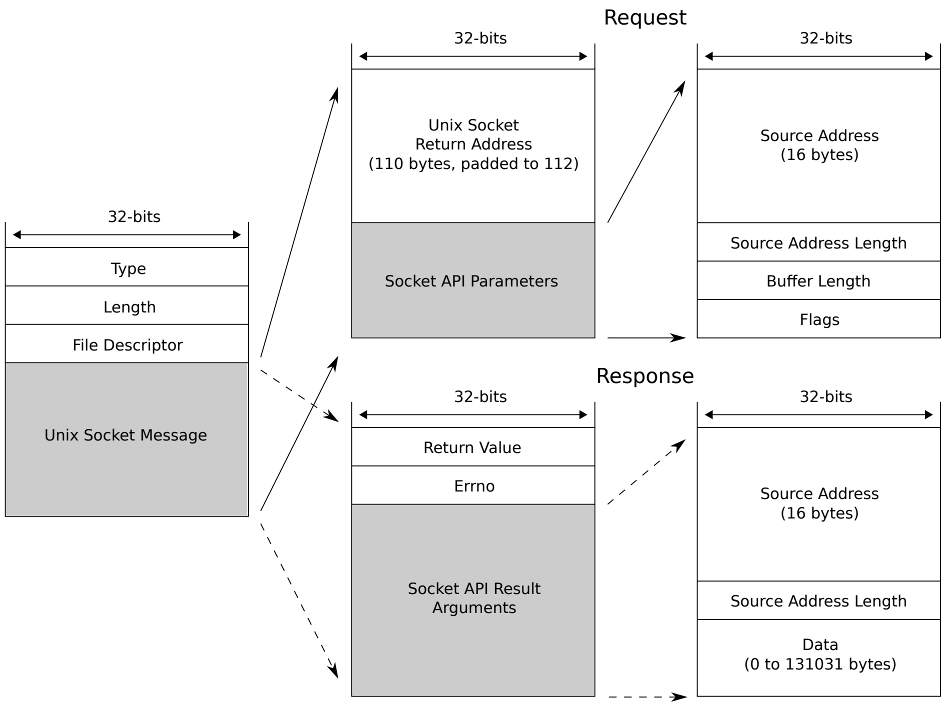


Figure 4.2: Unix Socket Message Format

contain the address of the Unix socket specific to that instance of the front-end static library. This is to ensure the back-end knows where to send the response; the back-end's Unix socket is well-known so this information is not necessary in response messages. All response messages contain a return value and `errno`. The right-most column shows the parameters unique to the `recv()` call and their placement within the message. The source address and associated length are value-result type arguments; they are used in both messages.

## 4.3 Back-end

We provide a common framework in which all transport protocols reside: the back-end. The back-end provides a rich development area, similar to a simulator, for building and testing transport layer protocols in user-space. Running as a daemon, it works with the front-end to provide a fast, flexible, and scalable solution to building and decomposing transport protocols.

API socket calls, represented as messages, are received from the front-end library via a Unix socket and processed in the back-end. When the back-end has finished processing each message, a response is returned to the front-end over a Unix socket. Each response contains the necessary data to return from the socket API call such as a return value and received data.

The back end utilizes an event-driven architecture to ensure high performance, flexibility, and scalability. Furthermore, we decompose the back-end into modules, each with a producer-consumer queue for processing events. At the core of the back-end is an event dispatcher that manages which modules receive which events. To support protocol development, the back-end includes timers, network communications, socket management, packet logging, and memory management. Our extensive use of design patterns [24, 40] helps to simplify development.

### 4.3.1 Event-driven Architecture

The back-end utilizes two types of events: low-priority *framework* events and high-priority *protocol* events. First, *framework* events are used for notifications among the various modules. Examples of when *framework* events are created include:

- A socket API message is received
- A timer expires
- A packet is received from the network

Second, *protocol* events are used for intra-protocol communication and are generally created within a protocol module as the result of receiving a *framework* event. Decomposing protocols into swappable components creates a problem when components need to communicate. If we make direct calls from one component to another, we introduce fine-grained dependencies. We avoid this by reusing the event infrastructure to have a common interface

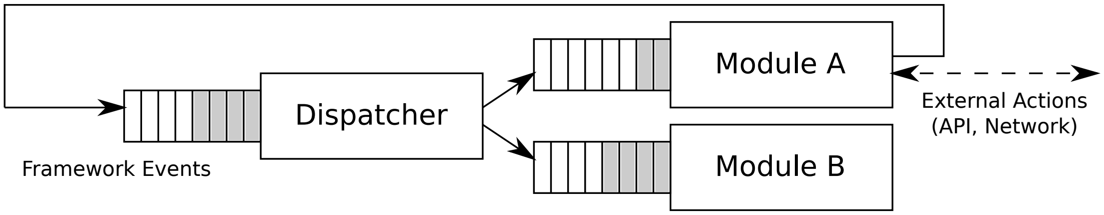


Figure 4.3: Dispatcher Event Processing

for communication between arbitrary components. Examples of when *protocol* events are created include when:

- A buffer is not empty
- A buffer is not full
- A packet is being resent
- A connection has been established

Events are identified by their pointer address. Some events, such as those that represent a socket API call, are stored in pre-allocated object pools to avoid expensive calls to `new` and `delete`. All events contain a pointer to the socket they belong to for correlation. This correlation ensures that each event is specific to one instance of a specific transport protocol. Events may be extended or added as needed to ensure future needs are met.

Events are processed in two places. First, the dispatcher provides a mechanism to organize the processing of events. All *framework* events pass through the dispatcher and are sent to modules. Since one event may need to go to many modules, the dispatcher architecture ensures that we avoid modifications to existing modules as we add other modules to the framework. At startup, each module registers with the dispatcher for the *framework* event(s) it would like to receive. Furthermore, modules should only register for *framework* events; *protocol* events are to be handled internally by the protocols.

Figure 4.3 shows the relationship between the dispatcher and modules. Events are processed in a strict FIFO order at the dispatcher and are sent to modules in the order that they registered for them. When a module wishes to send an event to one or many other modules, it simply enqueues it to the dispatcher. When the dispatcher dequeues an event, it iterates over each module that registered for that type of event and enqueues the event into each module's queue.

Second, individual modules receive *framework* events and in some cases *protocol* events in their queue. We use the command pattern [24, 40], where events are commands, to help in processing events at the module level. Once an event has been dequeued by a module, the event is passed to a callback function on the module's base class. The module callback function calls an `execute()` function on the event which, in turn, calls a correlated function on the module, passing the event as a parameter. This double dispatch architecture moves the decision of which function in the module the event should be passed from the callback function to the event itself. Encapsulating the module callback inside the event ensures that the base code that handles event dequeuing is left untouched when future events are added.

### 4.3.2 Transport Protocols and Decomposition

The main purpose of WiFu Transport is to support the quick development of transport protocols in user-space. Due to the unique needs of wireless networks, we need to be able to experiment with non-traditional solutions. To meet these needs, we provide a flexible architecture for implementing transport protocols.

#### The `Protocol` Class

We provide a well defined interface and a base `Protocol` class that developers use to implement all protocols. Figure 4.4 shows the interaction between a protocol, its components, and *framework* and *protocol* events. The `Protocol` class is a specific type of module that provides a priority, rather than a FIFO queue, so it can correctly handle both *framework* and *protocol* events. This design ensures that the state of a protocol can be completely updated before processing another *framework* event. Furthermore, the `Protocol` class provides

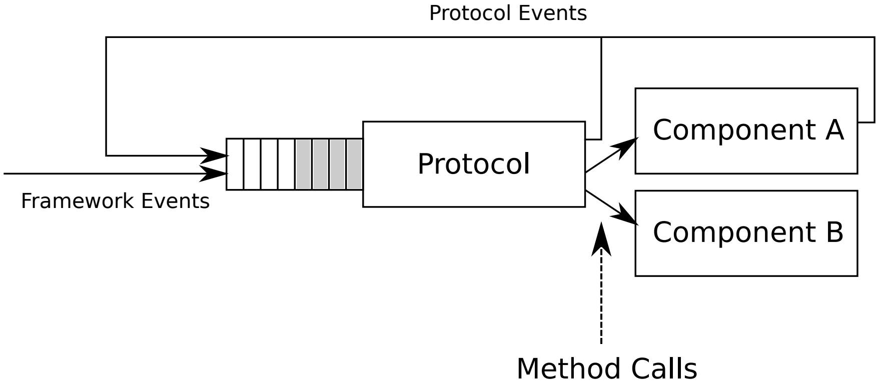


Figure 4.4: Protocol Event Processing

common functionality to all protocols such as event filtering, error checking, and delegation of events to a specific type of protocol (e.g. TCP). This delegation occurs through a well defined interface that all child `Protocol` classes must implement. Of course, if a protocol needs to deviate from this base functionality, it may simply override necessary methods.

### Decomposition

To further facilitate protocol development, we provide the ability to decompose each protocol into components. This decomposition is derived from Click [25] and is similar to CTP [9]. By decomposing protocols into components, combined with the event-driven architecture, WiFu Tranport can flexibly support many types of transport protocols, especially hybrids. Creating hybrids is particularly important because many new protocol specifications make simplifications and do not consider a complete protocol. For example, forward error correction is a reliability mechanism that, in order to be fully tested, would need to be combined with connection management and congestion control. Using the WiFu Transport protocol decomposition scheme, we can take partial protocols and merge them with other components for testing on a deployed testbed.

As part of this thesis we provide three components. Other components may be added or current components may be extended or overridden to provide custom functionality. The three components together implement base TCP Tahoe functionality.

- The **connection manager** component handles set-up and tear-down of a connection using TCP SYN, FIN, and ACK segments. This component also keeps track of the current state of connection (e.g. connected or not) and notifies the protocol via events when a state transition is about to occur (e.g. when state changes from not connected to connected).
- The **reliability** component ensures that all data arrives to the application in order and without duplication. This component uses cumulative ACKs, timers, and implements the fast retransmit algorithm. Furthermore, this component *enforces* flow control on any receiving host to ensure that the receive buffer does not overflow.
- The **congestion control** component manages the rate at which data is sent. Data transmission originates from this component and is subject to the slow start and congestion avoidance algorithms. In addition, this component *ensures* flow control by sending up to the minimum of the congestion and receive windows.

We developed these three components as we dissected the TCP Tahoe specifications with an emphasis on compartmentalizing functionality. Chapter 5 discusses our experiences developing WiFu TCP, a decomposed version of TCP.

### State Pattern

In order to further promote flexible, reusable, and understandable code, we utilize several design patterns [24, 40] in the implementation of WiFu Transport. The state pattern is particularly important for protocol decomposition because transport protocols are naturally state machines. For example, TCP has two main finite state machines. The first is for connection management (see [33] page 23). The second is in congestion control and only has two states: slow start and congestion avoidance. With these examples as a base, we implement each component using the state pattern. This allows developers to make small changes to a single state or how transitions work. Conversely, developers can add or remove states to make bigger changes as needed. Complete control is given to the developer to make changes at the protocol or component level by swapping components and at a smaller level, within individual components, by extending, adding, or removing states and associated transitions.

### 4.3.3 Timeout Management

Many transport protocols use timers. In addition, we have found the need for timers in other parts of the back-end, such as packet logging and the mock network module, which is used for unit testing transport protocol behavior. To support these needs, we provide a centralized system for managing timeouts consisting of three events and a timeout manager. The three types of events are: `TimeoutEvent`, `TimerFiredEvent`, and `CancelTimeoutEvent`. When a module wishes to set a timer, it creates a new `TimeoutEvent` and passes it the number of seconds and nanoseconds from now it would like to be notified, then sends it to the dispatcher. The timeout manager uses a priority queue to order `TimeoutEvent` pointers according to their expiration time, earliest first. Expiration of a timer is determined by using semaphores and the `sem_timedwait()` function. We use semaphores because it allows us to set an absolute future time and we can exit the function prior to timeout by calling `sem_post()`. This is used in case the timeout manager receives an `TimeoutEvent` that needs to fire prior to the current one. Once a timer expires, the timeout manager sends a `TimerFiredEvent`, which includes a pointer to the original `TimeoutEvent`, to the dispatcher. Including the original pointer ensures that the timer that expired can be identified by the module that created it. Modules may also cancel a timer by sending a `CancelTimeoutEvent`, which also includes the original `TimeoutEvent` pointer, to the dispatcher.

### 4.3.4 Network Communications

Communication between the network and WiFu Transport is done via a high-speed network module that uses raw sockets and the Internet Protocol (IP). Using raw sockets and the IP layer avoids the overhead, such as duplication of header fields, of running on top of other protocols such as UDP. Two event types are used in conjunction with the network module: `NetworkSendPacketEvent` and `NetworkReceievePacketEvent`. The network module registers for `NetworkSendPacketEvent` objects at startup. Likewise, all transport protocols register for `NetworkReceievePacketEvent` objects. Part of this registration includes specifying the protocol version, as specified in the IP header, and a packet factory to generate the exact type of packet object used with this protocol. The network manager associates each registered protocol with the packet factory to be used when receiving packets.

A `NetworkSendPacketEvent` is created when the transport protocol needs to send data to the network. A pointer to a `WiFuPacket` is assigned to this event and then the protocol sends it to the dispatcher. This packet type enforces the use of the IP header, as defined by the kernel, and source and destination ports as the first fields in the transport protocol header. This is the same structure used by UDP and TCP and is necessary for demultiplexing received packets. These ports are local to WiFu Transport and do not map to ports used by the kernel. When the network module receives the send event, the packet may be logged and then is physically sent over the raw socket.

The network module uses the `epoll` system calls to be notified that a packet is ready to be read from the kernel. Part of the `epoll` system calls includes knowing the file descriptor to read from. This file descriptor is directly associated with a protocol number; this allows us to use the packet factory for that protocol. Upon receipt of an IP packet, the network module gets a packet from the previously specified packet factory and calls `recv()`, passing the protocol-specific `WiFuPacket`'s buffer to read data into. The IP checksum is verified, the packet may be logged, and then it is demultiplexed to find the associated socket. Demultiplexing is first tried on the traditional four-tuple of source and destination IP addresses and ports. We do not include the protocol number because ports are unique across all protocols within WiFu Transport. If a match is found, the packet is destined for a connected socket. If no match is found, only the destination IP address and port is used for another query. This is necessary for packets that are used in connection establishment (e.g. a SYN) and connectionless protocols such as UDP. If no match is found at this point, the packet is discarded. Once the socket is found, a pointer to the packet is inserted into a `NetworkReceievePacketEvent`, which is sent to the dispatcher. The dispatcher, in turn, sends it to the appropriate transport protocol as indicated by the socket.

To help debugging and analyzing protocol behavior in predefined scenarios, we have extended the network module to use the mock object pattern. This gives developers complete control over a mock network, where packets may be dropped randomly according to a specified percentage. Furthermore, packets may be dropped or delayed according to sequence and acknowledgement numbers. Emulating network behavior has been invaluable when unit testing and analyzing protocols for correctness.

### 4.3.5 Socket Management

We provide a `Socket` object that contains common data that the back-end may use:

- Send, receive, and resend **buffers**.
- Source and destination **IP addresses** and **ports**.
- **Socket options** for use in conjunction with `getsockopt()` and `setsockopt()`.
- Socket **domain**, **type**, and **protocol** as passed in via the `socket()` API function.
- **Integer value** of the socket. This is randomly created when a socket is instantiated. Furthermore, it is the return value of `socket()` and used by the front and back-ends for socket identification.

