Trickles: A Stateless Network Stack for Improved Scalability, Resilience, and Flexibility

Alan Shieh　　Andrew C. Myers　　Emin Gün Sirer

Dept. of Computer Science
Cornell University
Ithaca, NY 14853
{ashieh,andru,egs}@cs.cornell.edu

Abstract

Traditional operating system interfaces and network protocol implementations force system state to be kept on both sides of a connection. Such state ties the connection to an endpoint, impedes transparent failover, permits denial-of-service attacks, and limits scalability. This paper 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 backwards-compatible, requiring no changes to sockets-based client applications.

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 session state across *both* sides of a point-to-point connection. Distributed state leads to three problems. First, connection failover and recovery is difficult, non-transparent, or both, as reconstructing lost state is often non-trivial. 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, increasing overhead for short connections, and Internet servers often do not allow long-running persistent HTTP connections, increasing overhead for bursty accesses [8]. Finally, state in protocol stacks limits scalability: servers cannot scale up to large numbers of clients because they need to commit per-client resources, and similarly cannot scale down to tiny embedded devices, as there is a lower bound on the resources needed per connection.

In this paper, 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 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 server state enable services to be replicated and migrated between servers. 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 the key concern about the suitability of anycast primitives [3] 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 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 kernel and application-level state.

Overall, this paper 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 paper 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

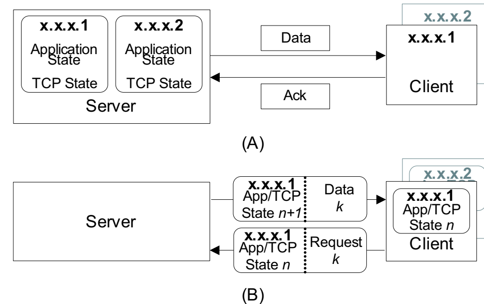

Figure 1: TCP versus Trickles state. (A) TCP holds state at server, even for idle connection x.x.x.2. (B) Trickles encapsulates and ships server state to the client.

implementation. Section 7 discusses related work and Section 8 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 [26]. Trickles also supports stateless user-level server applications; to permit a server application to suspend processing without retaining state, the application may add an analogous

user continuation to outgoing packets.

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 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.

With Trickles, responsibility for data retransmission lies with the server application, since a reliable queuing mechanism, such as that found in TCP implementations, is stateful and holds data in a send buffer until it is acknowledged. Therefore, 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 allows applications to control the amount of state devoted to each connection, and share buffer space where possible.

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 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 are $75+12m$ 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 [1].

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 [2]. 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.

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.

Hiding losses [19, 10] 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 clients signal losses using selective acknowledgment (SACK) proofs, computed from the packet nonces, that securely describe the set of packets received. The nonces are grouped by contiguous ranges, and are compressed into a compact range summary that can be checked efficiently. Let p_i be packet i's nonce. The range $[m,n]$ is summarized by XORing together each p_i in the range into a single word. Imposing additional structure on the nonces enables Trickles to generate per-packet nonces and to verify multi-packet ranges in $O(1)$ time. Define a sequence of random numbers $r_x = f(K,x)$, where f is a keyed cryptographic hash function. If $p_i = r_i \oplus r_{i+1}$, then $p_1 \oplus p_2 \oplus ... \oplus p_n = r_1 \oplus r_{n+1}$. Thus, the server can generate and verify nonces with a constant number of 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.

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

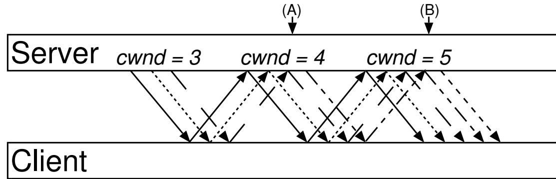

Figure 2: A sample TCP or Trickles connection. Each line pattern corresponds to a different trickle. Initially, there are $cwnd$ trickles. At points where $cwnd$ increases (A, B), trickles are split.

be small and independent of the number of connections. One possible replay defense is a simple hash table keyed on the transport continuation MAC. This bounds the effect of a replay attack, and if hash collisions indicate the presence of an attack, the size of the hash table can be increased. The hash table records recent packet history up until a time horizon. Each transport continuation carries a server-supplied timestamp that is checked on packet arrival; packets older than the time horizon are simply discarded. 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.

