WiFu: A Composable Toolkit for Experimental Wireless Transport Protocols

Randy Buck, Rich Lee, Philip Lundrigan, and Daniel Zappala
Computer Science Department
Brigham Young University
Provo, UT 84602, U.S.A.
Email: randy.buck@byu.net, leer@byu.net, philiplundrigan@gmail.com, zappala@cs.byu.edu

***Abstract*—Extensive research has been performed on improving TCP performance in multi-hop wireless networks, but there have been relatively few experimental evaluations of this work. To make it easier to conduct research in this area, we are releasing WiFu, an open-source toolkit for developing experimental wireless transport protocols. WiFu provides for user-space development of reliable transport and rate control algorithms, greatly simplifying the implementation effort required. In this paper, we describe the architecture of the WiFu toolkit, which decomposes transport protocols into smaller components that enable rapid, plug-and-play development of new variants. We present experiments to demonstrate that the performance of WiFu compares favorably to the Linux kernel for wireless networks. We illustrate the utility of WiFu by using it to conduct experiments with several wireless transport protocols, and show that the performance of some protocols differs significantly from previously reported results.**

I. Introduction

It is well known that application throughput can suffer in multi-hop wireless networks, due to interactions between the IEEE 802.11 MAC and TCP [1], [2], [3]. A wide range of research addresses this issue by extending or redesigning TCP, however relatively few solutions have been implemented and tested in wireless networks. This is somewhat understandable, since these solutions involve changes to the transport layer of wireless hosts and routers; modifying the operating system is time consuming and difficult. However, the scarcity of experimental evaluations severely limits the impact of current research on future wireless networks.

To address this problem, we are releasing WiFu[1], an open-source, user-level toolkit for experimental wireless transport protocols. Developing in user space rather than kernel space allows for rapid implementation and testing of new transport protocols, makes it easier to manage the code, and enables researchers to contribute code without needing to have the expertise to develop in the kernel. Once a protocol has demonstrated promise, it can then be moved into the kernel for deployment. To allow for a wider range of experimental designs, WiFu also enables cross-layer interactions, providing convenient user-level mechanisms for manipulating packets at routers and interacting with modified network drivers through the Linux */proc* interface.

A number of projects have ported the networking stack directly to user space or created user-level implementations of TCP [4], [5], [6]. However, in many cases this work is not flexible enough for radical transport protocol designs or is not fast enough to compete with the kernel. A key innovation in WiFu is that it separates transport protocol functionality into major components, providing flexibility while maintaining performance. By separating the transport protocol into components we can "plug-and-play" different mechanisms, allowing testing of radically different designs with relatively small amounts of new code.

In this paper we describe the architecture of WiFu, which consists of two parts: (a) *WiFu Transport*, which provides a framework for developing end-to-end protocols and (b) *WiFu Core*, which runs on every node in the network and enables hop-by-hop control over packets and cross-layer interactions. Both parts use a modular, event-driven architecture, allowing for greater flexibility than a traditional networking stack. Our contributions in this work include:

- A user-space framework for transport protocol design that is flexible, scalable, and has performance that is competitive with the Linux kernel. Our experiments show that application goodput in *WiFu Transport* is as fast as the Linux kernel for one-hop, two-hop, and three-hop wireless paths, and can saturate a 10 Mbps and 100 Mbps Ethernet. Packet interception within *WiFu Core* imposes negligible overhead for large file transfers, providing packet interception at speeds up to 935 Mbps.
- A clean-slate implementation of TCP that is separated into components, demonstrating the viability of component-based transport protocol development. While other research has investigated decomposing TCP using micro-protocols [7], we believe ours is the first to show packet-level fidelity with TCP while also providing equivalent performance with the kernel. Packet-level traces of our decomposed version of TCP are nearly identical to traces produced by the kernel.
- Investigation of two proposed TCP modifications for wireless networks, adaptive pacing [8] and delayed ACKs [9]. Our experiments show that adaptive pacing increases goodput and fairness for TCP, but only when multiple flows are active. Additional experiments show that delayed ACKs improve goodput more than was previously reported. We also describe our experience using WiFu to implement an experimental, hybrid transport protocol that combines TCP reliability with measurement-based rate control.

This material is based upon work supported by the National Science Foundation under Grant No. 0917240

[1]From a combination of WiFi and Kung Fu.

Overall, our results show that WiFu provides a flexible environment for user-space transport protocol development on an unmodified Linux kernel, promotes code reuse, maintains good performance and provides scalability.

II. Design Goals

Our primary purpose in creating WiFu is to enable rapid development of novel transport protocols for wireless networks. In support of this vision, our design has been guided by the following goals:

Flexibility. Researchers in wireless networks want to create radical new transport protocol designs, rather than being constrained by existing conventions. WiFu meets this goal by using protocol decomposition, so that a transport protocol consists of several components, such as a connection manager, reliable transfer, and a congestion controller. A developer can create a new protocol by mixing existing components with new ones, and can also override methods to create new functionality. Furthermore, WiFu supports multiple, simultaneous transport protocols.

Code Reuse. Encouraging code reuse helps developers rapidly create new protocols by avoiding work that is not unique to their research. One of the reasons why using a simulator to evaluate new research is so inviting is because a good network simulator offers a framework that provides event handling, timers, packet formats, and other convenient functionality that a developer doesn't want to have to reinvent each time. In WiFu, we replicate this experience by providing similar functionality and by orienting development around design patterns [10].

