Deploying Safe User-Level Network Services with icTCP

Haryadi S. Gunawi
Andrea Arpaci-Dusseau
Remzi Arpaci-Dusseau

Technical Report #1517

October

Deploying Safe User-Level Network Services with icTCP

Haryadi S. Gunawi, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau
Department of Computer Sciences, University of Wisconsin, Madison

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 [11], TCP Nice [50], UDP with congestion control [28], and the Congestion Manager [6]; we show how user-level libraries can safely control the duplicate acknowledgment threshold to make TCP more robust to packet reordering [9] or more appropriate for wireless LANs [47]; we also show how the retransmission timeout value can be adjusted dynamically [30]. 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, 6, 9, 10, 11, 19, 21, 26, 34, 41, 47, 50, 55]. Unfortunately, while some proposals are eventually adopted, many suggested modifications to the TCP stack do not become widely deployed [38].

In this paper, we address the problem of deployment by suggesting 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 [35, 38] 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 [37, 39].

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 [44]. 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 [32]? By building on top of the extant TCP Reno stack and allowing virtual variables to be set to values only within a range analyzed as safe, 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? We show services built on icTCP 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 seven 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 [11], TCP Nice [50], Congestion Manager (CM) [6], and congestion-controlled UDP [28]. In our second set of case studies, we focus on TCP modifications that behave differently in the presence of duplicate acknowledgments: avoid misinterpreting packet reordering as packet loss [9, 55] and efficient fast retransmit (EFR) [47]. In our third set, we explore an extension that adjusts the retransmit timeout value [30].

Finally, can these services be developed easily within the framework? We show that the number of statements 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 aspects 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

As described, icTCP has two components: exposing key internal state and providing safe control over important variables. The idea of exposing network state to end hosts has been explored in a number of contexts [6, 33, 42, 43]; given that this is not the primary contribution of icTCP, and due to space constraints, we do not discuss these projects further. Instead, we compare the icTCP approach for controlling variables to other work that provides some amount of networking extensibility.

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

Mogul *et al.* [35] 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. 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 [33] are developing an advanced management interface for TCP. Similar to the information component of icTCP, Web100 provides an instrumented TCP that exports a variety of per-connection statistics; however, Web100 does not propose exporting as detailed of 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 [14]. 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 also addresses the problem of TCP deployment [38]. 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 wrought with difficulty [44]. 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 an infokernel [5]. INFOTCP exposes internal TCP state, but does not provide any way to directly set TCP state. With INFOTCP, a user-level implementation of TCP Vegas can be performed on top of the in-kernel TCP Reno; we repeat this case study to show that icTCP implements this functionality more efficiently. We believe that icTCP has two important advantages over INFOTCP. First, icTCP provides more efficient and more accurate control: applications do not need to perform extra buffering and sleep/wake events as they do with INFOTCP. Second, icTCP supports a broader range of applications than INFOTCP.

User-Level TCP: Researchers have often found it useful to move portions of the conventional network stack to user-level [15, 16, 40]. User-level TCP can provide the same advantage as icTCP of simplifying protocol development. However, a user-level TCP implementation typically struggles with performance, due to extra buffering and/or context switching. Again, these implementations do not ensure TCP friendliness and thus do not provide a real solution to the deployment problem.

Application-Specific Networking: A large number of projects have investigated how to provide general extensibility of network services [18, 22, 31, 49, 51, 52]. These projects allow network protocols to be more specialized to application requirements 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.

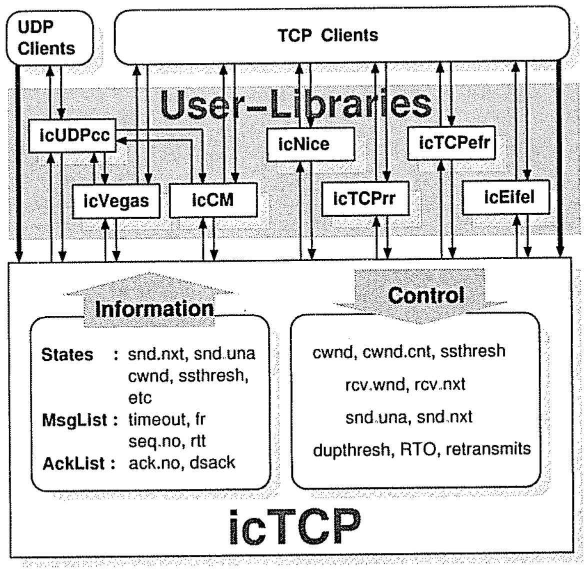

Figure 1: **icTCP Architecture.** *The diagram shows the icTCP architecture. At the base of the stack is icTCP, a slightly modified TCP stack that exports information and limited control. On top of icTCP, we have built a number of user-level libraries that implement various pieces of functionality suggested 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.*

Protocol Languages and Architectures: Network languages [1, 29] and structured TCP implementations [8] simplify the development of network protocols. Given the ability to replace or subclass modules, it is 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 the standard interfaces. Different TCP connections can use different icTCP libraries. Some icTCP libraries can also be stacked. For example, the icUDP$_{CC}$ library can either use the default congestion control algorithm included in icTCP, or it can use that provided by libTCP$_{Vegas}$. The implication is that each stackable user-level library must export the same interface as icTCP. The design of icTCP is such that in most cases, only the sending side needs to have icTCP deployed; receivers can be running icTCP or an unmodified kernel stack.