The replay defense mechanism 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 mechanism in the application is more flexible, because application-specific knowledge can be applied. For simplicity and flexibility, we have chosen to place replay defense in the application. In either case, Trickles is more robust against state-consumption attacks than TCP.

2.3 The trickle abstraction

Figure 2 depicts the exchange of packets between the two ends of a typical TCP or Trickles connection. For simplicity, there is no loss, packets arrive in order, and delayed acknowledgments are not used. 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 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 is decomposed into multiple disjoint trickles; each packet is a member of exactly one trickle. These trickles can be thought of as independent threads that exchange information only 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. However, 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.

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 split or terminate trickles to generate the same number of response packets. The subsequent sections describe how to statelessly perform these decisions to match the congestion avoidance, slow start, and fast retransmit behavior of TCP.

2.4 Trickle dataflow constraints

Statelessness complicates matching TCP behavior, because it fundamentally restricts the data flow allowed between the processing of different packets. This restriction is the main source of complexity in designing a stateless transport protocol.

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. Consider the comparison in Figure 3, illustrating 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 the point (B). At the earliest, this state update can be made available at point (D) in the figure, after a round trip through 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 ex-

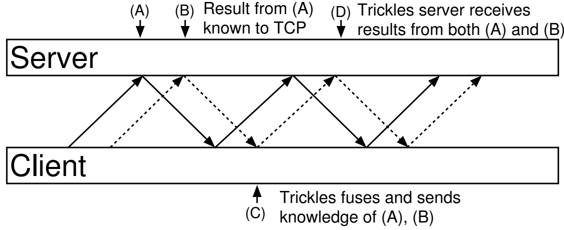

Figure 3: Difference in state/result availability between TCP and Trickles. TCP server knows the result of processing (A) earlier than Trickles server.

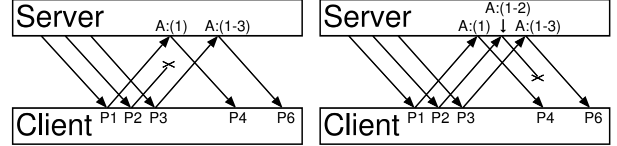

Figure 4: Equivalence of reverse and forward path loss in Trickles. Due to dataflow constraints, the packet following a lost packet does not compensate for the loss immediately. Neither the server nor the client can distinguish between (A) and (B). The loss will be discovered through subsequent SACK proofs.

ample illustrates that server state cannot propagate directly between the processing of consecutive packets, but is available to server-side processing a round trip later.

The round-trip delay in state updates makes it challenging to match the congestion control action of TCP. Trickles circumvents the delay by using prediction. 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 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.

Concurrent trickles must respond consistently and quickly to loss events. By providing each trickle with the information needed to predict 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 predict 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 fully reflects the server's response to those losses. This is guaranteed to occur after all loss events, once recovery or retransmission timeout finishes.

With any prediction scheme, it is sometimes necessary to recover from misprediction. 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 4). 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 [1]: congestion avoidance/slow start, 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 move the number of trickles toward this target. Between modes, the set of valid trickles changes to reflect the increase or decrease in $cwnd$. In general, the number of trickles will decrease in a mode transition; the valid trickles in the new mode are known as *survivors*.

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:

- $\texttt{CwndDelta} := \texttt{TCPCwnd}(k) - \texttt{TCPCwnd}(k-1)$
- Generate $\texttt{CwndDelta} + 1$ responses: continue the original trickle, and split CwndDelta times.

$$\texttt{TCPCwnd}(k) = \begin{cases} startCwnd + (k - TCPBase) & \textbf{if } k < A \\ ssthresh & \textbf{if } A \leq k < A + ssthresh \\ F(k - A) + 1 + ssthresh & \textbf{if } A + ssthresh \leq k \end{cases}$$

where

$$A = ssthresh - startCwnd + TCPBase$$

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

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

Figure 5: Closed-form solution of TCP simulation.

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 5, which can be computed in $O(1)$ time.

