Effects of Ensemble-TCP

Lars Eggert, John Heidemann and Joe Touch

USC Information Sciences Institute
4676 Admiralty Way, Suite 1001
Marina del Rey, CA 90292-6695 USA
{larse, johnh, touch}@isi.edu

December 20, 1999

Abstract*

TCP currently recalculates the state of each connection from a fixed set of initial parameters; this recalculation occurs over several round trips, during which the connection can be less than efficient. TCP control block sharing is a technique for reusing information among connections in series and aggregating it among connections in parallel. This paper explores the design space of a modified TCP stack that utilizes these two ideas, and one possible design (E-TCP) is presented in detail. E-TCP has been designed so that the network transmission behavior of group of parallel E-TCP connections closely resembles that of a single TCP/Reno connection. Simulated web accesses using HTTP/1.0 over E-TCP show a significant performance improvement compared to TCP/Reno connection bundles. This paper is first to evaluate performance using four different intra-ensemble schedulers for different workloads. In one scenario simulating a common case, E-TCP is 4-75% faster than Reno for transmitting the HTML parts of various pages, and 17-61% faster transmitting the whole pages. In the same scenario, reusing cached state speeds up repeated E-TCP page accesses by 17-53% for the HTML parts and 10-28% for the whole pages, compared to the initial access. E-TCP can also be integrated with other proposed TCP extensions (such as TCP/Vegas or TCP/SACK), to further improve performance.

1. Introduction

About 95% of all bytes and around 85-95% of all packets on the Internet [18] are transmitted using TCP [5], which is layered on top of IP [4] and provides a reliable byte-stream between two applications. World-Wide Web (WWW) transactions cause the largest fraction (65-75%) of all TCP traffic [18]. They are transmitted using the Hypertext Transport Protocol (HTTP) layered on top of TCP.

Most web pages consist of several objects (e.g. text and images). A web client will retrieve each object using a separate HTTP transaction. Usually, a client starts some of these transactions in parallel (as opposed to sequentially) to improve user-perceived page rendering time. Parallel transactions allow faster page transmission and avoid head-of-line blocking.

HTTP/1.0 [10] will start a separate, dedicated TCP connection for every web transaction. Thus, during page retrieval, a *bundle* of concurrent TCP connections is active between client and server. Persistent-connection HTTP (P-HTTP) [24], which has been included in HTTP/1.1 [9] since being proposed, allows multiple HTTP transactions to take place sequentially over one TCP connection. However, most clients still open more than one persistent connection to a server to speed up initial page rendering.

Due to the nature of the TCP congestion control algorithms, a bundle of N connections is roughly N times as aggressive as a single connection between the same hosts [6, 35]. While this behavior pays off during the first few round-trip times by transmitting more packets than a single connection would, it often overloads the network later, forcing individual connections to go through slow-start restart phases if packet loss becomes substantial. Over-aggressiveness can also be unfair towards other traffic in the network.

Touch [1] has proposed sharing TCP state among similar connections to improve the behavior of a connection bundle and to improve slow-start for repeated, similar connections. In this paper, we explore the design space for a state-sharing TCP. Ensemble-TCP (E-TCP) is one possible design for such a modified TCP stack and

* This research is partially supported by the Defense Advanced Research Projects Agency (DARPA) through FBI contract #J-FBI-95-185 entitled "Large Scale Active Middleware" and by DARPA and the Air Force Research Laboratory, Air Force Materiel Command, USAF, under agreement number F30602-98-1-0200. The US Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon. The views and conclusions contained in this document are those of the authors and should not be interpreted as necessarily representing the official policies, either expressed or implied, of the Department of the Army, DARPA, the Air Force Research Laboratory, or the U.S. Government. The authors can be contacted at 4676 Admiralty Way, Marina del Rey, CA 90292-6695, or by electronic mail at `larse@isi.edu`, `johnh@isi.edu` or `touch@isi.edu`.

described in detail, focusing on integrated congestion control and ensemble scheduling.

E-TCP has been designed so that the aggregate network transmission behavior of an ensemble closely resembles that of a single TCP/Reno connection. Thus, using an E-TCP ensemble instead of a Reno connection bundle for transmission of some data is equivalent to multiplexing all data over a single Reno connection. In essence, E-TCP is transport-layer multiplexing.

Our work presents the first comparison of performance simulations across a range of different traffic workloads, and the first to consider different scheduling policies for different kinds of traffic using common congestion information. A comparison of E-TCP against HTTP/1.1 with persistent connections shows that while HTTP/1.1 addresses some of the larger performance problems of HTTP/1.0, E-TCP has additional benefits. Several other proposed extensions (including TCP/Vegas, TCP/SACK, TCP/FACK and ECN) can improve TCP performance, and E-TCP works together with most of them to further improve performance.

2. Ensemble-TCP Design

TCP is layered on top of IP (a simple protocol that offers a connection-less, best-effort packet-delivery service) and provides a reliable byte-stream between two applications. TCP connections are independent; most TCP stacks keep state on a per-connection basis in a structure called *TCP control block* (TCB) or an equivalent construct. Some of the information in a per- TCB is not application-pair-based but depends on host-pairs (or even subnet-pairs). One example is round-trip time (RTT). If there are multiple connections between the same hosts, each will independently monitor its transmissions to estimate the RTT between the two hosts, starting with a conservative default value. An alternative scheme is to share RTT information between such parallel connections. Transient performance would improve, because individual connections do not have to rediscover information.

Braden [7, 8] first described how caching TCB state gathered by a previous connection instance can be used to avoid inefficiencies when opening a similar new connection. Touch named this case *temporal sharing* [1] and extended the idea of TCB caching by suggesting that TCB state can be aggregated and shared across similar concurrent connections. This is *ensemble sharing*, and a bundle of concurrent connections sharing TCB information an *ensemble*. E-TCP implements both these ideas to improve TCP service, similar to TCP-INT [6] (see Section 7) which was the first implementation of ensemble sharing.

Variable	Origin	Description
t_rtt	TCB	round trip time
t_srtt	TCB	smoothed round-trip time
t_rttvar	TCB	variance in round-trip time
snd_cwnd	TCB	congestion-controlled window
snd_ssthresh	TCB	snd_cwnd size threshold for slow start exponential to linear switch
tao_cc	T/TCP	latest CC in valid SYN
tao_ccsent	T/TCP	latest CC sent to peer
tao_mssopt	T/TCP	peer's cached MSS
members	E-TCP	list of member TCBs
osegs	E-TCP	list of unacknowledged segments of all members in chronological send order
rbp_mode	E-TCP	rate-based pacing flag

Figure 1. Per-ensemble state variables maintained in an ensemble control block (ECB), with description and place of origin (TCP control block, T/TCP cache or new E-TCP variable).

In the remainder of this section, the design space for a state sharing and caching TCP is explored. Design choices include:

- which state to share/cache
- which connections share state
- how to divide shared state among the connections
- how to maintain cached state
- update policy for shared state

A first design issue is which state E-TCP ensembles will share. Information that is costly to obtain is prime candidate for sharing. For TCP, RTT and congestion control information (a measure of available path bandwidth) are difficult to obtain for new connections: A new TCP connection needs to spend the first few RTTs measuring both, during this phase its performance is usually not efficient, because the initial default values are conservative. Other pieces of end-to-end information for which caching has been proposed are T/TCP's connection counters and the allowed maximum-segment-size (MSS) for connections between a host pair. The path maximum transmission unit (PMTU) proposal [23] suggests caching PMTU values (which the MSS is derived from) based on end point pairs, the E-TCP cache is a suitable place for this. Consequently, E-TCP caches all these values (Figure 1).