To simplify the implementation, icTCP exports information and provides control through the BSD socket interface with a few new options. Although this approach minimized our implementation work, it does impose unnecessary run-time overhead: obtaining state requires a copy from the kernel to user space. In our evaluation, we see that user-level network services can see an increase in CPU overhead if they frequently poll icTCP for state information.

Therefore, for efficiency, icTCP provides both a polling and an interrupt interface. Given that most TCP variables are updated only when an acknowledgment arrives or at the end of a round, applications can block until either an ACK is received or a round ends. At this point, the application will likely poll icTCP for the information of interest, determine how it would like to set the TCP variables for new control, and then call icTCP to set these variables.

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 new extensions on top of icTCP; 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 [24], 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 commonly need to examine information about a particular message. Therefore, icTCP exposes "standard" information about each packet (*e.g.*, its sequence number and round-trip time and whether it is being sent for a timeout or a fast retransmit) in a message list. Information about incoming acknowledgments (*e.g.*, the acknowledgment number and DSACK information) is similarly made available via an ack list.

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.

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	$x - rcv.wnd \leq v \leq x +$ 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 unack'ed seq num	$x \leq v \leq$ vsnd.next	Reject ack; enter FRFR
dupthresh	Duplicate threshold	$1 \leq v \leq$ vcwnd	Enter FRFR
RTO	Retransmission timeout	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 restransmit/fast recovery; finally, srtt and rttvar represent smoothed round-trip time and round-trip time variance, respectively.*

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 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*. 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 raises above cwnd, the value of cwnd is used instead.

Adding a virtual variable is not as trivial as it may appear; that is, 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 are harder to track and currently require careful manual inspection. Therefore, we limit the extent to which the original variable is replaced with the virtual variable.

We have analyzed many of the variables in the Linux TCP implementation to determine how each can be safely set. We have identified the settings for 10 interesting variables, as summarized in Table 1. Our terminology is that 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. 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 10 variables represent a complete collection of settable values, but that they do form a useful set.

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 [55]. 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 [54]. 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. In most scenarios, the sending endpoints run icTCP, whereas the receivers run stock Linux 2.4.18. In some experiments, icTCP is run on both senders and receivers.

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 NistNet [13] on the router nodes to emulate more complex behaviors such as packet reordering.

In most experiments, we vary the bottleneck bandwidth, delay, and/or maximum queue size through the intermediate router nodes. Experiments are run multiple times (usually 30) and averages are reported. Variance was low in those cases where it is not shown. In general, details of each experiment can be found in figure captions or within the text that describes the particular experiment.

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? 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 most of the remaining paper addressing the fourth question concerning the range of extensions that can be implemented.

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. These small changes make icTCP a highly practical approach and greatly increases its chances of being deployed in real systems.

5.2 Network Safety

Next, we investigate whether icTCP flows are TCP friendly. We empirically demonstrate that icTCP flows are not aggressive when competing with other TCP flows. In our experiments, one TCP Reno flow between one pair of hosts competes with one icTCP flow on another pair of hosts; the two flows share a single link. The achieved bandwidth of each flow is measured at the second shared router. To confirm that icTCP is friendly, we check that the icTCP flow does not obtain more of the available bandwidth than the TCP Reno flow, as a function of the packet loss rate.

To stress the safe setting of icTCP variables, we construct a user-level process that sets the virtual variables to random values; the distribution of random values is such that 10% of the values fall beneath the minimum safe value, 10% fall above the maximum safe value, and the remaining 80% are safe. As explained previously, when a user specifies an unsafe value for a parameter, icTCP coerces that parameter within the safe range. We have analyzed controlling each of the icTCP parameters in isolation as well as sets of the parameters simultaneously. In all cases, the icTCP flow obtains bandwidth that is either lower or nearly equal to the competing Reno flow.

