Deploying Safe User-Level Network Services with icTCP

Haryadi S. Gunawi, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau

Computer Sciences Department
University of Wisconsin, Madison
{*haryadi, dusseau, remzi*}@*cs.wisc.edu*

Abstract

We present icTCP, an "information and control" TCP implementation that exposes key pieces of internal TCP state and allows certain TCP variables to be set in a safe fashion. The primary benefit of icTCP is that it enables a variety of TCP extensions to be implemented at user-level while ensuring that extensions are TCP-friendly. We demonstrate the utility of icTCP through a collection of case studies. We show that by exposing information and safe control of the TCP congestion window, we can readily implement user-level versions of TCP Vegas, TCP Nice, and the Congestion Manager; we show how user-level libraries can safely control the duplicate acknowledgment threshold to make TCP more robust to packet reordering or more appropriate for wireless LANs; we also show how the retransmission timeout value can be adjusted dynamically. Finally, we find that converting a stock TCP implementation into icTCP is relatively straightforward; our prototype requires approximately 300 lines of new kernel code.

1 Introduction

Years of networking research have suggested a vast number of modifications to the standard TCP/IP protocol stack [3, 8, 11, 13, 14, 23, 27, 31, 40, 47, 50, 52, 57]. Unfortunately, while some proposals are eventually adopted, many suggested modifications to the TCP stack do not become widely deployed [44].

In this paper, we address the problem of deployment by proposing a small but enabling change to the network stack found in modern operating systems. Specifically, we introduce *icTCP* (pronounced "I See TCP"), a slightly modified in-kernel TCP stack that exports key pieces of state information and provides safe control to user-level libraries. By exposing state and safe control over TCP connections, icTCP enables a broad range of interesting and important network services to be built at user-level.

User-level services built on icTCP are more *deployable* than the same services implemented within the OS TCP stack: new services can be packaged as libraries and easily downloaded by interested parties. This approach is also inherently *flexible*: developers can tailor them to the exact needs of their applications. Finally, these extensions are *composable*: library services can be used to build more powerful functionality in a lego-like fashion. In general, icTCP facilitates the development of many services that otherwise would have to reside within the OS.

One key advantage of icTCP compared to other approaches for upgrading network protocols [41, 44] is the *simplicity* of implementing the icTCP framework on a new platform. Simplicity is a virtue for two reasons. First, given that icTCP leverages the entire existing TCP stack, it is relatively simple to convert a traditional TCP implementation to icTCP; our Linux-based implementation requires approximately 300 new lines of code. Second, the small amount of code change reduces the chances of introducing new bugs into the protocol; previous TCP modifications often do not have this property [43, 45].

Another advantage of icTCP is the *safe* manner in which it provides new user-level control. Safety is an issue any time users are allowed to modify the behavior of the OS [48]. With icTCP, users are allowed to control only a set of *limited virtual* TCP variables (*e.g.*, cwnd, dupthresh, and RTO). Since users cannot download arbitrary code, OS safety is not a concern. The remaining concern is *network safety*: can applications implement TCP extensions that are not friendly to competing flows [38]? By building on top of the extant TCP Reno stack and by restricting virtual variables to a safe range of values, icTCP ensures that extensions are no more aggressive than TCP Reno and thus are friendly.

In addition to providing simplicity and safeness, a framework such as icTCP must address three additional questions. First, are the overheads of implementing variants of TCP with icTCP reasonable? Our measurements show that services built on icTCP scale well and incur minimal CPU overhead when they use appropriate icTCP waiting mechanisms.

Second, can a wide range of new functionality be implemented using this conservative approach? We demonstrate the utility of icTCP by implementing six extensions of TCP. In the first set of case studies, we focus on modifications that alter the behavior of the transport with regard to congestion: TCP Vegas [14], TCP Nice [52], and Congestion Manager (CM) [8]. In our second set, we focus on TCP modifications that behave differently in the presence of duplicate acknowledgments: we build a reodering-robust (RR) extension that does not misinterpret packet reordering as packet loss [11, 57] and an extension with efficient fast retransmit (EFR) [50]. In our third set, we explore TCP Eifel [36] which adjusts the retransmit timeout value.

Finally, can these services be developed easily within the framework? We show that the amount of code required to build these extensions as user-level services on icTCP is similar to the original, native implementations.

The rest of this paper is structured as follows. In Section 2 we compare icTCP to related work on extensible network services. In Section 3 we present the design of icTCP and in Section 4 we describe our methodology. In Section 5 we evaluate five important aspects of icTCP: the simplicity of implementing icTCP for a new platform, the network safety ensured of new user-level extensions, the computational overheads, the range of TCP extensions that can be supported, and the complexity of developing those extensions. We conclude in Section 6.

2 Related Work

In this section, we compare icTCP to other approaches that provide networking extensibility.

Upgrading TCP: Four recent projects have proposed frameworks for providing limited extensions for transport protocols; that is, they allow protocols such as TCP to evolve and improve, while still ensuring safety and TCP friendliness. We compare icTCP to these proposals.

Mogul *et al.* [41] propose that applications can "get" and more radically "set" TCP state. In terms of getting TCP state, icTCP is similar to this proposal. The greater philosophical difference arises in how internal TCP state is set. Mogul *et al.* wish to allow arbitrary state setting and suggest that safety can be provided either with a cryptographic signature of previously exported state or by restricting this ability to the super-user. However, icTCP is more conservative, allowing applications to alter parameters only in a restricted fashion. The trade-off is that icTCP can guarantee that new network services are well behaved, but Mogul *et al.*'s approach is likely to enable a broader range of services (*e.g.*, session migration).

The Web100 and Net100 projects [39] are developing a management interface for TCP. Similar to the information component of icTCP, Web100 instruments TCP to export a variety of per-connection statistics; however, Web100 does not propose exporting as detailed information as icTCP (*e.g.*, Web100 does not export timestamps for every message and acknowledgment). The TCP-tuning daemon within Net100 is similar to the control component of icTCP; this daemon observes TCP statistics and responds by setting TCP parameters [18]. The key difference from icTCP is that Net100 does not propose allowing a complete set of variables to be controlled and does not ensure network safety. Furthermore, Net100 appears suitable only for tuning parameters that do not need to be set frequently; icTCP can frequently adjust in-kernel variables because it provides per-message statistics as well as the ability to block until various in-kernel events occur.

STP [44] also addresses the problem of TCP deployment. STP enables communicating end hosts to remotely upgrade the other's protocol stack. With STP, the authors show that a broad range of TCP extensions can be deployed. We emphasize two major differences between STP and icTCP. First, STP requires more invasive changes to the kernel to support safe downloading of extension-specific code; support for in-kernel extensibility is fraught with difficulty [48]. In contrast, icTCP makes minimal changes to the kernel. Second, STP requires additional machinery to ensure TCP friendliness; icTCP guarantees friendliness by its very design. Thus, STP is a more powerful framework for TCP extensions, but icTCP can be provided more easily and safely.

Finally, the information component of icTCP is derived from INFOTCP, proposed as part of the infokernel [7]; this previous work showed that INFOTCP enables user-level services to indirectly control the TCP congestion window, *cwnd*. We believe that icTCP improves on INFOTCP in three main ways. First, icTCP exposes information from a more complete set of TCP variables. Second, icTCP allows services to directly set *cwnd* inside of TCP; thus, applications do not need to perform extra buffering nor incur as many sleep/wake events. Finally, icTCP allows TCP variables other than *cwnd* to be controlled. Thus, icTCP not only allows more TCP extensions to be implemented, but is also more efficient and accurate.

User-Level TCP: Researchers have found it useful to move portions of the conventional network stack to user-level [19, 20, 46]. User-level TCP can simplify protocol development in the same way as icTCP. However, a user-level TCP implementation typically struggles with performance, due to extra buffering or context switching or both; further, there is no assurance of network safety.

Application-Specific Networking: A large body of research has investigated how to provide extensibility of network services [22, 28, 37, 51, 53, 54]. These projects allow network protocols to be more specialized to applications than does icTCP, and thus may improve performance more dramatically. However, these approaches tend to require more radical restructuring of the OS or networking stack and do not guarantee TCP friendliness.

Protocol Languages and Architectures: Network languages [1, 35] and structured TCP implementations [10] simplify the development of network protocols. Given the ability to replace or specialize modules, it is

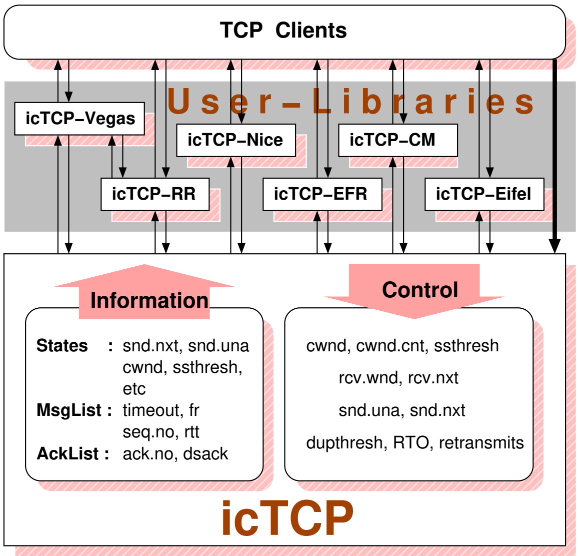


Figure 1: **icTCP Architecture.** *The diagram shows the icTCP architecture. At the base of the stack is icTCP, a slightly modified TCP stack that exports information and limited control. On top of icTCP, we have built a number of user-level libraries that implement various pieces of functionality suggested by the literature. The libraries can be composed (where applicable), thus enabling the construction of more powerful services in a plug-and-play fashion. Applications sit at the top of the stack and can choose the libraries that match their needs or directly use the kernel transport.*

generally easier to extend existing TCP implementations.

3 icTCP Design

The icTCP framework exposes information and provides control over key parameters in the TCP protocol implementation. In this section, we give a high-level overview of how user-level network services are deployed with icTCP. We then describe the classes of information and control exported by icTCP.

3.1 System Architecture

Figure 1 presents a schematic of the icTCP framework. As illustrated, user-level libraries implementing variants of TCP are built on top of icTCP. The user-level libraries can be transparently used by applications with standard interfaces. Different TCP connections can use different icTCP libraries. The design of icTCP is such that only the sending side needs to have icTCP deployed; receivers can be running icTCP or an unmodified kernel stack.