In the back-end, the `Socket` object pointer is an identifier to match an event with a specific transport protocol. All events require a `Socket` pointer during construction. This helps protocol modules correctly identify which events are for them. In addition, it allows the common data of a `Socket` to be available during any event processing. This is extremely convenient when processing events in any transport layer protocol module and components, as the necessary buffers and connection information are always available.

A pointer to each `Socket` object is stored in a singleton `Socket` collection. Because all events require the `Socket` pointer which they are associated with, we must be able to quickly locate the `Socket` based on either the integer value or source and destination IP addresses and ports. Our implementation uses a map with the source and destination IP addresses and ports as the key and the `Socket` pointer as the value. Because we use the standard C++ STL map, given the key, lookup is done in O(log(n)), where n is the number of key-value pairs. When looking up a `Socket` based on integer, the implementation iterates through the map until it finds the correct value; this is O(n). We reason that (1) more than a handful of sockets open at any given time is unlikely, so O(n) is not too bad, and (2) the most frequently used lookup method is likely via addresses and ports when receiving a packet, due to the fact that data sent via the `send()` and `recv()` API calls can handle much more data than the maximum transmission unit size. In other words, over the lifetime of a connection, WiFu Transport likely handles many more incoming packets than API calls.

### 4.3.6 Packet Logging

Packet logging is important in the evaluation of transport layer protocol correctness and behavior. Correctness can be validated by unit and integration testing, while behavior can be examined by data visualization. We provide a basic packet logger, implemented as a singleton, that outputs a standard PCAP-formatted log file.

Visualizing packet traces is key in understanding the behavior of protocols. Wireshark [11] provides a rich environment for viewing packet traces, graphs, and other useful debugging and visualization tools. We have modified a version of Wireshark so that TCP-variant protocols, implemented in WiFu, may be loaded and visualized even though their IP protocol field is non-TCP. Modification of Wireshark for non-TCP variants is future work.

Furthermore, we use the packet logs when testing protocols to validate correctness under different, controlled scenarios. Testing is quite simple; we provide an expected trace of packets and compare it against the actual packet log. We are able to compare most fields in both the IP and TCP header to ensure the correct order of packets.

Writing to disk is an expensive operation, especially as we consider transport protocols and their oft-times large traces with relatively small packets. We do not want to log each packet to disk as is is sent or received. Therefore, we provide command line options to control how often the logger logs packets. The two options put a cap on how many packets to save and a timeout value before logging, whichever comes first. All packets in memory are also logged when WiFu Transport exits.

### 4.3.7 Memory Management

Our choice of using C++ and an event-driven architecture poses the memory management problem of knowing when an event is finished and may be deleted. This issue may arise if many modules receive the same event and one or more of them hold on to the event for an arbitrary time. Furthermore, in our multi-threaded solution, the same event pointer may exist in multiple threads.

We choose to use the Boehm Garbage Collector [7] to solve this problem. The Boehm Garbage Collector uses a mark-sweep algorithm. Many other solutions exist to solve this problem, including reference counting and smart pointers. Our choice of memory management provides additional benefits including support for (1) object inheritance, (2) STL containers, and (3) the `pthread` library. The garbage collector also has an option we use that avoids calling the destructor on managed objects.

Other considerations in memory management include static objects and object pools. Garbage collection does not apply to many objects in WiFu Transport that are implemented as singletons in static space. This aides us in measuring performance by being able to save measurements in a collection and then write them to disk upon destruction (when the application exits). We also use object pools for API socket calls because there is generally a one-to-one relationship between an event triggered by an API call and the event in response to it. In other words, we know generally when we may reuse the API call request and associated response events.

### 4.3.8 Design Patterns

We use design patterns [24, 40] to foster flexible, reusable, and understandable code. Using design patterns ensures that *how* we are solving problems will be easily understood by other developers. It is much easier to explain that a class is a singleton or that these classes represent the state pattern than meticulously going over how the individual classes operate first individually and second collectively. To illustrate this, consider that design patterns are used in many other disciplines. For example, if we were to say that an object is a car, we would all know that the object likely has wheels, an engine, steering mechanism, etc. *and* where these parts are located. The absence of description shows common understanding, which facilitates code reuse.

Some of the design patterns we use are state, command, object pool, visitor, observer, factory, singleton, and mock object. We further detail these design patterns and their application in WiFu Transport.

- We use the **state** pattern to make it easy to modify an existing protocol and to decompose protocols into smaller pieces.
- The **command** pattern is used to ensure extensibility when adding new events and to help in our double dispatch event processing.
- **Object pools** are used in the back-end when an API call message is received and an event is needed.

- The **visitor** pattern is used to separate the code that iterates over a collection from the code that operates on each element. We use this extensively in conjunction with the `Socket` collection.

- Using the **observer** pattern, we are able to automatically update changes to the `Socket` collection when a `Socket` object's key changes, requiring a remapping of the key and value in the map. This occurs when a connection is being established (e.g. receipt of a SYN segment) and we need to add the remote IP address and port to the `Socket` object.

- We use the **factory** pattern to generate protocol-specific packets that we can use when receiving data from the network.

- The **singleton** pattern is used extensively where we only need one instance of the object. All modules are singletons.

- To facilitate protocol validation, we use the **mock object** pattern in our implementation of the network module. We provide two concrete network module objects that have the same interface: one that uses the real network, and another that provides complete control over packets. The mock object also aids debugging.

# Chapter 5

## Decomposing TCP Tahoe

Our research demonstrates that TCP can be decomposed into components, which has been held by network researchers as a common belief. This work addresses that belief directly by compartmentalizing connection management, reliability, and congestion control into separate components that, when combined, form a working TCP Tahoe implementation. To the best of our knowledge, this work is the first to implement a nearly-complete decomposed TCP that maintains TCP fidelity. This section describes our experiences and challenges in this achievement.

By implementing TCP Tahoe we show that we can use the WiFu Transport framework to house protocols. Furthermore, pulling apart the TCP Tahoe implementation into components provides a base that other protocols may use. This is particularly important for transport layer solutions that do not consider a complete protocol (e.g. only provide a congestion control solution). An unintended byproduct of pulling apart the Tahoe specifications and compartmentalizing them has been the creation of a methodology for implementing protocols in WiFu Transport. We also use our experience of building Tahoe in user-space as swappable components as a way to further refine the framework.

Our goals of flexibility and code reuse are the reason why our implementation of Tahoe decomposes the protocol into four major components: connection management, reliability, congestion control, and a child class of `Protocol`. The connection management component handles connection set-up and tear-down while the reliability component ensures that all data arrives to the application in order. Congestion control determines the speed at which data is sent. The fourth component is a child class of `Protocol`, named `TCPTahoe` that composes the previous three. The `TCPTahoe` class determines which, if any, of the composed components to delegate events to. This means that the `TCPTahoe` class includes some generic event and packet validation and processing. For example, it checks acknowledgement and sequence numbers in a received packet by asking the reliability manager for the necessary context to make such a decision and only further processes the packet if it is valid.

We focus on the major functionality of TCP Tahoe in our implementation. This lets us omit several concepts from the RFCs that we feel are not essential in developing new transport protocols. For example, we do not implement the functionality behind the RST, PSH, and URG control bits, even though they exist in the header (this can be added later, if needed or desired). Instead, we concentrate on the major concepts such as connection management (SYN, ACK, and FIN control bits), reliability (sequence and acknowledgement numbers, and ACK control bit), and congestion control (sequence and acknowledgement numbers, and ACK control bit). Our implementation supports the time stamp option [20], slow start, congestion avoidance, fast retransmit, and duplicate ACKs.

Composing, understanding, compartmentalizing, and implementing the various RFCs has been an exciting challenge. We believe that we have successfully navigated the RFCs and created an implementation that represents a correct implementation of TCP Tahoe.

The first issue faced when decomposing TCP Tahoe is analyzing the RFCs and coming to an understanding of what exactly is meant by TCP Tahoe. Surprisingly, there is no clear-cut manual or specification for TCP Tahoe, only a collection of RFCs [33, 41, 8, 3, 32, 10, 20, 30, 34] that propose various pieces and improvements to TCP. We have taken these RFCs and composed a working specification for what we understand to be TCP Tahoe. Understanding what portion of an RFC is Tahoe versus a later version (such as Reno) is clear in certain cases but not in others. For example, RFC 2001 [41] defines the TCP slow start, congestion avoidance, fast retransmit, and fast recovery algorithms and is the primary reference we have for each of these algorithms; TCP Tahoe implements the first three algorithms, but does not implement fast recovery [26]. Furthermore, to make things more confusing, RFC 2001 is obsoleted by RFC 2581 [3]. We are left on our own to determine what we should use from the latter RFC. This is especially troubling as we try to learn what to set the slow start threshold to after loss has been detected. RFC 2001 indicates that “one-half of the current window size (the minimum of [the congestion window] and the receiver’s advertised window, but at least two segments) is saved in [the slow start threshold variable].” However, RFC 2851 specifies “[w]hen a TCP sender detects segment loss using the retransmission timer, the [slow start threshold variable must] be set to no more than [the maximum of one-half the flight size and two maximum segment sizes (MSS)],” where flight size is the amount of outstanding data. Kurose [26], in Section 3.7, incompletely states that the slow start threshold be set to “half of the value of the congestion window.” There are many more examples, but we omit them for brevity, that illustrate the fact that knowing what to support and what not to support from the numerous RFCs is a challenging task.

The second issue faced is knowing how to separate TCP logic into components. This proved to be an interesting task because the RFCs have little to no notion of decomposition in them. This is especially challenging as we consider TCP variables and where they should be located. Throughout the WiFu Transport project, we have maintained a journal that contains the many questions we have had. We share a small sampling of these questions and issues and our solutions to them below.

### 5.1 Which component should send data?

Deciding which component should decide when to send data is tricky because the three defined components do not really include any notion of sending data. They merely handle data or manage how it may be sent. The connection manager clearly does not make sense; a connection just needs to be in the `ESTABLISHED` state in order to send data. The reliability manager needs to know that data is being sent so that it may set timers and update its internal state to account for the outgoing data. It could make sense to send data from reliability for this reason. However, reliability has no notion of *how much* data may be sent; this information is known only to the congestion control component. We decided to put the function of sending of data in the congestion control component as it is the one that regulates how fast data may be sent. The reliability module is notified by an event so that it may set a timer, if necessary, and update its internal state to account for the new unacknowledged data.

## 5.2 Which component should resend data?

In TCP, there are two reasons why data needs to be resent: (1) a timer fires or (2) three duplicate acknowledgements are received. It makes sense that the reliability component handle both as it is the one that set the timer and handles acknowledgements. But should it be the one that resends data? The congestion control component is the one that originally sent the data, so it would make sense for it to do it. Furthermore, both components may need to update their internal state upon a detected loss.

To reconcile these problems, the reliability module handles detecting loss, either by a timer expiring or receiving three duplicate ACK segments, and enqueues a resend event. The congestion control module then processes this event by setting its state accordingly and then resends data according to its sending policy.

## 5.3 Which component should check whether a received packet is valid?

TCP determines whether a received packet is valid by ensuring that it contains data within the receive window or that it acknowledges something sent but not yet acknowledged. If a packet is determined to be incorrect, an ACK segment is sent so that the TCP entity that sent the invalid ACK segment may know what the other TCP entity is expecting.

As we separated TCP into multiple components, we found that there are many cases when we would pull apart logic within a packet validation conditional statement. The question then arises: which component should perform this validation? Furthermore, all components need to have consensus as to what the correct reaction is to each packet.

For example, the TCPTahoe class should not determine that a packet is invalid according one component and preclude other components from processing it. The congestion control component determines that duplicate acknowledgements are invalid and does not process them. However, the TCPTahoe class should still allow the other components to process these packets because the reliability component absolutely needs to know about duplicate acknowledgements. An extreme solution would be to allow all components to process every packet. This would waste processing time. We need a solution that processes exactly the right packets without including packets that would be ignored by all components.

Our solution is to make the most generic validation checks as soon as possible in the TCPTahoe class. After the TCPTahoe class determines general validity, it delegates the event containing the packet to the components. Components are responsible for further checking more specific validity criteria on their own. This means that we may duplicate validation checks multiple times within the TCP implementation, but each component remains independent in doing so.

To continue the duplicate acknowledgement example, the TCPTahoe class ensures that all received packets' ACK numbers meet the condition:

$$\mathtt{SND.UNA} \leq \mathtt{SEG.ACK} \leq \mathtt{SND.MAX},$$

where SND.UNA is the first byte of unacknowledged data, SEG.ACK is the segment's acknowledgement number and SND.MAX is the maximum SND.NXT ever set. SND.NXT is the next data byte to be sent. This ensures that the segment either (1) acknowledges data sent but not yet acknowledged or (2) is a duplicate acknowledgement. If this condition fails, the TCPTahoe class delegates the event to the reliability component only to handle the invalid acknowledgement. If this condition and other packet validation checks pass, all components receive the event containing the packet.

The reliability component classifies the packet as a duplicate if the following condition is met:

$$\texttt{SEG.ACK} \leq \texttt{SND.UNA} < \texttt{SND.NXT}.$$

This captures any packet with an acknowledgement number that has already been seen before and ensures that there is data outstanding. The congestion control component needs to know about valid, non-duplicate acknowledgements, so that it may update, for example, the congestion window. It performs the following check, which is more constrained than the check performed by TCPTahoe:

$$\texttt{SND.UNA} < \texttt{SEG.ACK} \leq \texttt{SND.MAX}.$$

If SEG.ACK is equal to SND.UNA TCP does not update the congestion window. Through this example we have demonstrated how we do some generic packet validation at a high level to avoid processing unnecessary packets, and more specific packet validation at the component level so that each component may update their state appropriately.

### 5.4 Where does the receive window variable go?

The receive window is the amount of data that a TCP socket is willing to receive and is used to provide flow control. Flow control ensures that the sender does not send more data than the receiver of the data is willing to buffer before sending to the application. Each packet contains this variable in the TCP header [33] so any TCP sender will not overwhelm the other side of the connection.

These facts imply that whatever component is responsible for processing received data should update this variable. However, according to RFC 793 [33] page 69, TCP needs to know the value of the receive window in order to validate incoming segments. Furthermore, when TCP receives a packet it must store the receive window contained in the TCP header and send no more than this amount of data. This is called the send window and works in conjunction with the congestion window as the maximum amount of data that may be sent. Thus, the receive window is used in validating incoming segments, receiving data segments, and sending data segments.

Because the reliability component processes received data, ensuring that it has non-duplicate and in-order data to send to the application, it keeps track of the receive window variable. When TCP sends data, the reliability component is already filling other information into the TCP header, such as sequence and acknowledgement numbers, so it makes sense for it to also fill in the receive window variable. The congestion control component updates its send window to the value of the receive window in the TCP header for each valid acknowledgement it processes. The `TCPTahoe` class composes the other components and can query the reliability component for the current value of the receive window for generic validation purposes. Other packet validation, that does not need the value of the receive window, is done at the component level.

### 5.5 How do independent components share state?

There are elements of TCP that have dual purposes. For example, send buffer variables are used both for reliability and congestion control purposes. This results in questions regarding where to save state.

The `SND.NXT` and `SND.UNA` variables typify this problem. The `SND.NXT` variable represents the next byte to be sent and the `SND.UNA` variable represents the first byte of unacknowledged data. These variables are used in conjunction with the send buffer that resides in the `Socket` object. It does not make sense to put these TCP-specific variables in the generic `Socket` object as that would be too specialized.

