A Stateless Approach to Connection-Oriented Protocols

ALAN SHIEH, ANDREW C. MYERS, and EMIN GÜN SIRER
Cornell University

8

Traditional operating system interfaces and network protocol implementations force some system state to be kept on both sides of a connection. This state ties the connection to its endpoints, impedes transparent failover, permits denial-of-service attacks, and limits scalability. This article introduces a novel TCP-like transport protocol and a new interface to replace sockets that together enable all state to be kept on one endpoint, allowing the other endpoint, typically the server, to operate without any per-connection state. Called *Trickles*, this approach enables servers to scale well with increasing numbers of clients, consume fewer resources, and better resist denial-of-service attacks. Measurements on a full implementation in Linux indicate that Trickles achieves performance comparable to TCP/IP, interacts well with other flows, and scales well. Trickles also enables qualitatively different kinds of networked services. Services can be geographically replicated and contacted through an anycast primitive for improved availability and performance. Widely-deployed practices that currently have client-observable side effects, such as periodic server reboots, connection redirection, and failover, can be made transparent, and perform well, under Trickles. The protocol is secure against tampering and replay attacks, and the client interface is backward-compatible, requiring no changes to sockets-based client applications.

Categories and Subject Descriptors: C.2.0 [**Computer-Communication Networks**]: General—*Security and protection*; C.2.2 [**Computer-Communication Networks**]: Network Protocols—*Protocol architecture*; C.2.4 [**Computer-Communication Networks**]: Distributed Systems—*Client/server*; C.2.5 [**Computer-Communication Networks**]: Local and Wide-Area Networks—*Internet*; D.4.4 [**Operating Systems**]: Communications Management; D.4.7 [**Operating Systems**]: Organization and Design

General Terms: Design, Performance

Additional Key Words and Phrases: Stateless interfaces, stateless protocols

This work was supported by the Department of the Navy, Office of Naval Research, ONR Grant N00014-01-1-0968; and National Science Foundation grant 0430161. Andrew Myers is supported by an Alfred P. Sloan Research Fellowship. Opinions, findings, conclusions, or recommendations contained in this material are those of the authors and do not necessarily reflect the views of these sponsors. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon.
Author's address: A Shieh, Cornell University; email: ashieh@cs.cornell.edu.

ACM Reference Format:

Shieh, A., Myers, A. C., and Sirer, E. G. 2008. A stateless approach to connection-oriented protocols. ACM Trans. Comput. Syst. 26, 3, Article 8 (September 2008), 50 pages. DOI=10.1145/1394441.1394444 http://doi.acm. org/10.1145/1394441.1394444

1. INTRODUCTION

The flexibility, performance, and security of networked systems depend in large part on the placement and management of system state, including both the kernel-level and application-level state used to provide a service. A critical issue in the design of networked systems is where to locate, how to encode, and when to update system state. These three aspects of network protocol stack design have far reaching ramifications: they determine protocol functionality, dictate the structure of applications, and may enhance or limit performance.

Consider a point-to-point connection between a web client and server. The system state consists of TCP protocol parameters, such as window size, RTT estimate, and slow-start threshold, as well as application-level data, such as user ID, session ID, and authentication status. There are only three locations where state can be stored, namely, the two endpoints and the network in the middle. While the end-to-end argument provides guidance on where not to place state and implement functionality, it still leaves a considerable amount of design flexibility that has remained largely unexplored.

Traditional systems based on sockets and TCP/IP distribute hard state across both sides of a point-to-point connection. Distributed state leads to three problems. First, connection failover and recovery is difficult, nontransparent, or both, since reconstructing lost state is often nontrivial. Web server failures, for instance, can lead to user-visible connection resets. Second, dedicating resources to keeping state invites denial of service (DoS) attacks that use up these resources. Defenses against such attacks often disable useful functionality: few stacks accept piggybacked data on SYN packets, which increases the overhead of short connections, and Internet servers often do not allow long-running persistent HTTP connections, which increases the overhead of bursty accesses [Chakravorty et al. 2004]. Finally, state in protocol stacks limits scalability: servers cannot scale up to large numbers of clients because they need to commit per-client resources.

In this article, we investigate a fundamentally different way to structure a network protocol stack in which system state can be kept entirely on one side of a network connection. Our Trickles protocol stack enables encapsulated state to be pushed from the server to the client. The client then presents this state to the server when requesting service in subsequent packets, to reconstitute the server-side state. The encapsulated state thus acts as a form of *network continuation* (Figure 1). A new server-side interface to the network protocol stack, designed to replace sockets, allows network continuations to carry both kernel and application level state, and thus enables stateless network services. On the client side, a compatibility layer ensures that sockets-based clients can transparently migrate to Trickles. The use of the TCP packet format at the wire

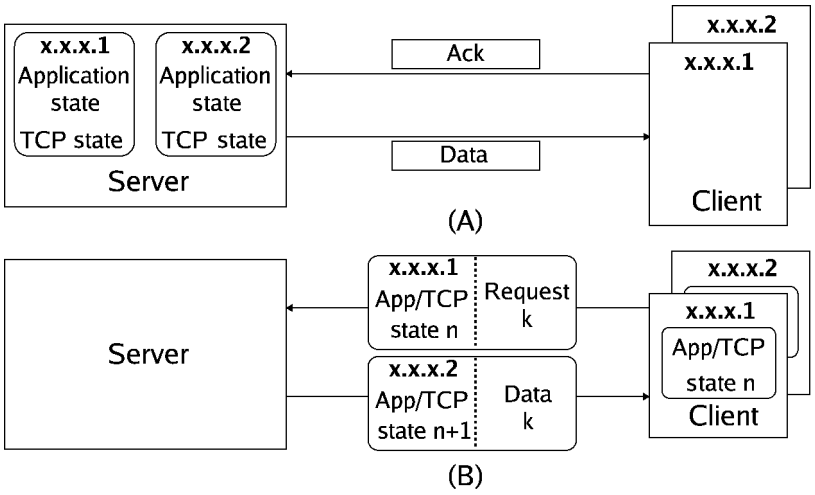

Fig. 1. TCP versus Trickles state. (A) TCP holds state at the server, even for idle connection x.x.x.2. ACKs from the client trigger server-side processing using the state associated with that connection. (B) Trickles encapsulates and ships server state to the client. The Trickles client embeds within each Trickles request any server state needed for processing the request. Like TCP ACKs, Trickles requests trigger congestion control actions.

level reduces disruption to existing network infrastructure, such as NATs and traffic shapers, and enables incremental deployment.

A stateless network protocol interface and implementation have many ramifications for service construction. Self-describing packets carrying encapsulated per-connection server state, enable services to be replicated and migrated between servers. So long as servers maintain the consistency of other state, such as the files exported by a Web server, failure recovery can be instantaneous and transparent, since redirecting a continuation-carrying Trickles packet to a live server replica will enable that server to respond to the request immediately. In the wide area, Trickles obviates a key concern about the suitability of anycast primitives [Ballani and Francis 2004] for stateful connection-oriented sessions by eliminating the need for route stability. Server replicas can thus be placed in geographically diverse locations, and satisfy client requests regardless of their past communications history. Eliminating the client-server binding obviates the need for DNS redirection and reduces the potential security vulnerabilities posed by redirectors. In wireless networks, Trickles enables connection suspension and migration [Snoeren 2002; Sultan 2004] to be performed without recourse to intermediate nodes in the network to temporarily hold state.

A stateless protocol stack can rule out many types of denial-of-service attacks on memory resources. While previous work has examined how to thwart DoS attacks targeted at specific parts of the transport protocol, such as SYN floods, Trickles provides a general approach applicable for all attacks against state residing in the transport protocol implementation and the application.

Overall, this article makes three contributions. First, it describes the design and implementation of a network protocol stack that enables all per-connection state to be safely migrated to one end of a network connection. Second, it

outlines a new TCP-like transport protocol and a new application interface that facilitates the construction of event-driven, continuation-based applications and fully stateless servers. Finally, it demonstrates through a full implementation that applications based on this infrastructure achieve performance comparable to that of TCP, interact well with other TCP-friendly flows, and scale well.

The rest of the article describes Trickles in more detail. Section 2 describes the Trickles transport protocol. Section 3 presents the new stateless server API, while Section 4 describes the behavior of the client. Section 5 presents optimizations that can significantly increase the performance of Trickles. Section 6 evaluates our Linux implementation and illustrates several applications enabled by the Trickles approach. Section 8 discusses related work, and Section 9 summarizes our contributions and their implications for server design.

2. STATELESS TRANSPORT PROTOCOL

The Trickles transport protocol provides a reliable, high-performance, TCP-friendly stream abstraction while placing per-connection state on only one side of the connection. Statelessness makes sense when connection characteristics are asymmetric; in particular, when a high-degree node in the graph of sessions (typically, a server) is connected to a large number of low-degree nodes (for example, clients). A stateless high-degree node would not have to store information about its many neighbors. For this reason we will refer to the stateless side of the connection as the *server* and the stateful side as the *client*, though this is not the only way to organize such a system.

To make congestion-control decisions, the stateless side needs information about the state of the connection, such as the current window size and prior packet loss. Because the server does not keep state about the connection, the client tracks state on the server's behalf and attaches it to requests sent to the server. The updated connection state is attached to response packets and passed to the client. This piggybacked state is called a *continuation* because it provides the necessary information for the server to later resume the processing of a data stream.

The Trickles protocol simulates the behavior of the TCP congestion control algorithm by shipping the kernel-level state, namely the TCP control block (*TCB*), to the client side in a *transport continuation*. The client ships the transport continuation back to the server in each packet, enabling the server protocol stack to regenerate state required by TCP congestion control [Allman et al. 1999]. The exchange of transport continuations between the client and server is shown in Figure 2. Trickles also enables stateful user-level server applications to migrate persistent state to the client by attaching a *user continuation* to outgoing packets. Applications are amenable to this transformation when changes to a connection's state are induced only by data carried by that connection. To minimize per-request overheads, user continuations should be small and inexpensive to update. Applications with large per-connection state requirements may need to store state on behalf of some connections rather than shipping the state in a large user continuation.

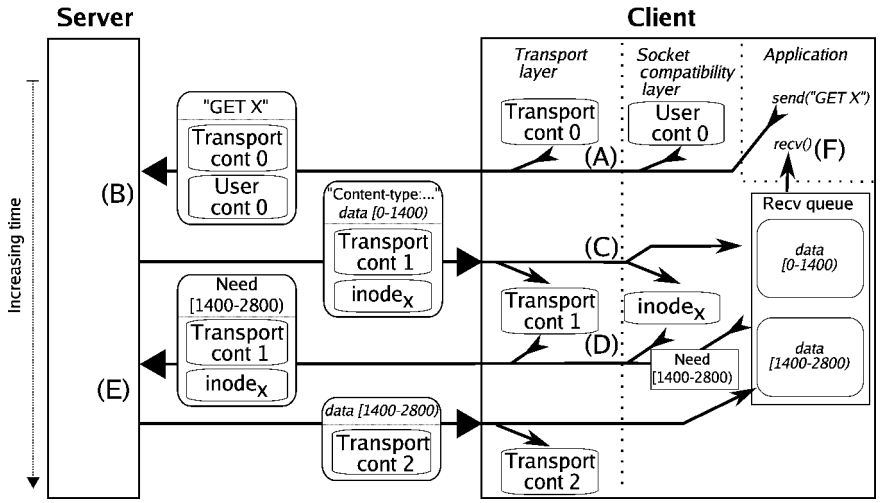

Fig. 2. Continuation processing and client-side compatibility layer. (A) The compatibility layer translates `send(''GET X'')` into a Trickles request, which uses continuations received from a previous request or from connection establishment. (B) Server receives GET request and continuations, and returns the updated continuations along with the first part of the file. The user continuation contains the inode of the requested object. (C) Client saves new continuations, and enqueues data for delivery to the application. (D) The client uses the new continuations to request the next piece of the file. (E) Server receives request for a range of data along with the previous continuations, and returns the requested data and updated transport continuation. (F) Client reads data using standard `recv()` call.

During the normal operation of the Trickles protocol, the client maintains a set of user and transport continuations. When the client is ready to transmit or request data, it generates a packet containing a transport continuation, any packet loss information not yet known to the server, a user continuation, and any user-specified data. On processing the request, the server protocol stack uses the transport continuation and loss information to compute a new transport continuation. The user data and user continuation are passed to the server application, along with the allowed response size. The user continuation and data are used by the application to compute the response.

As in TCP, the Trickles client-side networking stack is stateful, and provides reliable delivery within the transport layer for data sent to the server. In contrast, the Trickles server-side networking stack is stateless, and does not provide reliable delivery, since doing so would hold state in a send buffer until it is acknowledged. A Trickles server application must be able to reconstruct old data, either by supporting (stateless) reconstruction of previously transmitted data, or by providing its own (stateful) buffering. This design enables applications to control the amount of state devoted to each connection, and to share buffer space among connections.

2.1 Transport and User Continuations

The Trickles transport continuation encodes the part of the TCB needed to simulate the congestion control mechanisms of the TCP state machine. For

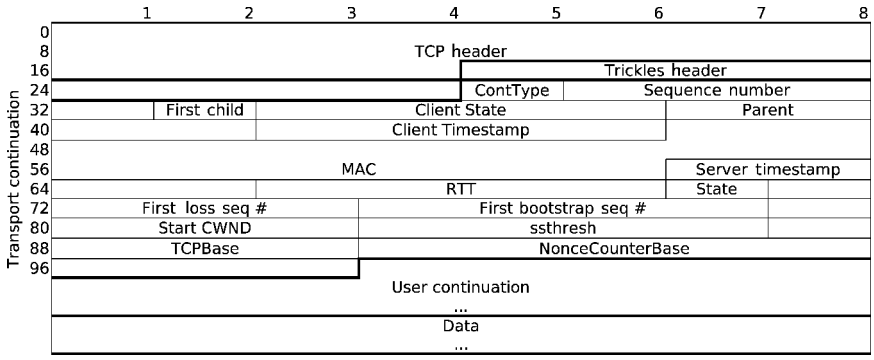

Fig. 3. Transport continuation wire format, in bytes. The transport continuation, user continuation, and data are encapsulated within a TCP packet. The transport continuation variables *cwnd*, *RTT*, *ssthresh*, and *TCPBase* encode the TCP-derived congestion control state.

example, the continuation includes the packet number, the round trip time (*RTT*), and the slow-start threshold (*ssthresh*). In addition, the client attaches a compact representation of the losses it has incurred. This information enables the server to recreate an appropriate TCB. Transport continuations (Figure 3) are $75 + 12\,m$ bytes, where m is the number of loss events being reported to the server (usually $m = 1$). Our implementation uses delayed acknowledgments, matching common practice for TCP [Allman et al. 1999].

The user continuation enables a stateless server application to resume processing in an application-dependent manner. Typically, the application will need information about what data object is being delivered to the client, along with the current position in the data stream. For a web server, this might include the URL of the requested page (or a lower-level representation such as an inode number) and a file offset. Of course, nothing prevents the server application from maintaining state where necessary.

2.2 Security

Migrating state to the client exposes the server to new attacks. It is important to prevent a malicious user or third party from tampering with server state in order to extract an unfair share of the service, to waste bandwidth, to launch a DDoS attack, or to force the server to execute an invalid state [Aura and Nikander 1997]. Such attacks might employ two mechanisms: modifying the server state—because it is no longer secured on the server, and performing replay attacks—because statelessness inherently admits replay of old packets. Furthermore, an attacker might issue requests that trigger computationally expensive code paths to exhaust server-side CPU resources.

Maintaining state integrity. Trickles protects transport continuations against tampering with a message authentication code (MAC), signed with a secret key known only to the server and its replicas. The MAC allows only the server to modify protected state, such as *RTT*, *ssthresh*, and window size. Similarly, a server application should protect its state by using a MAC over the

user continuation. Malicious changes to the transport or user continuation are detected by the server kernel or application, respectively.

Since MACs employ secret keys, their security properties are independent of the input size. To find a collision, the attacker has two options: it can perform brute-force search for the key, or search for a collision in the input space by generating and sending continuations to the server. In the general case, the keyspace search is necessary. If the space of inputs is smaller than the space of hashes, then searching the input space rather than the key space might be faster. However, this is not guaranteed to find a collision, since there are not enough unique possible inputs to force a collision.

Hiding losses [Savage et al. 1999; Ely et al. 2001] is a well known attack on TCP that can be used to gain better service or trigger a DDoS attack. Trickles avoids these attacks by attaching unique nonces to each packet. Because clients cannot predict nonce values, if a packet is lost, clients cannot substitute the nonce value for that packet.

Trickles servers associate each transmitted packet with a packet number i, and a nonce p_i. Trickles clients indicate which packets they have received, and thereby identify lost packets, by using *selective acknowledgment (SACK) proofs*, computed from the packet nonces, that securely describe the set of packets received by the client. Clients group all received nonces into g contiguous ranges by packet number. Any group $[m, n]$ is representable in $O(1)$ space by m, n, and a *range nonce*, that is the p_i's in the range combined by XOR. Thus a SACK nonce is encoded in $O(g)$ space.

