Concurrent Multipath Transfer Using SCTP Multihoming Over Independent End-to-End Paths

Janardhan R. Iyengar, *Student Member, IEEE*, Paul D. Amer, and Randall Stewart

***Abstract*— Concurrent Multipath Transfer (CMT) uses the Stream Control Transmission Protocol's (SCTP) multihoming feature to distribute data across multiple end-to-end paths in a multihomed SCTP association. We identify three negative side-effects of reordering introduced by CMT that must be managed before efficient parallel transfer can be achieved: (i) unnecessary fast retransmissions by a sender, (ii) overly conservative cwnd growth at a sender, and (iii) increased ack traffic due to fewer delayed acks by a receiver. We propose three algorithms which augment and/or modify current SCTP to counter these side-effects. Presented with several choices as to where a sender should direct retransmissions of lost data, we propose five retransmission policies for CMT. We demonstrate spurious retransmissions in CMT with all five policies, and propose changes to CMT to allow the different policies. CMT is evaluated against AppStripe, an idealized application that stripes data over multiple paths using multiple SCTP associations. The different CMT retransmission policies are then evaluated with varied constrained receive buffer sizes. In this foundation work, we operate under the strong assumption that the bottleneck queues on the end-to-end paths used in CMT are independent.**

***Index Terms*— Load balancing, load sharing, multipath, SCTP, transport layer, end-to-end.**

I. Introduction

A host is multihomed if it can be addressed by multiple IP addresses, as is the case when the host has multiple network interfaces. Though feasibility alone does not determine adoption of an idea, multihoming is increasingly economically feasible and can be expected to be the rule rather than the exception in the near future, particularly when fault tolerance is crucial. Multihomed nodes may be simultaneously connected through multiple access technologies, and even multiple end-to-end paths to increase resilience to path failure. For instance, a mobile user could have simultaneous Internet connectivity via a wireless local area network using 802.11b and a wireless wide area network using GPRS.

We propose using *Concurrent Multipath Transfer (CMT)* between multihomed source and destination hosts to increase an application's throughput. CMT is the concurrent transfer of new data from a source to a destination host via two or more end-to-end paths.

The current transport protocol workhorses, TCP and UDP, do not support multihoming; TCP allows binding to only one network address at each end of a connection. At the time TCP was designed, network interfaces were expensive components, and hence multihoming was beyond the ken of research. Changing economics and an increased emphasis on end-to-end fault tolerance have brought multihoming within the purview of the transport layer. While concurrency can be arranged at other layers (as discussed in Sections IV-D and VI), the transport layer has the best knowledge to estimate end-to-end paths' characteristics.

The Stream Control Transmission Protocol (SCTP) [1], [2] is an IETF standards track protocol that natively supports multihoming at the transport layer. SCTP multihoming allows binding of one transport layer *association* (SCTP's term for a connection) to multiple IP addresses at each end of the association. This binding allows a sender to transmit data to a multihomed receiver through different destination addresses. Simultaneous transfer of new data to multiple destination addresses is currently not allowed due primarily to insufficient research. This research attempts to provide that needed work.

Though CMT uses SCTP in our analysis, our goal is to study CMT at the transport layer in general. The issues and algorithms considered in this research would apply to any multihome-aware transport protocol. We chose SCTP primarily due to lack of mature multihoming mechanisms in any other practical transport layer protocol, and partly due to our expertise with it. We note that the Datagram Congestion Control Protocol (DCCP) [3] does provide "primitive multihoming" at the transport layer, but only for mobility support. DCCP multihoming is useful only for connection migration, and cannot be leveraged for CMT.

Following preliminary concepts and terminology in Section II, Section III specifies three algorithms resulting in CMT_{scd} - a protocol that uses SCTP's multihoming feature for *correctly* transferring data between multihomed end hosts using multiple independent end-to-end paths. A CMT sender can direct retransmissions to one of several destinations that are receiving new transmissions. In Section IV, we present an evaluation of CMT versus an "idealized" hypothetical application which stripes data across multiple SCTP associations (AppStripe). We also propose and evaluate five retransmission policies for CMT. We conclude our discussion of CMT in Section V, and discuss related work in Section VI.

Manuscript received Feb 18, 2005; revised Sept 13, 2005; approved by IEEE/ACM TRANSACTIONS ON NETWORKING Editor S. Seshan.

J. Iyengar and P. Amer are with the Protocol Engineering Lab, CIS Dept., University of Delaware, Newark, DE 19716 USA (email: iyengar@cis.udel.edu; amer@cis.udel.edu). R. Stewart is with the Internet Technologies Division at Cisco Systems (email: rrs@cisco.com).

Prepared through collaborative participation in the Communications and Networks Consortium sponsored by the U. S. Army Research Lab under the Collaborative Tech Alliance Program, Cooperative Agreement DAAD19-01-2-0011. The U. S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation thereon.

Supported in part by University Research Program of Cisco Systems, Inc.

II. PRELIMINARIES

We first overview several ideas and mechanisms used by SCTP; some are compared with TCP to highlight similarities and differences. SCTP is defined in RFC2960 [2] with changes and additions included in the Specification Errata [1]. An SCTP packet, or protocol data unit (PDU), consists of one or more concatenated building blocks called *chunks*: either control or data. For the purposes of reliability and congestion control, each data chunk in an association is assigned a unique Transmission Sequence Number (TSN). Since chunks are atomic, TSNs are associated with chunks of data, as opposed to TCP which associates a sequence number with each data octet in the bytestream. In our simulations, we assume one data chunk per PDU for ease of illustration; each PDU thus carries, and is associated with a single TSN.