Figure 2 shows a small subset of our safety results. The dark lines in the middle of each graph report the bandwidth achieved for the Reno TCP flow and for the icTCP flow with no user settings; as expected, the bandwidth of the two flows is nearly identical and drops as the packet loss rate increases. Across the graphs, we vary which icTCP parameters are randomly set. In the first graph, only one parameter, snd.una, is randomly set; we see that when snd.una is set, the icTCP flow obtains less bandwidth than previously and the Reno flow achieves more. In the second graph, we summarize results from a number of independent experiments, in each case randomly setting a single icTCP parameter. For clarity, we show only the bandwidth delivered to the icTCP connection; the bandwidth

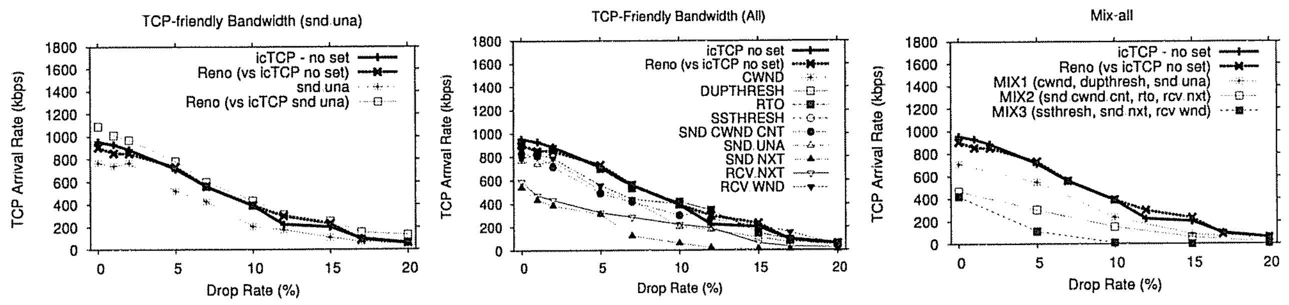

Figure 2: **Network Safety of icTCP.** *In these experiments, we vary the packet loss rate as one TCP Reno flow competes with one icTCP flow. Each data point is the mean of 5 random samples, each from a 10 second run. The dark lines in the middle of each graph report the bandwidth achieved for the Reno TCP flow and for the icTCP flow with no user settings. In the first graph, only one parameter, snd.una, is randomly set. In the second graph, results for a number icTCP parameters are shown. Finally, in the third graph, we consider the case where multiple icTCP parameters are set simultaneously.*

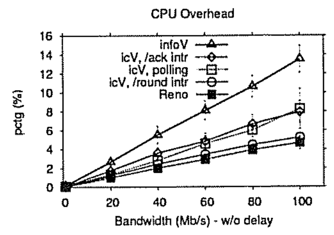

Figure 3: **icTCP$_{Vegas}$CPU Overhead.** *The figure compares the overall CPU utilization of Reno, infoVegas, and the three versions of icTCP$_{Vegas}$. Along the x-axis, we vary the bandwidth of the bottleneck link.*

for the competing Reno flow increases by the expected amount. This second graph shows that icTCP is not aggressive when any single variable is set. Finally, in the third graph, we consider the case where multiple icTCP parameters are set simultaneously. As desired, each combination with icTCP remains less aggressive than the TCP Reno flow. Although these experiments do not prove that icTCP ensures network safety, these measurements along with our analysis give us high confidence that icTCP can be safely deployed.

5.3 CPU Overhead

We evaluate the overhead imposed by the icTCP framework using a user-level implementation of TCP Vegas [11]. In our implementation of icTCP$_{Vegas}$, we explore ways in which the user-level library can reduce overhead by minimizing its interactions with the kernel.

Overview: We focus on the most interesting difference between TCP Vegas and Reno: congestion avoidance. TCP Vegas reduces latency and increases overall throughput 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 decreases its congestion control window, *cwnd*, by one.

Implementation: Our implementation of the Vegas congestion control algorithm is structured as follows. The operation of Vegas is placed in a user-level library, libTCP$_{Vegas}$. 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, libTCP$_{Vegas}$ 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 [5]. The primary difference between the two is that because INFOTCP provides only information and not control, INFOVEGAS must manage the functionality of *vcwnd* for itself. 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 3 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 in-

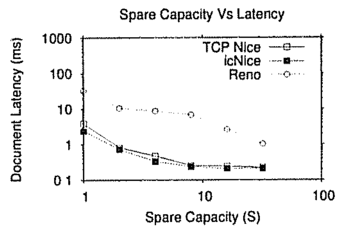

Figure 4: **icTCP$_{Nice}$: Spare capacity vs. Foreground Latency.** *A foreground flow competes with many background flows. Each line corresponds to a different run of the experiment with a protocol for background flows (i.e., icTCP, TCP Nice, Reno, or Vegas). The y-axis shows the average document transfer 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 spare capacity, S, of the network is varied along the x-axis. The spare capacity is achieved by setting the bottleneck link bandwidth to (1 + S) multiplied by the total number of bytes transferred in the trace divided by the duration of the trace.*

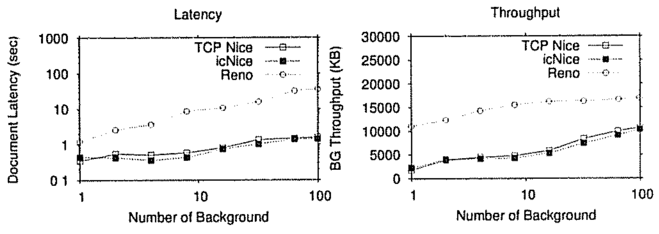

Figure 5: **icTCP$_{Nice}$: Impact of Background Flows.** *The two graphs correspond to the same experiment; the first graph shows the average document latency for the foreground traffic while the second graph shows aggregate throughput for background traffic. Each line corresponds to a different protocol for background flows (i.e., TCP Reno, icTCP$_{Nice}$, or TCP Nice). The number of background flows is varied along the x-axis. The spare capacity, S, of the network is set to 2. The experimental setup is identical to Figure 4.*

formation 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 six 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 [38].