Grouping connections into ensembles based on host pairs (as mentioned above) is only one possibility. Deciding which connections belong to an ensemble and thus share state is another design choice. Ideally, this should be based on the type of information shared. For example, RTT information depends on the network path between the two endpoints. Thus, it should be shared by all connections between a host pair. Under the assumption that LAN transmission delays are negligible, this

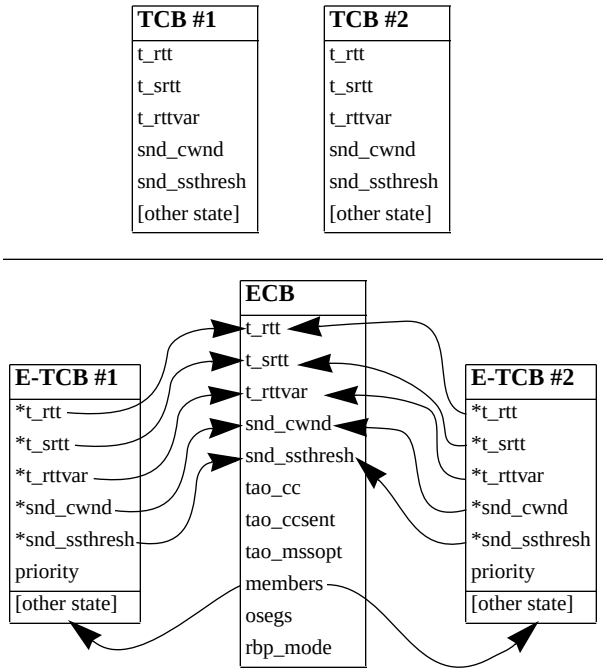

Figure 2. *Top diagram*: Two concurrent standard TCP connections, each using a separate TCB.
Bottom diagram: Two concurrent E-TCP connections sharing state maintained in a central per-ensemble ECB.

may even be extended to sharing between all connections between a subnet pair. Congestion control information, on the other hand, is bottleneck-dependent. Ideally, it should be shared among all connections (on all hosts in a network) sharing a particular network bottleneck. The group of connections causing and/or suffering from congestion could then coordinate their transmissions. This becomes difficult if these hosts are widely dispersed, as the overhead of sharing state may introduce intolerable delays and even increase congestion on the bottleneck. One can also imagine a grouping scheme that places all connections belonging to local or remote applications, users or services in ensembles; even though the immediate usefulness of such schemes remains questionable. E-TCP groups connections based on host pairs.

After deciding which connections share state and what information is shared, a third design decision is how to divide the aggregate state between the ensemble members. Some of that state is in fact a shared resource, such as congestion control information: If one connection uses more, less will be available for others. Many different schedulers for shared resources have been proposed. For E-TCP, a priority scheduler with four different priority assignment policies was implemented for sharing congestion control information. Other pieces of the shared state are not resources but measure properties of the network link, such as RTT and MSS information. No scheduling is required to share them. (See Section 8.)

A less obvious design choice is the degree of state sharing among ensemble members. One possible scheme is to only initialize a new TCB from shared state, then let normal TCP algorithms manage the connection, and update shared state when a connection closes. Small changes to existing TCP stacks can implement this scheme. However, it provides only very limited ongoing sharing, as concurrent connections are still isolated. Thus, E-TCP implements a scheme to continuously share state by aggregating it into a per-ensemble structure and removing it from the individual TCBs altogether (Figure 2).

Temporal sharing (i.e. caching ensemble state over time) requires additional design choices to be made, like how cached state should be maintained while it is not in use. Some state is likely to become invalid relatively quickly as the network environment changes, for example, congestion control parameters. Other state may be more stable and change only if routes between host pairs change, such as RTT and MSS information. For both kinds, older cached connection state is likely to be inaccurate, even if timescales vary for different pieces of information. If cached state overestimates the current network situation, a new connection using it may be too aggressive in its transmission behavior before it can update the state with new measurements. To avoid this problem, cached state should age over time (see Section 7). The specifics of such a cache management scheme are outside the scope of this paper.

3. Ensemble-TCP Operation

As mentioned before, standard TCP maintains all state on a per-connection basis, while E-TCP aggregates some of it per ensemble. This state is kept in an *ensemble control block* (ECB). There are two kinds of ECBs: active ones and cached ones. An *active ECB* has at least one open connection associated with it. A *cached ECB* has no open connections associated with it. Figure 1 lists the state kept in an ECB: RTT and congestion control state are simply aggregated from the individual TCBs. The T/TCP connection cache has been integrated into the ECB, holding connection counters to speed up the initial handshake, as well as MSS information. A list of outstanding (i.e. unacknowledged) packets is maintained in send order for all connections of the ensemble; this enables the integrated congestion control described below. Another list contains pointers to the TCBs of all member connections. Finally, a flag that specifies if rate-based pacing should be done is included in the ECB.

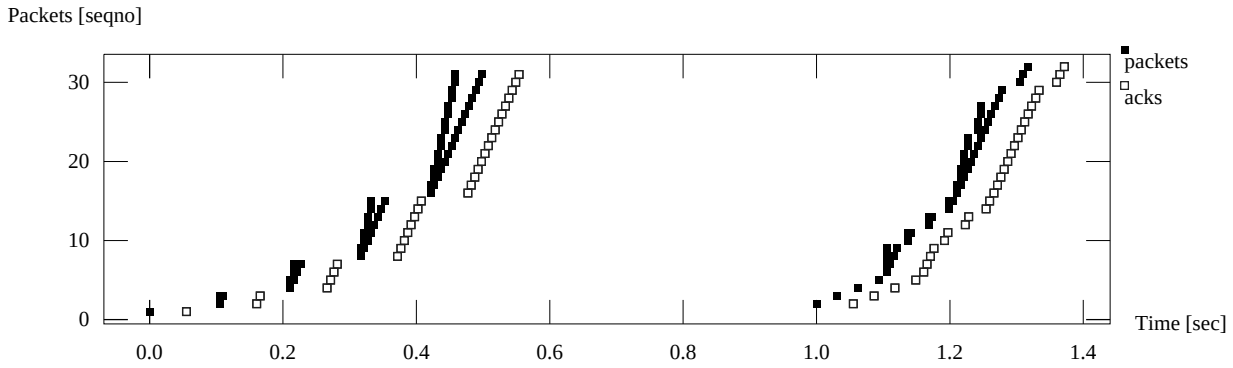

Figure 3. Example of E-TCP temporal sharing.

The TCBs of individual connections are stripped of RTT and congestion control variables. Instead, they simply contain a reference to the ECB of the ensemble they are part of. They also contain a new field that specifies the relative priority of a connection; this is used to allocate a proportional share of the send rate to a connection.

Note that all changes proposed here are sender-side-only, except for the T/TCP-like handshake-avoidance scheme, which requires receiver support. However, since this mechanism interoperates with non-E-TCP peers (by falling back to the standard handshake scheme if a peer does not understand the "connection count" TCP option), receivers need not be updated to deploy E-TCP. The remainder of this section will not discuss operation of mechanisms inherited from T/TCP, these are described elsewhere [7, 8].

3.1. Connection Open

Whenever a connection is opened, E-TCP checks if the new connection can be associated with either an active or cached ECB based on its destination. There are three cases:

1. no active or cached ECB can be found
2. a cached ECB is found
3. an active ECB is found

In the first case, a new ECB is created, and the new connection is associated with it. The fields of the new ECB are initialized to the same values they would have for a new standard TCB. Pacing is turned off, and the list of outstanding packets is empty.

In the second case, the ECB already holds state that was gathered during a prior incarnation of the ensemble. That state is likely to be more accurate than the conservative default values, so it will be reused for the new connection. Slow-start will be skipped; to avoid bursting the network at line-rate due to a large cached congestion window, E-TCP will pace the first packets until the acknowledgment (ACK) clock is going; a technique originally designed to overcome the slow-start restart problem [2]. (E-TCP will also use pacing during slow-start restart). The list of outstanding packets is empty.

Figure 3 shows the packet trace of a scenario where state generated by a previous connection instance (on the left) is reused by a later instance (on the right). Both connections transmit the same number of packets. Note that the first four packets of the later connection are paced based on the previously measured RTT and congestion window, until the connection's own ACK clock takes over. This graph also demonstrates the performance gain achieved by E-TCP's temporal sharing: The later connection needs about 0.18 seconds less than the previous one to transmit the same amount of data.