SCTP uses a selective ack scheme similar to SACK TCP [4]. SCTP's congestion control algorithms are based on RFC 2581 [5], and include SACK-based mechanisms for better performance. Similar to TCP, SCTP uses three control variables: a receiver's advertised window (rwnd), a sender's congestion window (cwnd), and a sender's slow start threshold (ssthresh). However, unlike TCP's cwnd which reflects which data to send, SCTP's cwnd dictates only how much data can be sent. In SCTP, rwnd is shared across an association. Since an SCTP association allows multihomed source and destination endpoints, a source maintains several parameters on a *per destination* basis: cwnd, ssthresh, and roundtrip time (RTT) estimates. An SCTP sender also maintains a separate retransmission timer per destination. RFC 2960 does not allow a sender to simultaneously send *new* data on multiple paths. New data must be sent to a single *primary destination*, while retransmissions may be sent to any alternate destination.

We operate under the assumption that the bottleneck queues on the end-to-end paths used in CMT are independent. Overlap in the paths is acceptable, but again bottlenecks are assumed independent. Two motivating examples where this assumption holds are telephony networks and battlefield networks.

(i) Signaling communication in telephony networks is being migrated to IP, and uses SCTP for transport. Given the stringent availability requirements on these networks, signaling devices are multihomed and are inter-connected via multiple, independent end-to-end paths for reasons of fault tolerance. The end-to-end paths share no network resource, thus avoiding any single point of failure [6].

(ii) The US Army's proposed Future Combat System for battlefield networks will equip mobile hosts with multiple interfaces, often connecting to independent wireless networks, for example, a terrestrial short-range radio, and a long-range communication to either low-flying or geostationary satellites. These different communication technologies will provide multiple independent paths between nodes [7].

We recognize that our assumption of independent paths is a strong one. If used in the presence of a shared bottleneck, CMT will be as aggressive as multiple SCTP associations sharing a bottleneck. This assumption can be dropped by employing an end-to-end bottleneck detection technique [8]–[12]. We will pursue this line of work in the future.

CMT schedules new data to different destinations as bandwidth becomes available on corresponding paths, i.e., as corresponding cwnds allow. When cwnd space is available simultaneously for two or more destinations, data is sent to these destinations in arbitrary order - a reasonable transmission policy when the CMT sender has no *apriori* knowledge of the paths' characteristics. Our choice to use the full cwnd of a path before using the other path was to reduce reordering.

For CMT, we do not disable heartbeats (HBs) or any other SCTP feature. As long as the application has some data to send, CMT will send data on all paths. Should the application stall in providing data, then even with CMT, HBs should be sent on idle paths. CMT does not need any feature modifications other than the ones described in this paper.

A note on language and terminology. A reference to "cwnd for destination X" means the cwnd maintained at the sender for destination X, and "timeout on destination X" refers to the expiration of a sender's retransmission timer maintained for destination X. Since bottleneck queues on the end-to-end paths are assumed independent, each destination in our topology uniquely maps to an independent path. Thus, "cwnd for destination X" may be used interchangeably with "cwnd for path Y", where path Y ends in destination X. SCTP acks carry cumulative and selective ack (also called *gap ack*) information and are called SACKs. In the paper, sometimes "SACK" is used rather than "ack" to emphasize when an ack carries both cumulative and selective acks.

The simulations presented in this paper use the University of Delaware's SCTP module for ns-2 [13], [14].

III. CMT ALGORITHMS

As is the case with TCP [15]–[17], reordering introduced in an SCTP flow degrades performance. When multiple paths being used for CMT have different delay and/or bandwidth characteristics, significant packet reordering can be introduced in the flow by a CMT sender. Reordering is a natural consequence of CMT, and is difficult to eliminate in an environment where the end-to-end path characteristics are changing or unknown apriori, as in the Internet. In this section, we identify and resolve the negative side-effects of sender-introduced reordering by CMT in SCTP.

Several algorithms propose to eliminate the effects of reordering due to the network [15]–[17]. In this paper, we discuss reordering introduced at the sender, not in the network. The sender has more information about sender-induced reordering, and can address this reordering more effectively.

To demonstrate the effects of reordering introduced in SCTP by CMT, we use a simple simulation setup. Two dualhomed hosts, sender A with local addresses A_1, A_2, and receiver B with local addresses B_1, B_2, are connected by two separate paths: Path 1 ($A_1 - B_1$), and Path 2 ($A_2 - B_2$) having end-to-end available bandwidths 0.2 Mbps and 1 Mbps, respectively. The roundtrip propagation delay on both paths is 90 ms, roughly reflecting the U. S. coast-to-coast delay. CMT sender A sends data to destinations B_1 and B_2 concurrently, and uses a scheduling algorithm that sends new data to a destination when allowed by the corresponding cwnd.

The simulation results described in this section (Figures 1 and 5) both show cwnd evolution over time. The figures show the CMT sender's (1) observed cwnd evolution for destination B_1 (+), (2) observed cwnd evolution for destination B_2 (×), (3) calculated aggregate cwnd evolution (sum of (1) and (2)) (△), and (4) expected aggregate cwnd evolution (–). This last curve represents our initial performance goal for CMT - the sum of the cwnd evolution curves of two independent SCTP runs, using B_1 and B_2 as the primary destination, respectively.