5.4.1 icTCP$_{Nice}$

In our first case study, we show that TCP Nice [50] 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: 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 affect 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 used for TCP Nice (*i.e.*, Figures 2, 3, and 4 from [50]). Our results show that icTCP$_{Nice}$ performs almost identically to the in-kernel TCP Nice, as desired.

Figure 4 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 5 show the latency of foreground connections and the throughput of background connections as the number of background connections increases. The graph on the left 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 right shows that icTCP$_{Nice}$ and TCP nice obtain more throughput as the number of flows increases. As desired, again both icTCP$_{Nice}$ and TCP Nice achieve similar results.

5.4.2 icCM

We show that some important components of the Congestion Manager (CM) [6] 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. Further, we show that different transport protocols (*e.g.*, TCP and UDP) can cooperate as well.

Overview: The Congestion Manager (CM) architecture [6] 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 the IP layer 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 PID and the destination address with the icCM server.

To track statistics, each client flow periodically obtains

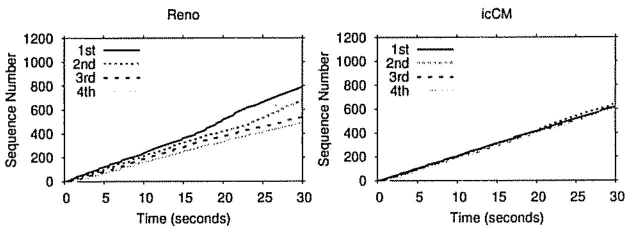

Figure 6: **icCM Fairness.** *The two graphs compare the performance of four concurrent transfers from one sender to one receiver, with the bottleneck link set to 1 Mb/s and a 120 ms delay. On the left, stock Reno is used, whereas on the right, icCM manages the four TCP flows.*

its own network state from icTCP (*e.g.*, its number of outstanding bytes, *snd.nxt - snd.una*) and shares this 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 [6]). Figure 6 shows two graphs of sequence number traces. In each graph, there are four flows within a macroflow; across graphs, we vary whether TCP Reno or TCP with icCM is used. The first graph shows that the four TCP Reno flows do not share the available bandwidth fairly; the performance of the four connections varies between 39 KB/s and 24 KB/s, a factor of 1.6 in transfer time between the fastest and slowest connections. The second graph shows that the four TCP icCM connections progress at very similar and consistent rates; all four connections achieve throughputs of roughly 30 KB/s.

We also performed the same experiment, but with a congestion-controlled UDP flow in the mix; the results were identical and are not shown due to lack of space. However, for UDP to work within our framework, it must also have some form of congestion control, the topic of the next subsection.

5.4.3 icUDP$_{CC}$

The previous case studies showed that icTCP can be used to change the congestion control behavior of a TCP con-

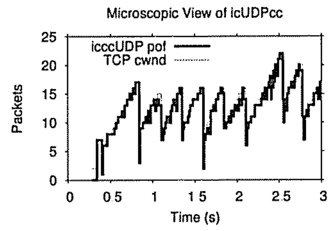

Figure 7: **Microscopic View of icUDP$_{CC}$.** *The y-axis shows the number of packets on the fly for icUDP$_{CC}$ and the TCP probe connection. In this setup, icUDP$_{CC}$ is competing with a TCP flow; the icUDP$_{CC}$ and TCP endpoints are all distinct, but an intermediate link between all four nodes with a limited bandwidth of 2 MB/s is shared.*

nection. In this case study, we show that icTCP can also be used to add congestion control to other protocols. Specifically, we show that we can easily build an unreliable UDP flow with congestion control, and that this congestion control can be based on Reno or Vegas. Again, the icTCP approach helps ensure that a correct congestion control algorithm is deployed, since one is able to directly leverage an existing, tested TCP implementation.

Overview: Applications that desire timeliness of data over reliability have typically used UDP, despite its lack of congestion control. However, the presence of a large number of flows without congestion control leads to clear problems in the Internet. Protocols designed to add congestion control to UDP, such as DCCP [28], have turned out to be more complex than expected; for example, DCCP requires a sequence number space to communicate packet arrivals and losses, a feedback channel to convey congestion information back to the sender, and well-defined mechanisms to set up and cleanly tear down state.