To simplify the implementation, icTCP uses the BSD socket interface, exporting information and providing control with a few new socket options. Although this approach minimized our implementation work, it imposes unnecessary run-time overhead: obtaining state requires a copy from the kernel to user space. Our evaluation shows that user-level network services that naively poll icTCP frequently for state information can incur a significant increase in CPU overhead.

To minimize this overhead, icTCP provides both a polling and an interrupt-based interface. Given that most TCP variables are updated only when an acknowledgment arrives or at the end of a round (*i.e.*, when one round-trip time has elapsed), applications can receive an interrupt for either condition. In our case studies, the icTCP user-level libraries are structured to use two threads; one thread injects packets into the kernel while the other performs sleep/wait and get/set operations.

3.2 Information

The first goal of icTCP is to expose information that is traditionally internal to TCP. The challenge is to determine which information should be exposed: if too little information is exposed, it may not be possible to build interesting extensions; if too much information is exposed, then future kernel implementations of TCP may be constrained by an undesirable, expanded interface.

Given that TCP implementations are constrained to adhere to the TCP specification [29], many internal variables are already specified and required. Therefore, icTCP explicitly exports all variables that are part of the TCP specification, such as the next sequence number to be sent (*snd.nxt*), the oldest unacknowledged sequence number (*snd.una*), the congestion window (*cwnd*), and the slow start threshold (*ssthresh*). Exposing this information from any TCP implementation should be straightforward.

However, we have found that for more interesting services, access to more information is needed. For example, libraries such as icTCP-Nice and icTCP-RR must examine information about a particular message. Therefore, icTCP exposes "standard" information about each packet. A *message list* provides a history of recent packets, reporting for each packet its sequence number, round-trip time, and whether it is being sent for a time-out or a fast retransmit. An *ack list* provides a history of recent acknowledgments, recording for each packet its acknowledgment number and type (*e.g.*, normal ACK, duplicate ACK, SACK, or DSACK).

Exposing per-packet and per-ack information may not be trivial for those TCP implementations where it does not already exist. Given that TCP Reno does not track the round-trip time of each packet, we add a high resolution timer to icTCP to record this information. An additional complexity is that recording new per-message information requires additional memory; therefore, icTCP creates these lists only when enabled by user-level services.

3.3 Control

The second goal of icTCP is to allow variables that are internal to TCP to be externally set in a safe manner. A new challenge is to determine not only which variables can be modified, but also to what values, while still ensuring that the resulting behavior is TCP-friendly. Our philosophy is

Variable	Description	Safe Range	Example usage
cwnd	Congestion window	$0 \leq v \leq x$	Limit number of sent packets
cwnd.cnt	Linear cwnd increase	$0 \leq v \leq x$	Increase cwnd less aggressively
ssthresh	Slow start threshold	$1 \leq v \leq x$	Move to SS from CA
rcv.wnd	Receive window size	$0 \leq v \leq x$	Reject packet; limit sender
rcv.nxt	Next expected seq num	$x \leq v \leq x +$ vrcv.wnd	Reject packet; limit sender
snd.nxt	Next seq num to send	vsnd.una $\leq v \leq x$	Reject ack; enter SS
snd.una	Oldest unacked seq num	$x \leq v \leq$ vsnd.nxt	Reject ack; enter FRFR
dupthresh	Duplicate threshold	$1 \leq v \leq$ vcwnd	Enter FRFR
RTO	Retransmission timeout	exp.backoff $*$ (srtt $+$ rttvar) $\leq v$	Enter SS
retransmits	Number of consecutive timeouts	$0 \leq v \leq$ threshold	Postpone killing connection

Table 1: **Safe Setting of TCP Variables.** *The table lists the 10 TCP variables which can be set in icTCP. We specify the range each variable can be safely set while ensuring that the result is less aggressive than the baseline TCP implementation. We also give an example usage or some intuition on why it is useful to control this variable. Notation: x refers to TCP's original copy of the variable and v refers to the new virtual copy being set; SS is used for slow start, CA for congestion avoidance, and FRFR for fast retransmit/fast recovery; finally, srtt, rttvar, and exp.backoff represent smoothed round-trip time, round-trip time variance, and the RTO exponential backoff, respectively.*

that icTCP must be conservative: control is only allowed when it is known to not cause aggressive transmission.

The basic idea is that for each variable of interest, icTCP adds a new *limited virtual variable*. Our terminology is as follows: for a TCP variable with the original name *foo*, we introduce a limited virtual variable with the name *vfoo*. However, when the meaning is clear, we simply use the original name. We restrict the range of values that the virtual variable is allowed to cover so that the resulting TCP behavior is friendly; that is, we ensure that the new TCP actions are no more aggressive than those of the original TCP implementation. Given that the acceptable range for a variable is a function of other fluctuating TCP variables, it is not possible to check at call time that the user has specified a valid value and reject invalid settings. Instead, icTCP accepts all settings and coerces the virtual variable into a valid range. For example, the safe range for the virtual congestion window, *vcwnd*, is $0 \leq$ vcwnd $\leq$ cwnd. Therefore, if *vcwnd* rises above *cwnd* the value of *cwnd* is used instead.

Converting a variable to a virtual variable within the icTCP stack is not as trivial as it may appear; one cannot simply replace all instances of the original variable with the new virtual one. One must ensure that the virtual value is never used to change the original variable. The simplest case is the statement cwnd $=$ cwnd$+1$, which clearly cannot be replaced with cwnd $=$ vcwnd $+\ 1$. More complex cases of control flow currently require careful manual inspection. Therefore, we limit the extent to which the original variable is replaced with the virtual variable.

Given that our foremost goal with icTCP is to ensure that icTCP cannot be used to create aggressive flows, we are conservative in the virtual variables we introduce. Although it would be interesting to allow all TCP variables to be set, the current implementation of icTCP only allows control of ten variables that we are convinced can be safely set from our analysis of the Linux TCP implementation. We do not introduce virtual variables when the original variable can already be set through other interfaces (*e.g.*, sysctl of tcp_retries1 or user_mss) or when they can be approximated in other ways (*e.g.*, we set RTO instead of srtt, mdev, rttvar, or mrtt). We do not claim that these ten variables represent the complete collection of settable values, but that they do form a useful set. These ten variables and their safe ranges are summarized in Table 1. We briefly discuss why the specified range of values is safe for each icTCP variable.

The first three variables (*i.e.*, cwnd, cwnd.cnt, and ssthresh) have the property that it is safe to strictly lower their value. In each case, the sender directly transmits less data, because either its congestion window is smaller (*i.e.*, cwnd and cwnd.cnt) or slow-start is entered instead of congestion avoidance (*i.e.*, ssthresh).

The next set of variables determine which packets or acknowledgments are accepted; the constraints on these variables are more complex. On the receiver, a packet is accepted if its sequence number falls inside the receive window (*i.e.*, between rcv.nxt and rcv.nxt + rcv.wnd); thus, increasing rcv.nxt or decreasing rcv.wnd has the effect of rejecting incoming packets and forces the sender to reduce its sending rate. On the sender, an acknowledgment is processed if its sequence number is between snd.una and snd.nxt; therefore, increasing snd.una or decreasing snd.nxt causes the sender to discard acks and again reduce its sending rate. In each case, modifying these values has the effect of dropping additional packets; thus, TCP backs-off appropriately.

The final set of variables (*i.e.*, dupthresh, RTO, and retransmits) control thresholds and timeouts; these variables can be set independently of the original values. For example, both increasing and decreasing dupthresh is believed to be safe [57]. Changing these values can increase the amount of traffic, but does not allow the sender to transmit new packets or to increase its congestion window.

Information	LOC	Control	LOC
States	25	cwnd	15
Message List	33	dupthresh	28
Ack List	41	RTO	13
High-resolution RTT	12	ssthresh	19
Wakeup events	50	cwnd_cnt	14
		retransmits	6
		rcv_nxt	20
		rcv_wnd	14
		snd_una	12
		snd_nxt	14
Info Total	**161**	**Control Total**	**155**
icTCP Total			**316**

Table 2: **Simplicity of Environment.** *The table reports the number of C statements (counted with the number of semicolons) needed to implement the current prototype of icTCP within Linux 2.4.*

4 Methodology

Our prototype of icTCP is implemented in the Linux 2.4.18 kernel. Our experiments are performed exclusively within the Netbed network emulation environment [56]. A single Netbed machine contains an 850 MHz Pentium 3 CPU with 512 MB of main memory and five Intel EtherExpress Pro 100Mb/s Ethernet ports. The sending endpoints run icTCP, whereas the receivers run stock Linux 2.4.18.

For almost all experiments, a dumbbell topology is used, with one or more senders, two routers interconnected by a (potential) bottleneck link, and one or more receivers. In some experiments, we use a modified Nist-Net [16] on the router nodes to emulate more complex behaviors such as packet reordering. In most experiments, we vary some combination of the bottleneck bandwidth, delay, or maximum queue size through the intermediate router nodes. Experiments are run multiple times (usually 30) and averages are reported; variance is low in those cases where it is not shown.

5 Evaluation

To evaluate whether or not icTCP is a reasonable framework for deploying TCP extensions at user-level, we answer five questions. First, how easily can an existing TCP implementation be converted to provide the information and safe control of icTCP? Second, does icTCP ensure that the resulting network flows are TCP friendly? Third, what are the computation overheads of deploying TCP extensions as user-level processes and how does icTCP scale? Fourth, what types of TCP extensions can be built and deployed with icTCP? Finally, how difficult is it to develop TCP extensions in this way? Note that we spend the bulk of the paper addressing the fourth question concerning the range of extensions that can be implemented and discussing the limitations of our approach.

```
// set internal TCP variables
tcp_setsockopt (option, val) {
    switch (option) {
    case TCP_USE_VCWND:
        use_vcwnd = val;
    case TCP_SET_VCWND:
        vcwnd = val;
    }
}

// check if data should be put on the wire
tcp_snd_test () {
    if (use_vcwnd)
        min_cwnd = min (vcwnd, cwnd);
    else
        min_cwnd = cwnd;
    // if okay to transmit
    if ((tcp_packets_in_flight < min_cwnd) &&
          /* ... other rules .... */)
        return 1;
    else
        return 0;
}
```

Figure 2: **In-kernel Modification.** *Adding vcwnd into the TCP stack requires few lines of code. icTCP applications set the virtual variables through the BSD setsockopt() interface. Based on the congestion window, tcp_snd_test checks if data should be put on the wire. We show that adding a virtual cwnd into the decision-making process is simple and straightforward: instead of using cwnd, icTCP uses the minimum of vcwnd and cwnd.*

5.1 Simplicity of Environment

We begin by addressing the question of how difficult it is to convert a TCP implementation to icTCP. Our initial version of icTCP has been implemented within Linux 2.4.18. Our experience is that implementing icTCP is fairly straightforward and requires adding few new lines of code. Table 2 shows that we added 316 C statements to TCP to create icTCP. While the number of statements added is not a perfect indicator of complexity, we believe that it does indicate how non-intrusive these modifications are. Figure 2 gives a partial example of how the *vcwnd* variable can be added to the icTCP stack.