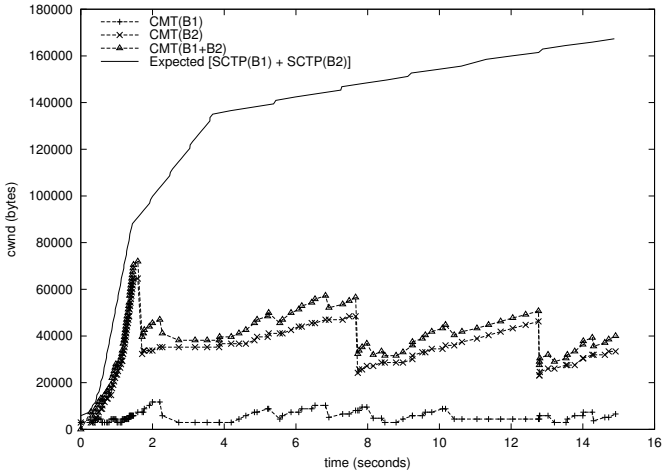

Fig. 1. CMT with SCTP: Evolution of the different cwnds

Figure 1 shows how, when using SCTP without any modifications, CMT reordering significantly hinders both B_1 and B_2's cwnd growth. Normally cwnd reductions are seen when a sender detects loss, but for Figure 1, no packet loss was simulated. The aggregate cwnd evolution (△) is significantly below the expected aggregate cwnd evolution (–).

We identify and resolve three negative side-effects of reordering introduced by CMT that must be managed before the full performance gains of parallel transfer can be achieved: (i) unnecessary fast retransmissions at the sender (Section III-A), (ii) reduced cwnd growth due to fewer cwnd updates at the sender (Section III-B), and (iii) increased ack traffic due to fewer delayed acks (Section III-C) [18]. While designing these algorithms, we implicitly assumed that any retransmission will be sent to the same destination as the original transmission. We revisit this assumption in Section IV.

A note on notation: CMT refers to a host performing concurrent multipath transfer using current SCTP. CMT$_s$, CMT$_c$, and CMT$_d$ refer to a host performing CMT with Split Fast Retransmit (SFR) (Section III-A), Cwnd Update for CMT (CUC) (Section III-B) and Delayed Ack for CMT (DAC) (Section III-C), respectively. Multiple subscripts indicates use of more than one algorithm.

A. Preventing Unnecessary Fast Retransmissions

When reordering is observed, a receiver sends gap reports (i.e., gap acks) to the sender which uses the reports to detect loss through a fast retransmission procedure similar to the one used by TCP [2], [5]. With CMT, unnecessary fast retransmissions can be caused due to reordering [19], with two negative consequences: (1) since each retransmission is assumed to occur due to a congestion loss, the sender reduces its cwnd for the destination on which the retransmitted data was outstanding, and (2) a cwnd overgrowth problem explained in [19] causes a sender's cwnd to grow aggressively for the destination on which the retransmissions are sent, due to acks received for original transmissions. In Figure 1, each cwnd reduction observed for B_1 and B_2 is due to an unnecessary fast retransmission. These unnecessary retransmissions are due to sender-introduced reordering, and not spurious retransmissions due to network effects [20], [21].

Conventional interpretation of a SACK chunk in SCTP (or SACK options in TCP) is that gap reports imply possible loss. The probability that a TSN is lost, as opposed to being reordered, increases with the number of gap reports received for that TSN. Due to sender-induced reordering, a CMT sender needs additional information to infer loss. Gap reports *alone* do not (necessarily) imply loss; but a sender can infer loss using gap reports *and* knowledge of each TSN's destination.

Algorithm Details: The proposed solution to address the side-effect of incorrect cwnd evolution due to unnecessary fast retransmissions is the Split Fast Retransmit (SFR) algorithm (Figure 2). SFR extends a previous incarnation which could not handle *cycling changeover* [19]. SFR introduces a *virtual queue* per destination within the sender's retransmission queue. A sender then deduces missing reports for a TSN correctly using SACK information in conjunction with state maintained about the transmission destination for each TSN in the retransmission queue. Thus, SFR enables a multihomed sender to correctly apply the fast retransmission procedure on a per destination basis. An advantage of SFR is that only the sender's behavior is affected. SFR introduces two additional variables per destination at a sender:

1) *highest_in_sack_for_dest* - stores the highest TSN acked per destination by the SACK being processed.
2) *saw_newack* - a flag used during the processing of a SACK to infer the causative TSN(s)'s destination(s). Causative TSNs for a SACK are those TSNs which caused the SACK to be sent (or TSNs that are acked in this SACK, and are acked for the first time).

In Figure 2, step (2) sets *saw_newack* to TRUE for the destinations to which the newly acked TSNs were sent. Step (3) tracks on a per destination basis, the highest TSN being acked. Step (4) uses information gathered in steps (2) and (3) to aid in inferring missing TSNs. Two conditions in step (4) ensure correct missing reports: (a) TSNs to be marked should be outstanding on the same destination(s) as TSNs which have been newly acked, and (b) at least one TSN, sent later than the missing TSN, but *to the same destination address*, should be newly acked.

B. Avoiding Reduction in Cwnd Updates

The cwnd evolution algorithm for SCTP [2] (and analogously for SACK TCP [4], [5]) allows growth in cwnd only when a new cum ack is received by a sender. When SACKs with unchanged cum acks are generated (say due to reordering) and later arrive at a sender, the sender does not modify its cwnd. This mechanism again reflects the conventional view

On receipt of a SACK containing gap reports [Sender side behavior]:

1) $\forall$ destination addresses d_i initialize $d_i.saw\_newack$ = FALSE;
2) **for** each TSN t_a being acked that has not been acked in any SACK thus far **do**
   let d_a be the destination to which t_a was sent;
   set $d_a.saw\_newack$ = TRUE;
3) $\forall$ destinations d_n, set $d_n.highest\_in\_sack\_for\_dest$ to highest TSN being newly acked on d_n;
4) to determine whether missing report count for a TSN t_m should be incremented:
   let d_m be the destination to which t_m was sent;
   if ($d_m.saw\_newack$ = TRUE) **and** ($d_m.highest\_in\_sack\_for\_dest > t_m$) **then**
   increment missing report count for t_m;
   else do not increment missing report count for t_m;