One possible solution would be to create a `TCPSocket` subclass of `Socket` and store information there. There are potentially two downsides to this solution. First, the framework needs to be able to create Socket objects. Each protocol would need to provide a factory so that the framework could create the correct type of Socket needed. Second, the number of variables a given protocol might need to keep its state is unknown. This may cause the various Socket subclasses to become large and unwieldy data holders.

Another solution would be to house the variables in one component and let the other component query and update the variables as necessary. However, this would couple the two components, preventing future protocols from using the dependent component without the other. We need a solution that allows for independent components with shared state to house this type of TCP-specific information.

Our solution is to ensure that all components have their own copy of any information it needs to function. Common variables may be composed into common base classes. Variable state may be updated via events. For example, SND.NXT is incremented by the amount of data in each outgoing packet. Any component needing to update the SND.NXT variable may do so by either incrementing it itself, if it is the component tasked with sending data, or by getting this information from the send packet event, which contains a pointer to the packet to be sent. This solution ensures component independence and reuses the event architecture so components may update their internal state.

## 5.6 Which component should be responsible for sending ACK segments?

The connection manager and the congestion control components both send segments that need to be acknowledged. Furthermore, the reliability component ensures data arrives to the application in order. Therefore, the reliability component should be responsible for telling the sender that data has been successfully received. This is reflected in our implementation. After the congestion control component sends data, the reliability component updates its state to take into account this new outgoing and unacknowledged data.

However, dealing with the SYN and FIN segments created by the connection manager component is less clear. Part of connection set-up and tear-down includes processing ACK segments. The question then arises: is acknowledging a SYN or FIN segment the same as acknowledging data, or are these ACK segments more transitional, signaling a TCP to move from one state to another (e.g. from `SYN RECEIVED` to `ESTABLISHED` in the three-way opening handshake)?

The answer is both for our decomposed TCP. When the connection manager sends a packet containing a SYN or FIN segment, the reliability component processes the event and keeps track of the fact that a SYN or FIN has been sent and sets any timers needed. However, because acknowledgements play a key part in transitioning between states, it is the connection manager that is responsible to send ACK segments for all connection manager sent segments. This also alleviates any issues with sending a SYN-ACK segment.

The matter is further complicated when we consider that the final ACK segment in the three-way opening handshake may contain data. However, to make things simple, and to continue to ensure as much independence amongst components as possible, we do not allow data to be sent in this ACK segment.

In summary, the congestion control component sends all data. The reliability component sends and receives acknowledgments for the data. The connection manager sends and receives all ACK segments that acknowledge SYN and FIN segments. The reliability component also keeps track of connection control bits and sets any associated timers. When an ACK for a SYN or FIN arrives, the reliability component ensures proper cancellation of any timers.

It is clear that there is some interdependence between components when sending ACK segments. It may be possible to refactor this decomposition in the future.

## 5.7 Discussion

In reality, the three steps of understanding and composing the RFCs, deciding where to put the various logic, and creating a TCP specification were performed many times as we implemented and integrated the protocol with the framework. As we tested what was in place and discovered new questions regarding how Tahoe should function, we would iterate back to the RFCs or other sources for clarification. Then we would determine the best place to put the logic, finally implement it as such, then test again. This iterative building cycle ensured that we almost always had a working version of TCP. Our confidence is high that, because we have built it in this way, that extensions or modifications to Tahoe will be able to be done easily, taking advantage of the decomposed architecture.

While our work is the first to decompose TCP into components while maintaining Tahoe fidelity, we do not state that it is the best, and we know that there is work to be done to achieve complete separation between modules. Of course, it is possible that because the specifications were not created with this decomposition in mind, that complete separation is impossible. In cases where it is clear that complete separation is not possible, we do our best to keep boundaries between the components and duplicate the variables within the components instead and ensure that the variables have the same semantics between modules.

# Chapter 6

## Using WiFu Transport to Build TCP Variants and Hybrid Protocols

This chapter outlines two WiFu developers' experiences with WiFu Transport. These experiences suggest that WiFu Transport itself, and the processes we use in creating it, provide flexible and reusable code that facilitate rapid development of transport protocols. This discussion is not intended to provide conclusive experimental results; rather, it provides a sample of what a user might expect as he works with WiFu Transport. The two WiFu developers work in the same lab as the primary developer and therefore have access to him for questions and clarifications about WiFu Transport. First we discuss a computer science graduate student's experiences using WiFu Transport to implement numerous TCP-variant protocols. In addition to implementing vanilla specifications, he created hybrid protocols by combining individual variants with the intention of finding out the feasibility of TCP on wireless mesh networks [27]. Second, we describe a computer engineering undergraduate student's experience creating a new protocol we call TP-$\alpha$.

These students helped take WiFu Transport from a one-protocol solution to a transport protocol framework that supports numerous solutions. It is difficult to generalize a framework that is intended to support many protocol solutions, such as WiFu Transport, without using it and understanding where the generalizations can and need to be made. Some generalizations can be made at a high-level. These include our base framework design decisions discussed in Chapters 3 and 4 to employ an event-driven architecture, to use modules, and to separate the socket API from the daemon, etc. These design decisions have created a rich environment for developers to implement protocols. However, these decisions do not account for protocol-specific questions, such as what logic is generic to all protocols versus what should be pushed down into an inheriting child protocol, such as `TCPTahoe`. In order to learn how we can improve our framework, we utilize these student experiences when implementing and integrating other protocols into WiFu Transport.

Because WiFu Transport's intended users are network researchers and developers, we asked other students who are not as familiar with the code to implement various protocols. This serves multiple purposes. First, we seek to provide a framework in which transport protocols can be implemented easily in user-space while maintaining high performance. Having these two students integrate protocols into WiFu Transport gives us confidence that other transport protocols can be supported similarly, even though we may not know of them or they have not been published yet. Second, we further the research in the networking community by implementing these protocols and testing them on our mesh network sooner, rather than waiting until WiFu Transport reaches some arbitrary finishing point. Third, we benefit from others as they use the code and subsequently find bugs in the framework. This is especially true of bugs that are invisible to the primary developer. Finally, developing protocols furthers other students' research.

### 6.1 Data Collection

We performed a semi-structured interview with each student. We asked each student several questions about his prior experience in programming and networking, what he had implemented, and what was his experience with WiFu Transport. In addition, we performed several informal interviews throughout the project and examined source code. These interviews and source code examination are summarized in the following sections.

### 6.2 WiFu Transport Developers

We outline the background of each participating WiFu developer. The first student used WiFu Transport and the provided TCP Tahoe implementation as a base for his master's thesis research [27] that examined many TCP variants. Prior to working with WiFu Transport, he had seven years of programming experience, numerous networking and related courses, and worked on at least one very large project outside of the classroom. This student also helped the primary developer create the very first protocol in WiFu Transport, a simple stop-and-wait protocol used only for proof-of-concept purposes. He therefore had a good mental model of the framework before he began developing protocols.

The second student created a hybrid protocol called TP-$\alpha$. This undergraduate student has been programming for four years and has taken one networking course. He had no experience with WiFu Transport before beginning to compose TP-$\alpha$; it took two to three weeks to get a mental model of the framework. This time was not all dedicated to understanding WiFu Transport; he was also trying to understand ATP [42].

### 6.3 TCP Variants

A graduate student integrated many TCP variants into WiFu Transport. This gives us a general idea of how feasible modifications to the existing components in WiFu Transport are. We give a brief description of each protocol followed by an indication of how difficult each was to implement in WiFu Transport and how long it took to implement each protocol in WiFu Transport.

- **TCP with Adaptive Pacing (TCP-AP)** [13] paces packets such that a packet is sent only after it determines the previous packet is four wireless hops away. In order to implement TCP-AP, the source collects round-trip times (RTTs) and uses them to generate an approximation of the four-hop delay (FHD). Implementing packet pacing in WiFu Transport turned out to be easy. The graduate student added another component, a rate limiter, to TCP Tahoe that computes the FHD based on RTTs and paces outgoing packets. Furthermore, he extended the base TCP Tahoe functionality to include the new rate limiter component. Overall, it took about 10 hours to implement TCP-AP.

- **TCP with Delayed ACKs** [4] aggregates acknowledgement packets to reduce load on the transmission medium. Acknowledgments are managed primarily in the reliability component of TCP Tahoe. TCP Tahoe sends an ACK segment for each data packet that it receives. To accommodate the new requirement, the graduate student extended the reliability component and overrode methods dealing with acknowledgements and timers. Only five method overrides and one method extension were necessary to complete the implementation. This protocol took between six and eight hours to implement.
- **TCP FeW** [31] is less aggressive in incrementing the TCP congestion window. Implementing this protocol is a modification to the congestion control module. The TCP Tahoe congestion control module provides two states: slow start and congestion avoidance. TCP FeW is implemented by extending both and overriding a few methods in each that correspond to setting the congestion window and new states. Even though this was conceptually easy, implementing it proved to be challenging. Furthermore, the graduate student admitted to breaking some of the existing code. The time to fix that and implement FeW took about 15 to 16 hours.

In addition to the three protocols listed above, the graduate student additionally experimented with combining these concepts and creating hybrid protocols. He developed all combinations of the three outlined above:

- Adaptive Pacing with FeW
- Adaptive Pacing with Delayed ACKs
- Few with Delayed ACKs
- Adaptive Pacing with FeW and Delayed ACKs

Assembling these four new hybrid protocols took between six and eight hours. No new protocol logic was developed; the time was spent hooking up the appropriate components. Overall, the graduate student was able to implement three protocols and create four hybrids, for a total of seven protocols to test in a relatively short amount of time. While he implemented seven protocols, due to poor performance in experiments, he does not report on all of them in his thesis.

## 6.4 TP-$\alpha$: A Transport Protocol Hybrid

TP-$\alpha$ seeks to take the promising piece of ATP [42], the rate-based congestion control algorithm, and use it with TCP Tahoe's connection manager and reliability components. Prior work in our lab has shown that ATP needs improvements to be a viable solution [43]. WiFu Transport enables us to create this hybrid due to the event-driven architecture and protocol decomposition.

The primary difference between the two protocols is the congestion control algorithm. TCP uses the congestion window, in conjunction with the receive window, to determine how much data may be sent. ATP uses a rate-based algorithm dependent on the average queuing and transmission delays at each hop in a wireless mesh network. Furthermore, ATP requires that a calculated delay be sent in the reverse direction of data packets (via acknowledgements). The undergraduate student extended the provided `TCPPacket` object to allow room in the header for this extra information. Merging these two ideas to be compatible with the rest of TCP while maintaining the integrity of the ATP algorithm proved to be difficult. For example, ATP does not explicitly describe how to handle data buffering at the receiver side. This led to questions about how to handle any buffer overflow in the reliability component. Creating hybrid protocols such as this may be a difficult task whether or not WiFu Transport is used because of inherent differences.

Our decomposed TCP implementation was extensively reused in building TP-$\alpha$. He extended the `TCPTahoe` class so base functionality could be preserved while adding the necessary ATP logic. TP-$\alpha$'s rate-based congestion controller extends the `TCPTahoe` congestion control component to reuse some of the code such as the `SND.NXT` and `SND.UNA` variables.

TCPTahoe accepts an IContextContainerFactory as an argument. The purpose of this object is to create the necessary components to be used by the protocol. TCPTahoe's IContextContainerFactory implementation creates a TCPTahoeContextContainer that contains the three Tahoe components. This architecture provides an easy way for components to be replaced or added. TP-$\alpha$ implements its own IContextContainer factory that returns an extension of the TCPTahoeContextContainer that simply replaces the congestion controller.

The TP-$\alpha$ class that extends the TCPTahoe class overrides two main functions that deal with sending and receiving packets. The main reason for the overrides is to use a TP-$\alpha$ packet instead of a TCPPacket. Specifically, TCPTahoe's reliability component sends acknowledgements in a TCPPacket. This packet type must be converted to a TP-$\alpha$ packet when it passes through the TP-$\alpha$ class. Because acknowledgements do not contain data, only the additional header information is appended. When TCPTahoe receives a packet and its sequence number is determined to be invalid, the protocol sends an acknowledgement in response. The TP-$\alpha$ implementation changes this acknowledgement from being a TCPPacket to a TP-$\alpha$ packet. A solution to this problem would be to utilize a packet factory, similar to what is done in the network communication module. This would ensure that the correct packet type is always used regardless of where it is created, reduce code duplication, and be more generic for future protocols.

### 6.5 Improvements to WiFu Transport

There are a several benefits, modifications, and additions to WiFu Transport that result from the participation of these students. First, with the help of other students, we added many new protocols to WiFu Transport. These protocols are TCP variants and hybrids protocols, including a radically new protocol TP-$\alpha$. Second, a new component, the rate limiter, was added to the existing ones. This may prove useful in the future as we consider protocols that use rate-based instead of window-based congestion control. Third, whenever a discussion of copying and pasting code arose, we tried to generalize the code to avoid the copy and paste and make the code more reusable. This resulted in the factory to handle the creation of the components used by protocols. Finally, using WiFu Transport in our lab during development proved invaluable to finding and eliminating bugs.

# Chapter 7

## Validation of WiFu TCP

Implementation and validation of the TCP Tahoe protocol serves two purposes. The first is to ensure that we can fairly compare the performance of WiFu Transport and the Linux kernel. Secondly, we show that TCP *can* be decomposed into separate modules while maintaining fidelity.

We use our implementation of TCP to make comparisons against the kernel's implementation and NS-2's [28] implementation. To distinguish between implementations, we henceforth call the WiFu implementation of TCP Tahoe **WiFu TCP**. We call the implementation that runs in the kernel, **Kernel TCP** and NS-2's implementation **NS-2 TCP**. NS-2 TCP is the Tahoe variant while Kernel TCP is TCP Reno with delayed acknowledgements. Because many of the leading TCP researchers and implementers wrote the code for the protocols in NS-2, it is a valuable tool to use for comparison and validation of correctness. The current Linux kernel does not provide a TCP Tahoe implementation, so we aim to come as close as possible. Table 7.1 shows a listing of parameters we changed to replicate TCP Tahoe as closely as we can in the kernel.

We analyze several packet traces to show that connection management, reliability, and congestion control (slow start and congestion avoidance) are working correctly. We do not validate all possible scenarios in this thesis; rather, we show a significant amount of correctness by examining key situations. Our code base utilizes a mock network module in a testing suite that rigorously examines many aspects of TCP Tahoe under various loss

| Option | Default | Modified |
|---|---|---|
| congestion_control | cubic | reno |
| timestamps | 1 | 1 |
| window_scaling | 1 | 0 |
| sack | 1 | 0 |
| fack | 1 | 0 |
| ecn | 2 | 0 |
| dsack | 1 | 0 |
| frto | 2 | 0 |
| low_latency | 0 | 0 |
| moderate_rcvbuf | 1 | 0 |

Table 7.1: Modified Linux kernel settings

scenarios. We are confident that our TCP implementation is correct to the extent we have implemented it.

To the best of our knowledge, no one has implemented TCP by separating key functionality and compartmentalizing those parts into components and demonstrated correctness of implementation. By using WiFu TCP in performance tests, we implicitly demonstrate that TCP *can* be decomposed into components. This gives reason to believe that other protocols can be decomposed in a similar manner.

### 7.1 Short Trace Comparison

We begin by examining two short packet traces, shown in Table 7.2: one generated by WiFu TCP (top), and the other by Kernel TCP (bottom). We use the same application code to run both experiments and simply alternate between the two TCP implementations. For each experiment, we send 500 bytes of data over localhost. Each trace shows connection setup (TCP three-way handshake), the data transmission, and connection tear-down. The WiFu TCP trace data is collected via the packet logger in WiFu, while the Kernel TCP data is gathered via `tcpdump` [22]. The source and destination of packets can be distinguished by the source and destination port columns.