Performance. Transport protocols in user space should be able to achieve performance equal to that of the Linux kernel. This is important for protocol developers because they are often trying to demonstrate improved performance relative to a particular version of TCP that is used by the kernel. Previous efforts to port transport protocols to user space have focused on flexibility or teachability to the exclusion of performance. We engineer WiFu so that it is as fast as possible, without sacrificing the goal of flexibility. We meet this goal in part by utilizing an event-driven architecture and raw sockets.

Scalability. In addition to performance, transport protocol developers are often concerned with fairness among multiple connections. Thus, our design of WiFu includes a socket interface so that multiple applications can utilize the transport stack simultaneously, and where threading is used we ensure thread safety. WiFu also allows multiple transport protocols to be operating simultaneously, so that performance and fairness can be directly compared.

We note that it can be difficult to achieve fast and scalable performance while also trying to provide a flexible environment for rapid development. We do not strictly prioritize one goal over the other, but seek to develop a toolkit that balances these concerns while pushing for performance optimizations wherever possible.

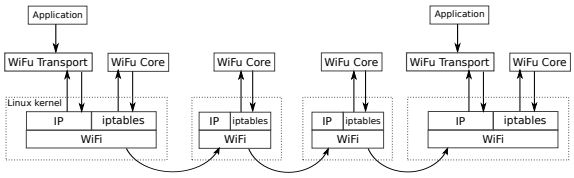


Fig. 1. Basic operation of the *WiFu* toolkit

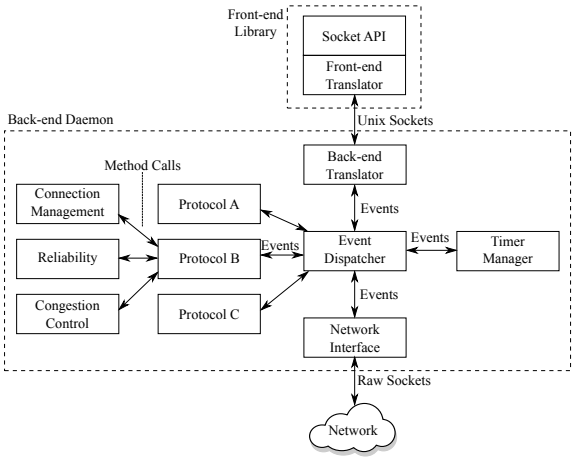


Fig. 2. *WiFu Transport* architecture

III. The WiFu System

Figure 1 shows the basic operation of the *WiFu* toolkit. To meet the diverse needs of transport protocol designers working in the wireless space, WiFu consists of two parts: *WiFu Transport* and *WiFu Core*. *WiFu Transport* provides a flexible environment for traditional end-to-end transport protocol designs, including the BSD socket API so that developers can test their protocols with standard applications. One focus of *WiFu Transport* is on enabling protocol decomposition, so that radical new designs can be easily created by combining both existing and newly-designed reliability and congestion control components. *WiFu Core* enables cross-layer interactions on each hop of a connection, by providing a simple method to intercept and modify packets. This allows for strategies as diverse as per-hop reliability, collecting delay feedback from the driver, or pacing packets to avoid collisions with other wireless devices. All WiFu code is object-oriented and written in C++. The following sections describe the architecture in more detail.

A. WiFu Transport

Figure 2 shows the *WiFu Transport* architecture, which includes a back-end daemon that implements an event-driven framework for transport protocol development, plus a front-end library that provides a socket API for linking to applications. We describe the major pieces of the architecture below:

1) Event Dispatcher: The central part of the *WiFu Transport* architecture is an event-driven dispatcher, which delivers events among the different modules. An event can include the arrival of a packet, a socket call, a timer firing or an indication

that data has arrived in the send buffer. WiFu ensures that once a protocol receives a new packet it is handled completely before processing the next packet or event. The dispatcher and the event-processing modules each run in a separate thread with thread-safe queues.

2) Protocol Decomposition: WiFu allows a developer to decompose a transport protocol into a set of components, similar to the decomposition used in Click [11] and CTP [7], [12]. This provides a tremendous amount of flexibility, because a protocol designer can easily swap components around to create hybrid transport functionality. For example, the standard TCP congestion control algorithm can be replaced with a rate-based controller. WiFu currently includes connection management, reliable transfer, congestion control, and rate limiter components.

In order to further divide transport protocol components into smaller reusable pieces, *WiFu Transport* uses the state design pattern [10]. To create a new protocol component, a developer simply creates a new state machine that re-uses existing states or defines new ones, along with associated methods. Modifying a protocol can be as simple as changing one of its states.

3) Network Interface: The network interface uses raw sockets to provide a high speed mechanism for sending and receiving IP packets. This avoids any overhead associated with running on top of UDP, and also enables us to re-use the kernel's TCP and IP packet headers. To prevent the kernel from processing WiFu packets, we use transport protocol numbers that are not currently in use by the Linux kernel.

The network interface supports logging of packets in `pcap` format, which allows Wireshark to read WiFu traces for easier protocol debugging and visualization. WiFu also includes a mock network object that provides complete control over an emulated network, such as dropping packets according to specified percentages. Packets can also be dropped or delayed based on sequence and ACK numbers, allowing protocols to be tested for correctness.