Imposing additional structure on the nonces enables Trickles servers to generate per-packet nonces and to verify the receipt of arbitrarily long ranges of packets in $O(1)$ time. We add this additional structure by defining a sequence of pseudorandom numbers, $r_x = f(K, x)$, where f is a cryptographic hash function keyed by K. If $p_i = r_i \oplus r_{i+1}$, then $p_m \oplus p_{m+1} \oplus ... \oplus p_n = r_m \oplus r_{n+1}$. Thus the server can generate and verify any group of contiguous nonces compressed with XOR with two r_x computations. Trickles distinguishes retransmitted packets from the original by using a different server key K' to derive retransmitted nonces. This suffices to keep an attacker from using the nonce from the retransmitted packet to forge a SACK proof that masks the loss of the original packet.

In Appendix A, we show that the SACK proof is secure, given the assumption that the underlying sequence r_i is drawn from a uniform random distribution.

Note that this nonce mechanism protects against omission of losses but not against insertion of losses; as in TCP, a client that pretends not to receive data is self-limiting because its window size shrinks.

Protection against replay. Stateless servers are inherently vulnerable to replay attacks. Since the behavior of a stateless system is independent of history, two identical packets will elicit the same response. Therefore, protection against replay requires some state. For scalability, this extra state should be small and independent of the number of connections. Trickles protects against replay attacks using a simple hash table keyed on the transport continuation MAC.

The replay protection system is designed to prevent attackers from using replay attacks to subvert the congestion control algorithm. However, it also should not degrade well-behaved connections. During normal operation, the replay protection system guarantees *at-most-once* semantics, in which each transport continuation can only be presented once to the server. However, strictly enforcing *at-most-once* semantics complicates recovery. Suppose a client sends all of its transport continuations to the server, but packet loss prevents any new continuations from reaching the client. *At-most-once* semantics would prevent the client from making forward progress. Trickles supports *retransmit timeout* requests, exempt from the *at-most-once* rule, that provides the client with fresh continuations. A client deprived of unused continuations issues such requests to acquire the new continuations it needs to continue sending requests. The server rate limits retransmit timeout requests, and resets the congestion parameters upon receiving them, to prevent clients from abusing them to gain an unfair share of bandwidth.

Trickles provides replay protection for the transport protocol. It is the responsibility of server applications to implement stronger protection mechanisms, should they be needed.

Abstractly, this hash table records all continuations seen since the server was started, and thus prevents any continuation from ever being replayed. A naïve implementation that achieves this property would store every continuation seen by the system, and so requires state proportional to server uptime. To limit the amount of space needed, Trickles attaches timestamps to each continuation, and discards packets with older continuations. Thus, the hash tables need only record continuations generated within a finite interval of time. The timeout is configured on a per-application basis such that it exceeds the RTT of the vast majority of connections. This RTT can in turn be managed by geographically distributing servers.

Replay detection can be implemented efficiently using a Bloom filter [Bloom 1970]. Bloom filters provide space-efficient, probabilistic membership tests in constant time: storage requirements are greatly reduced in exchange for a tunable degree of lookup inaccuracy. Previously seen continuations are hashed into the Bloom filter, and each newly arrived continuation is checked against the contents of the Bloom filter. The Bloom filter will always detect when a continuation is replayed. However, it may also return a false positive, resulting in the server misclassifying an unused continuation as one that has already been used. Thus false positives increase the effective packet loss rate seen for request packets sent to the server. Trickles uses a Bloom filter of modest size that results in a low false positive rate, ensuring that unused continuations are classified correctly with high probability (1 - false positive rate). Devoting 512 KB to Bloom filters results in a 0.0091% false positive rate on a Gigabit connection. False positives require no special processing: since the client cannot distinguish a missing packet due to network loss from one due to a false positive during server-side processing, both cases will trigger the Trickles recovery protocol.

The use of the Bloom filter might cause a request to be dropped due to a false positive. If the retransmitted requests were identical to the originals, then they

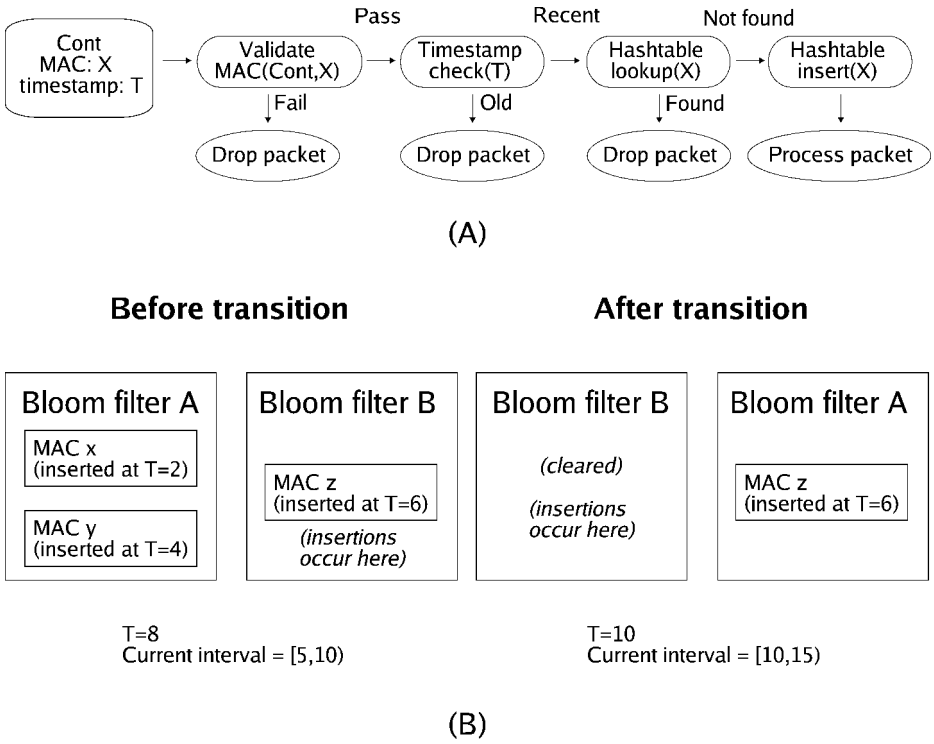

Fig. 4. (A) Bloom filter processing path. MAC filtering occurs first to prevent malicious attackers from inserting arbitrary values into Bloom filters. The timestamp check bounds the amount of state needed to store the history of previously seen continuations. (B) Bloom filter management. For illustrative purposes, the interval length is 5 seconds. At $T = 8$, the entry for continuation y is still needed, since it was encountered more recently than 5 seconds ago. Insertions always occur in the Bloom filter labeled B. Upon changing to a new epoch, Bloom filter A now contains only obsolete entries, and is emptied. A can now accept new entries, and swaps labels with B.

too would be dropped. Trickles instead tags each retransmitted request with a retransmission counter. We call all such requests with retransmission count greater than one, *recovery requests*. These tags distinguish each retransmission of a continuation in two important ways. The tag is incorporated into the continuation hash, so that the subsequent requests are unlikely to hash to the same bits, hence persistent false positives are unlikely to occur. The server uses the nonzero tags as a congestion indicator, and applies exponential backoff to the amount of data that each retransmission can cause the server to generate. This backoff limits the amount of extra service that an attacker can extract by exploiting the retransmit counter. Figure 4 summarizes the Bloom filter processing pipeline for incoming continuations.

A client that only possesses continuations older than the time horizon or has no unused continuations can explicitly send initiation requests for a retransmit timeout to restore the connection. These initiation requests are nearly identical to a normal request, except for a flag indicating that it is a retransmit timeout request. This flag instructs the server to adjust the congestion parameters as

appropriate, and to hash the request consistently on the server such that at most one recovery request can occur per connection, per horizon. This design is consistent with recommended TCP behavior, where old congestion control state is disregarded if the connection has been idle, since it is likely out of date. If an initiation request is lost on the way to the server, the client could potentially immediately send another request. However, if the response is lost on the way back to the client, then the client must wait until the next time horizon to transmit another initiation request.

Since the timestamp generation and freshness check are both performed on the server, clock synchronization between the client and server is not necessary. The growth of the hash table is capped by periodically purging and rebuilding it to capture only the packets within the time horizon T.

Two Bloom filters of identical size and using the same family of hash functions are used to simplify the periodic purge operation. Trickles divides time into intervals of length T. Let the current interval end at I. Filter A stores continuations seen between $[I-2T, I-T)$, and filter B those seen since $I-T$, that is, A captures those in the previous interval, and B those in the current interval. As the current time t crosses into a new interval, filter A is cleared and becomes the B for the new interval, while the previous B takes on the role of A. At all times, all continuations from at least $t-T$ seconds ago are hashed into one of the Bloom filters. To check whether a continuation has been seen within the past $t-T$ seconds, the server need only check the continuation against both Bloom filters.

A Trickles server can tune its Bloom filter parameters to trade off the increase in the false positive rate and horizon length against memory and computational constraints. The false positive rate p of a Bloom filter with k hash functions, m bits of storage, and containing n objects is [Fan et al. 1998]:

$$p = \left(1 - \left(1 - \frac{1}{m}\right)^{kn}\right)^k \approx (1 - e^{-kn/m})^k.$$

The insertion and lookup time increases with $O(k \log(m))$, the number of hash bits necessary to update or query the Bloom filter. The memory requirements increase linearly with m. For Trickles replay detection, n corresponds to the number of continuations that are hashed per interval. Under steady state, with each request packet resulting in one full-length data packet, this relationship is

$$n \approx T \frac{Bandwidth}{MTU}.$$

For instance, consider a server with a Gigabit connection, $MTU = 1500$, and $T = 1$. $n = 83333$ continuations are received each second in steady state. Choosing $k = 6$, $m = 2097152$ results in tables of size 256 KB each and computational cost of $6 \times 21 = 126$ hash bits. For the total space cost of 512 KB, the Bloom filters increase the effective loss rate by 0.0091%. Based on the bandwidth/loss rate dependency of $\frac{1}{\sqrt{p}}$ for a TCP-friendly flow [Floyd 1991], and assuming a median packet loss rate of 0.7% [National Internet Measurement Infrastructure 2005], this loss rate will decrease the bandwidth of connections

by less than 1%. Should the actual distribution of response packet sizes differ from this assumption, a Trickles stack can adaptively resize the Bloom filter to accommodate the increased number of outstanding continuations, or it can throttle the rate of continuations that each connection can acquire.

Transport-level replay defense can be implemented in the server kernel or in the server application. The advantage of detecting replay in the kernel is that duplicate packets can be flagged early in processing, reducing the strain on the kernel-to-application signaling mechanism. Placing the replay defense mechanism in the application is more flexible, because application-specific knowledge can be applied. In either case, Trickles is more robust against state consumption attacks than TCP.

Protection against CPU consumption attacks. The processing resources on a server are another potential target for denial of service attacks. An attacker might attempt to consume server-side CPU resources by sending expensive requests. The Trickles transport layer is resilient against such attacks. As shown in our results section, transport layer overheads at gigabit speeds are not a bottleneck. The replay detection and MAC mechanisms restrict the continuation set that an attacker can send to a server to only those that are available to legitimate users. Invalid continuations would only result in processing of additional network packets and MAC computations, after which the continuations are rejected. These overheads are present in any protocol that computes and verifies per-packet MACs.

User-level processing is another source of overhead. An attacker might craft requests for expensive objects, which a fully stateless server would recompute. Our Bloom filter replay detection mechanism can be extended to prevent an attacker from submitting a request for an object, or for a piece of an object, multiple times within a given interval. The server would insert hashes of the object identifier, salted by some connection identifier to enable different connections to request the same object without interference, into the hash table. If the link between the server and client is lossy, the server application can allow k retransmission requests per time interval by supporting k independent hash functions over the object. Replay detection can be extended with stateless cryptographic puzzles [Juels 1999] to rate-limit access to each object. Note that replay detection is essential to the success of puzzles: otherwise, an attacker can solve a puzzle once but replay the correct answer multiple times, expending CPU resources on each request.

2.3 The Trickle Abstraction

At any given time during a single Trickles connection, there are typically multiple continuations, encoding different server-side states. The existence of multiple parallel states in the stateless processing model is the main source of complexity in designing the Trickles protocol.

Figure 5 depicts the exchange of packets between the two ends of a typical TCP or Trickles connection. For simplicity, the depicted network conditions incur no packet losses and deliver packets in order; the flow does not use delayed

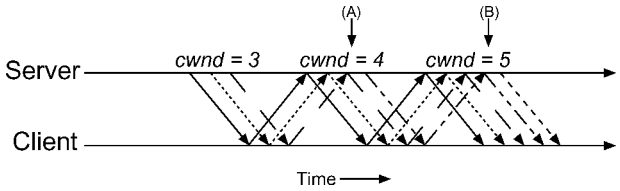

Fig. 5. A sample TCP or Trickles connection. Each line pattern corresponds to a different trickle. Initially, there are *cwnd* trickles. At points where *cwnd* increases (A, B), trickles are split.

acknowledgments. Except where the current window size (*cwnd*) increases (at times A and B), the receipt of one packet from the client enables the server to send one packet in response, which in turn triggers another packet from the client, and so on. This sequence of related packets that trace a series of state transitions forms a *trickle*.

A trickle captures the essential control and data flow properties of a stateless server. If the server does not remember state between packets, information can only flow forward along individual trickles, and so the response of the server to a packet is solely determined by the incoming trickle. A stream of packets, each associated with a unique packet number, is decomposed into multiple disjoint trickles: each packet is a member of exactly one trickle. These trickles can be thought of as independent, concurrent state machines: a server processes the continuations capturing the state of one state machine independently of the continuations from a different state machine. Two trickles can only exchange information on the client side.

In the Trickles protocol, the congestion control algorithm at the server operates on each trickle independently. These independent instances cooperate to mimic the congestion control behavior of TCP. At a given time there are *cwnd* simultaneous trickles. When a packet arrives at the server, there are three possible outcomes. In the common case, Trickles permits the server application to send one packet in response, *continuing* the current trickle. If packets were lost, the server may *terminate* the current trickle by not permitting a response packet; trickle termination reduces the current window size (*cwnd*) by 1. The server may also increase *cwnd* by *splitting* the current trickle into $k > 1$ response packets, and hence begin $k-1$ new trickles. The transport continuations of the newly-generated trickles differ in their packet numbers. By design, this initial difference in continuation state causes each trickle to trace out a disjoint subsequence of the packet number space.

Split and *terminate* change the number of trickles and hence the number of possible in-flight packets. Congestion control at the server consists of using the client-supplied SACK proof to decide whether to continue, terminate, or split the current trickle. Making Trickles match TCP's window size therefore reduces to splitting or terminating trickles whenever the TCP window size changes. When processing a given packet, Trickles simulates the behavior of TCP at the corresponding acknowledgment number based on the SACK proof, and then splits or terminates trickles to generate the same number of response packets. The subsequent sections describe how to statelessly perform these decisions to match the behavior of TCP as closely as possible.

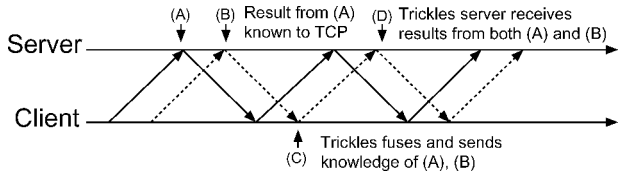

Fig. 6. Difference in state/result availability between TCP and Trickles. TCP server knows the result of processing (A) earlier than Trickles server.

2.4 Trickle Dataflow Constraints

Statelessness complicates matching TCP behavior, because it fundamentally restricts the possible data flow between the processing of different trickles. Because Trickles servers are stateless, the server forgets all the information for a trickle after processing the given packet, whereas TCP servers retain this state persistently in the TCB. Consequently, the only information available to a Trickles server when processing a request is that which was available to the client when the request was sent out. Figure 6 illustrates what happens when two packets from the same connection are received in succession. For Trickles, the state update from processing the first packet is not available when the second packet is processed at point (B), because that information was not yet available to the client when that packet was sent by the client. At the earliest, this state update can be made available at point (D) in the figure, after being processed by the client, during which the client fuses packet loss information from the two server responses and sends that information back with the second trickle. This example illustrates that server state cannot propagate directly between the processing of consecutive packets, but is available to server-side processing, a round-trip-later.

A key principle in designing Trickles is to provide each trickle with information similar to that in TCP, and then use this information to match TCP behavior. Client-side *state fusion* is used to provide each trickle with knowledge about other trickles. Since the client is stateful, it holds information about multiple trickles that the server may need in processing a subsequent trickle. The information from the other trickles is attached to this trickle. For instance, state fusion captures the dataflow of SACK proofs, which are dependent on the actions of multiple trickles.