The three-way handshake is nearly identical in both traces. Each implementation shows a SYN segment, followed by a SYN-ACK segment that acknowledges the SYN segment

| No. | Src Port | Dest Port | Flags | Seq Num | Ack Num | Len |
|---|---|---|---|---|---|---|
| *WiFu TCP* | | | | | | |
| 1 | 142 | 5000 | SYN | 1 | 0 | 0 |
| 2 | 5000 | 142 | SYN, ACK | 1 | 2 | 0 |
| 3 | 142 | 5000 | ACK | 2 | 2 | 0 |
| 4 | 142 | 5000 | ACK | 2 | 2 | 500 |
| 5 | 5000 | 142 | ACK | 2 | 502 | 0 |
| 6 | 142 | 5000 | FIN, ACK | 502 | 2 | 0 |
| 7 | 5000 | 142 | ACK | 2 | 503 | 0 |
| 8 | 5000 | 142 | FIN, ACK | 2 | 503 | 0 |
| 9 | 142 | 5000 | ACK | 503 | 3 | 0 |
| *Kernel TCP* | | | | | | |
| 1 | 60012 | 5000 | SYN | 3868376797 | 0 | 0 |
| 2 | 5000 | 60012 | SYN, ACK | 190283240 | 3868376798 | 0 |
| 3 | 60012 | 5000 | ACK | 3868376798 | 190283241 | 0 |
| 4 | 60012 | 5000 | PSH, ACK | 3868376798 | 190283241 | 500 |
| 5 | 60012 | 5000 | FIN, ACK | 3868377298 | 190283241 | 0 |
| 6 | 5000 | 60012 | ACK | 190283241 | 3868377298 | 0 |
| 7 | 5000 | 60012 | FIN, ACK | 190283241 | 3868377299 | 0 |
| 8 | 60012 | 5000 | ACK | 3868377299 | 190283242 | 0 |

Table 7.2: Comparison of TCP Traces

by setting the ACK number to one greater than the SYN sequence number, and finally an ACK segment that acknowledges the SYN portion of the SYN-ACK setting the ACK number to one greater than the SYN-ACK sequence number. The only differences are the source port, which is selected among available ports, and the initial sequence number (ISN). The ISN begins at one in WiFu TCP for better readability but is randomly selected to minimize security risk in Kernel TCP [5]. Sequence numbers are used to indicate the first byte of data a packet contains while the acknowledgement numbers indicate the next byte expected. SYN and FIN segments count as one byte each [33].

Upon successfully establishing a connection, both implementations send one data packet consisting of 500 bytes. Kernel TCP sets the PSH flag [33] when sending the data. WiFu TCP does not support the PSH flag. Once data has been sent, the sender must receive an acknowledgement in a timely manner or it will consider the packet lost. WiFu TCP's trace shows this ACK segment next in the trace, with the acknowledgement number set to the data packet's sequence number plus the length (2 + 500 = 502), while Kernel TCP combines this acknowledgement with the acknowledgement for the subsequent FIN segment.

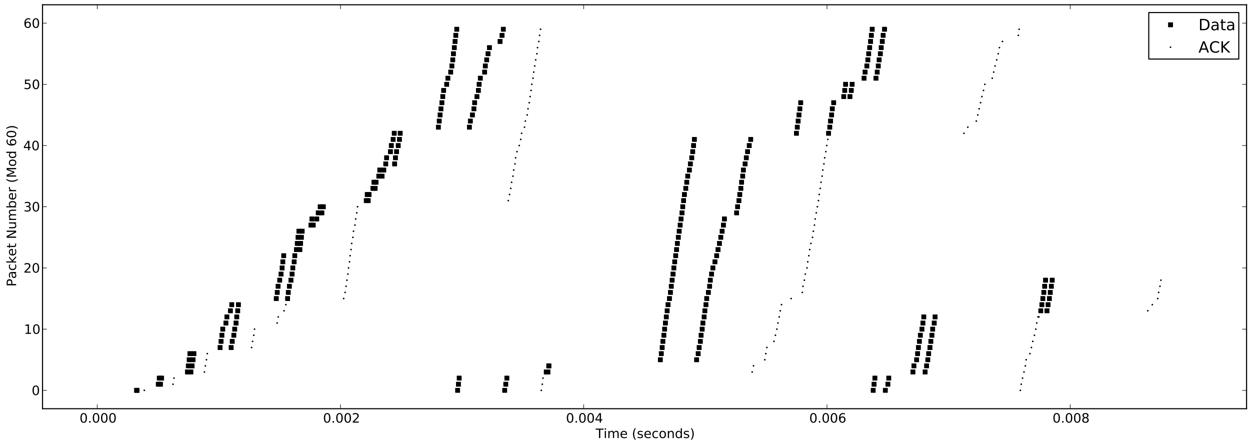


Figure 7.1: WiFu TCP Packet Trace: No Drops

This difference is attributed to the implementation of delayed acknowledgements by Kernel TCP [8]. Upon sending the data to TCP, the application issues a `close()` command to terminate the connection; this initiates connection tear-down. However, WiFu TCP waits for all data to be acknowledged before it allows the `close()` event to be processed. This clean separation reduces dependency between the reliability and connection management modules and increases flexibility for alternative modules to be plugged in. Kernel TCP does not have this requirement and thus relies on packet six to be a cumulative ACK segment, acknowledging both the data packet and the FIN segment.

To close the connection, both WiFu TCP and Kernel TCP send a FIN segment, receive an ACK segment (with acknowledgement number one greater than the sequence number of the associated FIN), receive a FIN segment, and send a final ACK segment (with acknowledgement number one greater than the sequence number of the associated FIN) .

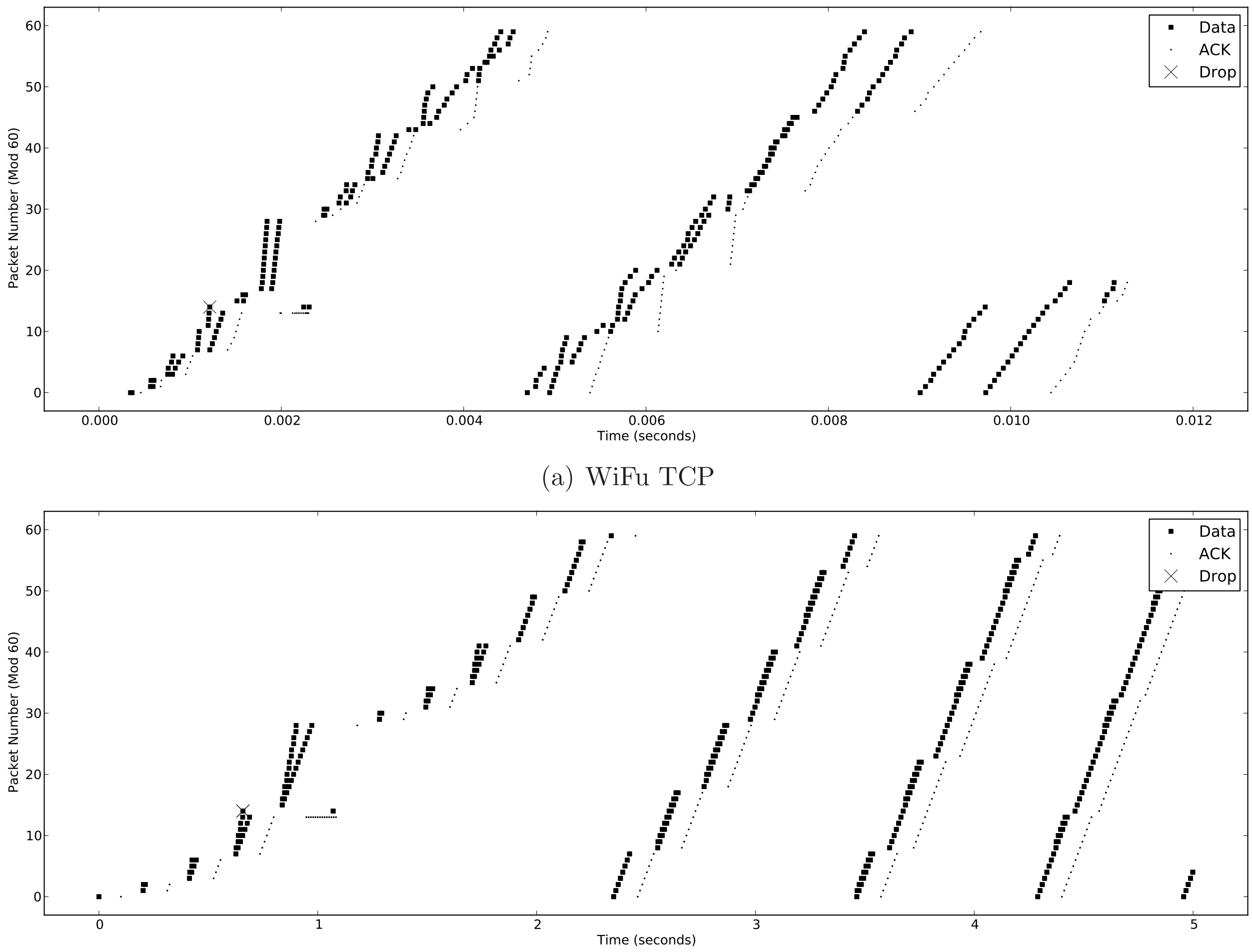
### 7.2 WiFu TCP: No Loss Scenario

We demonstrate that the slow start algorithm is working properly by examining a no-loss scenario. All WiFu TCP experiments are run using the WiFu Transport daemon with our decomposed implementation of TCP Tahoe. Unless otherwise noted we use the following settings for all TCP validation experiments. The TCP slow start threshold and the receive window are both initialized to 65535 bytes. The congestion window is initialized to one MSS equal to 1448 bytes. The receive window is a maximum value while the slow start threshold is dynamic, adjusting according to RFC 2581 [3]. Traces are gathered using the WiFu Transport packet logger and consist of both sending and receiving a packet, each indicated by a square. Acknowledgements are indicated by a single dot and indicate when the data sender received an acknowledgement for that packet. Packet traces are similar to the graphs by Fall and Floyd [15]. For clarity, we omit connection set-up and tear-down segments and associated acknowledgements from the graphs. This omission accounts for the gaps between the beginning of the traces (when time is zero) and the first packet shown as well as the gap between the end of the trace and the end of the graph.

Figure 7.1 shows a sample trace. The initial data packet is sent, received, and the acknowledgement is received, increasing the congestion window to two MSS. This exponential increase of the congestion window continues as groups of two, four, eight, 16, and 32 data packets are shown being sent, received and acknowledged. We conclude that slow start is functioning correctly. WiFu TCP switches from slow start to congestion avoidance when the congestion window exceeds the slow start threshold (65535 bytes). Upon receiving the acknowledgement of the 44th data packet, the congestion window is at 66608 bytes and transition is made to congestion avoidance. Due to the no-loss scenario and the fact that the congestion window is larger than the receive window maximum, we are governed by the receive window size and do not see the transition to or the effects of congestion avoidance. In order to see the full effects of all TCP Tahoe algorithms, we must introduce loss.

### 7.3 WiFu TCP Compared to NS-2 TCP

We introduce loss into our experiments in key scenarios that demonstrate correct implementation. This section specifically replicates loss scenarios after the simulations conducted by Fall and Floyd [15]. We use WiFu Core to drop packets as needed. If a packet is dropped, indicated by an X, the retransmission and the reception are both plotted. WiFu TCP traces



(a) WiFu TCP

(b) NS-2 TCP

Figure 7.2: WiFu TCP and NS-2 Trace Comparison: One Drop

show the same packet being sent and received, indicated by two squares at the same sequence number, while NS-2 TCP shows the same packet being enqueued and dequeued. Occasionally two squares on top of each other appear as one square. There are some dissimilarities between traces of the various implementations. NS-2 TCP traces are more smooth than WiFu TCP traces due to the exactness of a simulator and the variability when processing packets via the kernel. Furthermore, NS-2 TCP traces run for a fixed time period (approximately six seconds) while WiFu TCP traces show a fixed amount of data (200K) sent as fast as possible.

We show that sample traces of WiFu TCP are nearly identical to the traces generated by NS-2 TCP. Furthermore, we discuss the differences between the two traces and reason that, though not identical, they are both correct. We examine the one drop scenario in detail and provide graphs for the two, three, and four drop scenarios without explanation. The script used to generate the original traces in NS-2 has been deprecated; we use the suggested replacement instead, provided in NS-2 version 2.34.

Figure 7.2 demonstrates the one-drop scenario conducted by Fall and Floyd [15]. This loss occurs during slow start. We demonstrate loss in the congestion avoidance phase below. In this scenario, the congestion window should be reduced to one MSS (packet), the slow start threshold should be set to one-half of the number of bytes in flight, and TCP should begin again with slow start [3]. Furthermore, because the slow start threshold is halved, this example shows correct transition from slow start to congestion avoidance. We also examine the congestion avoidance algorithm's additive increase of the congestion window and determine that it is working properly.

The trace begins in the same manner as Figure 7.1. However, during slow start, WiFu Core purposely drops the last packet in the group of eight (packet 14). As the acknowledgements are received by the sender, the sender sends packets 15 through 28 (14 total packets). As expected, upon receiving three duplicate ACK segments (four ACK segments acknowledging the same packet), the sender reduces the congestion window to one MSS, restarts slow start, and performs fast retransmit by resending the dropped packet. However, WiFu TCP sets the slow start threshold to one-half of the number of bytes currently in flight [3] while NS-2 TCP sets it according to RFC 2001 [41]:

$$max(0.5 * min(\texttt{cwnd}, \texttt{rwnd}), 2 * MSS),$$

where `cwnd` is the congestion window and `rwnd` is the receiver's advertised window. There are 14 packets in flight when the loss is detected, so WiFu TCP's slow start threshold is set to 14 packets * MSS / 2 = 10860 bytes. NS-2's congestion window is in terms of packets, not bytes; at the time of the loss, the congestion window is 15 packets and thus reduced to 15 / 2 = 7 packets after the loss [15].

When the receiving side processes the resent packet, it sends an ACK segment for the next byte of data it expects to receive, namely, the first byte in packet 29. Upon receiving this acknowledgement, the congestion window is set to 2 * MSS = 2896 bytes and two packets are sent. When these two data packets are acknowledged, slow start continues by exponentially increasing the congestion window to 4 * MSS = 5792 bytes, etc. Slow start continues by sending packets in groups of two and then four. Up to this point both traces are identical.

In the case of WiFu TCP, when the last acknowledgement is received for the data segments in the group of four, the congestion window is increased to 11584 bytes (eight packets) which exceeds the slow start threshold. The congestion control component moves to the congestion avoidance state and continues increasing the congestion window according to the algorithm in RFC 2581 [3]:

$$\texttt{cwnd}+ = MSS * MSS/\texttt{cwnd}.$$

WiFu TCP sends eight packets, according to the current congestion window size. Upon receipt of all acknowledgements for the eight packets, the congestion window is set to 12958 bytes (8.94 packets). Due to problems sending multiple small fractional packets, WiFu TCP avoids sending partial packets, unless necessary, by limiting the amount of sent data to the floor of the congestion window. This is why another group of eight packets is sent, rather than nine, even though WiFu TCP is in the congestion avoidance phase. The trace continues by sending groups of nine, ten, etc. packets until all data is sent (the last group of data is a partial window).

