On Estimating End-to-End Network Path Properties

Mark Allman
NASA Glenn Research Center
and
GTE Internetworking
21000 Brookpark Rd. MS 54-2
Cleveland, OH 44135
mallman@grc.nasa.gov

Vern Paxson
AT&T Center for Internet Research at ICSI
and
Lawrence Berkeley National Labratory
1947 Center Street, Suite 600
Berkeley, CA 94704-1198
vern@aciri.org

Abstract

The more information about current network conditions available to a transport protocol, the more efficiently it can use the network to transfer its data. In networks such as the Internet, the transport protocol must often form its own estimates of network properties based on measurements performed by the connection endpoints. We consider two basic transport estimation problems: determining the setting of the retransmission timer (RTO) for a reliable protocol, and estimating the bandwidth available to a connection as it begins. We look at both of these problems in the context of TCP, using a large TCP measurement set [Pax97b] for trace-driven simulations. For RTO estimation, we evaluate a number of different algorithms, finding that the performance of the estimators is dominated by their minimum values, and to a lesser extent, the timer granularity, while being virtually unaffected by how often round-trip time measurements are made or the settings of the parameters in the exponentially-weighted moving average estimators commonly used. For bandwidth estimation, we explore techniques previously sketched in the literature [Hoe96, AD98] and find that in practice they perform less well than anticipated. We then develop a receiver-side algorithm that performs significantly better.

1 Introduction

When operating in a heterogeneous environment, the more information about current network conditions available to a transport protocol, the more efficiently it can use the network to transfer its data. Acquiring such information is particularly important for operation in wide-area networks, where a strong tension exists between needing to keep a large amount of data in flight in order to fill the bandwidth-delay product "pipe," versus having to wait lengthy periods of time to attain feedback regarding changing network conditions, especially the onset of congestion.

In a wide-area network, such as the Internet, that does not provide any explicit information about the network path, it is up to the transport protocol to form its own estimates of current network conditions, and then to use them to adapt as efficiently as possible. A classic example of such estimation and adaptation is how TCP infers the presence of congestion along an Internet path by observing packet losses, and either cuts its sending rate in the presence of congestion, or increases it in the absence [Jac88].

In this paper we examine two other basic transport estimation problems: determining the setting of the retransmission timer (RTO), and estimating the bandwidth available to a connection as it begins. We look at both problems in the context of TCP, using trace-based analysis of a large collection of TCP packet traces. The appeal of analyzing TCP in particular is that it is the dominant protocol in use in the Internet today [TMW97]. However, analyzing the behavior of actual TCP implementations also introduces complications, because there are a variety of different TCP implementations that behave in a variety of different ways [Pax97a]. Consequently, in our analysis we endeavor to distinguish between findings that are specific to how different TCPs are implemented today, versus those that apply to general TCP properties, versus those that apply to general reliable transport protocols.

Our analysis is based on the $\mathcal{N}_2$ subset of TCP trace data collected in 1995 [Pax97b]. This data set consists of sender-side and receiver-side packet traces of 18,490 TCP connections among 31 geographically-diverse Internet hosts. The hosts were interconnected with paths ranging from 64 kbps up to Ethernet speeds, and each connection transferred 100 KB of data, recorded using *tcpdump*. We modified *tcpanaly* [Pax97a] to perform our analysis.

The rest of the paper is organized as follows. In § 2 we look at the problem of estimating RTO, beginning with discussions of the basic algorithm and our evaluation methodology. We analyze the impact of varying a number of estimator parameters, finding that the one with the greatest effect is the lower bound placed on RTO, followed by the clock granularity, while other parameters have little effect. We then present evidence that argues for the intrinsic difficulty of finding optimal parameters, and finish with a discussion of the cost of retransmitting unnecessarily and ways to detect when it has occurred. In § 3 we look at the problem of estimating the bandwidth available to a connection as it starts up. We discuss our evaluation methodology, which partitions estimates into different regions reflecting their expected impact, ranging from no impact, to preventing loss, attaining steady state, optimally utilizing the path, or reducing performance. We then assess a number of estimators, finding that sender-side estimation such as previously proposed in the literature is fraught with difficulty, while receiver-side estimation can work considerably better. § 4 summarizes the analysis and possible future work.

2 Estimating RTO

For an acknowledgment-based reliable transport protocol, such as TCP, a fundamental question is how long, in the absence of receiving an acknowledgment (ACK), should a sender wait until retransmitting? This problem is similar to that of estimating the largest possible round-trip time (RTT) along an end-to-end network path. However, it differs from RTT estimation in three ways. First, the goal is not to accurately estimate the truly maximal possible RTT, but rather a

Estimación de las propiedades de una ruta de red de terminal a terminal

Mark Allman
Centro de investigación Glenn de la NASA
y
GTE Internetworking
21000 Brookpark Rd, MS 54-2
Cleveland, OH 44135
mallman@grc.nasa.gov

Vern Paxson
Centro de investigación Internet de AT&T en ICSI
y
Laboratorio nacional Lawrence de Berkley
1947 Center Street, Suite 600
Berkley, CA 94704-1198
vern@aciri.org

Resumen

Cuanta más información exista sobre las condiciones de red actuales que están disponibles en el protocolo de transporte, más eficientemente podrá usar éste la red para transmitir los datos. En redes como Internet, el protocolo de transporte debe elaborar a menudo sus propias estimaciones acerca de las propiedades de la red basándose en mediciones efectuadas por los puntos terminales de la conexión. Consideramos dos problemas básicos en la estimación del transporte: determinar la configuración del temporizador de retransmisión (RTO) para obtener un protocolo de confianza, y estimar el ancho de banda disponible al inicio de la conexión. Investigamos ambos problemas en el contexto del TCP, utilizando un conjunto amplio de mediciones para simulaciones de TCP [Pax97b] tomadas mediante trazas. Para la estimación del RTO, evaluamos varios algoritmos diferentes, concluyendo que la eficacia de las estimaciones está dominada por sus valores mínimos, y en menor medida, por la granularidad del temporizador, siendo virtualmente independiente de la frecuencia con que se realizan las mediciones en el tiempo de recorrido ida-vuelta, o de la configuración de los parámetros de los estimadores de promedio móvil ponderado exponencialmente comúnmente utilizados. Para la estimación del ancho de banda, indagamos en las técnicas planteadas anteriormente en la bibliografía [Hoe96, AD98], llegando a la conclusión de que en la práctica su eficacia está por debajo de lo esperado. Así, desarrollamos posteriormente un algoritmo de fuente receptora que se comporta significativamente mejor.

1 Introducción

Al operar en un entorno heterogéneo, el uso de una red para transferir datos resulta más eficiente cuanto mayor es la información disponible para el protocolo de transporte con respecto a las condiciones vigentes en la red. Obtener esta información resulta particularmente importante para operar en redes de área extendida, donde existe un fuerte antagonismo entre la necesidad de mantener una gran cantidad de datos en curso a fin de rellenar la tubería de producto de retardo por ancho de banda, y el largo tiempo de espera necesario para obtener información relativa a las condiciones cambiantes de red, especialmente al comienzo de una congestión.

En una red de área extendida, como Internet, que no provee ninguna información explícita respecto a la ruta de red, depende únicamente del protocolo de transporte el elaborar sus propias estimaciones acerca de las condiciones vigentes en la red, y luego usarlas para adaptarse de la manera más eficiente posible. Un ejemplo clásico de tal estimación y adaptación es la forma según la cual el TCP infiere la presencia de una congestión a lo largo de una ruta de Internet, midiendo la pérdida de paquetes y disminuyendo la frecuencia de transmisión ante la presencia de congestión, o bien aumentándola en su ausencia [Jac88].

En el presente artículo examinamos otros dos problemas básicos de las estimaciones de transporte: determinar la configuración del temporizador de transporte (RTO) y estimar el ancho de banda disponible al inicio de una conexión. Investigamos ambos problemas en el contexto del TCP, por medio de un análisis basado en trazas correspondientes a un gran conjunto de paquetes de TCP. El interés de analizar precisamente el TCP radica en que es éste el protocolo más usado actualmente en Internet [TMW97]. Sin embargo, el hecho de analizar el comportamiento de implementaciones reales de TCP también conlleva complicaciones, dado que existe una gran variedad de implementaciones TCP que se comportan de modo diferente [Pax97a]. Consecuentemente, en nuestro análisis hacemos hincapié en separar las conclusiones específicas a las distintas maneras de implementar TCP en la actualidad, de aquellas que son aplicables a las propiedades generales del TCP y de aquellas propias de los protocolos de transporte confiables de uso general.

Nuestro análisis está basado en el subconjunto N_2 de trazas TCP recogidas en 1995 [Pax97b]. Este conjunto de datos consiste en trazas de paquetes de fuente emisora y receptora correspondientes a 18,490 conexiones TCP entre servidores Internet ubicados en 31 emplazamientos geográficos diferentes. Los servidores estaban interconectados mediante rutas a velocidades que iban entre los 64 kbps y valores Ethernet, transfiriendo cada conexión 100 KB de información, la cual quedó registrada por medio de *tcpdump*. Para efectuar nuestro análisis modificamos el *tcpanaly* [Pax97a].

El resto del artículo está organizado de la siguiente manera. En § 2 abordamos el problema de la estimación RTO, tratando inicialmente el algoritmo básico y nuestra metodología de evaluación. Analizamos las consecuencias derivadas tras modificar cierto número de parámetros de cálculo, concluyendo que aquel de mayor efecto es el límite inferior establecido para el RTO, seguido por la granularidad del reloj, mientras que otros parámetros carecen de importancia. A continuación presentamos diversos argumentos relativos a la dificultad intrínseca de hallar parámetros óptimos, finalizando con una exposición acerca del coste en el que se incurre con el uso de retransmisiones innecesarias, y las maneras de detectar cuando han ocurrido éstas. En § 3 tratamos el

good compromise that balances avoiding unnecessary retransmission timeouts due to not waiting long enough for an ACK to arrive, versus being slow to detect that a retransmission is necessary. Second, the sender really needs to estimate the *feedback* time, which is the round-trip time from the sender to the receiver *plus* the amount of time required for the receiver to generate an ACK for newly received data. For example, a receiver employing the delayed acknowledgment algorithm [Bra89] may wait up to 500 msec before transmitting an ACK. Thus, estimating a good value for the retransmission timer not only involves estimating a property of the network path, but also a property of the remote connection peer. Third, if loss is due to congestion, it may behoove the sender to wait *longer* than the maximum feedback time, in order to give congestion more time to drain from the network—if the sender retransmits as soon as the feedback time elapses, the retransmission may also be lost, whereas sending it later would be successful.

It has long been recognized that the setting of the retransmission timer cannot be fixed but needs to reflect the network path in use, and generally requires dynamic adaptation because of how greatly RTTs can vary over the course of a connection [Nag84, DDK+90]. The early TCP specification included a notion of dynamically estimating RTO, based on maintaining an exponentially-weighted moving average (EWMA) of the current RTT and a static variation term [Pos81]. This estimator was studied by Mills in [Mil83], which characterizes measured Internet RTTs as resembling a Poisson distribution overall, but with occasional spikes of much higher RTTs, and suggests changing the estimator so that it more rapidly adapts to increasing RTTs and more slowly to decreasing RTTs. (To our knowledge, this modified estimator has not been further evaluated in the literature.) [Mil83] also noted that the balance between responding rapidly in the face of true loss versus avoiding unnecessary retransmissions appears to be a fundamental tradeoff, with no obvious optimal solution.

Zhang [Zha86] discusses a number of deficiencies with the standard TCP RTO estimator: ambiguities in measuring RTTs associated with retransmitted packets; the conservative RTO policy of retransmitting only one lost packet per round-trip; the difficulty of choosing an initial estimate; and the failure to track rapidly increasing RTTs during times of congestion. Karn and Partridge [KP87] addressed the first of these, eliminating ambiguities in measuring RTTs. The introduction of "selective acknowledgments" (SACKs) [MMFR96] addressed the second issue of retransmitting lost packets too slowly. Jacobson [Jac88] further refined TCP RTO estimation by introducing an EWMA estimate of RTT variation, too, and then defining:

$$\text{RTO} = \textit{SRTT} + k \cdot \textit{RTTVAR} \tag{1}$$

where *SRTT* is a smoothed estimate of RTT (as before) and *RTTVAR* is a smoothed estimate of the variation of RTT. In [Jac88], $k = 2$, but this was emended in a revised version of the paper to $k = 4$ [JK92].

While this estimator is in widespread use today, to our knowledge the only systematic evaluation of it against measured TCP connections is our previous study [Pax97b], which found that, other than for over-aggressive misimplementations, the estimator appears sufficiently conservative in the sense that it only rarely results in an unnecessary timeout.

The widely-used BSD RTO implementation [WS95] has several possible limitations: (1) the adaptive RTT and RTT variation estimators are updated with new measurements only once per round-trip, so they adapt fairly slowly to changes in network conditions; (2) the measurements are made using a clock with a 500 msec granularity, which necessarily yields coarse estimates (though [Jac88] introduces some subtle tricks for squeezing more precision out of these estimates); and (3) the resulting RTO estimate has a large minimum value of 1 second, which may make it inherently conservative.

With the advent of higher precision clocks and the TCP "timestamp" option [JBB92], all three of these limitations might be removed. It remains an open question, however, how to best reengineer the RTO estimator given these new capabilities: we know the current estimator is sufficiently conservative, but is it *too* conservative? If so, then how might we improve it, given a relaxation of the above limitations? These are the questions we attempt to answer.

2.1 The Basic RTO Estimation Algorithm

In Jacobson's algorithm, two state variables *SRTT* and *RTTVAR* estimate the current RTT and a notion of its variation. These values are used in Eqn 1 with $k = 4$ to attain the RTO. Both variables are updated every time an RTT measurement RTT_{meas} is taken. Since only one segment and the corresponding ACK is timed at any given time, updates occur only once per RTT (also referred to as once "per flight"). *SRTT* is updated using an EWMA with a gain of α_1:

$$\textit{SRTT} \leftarrow (1 - \alpha_1)\textit{SRTT} + \alpha_1 RTT_{\text{meas}} \tag{2}$$

and Jacobson [Jac88] recommends $\alpha_1 = \frac{1}{8}$, which leads to efficient implementation using fixed-point arithmetic and bit shifting. Similarly, *RTTVAR* is updated based on the deviation $|\textit{SRTT} - \text{RTT}_{\text{meas}}|$ using $\alpha_2 = \frac{1}{4}$.

Any time a packet retransmitted due to the RTO expiring is itself lost, the TCP sender doubles the current value of the RTO. Doing so both diminishes the sending rate in the presence of sustained congestion, and ameliorates the possible adverse effects of underestimating the RTO and retransmitting needlessly and repeatedly.

SRTT and *RTTVAR* are initialized by the first RTT_{meas} measurement using $\textit{SRTT} \leftarrow \text{RTT}_{\text{meas}}$ and $\textit{RTTVAR} \leftarrow \frac{1}{2}\text{RTT}_{\text{meas}}$. Prior to the first measurement, RTO $= 3$ sec.

Two important additional considerations are that all measurement is done using a clock *granularity* of G seconds, i.e., the clock advances in increments of G,[1] and the RTO is *bounded* by RTO_{min} and RTO_{max}. In the common BSD implementation of TCP, $G = 0.5$ sec, $\text{RTO}_{\text{min}} = 2G = 1$ sec, and $\text{RTO}_{\text{max}} = 64$ sec. As will be shown, the value of RTO_{min} is quite significant. Also, since the granularity is coarse, the code for updating *RTTVAR* sets a minimum bound on *RTTVAR* of G, rather than the value of 0 sec that can often naturally arise.

Three oft-proposed variations for implementing the RTO estimator are to time every segment's RTT, rather than only one per flight; use smaller values of G; and lower RTO_{min} in order to spend less time waiting for timeouts. RFC 1323 [JBB92] explicitly supports the first two of these, and our original motivation behind this part of our study was to evaluate whether these changes are worth pursuing.

2.2 Assessing Different RTO Estimators

There are two fundamental properties of an RTO estimator that we investigate: (1) how long does it wait before retransmitting a lost packet? and (2) how often does it expire mistakenly and unnecessarily trigger a retransmit? A very conservative RTO estimator might simply hardwire RTO $= 60$ sec and never make a mistake, satisfying the second property, but doing extremely poorly with regards to the first, leading to unacceptable delays; while a very aggressive estimator could hardwire RTO $= 1$ msec and reverse this relationship, flooding the network with unnecessary retransmissions.

Our basic approach to assess these two properties is to use trace-driven simulation to evaluate different estimators, using the following methodology, which mirrors the RTO estimator implementation in [WS95]:

1. For each data packet sent, if the RTO timer is not currently active, it is started. The timer is also restarted when the data packet is the beginning of a retransmission sequence.

[1] The BSD timer implementation also uses a "heartbeat" timer that expires every G seconds with a phase independent of when the timer is actually set. We included this behavior in our simulations.

problema de estimar el ancho de banda disponible para una conexión al inicio de la misma. Presentamos nuestra metodología de evaluación, la cual divide los resultados en diferentes regiones que reflejan el impacto esperado, desde carencia total de impacto, hasta la prevención de pérdida, pasando por alcance del estado continuo, utilización óptima de la ruta o reducción de la eficiencia. Posteriormente, evaluamos varios estimadores, encontrándonos con que la estimación en la fuente emisora, tal y como se documenta en previos estudios, resulta difícil de alcanzar, mientras que la estimación en fuente receptora puede funcionar considerablemente mejor. En § 4 se resume el trabajo realizado y las posibilidades de investigación futura.

2 Estimación del RTO

Para un protocolo de transporte seguro basado en reconocimientos, tal como el TCP, una pregunta fundamental es: ¿durante cuánto tiempo, en ausencia de recepción del reconocimiento (ACK), debe esperar un emisor para proceder con la retransmisión? Este problema es similar al de estimar el tiempo de recorrido ida-vuelta (RTT) más largo posible a lo largo de una ruta de red de terminal a terminal. Sin embargo, difiere de la estimación del RTT en tres aspectos. Primeramente, el objetivo no es calcular exactamente el verdadero RTT máximo posible, sino lograr un buen compromiso que equilibre la exigencia de evitar expiraciones de conexión a causa de retransmisiones innecesarias, debido a no esperar el suficiente tiempo para la llegada del ACK, con la prevención de no ser demasiado lentos en detectar que una retransmisión es necesaria. En segundo lugar, el emisor necesita realmente estimar el tiempo de *respuesta*, que es igual al tiempo de recorrido ida-vuelta entre emisor y receptor *más* la cantidad de tiempo requerida por el receptor para generar un ACK por cada nuevo dato recibido. Por ejemplo, un receptor que emplea el algoritmo de reconocimiento retardado [Bra89] puede esperar hasta 500 mseg. antes de transmitir el ACK. Por lo tanto, estimar un buen valor para el temporizador de retransmisión no sólo implica el estimar una propiedad de la ruta de la red, sino también una propiedad del par de conexión remoto. En tercer lugar, si la pérdida es debida a congestión, puede competer al emisor esperar *por encima* del tiempo de respuesta máximo a fin de garantizar un plazo mayor para que la congestión desaparezca de la red (si el emisor retransmite tan pronto como ha transcurrido el tiempo de respuesta, la retransmisión también puede perderse, mientras que un envío posterior podría resultar más exitoso).

Desde hace ya tiempo se ha reconocido que la configuración del temporizador de retransmisión no puede ser fija, sino que debe reflejar la ruta de red en uso, requiriendo generalmente una adaptación dinámica debido a lo mucho que puede variar el RTT durante el transcurso de una conexión [Nag84, DDK+90]. La especificación inicial del TCP incluía un concepto para la estimación dinámica del RTO basada en mantener un promedio móvil ponderado exponencialmente (EWMA) del RTT presente, y un término de variación estático [Pos81]. Esta fórmula fue estudiada por Mills en [Mil83], en la que describe las mediciones de RRT de Internet de forma similar, generalmente, a una distribución de Poisson, pero con picos ocasionales de RTT mucho más elevados, sugiriendo modificar el algoritmo de manera que se adaptase más rápidamente al incremento del RTT, y más despacio a su disminución. (Según nos consta, la bibliografía no indica que este cálculo modificado se haya evaluado con mayor detalle). [Mil83] también observó que el compromiso entre responder rápidamente a una pérdida real y el evitar retransmisiones innecesarias parece ser un conflicto de intereses fundamental, sin una solución óptima que resulte obvia.

Zhang [Zha86] describe diversas deficiencias en el cálculo estándar de RTO del TCP: ambigüedades en la medición del RTT asociadas a la retransmisión de paquetes, la posición conservadora del RTO en lo relativo a la retransmisión de sólo un paquete perdido en un recorrido ida-vuelta, la dificultad para establecer una estimación inicial y la incapacidad de seguir el rápido incremento del RTT durante momentos de congestión. Kan y Partridge [KP87] se ocuparon del primer aspecto, eliminando ambigüedades en la medición del RTT. La introducción de reconocimientos selectivos" (SACKs) [MMFR96] resolvió la cuestión del retardo en la retransmisión de paquetes perdidos. Jacobson [Jac88] mejoró más aún la estimación de RTO del TCP al introducir también una estimación EWMA de la variación del RTT para definir posteriormente la siguiente relación:

$$\text{RTO} = SRTT + k \cdot RTTVAR \qquad (1)$$

donde *SRTT* es una estimación nivelada del RTT (al igual que antes), y *RTTVAR* es una estimación nivelada de la variación del RTT. En [Jac88] $k = 2$, pero esto fue corregido en una versión revisada del trabajo haciendo $k = 4$ [JK92].

Mientras que el uso de este estimador está ampliamente difundido en la actualidad, según nos consta la única evaluación sistemática del mismo respecto a mediciones de conexiones TCP es la de nuestro trabajo previo [Pax97b], el cual demostró que, salvo ante implementaciones muy audaces, este estimador parece suficientemente conservador en el sentido de que sólo raramente genera interrupciones innecesarias.