4) Timer Manager: The timer manager provides the convenience of handling any event timers for transport protocols. Modules can create new timers and can also cancel timers.

5) Socket API: WiFu Transport implements a subset of the standard BSD socket API inside a static library, allowing developers to easily port applications to use a new transport protocol developed in WiFu. The front-end socket library communicates with the back-end using a UNIX socket. Note that some socket calls can be blocking, and the front-end library handles this via asynchronous communication with the back-end. The socket API can support multiple concurrent applications and is thread-safe.

The socket API in *WiFu Transport* supports the following socket calls: *socket(), bind(), listen(), accept(), connect(), send(), sendto(), recv(), recvfrom(), getsockopt(), setsockopt(), and close()*. WiFu does not support all functionality in each of these calls, but contains the bulk of what a typical application uses, with some less-used options and flags slated for future development. We have extended the *getsockopt()* and *setsockopt()* functions to configure transport protocols, since researchers may devise new parameters. This expedites experimentation as it is simply an extra function call at the application level and some minor work in the back-end to use any socket option data.

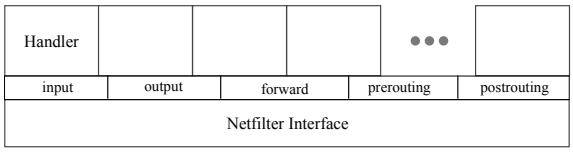


Fig. 3. *WiFu Core* architecture

B. WiFu Core

In designing *WiFu Core*, our goal is to provide an efficient and robust method for intercepting and processing IP packets, while still supporting user-space development on standard Linux kernels. Note that the Click modular router [11] can forward 333,000 64-byte packets per second, but only by using a specialized kernel module and modified device drivers. We want to achieve the best performance possible on an unmodified Linux system. Click also intercepts packets at the driver level, whereas we want to capture them at the IP level. This allows developers to focus on reliability and rate control, rather than routing.

To meet this goal, *WiFu Core* uses the Linux `iptables` software with the `netfilter` interface to intercept packets at each node in the wireless network. We use a feature of `iptables` that allows a rule to specify a queue in which matching packets are placed. WiFu can then read packets from the queue, modify them, reorder them, or deliver the packets back to the kernel for continued processing.

The *WiFu Core* architecture is shown in Figure 3. The `netfilter` interface reads the filter specifications from a configuration file, adds these rules to `iptables`, then reads packets from the `netfilter` queues. WiFu uses the IP protocol field to determine which handler to call for each intercepted packet, then passes the packet to the appropriate handler method – input, output, forward, prerouting, or postrouting – depending on where in the IP forwarding process the packet was intercepted. To add a new handler, a developer only has to add one line to a configuration file and then define a handler with those methods he wants to support.

Once a handler is finished with a packet, it delivers a *verdict* to the kernel through the `netfilter` interface. The packet can be either dropped or accepted for further processing by the kernel, for example to be delivered to an application or routed over the network.

IV. Protocol Decomposition

To demonstrate the feasibility and desirability of protocol decomposition, we have used *WiFu Transport* to develop a clean-slate implementation of TCP based on the RFC specifications, but separated into individual components. Network researchers have commonly believed that TCP reliability and

congestion control could be separated, to provide greater flexibility for new variants. Furthermore, research has shown that TCP-like functionality can be composed of even smaller components, called micro-protocols [7]. However, we believe our research is the first to demonstrate a nearly-complete, decomposed version of TCP that also maintains a very high degree of fidelity to TCP standards.

A major benefit that comes from decomposing TCP is that it allows these basic components to be easily extended or replaced by other protocol functionality. This is particularly important for transport layer solutions that focus only on one portion of TCP, for example, a new congestion control algorithm. By using a component framework, we can evaluate the integration of congestion control improvements with the rest of TCP's reliability and connection management features.

A. Challenges

At first glance, the division of TCP functionality into components seems trivial. Connection management handles connection setup and teardown. Reliability ensures that all data is delivered to the application in order. Congestion control determines the speed at which data is sent. A coordinating protocol composes these three into a functioning whole, and includes some generic event and packet validation and processing.

However, a number of challenges become apparent on closer examination. First, there are numerous versions of TCP that have been specified and implemented, including Tahoe, Reno, Vegas, BIC, Cubic and more. As a proof of concept, we focus on TCP Tahoe in our implementation. Second, TCP is complex. We concentrate on the major concepts such as connection management, reliability, and congestion control. Our implementation supports the time stamp option [13], slow start, congestion avoidance, fast retransmit, and duplicate ACKs. We do not implement the functionality behind the RST, PSH, and URG control bits. Third, there is no clear standard for TCP Tahoe, only a collection of RFCs [14], [15], [16], [17], [18], [19], [13], [20], [21] 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, and by comparing to what has been implemented in ns-2 [22].

The fourth and biggest challenge is knowing how to separate TCP logic into components. During development, we discovered a number of interesting questions, including the following.

1) Which component should send data?: This is a tricky question because all components need to be involved with sending data. The connection manager needs to send SYN, FIN, and ACK segments to start or end a connection. 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. Congestion control is the only component, however, that knows how much data may be sent and how fast. Thus we have the congestion control component send data, and each time this generates an event that notifies the reliability component for tracking.