Implementation: To build icUDP$_{CC}$, the key insight is that one can use information obtained from a TCP flow along the same network path to determine the amount of UDP data that should be sent. Thus, we associate a icTCP flow between the same sender and destination as the icUDP$_{CC}$ flow. One disadvantage of this approach is that we cannot ensure that these two flows are routed along the same paths; however, given current routing algorithms, this assumption often holds [23, 45, 48].

A second disadvantage is that the icTCP flow imparts a small amount of additional traffic on the network: each time icUDP$_{CC}$ sends a UDP packet, it also sends a one byte icTCP packet. However, given large UDP packets, the amount of additional traffic is small. This pair of icTCP and UDP packets behaves similarly to a TCP packet of the total size. Given that the TCP congestion window reflects the number of TCP packets that can be sent, icUDP$_{CC}$ obtains *cwnd* from icTCP and ensures that the corresponding amount of UDP data is sent. However, this approach raises two issues.

First, there is no existing mechanism to determine the number of UDP packets on the fly. Therefore, we added an ack-mechanism within the libUDP$_{CC}$ application layer: icUDP$_{CC}$ adds a UDP sequence number to each packet and the receiver acknowledges its highest-numbered UDP packet. libUDP$_{CC}$ uses these acks to determine how many packets are outstanding and either waits or sends more packets accordingly.

Second, lost UDP packets may cause libUDP$_{CC}$ to have an inaccurate tally of the number of packets on the fly. Specifically, if either the last UDP packet or ack lost, icUDP$_{CC}$ will believe more packets are still in transit and will send fewer UDP packets than allowed; in general, this is only a transient problem, since the next received UDP ack allows icUDP$_{CC}$ to catch up. However, a problem does occur for very small congestion windows: if all UDP acks are lost, libUDP$_{CC}$ will never wake. This problem is solved with a standard time-out mechanism.

One of the advantages of the icTCP approach is that each library can form a new layer in the system. Thus, icUDP$_{CC}$ can easily leverage the congestion control algorithm implemented by any of our existing layers. For example, by sending the icTCP probe packets through libTCP$_{Vegas}$ instead of icTCP, we have built icUDP$_{CC,Vegas}$, UDP with Vegas congestion control. Alternatively, by registering with the icCM server, a UDP flow can fairly share bandwidth with TCP flows. Specifically, the UDP client uses icTCP probe packets to obtain the statistics to share with the icCM server; the UDP client then subtract its number of bytes on the fly from its fair allocation and ensures that only that many bytes are sent.

Evaluation: To demonstrate the effectiveness of icUDP$_{CC}$, we show that it behaves as expected, that it incurs minimal overhead, that icUDP$_{CC}$ shares bandwidth fairly with TCP flows, and that the congestion control algorithm can be easily changed.

First, we demonstrate that icUDP$_{CC}$ obtains the expected AIMD (Additive Increase Multiplicative Decrease) behavior of the Reno congestion control algorithm. Figure 7 shows the the number of packets on the fly for icUDP$_{CC}$ and *cwnd* for the TCP probe connection. These results show that the two connections closely follow one another (in this experiment, the difference is never more than four); as a result, the icUDP$_{CC}$ flow sends a steady stream of packets, as desired.

Second, we show that the overhead incurred by icUDP$_{CC}$ is minimal. Deployed in an environment without contention, icUDP$_{CC}$ and icUDP$_{CC,Vegas}$ have slowdowns of 5.9% and 6.6%, respectively, relative to UDP. Given that the default TCP Reno exhibits a slowdown of 1.6%, some of the overhead is simply due to the cost of providing congestion control. The additional slowdown of the icUDP$_{CC}$ variants is due to the insertion of probe packets in the network and additional computation performed at the user level. We believe that providing congestion control is worth this degradation in bandwidth.

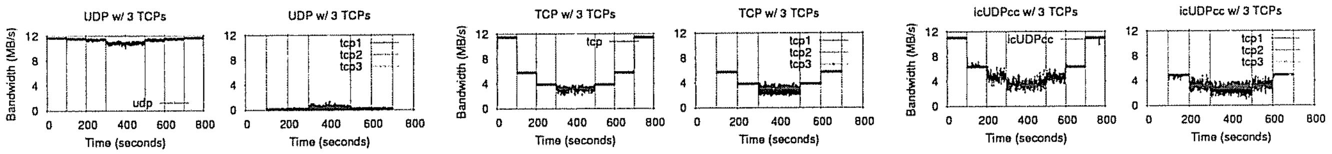

Figure 8: **icUDP$_{CC}$ Fairness.** *The three pairs of graphs correspond to three experiments. In each experiment, one UDP, one TCP, or one icUDP$_{CC}$ connection competes with three TCP connections. The first graph in each pair shows the bandwidth achieved by the UDP, TCP, or icUDP$_{CC}$ connection;the second graph in each pair shows the bandwidth achieved by the three TCP connections. The topology contains two sender/receiver pairs; two TCP flows run between one pair and the third TCP connection and the experimental flow runs between the other pair. The UDP, TCP, or icUDP$_{CC}$ connection runs the entire time. The two TCP connections that share a sending and destination node start sending at 100 and 200 seconds, and finish at 700 and 600 seconds, respectively; the third TCP connection starts at 300 seconds and ends at 500 seconds. The bottleneck bandwidth is set to 100 Mb/s with no delay.*