The $\texttt{TCPCwnd}(k)$ formula is directly valid only for connections where no losses occur. A connection with losses can be partitioned at the loss positions into multiple pieces without losses; $\texttt{TCPCwnd}(k)$ is valid within each individual piece. The free parameters in $\texttt{TCPCwnd}(k)$ are used to adapt the formula for each piece: $startCwnd$ and $ssthresh$ are initial conditions at the point of the loss, and $TCPBase$ corresponds to the last loss location.

Fast retransmit/recovery

In fast retransmit/recovery, TCP Reno uses duplicate acknowledgments to infer the position of a lost packet (Figure 6). The lost packet is retransmitted, the $cwnd$ is halved, and transmission of new data temporarily squelched 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 $newCwnd$ (Figure 7).

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. Since multiple trickles must execute the algorithm in parallel, each with a different recovery role, the SACK prefix property is critical to proper operation, as it allows each trickle to predict the input and recovery action of preceding trickles. A client that violates the prefix property in packets it sends to the server will cause

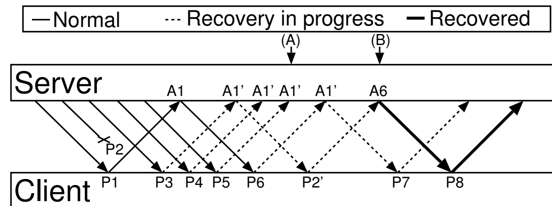

Figure 6: TCP recovery. Duplicate ACKs signal recovery (A). Subsequent ACKs are ignored until number of outstanding packets drops to new $cwnd$. Recovery ends when client acknowledges all packets (B).

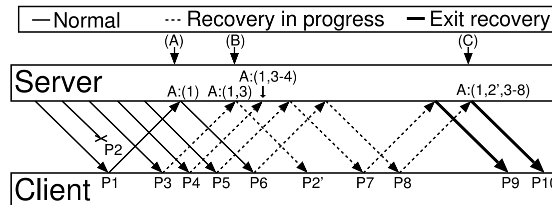

Figure 7: Trickles Recovery. First packet following loss triggers a retransmission (A). Trickles are subsequently terminated to deflate $cwnd$ (B). Recovery ends when $cwnd$ survivors are generated; $cwnd$ has dropped from the original value of 5 to 2 (C).

inconsistent computations on the server side, and may receive data and transport continuations redundantly or not receive them at all.

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

1. ```
   firstLoss := sequence 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, retransmit the lost packets (Figure 7). This is required to achieve the reliable delivery guarantees of TCP. A `burstLimit` parameter, similar to that suggested for TCP [1], 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 < newCwnd`, continue the trickle. 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 sequence 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, as 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.

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.

The client side of Trickles provides to the client application a standard Berkeley sockets interface, so the client application need not be aware of the existence of Trickles, and 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.

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, as 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.

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 all the response trickles. 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
   ```