5.2 Network Safety

We next investigate whether icTCP flows are TCP friendly. To perform this evaluation, we measure the throughput available to default TCP flows that are competing with icTCP flows. Our measurements show that icTCP is TCP friendly; as desired, the default TCP flows obtain at least as much bandwidth when competing with icTCP as when competing with other default TCP flows. We also show the need for constraining the values into a valid range within icTCP. To illustrate this need, we have created an unconstrained icTCP that allows virtual variables to be set to any value. When default TCP flows compete with unconstrained icTCP flows, the throughput

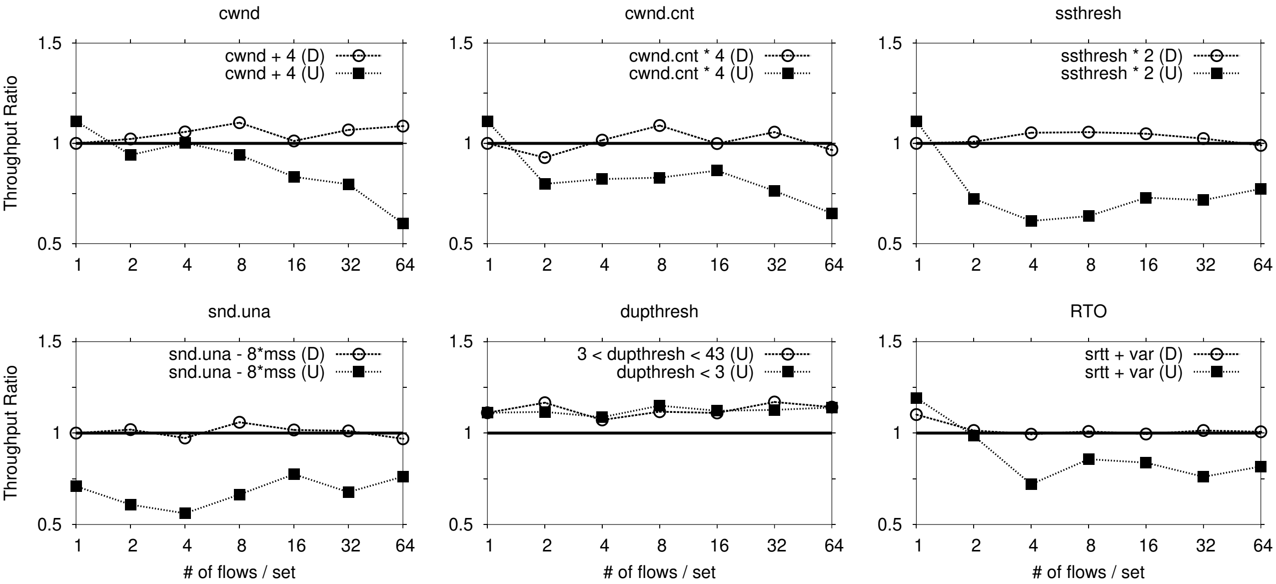


Figure 3: **Network Safety of icTCP.** *Each graph shows two lines: the first line ((D)=Default) uses the default icTCP that enforces parameters to values within their safe range; the second line ((U)=Unconstrained) uses icTCP that allows parameters to be set to any value. The* dupthresh *graph uses the unconstrained icTCP for both lines. The metric is the ratio of throughput achieved by the default TCP flows when competing with the icTCP flows versus when competing with other default TCP flows. Across the graphs, we vary which icTCP parameters are set; in each case, we set the variable to an unsafe value:* cwnd *to four packets larger,* cwnd.cnt *to four times larger,* ssthresh *to two times larger,* snd.una *to eight packets lower,* dupthresh *to random values below and above three (the default), and* RTO *remaining at an initial* srtt + rttvar *as packets are dropped. The topology used is a dumbbell with four senders and four receivers. For all experiments, except the RTO experiments, the bottleneck bandwidth is 100 Mbps with no delay; the RTO experiments use a bottleneck bandwidth of 2 Mbps with 15 percents drop rate.*

of the default TCP flows is reduced.

Our measurements are shown in Figure 3. Across graphs, we evaluate different icTCP parameters, explicitly setting each parameter to a value outside of its safe range. Along the x-axis of each graph, we increase the number of competing icTCP and TCP flows. Each graph shows two lines: one line has icTCP flows matching our proposal, in which virtual variables are limited to their safe range; the other line has unconstrained icTCP flows. Our metric is the ratio of throughput achieved by the default TCP flows when competing with the icTCP flows versus when competing with other default TCP flows. Thus, if the throughput ratio is around or above one, then the icTCP flows are friendly; if it is below one, then the icTCP flows are unfriendly.

The *cwnd*, *cwnd.cnt*, and *ssthresh* experiments show that these variables must be set within their safe range to ensure friendliness. As expected, icTCP flows that are not allowed to increase their congestion window beyond that of the default TCP remain TCP friendly. Unconstrained icTCP flows that allow larger congestion windows are overly aggressive; as a result, the competing TCP flows obtain less than their fair share of the bandwidth.

We next evaluate the variables that control which acknowledgments or packets are accepted. The behavior for *snd.una* is shown in the fourth graph. The *snd.una* variable represents the highest unacknolwedged packet. When the virtual *snd.una* is set below its safe range of the actual value, then unconstrained icTCP over-estimates the number of bytes acknowledged and increases the congestion window too aggressively. However, when icTCP correctly constrains *snd.una*, the flow remains friendly. The results for the other three variables (*i.e.*, *rcv.wnd*, *rcv.nxt*, and *snd.nxt*) are not shown. In these cases, the icTCP flows remain friendly, as desired, but the unconstrained icTCP flows can fail completely. For example, increasing the *rcv.wnd* variable beyond its safe range can cause the receive buffer to overflow.

The final two graphs explore the *dupthresh* and *RTO* thresholds. We do not experiment with the *retransmits* variable since it is only used to decide when a connection should be terminated. As expected for *dupthresh*, both decreasing and increasing its value from the default of three does not cause unfriendliness; thus, *dupthresh* does not need to be constrained. In the case of *RTO*, the graph shows that if *RTO* is set below *exp.backoff* * (*srtt* + *rttvar*) then the resulting flow is too aggressive.

These graphs represent only a small subset of the experiments we have conducted to investigate TCP friendliness. We have experimented with setting the icTCP variables to random values outside of the safe range and have controlled each of the icTCP parameters in isolation as well as sets of the parameters simultaneously. In all cases, the TCP Reno flows competing with icTCP obtain at least as

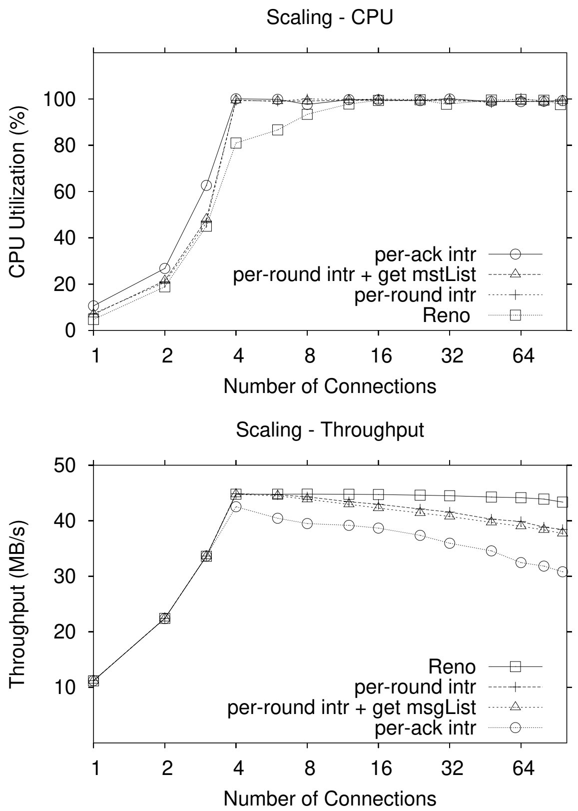


Figure 4: **CPU Overhead and Throughput in Scaling icTCP.** *We connect one sender host to four receiver hosts through different network interfaces. All links are 100 Mbps with no delay links, thus in aggragrate the sender host can send data outward at 400 Mbps. Along the x-axis, we increase the number of connections on the sender host. These connections are spread evenly across the four receivers. The first figure compares the overall CPU utilization of Reno, icTCP with per-ack and per-round interrupt. The second figure shows the icTCP throughput degradation when the sender load is high.*

much bandwidth as they do when competing with other TCP Reno flows, as desired. In summary, our results empirically demonstrate that icTCP flows require safe variable settings to be TCP friendly. Although these experiments do not prove that icTCP ensures network safety, these measurements combined with our analysis give us confidence that icTCP can be safely deployed.

5.3 CPU Overhead

We evaluate the overhead imposed by the icTCP framework in two ways. First, we explore the scalability of icTCP using synthetic user-level libraries; these experiments explore ways in which a user-level library can reduce CPU overhead by minimizing its interactions with the kernel. Second, we implement TCP Vegas [14] at user-level on top of icTCP; these experiments also allow us to directly compare icTCP to INFOTCP.

5.3.1 Scaling icTCP

We evaluate how icTCP scales as the number of connections is increased on a host. Different user-level extensions built on icTCP are expected to get and set different pieces of TCP information at different rates. The two factors that may determine the amount of overhead are whether the user process requires per-ack or per-round interrupts and whether or not the user process needs the icTCP message list and ack list data structures.