In the third case mentioned above, an active ECB is found. The new connection is then simply associated with it and will start sending packets when notified by the ensemble scheduler (see section 3.4). The pacing flag and the list of outstanding segments of the active ECB are not changed.

3.2. Connection Close

When a connection closes, there are two possibilities: If it is the last active connection of the ensemble, the ECB associated with the connection will be cached. If it is not, the connection simply stops sending and releases its TCB. Other active connections will continue to use the ECB.

3.3. Ensemble Congestion Control

In standard TCP, the send rate is controlled on a per-connection basis: Each connection has its own *congestion window* (cwnd) and *slow-start threshold* (ssthresh). The congestion window controls the number of packets the connection may have outstanding, while ssthresh controls how long a connection will remain in the slow-

start phase. E-TCP aggregates these values per ensemble. An incoming ACK on any connection increases the aggregate cwnd, while lost packets and lost ACKs will decrease the aggregate cwnd. The additive-increase/multiplicative-decrease algorithm of standard-TCP is kept but applied to the aggregate cwnd and ssthresh. Thus, an ensemble will only be as aggressive as a single standard TCP connection, independent of how many connections are part of it. A single E-TCP connection (if it is part of an ensemble) will thus be less aggressive than a Reno connection, or (if it is the only ensemble member) exactly as aggressive as a Reno one.

As mentioned above, an ensemble maintains a sorted list of sent but unacknowledged packets, each having an associated *skipped ACK* counter. E-TCP adopts this scheme from TCP-INT [6], for which it was originally designed. When a packet is sent, it is appended to the end of the list, and its skipped ACK counter is set to zero. Whenever an acknowledgment (ACK) for any connection of an ensemble is received, E-TCP increases the skipped ACK counters of all packets in the list that were sent before the ACK'ed packet by one. For all packets that have had their counters increased, E-TCP will increase the duplicate ACK counter of the packet's connection by one. All ACK'ed packets are then removed from the list. Standard TCP per-connection ACK rules are performed, which may trigger retransmits or close the congestion window.

Normally, the scheduler determines which connection may send a packet according to the ticket distribution of the ensemble members. However, if a connection needs to retransmit a packet, it is given priority, because it may not have used its assigned share due to packet loss. (Not having to wait to be scheduled might also avoid a costly retransmission time-out in some cases.) The retransmitted packet is moved from its original spot in the list to the end, and its skipped ACK counter is reset to zero. If a retransmission timer expires, all packets are removed from the list.

Note that this congestion control scheme only works well with immediate ACKs. With delayed ACKs, unnecessary retransmissions are scheduled (because of skipped ACKs), hurting performance. TCP-INT [6] has a more elaborate retransmission variant that supports delayed ACK receivers and can be adopted for E-TCP.

3.4. Ensemble Scheduling

The previous section explained how the aggregated congestion control state is being managed. Since E-TCP maintains the ACK clock per-ensemble, a new mechanism to schedule send operations for connections is needed. Each ensemble has a priority-based scheduler based on stride scheduling [3], allowing clients to hold a number of tickets determining the relative share of a the congestion window that will be allocated to them. The scheduled quantum has a granularity of one packet.

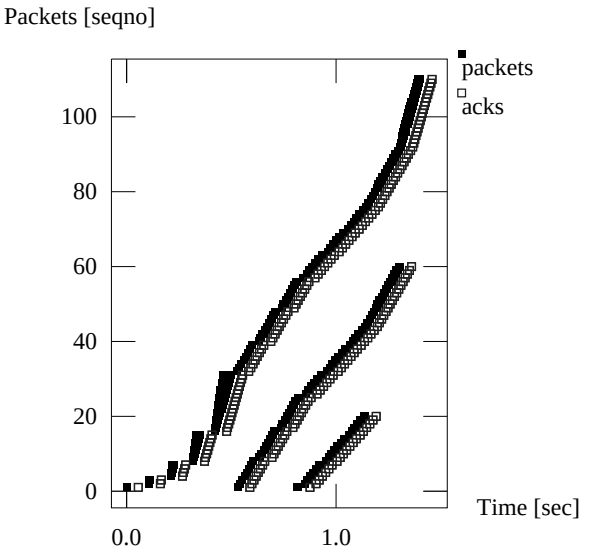

Figure 4. Example of E-TCP ensemble sharing using fair scheduling.

Figure 4 shows an example of an ensemble using fair scheduling (described in Section 4). The first connection starts at 0 seconds and uses the whole congestion window since it is the only ensemble member at that time. Note that the transmission pattern looks exactly like a standard TCP connection. When the second connection starts at 0.5 seconds into the trace, the two ensemble members immediately split up the aggregate congestion at a ratio of 1:1. This is illustrated by the decline in slope of the first connection's curve. The third connection starts at 0.8 seconds, and each connection now uses a third of the congestion window. After the third connection finishes (at 1.24 seconds), the first and second again split the congestion window at a ratio of 1:1; after the second one finishes (at 1.41 seconds), the first connection uses all of the congestion window.

4. Ensemble Sharing Evaluation

E-TCP was implemented for a beta version of the ns-2.1 network simulator [11], and its performance was evaluated by measuring web accesses using HTTP/1.0 over both E-TCP and TCP/Reno to the following five web pages (modeled after real pages found on the web in summer 1997):

Text Page:

- 1 large HTML part (29K)
- 3 medium images (7-13K)

Map Page:
- 1 small HTML part (5K)
- 3 small images (1-3K)
- 1 large image (67K)

Graphics Page:
- 1 medium HTML part (7K)
- 9 small images (1-3K)
- 4 medium images (9-12K)

Frames Page:
- 4 small HTML parts (1-2K)
- 1 medium HTML part (7K)
- 1 large HTML part (83K)
- 14 small images (1-3K)
- 7 medium images (5-19K)

Java Page:
- 1 medium HTML part (8K)
- 7 small images (1-2K)
- 3 medium images (4-11K)
- 14 small Java parts (1-3K)
- 11 medium Java parts (4-18K)

The main performance criterion during these experiments is *HTML transmission time*. The HTML parts of a page are important for two reasons: First, they normally contain the content the user requested (embedded graphics are often used for presentation purposes only). Second, HTML parts drive the page rendering process, since all links are contained within them. Thus, a client cannot request embedded objects before (at least part of) the HTML parts of a page have been received. In Figure 5, the solid bars plotted in the foreground depict HTML transmission times. A secondary performance criterion is *total transmission time*, which is plotted using striped bars in Figure 5. (For cases without a striped bar, an HTML-carrying connection finishes last and the HTML delivery time is also the total transmission time).

We measured one simulated page download to obtain HTML and total transmission times for each page and scenario. Since the setup does not contain any random elements, repeating a simulation run always yields the exact same result - thus, no error bars or confidence intervals are included in the graphs.

The simulated link between the client and server has a latency of 50ms, a bandwidth of 800Kb/s and an MSS of 512 bytes. A simple FIFO queue is used to model aggregate router behavior, queue buffer space was a parameter (unlimited and 10 packets). There is no other traffic present, all losses are thus due to self-interference.

When this work was begun, the ns simulator did not support true two-way TCP traffic and also did not include connection setup (SYN) and teardown (FIN) packets when simulation TCP. Thus, 1.5 RTT are added to the measured times to reflect the initial handshake. Another 0.5 RTT is added to the results to compensate for not sending the request message. (Requests typically fit in one packet [19], so their transmission time is negligible.) This implicitly assumes request messages are never dropped, and server-side processing time is zero.

To compare the performance of TCP/Reno and E-TCP, page accesses for all five test pages over each of the two transport protocols are simulated. Two parameters are modified: minimum path queue limit (10 packets and unlimited) and transactional concurrency (4 parallel connections, because a widely-used browser uses that number, and unlimited), resulting in four distinct configurations:

- *Model Scenario*: unlimited path queue, unlimited concurrency
- *Realistic Client Scenario*: unlimited path queue, concurrency limit of 4 connections
- *Realistic Network Scenario*: path queue of 10 packets, unlimited concurrency
- *Realistic Scenario*: path queue of 10 packets, concurrency limit of 4 connections

The first case simulates page accesses over an model network using a model client - the baseline case. The second case limits the concurrency of the client, which most real clients do, modeling the best network situation a realistic client could encounter. The third case simulates a model client over a realistic network having a limited path buffer. The final configuration is closest to the real-world case, where a concurrency-limited client issues requests over a network with a path queue limit.