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, since it sets timers and handles acknowledgements. However, the congestion control component is the one that sends data and it needs to update its state when loss is detected. Accordingly, the reliability component handles detecting loss and enqueues a resend event. The congestion control module processes this event by setting its state accordingly and then resends data according to its sending policy.

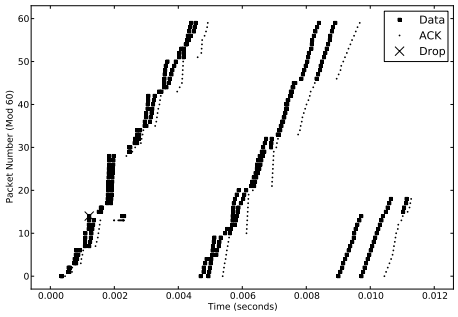
3) Which component should check whether a received packet is valid?: TCP determines whether a received packet is valid by ensuring that it contains data within the receive window or that it acknowledges something sent but not yet acknowledged. These checks are scattered throughout the specifications, and in some cases a general validity check is superseded by a more specific check by a particular component. Our solution is to make the most generic validation checks as soon as possible in the protocol object. If the packet passes this check, the protocol delegates the event to each of the components in turn. This means that a component may duplicate some validation checks, however this ensures that each component remains independent.

4) How do independent components share state?: There are numerous elements of TCP state that serve multiple purposes. For example, send buffer variables and the receive window variable are all used both for reliability and congestion control purposes. 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. This solution ensures component independence and re-uses the event architecture so components may update their internal state.

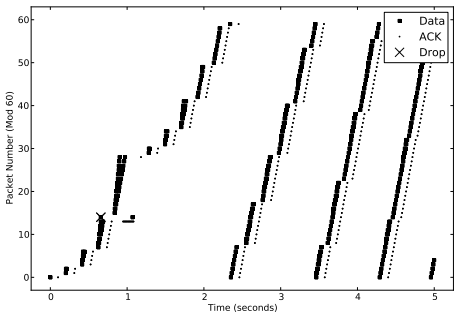
5) Which component should be responsible for sending ACK segments?: The connection manager and the congestion control components both send segments that need to be acknowledged. For the connection manager, an ACK bit indicates a change in state, whereas the reliability component uses the ACK bit to ensure data arrives to the application. Thus the connection manager sends ACK segments when opening or closing a connection, and the reliability manager sends ACK segments in response to data.

B. Validation

To validate that our implementation of Tahoe is correct, we compare with packet traces taken from the ns-2 simulator, since it is one of the only available versions of TCP that contains the Tahoe variant. While we have examined a large number of packet traces, we do not show them all here, due to space considerations. WiFu includes a mock network module in a testing suite that rigorously examines many aspects of TCP Tahoe under various loss scenarios. Thus, we



(a) WiFu TCP



(b) ns-2 TCP

Fig. 4. WiFu TCP and ns-2 Trace Comparison: One Drop

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

One set of scenarios we use are the loss scenarios in the simulations conducted by Fall and Floyd [23]. Figure 4 shows a trace of scenario where one packet is lost during slow start. In the WiFu TCP trace, the first square for a particular sequence number shows when a packet is sent and the second one shows when a packet is received. In the ns-2 TCP trace, the first square shows when a packet is queued and the second one shows when a packet leaves the queue. The small dots indicate when an ACK arrives at the sender. In this scenario, the congestion window should be reduced to one MSS, 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 [17]. Furthermore, because the slow start threshold is halved, this illustrates the correct transition from slow start to congestion avoidance.

We note that WiFu TCP acts nearly identically to ns-2 in this trace. ns-2 TCP handles the transition from slow start to congestion avoidance slightly different that WiFu TCP due to the congestion window and MSS being in terms of packets, not bytes. WiFu is more realistic on this point. Another small difference is that when detecting loss WiFu TCP sets the slow start threshold to one-half of the number of bytes currently in flight [17], while ns-2 TCP sets it to $0.5 * min(congestionwindow, receivewindow)$ [15].

The other dissimilarities are due to the deterministic discrete event simulation in ns-2. 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.

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

TABLE I
MODIFIED LINUX KERNEL SETTINGS

Furthermore, the ns-2 TCP traces run for a fixed time period (approximately six seconds) while WiFu TCP sends a fixed amount of data (200K) as fast as it can.

Fall and Floyd [23] did further research examining scenarios of two, three, and four specific dropped packets. We have replicated these experiments WiFu, and WiFu TCP is nearly identical to ns-2 in each case. The only differences are those explained above.

V. PERFORMANCE EVALUATION

To evaluate the performance of WiFu, we conduct experiments using both wireless and wired networks. Though our own research focus is on wireless networks, our experiments with a wired network provide an interesting test of WiFu's capabilities and indicate that WiFu will continue to be useful as wireless speeds increase.

We use the mesh testbed at BYU. These machines have an Intel Pentium 4 2.4 or 3.2 Ghz CPU with 767 or 1024 MB of RAM, and they run Ubuntu 10.04 with the 2.6.32 Linux kernel. Each machine has an IEEE 802.11a/b/g wireless card installed, plus an Ethernet card that is configurable to 10, 100, or 1000 Mbps. When we run wireless experiments, the radios in the other mesh nodes are turned off.