To show the scaling properties of user libraries built on icTCP, we construct three synthetic libraries that mimic the behavior of our later case studies. The first synthetic library uses per-ack interrupts (representing icTCP-EFR and icTCP-Eifel); the second library uses per-round interrupts (icCM); the final library uses per-round interrupts and also gets the message or ack list data structures (icTCP-Vegas, icTCP-Nice, and icTCP-RR).

The two graphs in Figure 4 show how icTCP and TCP Reno scale as the number of flows is increased on a host; the first figure reports CPU utilization and the second figure reports throughput. The first figure shows that icTCP with per-ack and per-round interrupts reaches 100% CPU utilization when there are three and four connections, respectively; the additional CPU overhead of also getting the icTCP message list is negligible. In comparison, TCP Reno reaches roughly 80% utilization with four connections, and then slowly increases to 100% at roughly 16 connections.

The second figure shows that throughput for icTCP starts to degrade when there are four or eight connections, depending upon whether they use per-ack or per-round interrupts, respectively. With 96 flows, the icTCP throughput with per-ack and per-round interrupts is lower than TCP Reno by about 30% and 12%, respectively. Thus, icTCP CPU overhead is noticeable but not prohibitive.

To measure the extent to which a user-level library can accurately implement TCP functionality, we measure the *interrupt miss rate*, defined as how frequently the user misses the interrupt for an ack or the end of a round. In the scaling experiments above with 96 connections, we observed a worst-case miss rate of 1.3% for per-ack interrupts and 0.4% for per-round interrupts. These low miss rates imply that functionality can be placed at the user-level that is responsive to current network conditions.

5.3.2 icTCP-Vegas

To further evaluate icTCP, we implement TCP Vegas congestion avoidance as a user-level library. TCP Vegas reduces latency and increases overall throughput, relative to TCP Reno, by carefully matching the sending rate to the rate at which packets are being drained by the network, thus avoiding packet loss. Specifically, if the sender sees that the measured throughput differs from the expected throughput by more than a fixed threshold, it increases or

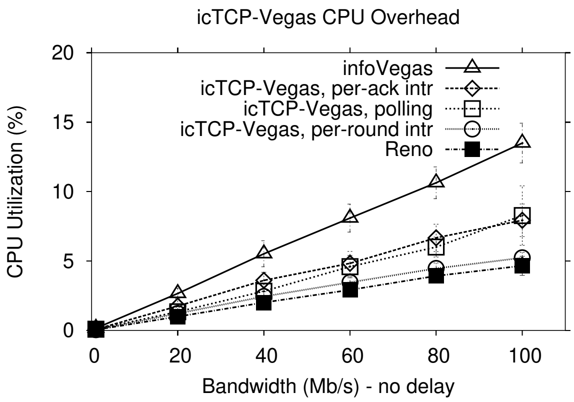


Figure 5: **icTCP-Vegas CPU Overhead.** *The figure compares the overall CPU utilization of Reno,* INFOVEGAS*, and the three versions of icTCP-Vegas. We vary bottleneck-link bandwidth along the x-axis.*

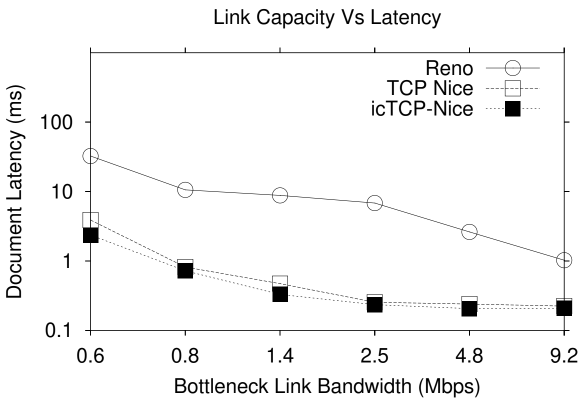


Figure 6: **icTCP-Nice: Link Capacity vs. Latency.** *A foreground flow competes with many background flows. Each line corresponds to a different run of the experiment with a protocol for background flows (i.e., icTCP, TCP Nice, Reno, or Vegas). The y-axis shows the average document transfer latency for the foreground traffic. The foreground traffic consists of a 3-minute section of a Squid proxy trace logged at U.C. Berkeley. The background traffic consists of long-running flows. The topology used is a dumbbell with 6 sending nodes and 6 receiving nodes. The foreground flow is alone on one of the sender/receiver pairs while 16 background flows are distributed across the remaining 5 sender/receiver pairs. The bottleneck link bandwidth is varied along the x-axis.*

decreases its congestion control window, *cwnd*, by one.

Implementation: Our implementation of the Vegas congestion control algorithm, icTCP-Vegas, is structured as follows. The operation of Vegas is placed in a user-level library. This library simply passes all messages directly to icTCP, *i.e.*, no buffering is done at this layer. We implement three different versions that vary the point at which we poll icTCP for new information: every time we send a new packet, every time an acknowledgment is received, or whenever a round ends. After the library gets the relevant TCP state, it calculates its own target congestion window, vcwnd. When the value of vcwnd changes, icTCP-Vegas sets this value explicitly inside icTCP.

We note that the implementation of icTCP-Vegas is similar to that of INFOVEGAS, described as part of an infokernel [7]. The primary difference between the two is INFOTCP must manage its own *vcwnd*, as it does not provide control over TCP variables. When INFOVEGAS calculates a value of *vcwnd* that is less than the actual *cwnd*, INFOVEGAS must buffer its packets and not transfer them to the TCP layer; INFOVEGAS then blocks until an acknowledgment arrives, at which point, it recalculates *vcwnd* and may send more messages.

Evaluation: We have verified that icTCP-Vegas behaves like the in-kernel implementation of Vegas. Due to space constraints we do not show these results; we instead focus our evaluation on CPU overhead.

Figure 5 shows the total (user and system) CPU utilization as a function of network bandwidth for TCP Reno, the three versions of icTCP-Vegas, and INFOVEGAS. As the available network bandwidth increases, CPU utilization increases for each implementation. The CPU utilization (in particular, system utilization) increases significantly for INFOVEGAS due to its frequent user-kernel crossings. This extra overhead is reduced somewhat for icTCP-Vegas when it polls icTCP on every message send or wakes on the arrival of every acknowledgment, but is still noticeable. Since getting icTCP information through the *getsockopt* interface incurs significant overhead, icTCP-Vegas can greatly reduce its overhead by getting information less frequently. Because Vegas adjusts *cwnd* only at the end of a round, icTCP-Vegas can behave accurately while still waking only every round. The optimization results in CPU utilization that is higher by only about 0.5% for icTCP-Vegas than for in-kernel Reno.

5.4 TCP Extensions

Our fourth axis for evaluating icTCP concerns the range of TCP extensions that it allows. Given the importance of this issue, we spend most of the remaining paper on this topic. We address this question by first demonstrating how five more TCP variants can be built on top of icTCP. These case studies are explicitly *not* meant to be exhaustive, but to instead illustrate the flexibility and simplicity of icTCP. We then briefly discuss whether icTCP can be used to implement a wider set of TCP extensions.

5.4.1 icTCP-Nice

In our first case study, we show that TCP Nice [52] can be implemented at user-level with icTCP. This study demonstrates that an algorithm that differs more radically from the base icTCP Reno algorithm can still be implemented. In particular, icTCP-Nice requires access to more of the internal state within icTCP, *i.e.*the complete message list.

Overview: TCP Nice provides a near zero-cost background transfer; that is, a TCP Nice background flow interferes little with foreground flows and reaps a large fraction of the spare network bandwidth. TCP Nice is similar to TCP Vegas, with two additional components: multiplicative window reduction in response to increasing round-trip times and the ability to reduce the congestion window below one. We discuss these components in turn.

TCP Nice halves its current congestion window when long round-trip times are measured, unlike Vegas which reduces its window by one and halves its window only when packets are lost. To determine when the window size should be halved, the TCP Nice algorithm monitors round-trip delays, estimates the total queue size at the bottleneck router, and signals congestion when the estimated queue size exceeds a fraction of the estimated maximum queue capacity. Specifically, TCP Nice counts the number of packets for which the delay exceeds $minRTT + (maxRTT - minRTT) * t$ (where $t = 0.1$); if the fraction of such delayed packets within a round exceeds f (where $f = 0.5$), then TCP Nice signals congestion and decreases the window multiplicatively.

TCP Nice also allows the window to be less than one; to effect this, when the congestion window is below two, TCP Nice adds a new timer and waits for the appropriate number of RTTs before sending more packets.

Implementation: The implementation of icTCP-Nice is similar to that of icTCP-Vegas, but slightly more complex. First, icTCP-Nice requires information about every packet instead of summary statistics; therefore, icTCP-Nice obtains the full message list containing the sequence number (*seqno*) and round trip time (*usrtt*) of each packet. Second, the implementation of windows less than one is tricky but can also use the *vcwnd* mechanism. In this case, for a window of $1/n$, icTCP-Nice sets *vcwnd* to 1 for a single RTT period, and to 0 for $(n-1)$ periods.

Evaluation: To demonstrate the effectiveness of the icTCP approach, we replicate several of the experiments from the original TCP Nice paper (*i.e.*, Figures 2, 3, and 4 in [52]).

Our results show that icTCP-Nice performs almost identically to the in-kernel TCP Nice, as desired.

Figure 6 shows the latency of the foreground connections when it competes against 16 background connections and the spare capacity of the network is varied. The results indicate that when icTCP-Nice or TCP Nice are used for background connections, the latency of the foreground connections is often an order of magnitude faster than when TCP Reno is used for background connections. As desired, icTCP-Nice and TCP Nice perform similarly.

The two graphs in Figure 7 show the latency of foreground connections and the throughput of background connections as the number of background connections increases. The graph on the top shows that as more background flows are added, document latency remains essentially constant when either icTCP-Nice or TCP Nice is used for the background flows. The graph on the bottom shows that icTCP-Nice and TCP Nice obtain more throughput as the number of flows increases. As desired, both icTCP-Nice and TCP Nice achieve similar results.