Four different scheduling policies are simulated for E-TCP; all are based on the priority scheduler described previously:

- *Content-dependent*: Tickets are assigned to connections based on the content of the response message they transmit. This ticket distribution was chosen: HTML 12, Java 6, image 3, FTP 1.
- *Ticket decay*: Connections start with 16 tickets. After sending the 5th, 10th, 20th and 40th packet, half of a connection's tickets decay.
- *Fair*: Each connection holds one ticket.
- *HTML-pre*: A variation of content-dependent scheduling. HTML-carrying connections hold an infinite number of tickets, thus preempting all non-HTML connections. Other connections hold tickets according to the content transmitted over them, as described above.

4.1. Model Scenario

In the "model" scenario, the network has unlimited buffering capacity, meaning no packets are ever dropped. The model client simulated is not limited in issuing concurrent requests. This scenario rewards aggressive behavior, so the Reno bundle is expected to outperform E-TCP. It is the baseline scenario, showing maximum obtainable performance for each case.

The solid bars in the top chart of Figure 5 show that E-TCP using HTML-pre scheduling is fastest in delivering HTML for all five pages; the improvements over Reno range between 6% and 46% for the five pages. E-TCP's other three scheduling mechanisms from 2% to 60% slower than Reno in all but one case (graphics page transmitted by E-TCP with content-dependent scheduling) - even there, the improvement is a minor 3%. HTML-pre performs well, because even though no packets are dropped, all other scheduling mechanisms (and Reno) send non-HTML packets interleaved with HTML ones as requests arrive at the server. Performance differences are thus due to queueing delay only.

Looking at total transmission times (striped bars), note that E-TCP needs from 2% to 26% more time than a Reno bundle for four pages, regardless of the scheduling employed. Even for the fifth page (map page), the Reno bundle is only 4% slower than E-TCP. Reno's aggressiveness clearly pays off in a scenario with unlimited resources.

Also note that all E-TCP scheduling mechanisms have the same total delivery time for the same page. This illustrates E-TCP's integrated scheduling.

4.2. Realistic Client Scenario

This scenario (second graph in Figure 5) models a realistic client that limits the number of parallel requests it will issue. Clients use multiple connections to increase transmission throughput. A concurrency limit bounds the aggressiveness of such a connection bundle, thus lowering the chance of substantial losses as the otherwise overaggressive bundle saturates the bottleneck. Since the network still has unlimited buffering in this scenario (rewarding aggressiveness) but the Reno bundles are now less aggressive, they are expected to take longer for total transmission times compared to the previous scenario. Compared to the first scenario, the performance impact of an unnecessary connection limit can be observed.

An E-TCP ensemble will (by design) always be as aggressive as a single Reno connection. Thus, an external scheme to control aggressiveness (like the concurrency limit) is not necessary to improve E-TCP's behavior (also see Section 4.3). Nevertheless, a concurrency limit of four connections was also enforced for E-TCP in the interest of keeping the experiment simple.

As in the previous scenario, the network has unlimited queuing capacity and there are no packet losses. Performance differences for the same combination of TCP flavor and scheduling mechanism compared to the previous scenario are thus due to changes in transmission delay only. A good example is the (now concurrency-limited) Reno bundle's HTML delivery time for the java page: It is about twice as fast compared to the previous experiment. This is due to the fact that at most three other connections (carrying non-HTML data) will be sending while HTML is being transmitted (and so preempt it), while tens of them were active during the previous experiment. Thus, the server will send less non-HTML packets interleaved with HTML ones, and so HTML transmission times will be reduced. Also, each connection ends faster - but later connections may start later, since they have to wait for one of the four "connection slots" in the bundle to become available. The frames page, on the other hand, is a special case: It consists of many objects, but by far the largest ones are HTML. Throughput plays a larger role here, and the gains due to improved delay are only minor (12%).

Limiting concurrency is only effective if a page consists of many objects. The text and map pages do not, and performance (both HTML transmission and total time) is identical to the previous scenario. These two cases will not be addressed further below. For the remaining three pages (frames, graphics, java), E-TCP with HTML-pre scheduling outperforms all other scheduling mechanisms and the plain Reno bundles for HTML performance. It is between 4% and 21% faster than a Reno bundle (depending on the page). E-TCP's three other scheduling schemes are from 2% to 27% slower than Reno in all but one case: E-TCP with content-dependent scheduling is 4% faster than Reno for the frames page.

Note that HTML delivery times for E-TCP with HTML-pre scheduling are unchanged from the previous experiment. Since HTML-pre scheduling lets HTML traffic preempt other content, the concurrency limit is meaningless during the HTML delivery phase.

For the three pages (graphics, frames, java) affected by the concurrency limit (because they consist of many parts), E-TCP ensembles transmit the whole page from 8% to 29% faster than Reno bundles in this scenario. E-TCP benefits from sharing state, because connections starting later during the page retrieval period can take advantage of a wider congestion window opened by prior ensemble members, while later Reno connections