A. WiFu Transport

To assess *WiFu Transport's* performance, we compare our implementation of TCP Tahoe in WiFu with the Linux kernel implementation of TCP Reno. We are unable to compare directly with Tahoe in Linux because recent kernels no longer contain an implementation of TCP Tahoe. To make as accurate a comparison as possible, we adjust kernel settings to convert the default choice of Cubic into TCP Reno instead, which provides a close match to Tahoe functionality. Table I shows the settings we changed to switch to Reno.

Our first experiment compares the goodput of the *WiFu Transport* Tahoe implementation with the kernel's Reno implementation on a one-hop to three-hop path through our wireless mesh testbed. This experiment uses IEEE 802.11a, with a maximum transmission power of 17 dBm and a fixed rate of 24 Mbps. We transfer a 1 MB file, with 50 repetitions, alternating between WiFu and the kernel.

Table II shows that the performance of WiFu is comparable the Linux kernel, with a very small (3%) difference over paths with two or three hops. This difference is likely attributable to

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 II
WIRELESS MEDIAN GOODPUTS, MBPS

Rate	WiFu TCP	Kernel TCP	Ratio
100 KB			
10	9.555	9.564	0.999
100	93.880	95.654	0.981
1 MB			
10	9.429	9.429	1.000
100	94.184	94.298	0.999
10 MB			
10	9.416	9.416	1.000
100	94.139	94.160	1.000

TABLE III
WIRED MEDIAN GOODPUTS, MBPS

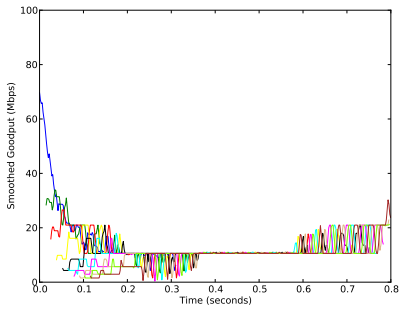
the differences between Tahoe and Reno; Reno is a little more aggressive when only a single packet is lost at a time. We will be implementing Reno in WiFu to verify this conclusion.

Our second experiment compares the goodput of *WiFu Transport* with the Linux kernel on a one-hop wired connection at several different Ethernet speeds. Our motivation for this experiment is to investigate how fast WiFu can go, without the limiting speeds of a wireless network. For this and following wired experiments we use newer machines with an Intel Core 2 Duo 3.16 Ghz processor and 4 GB of RAM. We transfer files of various sizes, alternating between WiFu and the kernel, and repeat each transfer 50 times.

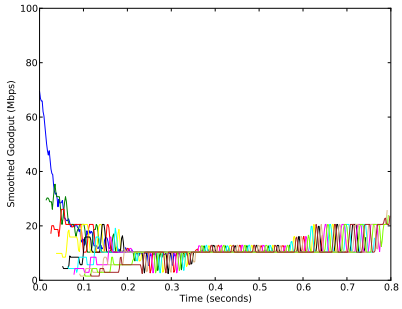
Table III shows that WiFu is able to achieve a median goodput that is equal to the kernel for both a 10 Mbps link and a 100 Mbps link, for large files. WiFu is 2% slower than the kernel for a small file on a 100 Mbps link. Currently, the maximum speed we have recorded for WiFu on a Gbps link is 412 Mbps. Clearly, there is a performance hit for running a user-space transport protocol and socket interface for high speed links. We are continuing to optimize our implementation to push the limits of a user-space implementation.

Our third experiment compares the performance of *WiFu Transport* and the kernel when each has multiple TCP flows. We vary the number of simultaneous connections from 2 to 10, and show representative results for the case where there are 10 competing flows. Each flow sends 1 MB of data over a 100 Mbps Ethernet link; we use a wired link here to avoid the confounding unfairness that can happen due to the 802.11 MAC. We stagger the start time of each thread by approximately 10 milliseconds.

Figure 5 shows that WiFu handles multiple competing flows nearly identically to the kernel. This further validates that our WiFu TCP implementation is correct. In these graphs, the goodput of each TCP flow is smoothed over time, using a window of 50 milliseconds that slides every two milliseconds.



(a) WiFu TCP



(b) Kernel TCP

Fig. 5. Instantaneous Goodput with 10 Flows

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

TABLE IV
WIRED MULTIPLE FLOWS MEDIAN JAIN'S FAIRNESS INDEX

Note that the kernel shows more variance between the 0.35 and 0.55 second time periods than WiFu TCP, but this is a minor difference.

We also measure the fairness of WiFu for multiple flows that start at the same time. Table IV shows the median of Jain's fairness index for each set of experiments. This shows that WiFu TCP is as fair as the kernel.

B. WiFu Core

To examine the performance of *WiFu Core*, we configure a basic handler that simply tells the kernel to accept every packet; this tests the overhead of the basic architecture without any additional packet handling.

We examine the performance of *WiFu Core* over the same point-to-point connection used in the second experiment of *WiFu Transport*, except we use the default kernel settings rather than modifying them to be more like TCP Reno. To show the high performance of *WiFu Core*, we configure the Ethernet card to 1000 Mbps with an MTU of 1500 bytes. Furthermore, we intercept all packets in the input and output hooks on *both* the sender and the receiver. This means that data packets and acknowledgements are brought into user space

twice: once when it is sent and once when it is received. We use *iperf* version 2.0.4 with default settings, which sends data over a TCP connection as fast as it can for 10 seconds. We run 50 iterations with *WiFu Core* turned on and 50 iterations with *WiFu Core* turned off.