La implementación BSD del RTO tan difundida [WS95] presenta varias limitaciones: (1) las estimaciones de ajuste del RTT y de variación del RTT se actualizan con nuevas mediciones sólo una vez por cada recorrido ida-vuelta, por lo cual se adaptan a los cambios en las condiciones de la red de manera bastante lenta; (2) las mediciones se realizan usando un reloj con 500 mseg. de granularidad, lo cual induce obviamente estimaciones muy generales (aunque [Jac88] presenta varios métodos ingeniosos para lograr más precisión en ellas); y (3) la estimación del RTO resultante tiene el elevado valor mínimo de un segundo, lo cual puede hacerlo inherentemente conservador.

Estas tres limitaciones pueden eliminarse con la introducción de relojes de mayor precisión y la posibilidad de usar "marcas temporales" propias del TCP [JBB92]. Sin embargo, quedaría pendiente saber cuál es la mejor manera de rediseñar la estimación del RTO dadas estas nuevas características: sabemos que el método de cálculo actual es suficientemente conservador, pero ¿sabemos si no es *demasiado* conservador? Si ese fuese el caso, ¿cómo podemos mejorarlo dada una atenuación de las limitaciones antes mencionadas? Estas son las preguntas que intentamos contestar.

2.1 Algoritmo básico de estimación del RTO

En el algoritmo de Jacobson, las dos variables de estado *SRTT* y *RTTVAR* estiman el RTT presente y una noción de su variación. Estos valores son usados en la Ecuación 1 con un $k = 4$ a fin de obtener el RTO. Dichas variables se actualizan cada vez que se efectúa una medición RTT_{med} del RTT. Puesto que un momento dado solamente se cronometra un segmento y su correspondiente ACK, las actualizaciones ocurren sólo una vez por RTT (a lo que también se hace referencia como una vez por "recorrido"). *SRTT* se actualiza usando un EWMA con una ganancia α_1:

$$SRTT \leftarrow (1 - \alpha_1)SRTT + \alpha_1 \text{RTT}_{med} \qquad (2)$$

y Jacobson [Jac88] recomienda $\alpha_1 = 1/8$, lo cual conduce a una implementación eficiente usando aritmética de punto fijo y desplazamiento de bit. Similarmente, *RTTVAR* se actualiza sobre la base de la desviación $|SRTT - \text{RTT}_{med}|$ usando $\alpha_2 = \_$.

Cada vez que se pierde un paquete retransmitido por expiración del RTO, el emisor de TCP duplica el valor presente del RTO. Al hacer

2. For each data packet retransmitted in the TCP trace due to a timeout, we assess whether the timeout was *unavoidable*, meaning that either the segment being retransmitted was lost, or all ACKs sent after the segment's arrival at the receiver (up until the arrival of the retransmission) were lost. This check is necessary because some of the TCPs in the $\mathcal{N}_2$ dataset used aggressive RTO estimators that often fired prematurely in the face of high RTTs [Pax97a], so these retransmissions are not treated as normal timeout events.

3. If the timeout was unavoidable, then the retransmission is classified as a "first" timeout if this is the first time the segment is retransmitted, or as a "repeated" timeout otherwise. The estimator is charged the current RTO setting as reflecting the amount of time that passed prior to retransmitting (consideration (1) above), with separate bookkeeping for "first" and "repeated" timeouts (for reasons explained below). The RTO timer is also backed off by doubling it.

4. If the timeout was avoidable, then it reflects a problem with the actual TCP in the trace, and this deficiency is not charged against the estimator we are evaluating.

5. For each arrival of an ACK for new data in the trace, the ACK arrival time is compared with the RTO, as computed by the given estimator. If the ACK arrived after the RTO would have fired we consider the expiration a "bad" timeout, reflecting that the feedback time of the network path at that moment exceeded the RTO.

   If the ACK covers all outstanding data the RTO timer is turned off.

   If the ACK also yielded an RTT measurement (because it acknowledged the segment currently being timed, or because every segment is being timed), *SRTT* and *RTTVAR* are updated based on the measurement and the RTO is recomputed.

   Finally, the RTO timer is restarted.

6. The sending or receiving of TCP SYN or FIN packets is not assessed, as these packets have their own retransmission timers, and if interpreted as simple ACK packets can lead to erroneous measurements of RTT.

Note this approach contains a subtle but significant difficulty. Suppose that in the trace packet P is lost and 3 seconds later the TCP's real-life RTO expires and P is retransmitted. We treat this as a "first timeout," and charge the estimator with the RTO, R, it computed for P. Suppose $R = 100$ msec. From examining the trace it is impossible to determine whether retransmitting P after waiting only 100 msec would have been successful. It could be that waiting any amount of time less than 3 seconds was in fact too short an interval for the congestion leading to P's original loss to have drained from the network. Conversely, suppose P is lost after being retransmitted 3 seconds later. It could be that the first loss and the second are in fact uncorrelated, in which case retransmitting after waiting only R seconds would yield a successful transmission.

The only way to assess this effect would be to conduct live experiments, rather than trace-driven simulation, which we leave for future work. Therefore, we assess *not* whether a given retransmission was *effective*, meaning that the retransmitted packet safely arrived at the receiver, but only whether the *decision* to retransmit was *correct*, meaning that the packet was indeed lost, or all feedback from the receiver was lost. Related to this consideration, only the effectiveness of an RTO estimator at predicting timely "first" timeouts is assessed. For repeated timeouts it is difficult to gauge exactly how many of the potential repeated retransmissions would have been necessary.

Given these considerations, for a given estimator and a trace i let T_i be the total time required by the estimator to wait for unavoidable first timeouts. Let g_i be the number of "good" (necessary) first timeouts, and b_i the total number of "bad" timeouts, including multiple bad timeouts due to backing off the timer (since we can soundly assess that all of these repeated retransmissions were indeed unnecessary). If $b_i + g_i > 0$, that is, trace i included some sort of timeout, then define $\rho_i = \frac{b_i}{b_i+g_i}$, the normalized number of bad timeouts in the trace; otherwise define $\rho_i = 0$. Note that ρ_i may not be a particularly good metric when considering transfers of varying length. However, this study focuses only on transfers of 100 KB.

For the jth good timeout, let RTO_i^j be the RTO setting of the expiring timer, and RTT_i^j be the most recently observed RTT (even if it was not an RTT that would have been measured for purposes of updating the *SRTT* and *RTTVAR* state variables). Let $\xi_i^j = \text{RTO}_i^j/\text{RTT}_i^j$, so ξ_i^j reflects the cost of the timeout in units of RTTs. We can then define an average, normalized timeout cost of $\psi_i = E_j[\xi_i^j]$, or 0 if trace i does not include any good timeouts.

For a collection of traces, we then define $W = \sum_i T_i$ as the total time spent waiting for (good) first timeouts; $\widetilde{W} = E_{i:g_i>0}[\psi_i]$ as the mean normalized timeout cost per connection that experienced at least one good timeout; and $B = E_i[\rho_i]$ as the mean proportion of timeouts that are *bad*, per connection, including connections that did not include any timeouts (because we want to reward estimators that, for a particular trace, don't generate any bad timeouts).

W can be dominated by a few traces with a large number of timeout retransmissions, for which the total time waiting for first timeouts can become very high, so it is biased towards highlighting how bad things can get. $\widetilde{W}$ is impartial to the number of timeouts in a trace, and so better reflects the overall performance of an estimator. B likewise better reflects how well an estimator avoids bad timeouts overall. For some estimators, there may be a few particular traces on which they retransmit unnecessarily a large number of times, as noted below.

Finally, of the 18,490 pairs of traces in $\mathcal{N}_2$, 4,057 pairs were eliminated from our analysis due to packet filter errors in recording the traces, the inability to pair packets across the two traces (this can occur due to packet filter drops or IP ID fields changed in flight by header compression glitches [Pax97c]), or *tcpanaly*'s inability to determine which retransmissions were due to timeouts. This leaves us with 14,433 traces to analyze, with a total of 67,073 timeout retransmissions. Of those, 53,110 are "first" timeouts, and 34% of the traces have no timeout retransmissions.

Minimum RTO	W	$\widetilde{W}$	B
1,000 msec	144,564	8.4	0.63%
750 msec	121,566	6.5	0.76%
500 msec	102,264	4.8	1.02%
250 msec	92,866	3.5	2.27%
0 msec	92,077	3.1	4.71%
RTO = 2,000 msec	229,564	15.6	2.66%
RTO = 1,000 msec	136,514	8.2	6.14%
RTO = 500 msec	85,878	4.5	12.17%

Table 1: Effect of varying RTO_{min}, $G = 1$ msec

2.3 Varying the Minimum RTO

It turns out that the setting of RTO_{min}, the lower bound on RTO, can have a major effect on how well the RTO estimator performs, so we begin by analyzing this effect. We first note that the usual setting for RTO_{min} is two clock "ticks" (i.e., $\text{RTO}_{\text{min}} = 2G$), because, given a "heartbeat" timer, a single tick translates into a time anywhere between 0 and G sec. Accordingly, for the usual coarse-grained estimator of $G = 0.5$ sec, RTO_{min} is 1 sec, which we will see is conservative (since a real BSD implementation would use a timeout between 0.5 sec and 1 sec). But for $G = 1$ msec, the two-

esto, se disminuye la velocidad de transmisión ante la presencia de congestión sostenida y se minimizan los posibles efectos adversos de una subestimación del RTO y las consecuentes retransmisiones innecesarias y repetidas.

SRTT y *RTTVAR* se inicializan con la primera medición RTT_{med} usando $SRTT \leftarrow RTT_{med}$ y $RTTVAR \leftarrow \_ RTT_{med}$. Antes de la primera medición, RTO = 3 seg.

Dos consideraciones adicionales importantes son que toda medición se efectúa usando una *granularidad* de reloj de *G* segundos (el reloj avanza en incrementos de *G*[1]) y que el RTO está *limitado* por el RTO_{min} y el RTO_{max}. En la implementación común BSD del TCP, *G* = 0,5 seg. , $RTO_{min} = 2G = 1$ seg. y $RTO_{max} = 64$ seg. Como se verá mas adelante, el valor de RTO_{min} es bastante significativo. Asimismo, dado que la granularidad es muy general, el código para actualizar *RTTVAR* establece un límite inferior de *G* para el *RTTVAR,* en lugar del valor de 0 segundos que puede surgir a menudo de forma natural.

Tres variantes propuestas a menudo para implementar el estimador del RTO son: cronometrar el RTT de cada segmento en vez de sólo uno por recorrido, usar valores de *G* inferiores, y disminuir el RTO_{min} a fin de consumir menos tiempo en espera de expiraciones. La RFC 1323 [JBB92] apoya explícitamente las dos primeras, siendo la motivación original de esta parte de nuestro estudio la de evaluar posibles beneficios con la concreción de estos cambios.

2.2 Valorando diferentes estimadores de RTO

Investigamos sobre dos propiedades fundamentales de un estimador de RTO: (1) ¿cuánto tiempo espera antes de retransmitir un paquete perdido?, y (2) ¿con qué frecuencia expira por error y genera innecesariamente una retransmisión? Un estimador de RTO muy conservador podría simplemente puentear un RTO = 60 segundos, sin cometer un error nunca, cumpliendo con la segunda propiedad, pero comportándose muy ineficazmente con respecto a la primera, provocando así demoras inaceptables. Por otra parte, un estimador muy agresivo podría puentear un RTO = 1 segundo, invirtiendo así esta relación e inundando la red con retransmisiones innecesarias.

Nuestro planteamiento básico para examinar estas dos propiedades es usar simulaciones basadas en trazas con objeto de evaluar diferentes estimadores, y aplicar la siguiente metodología, la cual sigue la implementación de estimadores de RTO descrita en [WS95]:

1. Para cada paquete de datos enviado, se inicia el temporizador de RTO si éste aún no está activo. El temporizador también se reinicia cuando el paquete de datos es el principio de una secuencia de retransmisión.

2. Para cada paquete de datos retransmitido en la traza de TCP debido a una expiración, determinamos si ésta era *inevitable*, lo cual significa que bien el segmento retransmitido se había perdido, o bien que todos los ACK enviados después de la llegada del segmento al receptor (y hasta la llegada de la retransmisión) se habían perdido. Esta verificación es necesaria debido a que algunos de los TCP en el conjunto de datos N_2 usaban estimadores de RTO agresivos que a menudo se generaban prematuramente ante altos valores de RTT [Pax97a], de manera tal que estas retransmisiones son consideradas como eventos de expiración normales.

3. Si la expiración fue inevitable, entonces la retransmisión se clasifica como “primera” expiración si es la primera vez que el segmento es retransmitido, o como expiración “repetida” en el caso opuesto. Al estimador se le aplica el valor corriente del RTO para reflejar la cantidad de tiempo transcurrido antes de la retransmisión (consideración (1) anterior), llevando registros separados para interrupciones “primarias” y ”repetidas” debido a razones aducidas posteriormente. El temporizador de RTO es asimismo retrasado mediante la duplicación de su valor.

4. Si la expiración era evitable, ello supone un problema con el TCP real de la traza, no considerándose esta deficiencia en contra del estimador que estamos evaluando.

5. Para cada llegada de un ACK debido a nuevos datos en la traza, el tiempo de llegada del ACK es comparado con el RTO, según los cálculos del estimador en cuestión. Si el ACK llegara una vez que el RTO se hubiese generado, consideramos el evento como una expiración o interrupción “errónea”, mostrando que el tiempo de respuesta de la ruta de red en eso momento excedía al RTO.

   Si el ACK responde a todos los datos de emisión, entonces el temporizador del RTO se desconecta.

   Si el ACK supuso también una medición del RTT (por que reconoció el segmento cronometrando en ese instante, o por que todos los segmentos están siendo cronometrados), se actualizan las variables *SRTT* y *RTTVAR* en función de la medición, y se recalcula el RTO.

   Finalmente se reinicia el temporizador de RTO.

6. No se evalúa la emisión y recepción de los paquetes SYN y FIN del TCP, puesto que estos paquetes tienen sus propios temporizadores de retransmisión, y si se interpretan como simples paquetes ACK pueden inducir a mediciones equivocadas del RTT.

Obsérvese que este esquema contiene una dificultad sutil al mismo tiempo que significativa. Supóngase que se pierde el paquete marcado *P, y* que 3 segundos después expira el RTO real retransmitiéndose entonces el paquete *P*. Consideramos que ello es una interrupción “primaria”, y cargamos al estimador con el valor *R* del RTO calculado para *P*. Asúmase que *R* = 100 mseg. Al examinar la traza resulta imposible determinar si la retransmisión de *P* , tras esperar solamente 100 mseg., hubiera tenido éxito. Podría ocurrir que el hecho de esperar un lapso de tiempo inferior a 3 seg. resultase de hecho un intervalo demasiado corto para que desapareciera de la red la congestión que produjo la pérdida original de *P*. Contrariamente, considérese que *P* se pierde después de haber sido retransmitido 3 seg. más tarde. Podría ocurrir ahora que la primera y la segunda pérdida no estuviesen relacionadas, en cuyo caso la retransmisión al cabo de sólo *R* seg. podría haber redundado en una transmisión satisfactoria.

La única manera de medir este efecto sería mediante experimentos reales, en vez de simulaciones basadas en trazas, algo que dejamos de lado para futuros trabajos. Por lo tanto, *no medimos* si una retransmisión dada resultó *efectiva*, implicando que el paquete retransmitido llegó a salvo al receptor, sino solamente si la *decisión* de retransmitir fue *correcta*, indicando con ello que el paquete en realidad se había extraviado, o que se perdió toda respuesta del receptor. En relación con esta consideración, solo se evalúa la eficacia de un estimador de RTO para detectar oportunamente interrupciones “primarias”. Para interrupciones repetidas, resulta difícil medir exactamente cuántas de ellas eran realmente necesarias.

Dadas estas consideraciones, para un cierto estimador y traza *i,* sea T_i el tiempo total requerido por el estimado para detectar interrupciones primarias inevitables, y g_i el número de interrupciones

1 La implementación de temporizador BSD también utiliza un cronómetro “marcapasos” que expira cada *G* segundos con una fase independiente del momento en el cual el temporizador es efectivamente ajustado. Hemos incluido este comportamiento en nuestras simulaciones.

Granularity	W	$\widetilde{W}$	B
500 msec	272,885	19.2	0.36%
[WS95] (500 msec)	245,668	15.4	0.23%
250 msec	167,360	10.2	0.67%
100 msec	142,940	8.4	0.95%
50 msec	143,156	8.4	0.84%
20 msec	143,832	8.4	0.70%
10 msec	144,175	8.4	0.67%
1 msec	144,564	8.4	0.63%

Table 2: Effect of varying granularity G, $\text{RTO}_{\text{min}} = 1$ sec

tick minimum is only 2 msec, and so setting RTO_{min} to larger values can have a major effect.

Table 1 shows W, $\widetilde{W}$ and B for different values of RTO_{min}, for $G = 1$ msec. We see that W runs from 144,564 seconds for a minimum of 1 sec to about 64% as much when using no minimum. The column for $\widetilde{W}$ shows that the 1 sec minimum means that a typical RTO costs a bit more than 8 RTTs, but much of this expense disappears as we decrease the minimum. B, on the other hand, shows that for a 1 sec minimum, on average only about 1 in 150 timeouts is bad, while for no minimum, nearly 1 in 20 is (these bad timeouts are not clustered among a particular small subset of the traces). Clearly, adjusting the minimum RTO provides a "knob" for directly trading off timely response with premature timeouts, with no obvious "sweet spot" yielding an optimal balance between the two.

As noted above, "delayed" acknowledgments in TCP can result in elevating RTTs by up to 500 msec, and in a number of common implementations, frequently elevate RTTs by up to 200 msec. Accordingly, it is not clear that a minimum RTO of two ticks for $G = 1$ msec is sound. However, for the bulk of our subsequent analysis, we consider estimators with no minimum bound, both to highlight the contribution to estimator efficiency of factors other than the quite-dominant minimum RTO, and to keep in mind that transport protocols different from TCP might not introduce such a minimum.

For comparison, we include three static timers that use a constant setting for RTO (except they double the RTO on repeated timeouts). The table highlights the heavy cost of not using an adaptive timer. The constant estimators generate about 10 times as many bad timeouts as the adaptive estimators with similar relative performance figures ($\widetilde{W}$). The values of B don't tell the whole story for the static timers, however, because their bad timeouts are clustered among relatively few traces. For example, RTO = 2,000 msec results in a bad timeout in 538 traces, while for $\text{RTO}_{\text{min}} = 250$ msec, which has a similar value of B, spreads its bad timeouts over more than twice as many traces.

2.4 Varying Measurement Granularity

With the above caution regarding the considerable importance of RTO_{min} in mind, we now look at the effect of varying G. In Table 2, G ranges from 500 msec down to 1 msec. In order to compare the different granularities on an even footing, we hold $\text{RTO}_{\text{min}} = 1$ sec constant, rather than having the relative differences between the granularities overwhelmed by using $\text{RTO}_{\text{min}} = 2G$. We include one additional row, "[WS95]," which is the estimator as implemented in [WS95]. This implementation includes fixed-point arithmetic and bit-shifting in order to estimate *SRTT* at an effective granularity of 62.5 msec and *RTTVAR* at a granularity of 125 msec, though RTO itself is computed with a granularity of 500 msec.

We first note that for $G \leq 100$ msec, the performance for good timeouts, both absolute (W) and relative ($\widetilde{W}$) is essentially identical, regardless of how fine the granularity becomes. But we steadily gain in avoiding bad timeouts (minimizing B) as the granularity becomes finer. The reason for the gain is that the more coarse granularities will often take no action in the face of a minor change in RTT, while the finer granularity estimator will adapt to reflect the change, and this gives it a slight edge.

Parameters	W	$\widetilde{W}$	B
[WS95]	245,668	15.4	0.23%
[WS95]-*every*	241,100	14.7	0.25%
take-first ($\alpha_1, \alpha_2 = 0, \text{RTO}_{\text{min}} = 1\ s$)	158,199	8.5	0.74%
take-first ($\alpha_1, \alpha_2 = 0$)	131,180	4.4	2.93%
very-slow ($\alpha_1 = \frac{1}{80}, \alpha_2 = \frac{1}{40}$)	113,903	3.9	3.97%
slow-every ($\alpha_1 = \frac{1}{32}, \alpha_2 = \frac{1}{16}$)	102,544	3.4	4.28%
slow ($\alpha_1 = \frac{1}{16}, \alpha_2 = \frac{1}{8}$)	96,740	3.4	3.84%
std ($\alpha_1 = \frac{1}{8}, \alpha_2 = \frac{1}{4}$)	92,077	3.1	4.71%
std-every ($\alpha_1 = \frac{1}{8}, \alpha_2 = \frac{1}{4}$)	94,081	3.1	5.09%
fast ($\alpha_1 = \frac{1}{2}, \alpha_2 = \frac{1}{4}$)	90,212	3.0	7.27%
take-last ($\alpha_1, \alpha_2 = 1$)	93,490	3.3	19.57%
take-last-every ($\alpha_1, \alpha_2 = 1$)	97,098	3.5	20.20%
take-last ($\alpha_1, \alpha_2 = 1, \text{RTO}_{\text{min}} = 1\ s$)	145,571	8.5	1.30%

Table 3: Effect of varying EWMA parameters α_1, α_2

Above $G = 100$ msec, however, we start trading off reduced performance for avoiding bad timeouts. We can cut the average rate of bad timeouts by nearly a factor of two by using $G = 500$ msec, but at a cost of more than a factor of two in performance. We also note that the [WS95] estimator clearly performs better than $G = 500$ msec, with both $\widetilde{W}$ and B lower. It gains by performing better on some very-large-RTT traces, because it is able to better reflect relatively small RTT changes due to its finer effective granularities for *SRTT* and *RTTVAR*.