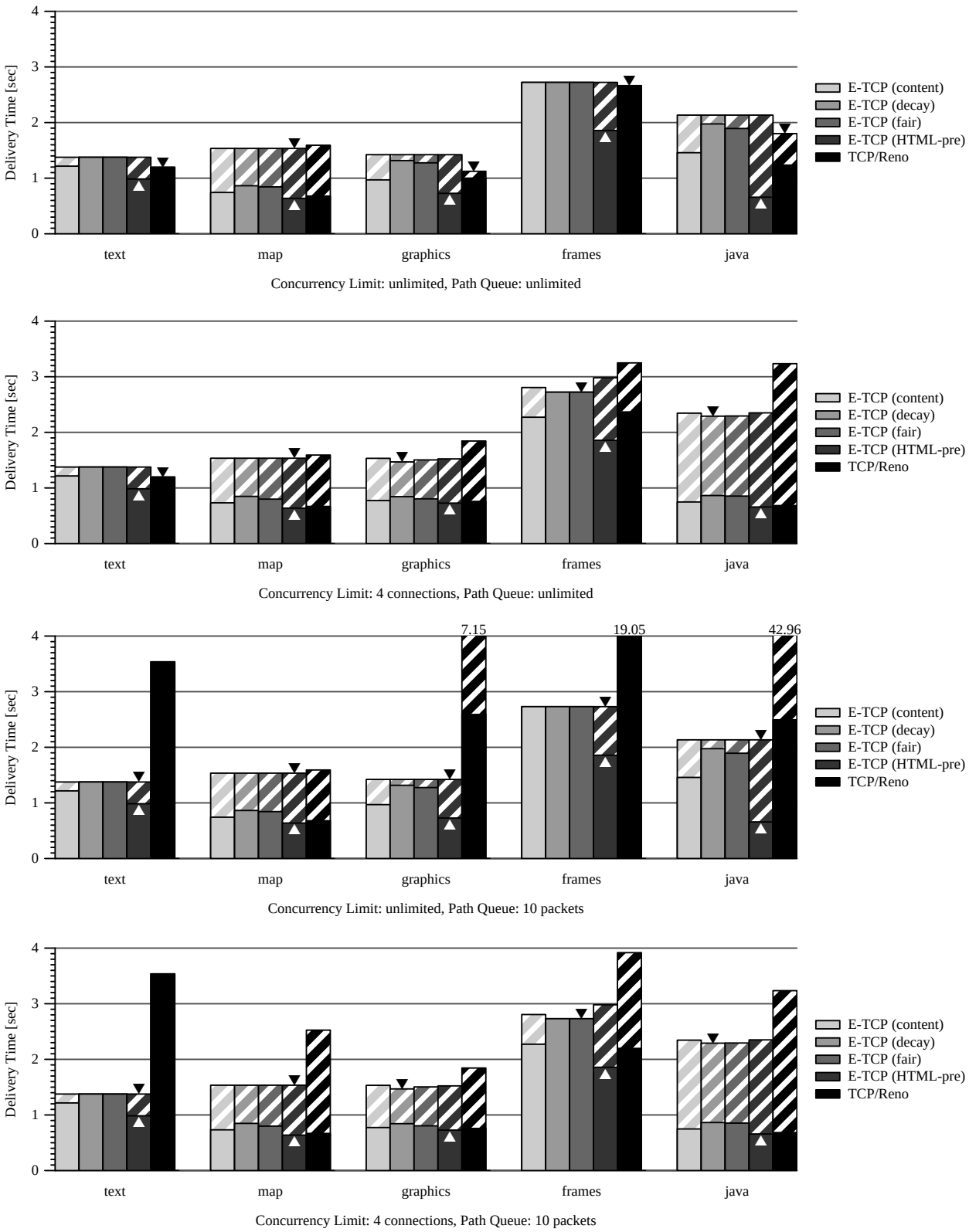

Figure 5. Comparison of HTML (solid bars) and total (striped bars) transmission times for five different pages (text, map, graphics, frames, java) over TCP/Reno and E-TCP, the latter with four scheduling mechanisms. Plotted for four combinations of concurrency limit and path queue limit. Bars with numbers above were cut off but continue to that value. Black triangles indicate fastest total transmission time for each page, white triangles indicate fastest HTML transmission time.

have to slow-start. In other words, the aggregate E-TCP congestion window is larger than the aggregate Reno congestion window during later phases of a page retrieval.

Reno total transmission times (striped bars) are between 22% and 80% higher (for the three pages affected by the concurrency limit) than in the previous scenario, but only up to 10% for E-TCP. This is expected, because the concurrency limit causes a Reno bundle to be less aggressive, and thus slower. For E-TCP, the difference is due to the interaction between limiting concurrency and rate-based pacing: E-TCP turns on pacing for an ensemble when all its connections are half-closed (i.e. have sent all data and are waiting for the last ACK). This mechanism prevents a situation where a new connection will burst the network with one congestion window's worth of packets when all other ensemble members are half-closed. Pacing slows down the first few packets of the new connection, thus causing the performance differences seen in the diagram. Another interesting observation is that total transmission times for E-TCP is no longer identical for the different scheduling mechanisms. Again, this is due to the interaction described above, because different scheduling schemes cause a different number of packets to be paced at different times, and thus transmission times change slightly. As mentioned before, a concurrency limit is not needed for E-TCP, so both these anomalies can easily be eliminated.

4.3. Realistic Network Scenario

Here (third graph in Figure 5), a model client issues requests over an realistic network. As in Section 4.1, the model client has no concurrency limit and will issue requests as it encounters links while parsing the HTML parts of a page. This scenario illustrates that a concurrency limit is essential for TCP/Reno to achieve acceptable performance in the presence of finite router buffering, especially for pages which consist of many parts. E-TCP, on the other hand, should perform much better than Reno in all cases, as a concurrency limit is not necessary. In fact, a concurrency limit may lower E-TCP's performance by interfering with the integrated scheduling mechanism by delaying requests.

The results for the map page are unchanged from the two previous scenarios, illustrating that for pages consisting of a few number of parts, a Reno bundle can sometimes achieve good performance. Thus, the map page will not be discussed further. On the other hand, for the text page, which also consists of few parts, Reno takes almost three times as long, because of multiple losses. This shows that for some pages, even a concurrency-limited Reno bundle is still too aggressive.

E-TCP with HTML-pre scheduling is again 6-73% faster than Reno in transmitting the HTML parts of the pages. It is also always faster compared to the other three scheduling schemes, which perform from 27% worse to 65% better than Reno.

For total transmission times, Reno fares much worse, being 2.5-20 *times* slower than E-TCP. This difference in performance between HTML and total times is caused by Reno's extreme aggressiveness during later phases of a page retrieval period, which leads to extensive losses (14% for the java page) and retransmission time-outs.

Note that E-TCP times are unchanged from the first scenario (Section 4.1). As explained above, E-TCP does not require a concurrency limit to perform well. In fact, a concurrency limit sometimes decreases E-TCP's performance, as can be seen be comparing the total transmission times of E-TCP with HTML-pre scheduling to the previous case: it is now up to 10% worse for the different pages.

4.4. Realistic Scenario

The last scenario (bottom graph in Figure 5) shows transmission times for a case where the router queue was limited to 10 packets (as in the previous scenario) and the client enforced a concurrency limit of 4 connections for both Reno bundles and E-TCP (as in the second scenario).

As in the three previous scenarios, E-TCP with HTML-pre scheduling outperforms all other scheduling mechanisms and plain Reno bundles for HTML transmission of all five pages. It is from 4% to 75% faster than Reno bundles (depending on the page). E-TCP's other schedulers do not perform as well, they are in the range of 2% to 27% slower than Reno; except for the text page, where the Reno connections had multiple losses during the transmission of the HTML part, here the three schedulers are 61% to 65% faster than Reno (but still slower than HTML-pre). For total delivery times, E-TCP is between 17% and 61% faster than a Reno bundle, depending on the page transmitted.

For Reno, two interesting observations can be made: First, for the frames page, HTML delivery time is lower than during the second scenario (which had no path queue limit). A packet trace shows that this is because several image connections have multiple losses, which cause fewer image packets to become interleaved with HTML packets in the router queue. Second, for the text page, delivery time is unchanged from the previous scenario (where no concurrency limit is enforced.) This shows that for some pages, a concurrency limit alone is not sufficient to improve Reno's performance.

5. Temporal Sharing Evaluation

To quantify the performance improvements of temporal sharing, a repeated access to a page (see Figure 6) is simulated. The pages and network configurations are unchanged from the ensemble sharing experiments (Section 4). However, the simulation now consists of two consecutive accesses to each page to model a situation where a user requests multiple pages from one server. HTML and total delivery times are measured for both accesses. One assumption is that the network parameters will not change between the accesses, thus the cached state is reused for the later connection without being aged or validated. This means the performance gains we measured are an upper bound; performance may be different if the network characteristics have changed. Another limitation is that only repeated accesses to the same page are simulated. More extensive experiments should include repeated accesses to different pages, as well as varying background traffic (see Section 8).

The ensemble experiments have shown that HTML-pre is the best of the four simple E-TCP scheduling schemes proposed. Thus, only E-TCP with HTML-pre will be compared against plain Reno in this section.

For each page, the graphs show HTML and total delivery times for the initial and the repeated access. Because TCP/Reno does not share state, the two times are always identical, and the diagram includes only one Reno access.

In the "model" scenario (top graph of Figure 6), the repeated page access using E-TCP is from 48% to 82% faster than a Reno bundle in delivering HTML for the five pages simulated, and between 4% and 72% faster overall. This means that even in an ideal Reno scenario (aggressiveness is rewarded), repeated page accesses to the same server are faster using E-TCP. The benefits of temporal sharing alone are demonstrated when comparing the first E-TCP access to the repeated one, the latter is 17% to 53% faster (depending on the page) in delivering HTML, and between 11% and 28% faster overall. Also, the repeated E-TCP access avoids the three-way handshake, reducing the user-perceived idle time at the beginning of the retrieval by 1 RTT. Thus, the system feels more responsive to a user.

In the "realistic client" scenario (second graph in Figure 6), a concurrency limit of four connections is enforced, but the network still has unlimited buffering. E-TCP's repeated access is between 35% and 55% faster than Reno for HTML (depending on the page) and from 17% to 37% faster overall. Temporal sharing benefits alone are illustrated by the increase in performance between E-TCP's first and repeated access, which lies between 17% and 53% (depending on the page) for HTML transmission time, and 10% to 28% overall.

For the "realistic network" scenario which has no concurrency limit, but a limit path queue, the repeated page access using E-TCP is 63% to 87% faster than a Reno bundle in delivering HTML and between 27% and 96% faster overall. Looking at benefits achieved by temporal sharing alone, E-TCP's repeated access is between 15% and 53% faster than the initial one for HTML, and from 8% to 28% faster overall, depending on the page accessed.