Fig. 2. SFR Algorithm – Eliminating unnecessary fast retransmissions

At beginning of an association [Sender side behavior]:
$\forall$ destinations d, reset
d.find_pseudo_cumack = TRUE;
On receipt of a SACK [Sender side behavior]:

1) $\forall$ destinations d, reset *d.new_pseudo_cumack* = FALSE;
2) **if** the SACK carries a new cum ack **then**
   for each TSN t_c being cum acked for the first time, that was not acked through prior gap reports **do**
   (i) let d_c be the destination to which t_c was sent;
   (ii) set $d_c.find\_pseudo\_cumack$ = TRUE;
   (iii) set $d_c.new\_pseudo\_cumack$ = TRUE;
3) **if** gap reports are present in the SACK **then**
   for each TSN t_p being processed from the retransmission queue **do**
   (i) let d_p be the destination to which t_p was sent;
   (ii) **if** ($d_p.find\_pseudo\_cumack$ = TRUE) **and** t_p was not acked in the past **then**
   $d_p.pseudo\_cumack = t_p$;
   $d_p.find\_pseudo\_cumack$ = FALSE;
   (iii) **if** t_p is acked via gap reports for first time **and** ($d_p.pseudo\_cumack = t_p$) **then**
   $d_p.new\_pseudo\_cumack$ = TRUE;
   $d_p.find\_pseudo\_cumack$ = TRUE;
4) **for** each destination d **do**
   if (*d.new_pseudo_cumack* = TRUE) **then** update cwnd [1], [2];

Fig. 3. Cwnd Update for CMT (CUC) Algorithm – Handling side-effect of reduced cwnd growth due to fewer cwnd updates

that a SACK which does not advance the cum ack indicates possibility of loss due to congestion.

Since a CMT receiver naturally observes reordering, many SACKs are sent containing new gap reports but not new cum acks. When these gaps are later acked by a new cum ack, cwnd growth occurs, but only for the data newly acked in the most recent SACK. Data previously acked through gap reports will not contribute to cwnd growth. This behavior prevents sudden increases in the cwnd resulting in bursts of data being sent. Even though data may have reached the receiver "in-order per destination", without changing the current handling of cwnd, the updated cwnd will not reflect this fact.

This inefficiency can be attributed to the current design principle that the cum ack in a SACK, which tracks the latest TSN received in-order at the receiver, applies to an entire association, not per destination. TCP and current SCTP use only one destination address at any given time to transmit new data to, and hence, this design principle works fine. Since CMT uses multiple destinations simultaneously, cwnd growth in CMT demands tracking the latest TSN received in-order *per destination*, information not coded directly in a SACK.

We propose a cwnd growth algorithm to track the earliest outstanding TSN *per destination* and update the cwnd, even in the absence of new cum acks. The algorithm uses SACKs and knowledge of transmission destination for each TSN to deduce in-order delivery per destination. The crux of the CUC algorithm is to track the earliest outstanding data *per destination*, and use SACKs which ack this data to update the corresponding cwnd. In understanding our proposed solution, we remind the reader that gap reports alone do not (necessarily) imply congestion loss; SACK information is treated only as a concise description of the TSNs received by the receiver.

Algorithm Details: Figure 3 shows the proposed Cwnd Update for CMT (CUC) algorithm. A *pseudo-cumack* tracks

On receipt of a data PDU [Receiver side behavior]:

1) delay sending an ack as given in [2], with the additional change that acks should be delayed even if reordering is observed.
2) in each ack, report number of data PDUs received since sending of previous ack.

When incrementing missing report count through SFR:Step (4) (Figure 2) [Sender side behavior]:

4) to determine whether missing report count for a TSN t_m should be incremented:
   let d_m be the destination to which t_m was sent;
   if ($d_m.saw\_newack$ = TRUE) **and** ($d_m.highest\_in\_sack\_for\_dest > t_m$) **then**
   (i) **if** ($\forall$ destinations d_o such that $d_o \neq d_m$, $d_o.saw\_newack$ = FALSE) **then**
   /** all newly acked TSNs were sent to the same destination as t_m **/
   (a) **if** ($\exists$ newly acked TSNs t_a, t_b such that $t_a < t_m < t_b$) **then**
   (conservatively) increment missing report count for t_m by 1;
   (b) **else if** ($\forall$ newly acked TSNs t_a, such that $t_a > t_m$) **then**
   increment missing report count for t_m by number of PDUs reported by receiver;
   (ii) **else**
   /** Mixed SACK - newly acked TSNs were sent to multiple destinations **/
   (conservatively) increment missing report count for t_m by 1;

Fig. 4. Delayed Ack for CMT (DAC) Algorithm – Handling side-effect of increased ack traffic

the earliest outstanding TSN per destination at the sender. An advance in a pseudo-cumack triggers a cwnd update for the corresponding destination, even when the actual cum ack is not advanced. The pseudo-cumack is used for cwnd updates; only the actual cum ack can dequeue data in the sender's retransmission queue since a receiver can reneg on data that is not cumulatively acked. An advantage of CUC is that only the sender's behavior is affected. The CUC algorithm introduces three variables per destination at a sender:

1) *pseudo_cumack* - maintains earliest outstanding TSN.
2) *new_pseudo_cumack* - flag to indicate if a new pseudo-cumack has been received.
3) *find_pseudo_cumack* - flag to trigger search for a new pseudo-cumack. This flag is set after a new pseudo-cumack has been received.