2.5 Varying the EWMA Parameters

Table 3 shows the estimator's performance when varying α_1 (per Eqn 2) and α_2, holding $G = 1$ msec and $\text{RTO}_{\text{min}} = 0$ msec fixed, except where noted. The first two rows are the [WS95] implementation, which uses $G = 500$ msec, with the second row reflecting a variant that derives an RTT measurement from every ACK arriving at the sender. We see that the more frequent *SRTT* and *RTTVAR* updates have little effect on the estimator's performance, only making it slightly more aggressive.

The remaining estimators all use $G = 1$ msec. The *take-first* extreme of $\alpha_1 = \alpha_2 = 0$ simply uses the first RTT measurement to initialize both *SRTT* $\leftarrow$ RTT and *RTTVAR* $\leftarrow \frac{1}{2}$RTT, yielding RTO $\leftarrow$ 3RTT. It never changes *SRTT*, *RTTVAR*, or RTO again (other than to back off RTO in the face of repeated retransmissions, and undo the backing off when the retransmission epoch ends). The first variant of it reflects using $\text{RTO}_{\text{min}} = 1$ sec, the second, $\text{RTO}_{\text{min}} = 0$ sec. At the other extreme, we have *take-last*, which always sets *SRTT* $\leftarrow$ RTT and *RTTVAR* $\leftarrow |\textit{SRTT}_{\text{prev}} - \text{RTT}|$. The *take-last-every* variant is the same except every packet is timed rather than just one packet per round trip, and the final variant raises the minimum RTT to 1 sec.

In between these extremes we run the gamut from *very-slow*, which uses one-tenth the usual parameters (which are given for the *std* estimator), to *fast*, which uses twice the parameters, with some time-every-packet variants.

From the table we see that the settings of the EWMA parameters make little difference in how well the estimator performs. Indeed, if our goal is to minimize the rate of bad timeouts and still remain aggressive, we might pick the exceedingly simple *take-first* estimator, which only barely adapts to the network path conditions;[2] or we

[2]Even though *take-first* and *take-last* show overall decent performance compared to the other RTO estimators, these RTO estimators could perform extremely poorly over network paths that exhibit large, sudden changes in RTT.

primarias "acertadas" (necesarias), y b_i el número total de interrupciones "erróneas", incluyendo interrupciones múltiples erróneas debidas al retraso del temporizador (dado que podemos deducir justificadamente que todas estas retransmisiones repetidas eran ciertamente innecesarias). Si $g_i + b_i > 0$, o sea, si la traza i tenía algún tipo de interrupción, entonces el número normalizado de interrupciones erróneas en el trazador resultará ser $\rho_i = b_i / b_i + g_i$. En caso contrario, definimos $\rho_i = 0$. Obsérvese que ρ_i no puede resultar una medida particularmente buena cuando se consideran transferencias de longitud variable. No obstante, este estudio se concentra únicamente en transferencias de 100 KB.

Para la j^{ava} interrupción acertada, sea RTO_i el ajuste del RTO para el tiempo de espera i, y RTT_i el RTT de medición más reciente (aun cuando no hubiera sido un RTT medido con el propósito de actualizar las variables de estado *SRTT* y *RTTVAR*). Defínase ahora $\xi_i = RTO_i / RTT_i$, de modo tal que ξ_i refleje el coste de la interrupción en términos de RTT. Podemos entonces establecer un coste de interrupciones promedio normalizado $\psi_i = E_i[\xi_i]$, igual a 0 cuando el trazador i no contenga ninguna interrupción correcta.

Para una colección de trazas definimos: $W = \Sigma_i T_i$ como el tiempo total consumido en espera de interrupciones primarias (correctas), $\sim W = E_{i:gi>0}[\psi_i]$ como el coste de interrupción promedio normalizado por conexión que experimentó al menos una interrupción correcta, y $B = E_i[\rho_i]$ como la proporción media de interrupciones *erróneas* por conexión, incluyendo conexiones que no sufrieron ninguna interrupción (dado que nos interesa recompensar los estimadores que, para una traza en particular, no generan ninguna interrupción errónea).

W puede resultar dominado por unas pocas trazas con gran cantidad de transmisiones por interrupción, para las cuales el tiempo total de espera para interrupciones primarias puede llegar a ser muy alto, por lo cual está orientado a destacar un estado de mal funcionamiento. $\sim W$ es independiente de la cantidad de interrupciones en una traza, y por tanto refleja mejor la eficacia general de un estimador. De igual modo, B refleja mejor la manera en que un estimador evita interrupciones erróneas de modo general. Como se observará con posterioridad, para ciertos estimadores pueden existir algunas trazas particulares sobre las que se realizan una gran cantidad de retransmisiones innecesarias.

Finalmente, sobre los 18,490 pares de trazas en $\mathbb{N}_2$, 4,057 fueron eliminados de nuestro análisis debido a errores de filtrado de paquetes en el registro de las trazas, la incapacidad para emparejar paquetes a lo largo de ambas trazas (ello puede acaecer por caídas en el filtrado de paquetes, o por cambios en los campos ID del IP durante el recorrido debido a errores de compresión [Pax97c]), o la incapacidad del *tcpanaly* para determinar qué retransmisiones se debían a interrupciones. Esto nos deja con 14,433 trazas para analizar, con un total de 67,073 retransmisiones debidas a interrupciones. De éstas, 53,110 corresponden a interrupciones "primarias", con un 34% las trazas que no sufren retransmisiones por interrupción.

2.3 Variando el RTO mínimo

Ocurre que la configuración del RTO_{min}, el límite inferior del RTO, puede tener un efecto importante sobre el buen funcionamiento del estimador de RTO. Comenzamos, por tanto, analizando dicho efecto. En primer lugar, observamos que la configuración usual del RTO_{min} es de 2 pulsos de reloj ($RTO_{min} = 2G$) pues, dado un cierto temporizador de "marcapaso", un pulso se traduce como un valor de tiempo cualquiera entre 0 y G seg. De acuerdo a ello, para el estimador usual de gran granularidad con G = 0,5 seg., el RTO_{min} es 1 seg., valor que según veremos resulta conservador (dado que una implementación real de BSD produciría una interrupción de 0,5 a 1 seg.). Pero para G = 1 mseg., el mínimo de 2 pulsos equivale a 2 mseg. y por tanto, establecer el RTO_{min} a valores mayores puede resultar muy significativo.

La Tabla 1 muestra W, $\sim W$ y B para diversos valores de RTO_{min} y G = 1 mseg. Observamos que G varía desde 144,564 segundos para un mínimo de 1 seg. hasta casi un 64% de este valor cuando no se establece mínimo. La columna de $\sim W$ muestra que para el mínimo de 1 seg., un RTO típico cuesta un poco más que 8 RTT, pero gran parte de dicho gasto desaparece a medida que disminuimos el mínimo. Por otra parte, B muestra que para un mínimo de 1 seg., en promedio sólo uno de cada 150 interrupciones es errónea, mientras que para mínimo nulo, casi una de cada 20 lo es (estas interrupciones erróneas no están agrupadas en ningún subconjunto particular de trazas). Claramente, ajustar el mínimo RTO ofrece un punto de apoyo para lograr un compromiso directo entre respuestas puntuales e interrupciones prematuras, sin un punto de equilibrio obvio que determine el balance óptimo entre las dos.

RTO Mínimo	*W*	*~W*	*B*
1.000 mseg.	144,564	8,4	0,63%
750 mseg.	121,566	6,5	0,76%
500 mseg.	102,264	4,8	1,02%
250 mseg.	92,866	3,5	2,27%
0 mseg.	92,077	3,1	4,71%
RTO = 2,000 mseg.	229,564	15,6	2,66%
RTO = 1,000 mseg.	136,514	8,2	6,14%
RTO = 500 mseg.	86,878	4,5	12,17%

Tabla 1: Efecto de variar el RTO_{min}, G = 1 mseg.

Como ya se ha mencionado anteriormente, los reconocimientos "retardados" en el TCP pueden elevar los RRT hasta en 500 mseg. En varias implementaciones comunes, aumentan los RTT con frecuencia hasta los 200 mseg. De acuerdo a ello, no resulta claro si un RTO mínimo de 2 pulsos para G = 1 resulta apropiado. Sin embargo, para el grueso de nuestros análisis subsiguientes, consideramos estimadores sin límite inferior, tanto para resaltar la contribución a la eficacia del estimador de factores distintos al tan dominante RTO mínimo, como para tener en cuenta que los protocolos de transporte diferentes del TCP podrían no originar ese mínimo.

A título comparativo, incluimos tres temporizadores estáticos que usan una configuración fija del RTO (salvo por la duplicación del RTO en interrupciones repetidas). La tabla destaca el alto coste de no usar un temporizador adaptativo. Los estimadores constantes generan casi 10 veces más interrupciones erróneas que los estimadores adaptativos con valores similares ($\sim W$) de eficacia relativa. No obstante, los valores de B no cuentan todo sobre los estimadores constantes, dado que sus interrupciones erróneas están concentradas en una cantidad de trazas relativamente pequeña. Por ejemplo, con RTO = 2.000 mseg. se producen interrupciones erróneas en 538 trazas, mientras que con RTO_{min} = 250 mseg, que tiene un valor de B similar, las interrupciones erróneas se distribuyen en más del doble de las trazas.

2.4 Variando la granularidad de la medida

Con la salvedad mencionada anteriormente respecto a la relativa importancia del RTO_{min}, nos centramos a continuación en el efecto de variar G. En la Tabla 2, G varía desde 500 mseg. hasta 1 mseg. A efectos de comparar las diferentes granularidades sobre una misma base, mantenemos constante el RTO_{min} = 1 seg., mejor que usar un $RTO_{min} = 2G$ y sobrecargar con ello las diferencias relativas entre las granularidades. Incluimos una fila adicional "[WS95]", que corresponde a la implementación del estimador en [WS95]. Esta implementación incluye aritmética de punto fijo y desplazamiento de bit a fin de estimar SRTT con

***RTTVAR* factor**	W	$\widetilde{W}$	B
$k = 16$	168,002	7.0	0.59%
$k = 12$	144,053	5.7	0.81%
$k = 8$	118,858	4.4	1.52%
$k = 6$	105,681	3.8	2.43%
adapt	94,220	3.2	4.44%
$k = 4$	92,077	3.1	4.71%
$k = 3$	85,264	2.8	7.68%
$k = 2$	78,565	2.5	13.64%
$\text{RTO}_{\text{min}} = 750$ msec, $k = 6$	128,266	6.7	0.50%
$\text{RTO}_{\text{min}} = 750$ msec	121,566	6.5	0.76%
take-first$_{250\text{msec}}$, $k = 6$	163,799	6.4	0.70%
$\text{RTO}_{\text{min}} = 500$ msec, $k = 6$	112,514	5.1	0.69%
$\text{RTO}_{\text{min}} = 500$ msec	102,264	4.8	1.02%
$\text{RTO}_{\text{min}} = 250$ msec, $k = 6$	106,139	4.0	1.29%
$\text{RTO}_{\text{min}} = 250$ msec	92,866	3.5	2.27%

Table 4: Effect of varying *RTTVAR* factor, k

might pick *slow*, which on average incurs 25% less normalized delay per timeout, and occupies a sweet spot that locally minimizes B. As we found for [WS95], timing every packet makes little difference over timing only one packet per RTT, even though by timing every packet we run many more measurements through the EWMAs per unit time. This in turn causes the EWMAs to adapt *SRTT* and *RTTVAR* more quickly to current network conditions, and to more rapidly lose memory of conditions further in the past, similar in effect to using larger values for α_1 and α_2.

We note that as the timer more quickly adapts, B steadily increases, with *take-last-every* generating on average one bad timeout in every five, indicating correlations in RTT variations that span multiple round-trips. We can greatly diminish this problem by raising RTT_{min} to 1 sec, but only by losing a great deal of the estimator's timely response, and we are better off instead using the corresponding *take-first* variant.

We also evaluated varying the EWMA parameters for $\text{RTO}_{\text{min}} = 500$ msec. We find that $\widetilde{W}$ increases by roughly 50%, with the variation among the estimators further diminishing, while B falls by a factor of 4–8, further illustrating the dominant effect of the RTO minimum.

Finally, a number of the paths in $\mathcal{N}_2$ contain slow, well-buffered links, which lead to steady, large increases in the RTT (up to many seconds). We might expect *take-first* to do quite poorly for these connections, since the first measured RTT has little to do with subsequent RTTs, but in fact *take-first* does quite well. The key is the last part of step 5 in § 2.2 above: the RTO timer is restarted with each arriving ACK for new data. Consequently, when data is flowing, the RTO has an implicit extra RTT term [Lud99], and for *take-first* this suffices to avoid bad timeouts even for RTTs that grow by two orders of magnitude. Indeed, *take-first* does *better* for such connections than estimators that track the changing RTT! It does so because more adaptive estimators wind up waiting much longer after the last arriving ACK before RTO expires, while *take-first* retransmits with appropriate briskness in this case. But this advantage is particular to the highly-regularized feedback of such connections. It does, however, suggest the notion of a "feedback timeout," discussed briefly in § 4.

2.6 Varying the *RTTVAR* Factor

The last RTO estimation parameter we consider is k, the multiplier of *RTTVAR* when computing RTO, per Eqn 1. For the standard implementation, $k = 4$. Table 4 shows the effects of varying k from 2–16, for $G = 1$ msec and $\text{RTO}_{\text{min}} = 0$ sec. The *adapt* estimator starts with $k = 4$ but doubles it every time it incurs a bad timeout.

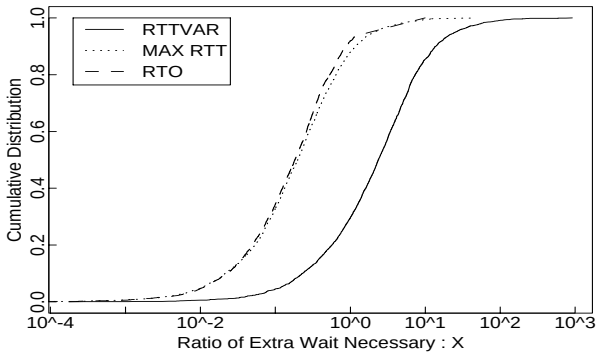

Figure 1: Extra waiting time necessary to avoid bad RTO

k clearly provides a knob for trading off waiting time for unnecessary timeouts, with no obvious sweet spot. This balance changes a bit, however, when we increase RTO_{min}, as shown in the second half of the table. For example, we find that $\text{RTO}_{\text{min}} = 250$ msec, $k = 4$ performs strictly better than the no-minimum $k = 6$ variant, and $\text{RTO}_{\text{min}} = 250$ msec, $k = 6$ performs better than the $k = 8$ variant. Even the extremely simple *take-first* estimator, if using $k = 6$ and $\text{RTO}_{\text{min}} = 250$ msec, performs a bit better than the regular $\text{RTO}_{\text{min}} = 750$ msec estimator.

2.7 Can We Estimate RTO Better?

Having evaluated the effects of different estimator parameters and, for the most part, only found tradeoffs and little in the way of compelling "sweet spots," we now turn to the question of whether there are indeed opportunities to devise still better estimators. A key consideration for answering this question is: when we underestimate, by how much is it? If, for example, underestimates tend to be off by less than RTT, then that would suggest a modification to Eqn 1 in which *SRTT* has a factor of 2 applied to it.

Let A denote the amount of additional waiting time needed to avoid a bad RTO. Figure 1 plots the cumulative distribution of the ratio of A to *RTTVAR* (solid), the maximum RTT seen so far (dotted), and RTO (dashed), for the usual $G = 1$ msec estimator. The ratio of A to *RTTVAR* ranges across several orders of magnitude, indicating that finding a particular value of k in Eqn 1 that efficiently takes care of most of the remaining bad timeouts is unlikely.

Also shown is that A is generally less than the current RTO and also the maximum RTT seen so far; this suggests adding one of those values to RTO to make it sufficiently conservative to avoid bad timeouts. However, doing so has much the same effect as other estimator variants that wait longer based on other factors (e.g., the value of k). For example, changing the standard $k = 4$ estimator shown in Table 4 to use twice the computed RTO (i.e., add in an additional RTO term) lowers B from 4.71% to 0.57%, but increases $\widetilde{W}$ from 3.1 to 5.7—a bit better than just using $k = 12$, but not compellingly better.

For $\text{RTO}_{\text{min}} = 0.5$ sec, the plot is very similar, with slightly more separation between the RTO and MAX RTT lines. Thus, Figure 1 suggests a fundamental tradeoff between aggressiveness and suffering bad timeouts.

A related question is: if a packet is unnecessarily retransmitted, does it reflect a momentary increase in RTT, or a sustained increase? We find that about 62% of the bad timeouts were followed by RTTs less than the current RTO, so the bad timeout reflected a transient RTT increase. Another 24% were followed by exactly one more elevated RTT, though a bit more than 2% were followed by 10 or more elevated RTTs. Thus, most of the time a significant RTT increase is quite transient—but there is non-negligible tail-weight for sustained RTT increases.

una granularidad efectiva de 62,5 mseg. y RTTVAR con 125 mseg., a pesar de que el RTO se calculó con una granularidad de 500 mseg.

En primer lugar, observamos que para $G \leq 100$ mseg. la eficacia de las interrupciones correctas, tanto absoluta (*W*) como relativa (*~W*), es esencialmente idéntica, independientemente de lo fina que llegue a ser la granularidad. Sin embargo, ganamos en evitar interrupciones erróneas de manera constante (minimización de *B*) a medida que se afina la granularidad. La razón de ello radica en que las granularidades más gruesas a menudo no reaccionarán ante un cambio menor del RTT, mientras que el estimador de granularidad más fina se adaptará para reflejar dicho cambio, lo cual le proporciona cierta ventaja.

Granularidad	*W*	*(~W)*	*B*
500 mseg.	272,885	19,2	0,36%
[WS95] 500 mseg.	245,668	15,4	0,23%
250 mseg.	167,360	10,2	0,67%
100 mseg.	142,940	8,4	0,95%
50 mseg.	143,156	8,4	0,84%
20 mseg.	143,832	8,4	0,70%
10 mseg.	144,175	8,4	0,67%
1 mseg.	144,564	8,4	0,63%

Tabla 2: Efecto de variar la granularidad *G*, RTO_{min} = 1 seg.

Sin embargo, por encima de $G = 100$ mseg., comenzamos a perder eficacia a cambio de evitar interrupciones erróneas. Podemos disminuir la frecuencia promedio de interrupciones erróneas en casi un factor de 2 usando $G = 500$ mseg., pero a costa de un factor de pérdida de eficacia superior a 2. También observamos que el estimador [WS95] se comporta claramente mejor que el de $G = 500$ mseg., tanto con un *W* como con un *~W* inferiores. Y resulta superior dado que se comporta mejor en el caso de ciertas trazas de RTT muy grande, pues es capaz de reflejar mejor los cambios de RTT relativamente pequeños debido a su granularidad efectiva más fina en el cálculo de SRTT y RTTVAR.

2.5 Variando los parámetros de EWMA

La Tabla 3 muestra el comportamiento del estimador al variar α_1 (ver Ecuación 2) y α_2, manteniendo constantes $G = 1$ mseg. y $RTO_{min} = 0$ mseg., excepto donde se indica lo contrario. Las primeras dos filas corresponden a la implementación [WS95], que usa $G = 500$ mseg., con la segunda fila mostrando una variación del mismo que realiza una medición de RTT por cada ACK que llega al emisor. Vemos que la frecuencia de actualización más alta de *SRTT* y *RTTVAR* tiene poco efecto sobre la eficacia del estimador, y que sólo lo convierte un poco más agresivo.

Parámetros	*W*	*pW*	*B*
[WS95]	245,668	15,4	0,23%
[WS95] medición del RTT por cada ACK	241,100	14,7	0,25%
Take-first (α_1, $\alpha_2 = 0$, $RTO_{min} = 1$ s)	158,199	8,5	0,74%
Take-first (α_1, $\alpha_2 = 0$)	131,180	4,4	2,93%
Very-slow ($\alpha_1 = 1/80$, $\alpha_2 = 1/40$)	113,903	3,9	3,97%
Slow-every ($\alpha_1 = 1/32$, $\alpha_2 = 1/16$)	102,544	3,4	4,28%
Slow ($\alpha_1 = 1/16$, $\alpha_2 = 1/8$)	96,740	3,4	3,84%
Std ($\alpha_1 = 1/8$, $\alpha_2 = 1/4$)	92,077	3,1	4,71%
Std-every ($\alpha_1 = 1/8$, $\alpha_2 = 1/4$)	94,081	3,1	5,09%
Fast ($\alpha_1 = 1/2$, $\alpha_2 = 1/4$)	90,212	3,0	7,27%
Take-last (α_1, $\alpha_2 = 1$)	93,490	3,3	19,57%
Take-last-every (α_1, $\alpha_2 = 1$)	97,098	3,5	20,20%
Take-last (α_1, $\alpha_2 = 1$, $RTO_{min} = 1$ s)	145,571	8,5	1,30%

Tabla 3: Efecto de variar los parámetros α_1 y α_2 del EWMA

Todos los restantes estimadores usan $G = 1$ mseg. El extremo *take-first* de $\alpha_1 = \alpha_2 = 0$ simplemente usa la primera medición de RTT para inicializar tanto *SRRTT* ← RTT como *RTTVAR* ← _ RTT, obteniendo RTO ← 3 RTT. Nunca cambia nuevamente *SRTT*, *RTTVAR* o RTO (salvo por el retardo de RTO ante retransmisiones repetidas, invirtiéndolo al finalizar el período de retransmisiones). La primera variante del mismo muestra el uso de $RTO_{min} = 1$ seg. y la segunda $RTO_{min} = 0$ seg. En el otro extremo de la tabla tenemos *take-last,* que siempre establece *SRTT* ← RTT y *RTTVAR* ← $|SRTT_{prev} - RTT|$. La variante *take-last-every* es igual, excepto en que se cronometran todos los paquetes en vez de sólo uno por recorrido de ida-vuelta. La última variante eleva el valor mínimo de RTT a 1 seg.