Our results show that with *WiFu Core* enabled, the median data transfer speed is 933 Mbps and the maximum is 935 Mbps. With *WiFu Core* turned off, the median and maximum data transfer speed is 941 Mbps. This means that the overhead of *WiFu Core* on a gigabit connection is less than one percent. Due to limited buffer space in the kernel, `ENOBUFS` errors may occur with `netfilter`. The sender experienced an average of 59.34 `ENOBUFS` errors per data transfer over the 50 iterations, while the receiver had 0.16. As computer speeds continue to increase, we expect this number to drop. Overall, our results show that *WiFu Core* can intercept packets and bring them into user space extremely fast. For wireless networks, where bandwidth is generally more constrained, *WiFu Core's* overhead is insignificant.

VI. Wireless Transport

To demonstrate the utility of WiFu, we have implemented and tested several transport protocols. Our goal is to determine how rate control can provide better fairness and performance for transport connections in wireless mesh networks. We begin by exploring two TCP extensions, called Adaptive Pacing [8] and Delayed ACKs [9]. We then consider a hybrid transport protocol we call *TP-*α, which removes TCP's congestion control algorithm and replaces it with a rate-based controller.

Our experience with each of these protocols demonstrates how simple it is to use WiFu for experimental research. A single developer was able to write the code for the two TCP extensions in only a few weeks because of the extensive code reuse available with a decomposed TCP. It was likewise very easy to retain TCP's connection management and reliable transfer components, and combine them with an entirely new congestion controller. This is important because numerous approaches have been proposed to significantly improve TCP performance in a mesh network based on promising simulation results, but relatively few have been implemented and tested experimentally. It is well-known that simulators use many simplifications to model radio propagation, such as a flat, level world and perfectly circular transmission areas. [24], [25], [26]. Thus, it is important to validate simulation results using experiments on testbeds.

Though we have done extensive testing on numerous topologies, we present here only selected results to illustrate the work we have been able to do with WiFu. In particular, we show results of testing the TCP extensions on a 9-hop wireless path in our mesh testbed. All radios are using IEEE 802.11a with MAC-layer retries set to zero, RTS/CTS turned off, and static routing. The transmission power for the network adapter is set to 6 dBm and the transmission rate is 24 Mbps. Note that the lower power enables us to construct a longer chain, similar to those used in the original simulations, while still maintaining near zero loss on each link when used in isolation. In each experiment we transfer a 1 MB file along each connection, experiments for different protocols are interleaved, and each protocol is run 10 times.

A. Adaptive Pacing

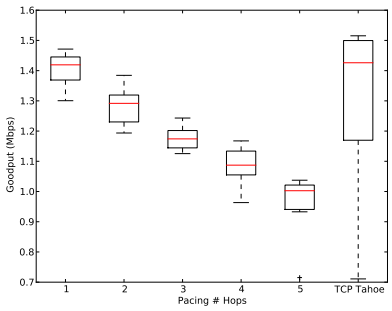
Adaptive Pacing is based on the observation that window-based congestion control leads to burstiness, which causes increased contention and loss in wireless networks. Because TCP reduces its rate in response to loss events, this leads to performance degradation. With Adaptive Pacing, a TCP sender uses RTT measurements to estimate the 4-hop propagation delay, and then derives a sending rate based on this delay. A rate limiter restricts TCP to pacing out packets at this rate. The idea is to transmit a new packet after the previous one has travelled four hops, in order to minimize contention. Simulations using ns-2 [22] show that Adaptive Pacing results in up to an 84% increase in goodput over TCP NewReno for a single flow.

Goodput and fairness results for Adaptive Pacing on the long wireless path are shown in Figure 6. We vary the pacing metric from 1 to 5 hops, so that we can explore the effectiveness of pacing in general, rather than limiting our experiments to just the proposed 4-hop delay metric. Figure 6(a) shows that when a single flow is active on the path, Adaptive Pacing provides little gain in goodput. In fact, goodput is generally much lower than an unmodified TCP Tahoe, and pacing only helps when using a 1-hop delay. The picture changes when five flows are active on the path, as shown in Figure 6(b). In this case, pacing does improve goodput as compared to Tahoe. More importantly, pacing significantly improves Jain's fairness index [27] among the competing flows, increasing it from 0.87 to 0.97 and higher. Thus our implementation experience indicates that the benefits of Adaptive Pacing are only apparent when multiple flows are active.

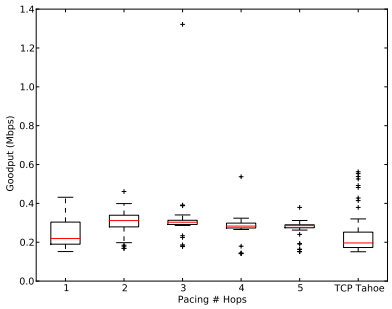
B. Delayed ACKs

TCP with Delayed ACKs is based on the observation that acknowledgements contribute to a high amount of transmission overhead in wireless networks, particularly as the bit rate of the network increases. TCP already combines up to two ACKs in certain circumstances, and the Delayed ACKs work proposes combining up to four ACKs [9]. The number of ACKs to combine at any time varies dynamically, beginning with no combination early in a connection and then building up to increasing amounts of aggregation as the connection continues. The idea is to prevent TCP from getting stuck with too few ACKs when the congestion window is small and combining more ACKs when the window is large enough. Simulations using ns-2 show that Delayed ACKs improves TCP goodput by 20% to 40%.