In Figure 3, step (2) initiates a search for a new *pseudo_cumack* by setting *find_pseudo_cumack* to TRUE for the destinations on which TSNs newly acked were outstanding. A cwnd update is also triggered by setting *new_pseudo_cumack* to TRUE for those destinations. Step (3) then processes the outstanding TSNs at a sender, and tracks on a per destination basis, the TSN expected to be the next *pseudo_cumack*. Step (4) finally updates the cwnd for a destination if a new *pseudo_cumack* was seen for that destination.

C. Curbing Increase in Ack Traffic

Sending an ack after receiving every 2 data PDUs (i.e., delayed acks) in SCTP (and TCP) reduces ack traffic in the Internet, thereby saving processing and storage at routers on the ack path. SCTP specifies that a receiver should use the delayed ack algorithm as given in RFC 2581, where acks are delayed only as long as the receiver receives data in order. Reordered PDUs should be acked immediately [5]. With CMT's frequent reordering, this rule causes an SCTP receiver to frequently *not* delay acks. Hence a negative side-effect of reordering with CMT is increased ack traffic.

To prevent this increase, we propose that a CMT receiver ignore the rule mentioned above. That is, a CMT receiver does not immediately ack an out-of-order PDU, but delays the ack. Thus, a CMT receiver always delays acks, irrespective of whether or not data is received in order[1]. Though this modification eliminates the increase in ack traffic, RFC 2581's rule has another purpose which gets hampered.

An underlying assumption that pervades SCTP's (and TCP's) design is that data in general arrives in order; data received out-of-order indicates possible loss. According to RFC 2581, a receiver should immediately ack data received above a gap in the sequence space to accelerate loss recovery by triggering the fast retransmit algorithm [5]. In SCTP, four acks with missing reports for a TSN indicate that a receiver received at least four data PDUs sent after the missing TSN. Receipt of four missing reports for a TSN triggers the sender's fast retransmit algorithm. In other words, the sender has a *reordering threshold* (or *dupack threshold*) of four PDUs. Since a CMT receiver cannot distinguish between loss and reordering introduced by a CMT sender, the modification suggested above by itself would cause the receiver to delay acks even in the face of loss. When a loss does occur with our modification to a receiver, fast retransmit would be triggered by a CMT sender only after the receiver receives eight(!) data PDUs sent after a lost TSN - an overly conservative behavior.

The effective increase in reordering threshold at a sender can be countered by reducing the actual number of acks required to trigger a fast retransmit at the sender, i.e., by increasing the number of missing reports registered per ack. In other words, if a sender can increment the number of missing reports more accurately per ack received, fewer acks will be required to trigger a fast retransmit. A receiver can provide more information in each ack to assist the sender in accurately

[1]We do not modify a receiver's behavior when an ack being delayed can be piggybacked on reverse path data, or when the delayed ack timer expires.

inferring the number of missing reports per ack for a lost TSN. We propose that in each ack, a receiver report the number of data PDUs received since the previous ack was sent. A sender then infers the number of missing reports per TSN based on the TSNs being acked in a SACK, number of PDUs reported by the receiver, and knowledge of transmission destination for each TSN. We note that additionally, heuristics (as proposed in [15]) may be used at a CMT sender to address network induced reordering.

Algorithm Details: The proposed Delayed Ack for CMT (DAC) algorithm (Figure 4) specifies a receiver's behavior on receipt of data, and also a sender's behavior when the missing report count for a TSN needs to be incremented[2]. Since SCTP (and TCP) acks are cumulative, loss of an ack will result in loss of the data PDU count reported by the receiver, but the TSNs will be acked by the following ack. Receipt of this following ack can cause ambiguity in inferring missing report count per destination. Our algorithm conservatively assumes a single missing report count per destination in such ambiguous cases. The DAC algorithm requires modifications to both the sender and the receiver.

No new variables are introduced in this algorithm, as we build on the SFR algorithm. An additional number is reported in the SACKs for which we use the first bit of the flags field in the SACK chunk header - 0 indicates a count of one PDU (default SCTP behavior), and 1 indicates two PDUs.

In Figure 4, at the receiver side, steps (1) and (2) are self explanatory. The sender side algorithm modifies step (4) of SFR, which determines whether missing report count should be incremented for a TSN. The DAC algorithm dictates *how many* to increment by. Step (4-i) checks if only one destination was newly acked, and allows incrementing missing reports by more than one for TSNs outstanding to that destination. Further, all newly acked TSNs should have been sent later than the missing TSN. If there are newly acked TSNs that were sent before the missing TSN, step (4-i-a) conservatively increments by only one. If more than one destinations are newly acked, step (4-ii) conservatively increments by only one.

Figure 5 shows cwnd evolution for CMT after including the SFR, CUC and DAC algorithms, i.e., CMT$_{scd}$. With the negative side-effects addressed, we hoped to see CMT$_{scd}$'s cwnd growth to come close to the expected aggregate cwnd growth. In fact, we observed that CMT$_{scd}$ cwnd growth *exceeded* the expected aggregate cwnd growth!