However, the round-trip delay in state updates induced by statelessness prevents the server from knowing recent information that had not yet been available to the client. *State inference* circumvents this delay constraint. When a packet arrives at the server, the server can only know about packet losses that happened one full window earlier. It optimistically assumes that all request packets since that point have arrived successfully, and accordingly makes the decision to continue, split, or terminate. Optimism makes the common case of infrequent packet loss work well. The optimism in Trickles does not allow the server to send more packets than TCP. Under TCP, if any ACKs were reordered or lost, an ACK will be treated as a cumulative ACK, and send the packets that would have been sent had all ACK packets arrived. Generally, a cumulative ACK of a longer range clocks out more packets than that of a shorter range.

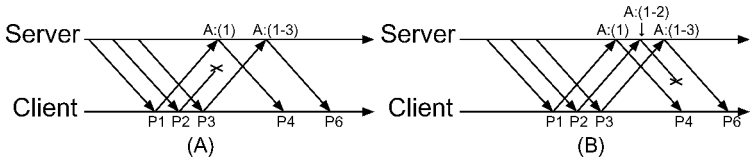

Fig. 7. Equivalence of reverse and forward path loss in Trickles. Due to dataflow constraints, the packet following a lost packet does not compensate for the loss immediately. Neither the server nor the client can distinguish between (A) and (B). The loss will be discovered through subsequent SACK proofs.

Since Trickles does not interpret the first request after one or more missing requests as a potentially long cumulative ACK, it cannot send out more packets than the corresponding TCP connection.

Concurrent trickles must respond consistently and quickly to loss events. By providing each trickle with the information needed to infer the actions of other trickles, redundant operations are avoided. Since the client-provided SACK proofs control trickle behavior, we impose an invariant on SACK proofs to allow a later trickle to infer the SACK proof of a previous trickle: given a SACK proof L, any proof L' sent subsequently contains L as a prefix. This *prefix property* allows the server to infer SACK proofs prior to L by simply computing a prefix. Conceptually, SACK proofs cover the complete loss history, starting from the beginning of the connection. As an optimization to limit the proof size, a Trickles server allows the client to omit initial portions of the SACK proof once the TCB state encoded within the transport continuation fully reflects the server's response to those losses. Trickles guarantees that this property holds after all loss events: each type of recovery action commits the updated congestion control parameters to the TCB within a finite number of steps.

It is possible that newly available information invalidates the original inference, thus requiring some recovery action. Suppose a packet is lost before it reaches the server. Then the server does not generate the corresponding response packet. This situation is indistinguishable from a loss of the response on the server to client path: in both cases, the client receives no response (Figure 7). Consequently, a recovery mechanism for response losses also suffices to recover from request packet losses, simplifying the protocol. Note, however, that Trickles is more sensitive to loss than TCP. While TCP can elide some ACK losses with implicit acknowledgments, such losses in Trickles require retransmission of the corresponding request and data.

2.5 Congestion Control Algorithm

We are now equipped to define the per-trickle congestion control algorithm. The algorithm operates in three modes that correspond to the congestion control mechanisms in TCP Reno [Allman et al. 1999]: slow start/congestion avoidance, fast recovery, and retransmit timeout. Trickles strives to emulate the congestion control behavior of TCP Reno as closely as possible by computing the target *cwnd* of TCP Reno, and performing split or terminate operations as needed to match the number of trickles with this target. Between modes, the set of valid trickles changes to reflect the increase or decrease in *cwnd*. In general, the

$$\texttt{TCPCwnd}(k) = \begin{cases} startCwnd + (k - TCPBase) & \textbf{if } k < A \\ ssthresh & \textbf{if } A \le k < A + ssthresh \\ F(k - A) & \textbf{if } A + ssthresh \le k \end{cases}$$

where

$$A = ssthresh - startCwnd + TCPBase$$

and $F(N)$ is the largest integer less than the positive value of x that is a zero of

$$\frac{(x-1)x - (ssthresh - 1)ssthresh}{2} - N$$

Fig. 8. Closed-form solution of TCP simulation.

number of trickles will decrease in a mode transition; the valid trickles in the new mode are known as *survivors*. As in most TCP implementations, Trickles acknowledges every other packet to reduce overhead. For clarity, all examples in this article use acknowledgments on every packet.

Slow start and congestion avoidance. In TCP Reno, slow start increases *cwnd* by one per packet acknowledgment, and congestion avoidance increases *cwnd* by one for every window of acknowledgments. Trickles must determine when TCP would have increased *cwnd* so that it can properly split the corresponding trickle. To do so, Trickles associates each request packet with a request number k, and uses the function $\texttt{TCPCwnd}(k)$ to map from request number k to TCP *cwnd*, specified as a number of packets. Abstractly, $\texttt{TCPCwnd}(k)$ executes a TCP state machine using acknowledgments 1 through k, and returns the resulting *cwnd*. Given the assumption that no packets are lost, and no ACK reordering occurs, the request number of a packet fully determines the congestion response of a TCP Reno server.

Upon receiving request packet k, the server performs the following trickle update:

(1) $\texttt{CwndDelta} := \texttt{TCPCwnd}(k) - \texttt{TCPCwnd}(k-1)$
(2) Generate $\texttt{CwndDelta} + 1$ responses: continue the original trickle, and split $\texttt{CwndDelta}$ times. Assuming in-order delivery, there are $\texttt{TCPCwnd}(k-1)$ packets in-flight when k is received, thus the next unused packet number is $s = k + \texttt{TCPCwnd}(k-1)$. The response trickles are assigned packet numbers starting from s.

Assuming $\texttt{TCPCwnd}(k)$ is a monotonically increasing function, which is indeed the case with TCP Reno, this algorithm maintains *cwnd* trickles per RTT, precisely matching TCP's behavior. If $\texttt{TCPCwnd}(k)$ were implemented with direct simulation as described above, it would require $O(n)$ time per packet, where n is the number of packets since connection establishment. Fortunately, for TCP Reno, a straightforward strength reduction yields the closed-form solution shown in Figure 8 and derived in Appendix B. This formula can be computed in $O(1)$ time. The extended example in Figure 9 illustrates the correspondence between Trickles and TCP under no loss.

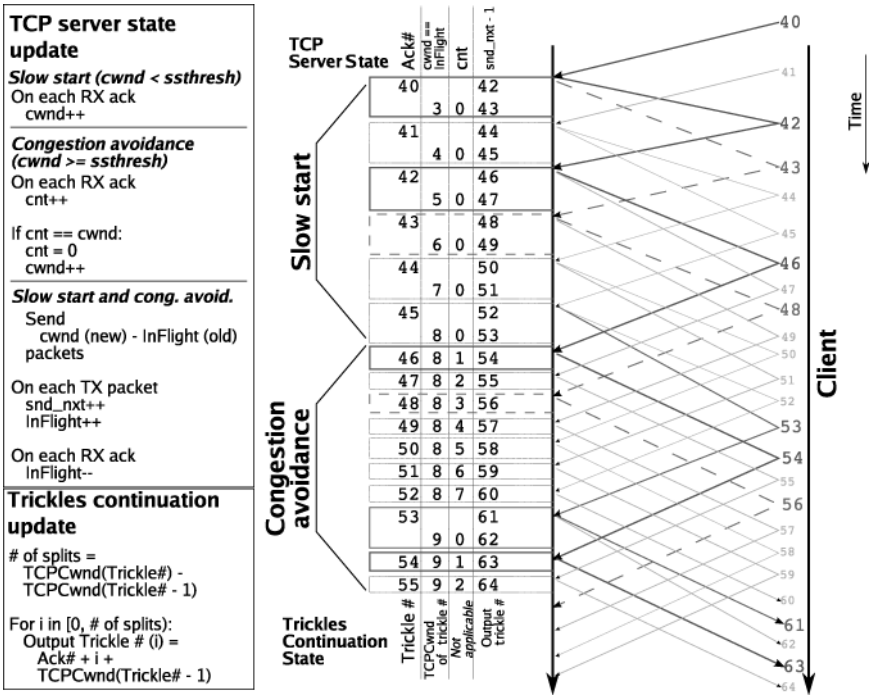

Fig. 9. Correspondence between TCP server-side state and Trickles transport continuation state, *TCPCwnd*() for Trickles initial conditions of *TCPBase* = 40, *startCwnd* = 3, *ssthresh* = 8. Arrows towards client are data packets, and arrows toward server are TCP ACKs or Trickles requests. For TCP, the state variable columns report the values of the server state after the server processes the acknowledgment and transmits the next data packet. For Trickles, the values correspond to the transport continuation variables. Note the correspondence between *cwnd* and *TCPCwnd*(), and between *snd_nxt* and output trickle #.

The TCPCwnd(k) formula is directly valid only for connections where no losses occur. A connection with losses can be partitioned at the loss positions into multiple loss-free epochs; TCPCwnd(k) is valid within each individual epoch. The free parameters in TCPCwnd(k) are used to adapt the formula for each epoch: *startCwnd* and *ssthresh* are initial conditions at the point of the loss, and *TCPBase* corresponds to the last loss location. The Trickles recovery algorithm computes new values for these parameters upon recovering from a loss.

Fast retransmit/recovery. Figure 10 compares Trickles and TCP recovery. In fast retransmit/recovery, TCP Reno uses duplicate acknowledgments to infer the position of a lost packet. The lost packet is retransmitted, the *cwnd* is halved, *ssthresh* is set to the new value of *cwnd*, and transmission of new data temporarily suppressed to decrease the number of in-flight packets to *newCwnd*. Likewise, Trickles uses its SACK proof to infer the location of lost packets, retransmits these packets, halves the *cwnd*, and terminates a sufficient number of trickles to deflate the number of in-flight packets to *newCwnd*.

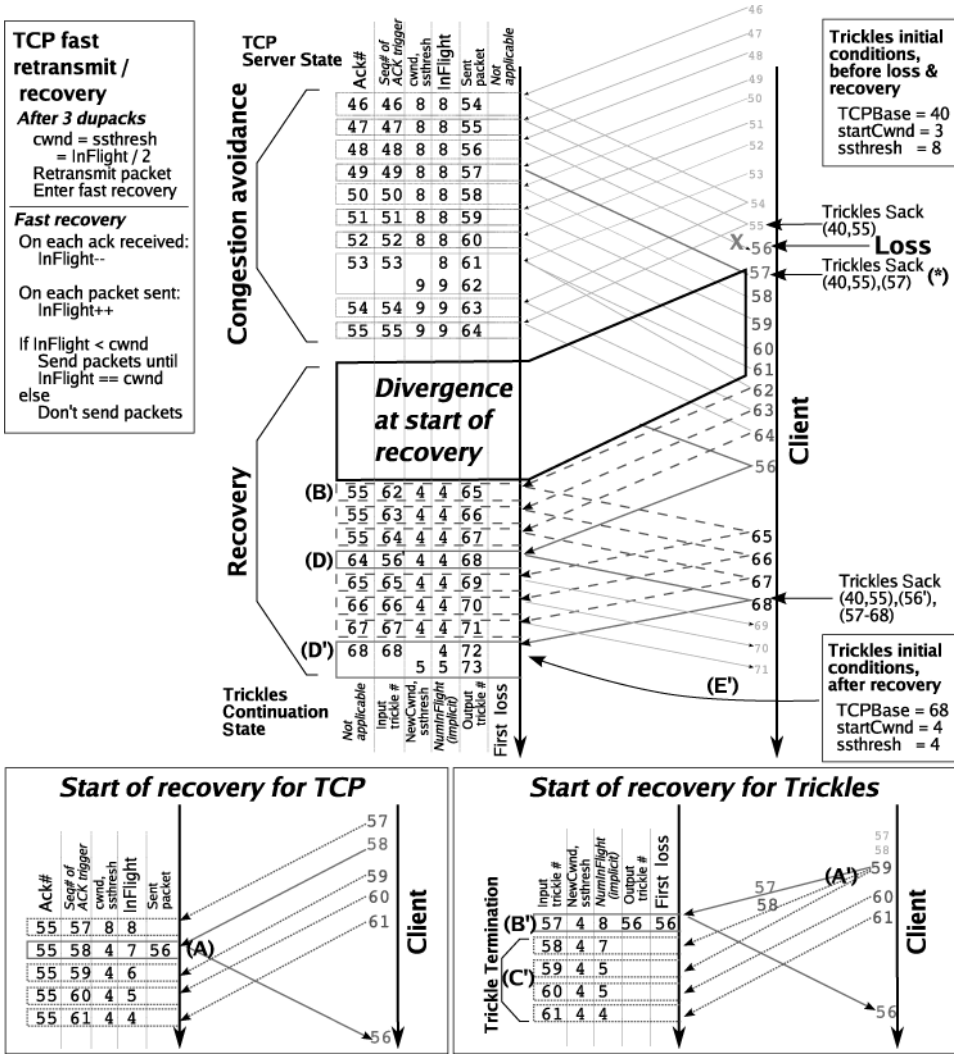

Fig. 10. *TCP recovery.* Duplicate ACKs signal a loss, and trigger fast retransmission (A). Subsequent ACKs do not trigger data transmission until number of in-flight packets drops to the new *cwnd* (B). Recovery ends when the client acknowledges all packets sent since time of loss (D). *Trickles recovery:* The client determines when loss occurs. This client deems a packet lost when three out-of-order data packets are received (A′), and begins transmitting deferred requests; the corresponding SACK reports the loss (*). The request acknowledging the packet after a loss triggers a retransmission (B′). Trickles are subsequently terminated to deflate the number of packets in flight to the new *cwnd* (C′). Once the target is reached, new survivors are generated, and data is transmitted. Recovery ends when *cwnd* survivors are generated; *cwnd* has dropped from the original value of 9 to 4 (D′). The split of trickle #68 is due to normal congestion avoidance. After recovery, the Trickles initial conditions are updated in accordance with piecewise simulation of TCP (E′). Implicit state is generated during processing, but not stored in the continuation.

Fast retransmit/recovery is entered when the SACK proof contains a loss. A successful fast retransmit/recovery phase is followed by a congestion avoidance phase. Abstractly, the Trickles protocol constructs a global plan from a given loss pattern, with each Trickle executing its part of the recovery plan. The SACK prefix property is critical to proper operation, since it allows each trickle to infer the loss pattern reported to previous trickles, and thus their input and recovery actions. A client that violates the prefix property in packets it sends to the server will cause inconsistent computations on the server side, and may receive data and transport continuations redundantly or not receive them at all.

For a request packet with packet number k during fast retransmit/recovery mode, Trickles performs the following operations:

(1)
```
firstLoss := packet number of
             first loss in input
cwndAtLoss := TCPCwnd(firstLoss - 1)
lossOffset := k - firstLoss
newCwnd := numInFlight / 2.
```
The protocol variable `firstLoss` is derived from the SACK proof. The SACK proofs for the trickle immediately after a loss, as well as all subsequent trickles before recovery, will report a gap. The SACK prefix invariant ensures that each trickle will compute consistent values for the protocol variables shown above.

(2) If k acknowledges the first packet after a run of losses, Trickles retransmits the lost packets. This is required to achieve the reliable delivery guarantees of TCP. A `burstLimit` parameter, similar to that suggested for TCP [Allman et al. 1999], limits the number of packets that may be retransmitted in this manner; losses beyond `burstLimit` are handled via a timeout and not via fast retransmit.

(3) The goal in fast retransmit is to terminate `n = cwndAtLoss - newCwnd` trickles, and generate `newCwnd` survivor trickles. We choose to terminate the first `n` trickles, and retain the last `newCwnd` trickles using the following algorithm: (a) If `cwndAtLoss - lossOffset + 1 <= newCwnd`, continue the trickle. The left hand side is an upper bound on the number of in-flight packets. Otherwise, terminate the trickle. (b) If `k` immediately follows a run of losses, generate the trickles for all missing requests that would have satisfied (a).

Test (a) deflates the number of trickles to `newCwnd`. First, a sufficient number of trickles are terminated to drop the number of trickles to `newCwnd`. Then, all subsequent trickles become *survivors* that will bootstrap the subsequent slow start/congestion avoidance phase. If losses occur while sending the surviving trickles to the client, then the number of outstanding trickles will fall below `newCwnd`. So condition (a) guarantees that the new window size will not exceed the new target, while condition (b) ensures that the new window will meet the target. Note that when the server decides to recreate multiple lost trickles per condition (b), it will not have access to corresponding user continuations for the lost packets. Consequently, the server transport layer cannot invoke the application and generate the corresponding

data payload. Instead, the server transport layer simply generates the transport continuations associated with the lost trickles and ships them to the client as a group. The client then regenerates the trickles by retransmitting these requests to the server with matching user continuations.

Following fast recovery, the simulation initial conditions are updated to reflect the conditions at the recovery packet number: `TCPBase` points to the recovery point, and `ssthresh` = `startCwnd` = `newCwnd`, reflecting the new window size.