```
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);
```

Figure 8: The minisocket API.

is bounded by the ingress bandwidth. The event queue eliminates a layer of multiplexing and demultiplexing found in the traditional sockets API that can cause excess processing overhead [4].

To avoid copying of events, the event queue is a synchronization-free linked list mapped in both the application and kernel; it is mapped read-only in the application, and can be walked by the application without holding locks. While processing requests, the kernel allocates all per-request data structures in the shared region.

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.

3.3 Minisocket operations

The minisocket API is shown in Figure 8. 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 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 [17] for TCP, Trickles can redistribute events at a packet-level granularity.

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

A Trickles client stack implements a Berkeley sockets interface using the Trickles transport protocol. Thus, the client application need not be aware of the presence of Trickles. 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.

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.

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.

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 client cannot request an arbitrarily-sized range because the congestion control algorithm restricts the amount of data that may be returned for each request. To compute the proper byte range size, 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.

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. 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 can enable the server to operate in a stateful manner whenever resources permit and reduce 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 [15], or complex application-level protocols, such as HTTP 1.1 [11], for framing. In contrast, trickles are inherently concurrent. Concurrency can improve the performance of both fetching and sending data to the server.

The Trickles protocol allows a client application to concurrently request different, non-adjoining sequence number ranges from the server on a single connection. The response packets from the server, which will carry data belonging to different objects distributed through the sequence number space, will nevertheless be subject to a single TCP-friendly flow equation, acting in effect like a single, HTTP/1.1-like flow with application level framing. Since, in some cases, the sizes of the objects may not be known in advance, Trickles clients can conservatively dedicate large regions of the sequence number space to each object. A server response packet may include a SKIP notification that indicates 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. Consequently, the inherent parallelism between Trickles can be used to multiplex logically separate transmissions on a given connection, while subjecting them to the same flow equation.

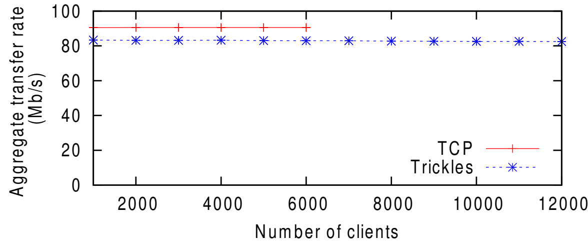

Figure 9: Aggregate throughput for Trickles and TCP. TCP fails for tests with more than 6000 clients.

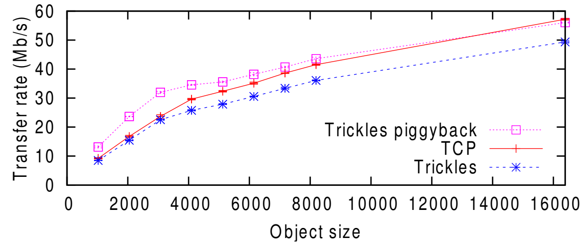

Figure 10: Trickles and TCP throughput for a single, isolated client at various object sizes.

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 parallel, a client can reduce the total latency of a sequence of such operations.

5.3 Delta encoding

While continuations add extra space overhead to each packet, predictive header compression 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, as the client cannot compute the secure server hash.

6 Evaluation

In this section, we evaluate the quantitative performance of Trickles through microbenchmarks, and show that it performs well compared to TCP, consumes few resources, scales well with the number of clients, 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 [9] for the keyed cryptographic hash function. All results include at least six data points; error bars indicate the 95% confidence interval.

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

6.1 Microbenchmarks

In this section, we use a high-performance server microbenchmark to examine the throughput, scaling, and TCP-friendliness properties of Trickles.

Throughput

We tested throughput using a point-to-point topology with a single server node placed behind a 100 Mb/sec bottleneck link. Varying numbers of simultaneous client instances (distributed across two real CPUs) repeatedly fetched a 500 kB file from the server. A fresh connection was established for each request.