To explain this surprising result, we remind the reader that the expected aggregate cwnd is the sum of the cwnd growth of two independent SCTP runs, each using one of the two destination addresses as its primary destination. In each SCTP run, one delayed ack can increase the cwnd by at most one MSS during slow start, even if the ack acks more than one MSS worth of data. On the other hand, we observe with CMT$_{scd}$ that if a delayed ack simultaneously acks an MSS of data on each of the two destinations, the sender simultaneously increases each of two cwnds by one MSS. Thus, a single delayed ack in CMT$_{scd}$ that acks data flows on two paths causes an aggregate cwnd growth of two MSS. With delayed acks during slow start, each SCTP association grows its cwnd by 1.5 times each RTT, whereas CMT$_{scd}$ can increase its cwnd by more than 1.5 times in each RTT (upto two times in the best case where every delayed ack acks an MSS on each path). *Delayed acks which simultaneously contribute to the cwnd growth of two destinations helped the aggregate cwnd growth of CMT$_{scd}$ exceed expected aggregate cwnd growth.*

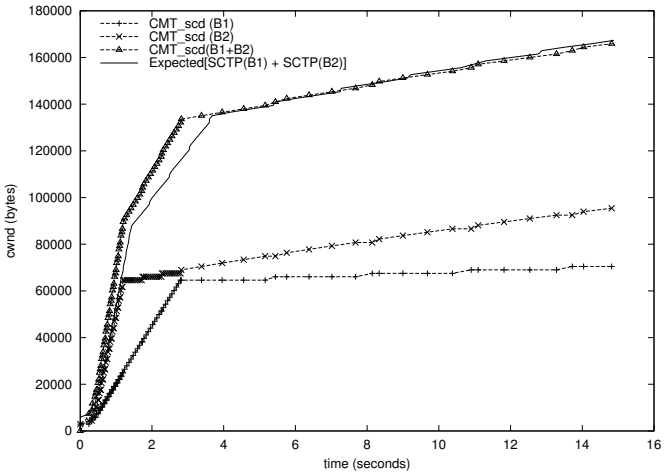

Fig. 5. CMT$_{scd}$: Evolution of the different cwnds

This phenomenon occurs in slow start, therefore benefiting CMT$_{scd}$ initially and during some timeout recovery periods. Though the aggregate cwnd growth exceeds expected aggregate cwnd growth, we argue that the sender is not overly aggressive, i.e., not TCP-unfriendly. The sender is able to clock out more data due to delayed acks that ack data flows on multiple paths. The sender does not create bursts of data during slow start, and builds up the ack clock as expected. Though it does not improve CMT$_{scd}$'s performance significantly, *this phenomenon demonstrates a benefit of* sequence space sharing *among flows on different paths that occurs within a CMT$_{scd}$ association.*

IV. CMT Performance Evaluation

With correct behavior ensured by CMT$_{scd}$ (henceforth referred to simply as CMT), we now evaluate its performance. In Section IV-A, we discuss our methodology for evaluating CMT. In Section IV-B, we present five retransmission policies for CMT. In Section IV-C, we identify two modifications that must be made to CMT to accomodate the different retransmission policies. In Section IV-D, we evaluate CMT against AppStripe - our reference application for performance evaluation of CMT. In Section IV-E, we compare and analyze the different retransmission policies to decide upon a recommended policy for CMT.

A. Evaluation Methodology

As a reference, we use *AppStripe* - a hypothetical multihome-aware application that achieves the highest throughput possible by an application that distributes data across multiple SCTP associations (see Figure 6). We emphasize that AppStripe performs idealized scheduling at the application layer, and is not doable in practice. End-to-end load

[2]The DAC algorithm also can be used when ack traffic lesser than with delayed acks is desirable, such as in data center environments [22].

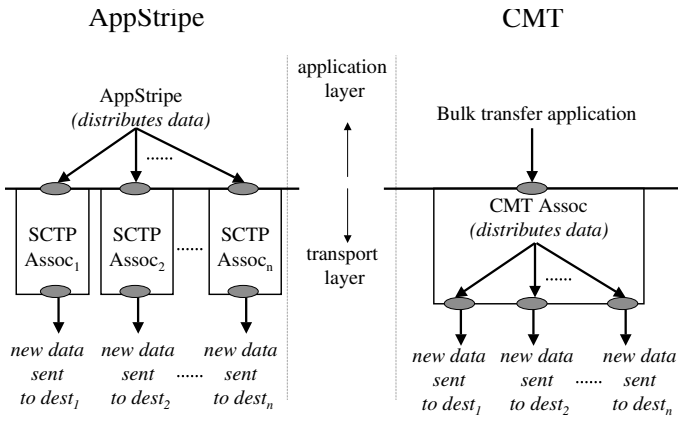

Fig. 6. Schematic - AppStripe and CMT

sharing is performed at the application layer by AppStripe, and at the transport layer by CMT.

We simulate AppStripe by post-processing simulation traces. We simulate separate file transfers over multiple separate SCTP associations, each on a separate path to the receiver. To find an "optimal" transfer time, we use these two traces to extract the time when the total amount of data transferred, across the two associations, equals the desired transfer size.

AppStripe hypothetically assumes the ability of an application to schedule data to each transport association immediately when a transport association is able to send data to a receiver. Such an ability requires a complex application-transport interface, which to our knowledge, is not realized in practice today. A typical application distributing data over multiple associations would have to use a heuristic to decide the fraction of data to be scheduled on each association. Thus AppStripe's performance in our paper represents the best achievable separation of data over multiple paths.