Entre estos extremos analizamos el rango desde *very-slow*, el cual reduce los parámetros a un décimo de su valor normal (dados para el estimador *std*) hasta *fast*, que duplica el valor de los mismos, y entre medio algunas variantes de *time-every-packet*.

En la tabla podemos ver que la configuración de los parámetros de EWMA produce escasa diferencia sobre el funcionamiento correcto del estimador. Tal es así, que si nuestro objetivo es minimizar la tasa de interrupciones erróneas manteniendo la intensidad, podríamos seleccionar el extremadamente simple estimador *take-first*, que apenas se adapta a las condiciones cambiantes de la ruta de la red,[2] o bien elegir el *slow*, que en promedio incurre en un 25% menos de retardo normalizado por interrupción, representando un punto de equilibrio que minimiza *B* localmente. Al igual que con [WS95], cronometrar cada paquete no resulta muy diferente a cronometrar solo un paquete por RTT, a pesar de que al cronometrar cada paquete ejecutamos muchas más mediciones a través de los EWMA por unidad de tiempo. Ello a su vez provoca que los EWMA ajusten más rápidamente los valores de SRTT y RTTVAR a las condiciones presentes en la red, y que se alejen más rápidamente de condiciones pasadas, efecto similar al de usar valores más altos para α_1 y α_2.

Observamos que a medida que el temporizador se adapta mas rápidamente, *B* aumenta continuamente, con *take-last-every* generando en promedio una interrupción errónea por cada cinco, mostrando correlaciones en las variaciones de RTT que cubren los recorridos de ida-vuelta múltiples. Podemos disminuir significativamente este inconveniente aumentando RTT_{min} a 1 seg. , pero a costa de degradar en gran medida la puntualidad de la respuesta del estimador, por lo que resulta mejor usar en su lugar la correspondiente variante *take-first*.

También evaluamos variar los parámetros de EWMA para RTO_{min} = 500 mseg. Hallamos que *~W* aumenta en torno al 50%, con la diferencia disminuyendo a través de las variantes de los estimadores, mientras que *B* se reduce en un factor de 4-8, demostrando todavía más el efecto dominante del RTO mínimo.

Finalmente, varios caminos en N_2 contiene vínculos lentos y con buenos buffers, que originan incrementos elevados y constantes del RTT (hasta de varios segundos). Podríamos esperar que *take-first* operase bastante mal para estas conexiones, pues la medida del primer RTT tiene escasa relación con los RTT subsiguientes, pero de hecho *take-first* funciona de un modo relativamente bueno. La clave reside en la última parte del paso 5 de § 2.2: el temporizador del RTO es reiniciado con la llegada de cada ACK correspondiente a nuevos datos. Consecuentemente, cuando los datos fluyen, el RTO tiene un término RTT adicional implícito [Lud99], y con *take-first* esto es suficiente para evitar interrupciones erróneas aun con los RTT aumentando en dos órdenes de magnitud. En realidad, ¡ *take-first* resulta mejor para dichas

[2] Aun cuando que *take-first* y *take-last* muestran en general un comportamiento aceptable en comparación con otros estimadores, estos estimadores de RTO pueden funcionar muy mal en caminos de red que sufren cambios de RTT grandes y repentinos.

2.8 Impact of Bad Timeouts

We finish our study of RTO estimators with brief comments concerning the impact of bad timeouts.

Any time a TCP times out unnecessarily, it suffers not only a loss of useful throughput, but, often more seriously, unnecessarily cuts *ssthresh* to half the current, sustainable window, and begins a new slow start. In addition, because the TCP is now sending retransmitted packets, unless it uses the TCP timestamp option, it cannot safely measure RTTs for those packets (per Karn's algorithm [KP87]), and thus it will take a long time before the TCP can adapt its RTT estimate in order to improve its broken RTO estimate. (See [Pax97a] for an illustration of this effect.)

Bad timeouts can therefore have a major negative impact on a TCP connection's performance. However, they do *not* have much of an adverse impact on the *network*'s performance, because by definition they occur at a time when the network is not congested to the point of dropping the connection's packets. This in turn leads to the observation that if we could undo the deleterious effects upon the TCP connection of cutting *ssthresh* and entering slow start, then a more aggressive RTO estimator would be more attractive, as TCP would be able to sustain bad timeouts without unduly impairing performance or endangering network stability.

When TCP uses the timestamp option, it can unambiguously determine that it retransmitted unnecessarily by observing a later ACK that echoes a timestamp from a packet sent prior to the retransmission. (A TCP could in principle also do so using the SACK option.) Such a TCP could remember the value of *ssthresh* and *cwnd* prior to the last retransmission timeout, and restore them if it discovers the timeout was unnecessary.

Even without timestamps or SACK, the following heuristic might be considered: whenever a TCP retransmits due to RTO, it measures ΔT, the time from the retransmission until the next ACK arrives. If ΔT is less than the minimum RTT measured so far, then arguably the ACK was already in transit when the retransmission occurred, and the timeout was bad. If the ACK only comes later than the minimum RTT, then likely the timeout was necessary.

We can assess the performance of this heuristic fairly simply. For our usual $G = 1$ msec estimator, a total of 8,799 good and bad timeouts were followed by an ACK arriving with ΔT less than the minimum measured RTT. Of these, fully 75% correspond to *good* timeouts, indicating that, surprisingly, the heuristic generally fails. The failure indicates that sometimes the smallest RTT seen so far occurs right after a timeout, which we find is in fact the case, perhaps because the lull of the timeout interval gives the network path a chance to drain its load and empty its queues.

However, if the threshold is instead $f = \frac{3}{4}$ of the minimum RTT, then only 20% of the corresponding timeouts are *good* (these comprise only 1% of all the *good* timeouts). For $f = \frac{1}{2}$, the proportion falls to only 2.5%. With these reduced thresholds the chance of detecting a bad timeout falls from 74% to 68% or 59%, respectively.

We evaluated the modified heuristic and found it works well: for $f = \frac{1}{2}$, B drops from 4.71% to 2.39%, a reduction of nearly a factor of two, and enough to qualify the estimator as a "sweet spot."

3 Estimating Bandwidth

We now turn to the second estimation problem, determining the amount of bandwidth available to a new connection. Clearly, if a transport protocol sender knows the available bandwidth, it would like to immediately begin sending data at that rate. But in the absence of knowing the bandwidth, it must form an estimate. For TCP, this estimate is currently made by exponentially increasing the sending rate until experiencing packet loss. The loss is taken as an implicit signal that the rate had grown too large, so the rate is effectively halved and the connection continues in a more conservative fashion. In the context of TCP, the goal in this section is to determine the efficacy of different algorithms a TCP connection might use during its start-up to determine the appropriate sending rate without pushing on the network as hard as does the current mechanism. In a more general context, the goal is to explore the degree to which the timing structure of flights of packets can be exploited in order to estimate how fast a connection can safely transmit.

We assume familiarity with the standard TCP congestion control algorithms [Jac88, Ste97, APS99]: the state variable *cwnd* bounds the amount of unacknowledged data the sender can currently inject into the network, and the state variable *ssthresh* marks the *cwnd* size at which a connection transitions from the exponential increase of "slow start" to the linear increase of "congestion avoidance." Ideally, *ssthresh* gives an accurate estimate of the bandwidth available to the connection, and congestion avoidance is used to probe for additional bandwidth that might appear in a conservative, linear fashion.

A new connection begins slow start by setting *cwnd* to 1 segment,[3] and then increasing *cwnd* by 1 segment for each ACK received. If the receiver acknowledges every k segments, and if none of the ACKs are lost, then *cwnd* will increase by about a factor of $\gamma = 1 + \frac{1}{k}$ every RTT. Most TCP receivers currently use a "delayed acknowledgment" policy for generating ACKs [Bra89] in which $k = 2$ and hence $\gamma = \frac{3}{2}$, which is the value we assume subsequently.

Note that if during one round-trip a connection has N segments in flight, then during slow start it is possible, during the next RTT, to overflow a drop-tail queue along the path such that $(\gamma - 1)N = N/k$ segments are lost in a group, if the queue was completely full carrying the N segments during the first round-trip. Such loss will in general significantly impede performance, because when multiple segments are dropped from a window of data, most current TCP implementations will require at least one retransmission timeout to resend all dropped segments [FF96, Hoe96]. However, during congestion avoidance, which can be thought of as a connection's steady-state, TCP increases *cwnd* by at most one segment per RTT, which ensures that *cwnd* will overflow a queue by at most one segment. TCP's fast retransmit and fast recovery algorithms [Jac90, Ste97, APS99] provide an efficient method for recovering from a single dropped segment without relying on the retransmission timer [FF96].

Hoe [Hoe96] describes a method for estimating *ssthresh* by multiplying the measured RTT with an estimate of the bottleneck bandwidth (based on the packet-pair algorithm outlined in [Kes91]) at the beginning of a transfer. [Hoe96] showed that correctly estimating *ssthresh* would eliminate the large loss event that often ends slow start (as discussed above). Given that Hoe's results were based on simulation, an important follow-on question is to explore the degree to which these results are applicable to actual, measured TCP connections.

There are several other mechanisms which mitigate the problems caused by TCP's slow start phase, and therefore lessen the need to estimate *ssthresh*. First, routers implementing Random Early Detection (RED) [FJ93, BCC$^+$98] begin randomly dropping segments at a low rate as their average queue size increases. These drops implicitly signal the connection to reduce its sending rate before the queue overflows. Currently, RED is not widely deployed. RED also does not guarantee avoiding multiple losses within a window of data, especially in the presence of heavy congestion. However, RED also has the highly appealing property of not requiring the deployment of any changes to current TCP implementations.

Alternate loss recovery techniques that do not rely on TCP's re-

[3] Strictly speaking, *cwnd* is usually managed in terms of bytes and not segments (full-sized data packets), but conventionally it is discussed in terms of segments for convenience. The distinction is rarely important. Also, [APS99] allows an initial slow start to begin with *cwnd* set to 2 segments, and an experimental extension to the TCP standard allows an initial slow start to begin with *cwnd* set to 3 or possibly 4 segments [AFP98]. We comment briefly on the implications of this change below.

conexiones que los estimadores que siguen el cambio del RTT! Sucede así porque los estimadores más adaptativos terminan esperando después de la llegada del primer ACK, y antes de disparar el RTO, mientras que en este caso *take-first* retransmite con la oportuna rapidez. Pero esta ventaja resulta especial en aquellas conexiones que presentan una respuesta altamente regularizada. No obstante, sugiere la idea de una interrupción por respuesta, lo cual se trata brevemente en § 4.

2.6 Variando el Factor RTTVAR

Factor *RTTVAR*	*W*	*~W*	*B*
k = 16	168,002	7.0	0.59%
k = 12	144,053	5.7	0.81%
k = 8	118,858	4.4	1.52%
k = 6	105,681	3.8	2.43%
Adapt	94,220	3.2	4.44%
k = 4	92,077	3.1	4.71%
k = 3	85,264	2.8	7.68%
k = 2	78,565	2.5	13.64%
RTO_{min} = 750 mseg., *k* = 6	128,226	6.7	0.50%
RTO_{min} = 750 mseg.	121,566	6.5	0.76%
Take-first$_{250mseg.}$, *k* = 6	163,799	6.4	0.70%
RTO_{min} = 500 mseg., *k* = 6	112,514	5.1	0.69%
RTO_{min} = 500 mseg.	102,264	4.8	1.02%
RTO_{min} = 250 mseg., *k* = 6	106,139	4.0	1.29%
RTO_{min} = 250 mseg	92,866	3.5	2.27%

Tabla 4: Efecto de variar el factor *k* de *RTTVAR*

El último parámetro de estimación del RTO que consideramos es *k*, el multiplicador de *RTTVAR* al calcular RTO según la Ecuación 1. Para la implementación estándar *k* = 4, la Tabla 4 muestra los efectos de variar *k* entre 2-16, para *G* = 1 mseg. y RTO_{min} = 0 seg. El estimador *adapt* comienza con k = 4, pero se duplica cada vez que incurre en una interrupción errónea.

Claramente, *k* ofrece un punto de apoyo para establecer un compromiso en el tiempo de espera de interrupciones innecesarias, pero sin un "punto de equilibrio" obvio. No obstante, como se muestra en la segunda mitad de la tabla, este equilibrio cambia ligeramente cuando aumentamos RTO_{min}. Por ejemplo, observamos que la combinación RTO_{min} = 250 mseg., *K* = 4 se comporta mucho mejor que la variante *k* = 6 *sin mínimo,* y que RTO_{min} = 250 mseg., *k* = 6 funciona mejor que la variante *k* = 8. Incluso el estimador *take-first*, tan extremadamente simple, cuando usa *k* = 6 y RTO_{min} = 250 mseg., se comporta ligeramente mejor que el estimador normal RTO_{min} = 750 mseg.

2.7 ¿Podemos estimar mejor el RTO?

Habiendo evaluado los efectos de diversos parámetros del estimador, y encontrado, en su mayor parte, únicamente compromisos y poca información acerca de la manera de establecer "puntos de equilibrio", abordamos ahora la pregunta de si existe en efecto alguna posibilidad de diseñar estimadores todavía mejores. Una consideración clave para responder a esta pregunta es: en el momento de subestimar, ¿por cuánto lo hacemos? Si, por ejemplo, las subestimaciones tienden a ser inferiores a un RTT, entonces ello sugeriría una modificación de la Ecuación 1, según la cual se aplique un factor de 2 a *SRTT*.

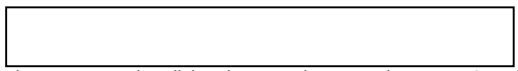

Figura 1: Retardo adicional necesario para evitar un RTO erróneo

Sea *A* un indicador del retardo adicional necesario para evitar un RTO erróneo. La figura 1 muestra las distribuciones acumuladas de la relación entre *A* y *RTTVAR* (línea continua), del máximo RTT observado hasta ese momento (línea punteada) y del RTO (línea punteada) para el estimador usual con *G* = 1 mseg. La relación entre A y *RTTVAR* varía en diversos órdenes de magnitud, indicando que es improbable hallar un valor particular de *k* en la Ecuación 1 que resuelva eficientemente la mayor parte de las restantes interrupciones erróneas.

También se observa que *A* es generalmente inferior al RTO presente y al RTT máximo detectado hasta ese punto. Ello sugiere sumar uno de estos valores al RTO con objeto de hacerlo suficientemente conservador, evitando así interrupciones erróneas. Sin embargo, hacer esto tiene un efecto bastante similar al de otras variantes del estimador que aumentan la espera basándose en otros factores (por ejemplo, el valor de *k*). Por ejemplo, cambiando el estimador estándar de *k* = 4 indicado en la Tabla 4 para usar el doble del RTO calculado (o sea, sumar un término adicional al RTO) disminuye *B* de 4,71% a 0,57%, pero aumenta *~W* de 3,1 a 5,7 (algo más favorable que usar simplemente *k* = 12, pero no del todo mejor).

Para RTO_{min} = 0,5 seg el gráfico es muy similar, con una separación ligeramente mayor entre las líneas de RTO y RTT MAX. Por lo tanto, la Figura 1 sugiere una relación fundamental entre agresividad y ocurrencia de interrupciones erróneas.

En relación a ello, puede plantearse que si un paquete se retransmite innecesariamente, ¿refleja ello un aumento transitorio del RTT, o un incremento considerable? Averiguamos que alrededor del 62% de las interrupciones erróneas fueron seguidas por valores de RTT inferiores al RTO, por lo tanto la interrupción errónea reflejó un aumento transitorio del RTT. Otro 24% fue seguido por un RTT con un valor exactamente doble, y un poco más del 2% fue seguido por valores de RTT diez veces superiores. Por lo tanto, la mayor parte de las veces un aumento significativo del RTT resultó mas bien transitorio, si bien queda un residuo no despreciable que implica un incremento sostenido.

2.8 El impacto de las interrupciones erróneas

Finalizamos nuestro estudio de los estimadores de RTO con un breve comentario relativo al impacto de las interrupciones erróneas.

Toda vez que un TCP se interrumpe innecesariamente, no sólo sufre una pérdida de salida útil, sino que también, a menudo de manera más seria, disminuye el *ssthresh* a la mitad de la ventana sostenible presente, comenzando un nuevo arranque lento. Además, dado que ahora el TCP está enviando paquetes retransmitidos, salvo que use la opción de marca temporal, el TCP no puede medir con exactitud el RTT de estos paquetes (ver algoritmo de Karn [KP87]), y por lo tanto transcurrirá bastante tiempo antes de que el TCP adapte su estimación del RTT a fin de mejorar su estimación truncada del RTO. (Ver en [Pax97a] una descripción de dicho efecto.)

Por lo tanto, las interrupciones erróneas pueden tener un efecto muy negativo en la eficiencia de una conexión TCP. Sin embargo, *no* presentan un gran impacto sobre la eficiencia de la *red*, pues por definición tienen lugar en un momento en el cual la red no está congestionada hasta el extremo de dejar perder los paquetes de la conexión. Esto a su vez conduce a pensar en que, si pudiéramos revertir los efectos degradantes de la conexión TCP que disminuyen el *ssthresh,* y entrar en un arranque *lento*, entonces resultaría más interesante un estimador de RTO más agresivo, pues de este modo el TCP sería capaz de soportar las interrupciones erróneas sin deteriorar indebidamente la eficiencia de la red, o bien poner en riesgo su estabilidad.

Cuando el TCP usa la opción de marca temporal, puede detectar ambiguamente que efectuó una retransmisión innecesaria al observar un ACK posterior que refleja una marca temporal enviada antes de la

transmission timer have been developed to diminish the impact of multiple losses in a flight of data. SACK-based TCPs [MM96, MMFR96, FF96] provide the sender with more complete information about which segments have been dropped by the network than non-SACK TCP implementations provide. This allows algorithms to quickly recover from multiple dropped segments (generally within one RTT following loss detection). One shortcoming of SACK-based approaches, however, is that they require implementation changes at both the sender and the receiver. Another class of algorithms, referred to as "NewReno" [Hoe96, FF96, FH99], does not require SACKs, but can be used to effectively recover from multiple losses without requiring a timeout (though not as quickly as when using SACK-based algorithms). In addition, NewReno only requires implementation changes at the sender. The estimation algorithms studied in this paper all require changes to the sender's TCP implementation. So, we assume that the sender TCP implementation will have some form of the NewReno loss recovery mechanism.

3.1 Methodology

In this section we discuss a number of algorithms for estimating *ssthresh* and our methodology for assessing their effectiveness. We begin by noting a distinction between *available bandwidth* and *bottleneck bandwidth*. In [Pax97b] we define the first as the maximum rate at which a TCP connection exercising correct congestion control can transmit along a given network path, and the second as the upper bound on how fast *any* connection can transmit along the path due to the data rate of the slowest forwarding element along the path.

Our ideal goal is to estimate *available bandwidth* in terms of the correct setting of *ssthresh* such that we fully utilize the bandwidth available to a given connection, but do not exceed it (more precisely: only exceed it using the linear increase of congestion avoidance). Much of our analysis, though, is in terms of bottleneck bandwidth, as this is both an upper bound on a good *ssthresh* estimate, and a quantity that is more easily identifiable from the timing structure of a flight of packets, since for any two data packets sent back-to-back along an uncongested path, their interarrival time at the receiver directly reflects the bottleneck bandwidth along the path.[4]

Note that in most TCP implementations *ssthresh* is initialized to an essentially unbounded value, while here we concentrate on lowering this value in an attempt to improve performance by avoiding loss or excessive queueing. Thus, all of the algorithms considered in this section are *conservative*, yet they also (ideally) do not impair a TCP's performance relative to TCPs not implementing the algorithm. However, if an estimator yields too small a value of *ssthresh*, then the TCP will indeed perform poorly compared to other, unmodified TCPs.

As noted above, one bottleneck bandwidth estimator is "packet pair" [Kes91]. In [Pax97b] we showed that a packet pair algorithm implemented using strictly sender-side measurements performs poorly at estimating the bottleneck bandwidth using real traffic. We then developed a more robust method, Packet Bunch Mode (PBM), which is based on looking for modalities in the timing structure of groups of back-to-back packets [Pax97b, Pax97c]. PBM's effectiveness was assessed by running it over the NPD datasets (including the $\mathcal{N}_2$ dataset referred to earlier), arguing that the algorithm was accurate because on those datasets it often produced estimates that correspond with known link rates such as 64 kbps, T1, E1, or Ethernet.

PBM analyzes an entire connection trace before generating any bottleneck bandwidth estimates. It was developed for assessing network path properties and is not practical for current TCP implementations to perform on the fly, as it requires information from both the sender and receiver (and is also quite complicated). However, for our purposes what we need is an accurate assessment of a given network path's bottleneck bandwidth, which we *assume* that PBM provides. Thus, we use PBM to calibrate the efficacy of the other *ssthresh* estimators we evaluate.

Of the 18,490 traces available in $\mathcal{N}_2$, we removed 7,447 (40%) from our analysis for the following reasons:

- Traces marred by packet filter errors [Pax97a] or major clock problems [Pax98]: 15%. Since these problems most likely do not reflect network conditions along the path between the two hosts in the trace, removing these traces arguably does not introduce any bias in our subsequent analysis.

- Traces in which the first retransmission in the trace was "avoidable," meaning had the TCP sender merely waited longer, an ACK for the retransmitted segment would have arrived: 20%. Such retransmissions are usually due to TCPs with an initial RTO that is too short [Pax97a, PAD+99]. We eliminate these traces because the retransmission results in *ssthresh* being set to a value that has little to do with actual network conditions, so we are unable to soundly assess how well a larger *ssthresh* would have worked. Removing these traces introduces a bias against connections with particularly high RTTs, as these are the connections most likely to engender avoidable retransmissions.

- Traces for which the PBM algorithm failed to produce a single, unambiguous estimate: 4%. We need to remove these traces because our analysis uses the PBM estimate to calibrate the different estimation algorithms we assess, as noted above. Removing these traces introduces a bias against network conditions that make PBM itself fail to produce a single estimate: multi-channel paths, changes in bottleneck bandwidth over the course of a connection, or severe timing noise.