For the "realistic" scenario (bottom graph in Figure 6), the network had a path queue limit of 10 packets, and only four concurrent connections were allowed. Here, E-TCP's repeated access is between 30% and 82% faster than a Reno bundle for HTML parts, and from 32% to 72% faster overall. Temporal sharing speeds up the repeated E-TCP access between 17% and 53% for HTML, and from 10% to 28% overall, compared to the initial one.

These results show that repeated E-TCP accesses (with HTML-pre scheduling) are faster in transmitting HTML and the total pages for all simulated scenarios both compared to the initial E-TCP request and a Reno bundle transmitting the same data. This indicates that temporal sharing of connection state can substantially increase performance of web accesses, at least in some scenarios.

6. Discussion

The results presented in the previous two sections show that E-TCP with HTML-pre scheduling is fastest in transmitting HTML in all simulated scenarios, even ones that reward aggressiveness and thus cater to Reno bundles. In scenarios with limited network buffering, E-TCP also outperformed Reno for total page transmission times.

Recall that the performance differences between the four E-TCP schedulers (in the second and fourth graph in Figure 5) are caused by an interaction between pacing and the concurrency limit. The limit was enforced to simplify the simulation, but which in fact slightly decreases E-TCP performance for some cases. Without it, HTML-pre scheduling is also fastest for total delivery transmission times (E-TCP performance will then be as shown in the third graph of Figure 5).

From both these observations follows that HTML-pre scheduling is clearly the best choice according to the performance criteria defined at the beginning of Section 4. However, other schedulers may perform better for other traffic patterns. One of the novel features of E-TCP is that different schedulers can be easily imple-

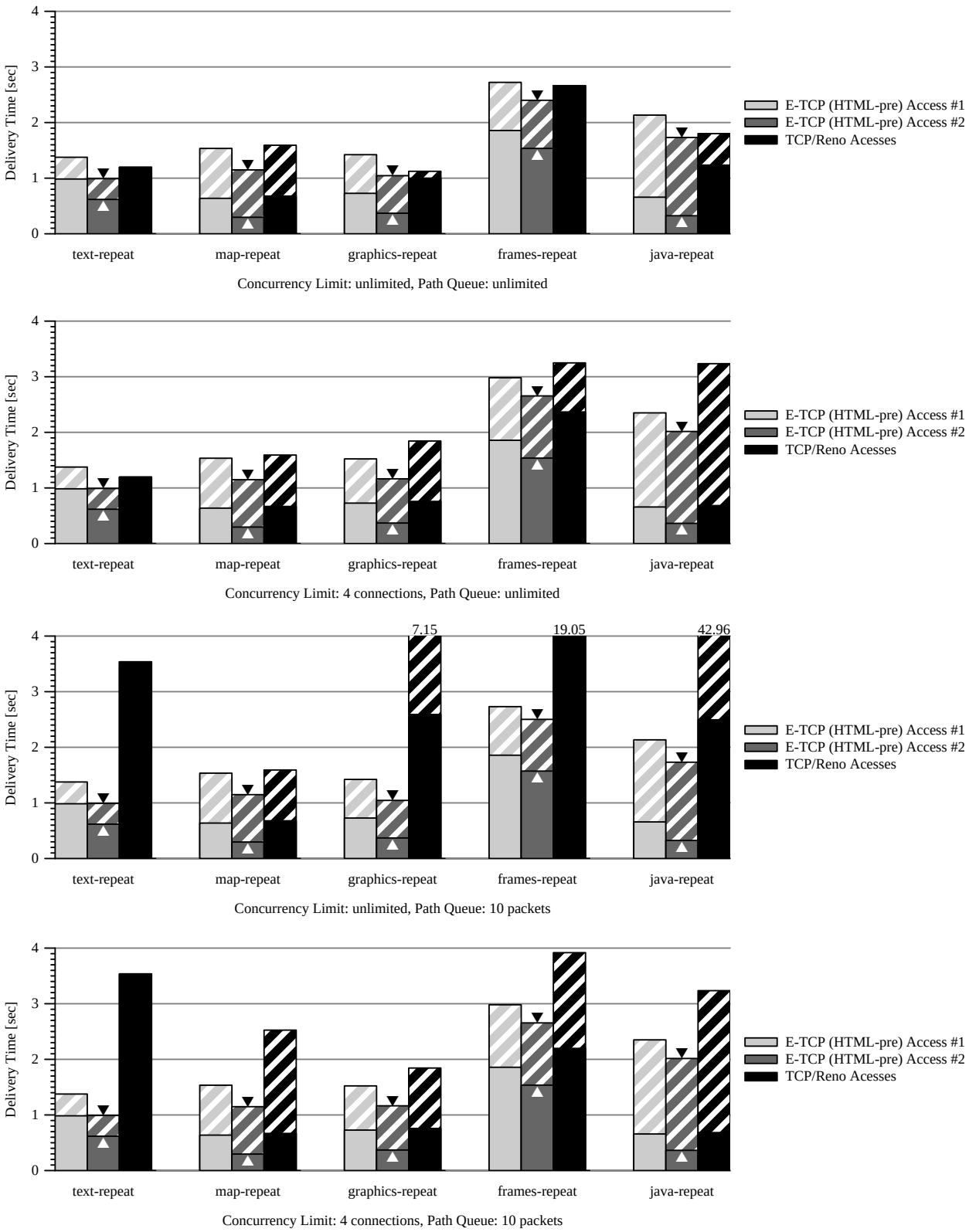

Figure 6. Comparison of HTML (solid bars) and total (striped bars) transmission times for repeated accesses for five different pages (text, map, graphics, frames, java) over TCP/Reno and E-TCP with HTML-pre scheduling. Plotted for four combinations of concurrency limit and path queue limit. Bars with numbers above were cut off but continue to that value. Black triangles indicate fastest total transmission time for each page, white triangles indicate fastest HTML time.

mented, because congestion control has been decoupled from packet scheduling.

The third and fourth graph of Figure 5 illustrate that a concurrency limit is vital to achieve acceptable Reno performance when transmitting pages with many parts. Without one (third graph), performance is substantially degraded. Even with a concurrency limit, performance cannot be improved in all cases (one example for this is the text page in the fourth graph). One reason is that even with a limit of four concurrent connections as during some of our simulations, a Reno bundle is still roughly four times as aggressive as a single Reno connection, and can thus overload the network.

Results in Section 5 show that E-TCP's temporal sharing can substantially improve performance for repeated accesses to the same server, compared to both an initial E-TCP request, and more importantly, a Reno bundle.

As mentioned in the introduction, the network transmission behavior of an E-TCP ensemble closely resembles that of a single Reno connection. Using E-TCP for web traffic is very similar to using HTTP/1.1 (see Section 6.1). Thus, even though our simulations were limited in scope (five pages, four network scenarios, no background traffic) there is strong indication that the idea of TCP state sharing can substantially increase performance under a more varied set of conditions. While more extensive experiments are certainly useful, E-TCP's behavior in other scenarios can be extrapolated from well-understood TCP/Reno behavior.

6.1. Interaction with HTTP/1.1

All experiments have used HTTP/1.0 to transmit web transactions. Many of its performance problems are due to the mapping of one web transaction to one TCP connection. The more recent HTTP/1.1 includes support for persistent TCP connections that can be reused for several web transactions. The connection reuse of HTTP/1.1 is very similar to E-TCP's ensemble sharing. The largest difference is that E-TCP allows multiple web transactions to occur in parallel, while HTTP/1.1 responses must complete only in the order requested. Thus, E-TCP offers packet-scale scheduling, while HTTP/1.1 allows only transaction-scale scheduling. A similar effect to packet-scale scheduling can be produced with HTTP/1.1: Clients need to issue several requests for parts of an object, instead of one request for the whole object. If the requested parts are small enough and clients issue them in similar ways to E-TCP, the effect could be comparable. Drawbacks of this technique are changes required to clients (E-TCP is mostly sender-side) and additional load put on servers (most spawn a new process per request).