Goodput and results for Delayed ACKs on the long wireless path are shown in Figure 7. With Delayed ACKs, several thresholds are used to determine when to start combining ACKs, based on how many bytes have been transferred so far. We use thresholds of 1 KB, 10 KB, and 100 KB to switch to combining 2, 3, and 4 ACKs, respectively. TCP will wait

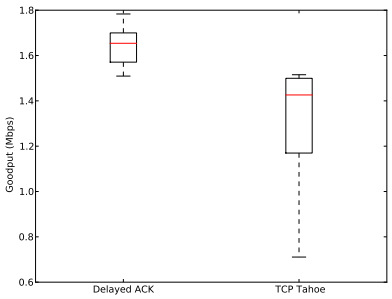


(a) Goodput for 1 flow

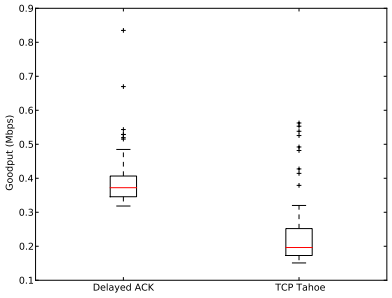


(b) Goodput for 5 flows

Fig. 6. Goodput for TCP with Adaptive Pacing on a 9 hop wireless path



(a) 1 flow



(b) 5 flows

Fig. 7. Goodput comparison of TCP and Delayed ACKS on a 9 hop wireless path

at most 0.1 seconds when combining ACKs. Since we use a 1 MB file transfer for our experiments, this means that 90% of each transfer is performed combining the maximum number of ACKs. When a single flow is active on the path, Delayed ACKs achieves nearly 20% higher goodput as compared to our Tahoe implementation. With five active flows, the aggregate goodput of the flows is 75% higher with Delayed ACKs. There is no substantive change in fairness in this case, with Delayed ACKs scoring 0.93 on Jain's fairness index, and Tahoe score 0.938. The gains for multiple flows sharing a path are much higher than previously reported, as prior work focused on a single flow.

C. TP-α

Our development of *TP-*α is inspired by ATP, a transport protocol designed to overcome the shortcomings of TCP in a mobile ad hoc network [28]. ATP uses a rate-based congestion control algorithm, where the rate is based on driver measurements of the average transmission and queueing delay along the path. We have previously implemented and tested ATP, showing how it needs several modifications to meet its performance goals [29].

We want to test the feasibility of delay-driven, rate-based congestion control for wireless networks, separately from the other modifications ATP makes to the connection management and reliability portions of TCP. To create *TP-*α, we used a combination of *WiFu Core* to provide cross-layer delay measurements and *WiFu Transport* to provide a mix of TCP and ATP functionality. First, we modified the `ath5k` driver to measure delay for each packet it forwards, then we wrote a handler in *WiFu Core* to read this delay from the */proc* file system and store it in a shim header as each packet is forwarded. Next, we modified the TCP Tahoe reliability component in *WiFu Transport* to report this delay back to the sender in each ACK. Finally, we wrote a new congestion control component in *WiFu Transport* to use this reported delay to calculate the transmission rate of the flow and to pace out packets at this rate.

TP-α is still undergoing significant development. In preliminary experiments we have used a fixed rate for *TP-*α, after careful testing to determine a rate that minimizes the delay reported by the modified `ath5k` driver. In some cases, we are able to obtain as much as a 50% goodput gain. This is encouraging, as it indicates that WiFu can be used to develop experimental transport protocols, and that there is room to improve significantly on the performance of TCP.

VII. Related Work

A number of projects implement a network stack in user space, making it easier to experiment with modifications to TCP – among them Alpine [4], Daytona [5], and Minet [6]. All of these projects use the Packet Capture library [30] to capture and inject packets directly from and to network interfaces, along with *iptables* or similar software to prevent the kernel from handling packets intended for user-space processing. While these projects all build TCP implementations at the user level, our goals differ significantly. We are building a more

general framework for transport protocols that supports both TCP and non-TCP variants. This desire for greater flexibility significantly impacts how WiFu is built. At the same time, we want to provide good performance. Alpine, for example, has a one millisecond sleep timer while polling for packets, so with a 1500 byte MTU it can support a maximum rate of only 12 Mbps. Finally, most of this source code is no longer available.

More recent related work includes the FINS framework [31] and CTP [7]. FINS moves the entire network stack into user space to facilitate experimental research. This is complementary to our work, which is not a direct port and supports greater flexibility in transport protocol design. FINS is also relatively new and does not yet have published results. CTP composes transport protocols out of fine-grained microprotocols, with a focus on grid computing. CTP was optimized to provide performance up to 900 Mbps by minimizing copies and providing more efficient event handling [12]. However, these speeds are for a minimal configuration, not a full version of TCP, and CTP does not provide for cross-layer interactions at intermediate nodes. The UDT/CCC library provides a convenient event-driven and object-oriented framework for congestion control algorithm design [32], but is more limited in scope than WiFu, which allows experimentation with all aspects of transport protocol design. Finally, the Click project [11] is similar in spirit to WiFu, but focuses on designing a software router, rather than a framework for transport protocols.