Retransmit timeout. During a retransmit timeout, the TCP Reno sender sets $ssthresh = cwnd/2$, $cwnd = InitialCwnd$, and enters slow start. In Trickles, the client kernel is responsible for generating the timeout, since the server is stateless and cannot keep such a timer. Let *firstLoss* be the first loss seen by the client since the last retransmit timeout or successful recovery. For a retransmission timeout request, the server executes the following steps to initiate slow start:

(1)
```
a := firstLoss
ssthresh := TCPCwnd(a-1)/2
cwnd := InitialCwnd
```
(2) Split $cwnd - 1$ times to generate *cwnd* survivors. Set `TCPCwnd`(k) initial conditions to equivalent TCP post-recovery state.

Trickles prevents malicious clients from suppressing retransmission timeout events. Unless the client possesses the necessary nonces and continuations to continue processing without retransmit timeouts, it would soon exhaust the other available protocol actions, and be forced to issue a timeout request to continue processing.

2.6 Compatibility with TCP

Trickles is backward compatible with TCP in several important ways, making it possible to incrementally adopt Trickles into the existing Internet infrastructure. Compatibility at the network level, due to similar wire format, similar congestion control algorithm, and TCP-friendly behavior, ensures interoperability with routers, traffic shapers, and NAT boxes.

Some of these boxes may perform repacketization, which impacts Trickles in the same way that IP fragmentation affects IP stacks. Similar solutions apply: a Trickles implementation could perform defragmentation on the servers and clients to piece together the continuations. An increased loss rate might occur, since the server must enforce reasonable bounds on state consumption, and hence time bounds, on how long it holds continuation fragments. Our current implementation does not implement these extensions.

The client side of Trickles provides the client application with a standard Berkeley socket interface (Figure 2), so the client application need not be aware of the existence of Trickles: only the client kernel needs modification.

Trickles-enabled clients are compatible with existing TCP servers. The initial SYN packet from a Trickles client carries a TCP option to signal the ability to support Trickles. Servers that are able to support Trickles respond to the client with a Trickles response packet, and a Trickles connection proceeds. Servers that understand only TCP respond with a standard TCP SYN-ACK, causing the client to enter standard TCP mode.

One way in which the Trickles client API extends the Berkeley socket API is that clients are allowed to include data in the very first packet sent to the server. Clients may perform a `write()` call before the `connect()` call; any enqueued data accompanies the SYN packet. A TCP server that is not equipped to handle Trickles clients will simply ignore the data; a Trickles server will recognize and process it.

A Trickles server can also be compatible with standard TCP clients, by handling standard TCP requests according to the TCP protocol. Of course, the server cannot be stateless for those clients, so some servers may elect to support only Trickles.

3. TRICKLES SERVER API

The network transport protocol described in Section 2 makes it possible to maintain a reliable communications channel between a client and server with no per-connection state in the server kernel. However, the real benefit of statelessness is obtained when the entire server is stateless. The Trickles server API allows servers to offload user-level state to the client, so that the server machine maintains no state at any layer of the network stack.

3.1 The Event Queue

In the Trickles server API, the server application does not communicate using per-connection file descriptors, since these would entail per-connection state. Instead, the API exports a queue of transport-level *events* to the application. For example, client data packets and ACKs appear as events. Since Trickles is stateless, events only occur in response to client packets. Events are generated in the order that they are received. Thus, both stateful and stateless server applications must handle potential reordering of requests from a single connection.

Upon processing a client request packet, the Trickles transport layer may either terminate the trickle, or continue the associated trickle and split off zero or more trickles. If the transport generates a response, a single event is passed to the application, describing the incoming packet and instructing the application on what response trickles to generate. The event includes all the data from the request and also the user continuation from the request to the application. API state is linear in the number of unprocessed requests, which is bounded by the ingress bandwidth. The event queue eliminates a layer of multiplexing and demultiplexing found in the traditional socket API that can cause excess processing overhead [Banga et al. 1999].

To avoid copying of events, the event queue is a linked list, mapped in both the application and kernel. It is mapped read-only in the application, and is

synchronization-free in that it can be walked by a single application thread without coordination with the kernel. While processing requests, the kernel allocates all per-request data structures in the shared region.

The server application is solely responsible for controlling when response data and continuations are sent: the kernel sends packets to the client only once the server application specifies the new data. Thus the minisocket processing time is an integral part of the total processing time, and is explicitly included in the RTT measurement encoded within each transport continuation. Hence the RTT estimate in Trickles is more sensitive to server implementations than in TCP.

3.2 Minisockets

The Trickles API object that represents a remote endpoint is called a *minisocket*. Minisockets are transient descriptors that are created when an event is received, and destroyed after being processed. Like standard sockets, each minisocket is associated with one client, and can send and receive data. Operationally, a minisocket acts as a transient TCP control block, created from the transport continuation in the associated packet. Because the minisocket is associated with a specific event, the extent of each operation is more limited. Receive operations on the minisocket can only return input data from the associated event, and send operations may not send more data than is allowed by congestion control. Trickles delivers *OPEN*, *REQUEST*, and *CLOSE* events when connections are created, client packets are received, and clients disconnect, respectively.

Stateful Trickles servers periodically deallocate per-connection state. As in TCP, application-level timeouts might be used to recover the state of an inactive connection. A TCP stack generally aborts a connection and notifies the application if a given segment is retransmitted many times, since this indicates that the peer has failed silently. This has no analog in Trickles: since a failed client cannot send requests, the Trickles server will not send any data.

The server application responds with new user continuations and data. To detect new events, applications can use the standard `poll()` system call, or simply read from the event queue. When an event arrives on a socket, any application waiting on that socket will be signaled.

3.3 Minisocket Operations

The minisocket API is shown in Figure 11. Minisockets are represented by the structure `minisock *`. All minisockets share the same file descriptor (`fd`), that of their listen (server) socket. To send data with a minisocket, applications use `msk_send()`. It copies packet data to the kernel, constructs and sends Trickles response packets, then deallocates the minisocket. `msk_setucont()` allows the application to install user continuations on a per-packet basis. Trickles also provides scatter-gather, zero copy, and packet batch processing interfaces.

Allowing servers to directly manipulate the minisocket queue enables new functionality not possible with sockets. Requests sent to a node in a cluster can be redirected to a different node holding a cached copy, without breaking

```
msk_send(int fd, minisock *, char *, size_t);
msk_sendv(int fd, minisock *, tiovec *, int);
msk_sendfilev(int fd, minisock *, fiovec *, int);
msk_setucont(int fd, minisock *, int pkt,
                  char* buf, size_t);
msk_sendbulk(int fd, mskdesc *, int len);
msk_drop(int fd, minisock *);
msk_detach(int fd, minisock *);
msk_extract_events(int fd, extract_mskdesc_in *,
           int inlen, msk_collection *, int *outlen);
msk_install_events(int fd, msk_collection *, int);
msk_request(int fd, char *req, int reqlen,
           int reservelen);
```

Fig. 11. The minisocket API.

the connection. During a denial of service attack, a server may elect to ignore events altogether. The event management interface enables such manipulations of the event queue. While these capabilities are similar to those proposed in Mogul et al. [2004] for TCP, Trickles can redistribute events at a packet-level granularity.

Because the queue is mapped read-only to the user application, the user application cannot perform such operations by itself, so the API provides queue manipulation downcalls. The `detach()` call removes the specified event from the queue, but does not deallocate it, so the corresponding minisocket is still valid for all operations. The `drop()` operation removes and deallocates a minisocket.

The `msk_extractEvents()` and `msk_insertEvents()` operations manipulate the event queue to extract or insert minisockets, respectively. The extracted minisockets are protected against tampering by MACs. Extracted minisockets can be migrated safely to other sockets, including those on other machines.

4. CLIENT-SIDE PROCESSING

The structure of Trickles allows client kernels to use a straightforward algorithm to maintain the transport protocol. The client kernel generates requests using the transport continuations received from the server, while ensuring that the prefix property holds on the sequence of SACK proofs reported to the server. Should the protocol stall, the client times out and requests a retransmission and slow start.

In addition to maintaining the transport protocol, a client kernel manages user continuations, storing new continuations and attaching them to requests as appropriate. For instance, the client must provide all continuations needed to generate a particular data request.

4.1 Standardized User Continuations

To facilitate client-side management of continuations, and to simplify server programming, Trickles defines standard user continuation formats understood by servers and clients. These formats encode the mapping between continuations and data requests, and provide a standard mechanism for bootstrapping and generating new continuations.

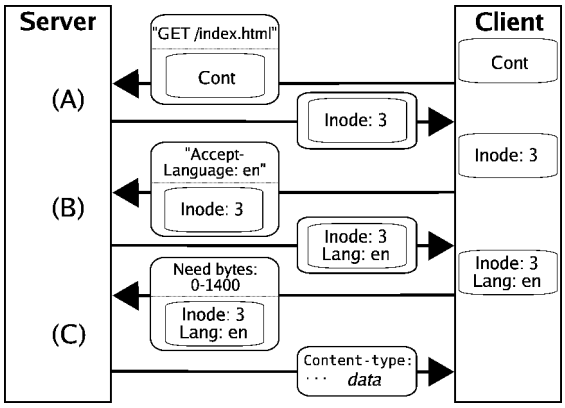

Fig. 12. Input and output continuations. Initially, the client has a continuation describing the server state after processing the preceding object. (A) Server receives first part of HTTP request and previous continuation, and returns a summary of this in an input continuation. (B) Server receives the second part of the HTTP request and input continuation. This part completes the request, and the server returns an output continuation describing the object. (C) Client uses this output continuation to request the data of the object.

Two kinds of continuations can be communicated between the client and server: *output* continuations that the server application uses to resume generating output to the client at the correct point in the server's output stream, and *input* continuations that the server application uses to help it resume correctly accepting client input. Having separate continuations allows the server to decouple input and output processing. Figure 12 contains a typical exchange of input and output continuations.

4.2 Input Continuations

When a client sends data to the server, it accompanies the data with an appropriate input continuation, except for the very first packet when no input continuation is needed. For single-packet client requests, an input continuation is not needed. For requests that span multiple packets, an input continuation contains a digest of the data seen thus far. Of course, if the server needs lengthy input from the client yet cannot encode it compactly into an input continuation, the server application will not be able to remain stateless.

If, after receiving a packet from the client, the server application is unable to generate response packets, it sends an updated input continuation back to the client kernel, which will respond with more client data accompanied by the input continuation. The server need not consume all of the client data; the returned input continuation indicates how much input was consumed, allowing the client's transmit queue to be advanced correspondingly. The capability to not read all client data is important because the server may not be able to compactly encode arbitrarily truncated client packets in an input continuation. In degenerate cases, repeated truncation of a byte stream might result in multiple transmissions of the same packet, reducing the efficiency of client-to-server transfers.

One way to improve performance is to modify the client to partition requests that are difficult to parse into smaller, more readily parsable pieces. Many Web applications already change the client to improve performance. Such applications typically ship JavaScript code to execute on the client [Crane et al. 2005]. Rather than fetch a large, complex webpage anew on each user action, this client-side code converts data returned from the Web server into locally-computed updates to the Web page, reducing the size of the network transfer and improving response time.

Similarly, a Trickles webserver could send JavaScript code to the client to simplify both parsing and result generation. The code sends requests that are efficient to parse, and the server-side code sends responses in a format that is convenient to generate statelessly. The client-side code uses these responses to update the web page in an application-specific manner. This optimization requires modifications to the JavaScript implementation so that the framing of Trickles requests is properly retained from the JavaScript code down to the network stack.

Servers might also use a hybrid approach to process difficult to parse input. If the server temporarily enters a state during parsing that is expensive to transmit or reconstitute, then the server might elect to retain state until the session proceeds to a point where stateless operation once again becomes efficient.

4.3 Output Continuations

When the server has received sufficient client data to begin processing a request, it provides the client with an output continuation for the response. The client can then use the output continuation to request the response data. For a Web server, the output continuation might contain an identifier for the data object being delivered, along with an offset into that data object.

In general, the client kernel will have a number of output continuations available that have arrived in various packets from the server. Client requests include the requested ranges of data, along with the corresponding output continuations. To allow the client to select the correct output continuation, an output continuation includes, in addition to opaque application-defined data, two standard fields, `validStart` and `validEnd`, indicating the range of bytes for which the output continuation can be used to generate data.

The congestion control algorithm restricts the amount of data that may be returned for each request. Generally, this amount is smaller than the total size of data covered by a single output continuation. Thus the client will need to split the range between `validStart` and `validEnd` into multiple requests. To compute the proper byte range size for a given request, the client simulates the server's congestion control action for a given transport continuation and SACK proof.

5. OPTIMIZATIONS

The preceding sections described the operation of the basic Trickles protocol. The performance of the basic protocol is improved significantly by three optimizations.

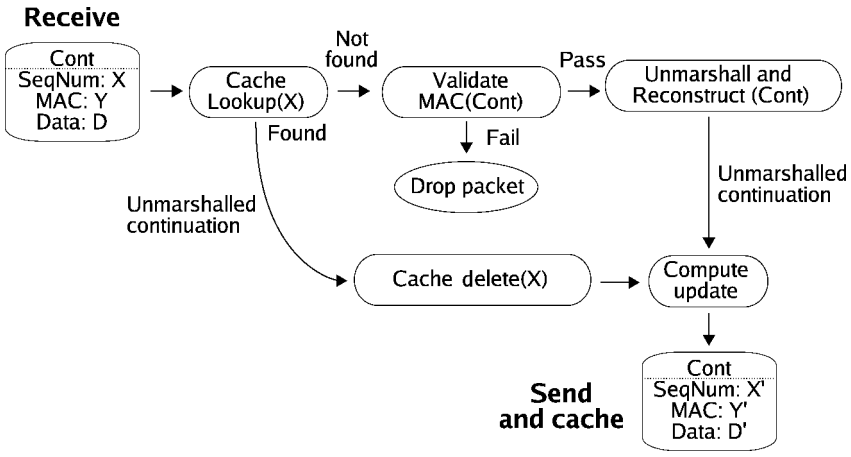

Fig. 13. Socket caching processing path. A hit in the continuation cache provides the server with a continuation state that is known to be good and already in a usable format, obviating the need for MAC validation and continuation update. The new continuation is added to the cache so that it may be available in subsequent requests.

5.1 Socket Caching

While the basic Trickles protocol is designed to be entirely stateless, and thereby consume little memory, it can be easily extended to take advantage of server memory when available. In particular, the server host need not discard minisockets and reconstitute the server-side TCB from scratch based on the client continuation. Instead, it can keep minisockets for frequently used connections in a server-side cache, and match incoming packets to this pool via a hash table. A cache hit will obviate the need to reconstruct the server-side state or to validate the MAC hash on the client-supplied continuation (Figure 13). When pressed for memory, entries in the minisocket cache can simply be dropped, as minisockets can be recreated at any time. Fundamentally, the cache acts as soft state that enables the server to operate in a stateful manner whenever resources permit, reducing the processing burden, while the underlying stateless protocol provides a safety net in case the state needs to be reconstructed from scratch.

5.2 Parallel Requests and Sparse Sequence Numbers

The concurrent nature of Trickles enables a second optimization for parallel downloads. Standard TCP operates serially, transmitting streams mostly in-order, and immediately filling any gaps stemming from losses. However, many applications, including web browsers, need to download multiple files concurrently. With standard TCP, such concurrent transactions either require multiple connections, leading to well-documented inefficiencies [Krishnamurthy et al. 1999], or complex application-level protocols, such as HTTP 1.1 [Fielding et al. 1999], for framing. Opening multiple connections also changes the congestion control behavior: though multiple connections result in higher throughput, this comes at the expense of applications that use only a single

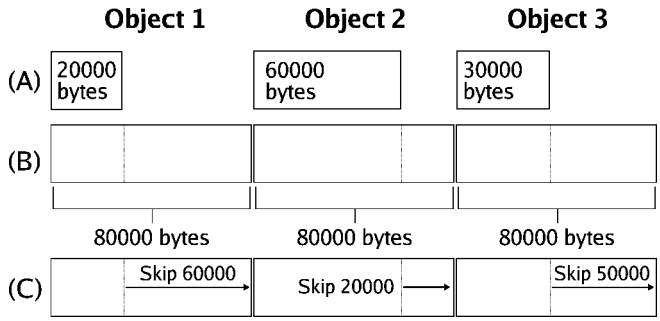

Fig. 14. Preallocating sequence number space for objects of unknown length. (A) Objects 1–3 have variable lengths, initially unknown to either the server or the client. (B) Each object is assigned 80,000 bytes in the sequence number space. (C) Once the length of the object is known, the server informs the client of the actual length of the object by specifying how much of this reserved space to skip over.

connection [Hacker et al. 2002]. By contrast, trickles are inherently concurrent. Concurrency can improve the performance of both fetching from and sending data to the server.