Overall transmission times of HTTP/1.1 and E-TCP should be exactly equal: Both will use one connection (a logical connection in E-TCP's case) to transmit a page, reusing it to transmit all components. However, multiple HTTP/1.1 connections are not integrated (and thus more aggressive) than multiple E-TCP connections, which will always cooperate. Thus, HTTP/1.1 aggressiveness may be higher than E-TCP's if clients use multiple persistent connections.

Another advantage of E-TCP is that it is a transport-level improvement: all TCP traffic will benefit. HTTP/1.1 will only work for web traffic, because it is an application-level protocol.

Furthermore, HTTP/1.1 does not have temporal sharing, which improves transmit times for repeated transactions. Instead, it allows to speculatively keep a connection open after a transaction finishes, with the intent to reuse it for possible future accesses. This triggers burstiness in TCP's slow-start restart algorithm after idle time [26]. Several proposed TCP extensions (including rate-based pacing employed by E-TCP) address this known issue.

6.2. Comparison with other TCP Extensions

A number of modifications have been proposed to TCP/Reno to improve its performance, including:

- TCP/Vegas [15], which replaces Reno's RTT management with a finer-grained scheme and modifies its congestion control algorithms to make decisions based on throughput rates
- TCP Selective Acknowledgments (TCP/SACK) [17], which improves retransmissions by explicitly notifying a sender about segments which were received
- TCP Forward Acknowledgments (TCP/FACK) [13], which builds on TCP/SACK to more accurately estimate the amount of data outstanding in the network
- Explicit Congestion Notification (ECN) [14], which uses a router-set flags in IP headers to indicate network congestion, and proposes algorithms on how TCP should do congestion control if segments with set ECN flags are encountered
- Allman et. al. [28] propose to increase TCP's initial congestion window to more than one packet
- Rate-halving [29] is a technique to space transmissions after an indication of congestion to avoid bursts

E-TCP is orthogonal to these improvements - except for TCP/SACK (see below). The idea of state sharing does not depend on specific algorithms to manage shared state. We chose to implement standard Reno algorithms, causing an ensemble to behave like one Reno connection. However, this can easily be changed to implement

different state management algorithms for the aggregate ensemble state, e.g. TCP/Vegas' RTT algorithms, or any combination of the algorithms above.

TCP/SACK receivers provide more detailed feedback about received segments to their senders using TCP options. It is possible to integrate TCP/SACK with E-TCP's retransmission scheme: One idea is to more carefully manage an ensemble's list of outstanding segments based on the additional information provided by TCP/SACK. However, the details of this integration need to be carefully studied.

7. Prior and Related Work

7.1. Integrated Congestion Management

TCP extensions for transactions (T/TCP) have been designed to improve (handshake) performance for repeated connection instances by caching limited TCP state (connection counters). Braden [7, 8] also briefly mentions how caching other state (RTT, MSS, congestion window) may further improve T/TCP performance.

Touch generalized the idea of TCP state sharing [1] and differentiated between temporal sharing (caching over time, as done by T/TCP) and ensemble sharing (concurrent use of aggregate state). In addition to the fully integrated sharing scheme implemented for E-TCP, he describes read-on-open and write-on-close semantics for the aggregate state, leading to less coupling among the members of an ensemble. E-TCP builds on both Braden's and Touch's proposals.

Balakrishnan et. al. [6] have implemented TCP-INT, an extension to TCP that integrates congestion control over a bundle of TCP connections. E-TCP reuses their congestion control scheme (based on skipped ACKs). TCP-INT does not support temporal sharing, so each new connection will go through the standard handshake and slow-start phase, while E-TCP caches ECBs and reuses them when a new ensemble instance is created. This allows E-TCP to avoid both the handshake and slow-start phase, resulting in improved performance for repeated connections. TCP-INT does also not support multiple scheduling schemes.

A more recent proposal by Balakrishnan et. al. [25] is the per-host Congestion Manager, which maintains network statistics on a per-receiver basis, performs congestion avoidance and control and schedules transmissions. It is similar to E-TCP's ensemble cache in that network parameters are cached and aged over time, but offers an API for applications and application-level protocols. It also supports congestion management for both UDP and TCP, however it requires a new packet header between IP and the transport layer.

TCP Fast Start [30] utilizes a cache of TCP state information (containing congestion window and RTT information), much like E-TCP temporal sharing does, to improve the start-up performance of new connections. TCP Fast Start sends more packets than slow-start would but marks those extra packets with a drop-preference flag. This avoids bursting the network, but requires router support - unlike E-TCP, which uses pacing to achieve a similar effect. Also, TCP Fast Start does not address ensemble sharing, i.e. how cached information is utilized by concurrent connections.

7.2. Other Prioritizing Systems

Crowcroft and Oechslin [12] propose MulTCP, which allows users to assign weights (priorities) to different connections, which will then receive a proportional share of the available bandwidth along a congested path. A MulTCP connection of weight N is made N times more aggressive than a standard Reno connection by introducing N as a factor into TCP's congestion control and slow-start algorithms. In a sense, MulTCP is the inverse of E-TCP: A single E-TCP connection is 1/Nth as aggressive as a single Reno connection (with N being the number of connections in its ensemble).

At the network-level, several proposals have been made to prioritize traffic. One such proposal is to extend IP for integrated services [32]. In this scheme, receivers initiate a resource reservation request to receive a guaranteed service commitment with the Resource Reservation Protocol (RSVP) [33]. A second proposal is to extend IP to support differentiated services [34]. This approach allows high priority traffic to take precedence over existing traffic on a per-packet basis. Compliant routers will respect priorities in their queueing and forwarding decisions. Both these systems may not interact well with E-TCP in some situations: One of the basic assumptions of state sharing is that the network characteristics are similar for all ensemble members. If members of an E-TCP ensemble fall into different service classes, this is no longer true and aggregating their different views of the network will most likely lead to less than efficient behavior. Additional work is needed to integrate network-level scheduling with transport-level scheduling.

7.3. Application-Level Multiplexing Systems

Several proposed mechanisms allow application-level multiplexing of byte-streams onto a single connection: The Transaction Internet Protocol (TIP) [20] includes the TIP Multiplexing Protocol (TIP/TMP). SCP [22] is a similar proposal from Spero, as is SMUX [21]. All three schemes add a layer of indirection to TCP by wrapping chunks of data from different logical connections with a demultiplexing header (subdividing the sequence num-

ber space). E-TCP does not incur the overhead of these extra headers, as connections remain separate entities at the application level. Also, E-TCP does not require the application to be modified to support the multiplexing effect.

7.4. Congestion Window Management Systems

The problem of reusing and adjusting the congestion window of a connection that has been idle for some time [31] is similar to the issue of managing cached ensemble state: In both cases, reuse of state possibly outdated information needs to be addressed. Pacing has been proposed as a means of minimizing burstiness [2] and is used in E-TCP, while Handley et. al. [27] propose a simple scheme that ages the congestion window by halving it once for every RTT that a connections is idle. This solution could be adopted to manage a cached ensemble's congestion window. However, E-TCP also requires solutions to manage cached RTT and MSS state, which is not addressed in Handley's paper.

8. Future Work

E-TCP caches measured network properties over time. Simply reusing cached state after an idle period may be problematic if those network properties (e.g. available path bandwidth) have changed. The current design of E-TCP does not address this critical issue yet (see Section 2). The impact of different cache management strategies (e.g. ageing state) on other network traffic must be carefully studied in real network configurations to better understand E-TCP's network dynamics.

Section 2 described different ways of grouping connections into ensembles. A possible extension of the grouping scheme would be to place connections in ensembles per piece of shared information. For example, a connection could share RTT information with some ensemble of other connections, and congestion control information with another ensemble of connections. While this approach is more general, its benefits compared to the simpler scheme we chose needs to be investigated.

Reno's RTT measurement algorithm uses one measurement per round-trip to estimate the RTT for a connection. E-TCP in its current form simply aggregates all measurements member connections take for an ensemble. This means that for an ensemble with N members, about N RTT measurements will be taken during one round-trip. The implications of this change need to be investigated in more detail.