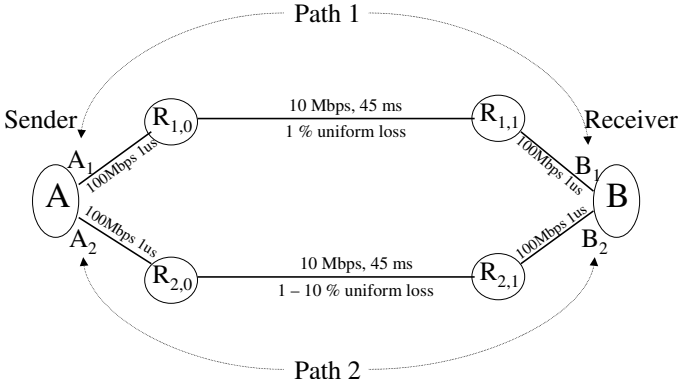

Fig. 7. Simulation topology used for evaluation

The simulation topology (see Figure 7) is simple - the edge links represent the last hop, and the core links represent end-to-end conditions on the Internet. This simulation topology does not account for effects seen in the Internet and other real networks such as network induced reordering, delay spikes, etc.; these effects are beyond the scope of this study. Our simulation evaluation provides insight into the fundamental differences between AppStripe and CMT, and between the different retransmission policies in a constrained environment. We chose a simple topology to avoid influence of other effects, and to focus on performance differences which we believe should hold true in a real environment as well[3]. The loss rate on Path 1 is maintained at 1%, and on Path 2 is varied from 1 to 10%. A loss rate of 1% means a forward path loss rate of 1%, and a reverse path loss rate of 1%.

Our choice of simulation parameters was based on our understanding that end-to-end throughput is influenced by loss rate and delay. We focus on loss rate differences since we believe loss rate has a more significant impact on the retransmission policy. We are currently studying the influence of different delays, and delay combinations on CMT [23].

The absolute bandwidths were chosen to be high enough so that end-to-end delays are dominated by propagation delay. The relative bandwidths of the links were chosen so that any queuing happens at intermediate routers where a uniform loss rate is applied to the packets. End-to-end delay was chosen as 45ms to represent a typical US coast-to-coast delay.

B. CMT Retransmission Policies

Multiple paths present an SCTP sender with several choices where to send a retransmission. But these choices are not well-informed since SCTP restricts sending new data, which can act as probes for information (such as available bandwidth, loss rate and RTT), to only one primary destination. Consequently, an SCTP sender has minimal information about other paths to a receiver. On the other hand, a CMT sender maintains more accurate information about all paths, since new data is being sent to all destinations concurrently. This information allows a CMT sender to better decide where to retransmit.

We present five retransmission policies for CMT [24]. In four policies, a retransmission may be sent to a destination other than the one used for the original transmission. Previous research on SCTP retransmission policies shows that sending retransmissions to an alternate destination degrades performance primarily because of the lack of sufficient traffic on alternate paths [25]. With CMT, data is concurrently sent on all paths, thus the results in [25] are not applicable. The five retransmission policies for CMT are:

- **RTX-SAME** - Once a new data chunk is scheduled and sent to a destination, all retransmissions of the chunk are sent to the same destination (until the destination is deemed *inactive* due to failure [2]).
- **RTX-ASAP** - A retransmission of a data chunk is sent to any destination for which the sender has cwnd space available at the time of retransmission. If multiple destinations have available cwnd space, one is chosen randomly.
- **RTX-CWND** - A retransmission is sent to the destination for which the sender has the largest cwnd. A tie is broken by random selection.
- **RTX-SSTHRESH** - A retransmission is sent to the destination for which the sender has the largest ssthresh. A tie is broken by random selection.

[3]The simulation topology is clearly simplistic. We are currently doing further study involving more complex topologies with variable cross-traffic [23]. Our initial results support our conclusions in this paper.

At beginning of an association [Sender side behavior]:
∀ destinations d, reset
d.find_pseudo_cumack = *d.find_rtx_pseudo_cumack* = TRUE;
On receipt of a SACK [Sender side behavior]:
1) ∀ destinations d, reset
d.new_pseudo_cumack = *d.new_rtx_pseudo_cumack* = FALSE;
2) **if** the ack carries a new cum ack **then**
for each TSN t_c being cum acked for the first time, that was not acked through prior gap reports **do**
(i) let d_c be the destination to which t_c was sent;
(ii) set $d_c.find\_pseudo\_cumack$ = $d_c.find\_rtx\_pseudo\_cumack$ = TRUE;
(iii) set $d_c.new\_pseudo\_cumack$ = $d_c.new\_rtx\_pseudo\_cumack$ = TRUE;
3) **if** gap reports are present in the ack **then**
for each TSN t_p being processed from the retransmission queue **do**
(i) let d_p be the destination to which t_p was sent;
(ii) **if** ($d_p.find\_pseudo\_cumack$ = TRUE) **and** t_p was not acked in the past **and** t_p was not retransmitted **then**
$d_p.pseudo\_cumack$ = t_p;
$d_p.find\_pseudo\_cumack$ = FALSE;
(iii) **if** t_p is acked via gap reports for first time **and** ($d_p.pseudo\_cumack$ = t_p) **then**
$d_p.new\_pseudo\_cumack$ = TRUE;
$d_p.find\_pseudo\_cumack$ = TRUE;
(iv) **if** ($d_p.find\_rtx\_pseudo\_cumack$ = TRUE) **and** t_p was not acked in the past **and** t_p was retransmitted **then**
$d_p.rtx\_pseudo\_cumack$ = t_p;
$d_p.find\_rtx\_pseudo\_cumack$ = FALSE;
(v) **if** t_p is acked via gap reports for first time **and** ($d_p.rtx\_pseudo\_cumack$ = t_p) **then**
$d_p.new\_rtx\_pseudo\_cumack$ = TRUE;
$d_p.find\_rtx\_pseudo\_cumack$ = TRUE;
4) **for** each destination d **do**
if (*d.new_pseudo_cumack* = TRUE) **or** (*d.new_rtx_pseudo_cumack* = TRUE) **then**
Update cwnd [1], [2];

Fig. 8. CUCv2 Algorithm - Modified Cwnd Update for CMT (CUC) Algorithm

- **RTX-LOSSRATE** - A retransmission is sent to the destination with the lowest loss rate path. If multiple destinations have the same loss rate, one is selected randomly.