The Trickles protocol allows a client to concurrently request multiple objects on a single connection. Each object is assigned a different portion of the sequence number space (Figure 14). In some cases, the sizes of the objects may not be known in advance. Trickles could extend the sequence number space with an additional object identifier, or conservatively dedicate large regions of the sequence number space to each object. Any unused sequence number space is collapsed using the SKIP mechanism. The server sends a SKIP notification to indicate that the object ended before the end of its assigned range. A client receiving a SKIP logically elides the remainder of the object region, without reserving physical buffer space, passing it to applications, or waiting for additional packets from the server. This sparse allocation scheme is simpler to implement, at the expense of consuming the sequence number space at a faster rate. The response packets from the server, which will carry data belonging to different objects distributed through the sequence number space, will be subject to a single TCP-friendly flow equation, acting in effect like a single, HTTP/1.1-like flow. This higher-level optimization exploits the inherent low-level parallelism between Trickles, to multiplex logically separate transmissions on a given connection, while subjecting them to the same flow equation.

Occasionally, an object may exceed the preallocated space. To recover from this condition, Trickles allocates additional sequence number space to complete the object. The remainder of the object is treated as if it were a request for a new object: the server informs the client of this exceptional condition, and provides a special input continuation. Using this input continuation, the client requests an output continuation corresponding to the rest of the object.

Trickles clients can also send multiple streams of data to the server using the same connection. A stateless server is oblivious to the number of different input sequences on a connection. By performing multiple server input operations in

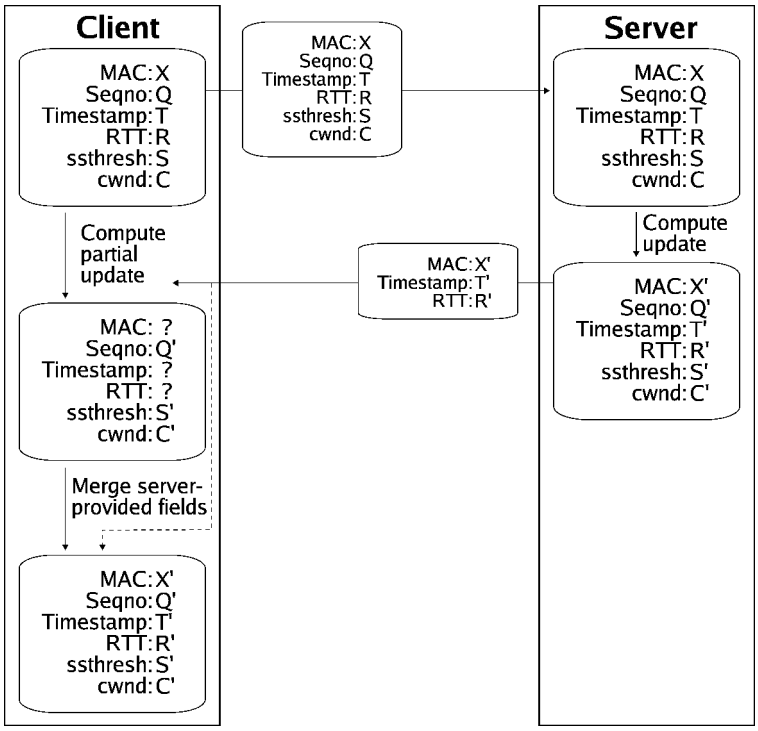

Fig. 15. Delta encoding processing path. The client computes most of the continuation update, and the server only sends those fields that the client cannot compute.

parallel, a client can reduce the total latency of a sequence of such operations. For instance, it can send multiple fetch requests in parallel, without incurring the RTT-dependent overhead of processing each input sequentially.

5.3 Delta Encoding

While continuations add extra space overhead to each packet, predictive header compression and delta encoding can be used to drastically reduce the size of the continuations transmitted by the server. Since the Trickles client implementation simulates the congestion control algorithm used by the server, it can predict the server's response. Consequently, the server need only transmit those fields in the transport continuation that the client mispredicts (e.g., a change due to an unanticipated loss), or cannot generate (e.g., timestamps). Of course, the server MAC still needs to be computed and transmitted on every continuation, since the client cannot compute the secure server hash (Figure 15). Presenting a delta to the server is not by itself sufficient to recreate the continuation, since the server has discarded the original packet. In principle, the client sends to the server the original continuation, the sequence of deltas, and the MACs for each continuation and delta. In practice, we employ an optimization that eliminates the overhead of sending and checking these deltas and MACs. The server MAC is computed on the full continuation, not the delta values. This MAC is

sent to the client along with the delta. The client updates its local copy of the continuation using its local predictions and this delta, and sends only the updated continuation along with the new MAC. Thus the server-side processing for verifying a delta-encoded continuation is identical to the non-delta-encoded case: the server computes the MAC over the continuation that had been reconstructed by the client from predictions and delta-values, and compares this value to the MAC sent from the client.

6. EVALUATION

In this section, we evaluate the quantitative performance of Trickles through microbenchmarks, and show that it scales well with the number of clients, performs well compared to TCP, consumes few resources, and interacts well with other TCP flows. We also illustrate, through macrobenchmarks, the types of new services that the Trickles approach enables.

We have implemented the Trickles protocol stack in the Linux 2.4.26 kernel. Our Linux protocol stack implements the full transport protocol, the interface, and the SKIP and parallel request mechanisms described earlier. The implementation consists of 15,000 total lines of code, structured as a loadable kernel module, with minimal hooks added to the base kernel. We use AES [Daemen and Rijmen 1999] for the keyed cryptographic hash function. All results include at least six data points; error bars indicate the 95% confidence interval. Replay detection is implemented and has negligible impact on performance. Thus it is not used during most experiments.

6.1 Microbenchmarks

Unless otherwise stated, all microbenchmarks in this section were performed on an isolated Gigabit Ethernet using 1.7 GHz Pentium 4 processors, with 512 MB RAM, and Intel e1000 gigabit network cards. To test the network layer in isolation, we served all content from memory rather than disk.

Scalability. We compared the scalability and throughput of Trickles and TCP using a topology with a single server node and two client nodes, with a single 100 Mb/s bottleneck link connecting the server and the clients. The bottleneck consisted of a single Linux machine with a 1 Gb/s NIC facing the server and a single 100 Mb/s NIC facing the clients. Varying numbers of simultaneous client instances, distributed across two machines, repeatedly fetched a 500 KB file from the server. A fresh connection was established for each request.

Trickles achieves throughput that is within 10% of TCP. As expected, Trickles consistently achieves better memory utilization than TCP (Figure 16). TCP memory utilization increases linearly with the number of clients, while statelessness enables Trickles to uses a constant, small amount of memory. TCP consumes memory separately for each connection to buffer outgoing data.

At 5000 clients, the Linux TCP stack begins moderating its memory usage, as its global memory utilization has reached the default limit. This is reflected in a change in slope of the TCP line, corresponding to the amount of memory

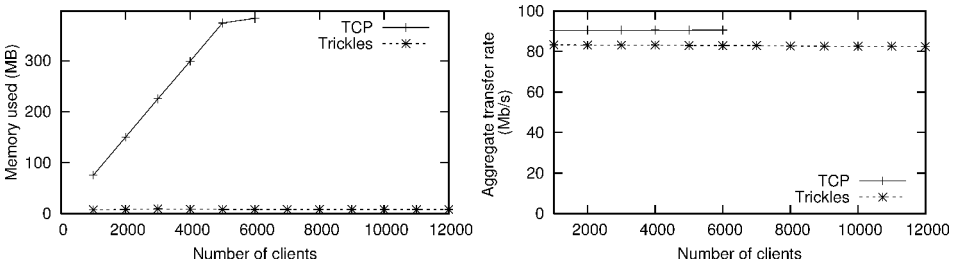

Fig. 16. Server-side memory utilization and aggregate throughput. Memory utilization includes socket structures, socket buffers, and shared event queue. TCP fails for tests with more than 6000 clients due to memory exhaustion.

needed to provide each new connection with a socket structure and minimal length transmit buffer. Beyond 6000 clients, TCP exhausts memory, forcing the kernel to kill the server process. In contrast, the Trickles kernel does not retain outgoing data, and recomputes lost packets as necessary from the original source. Consequently, it does not suffer from a memory bottleneck.

Reduced memory consumption in the network layer can improve system performance for a variety of applications. In Web server installations, persistent, pipelined HTTP connections are known to reduce download latencies. However, many Web sites disable persistent connections because increased connection duration can increase the number of simultaneous connections. Trickles can achieve the benefits of persistent connections without suffering from scalability problems. The low memory requirement of Trickles also enables small devices with restricted amounts of memory to support large numbers of connections. In such contexts, CPU power may also be a limited resource. Cryptographic operations such as nonce and MAC generation may be omitted to significantly reduce computational overhead if such devices are connected to a secure network. Finally, Trickles' smaller memory footprint provides more space for caching, benefiting all connections.

With Trickles, a client fetching small objects will achieve significant performance improvements because of the reduction in the number of control packets (Figure 17). Trickles requires fewer packets for connection setup than TCP. Trickles processes data piggybacked in SYN packets into output continuations without holding state, and can send an immediate response. In contrast, TCP must save or reject SYN data; because holding state increases vulnerability to SYN flooding, most TCP stacks reject SYN data. Care should be taken when piggybacking is enabled to avoid creating a DoS amplification vulnerability: if an attacker can spoof the source IP address of a victim, and if the server sends more data than is received, then the attacker can use the server to amplify the amount of traffic that it can send to the victim. The piggyback optimization can be used when spoofing can be ruled out, for instance, in a trusted private network or if egress filtering is widely deployed. Alternatively, the server could enforce reverse routability to a client by negotiating a session nonce with the client, valid for multiple requests. Subsequent Trickles SYN packets would include this nonce, which would not be available to an attacker. This technique

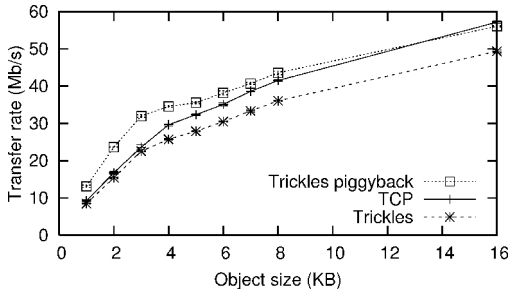

Fig. 17. Trickles and TCP throughput for a single, isolated client at various object sizes. Trickles can piggyback data on SYN packets while retaining good resilience against DoS attacks.

	Number of clients		
	1	2	3
TCP	83.1	91.0	92.9
Trickles	68.3	76.9	78.5

Fig. 18. Aggregate transfer rate, in Mb/s, with SpecWeb'99 request distribution, piggybacking disabled. At as few as three simultaneous clients, Trickles aggregate transfer rate is within 5% of the large file performance of Figure 16 (right hand side).

is orthogonal to the Trickles protocol, and can be applied to protocols such as T/TCP.

Unlike TCP, Trickles does not require FIN packets to clean up server-side transport layer state, as all state induced by this connection will be purged automatically during the recycling of the replay-prevention Bloom filters. Trickles clients still send FIN packets as a hint for the server to deallocate server-side application-level state; these FIN packets generate *CLOSE* events that are passed to the server application. The combination of SYN data and lower connection overhead improves small file transfer throughput for Trickles, with a corresponding improvement in transfer latency.

Trickles is particularly well-suited for Web servers, since they typically serve a large number of clients in a stateless fashion. To evaluate the suitability of Trickles in this environment, we subjected our implementation to a workload with the request size distribution based on SpecWeb'99 [Standard Performance Evaluation Corporation 1999]. To isolate the effects of the network stack, we served all files from memory. Piggybacking requests on SYN packets is disabled.

Each client issues a single request at a time, and establishes a new connection for each request. To establish a lower bound on performance, we employed a small number of simultaneous clients, as per-request latency due to protocol overheads are more apparent at low degrees of parallelism. Trickles performance tracks the trend of TCP performance (Figure 18), and at three simultaneous connections is already within 5% of its throughput in lower overhead experiments, such as those with large objects or a high degree of parallelism. Thus we expect Trickles to perform well on Web workloads that contain small objects.

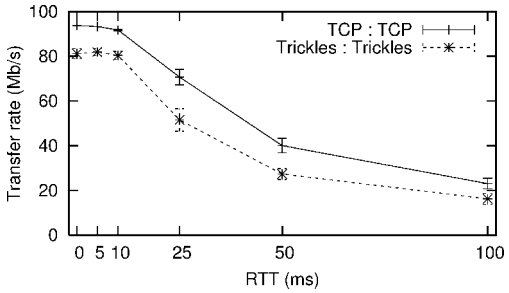

Fig. 19. Single client throughput versus RTT. Trickles achieves similar throughput to TCP under all RTTs.

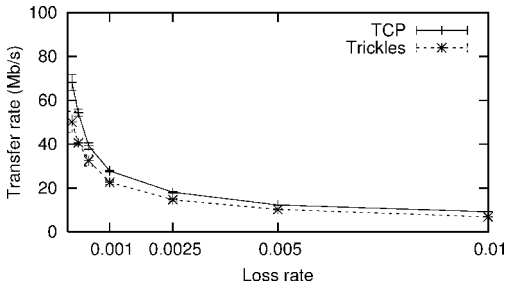

Fig. 20. Single client throughput versus packet loss rate. RTT is set to 10 ms to emphasize effects of loss. Trickles achieves similar throughput to TCP under all loss rates.

Sensitivity to RTT, packet loss, and reordering. We next consider the performance of a single Trickles client under different delay, packet loss, and reordering scenarios. In all of these experiments, the client transfers a sequence of 100 MB objects from the servers over a 100 Mb/s link.

Trickles closely matches TCP under different delay (Figure 19) and forward (server to client) loss rate conditions (Figure 20). As expected, TCP's cumulative acknowledgments enable it to maintain performance in the presence of reverse path (client to server) loss (Figure 21). To Trickles, reverse path losses are indistinguishable from forward path losses, so it achieves similar performance in both cases.

We used trace-driven emulation to characterize the effect of reordering on the throughput of Trickles connections. We collected reordering data for bursts, sent at one-minute intervals, consisting of 10 back-to-back, 1500 byte UDP packets, from 29 geographically diverse PlanetLab nodes to a node at Cornell. Over a 33-hour experiment, all but two nodes exhibited one or fewer reordering events.

We emulate the behavior of the nodes exhibiting a nontrivial degree of reordering in a 100 Mb/s LAN testbed as follows. Each reordered burst is converted to a potential pattern of reordering. Each pattern is associated with a given probability, based on the number of times the pattern occurs in the trace, over the event space of packets. On processing each packet, the network emulator will select a reordering pattern for subsequent packets using this probability distribution. The emulator introduces the measured RTT, but does not emulate the bottleneck bandwidth.

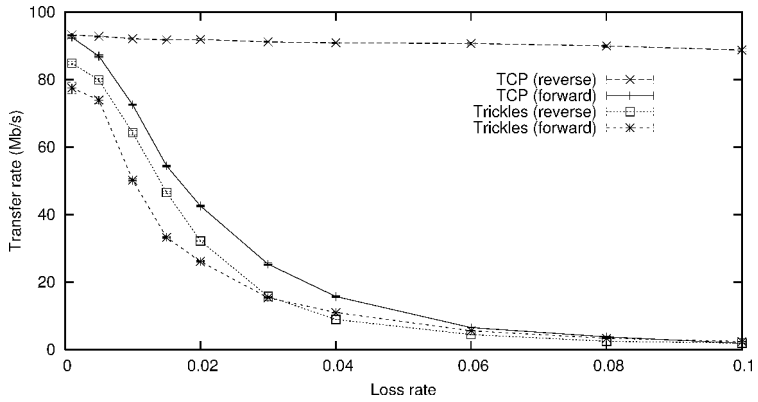

Fig. 21. Single client throughput versus forward and reverse path packet loss rate for RTT = 0ms. TCP cumulative acknowledgments make it robust against reverse path loss.

Figure 22 summarizes the parameters and results from transferring a 1 GB object over the two traces. The reordering performance is compared to two extrema: when no loss or reordering occurs, and when all reordering events are replaced by loss. The maximum achievable transfer rate for this RTT occurs under no losses, and is denoted by the top pair of horizontal lines. The bottom pair of horizontal lines denote the performance achieved when all reordering events are replaced by losses. On a sufficiently small timescale, reordering is indistinguishable from loss. Both TCP and Trickles attempt to improve performance by distinguishing between path reordering and loss. The difference in performance between the reordering and the loss experiments indicates the effectiveness of these optimizations.

Both TCP and Trickles achieve better performance under reordering than under loss. TCP's high performance is partly due to more sophisticated optimizations for reordering conditions. Reordering can trigger an erroneous back-off due to fast recovery/retransmit. After such an event, the Linux TCP stack will monitor the ACK stream for evidence that the back-off was triggered by a reordering and not a loss. If it determines that the back-off was due to reordering, any window size changes are rolled back. The Trickles protocol does not currently support rollback.