After removing the above traces, we are left with 11,043 connections for further analysis. We use trace-driven simulation to assess how well each of the bandwidth estimation algorithms perform. We base our evaluation on classifying the algorithm's estimate for each trace into one of several *regions*, representing different levels of impact on performance.

For each trace, we define three variables, B, L and E. B is the bottleneck bandwidth estimate made using the PBM algorithm. L is the *loss point*, meaning the transmission rate in effect when the first lost packet was sent (so, if the first lost segment was sent with *cwnd* corresponding to W bytes, then $L = W/RTT$ bytes/second). If the connection does not experience loss, L' is the bandwidth attained based on the largest *cwnd* observed during the connection.[5] When $L > B$ or $L' > B$, the network path is essentially free of competing traffic, and the loss is presumed caused by the connection itself overflowing a queue in the network path. Conversely, if L or L' is less than B, the path is presumed congested. Finally, E is the bandwidth estimate made by the *ssthresh* estimation algorithm being assessed.

In addition, define $\text{seg}(x) = (x \cdot \text{RTT})/\textit{segment size}$ representing the size of the congestion window, in segments, needed to achieve a bandwidth of x bytes/second, for a given TCP segment size and RTT. (Note that as defined, $\text{seg}(x)$ is continuous and not discrete.)

3.1.1 Connections With Loss

Given the above definitions, and a connection which contains loss, we assess an estimator's performance by determining which of the following six regions it falls into. Note that we analyze the regions in the order given, so an estimate will not be considered for any regions subsequent to the first one it matches.

[4] Providing the path isn't "multi-channel" or subject to routing changes [Pax97b].

[5] Strictly speaking, it's the largest flight observed during the connection, which might be smaller than *cwnd* due to the connection running out of data to send, or exhausting the (32-64KB) receiver window.

retransmisión (en principio, un TCP podría hacer lo mismo usando la opción SACK). Tal TCP podría memorizar el valor del *ssthresh* y *cwnd* previo a la última interrupción por retransmisión y restaurarlo si descubre que la interrupción resultó innecesaria.

Aun sin marca temporal o SACK, podría considerarse la siguiente heurística: siempre que un TCP retransmite debido al RTO, mide ΔT, lapso de tiempo comprendido entre la retransmisión y la llegada del siguiente ACK. Si ΔT resulta ser inferior al RTT mínimo medido hasta el momento, entonces es admisible que el ACK ya estaba en tránsito al tener lugar la retransmisión, y por lo tanto la interrupción fue errónea. Si el ACK llega después del RTT mínimo, entonces es probable que la interrupción fuese necesaria.

Podemos evaluar la eficacia de esta heurística de manera bastante simple. Para nuestro estimador usual de G = 1 mseg., un total de 8,799 retransmisiones acertadas y erróneas fueron seguidas por un ACK llegando con ΔT inferior al mínimo RTT medido. De estos, un total de 75% corresponden a interrupciones acertadas, indicando que, de manera sorprendente, la heurística fracasa de modo general. El fracaso indica que algunas veces el menor RTT detectado hasta el instante ocurre exactamente después de una interrupción, lo cual de hecho es el caso, debido quizá a que el silencio durante el período de interrupción da a la ruta de red la oportunidad de evacuar su carga y vaciar las colas de espera.

Sin embargo, si el umbral es en cambio *f* = _ del RTT mínimo, el 20% de las correspondientes interrupciones resultan *acertadas* (ello supone sólo el 1% de todas las interrupciones *acertadas*). Para *f* = _, la proporción cae a sólo 2.5%. Con estos umbrales reducidos, la probabilidad de detectar una interrupción errónea disminuye del 74% al 68% y 59% respectivamente.

Evaluamos la heurística modificada y encontramos que funciona bien: para *f* = _, *B* cae de 4,71% a 2,39%, una reducción con un factor de casi dos, suficiente para calificar al estimador como equilibrado.

3 Estimación del ancho de banda

Volvemos ahora al segundo problema de estimación, que es determinar la cantidad de ancho de banda disponible para una nueva conexión. Obviamente, si el emisor de un protocolo de transporte conoce el ancho de banda disponible, deseará comenzar a enviar datos a esa frecuencia inmediatamente. Pero si se desconoce el ancho de banda, es preciso elaborar una estimación. Para el TCP, dicha estimación se hace actualmente aumentado exponencialmente la frecuencia de transmisión hasta que se produzca una pérdida de paquete. La pérdida se considera como una señal implícita de que la frecuencia ha aumentado demasiado, por lo tanto la frecuencia se reduce efectivamente a la mitad y la conexión continúa de manera más conservadora. En el ámbito del TCP, el propósito de esta sección es determinar la eficacia de diferentes algoritmos que una conexión TCP podría utilizar durante su arranque con objeto de establecer la frecuencia de emisión apropiada sin comprometer a la red de manera tan exigente como lo hace el mecanismo actual. En un contexto más general, el objetivo es investigar el grado hasta el cual se puede explotar el mecanismo de regulación de emisión de paquetes a fin de estimar la mayor frecuencia a la que una red puede transmitir confiablemente.

Asumimos una cierta familiaridad con los algoritmos de control de congestión del TCP standard [Jac88, Ste97, APS99]. Así, la variable de estado *cwnd* limita la cantidad de datos sin reconocimiento que un emisor puede inyectar a la red en cada instante, y la variable de estado *ssthresh* establece el valor de *cwnd* al cual una conexión pasa del modo de "arranque lento" con incremento exponencial al de "prevención de congestiones" con incremento lineal. Idealmente, *ssthresh* da una estimación precisa del ancho de banda disponible para la conexión, y la prevención de congestión se usa, de manera conservadora y lineal, para detectar cualquier ancho de banda adicional que pudiera surgir durante la transmisión.

Una nueva conexión comienza en arranque lento fijando *cwnd* igual a 1 segmento[3], y luego incrementando *cwnd* en 1 segmento por cada ACK recibido. Si el receptor envía un reconocimiento cada *k* segmentos, y si no se pierde ningún ACK, entonces *cwnd* aumentará en un factor cercano a $\gamma = 1 + {}^1/_k$ cada RTT. La mayor parte de los receptores TCP usan actualmente un esquema de reconocimiento retardado para generar los ACK [Bra89], en el cual $k = 2$, y por tanto, $\gamma = {}^3/_2$, valor que asumiremos en adelante.

Obsérvese que si durante un recorrido ida-vuelta una conexión tiene *N* segmentos en curso, entonces en el arranque lento es posible que durante el próximo RTT, se sobrepase una cola de paso a lo largo del camino, de modo tal que se pierdan $(\gamma - 1)N = N / k$ segmentos de un grupo si la cola estaba transportando la totalidad de los *N* segmentos durante el primer recorrido ida-vuelta. En general, esta pérdida afectará significativamente a la eficiencia, pues cuando múltiples segmentos caen en una ventana de datos, la mayoría de las implementaciones actuales de TCP demandarán por lo menos una interrupción de retransmisión que reenvíe todos los segmentos caídos [FF96, Hoe96]. No obstante, durante el modo de prevención de congestión, que puede ser considerado como el estado continuo de una conexión, TCP aumenta *cwnd* en no más de un segmento por RTT, lo cual asegura que *cwnd* excederá a una cola como máximo en un segmento. Los algoritmos de retransmisión rápida y recuperación rápida de TCP [Jac90, Ste97, APS99] facilitan un método eficiente para recomponer la caída de un sólo segmento sin depender del temporizador de retransmisión [FF96].

Hoe [Hoe96] describe un método para estimar *sstresh* multiplicando el RTT medido por una estimación (basada en el algoritmo de par de paquete delineado en [Kes91]) del ancho de banda más estrecho al inicio de una transferencia. [Hoe96] demostró que la estimación correcta de *ssthresh* eliminaría el evento de alta pérdida que a menudo finaliza con el arranque lento (tal y como se ha descrito anteriormente). Dado que los resultados de Hoe estaban basados en simulaciones, una cuestión que queda pendiente es investigar hasta qué punto estos resultados son aplicables a las conexiones medibles del TCP real.

Existen otros mecanismos que mitigan los problemas causados por la fase de arranque lento del TCP, y que por lo tanto disminuyen la necesidad de estimar *ssthresh*. En primer lugar, los ruteadores que aplican Detección Temprana al Azar (RED por Random Early Detection) [FJ93, BCC+98], al aumentar el tamaño promedio de las colas, comienzan a descartar segmentos al azar a baja frecuencia. Estas caídas inducen implícitamente a la conexión a reducir su frecuencia de emisión antes de que la cola se desborde. Actualmente, RED no está ampliamente difundido. RED tampoco garantiza la prevención de pérdidas múltiples dentro de una ventana de datos, especialmente ante la posibilidad de que se produzca una alta congestión. No obstante, RED también presenta la atrayente propiedad de no demandar la realización de cambio alguno en las actuales implementaciones del TCP.

Para disminuir las pérdidas en un envío de datos también se han desarrollado técnicas alternativas que no dependen del temporizador de transmisión del TCP. Los TCP basados en SACK [MM96, MMFR96,

[3] En sentido estricto, *cwnd* se expresa usualmente en términos de bytes, y no de segmentos (paquetes de datos de tamaño completo). Por convención y conveniencia, se analiza en términos de segmentos. La diferencia no es generalmente importante. Asimismo, [ASP99] permite un arranque lento inicial comenzando con *cwnd* igual a 2 segmentos, y una extensión experimental del TCP estándar permite un arranque lento inicial comenzando con *cwnd* igual a 3, o incluso posiblemente 4 segmentos [AFP98]. Más adelante comentaremos brevemente las implicaciones de esta modificación.

No Estimate Made. The estimator failed to produce an *ssthresh* estimate before the first segment loss occurred in the trace.

No Impact. The estimate satisfies $E \geq \gamma L$. This means that E is a sufficiently large overestimate that the connection will behave no differently using that estimate than it would if no estimate were made.

Some Loss Prevention. When $L \leq E < \gamma L$ holds, the given *ssthresh* estimate prevents some, but not all, loss of data packets. While the estimate is greater than the loss point, it reduces the size of the last slow start flight by $N_s = \text{seg}(\gamma L - E)$ segments. Therefore, up to N_s segment drops may be prevented.

Steady-State. When $\frac{L}{2} \leq E < L$ holds, we classify the *ssthresh* estimate as "steady-state." During congestion avoidance, which defines TCP's steady-state behavior [Jac88, MSMO97], *cwnd* decreases by half upon loss detection and then increases linearly until another loss occurs. So, given the loss point of L, *cwnd* can be expected to oscillate between $\frac{L}{2}$ and L after the connection's second loss event.[6] By making an estimate between $\frac{L}{2}$ and L, the estimator has found the range about which the connection will naturally oscillate, assuming the loss point is stationary.

Optimal. When the analysis reaches this point, we know that $E < \frac{L}{2}$ since none of the above conditions hold. If $\text{seg}(E) \geq \text{seg}(B) - 1$ also holds, then the *ssthresh* estimate reduces the queueing requirement, as follows. Since E is very close to or larger than the bottleneck bandwidth, yet less than $\frac{L}{2}$, we know that the loss point is greater than the bottleneck bandwidth, yet the *ssthresh* estimate is no less than the bottleneck bandwidth or one segment less than the bottleneck bandwidth. (We consider one segment less than the bottleneck bandwidth to be within the range because both slow start and congestion avoidance will take a single RTT to increase *cwnd* to correspond with B—and we prefer to reach that point via congestion avoidance rather than slow start, so we don't overshoot it.)

Thus, assuming the connection lasts long enough, the queue will still be filled to L. However, we will fill the queue more slowly and smoothly than with slow start. Furthermore, when we exceed the queue during congestion avoidance, it is only by one segment, whereas during slow start we will exceed the capacity of the queue by as much as γ times the capacity.[7] When a connection falls into this region, the queue length is initially reduced by $N_q = (L - E) \cdot \text{RTT}$ bytes. Since this region reduces queueing, prevents loss, yet fully utilizes the network path, we deem it "optimal."

Reduce Performance. Finally, if none of the above conditions hold then $E < \frac{L}{2}$ and $E < B$ (these bounds are not tight). We therefore set *ssthresh* too low and force *cwnd* growth to continue linearly, rather than exponentially. When an estimator underestimates $\min(\frac{L}{2}, B)$ by more than half in 50+% of the connections in which performance would be reduced, we consider this to be an especially bad estimate. In this case, the reported percentage of connections experiencing reduced performance is marked with a "*".

[6] The size of *cwnd* when detecting the first loss event is roughly γL. Therefore, the first halving of *cwnd* causes it to be approximately $\frac{\gamma}{2} L$. Each subsequent loss event should only overflow the queue slightly and therefore *cwnd* will be reduced to $\frac{L}{2}$.

[7] Some implementations of congestion avoidance add a constant of $\frac{1}{8}$ times the segment size to *cwnd* for every ACK received during congestion avoidance. This non-standard behavior has been shown to lead to sometimes overflowing the queue by more than a single segment every time *cwnd* approaches L [PAD+99].

Algorithm	No Est.	No Imp.	Prv. Loss	Stdy. State	Opt.	Tot.	Red. Perf.
PBM′	23%	46%	9%	10%	11%	31%	0%
TSSF	42%	1%	1%	3%	0%	4%	52%*
$\text{CSA}_{n=3}^{\nu=0.1}$	62%	20%	6%	9%	2%	17%	2%
$\text{CSA}_{n=2}^{\nu=0.05}$	53%	37%	5%	4%	0%	9%	1%*
$\text{CSA}_{n=2}^{\nu=0.1}$	45%	32%	8%	10%	2%	19%	4%*
$\text{CSA}_{n=2}^{\nu=0.2}$	38%	24%	9%	13%	3%	25%	13%
TCSA	62%	14%	6%	11%	1%	19%	5%
TCSA′	70%	10%	6%	9%	2%	17%	2%
Recv_{min}	11%	32%	6%	13%	4%	23%	34%*
Recv_{avg}	11%	52%	10%	14%	9%	34%	3%
Recv_{med}	11%	48%	10%	14%	10%	34%	7%*
Recv_{max}	11%	65%	7%	8%	8%	23%	0%*

Table 5: Connections with Loss (8,257 traces)

3.1.2 Connections Without Loss

The following regions use L' to assess the impact of *ssthresh* estimation on connections in the dataset that do not experience loss. Each trace is placed into one of the following four regions. (Again, note that we analyze the regions in the order given, so an estimate will not be considered for any regions subsequent to the first one it matches.)

No Estimate Made. The estimator failed to produce an *ssthresh* estimate.

Unknown Effect. When $E \geq L'$ holds, the estimate does not limit TCP's ability to open *cwnd*, as it is above the maximum *cwnd* used by the connection. Since we do not have a good measure of the limit of the network path, nothing more can be assessed about the performance of the estimator.

Optimal. When $\text{seg}(E) \geq \text{seg}(B) - 1$ holds, the estimate is greater than the bottleneck bandwidth and therefore does not limit performance. However, we also know that $E < L'$ due to the above region. Therefore, the estimate reduces the initial queueing requirement similar to the "optimal" region in § 3.1.1.

Reduce Performance. At this point, $E < \min(L', B - \text{seg}^{-1}(1))$ holds, indicating that the estimate failed to provide exponential window growth to L', which is a known safe sending rate. Furthermore, our failure to reach L' is not excused by providing exponential *cwnd* growth long enough to fill the pipe (B bytes/second). We again mark with a "*" those connections for which the reduction is often particularly large.

3.2 Benchmark Algorithm

As noted above, we use PBM as our benchmark in terms of accurately estimating the bottleneck bandwidth. For *ssthresh* estimation, we use a revised version of the algorithm, PBM′, to provide some sort of *upper bound* on how well we might expect any algorithm to perform. (It is not a strong upper bound, since it may be that other algorithms estimate the *available* bandwidth considerably better than does PBM′, but it is the best we currently have available.) The difference between PBM′ and PBM is that PBM′ analyzes the trace only up to the point of the first loss, while PBM analyzes the trace in its entirety. Thus, PBM′ represents applying a detailed, heavyweight, but accurate algorithm on as much of the trace as we are allowed to inspect before perforce having to make an *ssthresh* decision.

As shown in Tables 5 and 6, the PBM′ estimate yields *ssthresh* values that rarely hurt performance, regardless of whether the connection experiences loss. Each column lists the percentage of traces which, for the given estimator, fell into each of the regions discussed in § 3.1.1. The **Tot.** column gives the percentage of traces for which the estimator improved matters by attaining either the **prevent loss**,

FF96] ofrecen al emisor información más completa que las implementaciones de TCP sin SACK respecto a los segmentos perdidos por la red. Esto permite a los algoritmos recuperarse rápidamente de la caída de múltiples segmentos (generalmente dentro del RTT que sigue a la detección de la pérdida). Sin embargo, un inconveniente de los esquemas basados en SACK es que requieren cambios en la implementación tanto del emisor como del receptor. Otra clase de algoritmos, denominados "NewRemo" [Hoe96, FF96, FH99] no requieren SACK, y pueden por tanto ser usados para recomponerse efectivamente de pérdidas múltiples (aunque no tan rápidamente como cuando se utilizan algoritmos basados en SACK). Además el NewRemo sólo exige cambios de implementación en el emisor. Todos los algoritmos de estimación estudiados en esta sección demandan cambios a la implementación del TCP en el emisor. Por lo tanto asumimos que la implementación TCP del emisor presentará, en cierto modo, un mecanismo de recuperación de pérdidas similar al de NewRemo.

3.1 Metodología

En esta sección presentamos algoritmos para estimar *ssthresh* , así como nuestra metodología para evaluar su efectividad. Comenzamos por resaltar una diferencia entre *ancho de banda disponible* y *ancho de banda de cuello de botella*. En [Pax97b] definimos el primero como la frecuencia máxima a la cual una conexión de TCP puede transmitir a lo largo de una cierta ruta de red ejerciendo un control de congestión correcto, y el segundo como el límite superior de velocidad a la cual *cualquier* conexión puede transmitir en la red debido a la tasa de datos del elemento encaminador más lento a lo largo de dicha red.

Nuestro objetivo ideal es estimar el ancho de banda disponible en términos de la correcta configuración del *ssthresh,* de modo tal que utilicemos completamente el ancho de banda disponible para una conexión dada, pero sin excederlo (o más precisamente: solo excediéndolo utilizando el incremento lineal de prevención de congestión). No obstante, la mayor parte de nuestro análisis se presenta en términos de ancho de banda del estrechamiento, dado que éste es tanto un límite superior para una buena estimación de *sstresh,* como una cantidad mas fácilmente identificable dentro del mecanismo de regulación de un recorrido de paquetes, ya que para cualquier par de paquetes enviados sucesivamente uno tras otro (back-to-back) a lo largo de una ruta descongestionada, el intervalo entre la llegada de ambos al receptor refleja directamente el ancho de banda del estrechamiento a lo largo del camino.[4]

Obsérvese que en la mayoría de las implementaciones de TCP, *ssthresh* se inicializa con un valor ciertamente ilimitado, mientras que aquí nos centramos en disminuir este valor en un intento de mejorar la eficiencia mediante la prevención de pérdidas, o la proliferación excesiva de colas de espera. Por lo tanto, todos los algoritmos considerados en esta sección son *conservadores*, pero aún así (idealmente) no degradan la eficiencia del TCP en cuestión respecto a otros que no utilicen el algoritmo. Sin embargo, si el estimador produce un valor muy bajo de *ssthresh*, entonces el TCP se comportará realmente de modo deficiente en comparación con otros TCP que no hayan sido modificados.

Como se observó anteriormente, un estimador de ancho de banda del estrechamiento está basado en "pares de paquetes" sucesivos [Kes91]. En [Pax97b] demostramos que un algoritmo de par de paquetes sucesivos implementado usando solamente mediciones en el lado emisor se comporta deficientemente al estimar el ancho de banda del estrechamiento en el tráfico real. Desarrollamos entonces un método más sólido, el Modo de Racimo de Paquetes (PBM por Packet Bunch Mode), que se fundamenta en la búsqueda de modalidades en el mecanismo de regulación de grupos de paquetes sucesivos [Pax97b, Pax97c]. La efectividad del PBM fue evaluada ejecutándolo sobre conjuntos de datos NPD (incluyendo el conjunto de datos N_2 mencionado anteriormente), demostrando que el algoritmo era preciso dado que en estos conjuntos de datos habitualmente daba estimaciones que correspondían a velocidades de conexión reconocidas tales como 64 kbps, T1, E1 o Ethernet.

PBM analiza una traza completa de conexión antes de generar cualquier estimación de ancho de banda al estrechamiento. Fue desarrollado para evaluar propiedades de rutas de red, y en consecuencia no resulta práctico para ser ejecutado en tiempo real por las implementaciones de TCP corrientes, ya que requiere información tanto del emisor como del receptor (y además es bastante complicado). Sin embargo, para nuestros fines, lo que necesitamos es una evaluación exacta del ancho de banda al estrechamiento de una ruta de red determinada, lo cual *asumimos* que puede ser ofrecido por el PBM. Por lo tanto, usamos el PBM para calibrar la eficacia de los otros estimadores de *ssthresh* evaluados.

De las 18,490 trazas disponibles en N_2, descartamos de nuestro análisis 7,447 (40%), debido a las siguientes razones:

- Trazas dañadas por errores de filtrado de paquetes [Pax97a] o graves errores de reloj [Pax98]: 15%. Dado que muy probablemente estos problemas no reflejan condiciones de red a lo largo de la ruta entre los dos servidores de la traza, resulta aceptable decir que quitar estas trazas no introduce ningún desvío en nuestro análisis subsiguiente.