NS-2 TCP handles the transition from slow start to congestion avoidance differently than WiFu TCP due to the congestion window and MSS being in terms of packets, not bytes. NS-2 TCP's congestion window is seven packets after the loss scenario and moves to congestion avoidance after receiving the ACK segment for the last data packet sent in the window of four after the loss. At this time the congestion window is set to 7.143 packets and thus only seven packets are sent in the following window. The NS-2 trace continues by increasing the congestion window nearly linearly by sending eight, nine, etc. packets.

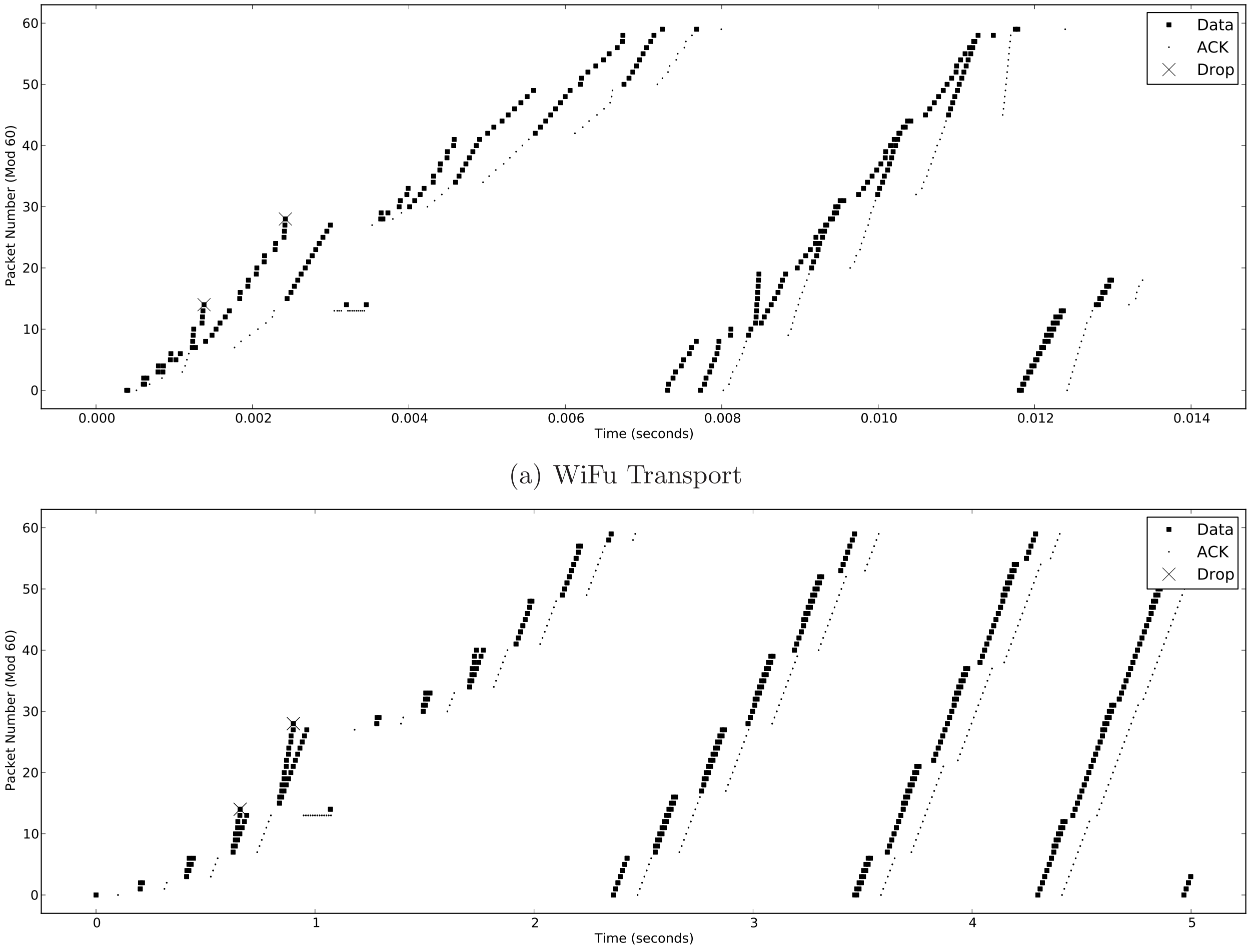
This trace validates that WiFu TCP correctly handles fast retransmit by setting the congestion window to one MSS and immediately restarting slow start. Furthermore, WiFu TCP correctly handles reducing the slow start threshold upon a loss. Because the slow start threshold is reduced significantly, this trace illustrates correct transition from slow start to congestion avoidance. The congestion avoidance algorithm is also working as the congestion window increases in a near-linear manner.

Fall and Floyd [15] did further research examining scenarios of two, three, and four specific dropped packets. To further validate our TCP Tahoe implementation, we replicate these experiments, similarly to the one-drop scenario. The results are in Figures 7.3, 7.4, and 7.5. They are nearly identical to their counterpart produced by Fall and Floyd [15] in each case. The only differences arise in the transition between slow start and congestion control after loss and dealing with the size of congestion window as explained in the one-loss scenario.

## 7.4 Further Evaluation of WiFu TCP

We further demonstrate that our WiFu TCP implementation is correct by examining loss during congestion avoidance and by inducing timeouts.

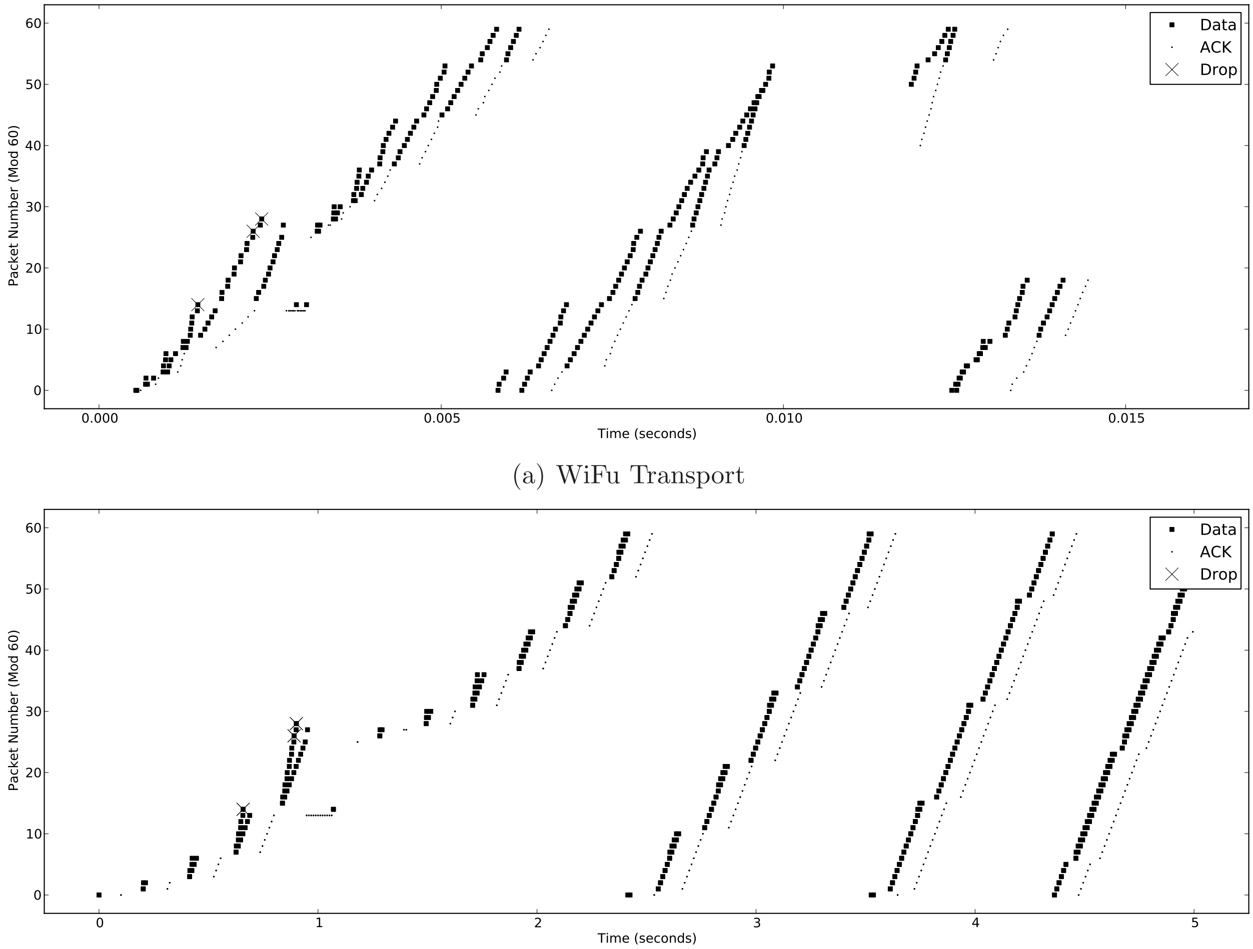
Figure 7.6 demonstrates a loss scenario while in the congestion avoidance phase. This graph also provides opportunity to examine the additive increase multiplicative decrease



(a) WiFu Transport

(b) NS-2

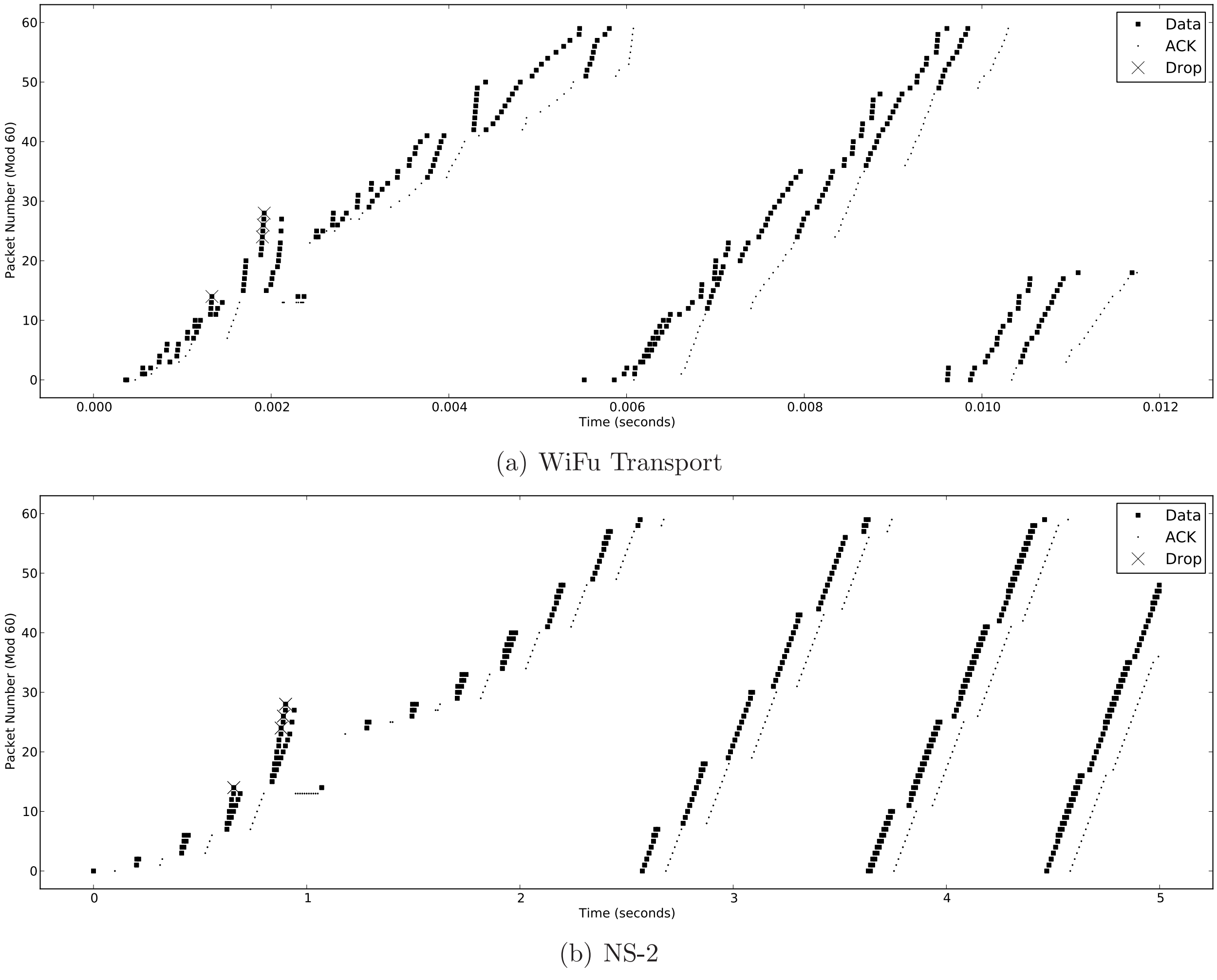
Figure 7.3: WiFu TCP and NS-2 Trace Comparison: Two Drops



(a) WiFu Transport

(b) NS-2

Figure 7.4: WiFu TCP and NS-2 Trace Comparison: Three Drops



(a) WiFu Transport

(b) NS-2

Figure 7.5: WiFu TCP and NS-2 Trace Comparison: Four Drops

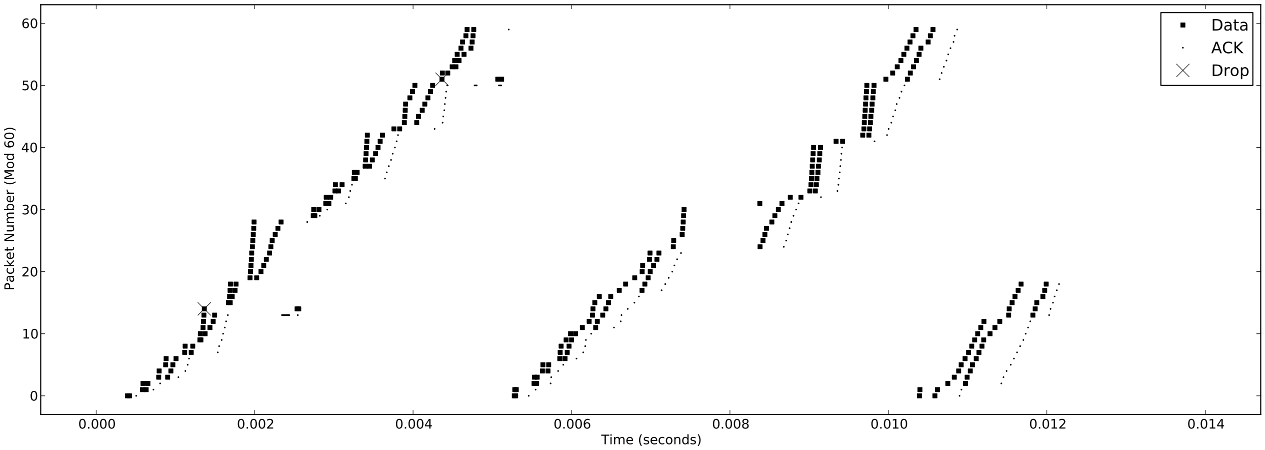


Figure 7.6: WiFu TCP Packet Trace: Loss During Congestion Avoidance

(AIMD) behavior of of the TCP congestion window. TCP Tahoe reacts to loss in both slow start and congestion avoidance in the same manner.