These scenarios cover a wide range of WAN conditions, and indicate how Trickles will perform on the Internet. Trickles and TCP exhibit similar behavior under delay and forward loss conditions. Reverse path loss and the occasional path with high reordering, can adversely impact Trickles performance. Client-side optimizations can potentially mitigate both. Since the client is stateful, it can record path statistics to infer reverse path loss or reordering. If reverse path loss is significant, the client can send redundant Trickles requests to the server. This will enhance performance with no changes to the server, as replay detection on the server will automatically drop the redundant requests. A client can adjust its fast recovery threshold in response to reordering, independent of help from the server; this is analogous to tuning a TCP sender's reordering threshold for the reordering/loss characteristics of a link. As seen in Figure 22,

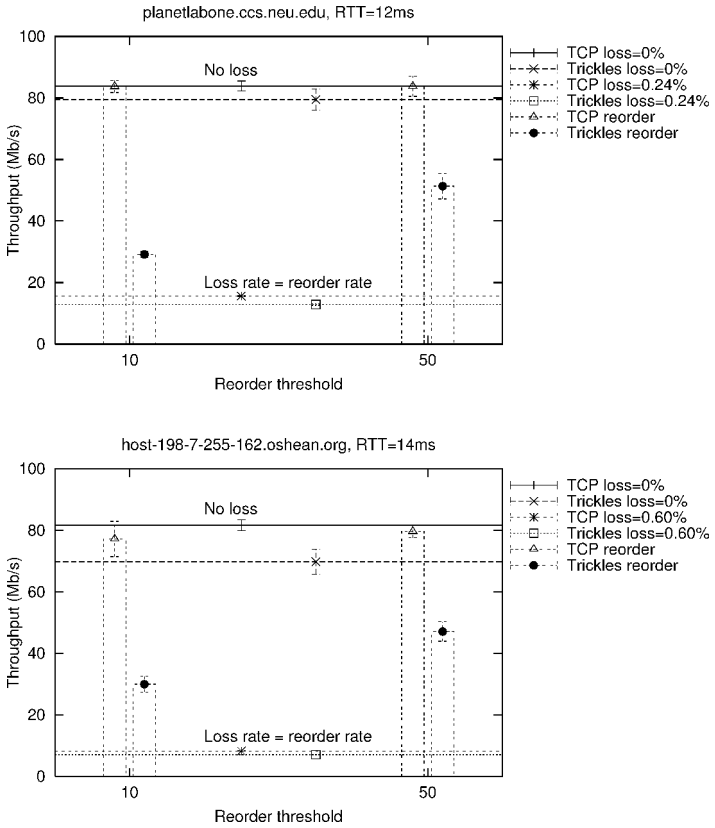

Fig. 22. Impact of reordering on TCP and Trickles on transferring 1 GB objects over a 100 Mb/s link. Trickles achieves higher performance under reordering than under an equivalent loss rate. Trickles performance can be improved purely by tuning the client to use a higher reordering threshold. TCP adapts to reordering by rolling back changes to congestion control state.

adjusting the reordering threshold can improve performance for links where reordering dominates loss, although this does not achieve the same performance as TCP. One can also extend the Trickles protocol to support rollback, at the expense of modifying both the client and the server.

Replay detection. Bloom filter-based replay detection prohibits all replays with a small amount of state, at the cost of potential false positives. We implemented the Bloom filter design using six hash functions of 21 bits in width to index a 2,097,152-entry table, and a one second horizon for swapping and flushing the two Bloom filters. As an optimization, the Bloom filter hash bits were extracted from the transport continuation MAC, thus amortizing the cost of MAC computation between integrity and replay prevention.

The performance impact of this Bloom filter is minimal. It imposed a 512 KB memory overhead, negligible CPU overhead (Figure 24), and minimal decrease in throughput. At 1 Gb/s interface speeds, the highest observed false positive rate was 0.00696%, lower than the theoretical rate of 0.0091%, and reduced throughput from 881 Mb/s to 876 Mb/s.

Replay detection parameter selection is determined by the expense of hash computations, the acceptable false positive rate, and the number of continuations hashed per time interval. These parameters provide the lowest false positive rate, given the 128 bits of *free* hash bits available from the MAC. Doubling the number of available hash bits enables 1,583,327-entry Bloom filters to achieve a similar false positive rate with 12 hash functions, for an overhead of 387 KB.

False positives increase the effective loss rate, which can degrade performance. For a given bound on performance degradation, the acceptable false positive rate depends on the loss rate of the underlying network. Higher loss rates or looser bounds on degradation both increase the acceptable false positive rate. Relaxing the degradation bound to 5% for a median Internet loss rate of 0.7% allows the use of 1,333,328-entry Bloom filter with seven hash functions, for an overhead of 323 KB.

For a given false positive rate, replay-detection Bloom filter size scales linearly with the number of continuations per time interval, which depends on the data rate of the connection and on the number of data packets generated per continuation. Our earlier analysis assumes that one data packet is generated per transport continuation check. Implementations that generate multiple data packets per continuation will encounter fewer continuations per time interval. Using one continuation per two packets halves the size of the Bloom filters, resulting in an overhead of 256 KB to achieve our original performance targets.

Optimizations. Delta encoding enables Trickles to transmit continuations with less space, reducing the overhead on each packet. We implemented delta encoding for transport continuations. In generating each request packet, each Trickles client computes the server's response transport continuation using the same algorithm as the server. Rather than sending a full transport continuation, the server sends only the differences between the client-computed continuation and the actual continuation. These differences are simply those fields that the client cannot predict: the timestamp, updated RTT, and MAC. This optimization increased throughput on the 100 Mb/s LAN bandwidth by 1.2 Mb/s, or 1.4%.

The SKIP and parallel-continuation request mechanisms allow Trickles to efficiently support pipelined transfers, enhancing protocol performance over wide area networks. We verified their effectiveness over WAN conditions by using nistnet [Carson and Santay 2005] to introduce artificial delays on a point-to-point, 100 Mb/s link. The single client maintained 10 outstanding pipelined requests, and the server sent SKIP notifications when 50% of the file was transmitted.

We compared the performance of TCP and Trickles for pipelined connections over a point-to-point link with 10 ms RTT. The object size was 250 KB. This object size ensures that the link can be filled, independent of the continuation request mechanism. Trickles achieves 86 Mb/s, and TCP 91 Mb/s. Thus with SKIP hints, Trickles achieves performance similar to that of TCP.

We also verified that issuing continuation requests in parallel improves performance. We added to the client-side API the `msk_request()` system call, which

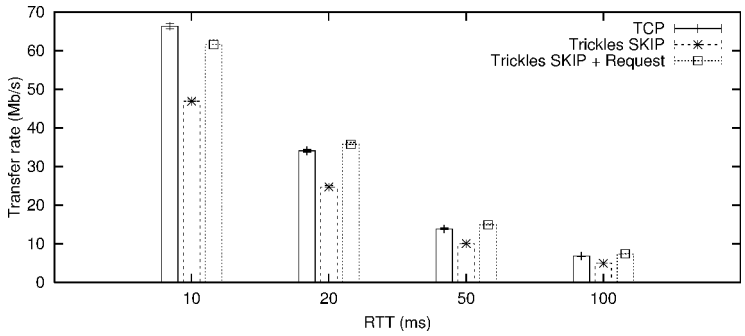

Fig. 23. Throughput comparison of pipelined transfers with 20 KB objects, smaller than the bandwidth-delay product. Trickles pipelining with Skip + Request significantly improves performance.

takes application-specified data and reliably transmits the data to the server for conversion into an output continuation. These requests immediately return control to the application, and multiple requests can be pending at any time. The `msk_request()` interface informs the client-side stack of the framing between requests. In the preceding experiments, which issued sequential requests, the compatibility layer cooperates with the server's input continuation processing to automatically translate socket `send()` on behalf of the application without this information. In Figure 23, object sizes are small, so a Trickles client using only SKIP with the sockets interface, cannot receive output continuations quickly enough to fill the link. A Trickles client using parallel requests can receive continuations more frequently, resulting in performance comparable to TCP, while still retaining the congestion control properties of a single connection.

CPU utilization. We compare the CPU overhead of Trickles with TCP under a highly stressful configuration. In this experiment, the server does not employ any soft-state caching, and verifies the state for every received continuation. In steady state, this corresponds to a continuation every two packets. A machine employing a 2.2GHz AMD Opteron CPU operating in 32-bit mode, acts as the server, with the MTU set to 1500 bytes. Six client machines connected to the server over a 1 Gb/s switch, simultaneously download a sequence of 1 GB objects.

Trickles achieves an average transfer rate of 881 Mb/s, while TCP achieves an average transfer rate of 947 Mb/s. Figure 24 shows a breakdown of the CPU overhead for Trickles and TCP. Not surprisingly, Trickles has higher CPU utilization than TCP, since it verifies and recomputes state that it does not keep locally. The overhead is evenly split between the cryptographic operations required for verification, and the packet processing required to simulate the TCP engine. While the Trickles CPU overhead is higher, it does not pose a server bottleneck, even at gigabit speeds. The higher overhead of Copy+Checksum for TCP is an artifact of the API differences. TCP's packetization is determined by the MTU and the amount of data specified in the `send()` system call. Maximum throughput is achieved with a large send buffer. However, a large send buffer

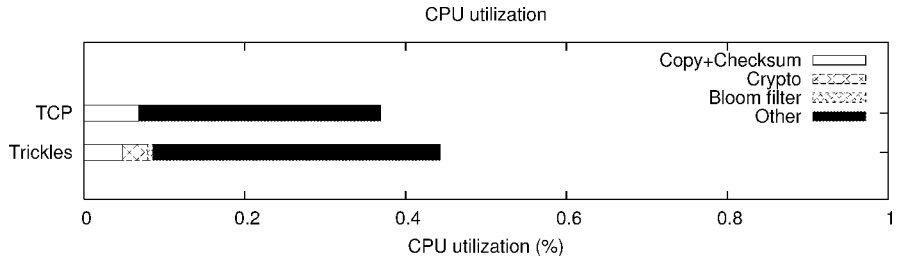

Fig. 24. Server-side CPU overhead on a 1 Gb/s link while transferring 1 GB objects to six simultaneous clients.

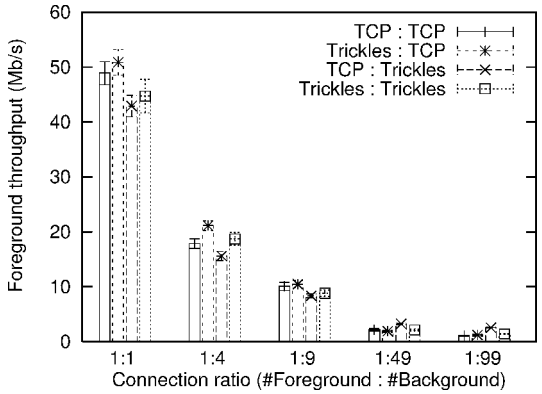

Fig. 25. Interaction of Trickles and TCP. Trickles has a similar impact on the foreground connection to TCP.

pollutes the processor cache. By contrast, we use a smaller, shared send buffer for Trickles, since we have better control over packetization.

Interaction with TCP flows. New transport protocols must not adversely affect existing flows on the Internet. Trickles is designed to generate similar packet-level behavior to TCP, and should therefore be TCP-friendly and achieve similar performance as TCP under similar conditions. To confirm this, we measured the bandwidth achieved by Trickles in the presence of background flows. We constructed a dumbbell topology with two servers on the same side of a 100 Mb/s bottleneck link, and two clients on the other side. The remaining links from the servers and clients to their respective bottleneck routers operated at 1 Gb/s. Each server was paired with one client, with connections occurring only within each server/client pair. One pair generated a single foreground TCP or Trickles flow. The other pair generated a variable number of background TCP or Trickles flows.

We compared the throughput achieved by the foreground flow for Trickles and TCP, versus varying numbers of background connections (Figure 25). In all cases, the foreground throughput of each configuration is comparable. Similarly, the ratio of slowest to fastest client is generally comparable, suggesting a similar degree of fairness for both Trickles and TCP (Figure 26). The anomaly for

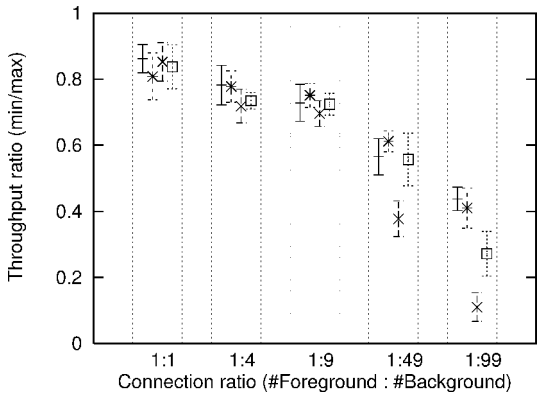

Fig. 26. Interaction of Trickles and TCP: ratio of slowest to fastest client. Trickles achieves similar fairness to TCP in most situations.

TCP:Trickles at ratios of 1:49 and 1:99 is due to the higher throughput achieved by the TCP foreground connection.

Microbenchmark Summary

Compared to TCP, Trickles achieves similar or better throughput under a variety of network conditions, and scales asymptotically better in terms of memory. It is also TCP-friendly. Trickles incurs a significant CPU utilization overhead compared to baseline TCP, but this additional CPU utilization does not pose a performance bottleneck, even at gigabit speeds. The continuation management mechanisms allow Trickles to achieve performance comparable to TCP over a variety of simulated network delays, and with both pipelined and nonpipelined connections.

6.2 Macrobenchmarks

The stateless Trickles protocol, and the new event-driven Trickles interface, enable a new class of stateless services. We examine three such services and evaluate Trickles under real-world network loss and delay. For multi-server configurations with k servers, loose clock synchronization between servers is needed for replay prevention. The choice of the horizon parameter T, the maximum age of a fresh packet, depends on δ, the maximum drift between server clocks. In the worst case, a fresh continuation may be misidentified as stale after only $T - \delta$ seconds, and a stale continuation may be misidentified as fresh after more than T seconds. Globally, each fresh continuation can be replayed at most k times, once at each server.

PlanetLab measurements. We validated Trickles under real Internet conditions using PlanetLab [Bavier et al. 2004]. We ran a variant of the throughput experiment in which both the server and the client were located in our local cluster, but with all traffic between the two nodes redirected (bounced) through a single PlanetLab node m. Packets are first sent from the source node to m, then from m to the destination node. Thus packets incur twice the underlying RTT to PlanetLab.

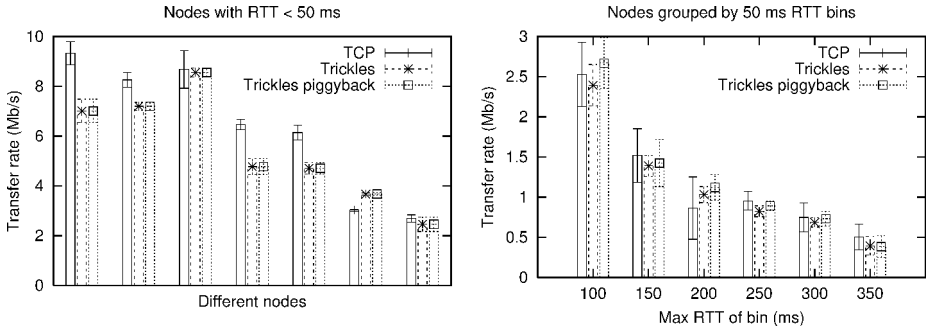

Fig. 27. Trickles and TCP PlanetLab transfer rate. Trickles achieves comparable transfer rate to TCP across a wide range of Internet paths.

Figure 27 summarizes the average throughput for a 160 KB file. With the exception of nodes with latency less than 50 ms, the PlanetLab nodes are grouped into 50 ms bins by the RTT measured by the endpoints. Due to widely varying measurements, the nodes with latency less than 50 ms are shown separately. These results show that Trickles achieves similar performance to TCP under comparable network conditions.

Geographically-aware anycast. Network redirection enables lower-level network services to flexibly adapt routing to changing network conditions independently of higher-level information. We implemented geographically aware anycast using this architecture. Since anycast infrastructure is not deployed on the Internet, we implemented an anycast primitive within the client-side kernel to create a simulated Internet infrastructure anycast, routing packets to the closest node associated with some IP address. Each Trickles connection is associated with a set of candidate servers, which are periodically pinged to determine the RTT. The kernel sends requests to the node with the shortest RTT, which is expected to be the geographically closest, and thus generally more appropriate than a more distant node. The kernel performs this redirection on a per-packet basis, just as packets are sent down to the IP layer, and thus interacts minimally with the Trickles stack itself.

To improve performance and reduce excess impact on concurrent flows, the congestion control parameters should be changed or reset on a routing change. In our prototype, the servers detect routing changes by attaching a site-specific identifier to transport continuations. Any received continuation with a nonmatching identifier is from a different site, and should be reset to slow-start at the server. For simplicity, our prototype uses a slightly less efficient approach: the servers drop all nonmatching requests except for slow-start requests. The resulting lack of response packets triggers a timeout request on the client, which resets the connection parameters.