- Trazas en las cuales la primera retransmisión acaecida era "evitable", implicando que si el emisor de TCP meramente hubiera esperado más tiempo, un ACK del segmento retransmitido hubiera llegado: 20%. Dichas retransmisiones son usualmente debidas a los TCP que tienen un RTO inicial demasiado bajo [Pax97a, PAD+99]. Eliminamos estas trazas por que las retransmisiones hacen que el *ssthresh* asuma un valor que tiene poca relación con las condiciones presentes en la red, siendo incapaces de evaluar con seguridad cómo hubiera funcionado un *ssthresh* mayor. Eliminar estas trazas introduce un desvío que afecta a las conexiones con RTT particularmente altos, dado que estas son las conexiones más susceptibles de generar retransmisiones evitables.

- Trazas para las cuales el algoritmo PBM no pudo obtener una única e inequívoca estimación: 4%. Debemos eliminar estas trazas por que nuestro análisis, como se mencionó anteriormente, usa la estimación del PBM para calibrar los diferentes algoritmos de estimación evaluados. Quitar estas trazas introduce un desvío enmascarando las condiciones de la red que impidieron al PBM obtener una estimación única: rutas multi-canal, cambios en el ancho de banda de cuello de botella durante el curso de una conexión, o severos niveles de ruido asociados a la temporización.

Tras eliminar las trazas antes indicadas, nos restan 11,043 conexiones para ser analizadas posteriormente. Usamos simulaciones dirigidas con trazadores para evaluar el buen comportamiento de cada uno de los algoritmos de estimación del ancho de banda. Basamos nuestra evaluación en clasificar la estimación del algoritmo para cada traza según una de las varias *regiones*, representando diferentes niveles de impacto sobre la eficiencia.

Para cada traza, definimos tres variables, *B*, *L* y *E*. *B* es la estimación de ancho de banda de cuello de botella obtenida mediante el algoritmo PBM. *L* es el *punto de pérdida*, equivalente a la frecuencia de transmisión cuando se envió el primer paquete perdido (por lo tanto, si el primer paquete perdido fue enviado con *cwnd* igual a *W* bytes, entonces

[4] Siempre y cuando la ruta no sea "multi-canal" o esté sujeta a cambios de ruteo [Pax97b].

Algorithm	No Est.	Unk. Imp.	Opt.	Red. Perf.
PBM′	0%	56%	44%	0%
TSSF	13%	2%	2%	82%*
$CSA_{n=3}^{\nu=0.1}$	24%	42%	13%	22%
$CSA_{n=2}^{\nu=0.05}$	19%	59%	11%	10%
$CSA_{n=2}^{\nu=0.1}$	14%	48%	11%	27%
$CSA_{n=2}^{\nu=0.2}$	13%	34%	11%	43%*
TCSA	24%	25%	8%	44%
TCSA′	27%	33%	11%	28%
$Recv_{min}$	1%	15%	2%	83%*
$Recv_{avg}$	1%	46%	23%	31%*
$Recv_{med}$	1%	45%	28%	26%
$Recv_{max}$	1%	71%	27%	1%

Table 6: Connections without Loss (2,786 traces)

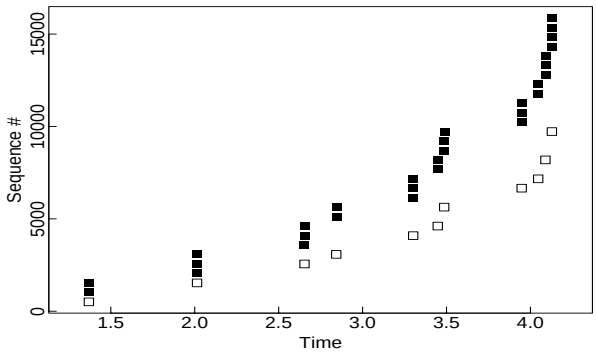

Figure 2: Delayed ACK leading to timing "lull"

steady-state, or **optimal** regions. This column can be directly compared with the last column (**reduce performance**) to assess how a given estimator trades off improvement in some cases with damage in others.

We see that PBM' provides some benefit (steady state, prevention of loss, or optimal) to 31% of the connections that experience loss, and, when no loss occurs, the estimate falls in the optimal region for 44% of the connections. The remaining estimates are overestimates, in the case when the connection experiences loss, or have an unknown impact (but, do not harm performance) in the connections that do not have dropped segments. This indicates that much of the time the *available* bandwidth is less than the raw bottleneck bandwidth that PBM measures, which accords with the finding given in [Pax97b].

3.3 Sender-Side Estimation Algorithms

The following is a description of the sender-side bandwidth estimation algorithms, and the corresponding *ssthresh* estimates, investigated in this paper. TCP's congestion control algorithms work on the principle of "self-clocking" [Jac88]. That is, data segments are injected into the network and arrive at the receiver at the rate of the bottleneck link, and consequently ACKs are generated by the receiver with spacing that reflects the rate of the bottleneck link. Therefore, sender-side estimation techniques measure the rate of the returning ACKs to make a bandwidth estimate. These algorithms assume that the spacing injected into the data stream by the network will arrive intact at the receiver and will be preserved in the returning ACK flow, which may not be true due to fluctuations on the return channel altering the ACK spacing (e.g., ACK compression [ZSC91, Mog92]). These algorithms have the advantage of being able to directly adjust the sending rate. In the case of TCP, they can directly set the *ssthresh* variable as soon as the estimate is made. However, a disadvantage of these algorithms is their reliance on the ACK stream accurately reflecting the arrival spacing of the data stream.

3.3.1 Tracking Slow Start Flights

The first technique we investigate is a TCP-specific algorithm that tracks each slow start "flight." The ACKs for a given flight are used to obtain an estimate of *ssthresh*. While this algorithm is TCP specific, the general idea of measuring the spacing introduced by the network in all segments transmitted in one RTT should be applicable to other transport protocols. We parameterize the algorithm by n, the number of ACKs used to estimate the bottleneck bandwidth. For our analysis, we used $n = 3$. Let F be the current flight size, in segments. The Tracking Slow Start Flights (TSSF) algorithm is then:

- Initialize the current segment S to the first data segment sent, and F to the initial value of *cwnd* in segments.
- For the current S and F, check whether S's ACK and the $n-1$ subsequent arriving ACKs are all within the sequence range of the flight. If so, then we use this flight to make an estimate. Otherwise, we continue to the next flight. However, if any of the ACKs arrive reordered or are duplicates, the algorithm terminates. When looking forward for the $n-1$ subsequent ACKs, the algorithm ignores any ACKs for a single segment, as they were presumably delayed.
- To find the next flight, advance S by F segments. If N_a is the number of ACKs for new data that arrive between the old value of S and its new value, then the size of the next flight is $F + N_a$ (the slow start increase).
- When we find a suitable flight, we estimate the bandwidth as the amount of data ACKed between the first and the nth ACK, divided by the time between the arrivals of these ACKs.

As the second rows of Tables 5 and 6 show, the performance of the TSSF algorithm is quite poor. The overwhelming problem with this estimator is underestimating the bandwidth, which would cause a reduction in performance.

The underestimation is caused in part by TCP's delayed acknowledgment algorithm. RFC 1122 [Bra89] encourages TCP receivers to refrain from ACKing every incoming segment, and to instead acknowledge every second incoming segment, though it also requires that the receiver wait no longer than 500 msec for a second segment to arrive before sending an ACK. Many TCP implementations use a 200 msec "heartbeat" timer for generating delayed ACKs. When the timer goes off, which could be any time between 0 and 200 msec after the last segment arrived, if the receiver is still waiting for a second segment it will generate an ACK for the single segment that has arrived. Using this mechanism can fail to preserve in the returning ACK stream the spacing imposed on the data stream by the bottleneck link. The time the receiver spends waiting on a second segment to arrive increases the time between ACKs, which is assumed by the sender to indicate the segments were further spaced out by the network, which leads to an underestimate of the bandwidth.

Furthermore, once a delayed ACK timer effect is injected into the ACK stream, the flight is effectively partitioned into two mini-flights for the duration of slow start, since data segments are sent in response to incoming ACKs. The sequence-time plot in Figure 2 illustrates this effect. In the plot, which is recorded from the sender's perspective, outgoing data segments are indicated with solid squares drawn at the upper sequence number of the segment, while incoming ACKs are drawn with hollow squares at the sequence number they acknowledge.

The first flight shown, which consists of two segments, elicits a single ACK that arrives at time $T = 2.0$. But the flight of three segments that this ACK triggers elicits two ACKs, one for two segments arriving at $T = 2.6$, but another for just one segment at time

$L = W / RTT$ bytes por segundo). Si la conexión no experimenta pérdida, L' es el ancho de banda alcanzado basado en el mayor *cwnd* observado durante la conexión.[5] Cuando $L > B$ o $L' > B$, la red está esencialmente libre de tráfico en competencia, suponiéndose que la pérdida ha sido provocada por un desbordamiento de la cola de espera de la conexión en la ruta de red. Contrariamente, si L o L' son inferiores a B, se considera que la ruta está congestionada. Finalmente, E es la estimación de ancho de banda efectuada por el algoritmo de estimación de *ssthresh* en evaluación.

Además, definimos seg(x) = (x RTT) / *tamaño de segmento* representando el tamaño de la ventana de congestión, en segmentos, necesaria para lograr un ancho de banda de x bytes/segundo, para un dado tamaño de segmento y RTT del TCP (obsérvese que, por definición, seg(x) es continua y no discreta).

3.1.1 Conexiones Con Pérdida

Dadas las definiciones anteriores y una conexión con pérdidas, evaluamos la eficiencia de un estimador determinando cuál de las siguientes seis regiones siguientes le corresponde. Obsérvese que analizamos las regiones en un orden dado, por lo cual un estimador no será considerado para ninguna de las regiones subsiguientes a aquella que le correspondió en primer lugar.

Sin estimación. El estimador no logró efectuar una estimación de *ssthresh* antes de la ocurrencia de la primera pérdida de segmento a lo largo de la traza.

Sin impacto. La estimación cumple con $E \geq \gamma L$. Ello significa que E es una sobre estimación lo suficientemente grande como para que la conexión se comporte igual, ya sea usando esta estimación, o bien sin efectuarse estimación alguna.

Cierta prevención de pérdida. Al resultar $L \leq E < \gamma L$, la estimación obtenida de *ssthresh* previene parte, pero no la totalidad, de las pérdidas asociadas a paquetes de datos. Mientras que la estimación es mayor que el punto de pérdida, reduce en N_s = seg($\gamma L - E$) segmentos el tamaño de la última emisión en arranque lento. Por lo tanto, se pueden prevenir hasta N_s caídas de segmentos.

Estado constante. Cuando se cumple que $L/2 <= E < L$, clasificamos al *ssthresh* como "estado constante" (o estable). Durante la prevención de congestión, la cual define el comportamiento de estado continuo del TCP [Jac88, MSMO97], *cwnd* disminuye a la mitad ante la detección de pérdida, aumentando linealmente hasta la ocurrencia de otra pérdida. Por lo tanto, dado el punto de pérdida L, después del segundo evento de pérdida de la conexión, se puede esperar que *cwnd* oscile entre $L/2$ y L.[6] Haciendo una estimación entre $L/2$ y L, *y* asumiendo que el punto de pérdida es estático, el estimador determina el rango dentro del cual la conexión oscilará naturalmente.

Optima. Cuando el análisis llega a este punto, sabemos que $E < L/2$, pues no se ha dado ninguna de las condiciones precedentes. Si se cumple asimismo que seg(E) ≥ seg(B) – 1, entonces la estimación de *ssthresh* reduce el requerimiento de la cola de espera según lo siguiente. Dado que E se aproxima o supera al ancho de banda de cuello de botella, pero siendo aún así inferior a $\gamma/2$, sabemos que el punto de pérdida es mayor que dicho ancho de banda, si bien la estimación de *sstresh* no es inferior al ancho de banda de cuello de botella o a este ancho de banda menos un segmento (consideramos que el ancho de banda de cuello de botella menos un segmento está dentro del rango, porque tanto el arranque lento como la prevención de congestión tomarán sólo un RTT para incrementar el *cwnd* hasta corresponder con B, prefiriendo alcanzar este punto a través de la prevención de congestión antes que con arranque lento, de manera tal que no se sobredimensione).

De este modo, asumiendo que la conexión dura lo suficiente, la cola se mantendrá completa en L. No obstante, llenaremos la cola de modo más lento y gradual que con el arranque lento. Sin embargo, cuando excedemos la cola durante una prevención de congestión, sólo es por un segmento, mientras que durante el arranque lento la sobrepasaríamos unas γ veces su capacidad.[7] Cuando una conexión cae dentro de esta región, la longitud de la cola de espera se reduce inicialmente en N_q = ($L - E$) TTT bytes. Consideramos esta región "optima", ya que reduce esperas, previene pérdidas y aun así utiliza la ruta de red al máximo.

Reducción de eficiencia. Finalmente , si no se verifica ninguna de las condiciones anteriores, entonces $E < L/2$ y $E < B$ (estos límites no son rigurosos). Por lo tanto asignamos a *ssthresh* un valor muy bajo, forzando el crecimiento de *cwnd* hasta continuar linealmente en lugar de exponencialmente. Consideramos que la estimación es especialmente mala cuando un estimador subestima min($L/2$, B) en más de la mitad del 50+% de las conexiones en las cuales se reduciría la eficiencia. En este caso, el porcentaje de conexiones registrado que experimentan una reducción de la eficiencia se marca con un "*".

[5] En sentido estricto, es el tiempo de recorrido más largo observado durante la conexión, que podría ser inferior al *cwnd* si la conexión se queda sin datos a transmitir o sobrepasa la capacidad de la ventana del receptor (32-64 KB).

[6] El tamaño de *cwnd* al detectar el primer evento de pérdida es más o menos γL. Por lo tanto, el primer desdoblamiento de *cwnd* hace que resulte aproximadamente $\gamma/2$ L. Cada evento de pérdida subsiguiente solo debería desbordar la cola de espera ligeramente, y por tanto *cwnd* se debería reducir a $L/2$.

[7] Algunas implementaciones de prevención de congestión suman a γ una constante de $1/8$ veces el tamaño de segmento por cada ACK recibido durante la prevención de congestión. Este comportamiento no habitual ha demostrado que a veces conduce a desbordar la cola de espera en más de un solo segmento cada vez que *cwnd* se acerca a L [PAD+99].

$T = 2.8$. The latter reflects a delayed ACK. The next flight of five packets then has a lull of about 200 msec in the middle of it. This lull is duly reflected in the ACKs for that flight, plus an additional delayed ACK occurs from the first sub-flight of three segments (times $T = 3.3$ through $T = 3.5$). The resulting next flight of 8 segments is further fractured, reflecting not only the lull introduced by the new delayed ACK, but also that from the original delayed ACK, and the general pattern repeats again with the next flight of 12 segments. None of the ACK flights give a good bandwidth estimate, nor is there much hope that a later flight might.

This mundane-but-very-real effect significantly complicates any TCP sender-side bandwidth estimation. While for other transport protocols the effect might be avoidable (if ACKs are not delayed), the more general observation is that sender-side estimation will significantly benefit from information regarding just when the packets it sent arrived at the receiver, rather than trying to infer this timing by assuming that the receiver sends its feedback promptly enough to generate an "echo" of the arrivals.

3.3.2 Closely-Spaced ACKs

The *ssthresh* estimation algorithms in [Hoe96] and [AD98] are based on the notion of measuring the time between "closely spaced ACKs" (CSAs). By measuring CSAs, these algorithms attempt to consider ACKs that are sent in response to closely spaced data segments, whose interarrival timing at the receiver then presumably reflects the rate at which they passed through the bottleneck link. However, neither paper defines exactly what constitutes a set of closely-space ACKs.

We explore a range of CSA definitions by varying two parameters. The first, ν, is the fraction of the RTT within which the consecutive ACKs of the closely-spaced group must arrive in order to be considered "close." We examined ν values of 0.0125, 0.025, 0.05, 0.1 and 0.2. The second parameter, n, is the number of ACKs that must be close in order to make an estimate. We examined $n = 2, 3, 4, 5$. The bandwidth estimate is made the first time n ACKs arrive (save the first) within $\nu \cdot$ RTT sec of their predecessors. This algorithm has the advantage of being easy to implement. Also, it does not depend on any of the details of TCP's congestion control algorithms, which makes the algorithm easy to use for other transport protocols. A disadvantage of the algorithm is that it is potentially highly dependent on the above two constants.

Our goal was to find a "sweet spot" in the parameter space that works well over a diverse set of network paths. Rows 3–6 of Tables 5 and 6 show the effectiveness of several of the points in the parameter space. Values of ν and n outside this range performed appreciably worse than those shown.

We chose $n = 3, \nu = 0.1$ as the sweet spot in the parameter space. However, the choice was not clear cut, as both $n = 2, \nu = 0.05$ and $n = 2, \nu = 0.1$ provide similar effectiveness. All of the parameter values shown, including the chosen sweet spot, reduce performance for a large number of connections that do not experience loss and yield no performance benefit in over 60% of the connections that did experience loss (due to an inability to form an estimate or overestimating).

3.3.3 Tracking Closely-Spaced ACKs

The *ssthresh* estimation algorithm in [AD98] assumes that the arrivals of closely-spaced ACKs are used to form tentative *ssthresh* estimates, with a final estimate being picked when these settle down into a form of consistency. We used a CSA estimator with $n = 3$ and $\nu = 0.1$ (the sweet spot above) to assess the effectiveness of their proposed approach. For their scheme, we take multiple samples and use the minimum observed sample to set *ssthresh*. We continue estimating until the point of loss, or we observe a sample within 10% of the minimum sample observed so far (in which case we are presumed to have converged). We show the effectiveness of using the "tracking closely-spaced ACKs" (TCSA) algorithm in Tables 5 and 6. As with the CSA method described above, the TCSA algorithm does not have a performance impact on the connection in over 75% of the connections with loss. Furthermore, the number of connections for which the performance would be reduced is increased by roughly a factor of 2 for both connections that experienced loss and those that did not when comparing TCSA with CSA.

Since TCSA shows an increase in the number of connections whose performance would be reduced, it clearly often estimates too low, so we devised a variant, TCSA$'$, that does not depend on the minimum observation (which is likely to be an underestimate). We compare each CSA estimate, E_i, with estimate E_{i-1} (for $i > 1$). If these two samples are within 10% of each other, then we use the average of the two bandwidth estimates to set *ssthresh*. Tables 5 and 6 show that TCSA$'$ is comparable to TCSA in most ways. The exception is that the number of underestimates that would reduce performance is decreased when using TCSA$'$, so it would be the preferred algorithm.

3.4 Receiver-Side Estimation Algorithm

The problems with sender-side estimation outlined above led to the evaluation of the following receiver-side algorithm for estimating the bandwidth. Estimating the bandwidth at the receiver removes the problems that can be introduced in the ACK spacing by delay fluctuations along the return path or due to the delayed ACK timer.

A disadvantage of this algorithm is that the receiver cannot properly control the sender's transmission rate.[8] However, the receiver could inform the sender of the bandwidth estimate using a TCP option (or some other mechanism, for a transport protocol other than TCP). For our purposes, we assume that this problem is solved, and note that alternate uses for the estimate by the receiver is an area for future work.

The receiver-side algorithm outlined below is TCP-specific. Its key requirement is that the receiver can predict which new segments will be transmitted back-to-back in response to the ACKs it sends, and thus it can know to use the arrivals of those segments as good candidates for reflecting the bottleneck bandwidth. Any transport protocol whose receiver can make such a prediction can use a related estimation technique. In particular, by using a timestamp inserted by the sender, the receiver could determine which segments were sent closely-spaced without knowledge of the specific algorithm used by the sender. This is an area for near-term future work.

For convenience, we describe the algorithm assuming that sequence numbers are in terms of segments rather than bytes. Let A_i denote the segment acknowledged by the ith ACK sent by the receiver. Let D_i denote the highest sequence number the sender can transmit after receiving the ith ACK. If we number the ACK of the initial SYN packet as 0, then $A_0 = 0$. Assuming that the initial congestion window after the arrival of ACK 0 is one segment, we have $D_0 = 1$. To accommodate initial congestion windows larger than one segment [AFP98], we increase D_0 accordingly.

The basic insight to how the algorithm works is that the receiver knows exactly which new segments the arrival of one of its ACKs at the sender will allow. These segments are presumably sent back to back, so the receiver can then form a bandwidth estimate based on their timing when they arrive at the receiver.

[8] The TCP receiver could attempt to do so by adjusting the advertised window to limit the sender to the estimated *ssthresh* value, even also increasing it linearly to reflect congestion avoidance. But when doing so, it diminishes the efficacy of the "fast recovery" algorithm [Ste97, APS99], because it will need to increase the artificially limited window, and, according to the algorithm, an ACK that does so will be ignored from the perspective of sending new data in response to receiving it.

Algoritmo	Sin Est.	Sin Imp.	Perd. Prev.	Est. Cont.	Opt.	Tot.	Efic. Red.
PBM'	23%	46%	9%	10%	11%	31%	0%
TSSF	42%	1%	1%	3%	0%	4%	52%*
$CSA^{v=0.10}_{n=3}$	62%	20%	6%	9%	2%	17%	2%
$CSA^{v=0.05}_{n=2}$	53%	37%	5%	4%	0%	9%	1%*
$CSA^{v=0.10}_{n=2}$	45%	32%	8%	10%	2%	19%	4%*
$CSA^{v=0.20}_{n=2}$	38%	24%	9%	13%	3%	25%	13%
TCSA	62%	14%	6%	11%	1%	19%	5%
TCSA'	70%	10%	6%	9%	2%	17%	2%
$Recep_{mínimo}$	11%	32%	6%	13%	4%	23%	34%*
$Recep_{promedio}$	11%	52%	10%	14%	9%	34%	3%
$Recep_{medio}$	11%	48%	10%	14%	10%	34%	7%*
$Recep_{máximo}$	11%	65%	7%	8%	8%	23%	0%*

Tabla 5: Conexiones con pérdida (8.257 trazas)

3.1.2 Conexiones sin pérdida

Las siguientes regiones usan *L'* para evaluar el impacto de la estimación de *sstresh* sobre las conexiones en el conjunto de datos que no experimentaron pérdida alguna. Cada traza se ubica en una de las cuatro regiones siguientes (nuevamente, obsérvese que analizamos las regiones en el orden dado, tal que una estimación no será considerada para ninguna de las regiones subsiguientes a la primera que le corresponda).