We drop two packets in this trace. The first packet we drop is identical to that in Figure 7.2 (packet 14). The second packet we drop is the first packet of the group of nine in the congestion avoidance phase (packet 51). WiFu TCP correctly detects both losses via duplicate acknowledgements and responds by reentering slow start and setting the congestion window to one MSS. Upon further examination of the second loss and the events that follow, we see that the slow start threshold is correctly reduced to four MSS. Before the second packet is dropped, the number bytes in flight is 9 packets * MSS = 13032 bytes. Upon receipt of three duplicate acknowledgements, the slow start threshold is cut to 13032 bytes / 2 = 6516 bytes or 4.5 packets. After receiving the acknowledgement for the first packet in the group of four, WiFu TCP sets the congestion window to 7240 bytes, which is larger than the slow start threshold, and transitions to congestion avoidance. The trace continues by sending groups of four packets, five packets, etc. to the end of the trace.

## 7.5 Reaction to Timeouts in WiFu TCP

We now explore the behavior of timeouts in our implementation of TCP Tahoe. TCP Tahoe handles timeouts in the same manner as handling detected packet loss. We show two

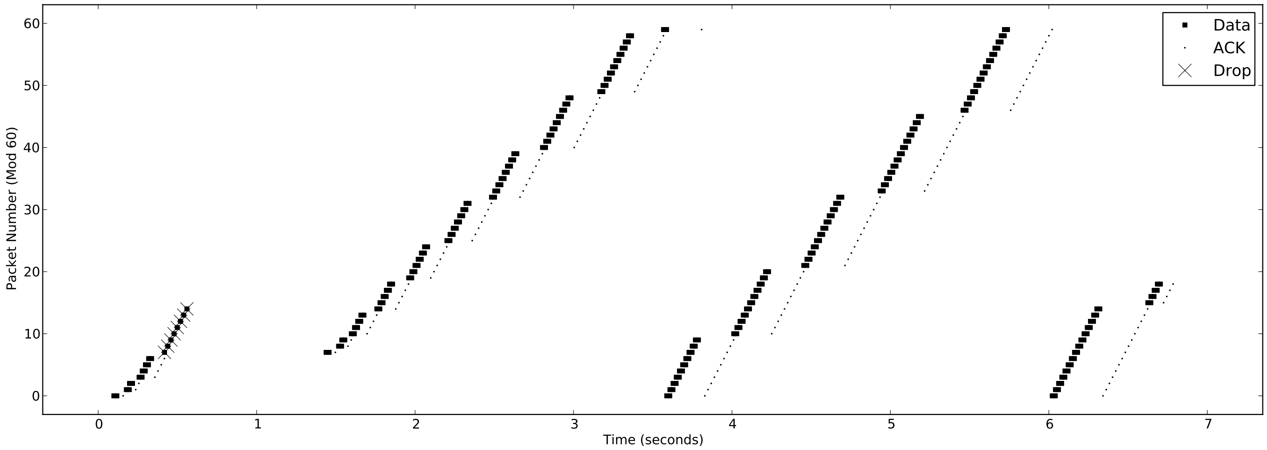


Figure 7.7: WiFu TCP Packet Trace: Timeout During Slow Start

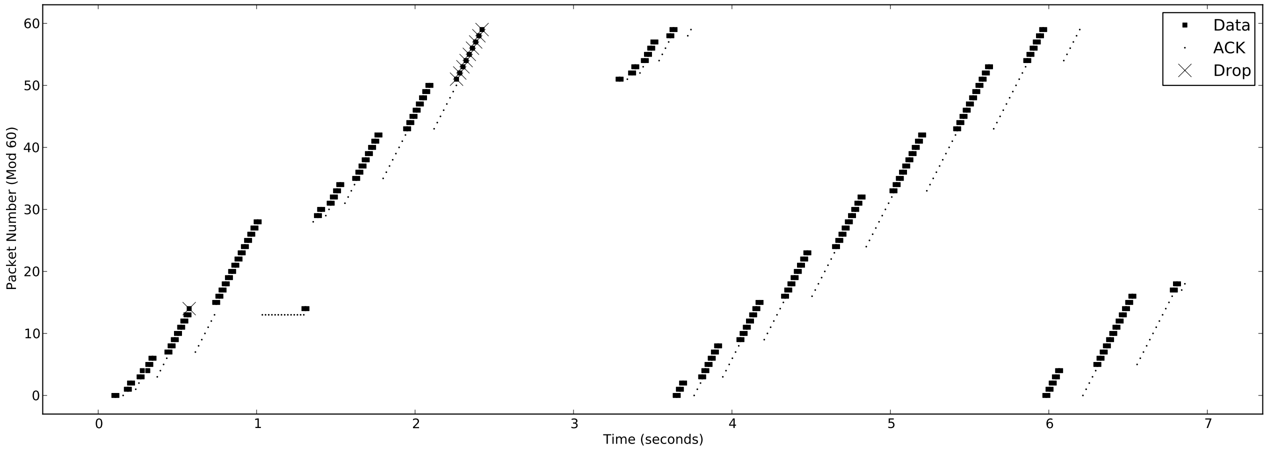


Figure 7.8: WiFu TCP Packet Trace: Timeout During Congestion Avoidance

timeout graphs. In order to better visualize the timeouts and subsequent reactions, we insert an artificial delay when sending and receiving packets. This makes the entire data transfer take much longer than before, but it is easier to see the TCP behavior in the context of a timeout. Both graphs drop an entire group of packets to ensure a timeout.

Figure 7.7 shows a timeout during slow start. We drop all packets in the first group of eight data packets. At this point the timer has been reduced to the minimum of one second [32]. The TCP timer is started for every packet except when the timer has already been started. Therefore, the timer is set for the first packet in the group of eight and goes off approximately one second later. WiFu TCP correctly handles the timeout by resetting the congestion window to one MSS, setting the slow start threshold to 8 packets in flight

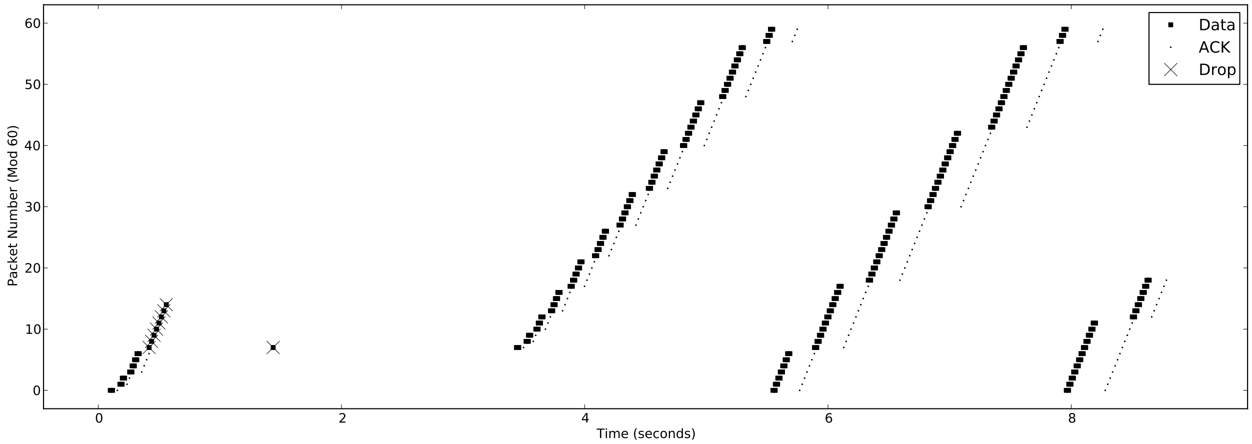


Figure 7.9: WiFu TCP Packet Trace: Two Timeouts

* MSS / 2 = 5792 bytes. WiFu TCP transitions from slow start to congestion avoidance after receiving the acknowledgement for the first data packet in the group of four. Five acknowledgements have been received at this point and the congestion window has been enlarged to 5 * MSS = 7240 bytes which exceeds the threshold. The congestion window is equal to the slow start threshold upon receiving four acknowledgements (4 * MSS = 5792 bytes) and could have transitioned at this point [3]; however, WiFu TCP chooses to wait until it is greater than the threshold, remaining in slow start when the congestion window and slow start threshold are equal. The trace continues by increasing the congestion window nearly linearly, sending packets in groups of five, six, etc.

A timeout in congestion avoidance has the same reaction and is shown in Figure 7.8. The congestion window is reset to one MSS and the slow start threshold is set to 9 packets in flight * MSS / 2 = 6516 bytes. The slow start threshold is exceeded upon receiving the fifth acknowledgment when WiFu TCP transitions from slow start to congestion avoidance and continues to increase the congestion window nearly linearly.

To ensure that WiFu TCP handles multiple timeouts, we induce a back-to-back timeout situation. This shows that WiFu TCP correctly doubles the retransmission timer ensuring correct back off [32]. Furthermore, we should see the effects of each timeout cause WiFu TCP to react appropriately. It should resend twice and the slow start threshold should be set to 2 * MSS. This is because only one packet is in flight at that time; cutting the slow start threshold in half would reduce it to 724 bytes. However, the slow start threshold must be at least 2 * MSS [3].

Figure 7.9 shows multiple timeouts and begins the same as Figure 7.8. After the first timeout, WiFu TCP reacts appropriately and resends the first dropped data packet, which we also drop. We see that the timeout value is correctly doubled from one second to two and that the slow start window is correctly reduced to 2 * MSS which is exceeded upon receiving the second acknowledgement after both timeouts. WiFu TCP transitions to congestion avoidance at this point and continues by sending data packets nearly linearly, in increasing groups of three, four, etc.

# Chapter 8

## Performance Evaluation of WiFu Transport

We evaluate the performance of WiFu Transport on wired and wireless networks. Our comparisons in this chapter pit WiFu TCP against Kernel TCP. We examine numerous topologies and methodologies in these experiments. Our results show that WiFu Transport is able to perform as fast as the kernel on 10 and 100 Mbps Ethernet connections and over a one-hop wireless connection.

Both the wired and wireless experiments share common elements. We use the `nice` command to set our transfer application and the WiFu daemon (applicable only to WiFu TCP) to the highest possible user-space priority. This is done with the intent to give WiFu Transport near kernel-like priority and to ensure that other applications running on our machines running the experiments interfere as little as possible. Each experiment is run for 50 iterations. An iteration consists of running WiFu TCP followed by Kernel TCP. Interleaving protocols balances any short term variance in computer and network processing. All machines are running Ubuntu 10.04.3 LTS with the Linux 2.6.32 kernel.

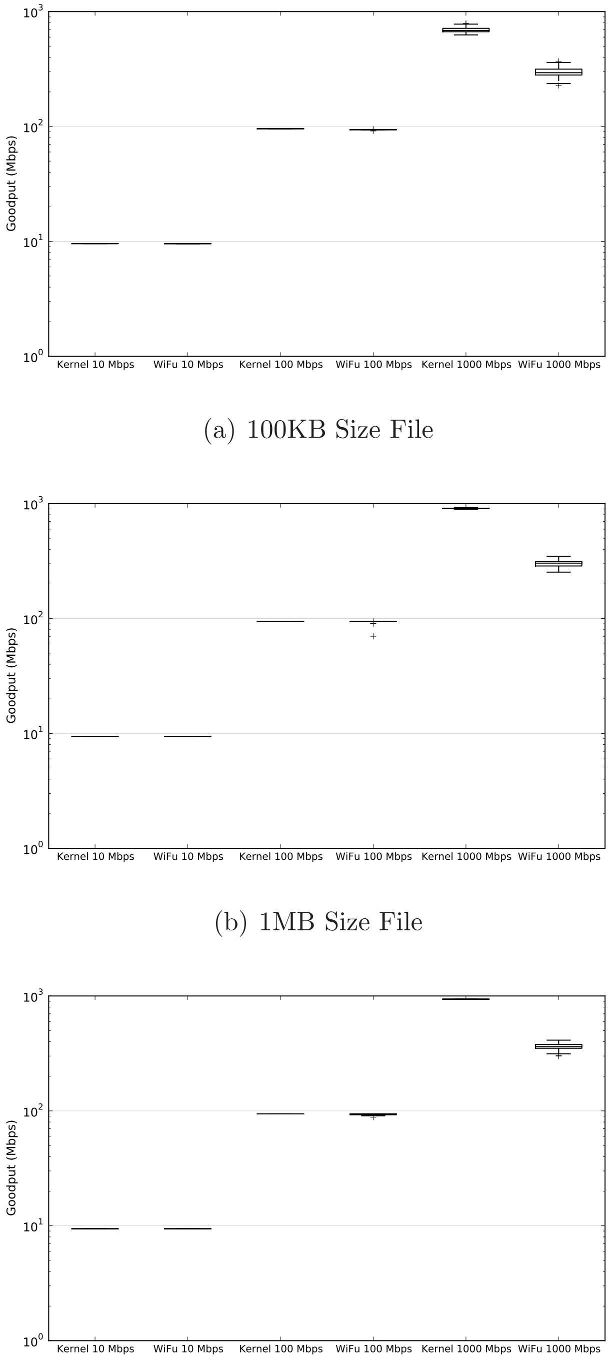
To accommodate some variances in implementation we make the following modifications to WiFu TCP and the transfer application. First, we change the way the initial congestion window is set from one packet to equation one in RFC 3390 [2]. With an MTU of 1448 bytes, this sets the initial congestion window to three packets. This is consistent with Kernel TCP's implementation. Second, we found that the kernel uses delayed acknowledgments in its implementation. Delayed acknowledgements significantly improve performance over wireless links [27]. Because the kernel does not support turning this algorithm off, we utilize the `TCP_QUICKACK` socket option. In our transfer application, if Kernel TCP is running, we set this socket option after each call to `recv()`. This forces the kernel to acknowledge any data received, effectively turning off delayed acknowledgements. Third, to ensure that the garbage collector does not run in the middle of a data transfer we preallocate `TCPPacket` objects on the receiver, expand the heap by 100 MB, use the entire heap before garbage collecting, and set the garbage collector's `free_space_divisor` variable to one.

All reported rates are in terms of goodput and are calculated at the receiver of data as follows. A timestamp is taken *after* the first `recv()` call returns and indicates the start time. Another timestamp, marking the end of the transfer, is taken or updated *before* each call to `recv()`. This allows us to avoid calculating the time for the last `recv()` call that contains no data and indicates the close of the connection by the peer. The only downsides to our approach are (1) missing the time to receive the first bytes of data, and (2) needing to take a timestamp value every time `recv()` is called. The first downside is unavoidable at the receiving side, the second avoids including non-data processing time. The difference between the two times is then divided by the number of bytes received at the application, resulting in goodput.

### 8.1 Wired Results

Running on a wired network shows that WiFu Transport is not limited to its target domain of wireless networks. Furthermore, we are able to examine various sending rates and experiment in a minimal-loss scenario. This enables us to find the maximum performance of WiFu Transport.

In addition to those described above, the wired experiments are run with the following parameters. Each machine has a 64-bit Intel Core 2 Duo 3.16 GHz processor with four GB of RAM and runs the Desktop Edition of Ubuntu. The Ethernet card is configurable to 10, 100, or 1000 Mbps. We run these experiments over a point-to-point direct connection to avoid background traffic impeding the results.