We verified the performance benefits of performing geographically-aware network redirection over our PlanetLab bounce infrastructure in two experiments. In each experiment, we compared the unicast performance of TCP from a single server, with that of Trickles to the same single server, and Trickles with

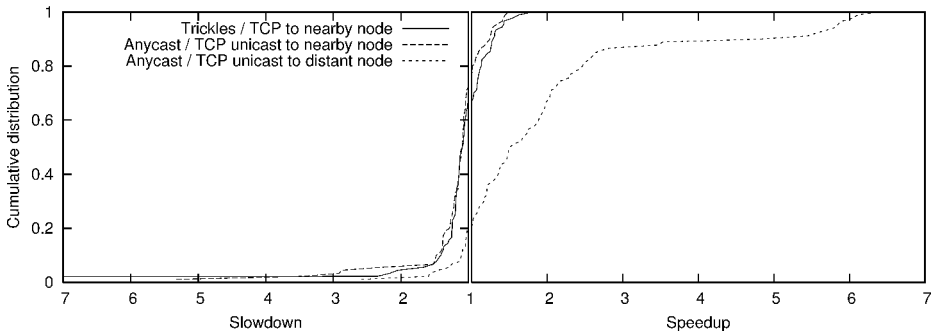

Fig. 28. Geographically-aware anycast: slowdown and speedup of Trickles relative to TCP. The performance of anycast selection closely matches that of unicast to the closer server, and exceeds that of unicast to the farther server.

two servers to select from, one nearby, the other distant. In all experiments, a single client transfers a sequence of 100 MB files. Figure 28 plots the slowdown and speedup of Trickles relative to TCP unicast. The speedup was computed using datapoints from experiments that are executed back to back, to increase the likelihood that the datapoints reflect similar network conditions. Each experiment consisted of 90 total runs, evenly distributed across pairs of nearby and distant nodes.

The first experiment compared the relative speed of unicast to a nearby node, with the speedup of server selection between nearby and distant servers. As expected, the speedups are similar, since the Trickles client will tend to choose the nearby node. Since Trickles achieves similar performance to TCP on most Internet links, the median speedup is 1. The wide distribution of slowdown and speedup is due to load variance on PlanetLab. The second experiment compared the relative speed of unicast to distant nodes with that of server selection. As expected, the anycast client performs better, with a median speedup of 1.5, since it is allowed to redirect to the nearby nodes. This result demonstrates the potential performance benefits of geographically-aware network redirection for applications such as content distribution networks.

Instantaneous failover. Trickles enables connections to fail-over from a failed server to a live backup, simply through a network-level redirection. If network conditions do not change significantly during the failover to invalidate the protocol parameters captured in the continuation, a server replica can resume packet processing transparently and seamlessly. In contrast, TCP recovery from server failure fundamentally requires several out-of-band operations. TCP needs to detect the disconnection, reestablish the connection with another server, and then ramp back up to the original data rate.

We compared Trickles and TCP failover on a 1 Gb/s single-server/single-client connection. To factor out overheads due to failure detection, we model a low-latency failure detector by killing and immediately restarting the server application. This is equivalent to failing-over to a live replica where total failure detection and route-change latency takes the same time as restarting the server. Figure 29 contains a trace illustrating the recovery of Trickles and TCP;

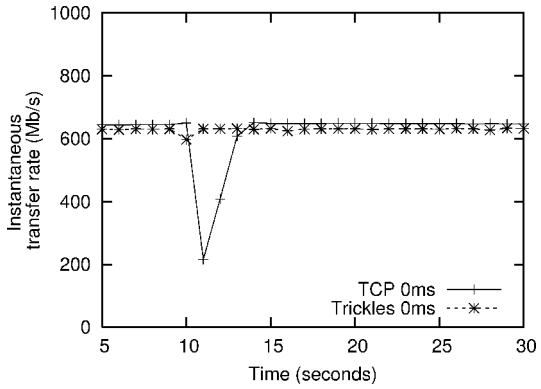

Fig. 29. Failover behavior. Disconnection occurs at $t = 10$ seconds.

the server is killed at 10 seconds. Since transient server failures are equivalent to packet loss at the network level, Trickles flows can recover quickly and transparently using fast recovery or slow-start. The explicit recovery steps needed by TCP increase its recovery time.

Packet-level load-balancing. Trickles requests are self-describing, and hence can be processed by any server machine. This allows the network to freely dispatch request packets to any server. With TCP, network-level redirection must ensure that packets from a particular flow are always delivered to the same server. Hence, Trickles allows load-balancing at packet granularity, whereas TCP allows load-balancing only at connection granularity.

Packet-level granularity improves the flexibility of bandwidth allocation. We used an IP layer packet sprayer to implement a clustered Web server with two servers and two clients. The IP packet sprayer uses NAT to present a single external server IP to the clients. In the test topology, the clients, servers, and packet sprayer are connected to a single Ethernet switch. The servers are connected to the switch at 100 Mb/s to introduce a single bottleneck on the server–client path.

TCP and Trickles tests used different load-balancing algorithms. TCP connections were assigned to servers using the popular *least connections* heuristic, which permanently assigns new TCP connections to the node with the fewest connections at arrival time. Trickles connections were processed using a per-packet algorithm that dispatched packets on a round-robin schedule. This scheme is incompatible with TCP, since packets within the same connection will be striped across multiple servers.

Figure 30 compares the Jain's fairness index [Jain 1991] of the total throughput versus the uniform allocation. For large numbers of connections, Trickles is fairer that TCP, since it more closely matches the uniform distribution.

Dynamic content. Loss recovery in a stateless system may require the recomputation of past data; this is more challenging for dynamic content. To demonstrate the generality of stateless servers, we implemented a watermarking media server that modifies standard media files to custom versions

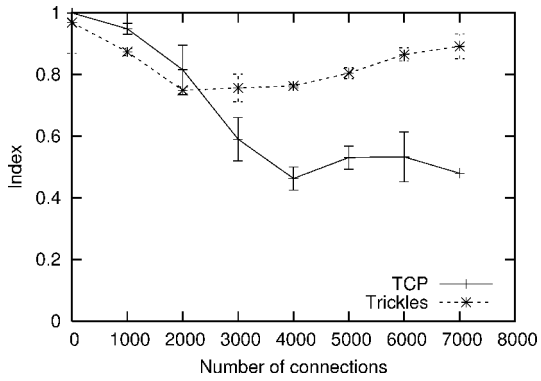

Fig. 30. Jain's fairness index in load-balancing cluster with two servers and two clients. Allocation is fair when each client receives the same number of bytes.

containing a client-specific watermark. Such servers are relevant for DRM media distribution systems, where content providers may apply client-specific transforms to digital media before transmission. Client customization inherently prevents multiple simultaneous downloads of the same object from sharing socket buffers, thus increasing the memory footprint of the network stack.

Stateless dynamic processing is possible with transforms that operate on individual blocks; the user continuation contains the location of the source file, and any client-specific parameters for driving the transformation, such as the watermarking client identifier or cryptographic key. Since each output is dependent on processing only a small window of the input file, reprocessing due to packet loss does not require rereading the entire prefix of the input file. Thus the additional overhead is linear in the packet loss rate.

We built a JPEG watermarking application that provides useful insights into continuation encoding for stateless operation. JPEG relies on Huffman coding of image data, which requires a nontrivial continuation structure. The exact bit position of a particular symbol after Huffman coding is not purely stateless, since it is dependent on the bit position of the previous symbols.

In our Trickles-based implementation of such a server, the output continuation records the bit alignments of encoded JPEG coding units at regular intervals. When generating output continuations, the server runs the watermarking algorithm to determine these bit positions, and discards the actual data. While processing a request, the server consults the bit positions in the output continuation for the proper bit alignment to use for the response.

An extended SKIP optimization can eliminate the preprocessing step: rather than requiring the server to provide precise bit alignment, the server would return unaligned data with SKIP markers that indicate the number of bits to skip when reassembling the data. Using application-defined sequence numbers can further improve performance. The watermarking application addresses images in terms of coding units (MCUs), while the transport layer addresses the datastream in bytes; the translation between the two is nontrivial. By specifying the datastream in terms of MCU number, the server could avoid the translation cost.

7. DISCUSSION

Trickles requires both the modification of server applications, and client-side changes. Here, we discuss the design of Trickles server applications, and propose ways to simplify Trickles deployment to clients.

7.1 Designing Trickles Server Applications

The primary source of complexity on the server side is in designing appropriate user continuations, since the congestion control and replay detection components are easy to use. We describe here the properties that determine whether Trickles is appropriate for a given application, and the user continuation structure used to support such applications. We present these techniques within an online shopping example.

Trickles is most appropriate for applications with modest server-side state requirements that can be captured with continuations. Trickles is not applicable to applications where servers hold a large amount of per-connection state. User continuations for interactive applications typically encode a complex state machine to capture the client session's state. In certain cases, such as providing shell access via TELNET or SSH, the session state space is prohibitively large for encoding within a user continuation, and must be held on the server. A stateless transport layer provides little benefit in this scenario, since the large application-level state dominates the scalability of the system.

Trickles imposes fewer constraints on applications whose server-side state is independent of the number of connections. Storing such state on the server does not affect scalability with respect to the number of connections. The content in content-delivery applications constitutes largely invariant state shared between many different clients. These generally use simple user continuations that summarize the name and location of the object.

A mismatch between the idempotency assumed by Trickles, and that of the server actions triggered by data sent from clients, can constrain the system design. A deterministic, fully stateless server cannot respond to the original request differently from the retransmitted requests in that it must execute the same operations and return the same data. Maintaining some server state is necessary when operations must not be repeated, for instance, when making an electronic payment, or posting to a bulletin board.

In some cases, a nonidempotent action can be made idempotent by restructuring where the state transitions occur. For instance, retransmissions should not cause items to be added to a shopping cart multiple times. This can be achieved by encoding the shopping cart contents within the user continuation. Suppose the client retransmits an *add to cart* request. Each (re)transmission uses the same user continuation, thus the same previous cart contents. Each time the server receives such a request, *add to cart* generates a continuation containing an updated shopping cart with the same new items. The client selects the first successfully received new user continuation as the next user continuation to use for the connection.

Support for multiple in-flight packets enhances performance, especially for higher RTTs; as seen in transport continuations, supporting this level of

parallelism increases the complexity of continuations. Simpler user continuations that support a lower level of parallelism are acceptable in certain situations. Close nodes might be serviced with lower parallelism if the increase in delay is below the perceptual threshold. With a sufficient number of clients, the bottleneck might shift from the low user continuation parallelism to the network link. Under high load or attack, a normally stateful Trickles server could resort to low parallelism user continuations to shed load, thus providing degraded service rather than rejecting new requests or failing.

Trickles enables new service optimizations based on partitioning an application or workflow into stateless and stateful portions, in a manner analogous to how different types of content, such as static versus dynamic, are handled differently by caching systems. Rather than executing a full client/server interaction in a stateful fashion, the inherently stateless portions of the interaction can now benefit from increased scalability, allowing the system to devote more resources to servicing the stateful operations.

This partitioning principle can be applied in both high level application design and in state caching. Many operations on a shopping site can operate statelessly. Browsing and searching for items in the catalog is stateless and idempotent. Though some sites do change state on browsing, such as in data-mining client activity for relationships between items in the catalog, this state change is not inherently per-client, since it grows with the number of items, rather than the number of clients. Furthermore, the statistical nature of these updates suggests that Bloom filters are sufficient for preventing retransmissions from skewing the analysis.

The user continuation for browsing simply points to the file that describes an object or category. Search results are often generated with a database query; the user continuation would encode the position within the array of query results from which to begin generating the next packet of data. Trickles can implement a stateless shopping cart using the technique described earlier. After checkout, any purchased digital goods would be delivered using a content distribution model: the checkout returns a continuation to bootstrap a stateless download of the item. Note that checkout server application is stateful to enforce *at-most-once* semantics on operations such as charging the customer's account.

State caching can improve the performance of the stateless portion. The state cache might cache the continuation state or the output queue. The replacement algorithm balances the size of server-side state, the size of the output, and the cost of recomputing data. The result of an expensive computation should be considered for long-term retention in the server cache. The smaller between the expanded state of a continuation and the total generated output should be inserted into the cache.

7.2 Client-Side Deployment

Several techniques can simplify the deployment of Trickles on the client side. Our prototype was implemented in the kernel to improve performance and transparently provide Trickles support to all applications running on a Trickles kernel. Implementing Trickles at user-level on the client would enable

incremental deployment, though existing operating systems restrict such a Trickles stack to operating over UDP. Trickles might be distributed via self-spreading transport protocols (STP) [Patel et al. 2003]. To fully support Trickles optimizations such as parallel request, STP should be augmented to support extensibility via protocol-specific system calls.

The SKIP optimization and support for partial input parsing (Section 4.2) enable the standard processing model for user continuations (Section 4.1) to transparently provide a sockets interface to any application where the server can provide a reasonable bound for the size of the data that a continuation might generate, that is, a bound for which multiple requests will not exhaust the sequence number space. Request sequence optimizations can improve the goodput of a system (Section 4.2), and custom data reconstruction can reduce the server-side processing of dynamic content generation (Section 6.2). To fully exploit these opportunities, Trickles and its deployment technology should provide mechanisms for servers to ship input preprocessing and output reconstruction code to clients.

8. RELATED WORK

This article builds upon our previous work on Trickles [Shieh et al. 2005], which introduced our stateless design principles and described our experience with the prototype implementation. Here, we validate Trickles in a wide range of LAN and WAN conditions, explore the parameter space for replay detection, discuss the design space for Trickles applications, and provide a detailed description of the protocol and algorithms.

Previous work has noted the enhanced scalability and security properties of stateless protocols and algorithms. Aura and Nikander [1997] describe a general framework for converting stateful protocols to stateless protocols, and apply it to authentication protocols. Since this framework assumes that the underlying communication channel is reliable, it is not directly applicable to a high-performance transport protocol. Stateless Core Routing (SCORE) [Stoica 2000] redistributes state in routing algorithms to improve scalability. Rather than placing state at the core routers, where holding state is expensive and often unfeasible, SCORE moves the state to the edge of the network. In contrast with these approaches, Trickles deals with the more general problem of streaming data, and provides a high performance stateless transport protocol.

Continuations are used in several existing network protocols and services. SYN cookies are a classic modification to TCP that use a simple continuation to eliminate per-connection state during connection setup [Bernstein 2005; Zúquete 2002]. NFS directory cookies [Sun Microsystems 1989] are application continuations. Continuations for Internet services have been explored at a coarser granularity than in Trickles. Session-based mobility [Snoeren 2002] adds continuations at the application layer to support migration and load balancing. Service Continuations [Sultan et al. 2003; Sultan 2004] record state snapshots, and move these to new servers during migration. In these systems, continuations are large and used infrequently in explicit migration operations controlled by connection endpoints. Trickles provides continuations at packet

level, enabling new functionality within the network infrastructure.

Receiver-driven protocols [Gupta et al. 2000; Hsieh et al. 2003] provide clients with more control over congestion control. Since congestion often occurs near clients, and is consequently more readily detectable by the client, such systems can adapt to congestion more quickly. Trickles contributes a secure, light-weight congestion control algorithm that enforces strong guarantees on receiver behavior, and low CPU and memory overhead mechanisms for protecting the server against denial of service attacks.

Transactional TCP (T/TCP) [Braden 1994] is an extension of TCP to improve the performance of workloads that are dominated by short requests and responses, each of which fit within a small number of packets. TCP Accelerated Open (TAO) enhances the performance of TCP by eliminating the three-way handshake in the common case and enabling SYN data to be delivered directly to applications in much the same way as Trickles with SYN data piggybacking. The *at-most-once* semantics of T/TCP constrain servers to allocate state to store a connection counter for each client. Reordering of SYN packets is likely to force a TAO failure, which forces the server to queue the data and revert to the standard three-way handshake. Since attackers can craft a sequence of SYN packets to trigger the three-way handshake, T/TCP is vulnerable to similar denial of service attacks as TCP [route|daemon9 1998]. Thus Trickles provides better denial of service protection than T/TCP.

Several kernel interfaces address the memory and event-processing overhead of network stacks. IO-lite [Pai et al. 1999] reduces memory overhead by enabling buffer sharing between different connections and the filesystem. Dynamic buffer tuning [Semke et al. 1998] allocates socket buffer space to connections where it is most needed. Event interfaces such as `epoll()`, `kqueue()`, and others [Banga et al. 1999; Lemon 2001] provide efficient mechanisms for multiplexing events from different connections.

The Cyclone shared TCP buffer provides memory footprint reduction by serving all clients fetching a particular file from a single digital fountain packet stream [Rost et al. 2001]. Cyclone uses traditional transport protocols, requiring less pervasive changes to the transport layer, at the expense of incurring per-connection state. Cyclone is designed for delay-insensitive applications, since Cyclone can pass data to client applications only when a complete coding block is received and decoded. Trickles is a complementary approach to Cyclone. Since the Trickles API provides fine-grain control of server-side state and retransmission in much the same way as the Cyclone server architecture, delay insensitive applications can layer Cyclone over a Trickles stack entirely with user-space server code. Also, Trickles enables delay-sensitive applications to achieve memory savings over TCP.