Figure 9 shows that the aggregate throughput achieved by Trickles is within 10% of TCP at all client counts. Regular TCP consumes memory separately for each connection to buffer outgoing data until it is acknowledged. 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.

With Trickles, a client fetching small objects will achieve significant performance improvements because of the reduction in the number of control packets (Figure 10). Trickles requires fewer packets for connection setup than TCP. Trickles processes data embedded in SYN packets into output continuations without holding state, and can send an immediate response; to avoid creating a DoS amplification vulnerability, the server should not respond with more data than it received. In contrast, TCP must save or reject SYN data; because holding state increases vulnerability to SYN flooding, most TCP stacks reject SYN data. Unlike TCP, Trickles does not require FIN packets to clean up server-side state. The combination of SYN data and lower connection overhead improves small file transfer throughput for Trickles, with a corresponding improvement in transfer latency.

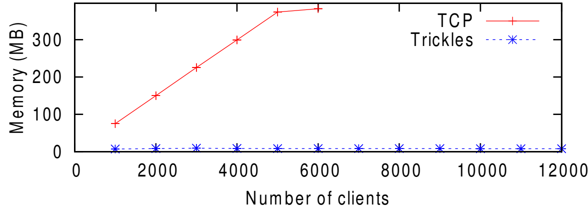

Figure 11: Memory utilization. Includes socket structures, socket buffers, and shared event queue.

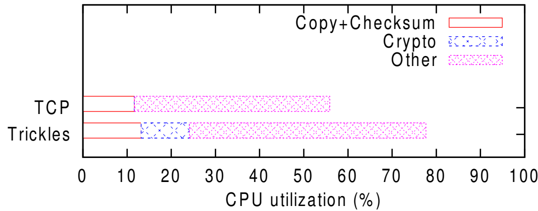

Figure 12: Server-side CPU overhead on a 1 Gb/s link.

Memory and CPU utilization

We next examine the memory and CPU utilization of the Trickles protocol. For this experiment, we eliminated the bottleneck link in the network and connected the clients to the server through the full 1Gb/sec link to pose a worst-case scenario.

Not surprisingly, Trickles consistently achieves better memory utilization than TCP (Figure 11). TCP memory utilization increases linearly with the number of clients, while statelessness enables Trickles to uses a constant, small amount of memory.

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, though they pose a risk because increased connection duration can increase the number of simultaneous connections. Consequently, many websites disable persistent connections to the detriment of their users. 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. Finally, Trickles's smaller memory footprint provides more space for caching, benefiting all connections.

Figure 12 shows a breakdown of the CPU overhead for Trickles and TCP on a 1 Gb/s link when Trickles is reconstructing its state for every packet (i.e. soft-state caching is turned off). 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.

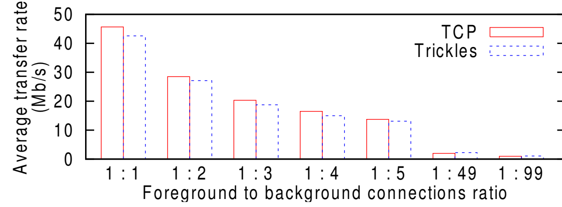

Figure 13: Interaction of Trickles and TCP.

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 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 bottleneck link, and two clients on the other side. The remaining links from the servers and clients to their respective bottleneck routers operated at 1000 Mb. 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 flows. We compared the throughput achieved by the foreground flow for Trickles and TCP, versus varying numbers of background connections (Figure 13). In all cases, Trickles performance was similar to that of TCP.

Continuation optimizations

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 [7] to introduce artificial delays on a point-to-point, 100 Mb link. The single client maintained 10 outstanding pipelined requests, and the server sent advanced 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 10ms RTT. The file size was 250kB. 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.

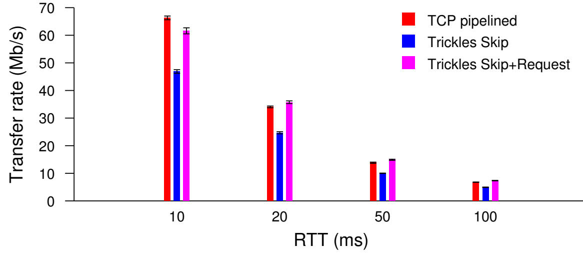

Figure 14: Throughput comparison of pipelined transfers with 20 kB objects, smaller than the bandwidth-delay product.

We also verified that issuing continuation requests in parallel improves performance. We added the `msk_request()` interface that takes application-specified data and reliably transmits the data to the server for conversion into an output continuation. These requests are non-blocking, and multiple such requests can be pending at any time. In Figure 14, the object sizes are small, so a Trickles client using SKIP with the sockets interface cannot receive output continuations quickly enough to fill the link. The Trickles client supporting parallel requests can receive continuations more frequently, resulting in performance comparable to TCP.

Summary

Compared to TCP, Trickles achieves similar or better throughput and scales asymptotically better in terms of memory. It is also TCP-friendly. Trickles incurs a significant CPU utilization overhead versus 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 non-pipelined 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 we also evaluate Trickles under real-world network loss and delay.