VIII. Conclusion

WiFu provides a flexible environment for user-space development of experimental transport protocols, with an emphasis on protocol decomposition. The performance of WiFu is close to that of the kernel for wireless networks and for Ethernet links up to 100 Mbps. We have used WiFu to develop a new implementation of TCP Tahoe that is divided into components, as well as extensions to TCP and a new transport protocol that uses cross-layer measurements to control the transmission rate. Our experiences demonstrate that WiFu has largely met its design goals of flexibility, code reuse, performance and scalability.

We are releasing WiFu as an open source project and will continue to make improvements. One area we will focus on is improving the overall goodput that *WiFu Transport* is able to achieve, past the 412 Mbps mark. Reducing the number of data copies and the number of events throughout the architecture may increase performance. In addition to performance, we are investigating pluggable protocols, where a new transport protocol could be dynamically loaded in, rather than compiled into the framework. Finally, we are actively using WiFu to develop new transport protocols based on optimal rate control.

References

[1] R. de Oliveira and T. Braun, "A dynamic adaptive acknowledgment strategy for TCP over multihop wireless networks," in *IEEE INFOCOM*, 2005.

[2] Z. Fu, P. Zerfos, H. Luo, S. Lu, L. Zhang, and M. Gerla, "The impact of multihop wireless channel on TCP throughput and loss," in *IEEE INFOCOM*, 2003.

[3] T. Nandagopal, T.-E. Kim, X. Gao, and V. Bharghavan, "Achieving MAC layer fairness in wireless packet networks," in *ACM MobiCom*, New York, NY, USA, 2000, pp. 87–98.

[4] D. Ely, S. Savage, and D. Wetherall, "Alpine: a user-level infrastructure for network protocol development," in *USITS*, Berkeley, CA, USA, 2001, pp. 15–15.

[5] P. Pradhan, S. Kandula, W. Xu, A. Shaikh, and E. Nahum, "Daytona: A user-level TCP stack," unpublished manuscript.

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

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

[8] S. M. ElRakabawy, A. Klemm, and C. Lindemann, "TCP with adaptive pacing for multihop wireless networks," in *ACM MobiHoc*, 2005.

[9] E. Altman and T. Jiménez, "Novel delayed ACK techniques for improving TCP performance in multihop wireless networks," *PersonalWireless Communications*, pp. 237–250, 2003.

[10] R. Johnson, E. Gamma, R. Helm, and J. Vlissides, "Design patterns: Elements of reusable object-oriented software," *Addison-Wesley*, vol. 1, pp. 1–2, 1995.

[11] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek, "The Click modular router," *ACM Transactions on Computer Systems*, vol. 18, no. 3, pp. 263–297, August 2000.

[12] R. Wu, A. Chien, M. Hiltunen, R. Schlichting, and S. Sen, "A high performance configurable transport protocol for grid computing," in *IEEE International Symposium on Cluster Computing and the Grid*, vol. 2, may 2005, pp. 1117 – 1125 Vol. 2.

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

[14] J. Postel, "RFC 793: Transmission control protocol," 1981.

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

[16] R. Braden, "RFC 1122: Communication layers," 1989.

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

[18] V. Paxson and M. Allman, "RFC 2988: Computing TCP's transmission timer," 2000.

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

[20] J. Nagle, "RFC 896: Congestion control in IP/TCP internetworks," 1984.

[21] J. Postel, "RFC 879: The TCP maximum segment size and related topics," 1983.

[22] S. McCanne and S. Floyd, *ns-2 network simulator*, http://www.isi.edu/nsnam/ns/.

[23] 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.

[24] K. M. Reineck, "Evaluation and comparison of network simulation tools," Master Thesis, Department of Computer Science, University of Applied Sciences Bonn-Rhein-Sieg, August 2008.

[25] J. Heidemann, N. Bulusu, J. Elson, C. Intanagonwiwat, K. Lan, Y. Xu, W. Ye, D. Estrin, and R. Govindan, "Effects of detail in wireless network simulation," *Proceedings of the SCS Multiconference on Distributed Simulation*, pp. 3–11, 2001.

[26] D. Kotz, C. Newport, R. S. Gray, J. Liu, Y. Yuan, and C. Elliott, "Experimental evaluation of wireless simulation assumptions," in *MSWiM*, 2004, pp. 78–82.

[27] R. Jain, D.-M. Chiu, and W. Hawe, "A quantitative measure of fairness and discrimination for resource allocation in shared computer systems," *Computing Research Repository*, 1998.

[28] K. Sundaresan, V. Anantharaman, H.-Y. Hsieh, and R. Sivakumar, "ATP: a reliable transport protocol for ad-hoc networks," in *ACM MobiHoc*, 2003.

[29] X. Zhang, R. Buck, and D. Zappala, "Experimental performance evaluation of ATP in a wireless mesh network," in *IEEE MASS*, 2011.

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

[31] 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 *WiNTECH*, 2011.

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