9. CONCLUSIONS

Trickles demonstrates that it is possible to build a completely stateless network stack that offers many of the desirable properties of TCP: namely, efficient and reliable transmission of data streams between two endpoints. As a result, the stateless side of a Trickles connection can offer good performance

with a much smaller memory footprint. Statelessness in Trickles extends all the way into applications: the server-side API enables servers to export their state to the client through a user continuation mechanism. Cryptographic hashes prevent untrusted clients from tampering with server state. Trickles is backwards compatible with existing TCP clients and servers, and can be adopted incrementally.

Beyond efficiency and scalability, statelessness enables new functionality that is awkward or impossible in a stateful system. Trickles enables load-balancing at packet granularity, instantaneous failover via packet redirection, and transparent connection migration. Trickles servers can be replicated, geographically distributed, and contacted through an anycast primitive, yet provide the same semantics as a single stateful server.

Statelessness is a valuable property in many domains. The techniques used to convert TCP to a stateless protocol—for example, the methods for working around the intrinsic information propagation delays—may also have applications to other network protocols and distributed systems.

APPENDIX

A. RANGE NONCE SECURITY PROOF

THEOREM A.1. *Let $p_1 \dots p_n$ be a set of nonces defined on a random sequence $r_1 \dots r_{n+1}$, where r_j and p_j are bitstrings of length N, with $p_k = r_k \oplus r_{k+1}$. Consider the range nonce $p_1 \oplus p_2 \oplus \dots \oplus p_n = r_1 \oplus r_{n+1}$. Then $P(r_1 \oplus r_{n+1} = s | p_1 \dots p_{j-1}, p_{j+1} \dots p_n) = P(r_1 \oplus r_{n+1} = s)$, that is, possessing all but one nonce yields no more information than having none at all.*

PROOF. Trivially, $P(r_1 \oplus r_{n+1} = s) = \frac{1}{2^N}$. We first show informally that

$$P(r_1 = t | p_1 \dots p_{j-1}, p_{j+1} \dots p_n) = P(r_1 = t | p_1 \dots p_{j-1}) = \frac{1}{2^N}.$$

Let w be some arbitrary bitstring of length N. Define a second sequence $y_1 \dots y_{n+1}$, $\forall_{i,1 \le i \le j} y_i = r_i \oplus w$, $\forall_{i,j+1 \le i \le n+1} y_i = r_i$. Consider the nonces p'_k defined on this sequence, $p'_k = y_k \oplus y_{k+1}$. $\forall_{i,i \ne j} p'_i = r_i \oplus w \oplus r_{i+1} \oplus w = p_i$, and $p'_j = p_j \oplus w$. This sequence of nonces is indistinguishable from the original sequence, since the distinguishing nonce p'_j is not available. Thus there are 2^N equally valid possibilities for r_1, and so the claim follows.

We will extend this intuition to prove the complete theorem. Let $Nonce(R)$ be the nonce sequence induced by some random sequence R. Let $A \sqsubset B$, where A, and B are each nonce sequences, represent the *consistency* relation. Each sequence may be incompletely specified, that is, potentially missing some indices. $A \sqsubset B$ holds if, for all indices i present in A, that index is present in B, and $\forall_i a_i = b_i$. Now, consider the sample space of the possible sequences. We wish to characterize those points of this space that are consistent with $P = \{p_1 \dots p_{j-1}, p_{j+1} \dots p_n\}$. This consists of at least the $2^N \times 2^N$ sequences of the form $R^{p,q}$, such that $\forall_{i,i \le j} R_i^{p,q} = r_i \oplus p$ and $\forall_{i,i > j} R_i^{p,q} = r_i \oplus q$.

We will show that all sequences $R, Nonce(R) \sqsubset P$ are of the preceding form, and hence the preceding enumeration of sequences using the parameters p, q is complete. Consider a sequence Q consistent with P, where this property is

violated for some q_i and q_{i+1}. $p_i = q_i \oplus q_{i+1}$, for otherwise $Nonce(Q) \not\sqsubset P$. There exists some unique l where $q_i = r_i \oplus l$. Suppose $q_{i+1} = r_{i+1} \oplus l'$ for $l' \neq l$. Then $q_i \oplus q_{i+1} = r_i \oplus l \oplus r_{i+1} \oplus l' = p_i \oplus (l \oplus l') \neq p_i$; a contradiction. Thus $l' = l$. From this result, it can be shown inductively that all q_i to the left of j must be of this form, all for the same p, and similarly for all q_i to the right of j, for a possibly different q.

Now, consider the range nonce value induced by the $R^{p,q}$. Let *range* be the range nonce of the actual nonce sequence, $range = r_1 \oplus r_{n+1}$, and $range^{p,q}$ be the range nonce corresponding to that of one of the equivalent worlds, given that one nonce is unknown. $range^{p,q} = (r_1 \oplus p) \oplus (r_{n+1} \oplus q) = (r_1 \oplus r_{n+1}) \oplus (p \oplus q) = range \oplus (p \oplus q)$. p and q can be picked independently, and each choice fully defines a world of probability $\frac{1}{2^N \times 2^N}$. Consider those worlds where $r_1 \oplus r_{n+1} = s$. These result when $s = range \oplus p \oplus q$. There are 2^N such points for each s, since p can be chosen arbitrarily from 2^N choices, and once p is chosen, q is uniquely determined. Thus each s occurs with probability $\frac{2^N}{2^N \times 2^N} = \frac{1}{2^N}$. This analysis was conducted given $p_1 \ldots p_{j-1}$, $p_{j+1} \ldots p_n$, and so

$$P(r_1 \oplus r_{n+1} = s | p_1 \ldots p_{j-1}, p_{j+1} \ldots p_n) = \frac{1}{2^N} = P(r_1 \oplus r_{n+1} = s). \qquad \square$$

B. CLOSED-FORM TCP SIMULATION

THEOREM B.1. *If startCwnd and ssthresh are the initial cwnd and ssthresh, respectively, at TCPBase, then the* `TCPCwnd(k)` *is a closed form solution mapping request packet number to TCP Reno's cwnd after processing that request, assuming that no requests are reordered, and no losses occur.*

PROOF. We consider the simplified problem where TCP Reno has just recovered from the last loss at *lastLossPosition*. This is equivalent to the original problem under a variable substitution: the request packet number of the following derivation is *relative* to the onset of the current loss-free epoch. We denote this class of relative sequence numbers as epoch-relative.

We will first derive k, the sequence number, as a function of *cwnd*, and then invert this function to define *cwnd* as a function of k. We first solve congestion avoidance with sequence numbers relative to the start of congestion avoidance, denoted as *loss-relative* packet numbers. Let s be the first sequence number at which congestion avoidance begins (that is, the point of the loss), and $cwnd = avoidanceCwnd$ at that point. The general case of slow-start followed by congestion avoidance simply requires a transform from epoch-relative packet numbers k' to loss-relative request packet numbers k, that is, taking $k = k' - s$.

Consider the first request sequence number (K_n in loss-relative sequence numbers, $s + K_n$ in epoch-relative sequence numbers) for which TCP Reno uses n as the *cwnd*. TCP Reno uses $cwnd = n$ for n consecutive requests, and $cwnd = n+1$ after the nth request. Then, K_n can be defined with the recurrence:

$$\begin{aligned} K_{avoidanceCwnd+1} &= avoidanceCwnd \\ K_{n+1} &= K_n + n. \end{aligned}$$

The closed form solution to this standard recurrence is:

$$\begin{aligned} K_n &= \Sigma_{j=avoidanceCwnd+1}^{n} j - 1 \\ &= \Sigma_{j=avoidanceCwnd}^{n-1} j \\ &= \frac{(n-1)n - (avoidanceCwnd - 1)avoidanceCwnd}{2}. \end{aligned}$$

We can consider this expression for K_n as a function from *cwnd* to a loss-relative sequence number. We can invert this function, yielding a function from loss-relative sequence number K to *cwnd*, by finding the zeroes of the following quadratic with indeterminant n:

$$\frac{(n-1)n - (avoidanceCwnd - 1)avoidanceCwnd}{2} - K.$$

Denote the positive root of this polynomial by $g(K)$.

While this derivation strictly applies only to the transition points where *cwnd* changes, the derivation generalizes to k, where $k \neq K_n$ for some n, those sequence numbers not at transition points. Since the positive root is a monotonically increasing function of the constant term of the quadratic, and the original TCP algorithm also monotonically increases *cwnd*, $G(k) = \lfloor g(k) \rfloor$ yields the *cwnd* for k. //

We are now prepared to prove the correctness of each case of `TCPCwnd`(k).

In slow-start, *cwnd* increases by one on every request, until *cwnd* reaches *ssthresh*. This is exactly the behavior captured by the first case of `TCPCwnd`(k). //

Slow start terminates at $k = A = ssthresh - startCwnd$, with $avoidanceCwnd = ssthresh$. Substituting into the definition of $g(K)$ yields a new function $f(K)$ for the points where *cwnd* changes, which represents the positive root of

$$\frac{(n-1)n - (ssthresh - 1)ssthresh}{2} - K.$$

As before, this function can be generalized to arbitrary sequence numbers r' by defining $F(r') = \lfloor f(r') \rfloor$, where r' is loss-relative. Applying the substitution $r' = r - A$ generates an expression in terms of epoch-relative sequence numbers. So $F(r - A)$ yields the *cwnd* for request packet number r. This proves the third case of `TCPCwnd`(k). //

The second case of `TCPCwnd`(k) properly handles relative sequence numbers prior to $A + K_{ssthresh}$: in this region *cwnd* has not yet increased from *ssthresh*. □

ACKNOWLEDGMENTS

We would like to thank Paul Francis, Larry Peterson, and the anonymous reviewers for their feedback.

REFERENCES

ALLMAN, M., PAXSON, V., AND STEVENS, W. 1999. RFC 2581: TCP Congestion Control.

AURA, T. AND NIKANDER, P. 1997. Stateless connections. In *Proceedings of the International Conference on Information and Communication Security*. Beijing, China, 87–97.

BALLANI, H. AND FRANCIS, P. 2004. Towards a deployable IP anycast service. In *Proceedings of the Workshop on Real, Large Distributed Systems*. San Francisco, CA.

BANGA, G., MOGUL, J. C., AND DRUSCHEL, P. 1999. A scalable and explicit event delivery mechanism for UNIX. In *Proceedings of the USENIX Annual Technical Conference*. Monterey, CA, 253–265.

BAVIER, A., BOWMAN, M., CHUN, B., CULLER, D., KARLIN, S., MUIR, S., PETERSON, L., ROSCOE, T., SPALINK, T., AND WAWRZONIAK, M. 2004. Operating systems support for planetary-scale network services. In *Proceedings of the Symposium on Networked Systems Design and Implementation*. San Francisco, CA.

BERNSTEIN, D. 2005. SYN Cookies. http://cr.yp.to/syncookies.html.

BLOOM, B. H. 1970. Space/time tradeoffs in hash coding with allowable errors. In *Commun. ACM*.

BRADEN, R. 1994. RFC 1644: T/TCP – TCP Extensions for Transactions.

CARSON, M. AND SANTAY, D. 2005. NIST Net. http://www-x.antd.nist.gov/nistnet.

CHAKRAVORTY, R., BANERJEE, S., RODRIGUEZ, P., CHESTERFIELD, J., AND PRATT, I. 2004. Performance optimizations for wireless wide-rea networks: comparative study and experimental evaluation. In *Proceedings of the International Conference on Mobile Computing and Networking*. Philadelphia, PA.

CRANE, D., PASCARELLO, E., AND JAMES, D. 2005. *Ajax in Action*. Manning Publications, New York, NY.

DAEMEN, J. AND RIJMEN, V. 1999. AES Proposal: Rijndael. http://csrc.nist.gov/encryption/aes/rijndael/Rijndael.pdf.

ELY, D., SPRING, N., WETHERALL, D., SAVAGE, S., AND ANDERSON, T. 2001. Robust congestion signaling. In *Proceedings of the International Conference on Network Protocols*. Riverside, CA, 332–341.

FAN, L., CAO, P., AND ALMEIDA, J. 1998. Summary cache: a scalable wide-Area Web cache sharing protocol. In *Proceedings of ACM SIGCOMM*. Vancouver, Canada.

FIELDING, R., GETTYS, J., MOGUL, J., FRYSTYK, H., MASINTER, L., LEACH, P., AND BERNERS-LEE, T. 1999. RFC 2616: Hypertext Transfer Protocol – HTTP / 1.1.

FLOYD, S. 1991. Connections with multiple congested gateways in packet-switched networks part 1: one-way traffic. *SIGCOMM Comput. Commun. Rev. 21,* 5, 30–47.

GUPTA, R., CHEN, M., MCCANNE, S., AND WALRAND, J. 2000. A receiver-driven transport protocol for the web. In *Proceedings of the INFORMS Telecommunications Conference*. San Antonio, TX.

HACKER, T. J., NOBLE, B. D., AND ATHEY, B. D. 2002. The effects of systemic packet loss on aggregate TCP flows. In *Proceedings of IEEE/ACM Supercomputing*. Baltimore, MD.

HSIEH, H.-Y., KIM, K.-H., ZHU, Y., AND SIVAKUMAR, R. 2003. A receiver-centric transport protocol for mobile hosts with heterogeneous wireless interfaces. In *Proceedings of the International Conference on Mobile Computing and Networking*. San Diego, CA.

JAIN, R. 1991. *The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling*. John Wiley and Sons, Inc.

JUELS, A. 1999. Client puzzles: a cryptographic countermeasure against connection depletion attacks. In *Proceedings of Networks and Distributed Security Systems*. San Diego, CA.

KRISHNAMURTHY, B., MOGUL, J. C., AND KRISTOL, D. M. 1999. Key differences between HTTP/1.0 and HTTP/1.1. In *Proceedings of the World Wide Web Conference*. Toronto, Canada.

LEMON, J. 2001. Kqueue: a generic and scalable event notification facility. In *Proceedings of the USENIX Annual Technical Conference*. Boston, MA.

MOGUL, J., BRAKMO, L., LOWELL, D. E., SUBHRAVETI, D., AND MOORE, J. 2004. Unveiling the transport. *SIGCOMM Comput. Commun. Rev. 34,* 1, 99–106.

NATIONAL INTERNET MEASUREMENT INFRASTRUCTURE. 2005. Distribution of packet drop rates. http://www.icir.org/models/NIMI-drop-rates.ps.

PAI, V. S., DRUSCHEL, P., AND ZWAENEPOEL, W. 1999. IO-Lite: a unified I/O buffering and caching system. In *Proceedings of the Symposium on Operating Systems Design and Implementation*. New Orleans, LA.

PATEL, P., WHITAKER, A., WETHERALL, D., LEPREAU, J., AND STACK, T. 2003. Upgrading transport protocols using untrusted mobile code. In *Proceedings of the Symposium on Operating Systems Principles*. Bolton Landing, NY.

ROST, S., BYERS, J., AND BESTAVROS, A. 2001. Cyclone server architecture: streamlining the delivery of popular content. In *Proceedings of the International Workshop on Web Caching and Content Distribution*. Boston, MA.

ROUTE|DAEMON9. 1998. T/TCP vulnerabilities. *Phrack Magazine 8,* 53.

SAVAGE, S., CARDWELL, N., WETHERALL, D., AND ANDERSON, T. 1999. TCP congestion control with a misbehaving receiver. *SIGCOMM Comput. Commun. Rev. 29,* 5, 71–78.

SEMKE, J., MAHDAVI, J., AND MATHIS, M. 1998. Automatic TCP buffer tuning. In *Proceedings of ACM SIGCOMM*. Vancouver, Canada.

SHIEH, A., MYERS, A. C., AND SIRER, E. G. 2005. Trickles: a stateless network stack for improved scalability, resilience, and flexibility. In *Proceedings of the Symposium on Networked Systems Design and Implementation*. Boston, MA.

SNOEREN, A. C. 2002. A session-based approach to internet mobility. Ph.D. thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology.

STANDARD PERFORMANCE EVALUATION CORPORATION. 1999. The SPECweb99 benchmark.

STOICA, I. 2000. Stateless core: a scalable approach for quality of service in the internet. Ph.D. thesis, Department of Electrical and Computer Engineering, Carnegie Mellon University.

SULTAN, F. 2004. System support for service availability, remote healing and fault tolerance using lazy state propagation. Ph.D. thesis, Division of Computer and Information Sciences, Rutgers University.

SULTAN, F., BOHRA, A., AND IFTODE, L. 2003. Service ontinuations: an operating system mechanism for dynamic migration of Internet service sessions. In *Proceedings of the Symposium on Reliable Distributed Systems*. Florence, Italy.

SUN MICROSYSTEMS. 1989. RFC 1094: NFS: Network File System Protocol Specification.

ZÚQUETE, A. 2002. Improving the functionality of SYN cookies. In *Proceedings of the IFIP Communications and Multimedia Security Conference*. Portoroz, Slovenia.

Received July 2005; revised January 2008; accepted July 2008