PlanetLab measurements

We validated Trickles under real Internet conditions using PlanetLab [5]. 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.

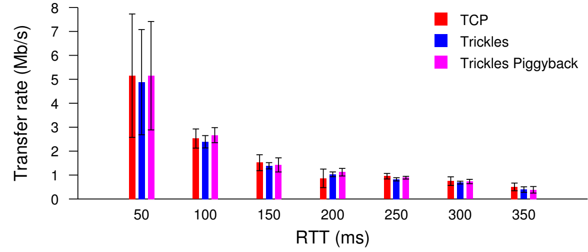

Figure 15: Trickles and TCP PlanetLab throughput.

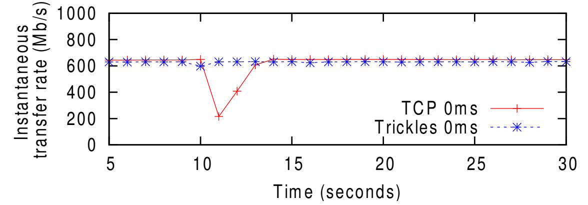

Figure 16: Failover behavior. Disconnection occurs at $t = 10$ seconds.

Figure 15 summarizes the average throughput for a 160kB file. PlanetLab nodes are grouped into 50 ms bins by the RTT measured by the endpoints. Trickles achieves similar performance to TCP under comparable network conditions.

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, re-establish the connection with another server, and then ramp back up to the original data rate.

We compared Trickles and TCP failover on a 1000 Mb single server/single client connection. At 10 seconds, the server application is killed and immediately restarted. Figure 16 contains a trace illustrating the recovery of Trickles and TCP. 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 increases 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

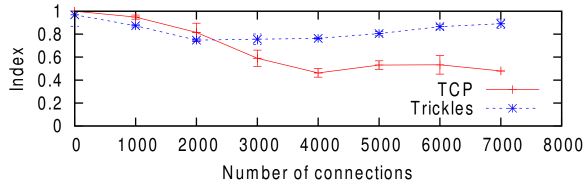

Figure 17: Jain's fairness index in load balancing cluster with two servers and two clients. Allocation is fair when each client receives the same number of bytes.

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 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 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 least number of connections at arrival time. Trickles connections were processed using a per-packet algorithm that dispatched packets on a round-robin schedule.

Figure 17 compares the Jain's fairness index[14] of the total throughput versus the uniform allocation. For most data points, Trickles more closely matches the uniform distribution than TCP does.

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 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.

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 non-trivial continuation structure. The exact bit position of a particular symbol after Huffman coding is not purely stateless, as 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.

7 Related Work

Previous work has noted the enhanced scalability and security properties of stateless protocols and algorithms. Aura et al. [2] developed a general framework for converting stateful protocols to stateless protocols, and applied this to authentication protocols, and noted denial-of-service resilience and potential for anycast applications as benefits of stateless protocols. Trickles deals with the more general problem of streaming data, provides a high performance stateless congestion control algorithm. Stateless Core Routing (SCORE) [22] redistributes state in routing algorithms to improve scalability. Rather than placing state at the core routers, where holding state is expensive and often infeasible, SCORE moves the state to the edge of the network.

Continuations are used in several existing systems. SYN cookies are a classic modification to TCP that uses a simple continuation to eliminate per-connection state during connection setup [6, 27]. NFS directory cookies [25] are application continuations.

Continuations for Internet services have been explored at a coarser granularity than in Trickles. Session-based mobility [21] adds continuations at the application layer to support migration and load balancing. Service Continuations [23, 24] 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 [12, 13] 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.

Several kernel interfaces address the memory and event-processing overhead of network stacks. IO-lite [18] reduces memory overhead by enabling buffer sharing between different connections and the filesystem. Dynamic buffer tuning [20] allocates socket buffer space

to connections where it is most needed. Event interfaces such as epoll(), kqueue(), and others [16, 4] provide efficient mechanisms for multiplexing events from different connections.

8 Conclusions and future work

Trickles demonstrates that it is possible to build a completely stateless network stack that offers many of the desirable properties of TCP; namely, efficient, reliable transmission of data streams between two endpoints. As a result, the stateless side of a Trickles connection can offer good performance with a very small 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 may be replicated, geographically distributed, and contacted through an anycast primitive, and 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.