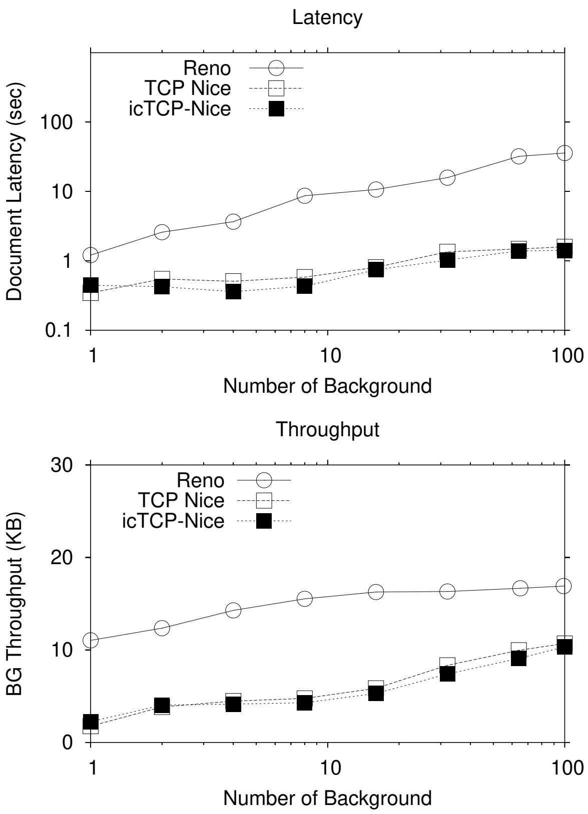


Figure 7: **icTCP-Nice: Impact of Background Flows.** *The two graphs correspond to the same experiment; the first graph shows the average document latency for the foreground traffic while the second graph shows the number of bytes the background flows manage to transfer during the 3 minutes period. Each line corresponds to a different protocol for background flows (i.e., TCP Reno, icTCP-Nice, or TCP Nice). The number of background flows is varied along the x-axis. The bottleneck link bandwidth is set to 840 kbps with a 50 ms delay. The experimental setup is identical to Figure 6.*

5.4.2 icCM

We now show that some important components of the Congestion Manager (CM) [8] can be built on icTCP. The main contribution of this study is to show that information can be shared across different icTCP flows and that multiple icTCP flows on the same sender can cooperate.

Overview: The Congestion Manager (CM) architecture [8] is motivated by two types of problematic behavior exhibited by emerging applications. First, applications that employ multiple concurrent flows between sender and receiver have flows that compete with each other for resources, prove overly aggressive, and do not share network information with each other. Second, applications which use UDP-based flows without sound congestion control do not adapt well to changing network conditions.

CM addresses these problems by inserting a module above IP at both the sender and the receiver; this layer maintains network statistics across flows, orchestrates data transmissions with a new hybrid congestion control algorithm, and obtains feedback from the receiver.

Implementation: The primary difference between icCM and CM is in their location; icCM is built on top of the icTCP layer rather than on top of IP. Because icCM leverages the congestion control algorithm and statistics already present in TCP, icCM is considerably simpler to implement than CM. Furthermore, icCM guarantees that its congestion control algorithm is stable and friendly to existing TCP traffic. However, the icCM approach does have the drawback that non-cooperative applications can bypass icCM and use TCP directly; thus, icCM can only guarantee fairness across the flows for which it is aware.

The icCM architecture running on each sending endpoint has two components: icCM clients associated with each individual flow and an icCM server; there is no component on the receiving endpoint. The icCM server has two roles: to identify macroflows (*i.e.*, flows from this endpoint to the same destination), and to track the aggregate statistics associated with each macroflow. To help identify macroflows, each new client flow registers its process ID and the destination address with the icCM server.

To track statistics, each client flow periodically obtains its own network state from icTCP (*e.g.*, its number of outstanding bytes, *snd.nxt* - *snd.una*) and shares this state with the icCM server. The icCM server periodically updates its statistics for each macroflow (*e.g.*, sums together the outstanding bytes for each flow in the macroflow). Each client flow can then obtain aggregate statistics for the macroflow for different time intervals.

To implement bandwidth sharing across clients in the same macroflow, each client calculates its own window to limit its number of outstanding bytes. Specifically, each icCM client obtains from the server the number of flows in this macroflow and the total number of outstanding bytes in this flow. From these statistics, the client calculates the number of bytes it can send to obtain its fair share of the bandwidth. If the client is using TCP for transport, then it simply sets *vcwnd* in icTCP to this number. Thus, icCM clients within a macroflow do not compete with one another and instead share the available bandwidth evenly.

Evaluation: We demonstrate the effectiveness of using icTCP to build a congestion manager by replicating one of the experiments performed for CM (*i.e.*, Figure 14 in [8]). In the experiments shown in Figure 8, we place four flows within a macroflow. As shown in the first graph, when four TCP Reno flows are in a macroflow, they do not share the available bandwidth fairly; the performance of the four connections varies between 39 KB/s and 24 KB/s with standard deviation of 5.5 KB/s. In contrast, as shown in the second graph, when four icCM flows are in a macroflow, the connections progress at similar and consistent rates; all four icCM flows achieve throughputs of roughly 30 KB/s with a standard deviation of 0.6 KB/s.

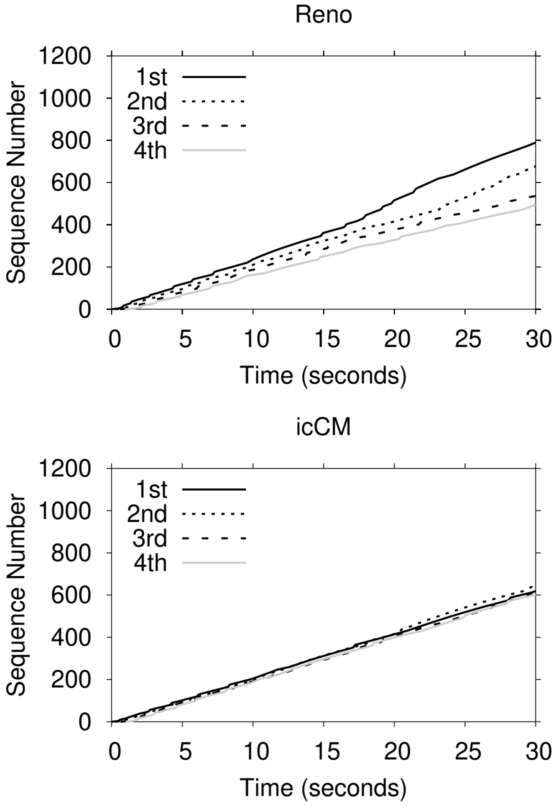


Figure 8: **icCM Fairness.** *The two graphs compare the performance of four concurrent transfers from one sender to one receiver, with the bottleneck link set to 1 Mb/s and a 120 ms delay. In the first graph, stock Reno is used; in the second graph, icCM manages the four TCP flows.*

5.4.3 icTCP-RR

TCP's fast retransmit optimization is fairly sensitive to the presence of duplicate acknowledgments. Specifically, when TCP detects that three duplicate acks have arrived, it assumes that a loss has occurred, and triggers a retransmission [5, 30]. However, recent research indicates that packet reordering may be more common in the Internet than earlier designers suspected [3, 9, 11, 57]. When frequent reordering occurs, the TCP sender receives a rash of duplicate acks and wrongly concludes that a loss has occurred. As a result, segments are unnecessarily retransmitted (wasting bandwidth) and the congestion window is needlessly reduced (lowering client performance).

Overview: A number of solutions for handling duplicate acknowledgments have been suggested in the literature [11, 57]. At a high level, the algorithms detect the presence of reordering (*e.g.*, by using DSACK) and then increase the duplicate threshold value (*dupthresh*) to avoid triggering fast retransmit. We base our implementation on that of Blanton and Allman's work [11], which limits the maximum value of *dupthresh* to 90% of the window

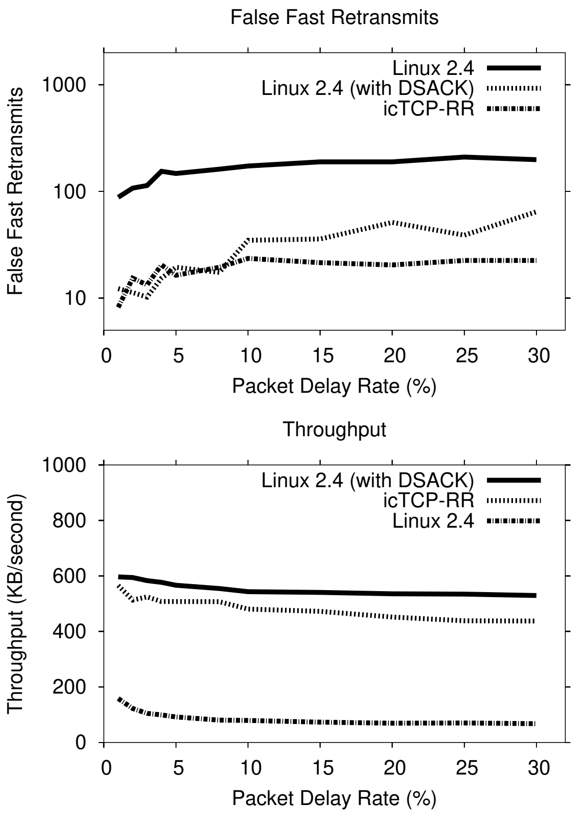


Figure 9: **Avoiding False Retransmissions with icTCP-RR.** *On the top is the number of false retransmissions and on the bottom is throughput, as we vary the fraction of packets that are delayed (and hence reordered) in our modified NistNet router. We compare three different implementations, as described in the text. The experimental setup includes a single sender and receiver; the bottleneck link is set to 5 Mb/s and a 50 ms delay. The NistNet router runs on the first router, introducing a normally distributed packet delay with mean of 25 ms, and standard deviation of 8 ms.*

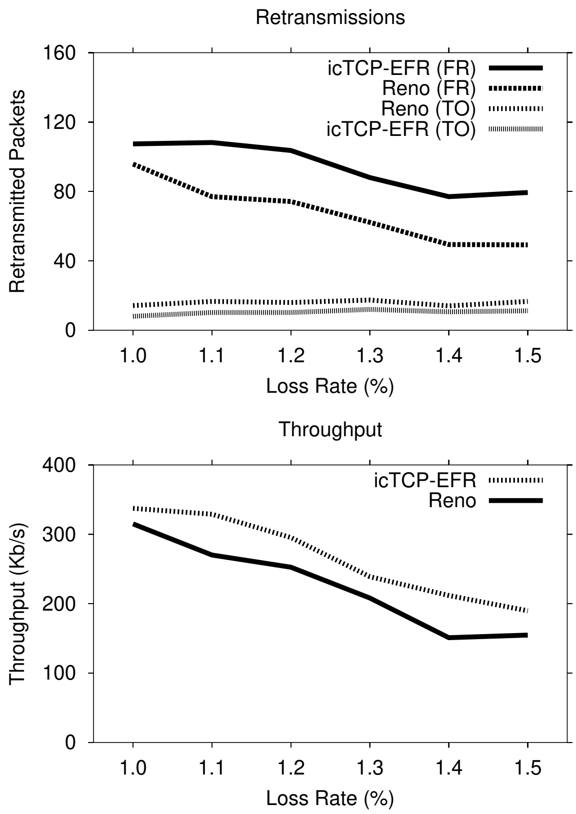


Figure 10: **Aggressive Fast Retransmits with icTCP-EFR.** *On the top is the number of retransmitted packets for both Reno and icTCP-EFR – due to both retransmission timeouts (TO) and fast retransmits (FR) – and on the bottom is the achieved bandwidth. Along the x-axis, we vary the loss rate so as to mimic a wireless LAN. A single sender and single receiver are used, and the bottleneck link is set to 600 Kb/s and a 6 ms delay.*

size and, when a timeout occurs, sets *dupthresh* back to its original value of 3.