Sin estimación. El estimador no logró efectuar una estimación de *sstresh*.

Efecto desconocido. Cuando se cumple que $E \geq L'$, la estimación no limita la capacidad del TCP para abrir *cwnd*, ya que está por encima del cwnd máximo usado por la conexión. Puesto que no tenemos una medida adecuada del límite de la ruta de red, no se puede evaluar nada más en materia de eficiencia del estimador.

Optima. Cuando se cumple que $seg(E) \geq seg(B) - 1$, la estimación es mayor que el ancho de banda de cuello de botella, y por lo tanto no limita la eficiencia. Sin embargo, también sabemos que $E < L'$ debido a la región anterior. Por lo tanto, la estimación reduce los requerimientos iniciales de cola de espera similarmente a la región "óptima" descrita en § 3.1.1.

Reducción de eficiencia. Llegados aquí, $E < \min(L', B - seg^{-1}(1))$, indicando que la estimación no logró producir un crecimiento exponencial de ventana a *L'*, lo cual se conoce como frecuencia de emisión segura. Además, nuestra imposibilidad de alcanzar *L'* no puede excusarse facilitando un crecimiento exponencial a *cwnd* por el tiempo suficiente para llenar la tubería (*B* bytes/segundo). Nuevamente marcamos con "*" aquellas conexiones para las cuales la reducción es a menudo especialmente grande.

3.2 Algoritmo de prueba

Como se mencionó, usamos PBM como banco de pruebas en términos de estimación exacta del ancho de banda de botella. Para la estimación de *ssthresh* usamos una versión revisada del algoritmo, PBM', a fin de proveer algún tipo de *límite superior* con respecto al buen comportamiento que podemos esperar de cualquier algoritmo (no es un límite superior muy elevado, puesto que puede ser que otros algoritmos estimen el ancho de banda disponible de manera ostensiblemente más adecuada que como lo hace el PBM', pero es lo mejor que disponemos actualmente). La diferencia entre PBM' y PBM reside en que PBM' analiza la traza solamente hasta el punto de la primera pérdida, mientras que PBM analiza la traza en su totalidad. Por lo tanto, PBM' implica aplicar un algoritmo detallado, pesado pero preciso, sobre la traza en la medida en que podamos inspeccionarla antes de vernos forzados a realizar una decisión respecto al *ssthresh*. Como se puede apreciar en las Tablas 5 y 6, la estimación del PBM produce valores de *ssthresh* que raramente afectan a la eficiencia, independientemente de si la conexión experimenta pérdidas o no. Cada columna indica el porcentaje de trazas que, para cada estimador, corresponden a cada una de las regiones tratadas en § 3.1.1. La columna **Tot**. muestra el porcentaje de trazas para el cual el estimador mejoró la situación, bien fuera por alcanzar una **prevención de pérdida**, el **estado continuo** o la región **óptima**. Esta columna puede compararse directamente con la última (**eficiencia reducida**) con objeto de evaluar como un estimador dado equilibra con mejoras en algunos casos, y con degradación en otros.

Vemos que PBM' ofrece algún beneficio (estado continuo, prevención de pérdida u óptimo) sobre el 31% de las conexiones que experimentan pérdidas y, cuando éstas no ocurren, la estimación cae en la región óptima para el 44% de las conexiones. El resto de las estimaciones son sobrestimaciones, en los casos en los cuales la conexión experimenta pérdida, o tienen un impacto desconocido (pero no dañan la eficiencia) en las conexiones que no tienen segmentos caídos. Ello indica que la mayoría del tiempo el ancho de banda disponible es inferior al rudimentario ancho de banda de cuello de botella medido por PBM, lo cual está de acuerdo con las conclusiones descritas en [Pax97b].

Algoritmo	Sin Est.	Efec. Desc.	Opt.	Efic. Red.
PBM'	0%	56%	44%	0%
TSSF	13%	2%	2%	82%*
$CSA^{v=0.10}_{n=3}$	24%	42%	13%	22%
$CSA^{v=0.05}_{n=2}$	19%	59%	11%	10%
$CSA^{v=0.10}_{n=2}$	14%	48%	11%	27%
$CSA^{v=0.20}_{n=2}$	13%	34%	11%	43%*
TCSA	24%	25%	8%	44%
TCSA'	27%	33%	11%	28%
$Recep_{mínimo}$	1%	15%	2%	83%*
$Recep_{promedio}$	1%	46%	23%	31%*
$Recep_{medio}$	1%	45%	28%	26%
$Recep_{máximo}$	1%	71%	27%	1&

Tabla 6: Conexiones SIN Pérdida (2,786 trazas)

Any time the receiver sends the $j+1$st ACK, it knows that upon receipt of the ACK by the sender, the flow control window will slide $A_{j+1} - A_j$ segments, and the congestion window will increase by 1 segment, so the total number of packets that the sender can now transmit will be $A_{j+1} - A_j + 1$. Furthermore, their sequence numbers will be $D_j + 1$ through D_{j+1}, so it can precisely identify their particular future arrivals in order to form a sound measurement. Finally, we take the first K such measurements (or continue until a data segment was lost), and from them form our bandwidth estimate. For our assessment below, we used $K = 50$.

(We note that the algorithm may form poor estimates in the face of ACK loss, because it will then lose track of which data packets are sent back-to-back. We tested an oracular version of the algorithm that accounts for lost ACKs, to serve as an upper bound on the effectiveness of the algorithm. We found that the extra knowledge only slightly increases the effectiveness of the algorithm.)

This algorithm provides estimates for more connections than any of the other algorithms studied in this paper, because every ACK yields an estimate. Tables 5 and 6 show the receiver-based algorithm using four different methods for combining the K bandwidth estimates. The first "Recv" row of each table shows the effectiveness of using the minimum of the K measurements as the estimate. This yields an underestimate in a large number of the connections, decreasing performance (34% of the time when the connection experiences loss and 83% of the time when no loss is present). The next row shows that averaging the samples improves the effectiveness over using the minimum: the number of connections with reduced performance is drastically reduced when the connection experiences loss, and halved in the case when no loss occurs. However, the flip side is the number of cases when we overestimate the bandwidth increases when loss is present in the connection. Taking the median of the K samples provides similar benefits to using the average, except the number of connections experiencing reduced performance increases by a factor of 2 over averaging when loss occurs. Finally, using the maximum of the K estimates further increases the number of overestimates for connections experiencing loss. However, using the maximum also reduces the number of underestimates to nearly none, regardless of whether the connection experiences loss. Of the methods investigated here, using the maximum appears to provide the most effective *ssthresh* estimate. However, we note that alternate algorithms for combining the K estimates is an area for near-term future work.

Finally, we varied the number of bandwidth samples, K, used to obtain the average and maximum estimates reported above to determine how quickly the algorithms converge. We find that when averaging the estimates, the effectiveness increases slowly but steadily as we increase K to 50 samples. However, when taking the maximum sample as the estimate, little benefit is derived from observing more than the first 5–10 samples.

4 Conclusions and Future Work

Our assessment of different RTO estimators yielded several basic findings. The minimum value for the timer has a major impact on how well the timer performs, in terms of trading off timely response to genuine lost packets against minimizing incorrect retransmissions. For a minimum RTO of 1 sec, we also realize a considerable gain in performance when using a timer granularity of 100 msec or less, while still keeping bad timeouts below 1%. On the other hand, varying the EWMA constants has little effect on estimator performance. Also, an estimator that simply takes the first RTT measurement and computes a fixed RTO from it often does nearly as well as more adaptive estimators. Related to this finding, it makes little difference whether the estimator measures only one RTT per flight or measures an RTT for every packet. This last finding calls into question some of the assumptions in RFC 1323 [JBB92], which presumes that there is benefit in timing every packet. Given that such benefit is elusive, the other goals of [JBB92] currently accomplished using timestamp options should be revisited, to consider using a larger sequence number space instead. We finished our RTO assessment by noting that timestamps, SACKs, or even a simple timing heuristic can be used to reverse the effects of bad timeouts, making aggressive RTO algorithms more viable.

Our assessment of various bandwidth estimation schemes found that using a sender-side estimation algorithm is problematic, due to the failure of the ACK stream to preserve the spacing imposed on data segments by the network path, and we developed a receiver-side algorithm that performs considerably better. A lingering question is whether the complexity of estimating the bandwidth is worth the performance improvement, given that only about a quarter of the connections studied would benefit. However, in the context of other uses or other transports, estimating the bandwidth using the receiver-side algorithm may prove compelling.

Our study was based on data from 1995, and would benefit considerably from verification using new data and live experiments. For RTO estimation, a natural next step is to more fully explore whether combinations of the different algorithm parameters might yield a significantly better "sweet spot." Another avenue for future work is to consider a bimodal timer, with one mode based on estimating RTT for when we lack feedback from the network, and the other based on estimating the variation in the feedback interarrival process, so we can more quickly detect that the receiver feedback stream has stalled. For bandwidth estimation, an interesting next step would be to assess algorithms for using the estimates to ramp up new connections to the available bandwidth more quickly than TCP's slow start. Finally, both these estimation problems merit further study in scenarios where routers use RED queueing rather than drop-tail, as RED deployment should lead to smaller RTT variations and a source of implicit feedback for bandwidth estimation.

5 Acknowledgments

This paper significantly benefited from discussions with Sally Floyd and Reiner Ludwig. We would also like to thank the SIGCOMM reviewers, Sally Floyd, Paul Mallasch and Craig Partridge for helpful comments on the paper. Finally, the key insight that the receiver can determine which sender packets are sent back to back (§ 3.4) is due to Venkat Rangan.

References

[AD98] Mohit Aron and Peter Druschel. TCP: Improving Startup Dynamics by Adaptive Timers and Congestion Control. Technical Report TR98-318, Rice University Computer Science, 1998.

[AFP98] Mark Allman, Sally Floyd, and Craig Partridge. Increasing TCP's Initial Window, September 1998. RFC 2414.

[APS99] Mark Allman, Vern Paxson, and W. Richard Stevens. TCP Congestion Control, April 1999. RFC 2581.

[BCC+98] Robert Braden, David Clark, Jon Crowcroft, Bruce Davie, Steve Deering, Deborah Estrin, Sally Floyd, Van Jacobson, Greg Minshall, Craig Partridge, Larry Peterson, K. Ramakrishnan, S. Shenker, J. Wroclawski, and Lixia Zhang. Recommendations on Queue Management and Congestion Avoidance in the Internet, April 1998. RFC 2309.

[Bra89] Robert Braden. Requirements for Internet Hosts – Communication Layers, October 1989. RFC 1122.

3.3.2 ACKs poco espaciados

Los algoritmos de estimación de *ssthresh* desarrollados en [Hoe96] y [AD98] están basados en la idea de medir el tiempo entre ACK "poco espaciados" (CSAs, o *closely spaced ACKs*). Mediante la medición de CSAs, estos algoritmos intentan considerar los ACK que son enviados en respuesta a segmentos de datos ligeramente espaciados, cuyo tiempo entre llegadas al receptor presumiblemente refleja la frecuencia a la cual pasaron por el vínculo más estrecho. Sin embargo, ninguno de estos artículos define exactamente qué es lo que constituye un conjunto de ACK poco espaciados.

Investigamos un rango de definiciones de CSA mediante la variación de dos parámetros. El primero, *v*, es la fracción de RTT dentro de la cual los ACK consecutivos de un grupo poco espaciado deben llegar a fin de ser considerados "cercanos". Examinamos valores de *v* iguales a 0.0125, 0.025, 0.05, 0.01 y 0.2. El segundo parámetro, *n*, es el número de ACKs que deben quedar cercanos a fin de efectuar una estimación. Examinamos valores de *n* iguales a 2, 3, 4 y 5. La estimación del ancho de banda se efectúa la primera vez que llegan *n* ACKs (excepto la primera) dentro de los *v* RTT segundos de su predecesor. Este algoritmo tiene la ventaja de su facilidad de implementación. Además, no depende de ninguna característica de los algoritmos de control de congestión del TCP, lo cual hace que el algoritmo resulte fácil de usar en otros protocolos de transporte. Una desventaja de este algoritmo es que potencialmente resulta muy dependiente de las dos constantes mencionadas anteriormente.

Nuestro objetivo fue hallar un "punto de equilibrio" en los parámetros de espaciado, de modo tal que funcionen bien sobre un conjunto variado de rutas de red. Las filas 3-6 de las Tablas 5 y 6 muestran la efectividad para diversos valores de los parámetros. Los valores de *v* y *n* fuera de este rango se comportaron ostensiblemente peor que aquellos mostrados.

Elegimos $n = 3$ y $v = 0.1$ como punto de equilibrio en los parámetros de espaciado. Sin embargo, la elección no fue determinante, ya que tanto $n = 2$, $v = 0.5$ como $n = 2$, $v = 0.1$ resultaron de una efectividad similar. Todos los valores de parámetros mostrados, incluso los del punto de equilibrio elegido, disminuyen la eficiencia de un gran número de conexiones que no sufren pérdidas y no producen ninguna mejora sobre la eficacia en más del 60% de las conexiones que sí las experimentaron (debido a la incapacidad de elaborar una estimación o sobreestimación).

3.3.3 Seguimiento de ACKs poco espaciados

El algoritmo de estimación de *ssthresh* descrito en [AD98] establece que las llegadas de ACKs ligeramente espaciados son empleadas para elaborar estimaciones tentativas de *ssthresh*, tomándose una estimación final cuando éstas se normalizan en alguna forma de consistencia. Usamos un estimador CSA con $n = 3$ y $v = 0.1$ (el punto de equilibrio antes indicado) con objeto de evaluar la efectividad del mecanismo propuesto. Para este esquema, tomamos múltiples muestras y usamos la mínima observada hasta un cierto instante para establecer *ssthresh*. Continuamos estimando hasta el punto de pérdida, o bien hasta encontrar una muestra dentro del 10% de la muestra mínima observada hasta el momento (en cuyo caso suponemos convergencia). En las Tablas 5 y 6 mostramos la efectividad de usar el algoritmo de "seguimiento de ACKs poco espaciados" (TCSA, o *tracking closely-spaced ACKs*). Al igual que el método CSA descrito anteriormente, el algoritmo TCSA no tiene impacto sobre la conexión en materia de eficiencia en más del 75% de las conexiones con pérdidas. Asimismo, al comparar TCSA con CSA, la cantidad de conexiones para las cuales la eficiencia resultaría disminuida se incrementa en un factor de 2, tanto para las conexiones que experimentaron pérdida como para aquellas que no la sufrieron.

Puesto que TCSA muestra un aumento en el número de conexiones cuya eficiencia resultaría disminuida, está claro que a menudo se estima demasiado bajo. Ideamos, por tanto, una variante, TCSA', que no dependa de la observación mínima (la cual probablemente es una subestimación). Comparamos cada estimación CSA, E_i con la estimación E_{i-1} (para $i > 1$). Si estas dos muestras están dentro del 10% una de otra, para establecer el *ssthresh*, usamos el promedio de las dos estimaciones del ancho de banda. Las Tablas 5 y 6 demuestran que TCSA' resulta comparable con TCSA en la mayor parte de circunstancias. La excepción es que la mayoría de las subestimaciones que degradarían la eficiencia disminuye al usar TCSA', por lo cual resulta ser el algoritmo preferido.

3.4 Algoritmos de fuente receptora

Los inconvenientes planteados anteriormente en relación a la estimación de fuente emisora condujo a la evaluación del siguiente algoritmo de estimación del ancho de banda en fuente receptora. Estimar el ancho de banda en el receptor elimina el problema que puede surgir en la separación de los ACK debido a fluctuaciones del retardo a lo largo de la ruta de regreso o debido al temporizador del ACK retardado.

Una desventaja de este algoritmo es que el receptor no puede controlar apropiadamente la frecuencia de transmisión del emisor[8]. No obstante, el receptor podría informarle al emisor sobre el ancho de banda estimado usando una opción del TCP (o algún otro mecanismo, en el caso de un protocolo de transporte distinto al TCP). Para nuestros propósitos, asumimos resulto este problema, matizando que los usos alternativos de la estimación por el receptor serán objeto de trabajo futuro.

El algoritmo de fuente receptora descrito a continuación es específico del TCP. Su clave es que el receptor pueda predecir los nuevos segmentos a transmitir recíprocamente en respuesta a los ACK que envíe, y de esta manera saber usar las llegadas de estos segmentos como buenos candidatos para reflejar el ancho de banda de cuello de botella. Cualquier protocolo de transporte cuyo receptor sea capaz de efectuar tal predicción puede usar una técnica de estimación relacionada. En particular, usando la marca temporal insertada por el emisor, el receptor podría determinar qué segmentos fueron enviados ligeramente espaciados sin conocimiento específico del algoritmo usado por el emisor. Esto será tema de trabajo a corto plazo.

Por conveniencia, describimos este algoritmo asumiendo que los números de la secuencia se expresan en términos de segmentos, y no en bytes. Sea A_i el segmento reconocido por el i^{esimo} ACK enviado por el receptor, y D_i el número de secuencia más alto que el emisor puede transmitir después de recibir el i^{esimo} ACK. Si numeramos el ACK del paquete SYN incial como 0, entonces $A_0 = 0$. Asumiendo que la ventana de congestión inicial después de la llegada del ACK 0 es de un segmento, resulta $D_0 = 1$. Para ajustar las ventanas de congestión inicial a tamaños superiores a un segmento [AFP98], disminuimos D_0 en consecuencia.

La idea básica en el funcionamiento del algoritmo es que el receptor sabe exactamente cuáles son los nuevos segmentos de datos resultantes de la llegada de uno sus ACK al emisor. Presumiblemente,

[8] El receptor de TCP podría intentar hacer esto ajustando la ventana publicada para limitar al emisor al valor de *ssthresh* estimado, incluso también agrandándola linealmente para reflejar la prevención de congestión. Pero al hacer esto, disminuye la eficacia del algoritmo de "recuperación rápida" [Ste97, APS99], pues necesitará agrandar la ventana artificialmente limitada y, de acuerdo con el algoritmo, un ACK que haga esto será ignorado desde la perspectiva de enviar nuevos datos como respuesta a su recepción.

[DDK+90] Willibald Doeringer, Doug Dykeman, Matthias Kaiserswerth, Bernd Werner Meister, Harry Rudin, and Robin Williamson. A Survey of Light-Weight Transport Protocols for High-Speed Networks. *IEEE Transactions on Communications*, 38(11):2025–2039, November 1990.

[FF96] Kevin Fall and Sally Floyd. Simulation-based Comparisons of Tahoe, Reno, and SACK TCP. *Computer Communications Review*, 26(3), July 1996.

[FH99] Sally Floyd and Tom Henderson. The NewReno Modification to TCP's Fast Recovery Algorithm, April 1999. RFC 2582.

[FJ93] Sally Floyd and Van Jacobson. Random Early Detection Gateways for Congestion Avoidance. *IEEE/ACM Transactions on Networking*, 1(4):397–413, August 1993.

[Hoe96] Janey Hoe. Improving the Start-up Behavior of a Congestion Control Scheme for TCP. In *ACM SIGCOMM*, August 1996.

[Jac88] Van Jacobson. Congestion Avoidance and Control. In *ACM SIGCOMM*, 1988.

[Jac90] Van Jacobson. Modified TCP Congestion Avoidance Algorithm, April 1990. Email to the end2end-interest mailing list. URL: ftp://ftp.ee.lbl.gov/email/vanj.90apr30.txt.

[JBB92] Van Jacobson, Robert Braden, and David Borman. TCP Extensions for High Performance, May 1992. RFC 1323.

[JK92] Van Jacobson and Michael Karels. Congestion Avoidance and Control, 1992. ftp://ftp.ee.lbl.gov/papers/congavoid.ps.Z.

[Kes91] Srinivasan Keshav. A Control Theoretic Approach to Flow Control. In *ACM SIGCOMM*, pages 3–15, September 1991.

[KP87] Phil Karn and Craig Partridge. Improving Round-Trip Time Estimates in Reliable Transport Protocols. In *ACM SIGCOMM*, pages 2–7, August 1987.

[Lud99] Reiner Ludwig. A Case for Flow-Adaptive Wireless Links. Technical report, Ericsson Research, February 1999.

[Mil83] David Mills. Internet Delay Experiments, December 1983. RFC 889.

[MM96] Matt Mathis and Jamshid Mahdavi. Forward Acknowledgment: Refining TCP Congestion Control. In *ACM SIGCOMM*, August 1996.

[MMFR96] Matt Mathis, Jamshid Mahdavi, Sally Floyd, and Allyn Romanow. TCP Selective Acknowledgement Options, October 1996. RFC 2018.

[Mog92] Jeffrey C. Mogul. Observing TCP Dynamics in Real Networks. In *ACM SIGCOMM*, pages 305–317, 1992.

[MSMO97] Matt Mathis, Jeff Semke, Jamshid Mahdavi, and Teunis Ott. The Macroscopic Behavior of the TCP Congestion Avoidance Algorithm. *Computer Communication Review*, 27(3), July 1997.

[Nag84] John Nagle. Congestion Control in IP/TCP Internetworks, January 1984. RFC 896.

[PAD+99] Vern Paxson, Mark Allman, Scott Dawson, William Fenner, Jim Griner, Ian Heavens, Kevin Lahey, Jeff Semke, and Bernie Volz. Known TCP Implementation Problems, March 1999. RFC 2525.

[Pax97a] Vern Paxson. Automated Packet Trace Analysis of TCP Implementations. In *ACM SIGCOMM*, September 1997.

[Pax97b] Vern Paxson. End-to-End Internet Packet Dynamics. In *ACM SIGCOMM*, September 1997.