(a) 100KB Size File

(b) 1MB Size File

(c) 10MB Size File

Figure 8.1: Wired Results

| **Rate** | **WiFu TCP** | **Kernel TCP** | **Ratio** |
|---:|---:|---:|---:|
| *100 KB* | | | |
| 10 | 9.555 | 9.564 | 0.999 |
| 100 | 93.880 | 95.654 | 0.981 |
| 1000 | 293.150 | 683.179 | 0.429 |
| *1 MB* | | | |
| 10 | 9.429 | 9.429 | 1.000 |
| 100 | 94.184 | 94.298 | 0.999 |
| 1000 | 302.071 | 906.156 | 0.333 |
| *10 MB* | | | |
| 10 | 9.416 | 9.416 | 1.000 |
| 100 | 94.139 | 94.160 | 1.000 |
| 1000 | 361.862 | 937.729 | 0.386 |

Table 8.1: Wired Median Goodputs

Our first set of experiments compares the goodput of WiFu TCP and Kernel TCP. We test three Ethernet speeds of 10, 100, and 1000 Mbps with file sizes of 100 K, one MB, and 10 MB. Figure 8.1 shows the results. The y-axis scale is logarithmic. Table 8.1 provides actual median numbers for the Ethernet tests. All values are in terms of Mbps with the exception of ratio which has no units. The ratio is defined as the rate of WiFu TCP divided by the rate of Kernel TCP.

The results show that WiFu TCP does extremely well, as compared with Kernel TCP for 10 and 100 Mbps connections. For the 100 KB file size, WiFu TCP is able to perform nearly as well as the Kernel TCP. The larger file sizes of one and 10 MB show virtually no difference between WiFu and Kernel TCP. There is also a slight progression of increased relative performance as the amount of data sent increases.

WiFu TCP performs poorly on the gigabit connection for all file sizes. We achieve between 33.3 and 42.9 percent of the kernel's speed, up to a median of 361 Mbps for the 10 MB data size. These gigabit results show that there is work to be done if WiFu is to be used extensively on Ethernet links faster than 100 Mbps. However, we are encouraged that WiFu is able to process data well above the 100 Mbps rate. We should be able to easily fill the pipe in our target domain of wireless mesh network connections, with a maximum rate of 56 Mbps.

| Hops | WiFu TCP | Kernel TCP | Ratio |
|---|---|---|---|
| 1 | 12.143 | 12.113 | 1.002 |
| 2 | 5.732 | 5.929 | 0.967 |
| 3 | 4.020 | 4.138 | 0.971 |

Table 8.2: Wireless Median Goodputs

## 8.2 Wireless Results

We run wireless experiments on our wireless mesh network at Brigham Young University. Our mesh network consists of donated lab machines; therefore, they are older than the machines used for the wired experiments. We utilize four of our mesh machines to achieve a three-hop topology for our experiments. Two of the mesh machines used in our wireless experiments have a 32-bit Intel Pentium 4 2.4 GHz CPU with 768 MB of RAM and the other two have a 32-bit Intel Pentium 4 3.2 GHz with 1024 MB of RAM. All mesh nodes run the Server Edition of Ubuntu. Each machine has an 802.11 a/b/g wireless card. All experiments are run on 802.11a to avoid interference with other wireless traffic in the building. Furthermore, the wireless cards are set to 24 Mbps and both MAC-layer retries and RTS/CTS are turned off. IP routing is also made static and the transmission power of the wireless cards is set to 17 dBm. Setting the speed lower than the maximum with a high transmission power gives minimal-loss links. We set the transmission power of mesh nodes' cards not used in the experiment to off.

We examine the transfer of one megabyte over one, two, and three hops. The results are shown in Figure 8.2. Detailed median goodput for the 50 iterations for each hop count is found in Table 8.2.

As expected, WiFu TCP performs on par with Kernel TCP at one hop. However, the relative performance diminishes when the hop count is greater than one. This may be due to the different TCP implementations, specifically how they react to detected loss. This is discussed below.

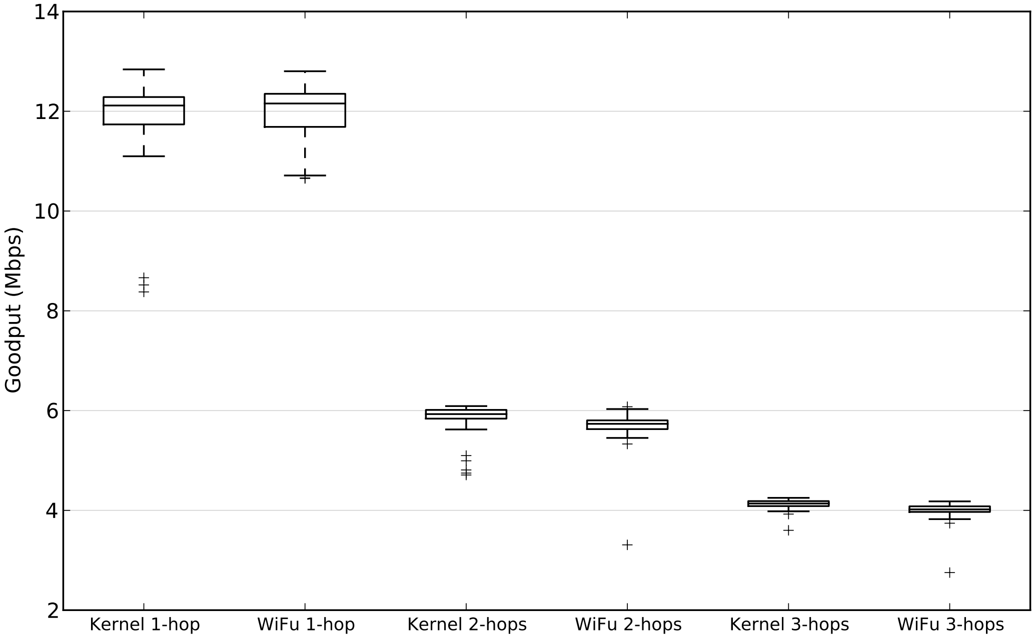


Figure 8.2: Wireless Results

### 8.3 Discussion

WiFu Transport is able to perform as fast as the kernel. There are instances where WiFu Transport is faster than the kernel and vica-versa. These discrepancies may be attributed to one or more of the following implementation details.

First, our implementation is TCP Tahoe, where the kernel is TCP Reno. The biggest impact this has is in a lossy situation, such as a wireless connection spanning more than one hop. Reno approximately cuts the congestion window in half upon loss whereas Tahoe reduces it to one MSS. This may be the reason that our wireless results for more than one hop are a bit slower than the kernel.

Second, our transfer application measures rate according to the amount of data received. This does not account for acknowledgements that still need to be sent after all data has been received and thus we stop our timer. We have seen slight variances when acknowledgements are sent. One additional acknowledgement sent, received, and processed during the time data is being received as opposed to after all data has been received can cause slight variations in performance. This may explain the slight progression of increased relative performance on the slower wired links as the amount of data sent increases. The relative impact of an extra acknowledgement diminishes.

Third, setting a socket option when the kernel is running after each `recv()` call obviously introduces overhead and biases the results in favor of WiFu TCP. It is unknown how much impact this has on the results, but we expect it to be negligible for all experiments except those run with a gigabit connection. A more fair comparison would be to turn on delayed acknowledgements in WiFu TCP as done by Lee [27]. Further investigation into these and other implementation details could provide for different results. However, we have shown that WiFu TCP can perform as fast as Kernel TCP under certain scenarios.

## 8.4 Scalability Results

WiFu Transport supports multiple concurrent connections. We run several experiments to show that WiFu does handle this and that it performs on par with the kernel. The experiments have the same topology as described above for the wired 100 Mbps point-to-point experiments. We only show wired experiment results for scalability because the results will be less conclusive on wireless links due to the inherit issues of TCP reacting to the wireless medium.

### 8.4.1 Instantaneous Goodput

We first examine the instantaneous goodput of both WiFu TCP and Kernel TCP. In these experiments we run a single WiFu Transport daemon instance and the same transfer application used previously. However, we increase the number of threads running in the transfer application to two, five, and 10. All threads establish a TCP connection before any thread begins sending data to ensure that connection setup is excluded in any measurement. Each connection sends one megabyte of data over a 100 Mbps Ethernet link. We stagger the start time of each thread by approximately 10 milliseconds. This shows the progression of threads being added and the bandwidth sharing that occurs. Our results show that running multiple threads in no way inhibits WiFu's performance as compared to the kernel. Furthermore, we show that WiFu TCP and Kernel TCP react in nearly the same manner for multiple flows. This further validates that our WiFu TCP implementation is correct.

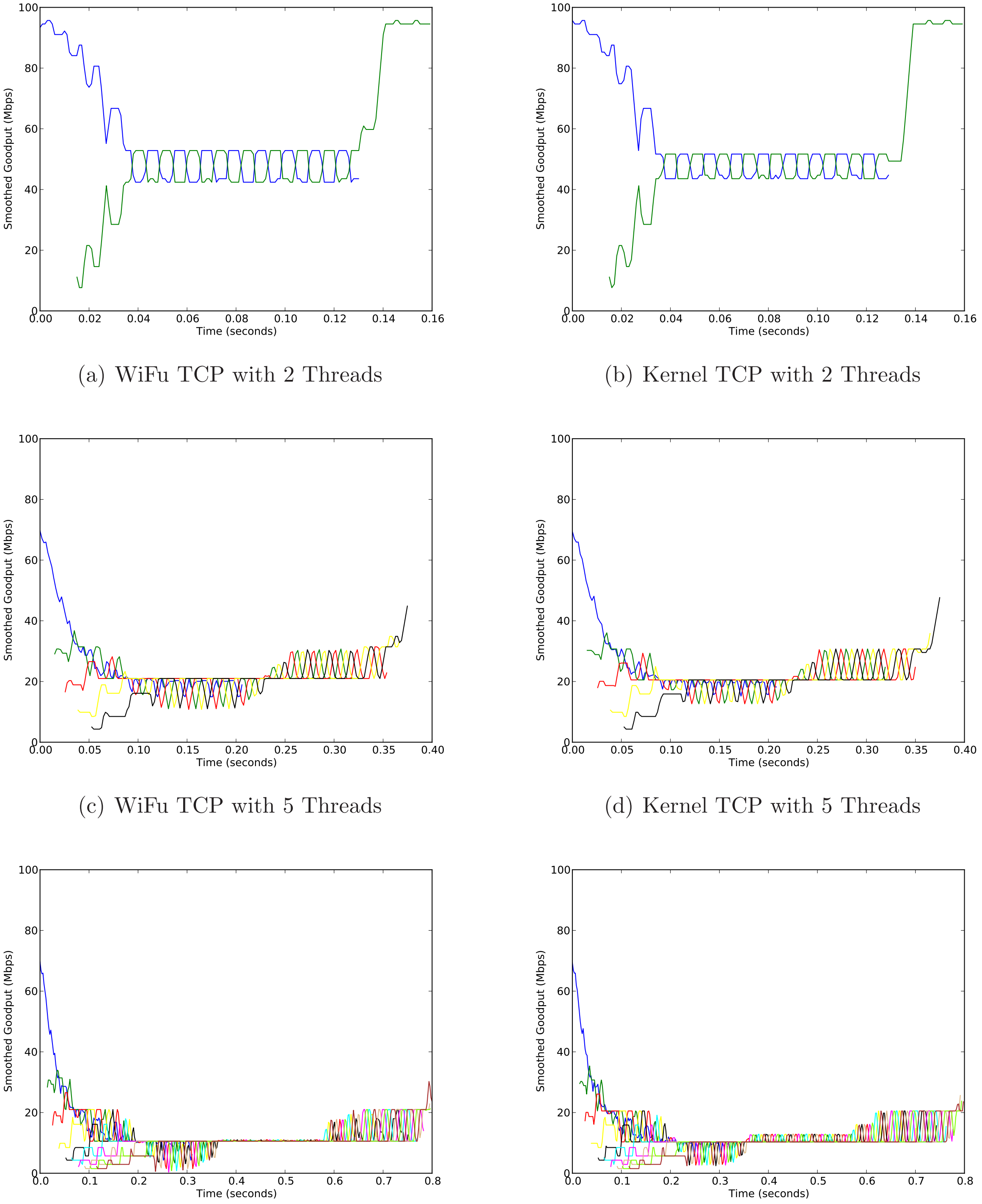
The results are shown in Figure 8.3. These graphs represent a single set of flows smoothed over time. We smooth each flow by averaging the goodput over a sliding window. Figures 8.3(a) and 8.3(b) have a 10 millisecond window that slides every one millisecond. Figures 8.3(c), 8.3(d), 8.3(e), and 8.3(f) have a window of 50 milliseconds that slides every two milliseconds.

The pairwise graphs are virtually identical in behavior. There are only two differences worth mentioning. The first, best shown in Figures 8.3(a) and 8.3(b), is the peak-to-peak amplitude of the curve when the link is shared. This is repeated among the graphs. Generally WiFu TCP has a bigger peak-to-peak amplitude than Kernel TCP. Secondly, Kernel TCP (Figure 8.3(f)) shows more variance between the 0.35 and 0.55 second time periods than WiFu TCP (Figure 8.3(e)) does. We consider these differences minor and conclude that the general behavior is identical.

### 8.4.2 Multiple Flow Results

We now examine the performance of WiFu TCP and Kernel TCP in multiple flow situations. For these experiments we run several flows, each transferring one megabyte of data, over the same point-to-point Ethernet connection described earlier. We set the connection speed to 100 Mbps and test two, five, and 10 concurrent flows. As before, we alternate running WiFu TCP and Kernel TCP and run a total of 50 iterations for per flow count. Rather than staggering start times, all flows begin at approximately the same time.

We measure aggregate goodput and fairness to determine how well WiFu TCP compares with Kernel TCP. Aggregate goodput is the sum of all the goodputs for all flows in



(a) WiFu TCP with 2 Threads

(b) Kernel TCP with 2 Threads

(c) WiFu TCP with 5 Threads

(d) Kernel TCP with 5 Threads

(e) WiFu TCP with 10 Threads

(f) Kernel TCP with 10 Threads

Figure 8.3: Instantaneous Results

| Flows | WiFu TCP | Kernel TCP | Ratio |
|---|---|---|---|
| 2 | 94.855 | 94.719 | 1.001 |
| 5 | 95.206 | 95.894 | 0.993 |
| 10 | 95.311 | 96.111 | 0.992 |

Table 8.3: Wired Multiple Flows Median Aggregate Goodputs

| Flows | WiFu TCP | Kernel TCP |
|---|---|---|
| 2 | 0.999977 | 0.999972 |
| 5 | 0.999972 | 0.999901 |
| 10 | 0.999971 | 0.999882 |

Table 8.4: Wired Multiple Flows Median Jain's Fairness Index

a single iteration and gives a general sense for how much of the link is being used. We use Jain's fairness index [23] to measure fairness. Measuring fairness indicates whether all concurrent flows are receiving an equal share of the available bandwidth. A Jain's fairness index of one indicates perfect sharing between competing flows; anything below one indicates some unfairness.

The aggregate goodput results are shown in Figure 8.4. We show the median aggregate goodputs for each thread count in Table 8.3. WiFu TCP generally performs on par with Kernel TCP in all areas. Surprisingly, in the five flow experiment several runs have an aggregate goodput greater than the link capacity. This may be explained as follows. All rates are computed at the receiving side; we cannot start a timer until the first portion of data has arrived. It is possible that in these instances a portion of the flows were delayed a little bit, the timer started later, and that the delayed flow received a larger portion of the link capacity because other threads had already finished. This can further be confirmed by examining fairness.

Examining the fairness ensures that the WiFu Transport daemon treats all sockets using it fairly. Furthermore, we can evaluate our TCP implementation to show that competing TCP flows are each given an equal share of the available link capacity. We compute fairness on a per-iteration basis. All iterations are plotted in Figure 8.5. We show the median Jain's fairness index for each flow count in Table 8.4. This table shows that WiFu TCP is slightly more fair on all tested flow counts than Kernel TCP. For all flow counts, both WiFu TCP and Kernel TCP have a Jain's fairness index near one, indicating that these implementations fairly share the bandwidth.