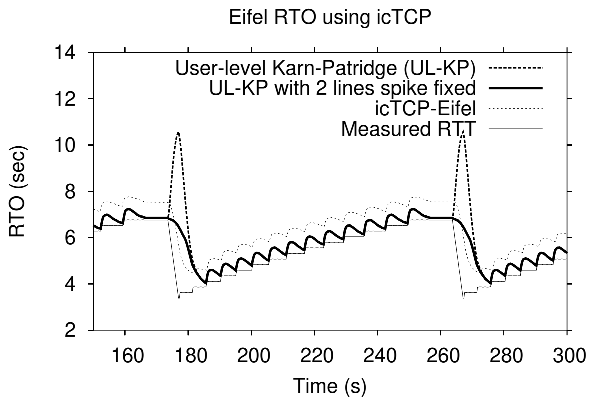
Implementation: The user-level library implementation, icTCP-RR, is straight-forward. The library keeps a history of acks received; this list is larger than the kernel exported ack list because the kernel may be aggressive in pruning its size, thus losing potentially valuable information. When a DSACK arrives, icTCP places the sequence number of the falsely retransmitted packet into the ack list. The library consults the ack history frequently, looking for these occurrences. If one is found, the library searches through past history to measure the reordering length and sets *dupthresh* accordingly.

Evaluation: Figure 9 shows the effects of packet reordering. We compare three different implementations: stock Linux 2.4 without the DSACK enhancement, Linux 2.4 with DSACK and reordering avoidance built into the kernel, and our user-level icTCP-RR implementation. In the first graph, we show the number of "false" fast retransmissions that occur, where a false retransmission is one that is caused by reordering. One can see that the stock kernel issues many more false retransmits, as it (incorrectly) believes the reordering is actual packet loss. In the second graph, we observe the resulting bandwidth. Here, the DSACK in-kernel and icTCP-RR versions perform much better, essentially ignoring duplicate acks and thus achieving much higher bandwidth.

5.4.4 icTCP-EFR

Our previous case study showed that increasing *dupthresh* can be useful. In contrast, in environments such as wireless LANs, loss is much more common and duplicate acks should be used a strong signal of packet loss, particularly when the window size is small [50]. In this case, the opposite solution is desired; the value of *dupthresh* should be lowered, thus invoking fast retransmit aggressively so as to avoid costly retransmission timeouts.

Overview: We next discuss icTCP-EFR, a user-level library of implementation of EFR (Efficient Fast Retransmit) [50]. The observation underlying EFR is simple: the



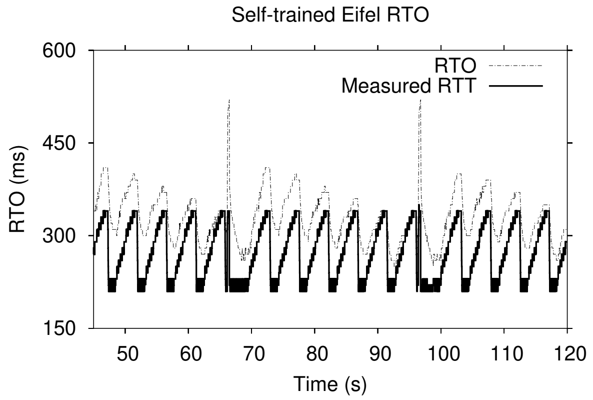


Figure 11: **Adjusting RTO with icTCP-Eifel.** *The graph on the top shows three versions of icTCP-Eifel. For each experiment, the measured round-trip time is identical; however, the calculated RTO differs. The first line shows when the Karn-Partridge RTO algorithm [33] is disabled in the kernel that it can be implemented at user-level with icTCP. In the second experiment, we remove two lines of TCP code that were added to fix the RTO spike; we show that this same fix can be easily provided at user-level. In the third experiment, we implement the full Eifel RTO algorithm at user-level. In these experiments, we emulate a bandwidth of 50 kbps, 1 second delay, and a queue size of 20. The graph on the bottom shows the full adaptive Eifel RTO algorithm with a bandwidth of 1000 kbps, 100 ms delay, and a queue size of 12.*

sender should adjust *dupthresh* so as to match the number of duplicate acks it could receive.

Implementation: The icTCP-EFR implementation is also quite straightforward. For simplicity, we only modify *dupthresh* when the window is small; this is where the EFR scheme is most relevant. When the window is small, the library frequently checks the message list for duplicate acks; when it sees one, it computes and sets a new value for *dupthresh*.

Evaluation: Figure 10 shows the behavior of icTCP-EFR versus the in-kernel Reno as a function of loss rate in an emulated wireless network. Because icTCP-EFR interprets duplicate acknowledgments as likely signs of loss, the number of fast retransmits increases (as shown in the graph on top) and more importantly, the number of costly retransmission timeouts is reduced. The graph on the bottom shows that bandwidth increases as a result.

5.4.5 icTCP-Eifel

The retransmission timeout value (RTO) determines how much time must elapse after a packet has been sent until the sender considers it lost and retransmits it. Therefore, the RTO is a prediction of the upper limit of the measured round-trip time (mRTT). Correctly setting RTO can greatly influence performance: an overly aggressive RTO may expire prematurely, forcing unnecessary spurious retransmission; an overly-conservative RTO may cause long idle times before lost packets are retransmitted.

Overview: The Eifel RTO [36] corrects two problems with the traditional Karn-Partridge RTO [33]. First, immediately after mRTT decreases, RTO is incorrectly increased; only after some period of time does the value of RTO decay to the correct value. Second, the "magic numbers" in the RTO calculation assume a low mRTT sampling rate and sender load; if these assumptions are incorrect, RTO incorrectly collapses into mRTT.

Implementation: We have implemented the Eifel RTO algorithm as a user-level library, icTCP-Eifel. This library needs access to three icTCP variables: mRTT, ssthresh, and cwnd; from mRTT, it calculates its own values of srtt (smoothed round-trip) and rttvar (round-trip variance). The icTCP-Eifel library operates as follows: it wakes when an acknowledgment arrives and polls icTCP for the new mRTT; if mRTT has changed, it calculates the new RTO and sets it within icTCP. Thus, this library requires safe control over RTO.

Evaluation: The first graph of Figure 11 shows a progression of three improvements in icTCP-Eifel; these experiments approximately match those in the Eifel RTO paper (*i.e.*, Figure 6 in [36]). In the first implementation, we disable the Karn-Partridge RTO algorithm in the kernel and instead implement it in icTCP-Eifel; as expected, this version incorrectly increases RTO when mRTT decreases. The second implementation corrects this problem with two additional lines of code at user-level; however, RTO eventually collapses into mRTT. Finally, the third version of icTCP-Eifel adjusts RTO so that it is more conservative and avoids spurious retransmissions. The second graph of Figure 11 is similar to Figure 10 in the Eifel paper and shows that we have implemented the full Eifel RTO algorithm at user-level: this algorithm allows RTO to become increasingly aggressive until a spurious timeout occurs, at which point it backs off to a more conservative value.

5.4.6 Summary

From our case studies, we have seen a number of strengths of the icTCP approach. First, icTCP easily enables TCP variants that are less aggressive than Reno to be implemented simply and efficiently at user-level (*e.g.*, TCP Vegas and TCP Nice); thus, there is no need to push such changes into the kernel. Second, icTCP is ideally suited for tuning parameters whose optimal values depend upon the environment and the workload (*e.g.*, the value of dupthresh). Third, icTCP is useful for correcting errors in parameter values (*e.g.*, the behavior of RTO).

Our case studies have illustrated limitations of icTCP as well. From icCM, we saw how to assemble a framework that shares information across flows; however, any information that is shared across flows can only be done voluntarily. Furthermore, congestion state learned from previous flows cannot be directly inherited by later flows; this limitation arises from icTCP's reliance upon the in-kernel TCP stack, which cannot be forcibly set to a starting congestion state.

5.4.7 Implementing New Extensions

We evaluate the ability of icTCP to implement a wider range of TCP extensions by considering the list discussed for STP [44]. Of the 27 extensions, 9 have already been standardized in Linux 2.4.18 (*e.g.*, SACK, DSACK, FACK, TCP for high performance, ECN, New Reno, and SYN cookies) and 4 have been implemented with icTCP (*i.e.*, RR-TCP, Vegas, CM, and Nice). We discuss some of the challenges in implementing the remaining 14 extensions. We place these 14 extensions into three categories: those that introduce new algorithms on existing variables, those that modify the packet format, and those that modify the TCP algorithm structure or mechanisms.

Existing Variables: We classify three of the 14 extensions as changing the behavior of existing variables: appropriate byte counting (ABC) [2], TCP Westwood [55]), and equation-based TCP (TFRC) [26]. Other recently proposed TCP extensions that fall into this category include Fast TCP [17], Limited Slow-Start [25], and HighSpeed TCP [24].