The current design of E-TCP does not aggregate retransmission timers, each member connection has one and manages it independently of the others. Also, E-TCP implements the Nagle algorithm on a per-connection basis only. Additional research is needed to determine if and how E-TCP would benefit if these techniques were extended to the whole ensemble.

Ensemble sharing works under the assumption that all connections receive a similar network service. Section 7.2 described how proposals for establishing different levels of network service will conflict with this. Another case where connections between the same hosts may encounter different network service is if one (or both) of the end hosts lies behind router that does network address translation (NAT). To its peer, all connections leaving the NAT router will look like they originate at one host (the NAT router), while in fact they may originate at different physical machines. If the network behind the NAT router has sections with widely different characteristics, E-TCP operation will be affected.

9. Conclusion

This paper shows how a modified TCP stack can utilize the ideas of caching and reusing TCP state information for repeated connection instances, and share such information among concurrent flows to improve performance. The design space of such a TCP variant was explored and one possible design (E-TCP) was described in detail.

Simulated web accesses using HTTP/1.0 over E-TCP (and four different intra-ensemble schedulers) show a significant performance improvement compared to standard TCP/Reno connection bundles for four different network scenarios. In one scenario simulating a common case, E-TCP is 4-75% faster than Reno for transmitting the HTML parts of various pages, and 17-61% faster transmitting the whole pages. In the same scenario, reusing cached state speeds up repeated E-TCP page accesses by 17-53% for the HTML parts and 10-28% for the whole pages, compared to the initial access. E-TCP can also be integrated with other proposed TCP extensions (such as TCP/Vegas or TCP/SACK), to further improve performance.

We have also discussed how the results compare against HTTP/1.1 as well as several proposed TCP extensions (TCP/Vegas, TCP/SACK, etc.) and find that most of those proposals could be integrated into E-TCP.

Acknowledgments

The ns implementation of E-TCP and the simulation scripts used for experiments in this paper are available online at http://www.isi.edu/~larse/software/.

References

[1] J. Touch. “TCP Control Block Interdependence.” Internet Request For Comments, RFC 2140. April 1997.

[2] V. Visweswaraiah, J. Heidemann. “Improving Restart of Idle TCP Connections.” Technical Report 97-661, University of Southern California, November, 1997.

[3] C. Waldspurger, W. Weihl. “Stride Scheduling: Deterministic Proportional-Share Resource Management.” Technical Memorandum MIT/LCS/TM-528. MIT Laboratory for Computer Science, June 22, 1995.

[4] J. Postel. “DARPA Internet Protocol Specification.” RFC 791, Internet Request For Comments. September 1981.

[5] J. Postel. “Transmission Control Protocol.” RFC 793, Internet Request For Comments. September 1981.

[6] H. Balakrishnan, V. Padmanabhan, S. Seshan, M. Stemm R. Katz. “TCP Behavior of a Busy Internet Server: Analysis and Improvements.” In *Proc. IEEE Infocom*, San Francisco, CA, USA, March 1998.

[7] R. Braden. “T/TCP - TCP Extensions for Transactions. Functional Specification.” Internet Request For Comments, RFC 1644. July 1994.

[8] R. Braden. “Extending TCP for Transaction - Concepts.” Internet Request For Comments, RFC 1379. November 1992.

[9] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, T. Berners-Lee. “Hypertext Transfer Protocol - HTTP/1.1.” Work In Progress (Internet Draft `draft-ietf-http-v11-spec-rev-06`). November 1998.

[10] T. Berners-Lee, R. Fielding, H. Frystyk. “Hypertext Transfer Protocol - HTTP/1.0.” RFC 1945, Internet Request For Comments. May 1996

[11] K. Fall (ed.) and K. Varadhan (ed.). “ns Notes and Documentation.” March 1998.
http://www-mash.cs.berkeley.edu/ns/ns-documentation.html

[12] J. Crowcroft and P. Oechslin. “Differentiated End-to-End Internet Services using a Weighted Proportional Fair Sharing TCP.” To Appear *ACM Computer Communication Review*, 1998.

[13] M. Mathis and J. Mahdavi. “Forward Acknowledgment: Refining TCP Congestion Control.” In *ACM SIGCOMM Computer Communication Review*, Vol. 26.4, August 1996, pp. 281-292.

[14] S. Floyd. “TCP and Explicit Congestion Notification.” In: *ACM Computer Communication Review*, V. 24 N. 5, October 1994, p. 10-23.

[15] L. Brakmo, S. O’Malley and L. Peterson. “TCP Vegas: New Techniques for Congestion Detection and Avoidance.” In *Proc. ACM SIGCOMM*, May 1994, pp. 24-35.

[16] V. Jacobson. “Congestion Avoidance and Control.” In *Proc. ACM SIGCOMM*, Stanford, CA, USA, August 1988, pp. 273-288.

[17] M. Mathis, J. Mahdavi, S. Floyd and A. Romanow. “TCP Selective Acknowledgment Options.” RFC 2018, Internet Request For Comments. October 1996.

[18] K. Thompson, G.J. Miller and R. Wilder. “Wide-Area Internet Traffic Patterns and Characteristics.” In *IEEE Network*, November 1997.

[19] B. Mah. “An Empirical Model of HTTP Network Traffic.” In *Proceedings of the IEEE INFOCOM ‘97*, IEEE Computer Society Press, Los Alamitos, CA, pp. 592-600.

[20] J. Lyon, K. Evans and J. Klein. “Transaction Internet Protocol Version 3.0.” RFC 2371, Internet Request For Comments. July 1998.

[21] J. Gettys and H.F. Nielsen. “SMUX Protocol Specification.” Work In Progress (W3C Working Draft `WD-mux-19980710`). July 1998.

[22] S. Spero. “Session Control Protocol, Version 2.0.” Work In Progress. November 1996.
http://metalab.unc.edu/ses/scp.html

[23] J. Mogul and S. Deering. “Path MTU Discovery.” RFC 1191, Internet Request For Comments. November 1990.

[24] J. Mogul. “The case for persistent-connection HTTP.” In *ACM Computer Communication Review*, vol. 25, pp. 299-313 October 1995.

[25] H. Balakrishnan, H. S. Rahul and S. Seshan. “An Integrated Congestion Management Architecture for Internet Hosts.” In *Proc. ACM/SIGCOMM*, Cambridge, MA, USA, August 1999.

[26] A. Hughes, J. Touch and J. Heidemann. “Issues in TCP Slow-Start Restart After Idle.” Work In Progress. March 1998.
http://www.isi.edu/~ahughes/pubs/draft-xxx.txt

[27] M. Handley, J. Padhye and S. Floyd. “TCP Congestion Window Validation.” September 1999.

[28] M. Allman, S. Floyd and C. Partridge. “Increasing TCP’s Initial Window.” RFC 2414, Internet Request For Comments. September 1998.

[29] M. Mathis, J. Semke, J. Mahdavi and K. Lahey. “The Rate-Halving Algorithm for TCP Congestion Control.” Work In Progress. June 1999.
http://www.psc.edu/networking/ftp/papers/draft-ratehalving.txt

[30] V. Padmanabhan and R. Katz. “TCP Fast Start: A Technique For speeding Up Web Transfers.” In *Proc. IEEE Globecom ‘98 Internet Mini-Conference*, Sydney, Australia, November 1998.

[31] V. Jacobson and M. Karels. “Congestion Avoidance and Control (revised).” In Proc. *ACM SIGCOMM’88*, pp. 314-329. August 1988.

[32] J. Wroclawski. “The Use of RSVP with IETF Integrated Services.” RFC 2210, Internet Request For Comments, September 1997.

[33] L. Zhang, S. Deering, D. Estrin, S. Shenker and D. Zappala, “RSVP: A New Resource ReSerVation Protocol,” *IEEE Network 7*, 5, 1993, pp. 8-18.

[34] S. Blake, D. Black, M. Carlson, E. Davies, Z. Wang and W. Weiss (1998), “An Architecture for Differentiated Services,” RFC 2475, Internet Request For Comments.

[35] S. Floyd and K. Fall. “Promoting the Use of End-to-End Congestion Control in the Internet.” In: *IEEE/ACM Transactions on Networking*, August 1999.