Figure 8.5 shows that WiFu TCP with five flows has unfair outliers. Upon further examination we have confirmed that these outliers correspond with the aggregate goodput outliers in Figure 8.4. It is possible that these outliers have been caused simply by being in user-space. Further investigation into these outliers is necessary to confirm this hypothesis. Overall, we conclude that both WiFu Transport and WiFu TCP are fair in processing data.

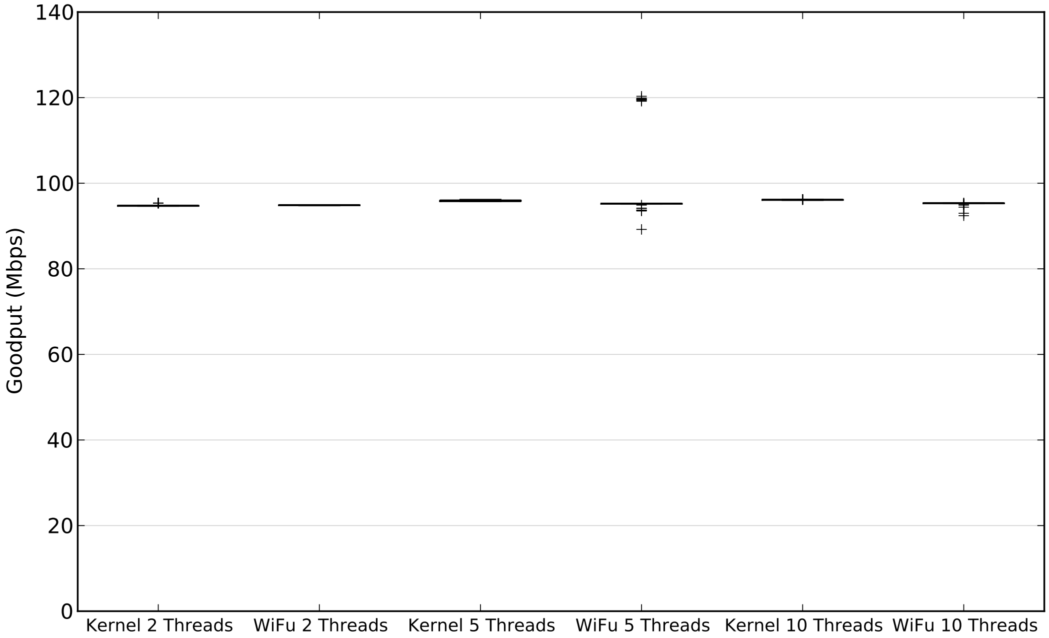


Figure 8.4: Wired Multiple Flows Aggregate Goodput

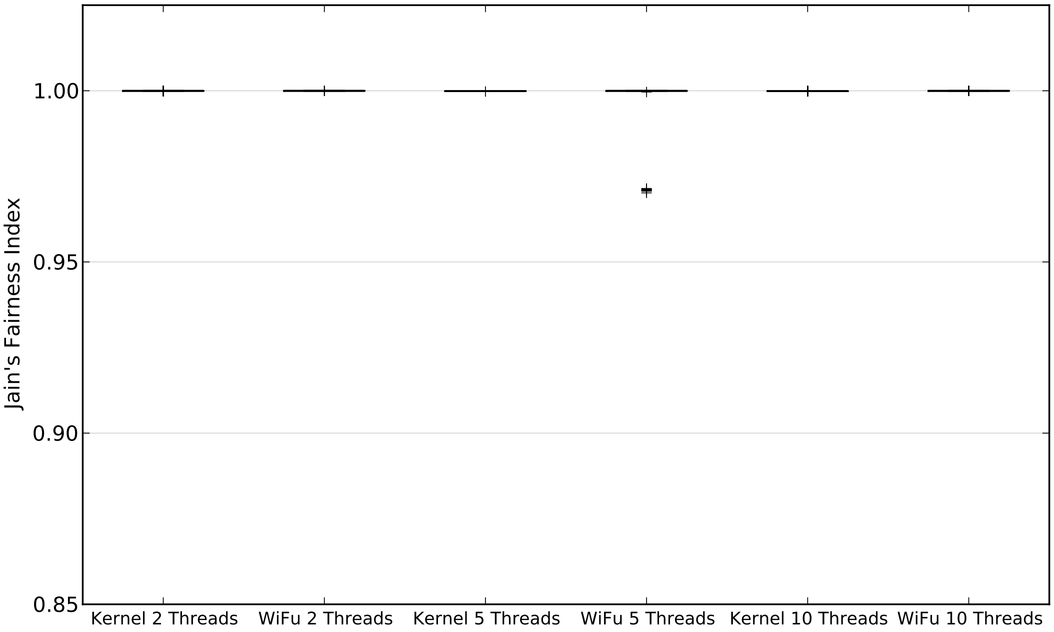


Figure 8.5: Wired Multiple Flows Jain's Fairness Index

# Chapter 9

# Conclusions and Future Work

## 9.1 Conclusions

This thesis makes four major contributions. First, we have designed and built WiFu Transport: a user-level protocol framework. Second, we have decomposed TCP into three components: connection management, congestion control, and reliability. Third, we have shown that our TCP implementation is correct, validating that it is possible to split apart TCP. Finally, we used our TCP implementation to run performance and scalability tests of WiFu Transport and compare the results against the Linux kernel. WiFu Transport is able to achieve performance on par with the kernel on 10 and 100 Mbps Ethernet links and in its target domain of wireless links. Furthermore, it fairly supports multiple threads and flows running concurrently making it viable for extensive network research. Others have already used the framework to develop and test TCP variants, a hybrid, and new protocols. Our work shows that WiFu Transport is flexible, promotes code reuse, and is scalable without imposing significant performance problems.

## 9.2 Future Work

There are many places where WiFu Transport could be improved or explored that we leave as future work. Relying on a third-party library has been extremely helpful in solving memory management issues. However, the garbage collector we use poses issues when conducting performance evaluations. Networking code needs to be fast and run uninterrupted. Though the garbage collector runs extremely fast and may be suitable for most applications, work needs to be done to make it viable for networking software. A potential solution would be to not use a garbage collector and handle memory management manually. This poses several problems discussed earlier. Another solution might be to manage when the garbage collector runs to ensure that it impedes as little as possible.

Another issue that needs to be addressed is the poor performance on gigabit connections. This will especially be important as network speeds increase. Possible gains in performance include better interprocess communication such as using shared memory instead of a Unix socket. Furthermore, examining the generation and use of events in WiFu Transport may show that improvements can be made. For example, our current event system might be too fine-grained or not used efficiently. Processing fewer events more intelligently may lead to performance improvements.

WiFu Transport is extremely fair in packet processing a majority of the time. However, we have shown that there are isolated cases of unfairness. This might simply be a manifestation of a user-space solution. This area should be explored to see why unfairness occurs and what can be done about it.

Currently, we have only tested WiFu Transport with our own transfer application. Moving forward, it will be necessary to make it as usable as possible. One step in this direction would be to enable existing applications to link in the WiFu Transport front-end library. Another step would be to test interoperability of WiFu TCP with other TCPs.

We have taken a first cut at compartmentalizing TCP into connection manager, congestion control and reliability components. There are potential areas where this could be improved. The components could be better compartmentalized. For example, we currently have a lot of code in the `TCPTahoe` class that composes the various TCP components. Some of this code may be able to be pushed into the components. Furthermore, it may make sense to create specialized `Socket` objects that are protocol specific to avoid TCP variable information from being duplicated in multiple components.

We have documented a few known bugs and potential areas for refactoring or improvement. These are issues and questions that have arisen as the project proceeded and we feel are less important to deal with immediately. Some of the areas of improvement for WiFu TCP include supporting more TCP options, such as the MSS header option [34], handling path MTU discovery [29], and Nagle's Algorithm [30]. We intend to work on these as needed and as resources permit.

Finally, we request that others use this software to build and test alternative transport layer protocol solutions. We have experimented with only a few protocols; we anticipate that the research community will find WiFu Transport useful for experimenting with novel transport protocols rather than relying on simulation.

# References

[1] A. S. Abdallah, M. D. Horvath, M. S. Thompson, A. B. MacKenzie, and L. A. DaSilva, "Facilitating experimental networking research with the FINS framework," in *Proceedings of the 6th International Workshop on Wireless Network Testbeds, Experimental Evaluation and Characterization*, 2011, pp. 103–104.

[2] M. Allman, S. Floyd, and C. Partridge, "RFC 3390: Increasing TCP's initial window," 2002.

[3] M. Allman, V. Paxson, and W. Stevens, "RFC 2581: TCP congestion control," 1999.

[4] E. Altman and T. Jiménez, "Novel delayed ACK techniques for improving TCP performance in multihop wireless networks," in *Personal Wireless Communications*, M. Conti, S. Giordano, E. Gregori, and S. Olariu, Eds., 2003, vol. 2775, pp. 237–250.

[5] S. Bellovin, "RFC 1948: Defending against sequence number attacks," 1996.

[6] B. Boehm, "A spiral model of software development and enhancement," *Computer*, vol. 21, no. 5, pp. 61–72, 1988.

[7] H.-J. Boehm and M. Weiser, "Garbage collection in an uncooperative environment," *Software–Practice & Experience*, vol. 18, pp. 807–820, 1988.

[8] R. Braden, "RFC 1122: Requirements for internet hosts – communication layers," 1989.

[9] P. G. Bridges, G. T. Wong, M. Hiltunen, R. D. Schlichting, and M. J. Barrick, "A configurable and extensible transport protocol," *IEEE/ACM Transactions on Networking*, vol. 15, pp. 1254–1265, 2007.

[10] D. Clark, "RFC 813: Window and acknowledgement strategy in TCP," 1982.

[11] G. Combs *et al.*, "Wireshark network protocol analyzer." [Online]. Available: http://www.wireshark.org/

[12] P. A. Dinda, "The Minet TCP/IP stack," Northwestern University Department of Computer Science, Tech. Rep. NWU-CS-02-08, 2002.

[13] S. ElRakabawy, A. Klemm, and C. Lindemann, "TCP with adaptive pacing for multihop wireless networks," in *Proceedings of the 6th ACM International Symposium on Mobile Ad Hoc Networking and Computing*, 2005, pp. 288–299.

[14] D. Ely, S. Savage, and D. Wetherall, "Alpine: A user-level infrastructure for network protocol development," in *Proceedings of the 3rd Conference on USENIX Symposium on Internet Technologies and Systems-Volume 3*, 2001, pp. 15–15.

[15] K. Fall and S. Floyd, "Simulation-based comparisons of Tahoe, Reno and SACK TCP," *ACM SIGCOMM Computer Communication Review*, vol. 26, no. 3, pp. 5–21, 1996.

[16] M. Faulkner, M. Jakeman, and S. Pink, "Architectural implications of performing network protocol processing closer to the application," in *Proceedings of the International Symposium on Performance Evaluation of Computer and Telecommunication Systems*, 2007.

[17] V. Gambiroza, B. Sadeghi, and E. Knightly, "End-to-end performance and fairness in multihop wireless backhaul networks," in *Proceedings of the 10th Annual International Conference on Mobile Computing and Networking*, 2004, pp. 287–301.

[18] Y. Gu and R. L. Grossman, "Supporting configurable congestion control in data transport services," in *Proceedings of ACM/IEEE Conference on Supercomputing*, 2005, p. 31.

[19] G. Holland and N. Vaidya, "Analysis of TCP performance over mobile ad hoc networks," *Wireless Networks*, vol. 8, no. 2/3, pp. 275–288, 2002.

[20] V. Jacobson, R. Braden, and D. Borman, "RFC 1323: TCP extensions for high performance," 1992.

[21] V. Jacobson and B. Felderman, "A modest proposal to help speed up & scale up the linux networking stack," in *Proceedings of Linux Conference Australia*, 2006.

[22] V. Jacobson, C. Leres, and S. McCanne, "Packet capture library." [Online]. Available: http://www.tcpdump.org/

[23] R. Jain, D. Chiu, and W. Hawe, "A quantitative measure of fairness and discrimination for resource allocation in shared computer systems," *DEC Research Report TR-301*, 1984.

[24] R. Johnson, E. Gamma, R. Helm, and J. Vlissides, *Design patterns: elements of reusable object-oriented software.* Addison-Wesley, 1995.

[25] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. Kaashoek, “The click modular router,” *ACM Transactions on Computer Systems*, vol. 18, no. 3, pp. 263–297, 2000.

[26] J. Kurose and K. Ross, *Computer networking: a top-down approach.* Addison-Wesley, 2009.

[27] R. Lee, “Feasibility of TCP for wireless mesh networks,” Master Thesis, Department of Computer Science, Brigham Young University, 2012.

[28] S. McCanne and S. Floyd, “ns network simulator.” [Online]. Available: http://www.isi.edu/nsnam/ns

[29] J. Mogul and S. Deering, “RFC 1191: Path MTU discovery,” 1990.

[30] J. Nagle, “RFC 896: Congestion control in IP/TCP internetworks,” 1984.

[31] K. Nahm, A. Helmy, and C. Jay Kuo, “TCP over multihop 802.11 networks: Issues and performance enhancement,” in *Proceedings of the 6th ACM international symposium on Mobile ad hoc networking and computing*, 2005, pp. 277–287.

[32] V. Paxson and M. Allman, “RFC 2988: Computing TCP’s retransmission timer,” 2000.

[33] J. Postel, “RFC 793: Transmission control protocol,” 1981.

[34] ——, “RFC 879: The TCP maximum segment size and related topics,” 1983.

[35] P. Pradhan, S. Kandula, W. Xu, A. Shaikh, and E. Nahum, “Daytona: A user-level TCP stack.” [Online]. Available: http://nms.csail.mit.edu/~kandula/data/daytona.pdf

[36] R. Russell and H. Welte, “Linux netfilter hacking howto,” 2002. [Online]. Available: http://www.netfilter.org/documentation/HOWTO/netfilter-hacking-HOWTO.html

[37] J. Saltzer, D. Reed, and D. Clark, “End-to-end arguments in system design,” *ACM Transactions on Computer Systems*, vol. 2, no. 4, pp. 277–288, 1984.

[38] D. Scofield, L. Wang, and D. Zappala, “HxH: a hop-by-hop transport protocol for multi-hop wireless networks,” in *Proceedings of the 4th Annual International Conference on Wireless Internet*, 2008.

[39] J. Shi, O. Gurewitz, V. Mancuso, J. Camp, and E. Knightly, “Measurement and modeling of the origins of starvation in congestion controlled mesh networks,” in *Proceedings of The 27th Conference on Computer Communications*, 2008, pp. 1633–1641.

[40] A. Shvets, “Design patterns.” [Online]. Available: http://sourcemaking.com/design_patterns

[41] W. Stevens, “RFC 2001: TCP slow start, congestion avoidance, fast retransmit, and fast recovery algorithms,” 1997.

[42] K. Sundaresan, V. Anantharaman, H. Hsieh, and R. Sivakumar, “ATP: A reliable transport protocol for ad hoc networks,” *IEEE Transactions on Mobile Computing*, vol. 4, no. 6, pp. 588–603, 2005.

[43] X. Zhang, R. Buck, and D. Zappala, “Experimental performance evaluation of ATP in a wireless mesh network,” in *Proceedings of Eighth IEEE International Conference on Mobile Ad-Hoc and Sensor Systems*, 2011, pp. 122–131.