These extensions are the most natural match with icTCP and can be implemented to the extent that they are no more aggressive than TCP Reno. For example, equation-based TCP specifies that the congestion window should increase and decrease more gradually than Reno; icTCP-Eqn allows *cwnd* to increase more gradually, as desired, but forces *cwnd* to decrease at the usual Reno rate. We believe that conservative implementations of these extensions are still beneficial. For example, ABC implemented on icTCP cannot aggressively increase *cwnd* when a receiver delays an ack, but icTCP-ABC can still correct for ack division. In the case of HighSpeed TCP, the extension cannot be supported in a useful manner because it is strictly more aggressive, specifying that *cwnd* should be decreased by a smaller amount than TCP Reno does.

One issue that arises with these extensions is how icTCP enforces TCP friendliness: icTCP constrains each TCP virtual variable within a safe range, which may be overly conservative. For example, icTCP does not allow small increases in TCP's initial congestion window [4], even though over a long time period these flows are generally considered to be TCP friendly. Alternatively, STP [44] uses a separate module to enforce TCP friendliness; this module monitors the sending rate and verifies that it is is below an upper-bound determined by the state of the connection, the mean packet size, the loss event rate, round-trip time, and retransmission timeout. Although icTCP could use a similar modular approach, we believe that the equation-based enforcer has an important drawback: non-conforming flows must be terminated, since packets cannot be buffered in a bounded size and then sent at a TCP-friendly rate. Rather than terminate flows, icTCP naturally modulates agressive flows in a manner that is efficient in both space and time.

Packet Format: We classify six of the 14 extensions as changing the format or the contents of packets; for example, extensions that put new bits into the TCP reserved field, such as the Eifel algorithm [31] or robust congestion signaling [21]. These extensions cannot be implemented easily with icTCP in its current form; therefore, we believe that it is compelling to expand icTCP to allow variables in the packet header to be set. However, it may be difficult to ensure that this is done safely.

We can currently approximate this behavior by encapsulating extra information in application data and requiring both the sender and receiver to use an icTCP-enabled kernel and an appropriate library; this technique allows extra information to be passed between protocol stacks while remaining transparent to applications. With this technique, we have implemented functionality similar to that of DCCP [34]; in our implementation, a user-level library that transmits packets with UDP obtains network information from an icTCP flow between the same sender and receiver. We are currently investigating this approach in more detail.

Structure and Mechanism: Approximately five of the 14 extensions modify fundamental aspects of the TCP algorithm: some extensions do not follow the existing TCP states (*e.g.*, T/TCP [13] and limited transmit [3]) and some define new mechanisms (*e.g.*, the SCTP checksum [49]). Given that these extensions deviate substantially from the base TCP Reno algorithm, we do not believe that icTCP can implement such new behavior.

An approach for addressing this limitation, as well as for modifying packet headers, may be for icTCP to provide control underneath the kernel stack with a packet filter [42]. In this way, users could exert control over their packets, perhaps changing the timing, ordering, or altogether suppressing or duplicating some subset of packets as they pass through the filter. Again, such control must be meted out with caution, since ensuring such changes remain TCP friendly is a central challenge.

In summary, icTCP is not as powerful as STP [44] and thus can implement a smaller range of TCP extensions.

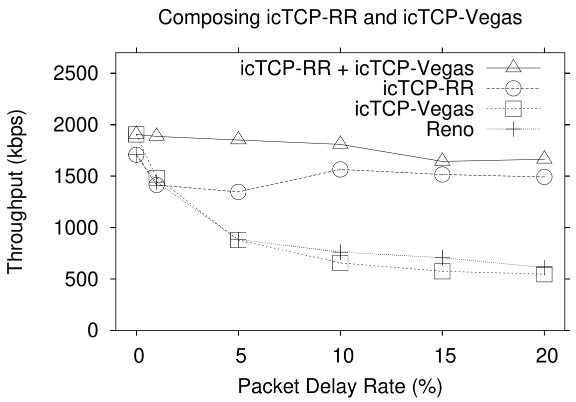


Figure 12: **Composing icTCP-Vegas and icTCP-RR.** *The figure shows the strength of composing multiple icTCP libraries in an environment where reordering occurs and the available space in the bottleneck queue is low. When both libraries are used at the same time in this particular environment, the throughput is higher compared to when only one of the libraries is used. The experimental setup includes a single sender and receiver; the bottleneck queue size is set to 5 and the link is set to 2 Mb/s and a 50 ms delay. The NistNet router runs on the first router, introducing a normally distributed packet delay with mean of 25 ms, and standard deviation of 8. On the x-axis we vary the percentage of delayed packets.*

However, we believe that the simplicity of providing an icTCP layer on a real system may outweigh this drawback.

5.5 Ease of Development

For our fifth and final question we address the complexity of using the icTCP framework to develop TCP extensions. We answer this question first by showing the ease with which user-level libraries on icTCP can be combined to perform new functionality. We then directly compare the complexity of building TCP extensions at user-level to building them directly in the kernel.

The icTCP framework enables functional composition: given that each user-level library exports the same interface as icTCP, library services can be stacked to build more powerful functionality. In the simplest case, the stacked libraries control disjoint sets of icTCP variables. For example, if the icTCP-Vegas and icTCP-RR libaries are stacked, then the combination controls the values of both *cwnd* and *dupthresh*. Figure 12 shows the advantage of stacking these two libraries: flows running in an environment with both packet reordering and small bottleneck queues exhibit higher throughput with both libraries than with either libary alone. Alternatively, the stacked libaries may control overlapping sets of icTCP variables. In this case, each layer further constrains the range of safe values for a virtual variable.

To quantify the complexity of building functionality either on top of icTCP or within the kernel, we count the number of C statements in the implementation (*i.e.*, the number of semicolons), removing those that are used only for printing or debugging. Table 3 shows the number of C statements required for the four case studies with reference implementations: Vegas, Nice, CM, and RR. Comparing the icTCP user-level libraries to the native implementations, we see that the number of new statements across the two is quite comparable. We conclude that developing services using icTCP is not much more complex than building them natively and has the advantage that debugging and analysis can be performed at user-level.

Case Study	icTCP	Native
icTCP-Vegas	162	140
icTCP-Nice	191	267
icCM	438	1200*
icTCP-RR	48	26

Table 3: **Ease of Development with icTCP.** *The table reports the number of C statements (counted with the number of semicolons) needed to implement the case studies on icTCP compared to a native reference implementation. For the native Vegas implementation, we count the entire patch for Linux 2.2/2.3 [15]. For TCP Nice, we count only statements changing the core transport layer algorithm. For CM, quantifying the number of needed statements is complicated by the fact that the authors provide a complete Linux kernel, with CM modifications distributed throughout; we count only the transport layer. (*) However, this comparison is still not fair given that CM contains more functionality than icCM. For RR, we count the number of lines in Linux 2.4 to calculate the amount of reordering. In-kernel RR uses SACK/DSACK, whereas icTCP-RR traverses the ack list.*

6 Conclusions

We have presented the design and implementation of icTCP, a slightly modified version of Linux TCP that exposes information and control to applications and user-level libraries above. We have evaluated icTCP across five axes and our findings are as follows.

First, converting a TCP stack to icTCP requires only a small amount of additional code; however, determining precisely where limited virtual parameters should be used in place of the original TCP parameters is a non-trivial exercise. Second, icTCP allows ten internal TCP variables to be safely set by user-level processes; regardless of the values chosen by the user, the resulting flow is TCP friendly. Third, icTCP incurs minimal additional CPU overhead relative to in-kernel implementations as long as icTCP is not polled excessively for new information; to help reduce overhead, icTCP allows processes to block until an acknowledgment arrives or until the end of a round. Fourth, icTCP enables a range of TCP extensions to be implemented at user-level. We have found that icTCP framework is particularly suited for extensions that implement congestion control algorithms that are less aggressive than Reno and for adjusting parameters to better match workload or environment conditions. To support more radical TCP extensions, icTCP will need to be developed further, such as by allowing TCP headers to be safely set or packets and acknowledgments to be reordered or delayed. Fifth, and finally, developing TCP extensions on top of icTCP is not more complex than implementing them directly in the kernel and are likely easier to debug.

We believe that exposing information and control over other layers in the network stack will be useful as well. For example, given the similarity between TCP and SCTP [6], we believe that SCTP can be extended in a straight-forward manner to icSCTP. An icSCTP framework will allow user-level libraries to again deal with problems such as spurious retransmission [12] as well as implement new functionality for network failure detection and recovery [32].

Our overall conclusion is that icTCP is not quite as powerful as other proposals for extending TCP or other networking protocols [41, 44]. However, the advantage of icTCP is in its simplicity and pragmatism: it is relatively easy to implement icTCP, flows built on icTCP remain TCP friendly, and the computational overheads are reasonable. Thus, we believe that systems with icTCP can, in practice and not just in theory, reap the benefits of user-level TCP extensions.

7 Acknowledgments

The experiments in this paper were performed exclusively in the Netbed network emulation environment from Utah [56]. We are greatly indebted to Robert Ticci, Tim Stack, Leigh Stoller, Kirk Webb, and Jay Lepreau for providing this superb environment for networking research.

We would like to thank Nitin Agrawal, Lakshmi Bairavasundaram, Nathan Burnett, Vijayan Prabhakaran, and Muthian Sivathanu for their helpful discussions and comments on this paper. We would also like to thank Jeffrey Mogul for his excellent shepherding, which has substantially improved the content and presentation of this paper. Finally, we thank the anonymous reviewers for their many helpful suggestions. This work is sponsored by NSF CCR-0092840, CCR-0133456, CCR-0098274, NGS-0103670, ITR-0086044, ITR-0325267, IBM, EMC, and the Wisconsin Alumni Research Foundation.

References

[1] M. B. Abbott and L. L. Peterson. A Language-based Approach to Protocol Implementation. *IEEE/ACM Transactions on Networking*, 1(1):4–19, Feb. 1993.

[2] M. Allman. TCP Congestion Control with Appropriate Byte Counting. RFC 3465, Feb. 2002.

[3] M. Allman, H. Balakrishnan, and S. Floyd. Enhancing TCP's Loss Recovery Using Limited Transmit, Jan. 2001. RFC 3042.

[4] M. Allman, S. Floyd, and C. Patridge. Increasing TCP's Initial Window. RFC 3390, Internet Engineering Task Force, 2002.