Acknowledgments

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

This work was supported by the Department of the Navy, Office of Naval Research, ONR Grant N00014-01-1-0968; and National Science Foundation grants 0208642, 0133302, and 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.

References

[1] M. Allman, V. Paxson, and W. Stevens. TCP Congestion Control. *RFC 2581*, Apr. 1999.

[2] T. Aura and P. Nikander. Stateless Connections. In *Proceedings of the International Conference on Information and Communication Security*, pages 87–97, Beijing, China, Nov. 1997.

[3] H. Ballani and P. Francis. Towards a Deployable IP Anycast Service. In *Proceedings of the Workshop on Real, Large Distributed Systems*, San Francisco, CA, Dec. 2004.

[4] G. Banga, J. C. Mogul, and P. Druschel. A Scalable and Explicit Event Delivery Mechanism for UNIX. In *Proceedings of the USENIX Annual Technical Conference*, pages 253–265, Monterey, CA, June 1999.

[5] A. Bavier, M. Bowman, B. Chun, D. Culler, S. Karlin, S. Muir, L. Peterson, T. Roscoe, T. Spalink, and M. Wawrzoniak. Operating Systems Support for Planetary-Scale Network Services. In *Proceedings of the Symposium on Networked Systems Design and Implementation*, San Francisco, CA, Mar. 2004.

[6] D. Bernstein. SYN Cookies. http://cr.yp.to/syncookies.html.

[7] M. Carson and D. Santay. NIST Net. http://www-x.antd.nist.gov/nistnet.

[8] R. Chakravorty, S. Banerjee, P. Rodriguez, J. Chesterfield, and I. Pratt. Performance Optimizations for Wireless Wide-Area Networks: Comparative Study and Experimental Evaluation. In *Proceedings of MobiCom*, Philadelphia, PA, Sept. 2004.

[9] J. Daemen and V. Rijmen. AES Proposal: Rijndael, 1999.

[10] D. Ely, N. Spring, D. Wetherall, S. Savage, and T. Anderson. Robust Congestion Signaling. In *Proceedings of the International Conference on Network Protocols*, pages 332 – 341, Riverside, CA, Nov. 2001.

[11] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext Transfer Protocol – HTTP / 1.1. *RFC 2616*, June 1999.

[12] R. Gupta, M. Chen, S. McCanne, and J. Walrand. A Receiver-Driven Transport Protocol for the Web. In *Proceedings of INFORMS*, San Antonio, TX, Nov. 2000.

[13] H.-Y. Hsieh, K.-H. Kim, Y. Zhu, and R. Sivakumar. A Receiver-Centric Transport Protocol for Mobile Hosts with Heterogeneous Wireless Interfaces. In *Proceedings of MobiCom*, San Diego, CA, Sept. 2003.

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

[15] B. Krishnamurthy, J. C. Mogul, and D. M. Kristol. Key Differences between HTTP/1.0 and HTTP/1.1. In *Proceedings of the World Wide Web Conference*, Toronto, Canada, May 1999.

[16] J. Lemon. Kqueue: A Generic and Scalable Event Notification Facility. In *Proceedings of the USENIX Annual Technical Conference*, Boston, MA, June 2001.

[17] J. Mogul, L. Brakmo, D. E. Lowell, D. Subhraveti, and J. Moore. Unveiling the Transport. *SIGCOMM Computer Communications Review*, 34(1):99–106, 2004.

[18] V. S. Pai, P. Druschel, and W. Zwaenepoel. IO-Lite: A Unified I/O Buffering and Caching System. In *Proceedings of the Symposium on Operating Systems Design and Implementation*, New Orleans, LA, Feb. 1999.

[19] S. Savage, N. Cardwell, D. Wetherall, and T. Anderson. TCP Congestion Control with a Misbehaving Receiver. *SIGCOMM Computer Communications Review*, 29(5):71–78, 1999.

[20] J. Semke, J. Mahdavi, and M. Mathis. Automatic TCP Buffer Tuning. In *Proceedings of ACM SIGCOMM*, Vancouver, Canada, Aug. 1998.

[21] A. C. Snoeren. *A Session-Based Approach to Internet Mobility*. PhD thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Dec. 2002.

[22] I. Stoica. *Stateless Core: A Scalable Approach for Quality of Service in the Internet*. PhD thesis, Department of Electrical and Computer Engineering, Carnegie Mellon University, 2000.

[23] F. Sultan. *System Support for Service Availability, Remote Healing and Fault Tolerance Using Lazy State Propagation*. PhD thesis, Division of Computer and Information Sciences, Rutgers University, Oct. 2004.

[24] F. Sultan, A. Bohra, and L. Iftode. Service Continuations: An Operating System Mechanism for Dynamic Migration of Internet Service Sessions. In *Proceedings of the Symposium on Reliable Distributed Systems*, Florence, Italy, Oct. 2003.

[25] Sun Microsystems. NFS: Network File System Protocol Specification. *RFC 1094*, Mar. 1989.

[26] G. Wright and W. Stevens. *TCP/IP Illustrated, Volume 2*. Addison-Wesley Publishing Company, Reading, MA, Oct. 1997.

[27] A. Zúquete. Improving the Functionality of SYN Cookies. In *Proceedings of the IFIP Communications and Multimedia Security Conference*, Portoroz, Slovenia, Sept. 2002.