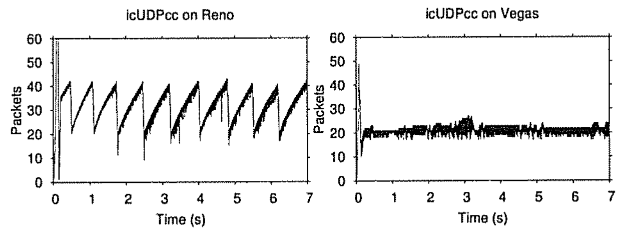

Figure 9: **icUDP$_{CC}$ and icUDP$_{CC,Vegas}$.** *The figures show the behavior of icUDP$_{CC}$ when the control channel is built upon the stock kernel Reno implementation (left) and on top of the icTCP$_{Vegas}$ layer (right). In both experiments, the number of outstanding packets is graphed over time, and the bottleneck link is set to a 20 Kb/s throughput and 5 ms delay.*

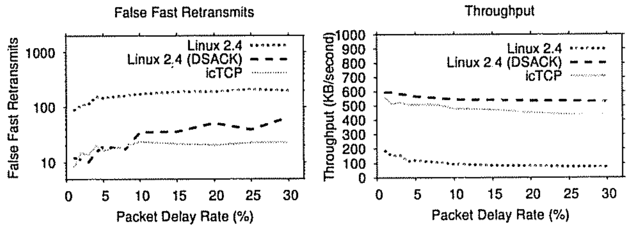

Figure 10: **Avoiding False Retransmissions with icTCP$_{RR}$.** *On the left is the number of false retransmissions and on the right is throughput, both as we vary the fraction of packets that are delayed (and hence reordered) in our modified NistNet router. We compare three different implementations, 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.*

Third, we demonstrate that icUDP$_{CC}$ fairly shares network bandwidth with competing TCP flows. Figure 8 shows how bandwidth is divided when UDP, TCP, or icUDP$_{CC}$ compete against three TCP flows. The first pair of graphs show that one UDP connection sending a large amount of data prevents the three TCP connections from obtaining much bandwidth. The second pair of graphs show that when all four connections use TCP, the delivered bandwidth is approximately equal. Finally, the third pair of graphs shows that one icUDP$_{CC}$ connection competing with TCP behaves similarly to TCP; that is, the icUDP$_{CC}$ flow backs off when there is contention. However, we note that icUDP$_{CC}$ does not behave exactly like TCP. In the TCP graphs, bandwidth is smooth with three total connections because the available bandwidth (100 Mb/s) is not saturated; it is only with four TCP connections (from 30 to 50 seconds), that fluctuations occur. In the icUDP$_{CC}$ graphs, bandwidth fluctuations occur when less contention exists.

Finally, we show that icUDP$_{CC}$ can be configured to use different types of congestion control. Figure 9 shows over a longer time-scale that icUDP$_{CC}$ follows the AIMD behavior of the TCP Reno congestion window whereas icUDP$_{CC,Vegas}$ follows TCP Vegas behavior.

5.4.4 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 [4, 25]. However, recent research indicates that packet reordering may be more common in the Internet than earlier designers suspected [3, 7, 9, 55]. 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).

A number of solutions for handling duplicate acknowledgments have been suggested in the literature [9, 55]. 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.

Overview: icTCP$_{RR}$ is a user-level library that is robust to packet reordering; we base our implementation on that of Blanton and Allman's work [9], although more sophisticated approaches have been proposed [55]. icTCP$_{RR}$ mimics Blanton and Allman's proposal quite closely: we limit the maximum value of dupthresh to 90% of the window size and when a timeout occurs, dupthresh is set back to its original setting of 3.