[5] M. Allman, V. Paxson, and W. R. Stevens. TCP Congestion Control. RFC 2581, Internet Engineering Task Force, Apr. 1999.

[6] J. Armando L. Caro, J. R. Iyengar, P. D. Amer, S. Ladha, I. Gerard J. Heinz, and K. C. Shah. SCTP: A Proposed Standard for Robust Internet Data Transport. IEEE Computer, November 2003.

[7] A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, N. C. Burnett, T. E. Denehy, T. J. Engle, H. S. Gunawi, J. Nugent, and F. I. Popovici. Transforming Policies into Mechanisms with Infokernel. In *SOSP '03*, 2003.

[8] H. Balakrishnan, H. S. Rahul, and S. Seshan. An Integrated Congestion Management Architecture for Internet Hosts. In *SIGCOMM '99*, pages 175–187, 1999.

[9] J. Bellardo and S. Savage. Measuring Packet Reordering. In *Proceedings of the 2002 ACM/USENIX Internet Measurement Workshop*, Marseille, France, Nov. 2002.

[10] E. Biagioni. A Structured TCP in Standard ML. In *Proceedings of SIGCOMM '94*, pages 36–45, London, United Kingdom, Aug. 1994.

[11] E. Blanton and M. Allman. On Making TCP More Robust to Packet Reordering. *ACM Computer Communication Review*, 32(1), Jan. 2002.

[12] E. Blanton and M. Allman. Using TCP DSACKs and SCTP Duplicate TSNs to Detect Spurious Retransmissions. RFC 3708, Internet Engineering Task Force, February 2004.

[13] R. Braden. T/TCP - TCP Extensions for Transactions. RFC 1644, Internet Engineering Task Force, 1994.

[14] L. S. Brakmo, S. W. O'Malley, and L. L. Peterson. TCP Vegas: New Techniques for Congestion Detection and Avoidance. In *Proceedings of SIGCOMM '94*, pages 24–35, London, United Kingdom, Aug. 1994.

[15] N. Cardwell and B. Bak. A TCP Vegas Implementation for Linux. http://flophouse.com/ neal/uw/linux-vegas/.

[16] M. Carson and D. Santay. NIST Network Emulation Tool. snad.ncsl.nist.gov/nistnet, January 2001.

[17] D. X. W. Cheng Jin and S. H. Low. FAST TCP: Motivation, Architecture, Algorithms, Performance. In *INFOCOM '04*, 2004.

[18] T. Dunigan, M. Mathis, and B. Tierney. A TCP Tuning Daemon. In *SC2002*, Nov. 2002.

[19] A. Edwards and S. Muir. Experiences Implementing a High-Performance TCP in User-space. In *SIGCOMM '95*, pages 196–205, Cambridge, Massachusetts, Aug. 1995.

[20] D. Ely, S. Savage, and D. Wetherall. Alpine: A User-Level Infrastructure for Network Protocol Development. In *Proceedings of the 3rd USENIX Symposium on Internet Technologies and Systems (USITS '01)*, pages 171–184, San Francisco, California, Mar. 2001.

[21] D. Ely, N. Spring, D. Wetherall, and S. Savage. Robust Congestion Signaling. In *ICNP '01*, Nov. 2001.

[22] M. E. Fiuczynski and B. N. Bershad. An Extensible Protocol Architecture for Application-Specific Networking. In *Proceedings of the USENIX Annual Technical Conference (USENIX '96)*, San Diego, California, Jan. 1996.

[23] S. Floyd. The New Reno Modification to TCP's Fast Recovery Algorithm. RFC 2582, Internet Engineering Task Force, 1999.

[24] S. Floyd. HighSpeed TCP for Large Congestion Windows. RFC 3649, Internet Engineering Task Force, 2003.

[25] S. Floyd. Limited Slow-Start for TCP with Large Congestion Windows. RFC 3742, Internet Engineering Task Force, 2004.

[26] S. Floyd, M. Handley, J. Padhye, and J. Widmer. Equation-based Congestion Control for Unicast Applications. In *Proceedings of SIGCOMM '00*, pages 43–56, Stockholm, Sweden, Aug. 2000.

[27] S. Floyd, J. Mahdavi, M. Mathis, and M. Podolsky. An Extension to the Selective Acknowledgment (SACK) Option for TCP. RFC 2883, Internet Engineering Task Force, 2000.

[28] G. R. Ganger, D. R. Engler, M. F. Kaashoek, H. M. Briceno, R. Hunt, and T. Pinckney. Fast and Flexible Application-level Networking on Exokernel Systems. *ACM TOCS*, 20(1):49–83, Feb. 2002.

[29] ISI/USC. Transmission Control Protocol. RFC 793, Sept. 1981.

[30] V. Jacobson. Congestion avoidance and control. In *Proceedings of SIGCOMM '88*, pages 314–329, Stanford, California, Aug. 1988.

[31] V. Jacobson, R. Braden, and D. Borman. TCP Extensions for High Performance. RFC 1323, Internet Engineering Task Force, 1992.

[32] A. L. C. Jr., J. R. Iyengar, P. D. Amer, and G. J. Heinz. A Two-level Threshold Recovery Mechanism for SCTP. SCI, 2002.

[33] P. Karn and C. Partridge. Improving Round-Trip Time Estimates in Reliable Transport Protocols. In *Proceedings of SIGCOMM '87*, Aug. 1987.

[34] E. Kohler, M. Handley, and S. Floyd. Designing DCCP: Congestion Control Without Reliability. www.icir.org/kohler/dcp/dccp-icnp03s.pdf, 2003.

[35] E. Kohler, M. F. Kaashoek, and D. R. Montgomery. A Readable TCP in the Prolac Protocol Language. In *Proceedings of SIGCOMM '99*, pages 3–13, Cambridge, Massachusetts, Aug. 1999.

[36] R. Ludwig and K. Sklower. The Eifel Retransmission Timer. *ACM Computer Communications Review*, 30(3), July 2000.

[37] C. Maeda and B. N. Bershad. Protocol Service Decomposition for High-performance Networking. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles (SOSP '93)*, pages 244–255, Asheville, North Carolina, Dec. 1993.

[38] J. Mahdavi and S. Floyd. TCP-friendly unicast rate-based flow control. end2end-interest mailing list, http://www.psc.edu/networking/papers/tcp_friendly.html, Jan. 1997.

[39] M. Mathis, J. Heffner, and R. Reddy. Web100: Extended tcp instrumentation. *ACM Computer Communications Review*, 33(3), July 2003.

[40] M. Mathis, J. Mahdavi, S. Floyd, and A. Romanow. TCP Selective Acknowledgment Options. RFC 2018, Internet Engineering Task Force, 1996.

[41] J. Mogul, L. Brakmo, D. E. Lowell, D. Subhraveti, and J. Moore. Unveiling the Transport. In *HotNets II*, 2003.

[42] J. C. Mogul, R. F. Rashid, and M. J. Accetta. The Packet Filter: an Efficient Mechanism for User-level Network Code. In *Proceedings of the 11th ACM Symposium on Operating Systems Principles (SOSP '87)*, Austin, Texas, November 1987.

[43] J. Padhye and S. Floyd. On Inferring TCP Behavior. In *SIGCOMM '01*, pages 287–298, August 2001.

[44] P. Patel, A. Whitaker, D. Wetherall, J. Lepreau, and T. Stack. Upgrading Transport Protocols using Untrusted Mobile Code. In *SOSP '03*, 2003.

[45] V. Paxson, M. Allman, S. Dawson, W. Fenner, J. Griner, I. Heavens, K. Lahey, J. Semke, and B. Volz. Known TCP Implementation Problems. RFC 2525, Internet Engineering Task Force, Mar. 1999.

[46] P. Pradhan, S. Kandula, W. Xu, A. Shaikh, and E. Nahum. Daytona: A user-level tcp stack. http://nms.lcs.mit.edu/ kandula/data/daytona.pdf, 2002.

[47] K. Ramakrishnan, S. Floyd, and D. Black. The Addition of Explicit Congestion Notification (ECN) to IP. RFC 3168, Internet Engineering Task Force, 2001.

[48] M. I. Seltzer, Y. Endo, C. Small, and K. A. Smith. Dealing With Disaster: Surviving Misbehaved Kernel Extensions. In *Proceedings of the 2nd Symposium on Operating Systems Design and Implementation (OSDI '96)*, pages 213–228, Seattle, Washington, Oct. 1996.

[49] J. Stone, R. Stewart, and D. Otis. Stream control transmission protocol. RFC 3309, Sept. 2002.

[50] Y. Tamura, Y. Tobe, and H. Tokuda. EFR: A Retransmit Scheme for TCP in Wireless LANs. In *IEEE Conference on Local Area Networks*, pages 2–11, 1998.

[51] C. A. Thekkath, T. D. Nguyen, E. Moy, and E. D. Lazowska. Implementing Network Protocols at User Level. *IEEE/ACM Transactions on Networking*, 1(5):554–565, 1993.

[52] A. Venkataramani, R. Kokku, and M. Dahlin. Tcp-nice: A mechanism for background transfers. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI '02)*, pages 329–344, Boston, Massachusetts, Dec. 2002.

[53] T. von Eicken, A. Basu, V. Buch, and W. Vogels. U-Net: A User-Level Network Interface for Parallel and Distributed Computing. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP '95)*, pages 40–53, Copper Mountain Resort, Colorado, Dec. 1995.

[54] D. A. Wallach, D. R. Engler, and M. F. Kaashoek. ASHs: Application-specific Handlers for High-performance Messaging. *IEEE/ACM Transactions on Networking*, 5(4):460–474, Aug. 1997.

[55] R. Wang, M. Valla, M. Sanadidi, and M. Gerla. Adaptive Bandwidth Share Estimation in TCP Westwood. In *IEEE Globecom '02*, 2002.

[56] B. White, J. Lepreau, L. Stoller, R. Ricci, S. Guruprasad, M. Newbold, M. Hibler, C. Barb, and A. Joglekar. An Integrated Experimental Environment for Distributed Systems and Networks. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI '02)*, pages 255–270, Boston, Massachusetts, Dec. 2002.

[57] M. Zhang, B. Karp, S. Floyd, and L. Peterson. RR-TCP: A Reordering-Robust TCP with DSACK. In *11th International Conference on Network Protocols (ICNP '03)*, June 2003.