[Pax97c] Vern Paxson. *Measurements and Analysis of End-to-End Internet Dynamics*. Ph.D. thesis, University of California Berkeley, 1997.

[Pax98] Vern Paxson. On Calibrating Measurements of Packet Transit Times. In *ACM SIGMETRICS*, June 1998.

[Pos81] Jon Postel. Transmission Control Protocol, September 1981. RFC 793.

[Ste97] W. Richard Stevens. TCP Slow Start, Congestion Avoidance, Fast Retransmit, and Fast Recovery Algorithms, January 1997. RFC 2001.

[TMW97] Kevin Thompson, Gregory Miller, and Rick Wilder. Wide-Area Internet Traffic Patterns and Characteristics. *IEEE Network*, 11(6):10–23, November/December 1997.

[WS95] Gary R. Wright and W. Richard Stevens. *TCP/IP Illustrated Volume II: The Implementation*. Addison-Wesley, 1995.

[Zha86] Lixia Zhang. Why TCP Timers Don't Work Well. In *ACM SIGCOMM*, pages 397–405, August 1986.

[ZSC91] Lixia Zhang, Scott Shenker, and David Clark. Observations on the Dynamics of a Congestion Control Algorithm: The Effects of Two- Way Traffic. In *ACM SIGCOMM*, September 1991.

estos segmentos son enviados recíprocamente, de modo tal que el receptor pueda elaborar una estimación del ancho de banda basándose en la cadencia con la cual llegan a su destino.

En el instante en que el receptor envía el $j + 1$ primer ACK, sabe que tras la recepción del ACK en el emisor, la ventana de control de flujo se desplazará en $A_{j+1} - A_j$ segmentos, y la ventana de congestión crecerá en 1 segmento, de manera que el número total de paquetes que el emisor podrá ahora transmitir será $A_{j+1} - A_j + 1$. Además, sus números de secuencia serán $D_j + 1$ hasta D_{j+1}, y por lo tanto puede identificar precisamente sus llegadas futuros a fin de elaborar una medición segura. Finalmente, tomamos las primeras K de estas mediciones (o continuamos hasta la pérdida de un segmento de datos), y a partir de ellas elaboramos nuestra estimación del ancho de banda. Para la siguiente evaluación usamos $K = 50$.

(Observamos que el algoritmo puede elaborar estimaciones deficientes ante la pérdida de ACKs, pues entonces perderá el control de cuáles serán los paquetes enviados recíprocamente. Probamos una versión oracular del algoritmo que tiene en cuenta la pérdida de ACKs, con objeto de de establecer una expectativa máxima de efectividad. Comprobamos que el conocimiento adicional sólo aumenta levemente la efectividad del algoritmo).

Puesto que cada ACK genera una estimación, este algoritmo produce estimaciones para más conexiones que cualquiera de los otros estudiados en este artículo. Las Tablas 5 y 6 muestran el algoritmo de fuente receptora usando cuatro métodos diferentes de combinación de las K estimaciones de ancho de banda. La primera fila "Recv" de cada tabla muestra la efectividad del hecho de utilizar un mínimo de K medidas como estimación. Ello produce una subestimación en una gran cantidad de conexiones, disminuyendo la eficiencia (un 34% de las veces en las cuales la conexión experimenta pérdidas, y un 83% de las veces en las cuales no se presentan pérdidas). La siguiente fila demuestra que promediar las muestras mejora la efectividad respecto al uso del mínimo: el número de conexiones con eficiencia degradada se reduce drásticamente para las conexiones que experimentan pérdidas, bajando a la mitad para el caso en el que no ocurren pérdidas. Sin embargo, el lado débil es la cantidad de casos en los que sobre estimamos el aumento de ancho de banda cuando se presentan pérdidas en la conexión. Tomar la mediana de las K muestras ofrece los mismos beneficios que usar el promedio, salvo que cuando ocurren pérdidas, el número de conexiones con eficiencia disminuida aumenta por un factor de 2 sobre el correspondiente a las muestras promediadas. Finalmente, usar el máximo de las K estimaciones aumenta aun más la cantidad de sobre estimaciones para las conexiones que experimentan pérdidas. No obstante, usar el máximo también reduce el número de subestimaciones a casi cero, independientemente de que las conexiones experimenten pérdidas o no. De los métodos investigados en esta etapa, el uso del máximo parece ser el que produce la estimación de *ssthresh* más efectiva. Aun así, consideramos que es asunto de trabajo futuro a corto plazo el investigar algoritmos alternativos de combinación de las K estimaciones.

Por último, para determinar la velocidad a la cual convergen los algoritmos, variamos la cantidad K de muestras de ancho de banda usadas para obtener el promedio y el máximo de las estimaciones indicadas anteriormente. Descubrimos que al promediar las estimaciones, la efectividad aumenta lenta pero continuamente a medida que incrementamos K, hasta llegar a 50 muestras. Sin embargo, al tomar la muestra máxima como estimación, no resulta más beneficioso el hecho de observar más que las primeras 5-10 muestras.

4 Conclusiones y trabajo futuro

Nuestra estimación de diferentes estimadores de RTO redundó en varios descubrimientos importantes. El valor mínimo del temporizador tiene un gran efecto sobre el buen funcionamiento del temporizador en cuanto a equilibrar respuestas correctas con genuinas pérdidas de paquetes y minimizar las retransmisiones incorrectas. También observamos que, para un RTO mínimo de 1 seg., se obtiene una mejora importante en la eficiencia al utilizar una granularidad de reloj de 100 mseg. o inferior, manteniendo las interrupciones erróneas por debajo del 1%. Por otra parte, el variar las constantes del EWMA tiene escaso efecto sobre la eficiencia del estimador. Asimismo, un estimador que simplemente considera la primera medición de RTT y calcula a partir de la misma un RTO fijo, se comporta casi tan bien como los estimadores más adaptativos. En relación a estos hallazgos, es indiferente que el estimador mida sólo un RTT por envío, o el RTT de cada paquete. Esta última conclusión llama la atención sobre algunas de las premisas en RFC 1323 [JBB92], la cual presupone que hay un beneficio en cronometrar cada paquete. Dado que este beneficio resulta elusivo, deberían revisarse los restantes objetivos de [JBB92], actualmente alcanzados mediante el uso de las opciones de marca temporal, a fin de considerar a cambio un espaciado de números secuenciales mayor. Finalizamos nuestra evaluación de RTO observando que las marcas temporales, SACKs y aún una simple heurística, puede ser usada para invertir los efectos de las interrupciones erróneas, haciendo que los algoritmos de RTO agresivos resulten mas plausibles.

Nuestra evaluación de varios esquemas de estimación de ancho de banda nos hizo llegar a la conclusión de que usar un algoritmo de estimación en fuente emisora resulta problemático debido a la incapacidad del flujo de ACK en cuanto a preservar el espaciado impuesto por la ruta de red sobre los segmentos de datos. Desarrollamos entonces un algoritmo de estimación en fuente receptora que funciona considerablemente mejor. Dado que sólo un cuarto de las conexiones estudiadas resultarían beneficiadas, una pregunta pendiente es saber si la mejora de la eficiencia justifica la complejidad de estimar el ancho de banda. No obstante, en el contexto de otros usos o protocolos, la estimación del ancho de banda en fuente receptora puede resultar obligatorio.

Nuestro estudio se basó en datos de 1995, enriqueciéndose considerablemente con la verificación al usar nuevos datos y experimentos reales. Para la estimación del RTO, el siguiente paso natural es investigar con más profundidad si la combinación de parámetros de diferentes algoritmos puede brindar un "punto de equilibrio" significativamente mejor. Otra vía de trabajos futuros es considerar un temporizador bimodal, con un modo basado en la estimación de RTT para cuando no contamos con respuesta de la red, y el otro modo basado en estimar variaciones en el proceso de llegada de respuestas, de modo tal que podamos detectar más rápidamente la obstrucción del flujo de respuestas del receptor. Para la estimación del ancho de banda, un paso siguiente interesante sería evaluar algoritmos que usen las estimaciones para acelerar las nuevas conexiones hasta el ancho de banda disponible más rápidamente que el arranque lento del TCP. Por último, estos dos problemas de estimación merecen más investigación en medios donde los ruteadores utilicen colas RED en vez de colas invertidas, ya que la aplicación de los RED debería conducir a menores variaciones de RTT, así como a una fuente de respuesta implícita en la estimación del ancho de banda.


SIGCOMM
América Latina

3.3 Algoritmos de fuente emisora

A continuación se presenta una descripción de los algoritmos de estimación de fuente emisora y las correspondientes estimaciones de *ssthresh* investigadas en este artículo. Los algoritmos de control de congestión de TCP operan sobre el principio de auto-medición temporal [Jac88]. Esto es, los segmentos de datos son inyectados en la red, llegando al receptor a la frecuencia del vínculo más estrecho, y consecuentemente los ACK son generados por el receptor con un espaciado que refleja esta frecuencia. Por lo tanto, para efectuar una estimación del ancho de banda, las técnicas de estimación de fuente emisora miden la frecuencia de los ACK devueltos. Estos algoritmos asumen que el espaciado inyectado por la red al flujo de datos llegará intacto al receptor, preservándose en el flujo de los ACK retornantes, lo cual puede no ser cierto debido a fluctuaciones en el canal de retorno que alteran la cadencia de los ACK (por ejemplo, la compresión de ACK [ZSC91, Mog92]). Dichos algoritmos tienen la ventaja de ser capaces de ajustar directamente la frecuencia de emisión. En el caso del TCP, pueden ajustar directamente la variable *ssthresh* tan pronto como se realiza la estimación. Sin embargo, una desventaja de estos algoritmos es su dependencia respecto al flujo de ACK, mostrando exactamente el intervalo de llegada de la corriente de datos.

3.3.1 Seguimiento de envíos de arranque lento

La primera técnica que investigamos es un algoritmo específico del TCP que sigue cada "recorrido" de arranque lento. Los ACK de un cierto recorrido son utilizados para obtener una estimación de *ssthresh*. Aunque este algoritmo es específico del TCP, la idea general de medir el espaciado introducido por la red en todos los segmentos transmitidos en un RTT debería ser aplicable a otros protocolos de transporte. Parametrizamos al algoritmo mediante *n,* el número de los ACK usados para estimar el ancho de banda de cuello de botella (o estrechamiento). Para nuestro análisis usamos $n = 3$. Sea F el tamaño actual del recorrido, medido en segmentos. El algoritmo de Seguimiento de envíos de arranque lento (TSSF, o *Tracking Slow Start Flights*) consiste entonces en:

- Inicializar el segmento presente *S* como primer segmento de datos enviado, y *F* como valor inicial de *cwnd* en segmentos.
- Para los *S* y *F presentes*, verificar si tanto el ACK de *S* como de los $n - 1$ subsiguientes ACK devueltos están todos dentro del mismo rengo de secuencia que la del envío. Si es así, entonces usamos este envío para elaborar una estimación. En caso contrario, continuamos con el siguiente envío. No obstante, si alguno de los ACK llega reordenado, o es un duplicado, el algoritmo finaliza. Al recibir a los $n - 1$ ACK subsiguientes, el algoritmo ignora cualquier ACK de un solo segmento, pues presumiblemente estaba demorado.
- Para encontrar el siguiente envío, avanzar *S* en *F* segmentos. Si N_a es el número de ACK de nuevos datos que llega entre el valor anterior de S y su nuevo valor, entonces el tamaño del siguiente envío es $F + N_a$ (el incremento del arranque lento).
- Cuando encontramos un envío apropiado, estimamos el ancho de banda como la cantidad de datos con ACK entre el primer y el $n^{ésimo}$ ACK, dividido por el intervalo de tiempo entre la llegada de estos ACK.

Como lo demuestran las segundas filas de las Tablas 5 y 6, la eficiencia del algoritmo TSSF es bastante pobre. El mayor problema de este estimador reside en subestimar el ancho de banda, lo cual provoca una reducción de la eficiencia.

En parte, la subestimación está producida por el algoritmo de reconocimiento retardado del ACK. RFC 1122 [Bra89] recomienda que los receptores de TCP eviten generar un ACK por cada segmento entrante, sino uno por cada dos, aunque también exige que el receptor no espere más de 500 mseg. la llegada del segundo segmento antes de enviar el ACK. Muchas implementaciones de TCP usan un temporizador de "marcapasos" de 200 mseg. para generar ACK retardados. Cuando el temporizador finaliza, lo cual puede ocurrir en cualquier momento entre 0 y 200 mseg., después de la llegada del último segmento, si el receptor todavía esta esperando la llegada del segundo segmento, entonces genera un ACK para el único segmento recibido. La utilización de este mecanismo puede fallar en el sentido de preservar en el flujo de ACK devueltos el espaciado impuesto sobre la corriente de datos por el vínculo más estrecho. El tiempo consumido por el receptor esperando la llegada del segundo ACK aumenta el intervalo entre los ACK, lo cual es interpretado por el emisor como una indicación de que la red ha aumentado el espaciado entre los segmentos, lo cual a su vez conduce a una subestimación del ancho de banda.

Asimismo, durante la fase de arranque lento, y una vez que el efecto de temporizador de un ACK demorado es inyectado a la red, el envío se divide efectivamente en dos mini envíos, puesto que los segmentos de datos son enviados en respuesta a los ACK devueltos. El gráfico de secuencia en función de tiempo de la Figura 2 ilustra este efecto. En el gráfico, registrado desde la perspectiva del emisor, los segmentos de datos enviados están representados mediante cuadros rellenos en la posición del mayor número de secuencia del segmento, mientras que los ACK recibidos están representados mediante cuadros vacíos en la posición del número de secuencia al que responden.

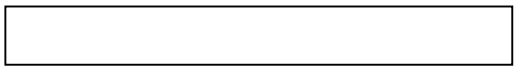

Figura 2: ACK retardados produciendo "intervalos de silencio" de temporización

El primer envío mostrado, que consiste de dos segmentos, replica un único ACK que llega al tiempo $T = 2{,}0$. Pero el envío de 3 de segmentos disparado por este ACK replica en dos ACK, uno para dos segmentos llegando en $T = 2{,}6$ y otro para solo un segmento al tiempo $T = 2{,}8$. Este último refleja el efecto del ACK demorado. El siguiente envío de 5 paquetes tiene en medio de ellos un silencio de aproximadamente 200 mseg. Este silencio se refleja puntualmente en los ACK de dicho envío más un ACK demorado, lo cual tiene lugar a partir del primer sub-envío de tres segmentos (tiempos $T = 3{,}3$ hasta $T = 3{,}5$). El siguiente envío resultante de 8 segmentos está todavía más fracturado, reflejando no solamente el silencio introducido por el último ACK demorado, sino también el producido por el ACK retardado original, repitiéndose el mismo esquema nuevamente para el siguiente envío de 12 segmentos. Ninguno de los recorridos de ACK produce una buena estimación del ancho de banda, ni tampoco hay mucha esperanza de que un envío posterior pudiera hacerlo.

Este efecto mundano, pero completamente real, complica significativamente cualquier estimación de ancho de banda en la fuente emisora del TCP. Aunque para otros protocolos de transporte el efecto podría ser evitable (si los ACK no son demorados), la observación más general es que la estimación en fuente emisora se beneficiaría significativamente con información referida tan sólo al momento en que los paquetes enviados llegaron al receptor, en vez de intentar inferir esta cadencia asumiendo que el receptor envía su respuesta de forma lo suficientemente rápida como para generar un "eco" de las llegadas.


SIGCOMM
América Latina

5 Agradecimientos

La presente publicación se ha beneficiado significativamente de conversaciones mantenidas con Sally Floyd y Reiner Ludwig. Desearíamos agradecer, asimismo, a los revisores de SIGCOMM: Sally Floyd, Paul Mallasch y Craig Partridge por sus útiles comentarios sobre este artículo. Finalmente, la idea clave de que el receptor puede determinar los paquetes del emisor enviados recíprocamente (§ 3.4) corresponde a Venkat Rangan.

Referencias

[AD98]Mohit Aron y Peter Druschel. TCP: Improving Startup Dynamics by Adaptive Timers and Congestion Control. Informe Técnico TR98-318, Universidad Rice de Ciencias de la Computación, 1998.

[AFP98] Mark Allman, Sally Floyd y Craig Partridge. Increasing TCP's Initial Window, Septiembre 1998. RFC 2414.

[APS99] Mark Allman, Vern Paxson y W. Richard Stevens. TCP Congestion Control, Abril 1999. RFC 2581.

[BCC+98] Robert Braden, David Clark, Jon Crowcroft, Bruce Davie, Steve Deering, Deborah Estrin, Sally Floyd, Van Jacobson, Greg Minshall, Craig Partridge, Larry Peterson, K. Ramakrishnan, S. Shenker, J. Wroclawski y Lixia Zhang. Recommendations on Queue Management and Congestion Avoidance in the Internet, Abril 1998. RFC 2309.

[Bra89]Robert Braden. Requirements for Internet Hosts – Communication Layers, Octubre 1989. RFC 1122..

[DDK+90] Willibald Doeringer, Doug Dykeman, Matthias Kaiserswerth, Bernd Werner Meister, Harry Rudin y Robin Williamson. A Survey of Light-Weight Transport Protocols for High-Speed Networks. *IEEE Transactions on Communication*s, 38(11):2025–2039, Noviembre 1990.

[FF96] Kevin Fall y Sally Floyd. Simulation-based Comparisons of Tahoe, Reno, and SACK TCP. *Computer Communications Revie*w, 26(3), Julio 1996.

[FH99]Sally Floyd y Tom Henderson. The NewReno Modification to TCP's Fast Recovery Algorithm, Abril 1999. RFC 2582.

[FJ93] Sally Floyd y Van Jacobson. Random Early Detection Gateways for Congestion Avoidance. *IEEE/ACM Transactions on Networkin*g, 1(4):397–413, Agosto 1993.

[Hoe96] Janey Hoe. Improving the Start-up Behavior of a Congestion Control Scheme for TCP. En *ACM SIGCOM*M, Agosto 1996.

[Jac88]Van Jacobson. Congestion Avoidance and Control. En *ACM SIGCOM*M, 1988.

[Jac90]Van Jacobson. Modified TCP Congestion Avoidance Algorithm, Abril 1990. Email a la lista end2end-interest URL: ftp://ftp.ee.lbl.gov/email/vanj.90apr30.txt.

[JBB92] Van Jacobson, Robert Braden y David Borman. TCP Extensions for High Performance, Mayo 1992. RFC 1323.

[JK92] Van Jacobson y Michael Karels. Congestion Avoidance and Control, 1992. ftp://ftp.ee.lbl.gov/papers/congavoid.ps.Z.

[Kes91] Srinivasan Keshav. A Control Theoretic Approach to Flow Control. En *ACM SIGCOM*M, páginas 3–15, Septiembre 1991.

[KP87]Phil Karn y Craig Partridge. Improving Round-Trip Time Estimates in Reliable Transport Protocols. En *ACM SIGCOM*M, páginas 2–7, Agosto 1987.

[Lud99] Reiner Ludwig. A Case for Flow-Adaptive Wireless Links. Informe Técnico, Ericsson Research, Febrero 1999.

[Mil83] David Mills. Internet Delay Experiments, Diciembre 1983. RFC 889.

[MM96] Matt Mathis y Jamshid Mahdavi. Forward Acknowledgment: Refining TCP Congestion Control. En *ACM SIGCOM*M, Agosto 1996.

[MMFR96] Matt Mathis, Jamshid Mahdavi, Sally Floyd y Allyn Romanow. TCP Selective Acknowledgement Options, Octubre 1996. RFC 2018.

[Mog92] Jeffrey C. Mogul. Observing TCP Dynamics in Real Networks. En *ACM SIGCOM*M, páginas 305–317, 1992.

[MSMO97] Matt Mathis, Jeff Semke, Jamshid Mahdavi y Teunis Ott. The Macroscopic Behavior of the TCP Congestion Avoidance Algorithm. *Computer Communication Revie*w, 27(3), Julio 1997.

[Nag84] John Nagle. Congestion Control in IP/TCP Internetworks, Enero 1984. RFC 896.

[PAD+99] Vern Paxson, Mark Allman, Scott Dawson, William Fenner, Jim Griner, Ian Heavens, Kevin Lahey, Jeff Semke y Bernie Volz. Known TCP Implementation Problems, Marzo 1999. RFC 2525.

[Pax97a] Vern Paxson. Automated Packet Trace Analysis of TCP Implementations. En *ACM SIGCOM*M, Septiembre 1997.

[Pax97b] Vern Paxson. End-to-End Internet Packet Dynamics. En *ACM SIGCOM*M, Septiembre 1997.

[Pax97c] Vern Paxson. *Measurements and Analysis of End-to-End Internet Dynamic*s. Tésis de Ph.D., Universidad de California Berkeley, 1997.

[Pax98] Vern Paxson. On Calibrating Measurements of Packet Transit Times. En *ACM SIGMETRIC*S, Junio 1998.

[Pos81] Jon Postel. Transmission Control Protocol, Septiembre 1981. RFC 793.

[Ste97]W. Richard Stevens. TCP Slow Start, Congestion Avoidance, Fast Retransmit, and Fast Recovery Algo-rithms, Enero 1997. RFC 2001.

[TMW97] Kevin Thompson, Gregory Miller y Rick Wilder. Wide-Area Internet Traffic Patterns and Characteristics. *IEEE Networ*k, 11(6):10–23, Noviembre/Diciembre 1997.

[WS95]Gary R. Wright y W. Richard Stevens. *TCP/IP Il-lustrated Volume II: The Implementatio*n. Addison-Wesley, 1995.

[Zha86] Lixia Zhang. Why TCP Timers Don't Work Well. En *ACM SIGCOM*M, páginas 397–405, Agosto 1986.

[ZSC91] Lixia Zhang, Scott Shenker y David Clark. Observations on the Dynamics of a Congestion Control Algorithm: The Effects of Two-Way Traffic. En *ACM SIGCOM*M, Septiembre 1991.