Implementation: The library implementation of icTCP$_{RR}$, libTCP$_{RR}$, is straightforward. The library keeps a history of acks received; this list is larger than the ker-

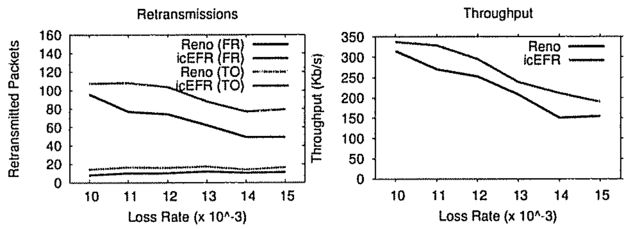

Figure 11: **Aggressive Fast Retransmits with icTCP$_{EFR}$.** *On the left is the number of retransmitted packets for both Reno and icTCP$_{EFR}$ – due to both retransmission timeouts (TO) and fast retransmits (FR) – and on the right is the achieved bandwidth. Along the x-axis, we vary the loss rate 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.*

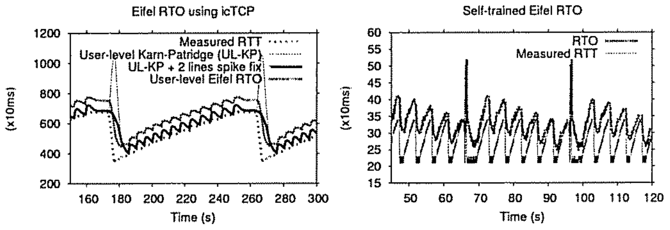

Figure 12: **Adjusting RTO with icTCP$_{Eifel}$.** *The graph on the left shows three versions of libTCP$_{Eifel}$. For each experiment, the measured round-trip time is identical; however, the calculated RTO differs. The first line shows when the karn-partridge RTO algorithm [27] is disabled in the kernel 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 right shows the full adaptive Eifel RTO algorithm with a bandwidth of 1000 kbps, 100 ms delay, and a queue size of 12.*

nel 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 will search through past history to measure the reordering length, and thus set dupthresh accordingly.

Evaluation: Figure 10 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 libTCP$_{RR}$ implementation. On the left, 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. On the right, we observe the resulting bandwidth. Here, the DSACK in-kernel and libTCP$_{RR}$ versions perform much better, essentially ignoring duplicate acks and thus achieving much higher bandwidth.

5.4.5 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 [47]. 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 that interprets small numbers of duplicate acks as strong signals of loss and thus triggers fast retransmit aggressively [47]. The observation underlying EFR is simple: the sender should adjust dupthresh so as to meet the number of duplicate acks it could receive. This optimization is of particular importance in a wireless setting, where a small window size makes it unlikely that the default setting of 3 can detect loss and thus trigger fast retransmit.

Implementation: The libTCP$_{EFR}$ 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 a new value for dupthresh and sets it.

Evaluation: Figure 11 shows the behavior of libTCP$_{EFR}$ versus the in-kernel Reno as a function of loss rate in an emulated wireless network. Because libTCP$_{EFR}$ interprets duplicate acknowledgments as likely signs of loss, the number of fast retransmits increases (as shown in the graph on the left) and more importantly, the number of costly retransmission timeouts is reduced; the graph on the right shows that achieved bandwidth increases as a result.

5.4.6 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; a too conservative RTO may cause long idle times before lost packets are retransmitted.

Overview: The Eifel RTO [30] corrects a number of problems with the traditional karn-partridge RTO [27]. First, immediately after mRTT decreases, RTO is incorrectly increased; after some period of time, the value of RTO decays to the correct value again. Second, the "magic numbers" in the RTO calculation assume a low mRTT sampling rate and low 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, libTCP$_{Eifel}$. This library needs access to three icTCP variables: mRTT, ssthresh, and cwnd; from mRTT, it calculate its own values of srtt (smoothed round-trip) and rttvar (round-trip variance). The libTCP$_{Eifel}$ library operates as follows: it wakes when an acknowledgment arrives, polls icTCP for the new mRTT; if mRTT has changed, it calculates the new RTO and sets it with icTCP. Thus, this library requires safe control over RTO.

Evaluation: We have verified that libTCP$_{Eifel}$ corrects these RTO problems and behaves identically as when these fixes are implemented in the kernel; due to space constraints, we do not show the (identical) in-kernel behavior. The progression of libTCP$_{Eifel}$ improvements is shown in the first graph of Figure 12; these experiments were chosen to approximately match those in the Eifel RTO paper (*i.e.*, Figure 6). In the first experiment, we disable the karn-partridge RTO algorithm [27] in the kernel and show that karn-partridge RTO can be implemented at user-level with icTCP; as expected, this version incorrectly increases RTO when mRTT decreases. The second version corrects this problem, but RTO eventually collapses into mRTT. Finally, the third version of libTCP$_{Eifel}$ adjusts RTO so that it is more conservative and can avoid spurious retransmissions.

The second graph of Figure 12 is similar to Figure 10 of 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.

5.4.7 Discussion

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 easily enables functional compositions: for example, we can plug in our icTCP$_{Vegas}$ library underneath of our icUDP$_{CC}$ library. Third, icTCP is ideally suited for tuning parameters whose optimal values depend upon the environment and the workload (*e.g.*, the value of dupthresh). Finally, 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. From icUDP$_{CC}$, we saw that sequence numbers and acknowledgments had to be added the data payload, requiring both sender and received to utilize the icUDP$_{CC}$ library. Further, icUDP$_{CC}$ requires a separate TCP channel along the same path; this may lead to additional overhead as well as inaccuracy.

5.4.8 Other Extensions

We evaluate the ability of icTCP to implement a wider range of TCP extensions by considering the list discussed for STP [38]. Of the 27 extensions, 8 have already been standardized in Linux 2.4.18 (*e.g.*, SACK, DSACK, FACK, ECN, New Reno, and SYN cookies) and 5 have been implemented with icTCP (*i.e.*, RR-TCP, DCCP, Vegas, CM, and Nice). We discuss some of the challenges in implementing the remaining 14 extensions.

Packet Format Modifications: The icTCP framework does not allow changes in the format or content of packets. For example, an extension that puts new bits into the TCP reserved field (*e.g.*, the Eifel algorithm [26]) cannot be implemented easily with icTCP. However, one approach we can take is to encapsulate extra information within application data; while requiring both sender and receiver to use an icTCP-enabled kernel and the appropriate library, this technique allows extra information to be passed between protocol stacks while remaining transparent to applications. Another solution we plan to investigate is how icTCP can allow header information to be safely added or modified.

Low-level Protocol Modifications: Another limitation of the icTCP approach is its inability to directly alter the low-level protocol that the stock TCP or UDP implements. For example, if receipt of a message generates an acknowledgment, the user-level library on top of the protocol cannot alter this basic behavior. One idea is to place a packet filter [36] beneath the kernel stack and to allow a library some control over its own packets, perhaps changing the timing, ordering, or altogether suppressing or duplicating some subset of packets as they pass through the filter. However, such control must be meted out with caution; ensuring such changes remain TCP friendly is a central challenge.

Of the remaining 14 extensions, we believe that icTCP provides sufficient information and control to implement 6, to at least some extent (*i.e.*, limited transmit [3], robust congestion signaling [17], appropriate byte counting [2], Eifel [26], equation-based TCP [20], and TCP Westwood [53]). However, the icTCP versions of many of these extensions will be more conservative than their native implementations. For example, equation-based TCP specifies that the congestion window should increase and

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 [12]. 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.*

decrease more gradually than Reno; icTCP$_{Eqn}$ will allow cwnd to increase more gradually, as desired, but will force cwnd to decrease at the usual Reno rate. Nevertheless, limited implementations of these extensions can still be beneficial. For example, even though appropriate byte counting (ABC) implemented on icTCP cannot aggressively increase cwnd when a receiver delays an ack, icTCP$_{ABC}$ can still correct for ack division.

In summary, approximately 8 of the 27 extensions cannot be implemented with icTCP because the extensions either do not follow the existing TCP states (*e.g.*T/TCP [10]) or they define a new mechanism (*e.g.*, SCTP checksum [46]). Even though icTCP is not as flexible as STP [38], we believe that the simplicity of providing an icTCP layer far more than outweighs this drawback.

5.5 Ease of Development

Finally, we address the question of how the icTCP framework simplifies or complicates the development of TCP extensions relative to directly extending TCP in the kernel. To quantify complexity, 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 three 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 comparable. Thus, we conclude that developing services using icTCP is no more complex than building them natively and has the advantage that debugging and analysis can be performed at user-level.

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.

Our overall conclusion is that icTCP is not quite as powerful as other proposals for extending TCP or other networking protocols [38, 35]. 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.

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, V. Paxson, and W. R. Stevens. TCP Congestion Control. RFC 2581, Internet Engineering Task Force, Apr. 1999.

[5] 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.

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

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

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

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

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

[11] 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.

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

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

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

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

[16] 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.

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

[18] 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.

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

[20] 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.

[21] 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.

[22] 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.

[23] I. Gojmerac, T. Ziegler, F. Ricciato, and P. Reichl. Adaptive Multipath Routing for Dynamic Traffic Engineering. *IEEE Globecom '03*, 2003.

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

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

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

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

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

[29] 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.

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

[31] 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.

[32] 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.

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

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

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

[36] 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.

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

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

[39] 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.

[40] 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.

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

[42] S. Savage. Sting: a TCP-based Nework Performance Measurement Tools. In *Proceedings of the 2rd USENIX Symposium on Internet Technologies and Systems (USITS '99)*, pages 71–79, Boulder, Colorado, Oct. 1999.

[43] S. Savage, N. Cardwell, and T. Anderson. The Case for Informed Transport Protocols. In *Workshop on Hot Topics in Operating Systems*, pages 58–63, Rio Rico, Arizona, Mar. 1999.

[44] 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.

[45] G. Siganos and M. Faloutsos. BGP Routing: A Study at Large Time Scale, 2002.

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

[47] 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.

[48] R. Teixeira, K. Marzullo, S. Savage, and G. M. Voelker. In Search of Path Diversity in ISP Networks. In *Proceedings of the Internet Measurement Conference*, pages 313–318, 2003.

[49] 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.

[50] 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.

[51] 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.

[52] 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.

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

[54] 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.

[55] 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.