Of the policies, RTX-SAME is simplest. RTX-ASAP is a "hot-potato" policy - retransmit as soon as possible without regard to loss rate. RTX-CWND and RTX-SSTHRESH practically track, and attempt to move retransmissions onto the path with the estimated lowest loss rate. Since ssthresh is a slower moving variable than cwnd, the values of ssthresh may better reflect the conditions of the respective paths. RTX-LOSSRATE uses information about loss rate provided by an "oracle" - information that RTX-CWND and RTX-SSTHRESH estimate. This policy represents a hypothetically ideal case; hypothetical since in practice, a sender typically does not know *apriori* path loss rates; ideal since the path with the lowest loss rate has highest chance of having a packet delivered. We hypothesized that retransmission policies that take loss rate into account would outperform ones that do not.

C. Modifications to Protocol Mechanisms

Two modifications are needed to allow redirecting retransmissions to a different destination than the original.

1) CUCv2: Modified CUC Algorithm: The CUC algorithm (Figure 3) enables correct cwnd updates in the face of increased reordering due to CMT. To recap, this algorithm recognizes a set of TSNs outstanding per destination, and the per-destination *pseudo_cumack* traces the left edge of this list of TSNs, per destination. CUC assumes that retransmissions are sent to the same destination as the original transmission. The per-destination *pseudo_cumack* therefore moves whenever the corresponding left edge is acked; the TSN on the left edge being acked may or may not have been retransmitted.

If the assumption about the retransmission destination is violated, and a retransmission is made to a different destination from the original, CUC cannot faithfully track the left edge on either destination. We modify CUC to permit the different retransmission policies. The modified algorithm, named CUCv2 is shown in Figure 8.

CUCv2 recognizes that a distinction can be made about the TSNs outstanding on a destination - those that have been retransmitted, and those that have not. CUCv2 maintains two left edges for these two sets of TSNs - *rtx_pseudo_cumack* and *pseudo_cumack*. Whenever either of the left edges moves, a cwnd update is triggered.

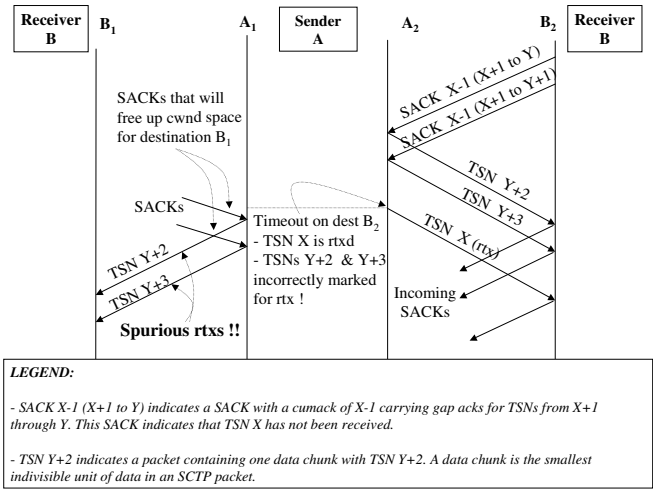

Fig. 9. Example of spurious retransmissions after timeout in CMT

2) Spurious Timeout Retransmissions: When a timeout occurs, an SCTP sender is expected to bundle and send as many of the earliest TSNs outstanding on the destination for which the timeout occurred as can fit in an MSS (Maximum Segment Size) PDU. Per RFC 2960, more TSNs that are outstanding on that destination "should be marked for retransmission and sent as soon as cwnd allows (normally when a SACK arrives)". This rule is intuitive. While sending, retransmissions are generally given priority over new transmissions. As in TCP, the cwnd is also collapsed to 1 MSS for the destination on which a timeout occurs.

A timeout retransmission can occur in SCTP (as in TCP) for several reasons. One reason is loss of the fast retransmission of a TSN. Consider Figure 9. When a timeout occurs due to loss of a fast retransmission, some TSNs that were just sent to the destination on which the timeout occurred are likely awaiting acks (in Figure 9, TSNs Y+2 and Y+3). These TSNs get incorrectly marked for retransmission on timeout. With the different CMT retransmission policies, these retransmissions may be sent to a different destination than the original transmission. In Figure 9, spurious retransmissions of TSNs Y+2 and Y+3 are sent to destination B_1, on receipt of acks freeing up cwnd space for destination B_1. Spurious retransmissions are exacerbated in CMT, as shown through this illustration, due to the possibility of sending data (including retransmissions) to multiple destinations concurrently.

We simulated the occurrence of such spurious retransmissions with the different retransmission policies in CMT. The simulation topology used was the one described in Section IV-A. Figure 10(a) shows the ratio of retransmissions relative to the number of actual packet drops at the router. Ideally, the two numbers should be equal; all curves should be straight lines at $y = 1$. Figure 10(a) shows that spurious retransmissions occur commonly in CMT with the different retransmission policies.

We propose a heuristic to avoid these spurious retransmissions. Our heuristic assumes that a timeout cannot be triggered on a TSN until the TSN has been outstanding for at least one RTT. Thus, if a timeout is triggered, TSNs which were sent within one RTT are not marked for retransmission. We use an average measure of the RTT for this purpose - the smoothed RTT, which is maintained at a sender. This heuristic requires the sender to maintain a timestamp for each TSN indicating the time at which the TSN was last transmitted (or retransmitted). Figure 10(b) shows how the application of this heuristic dramatically reduces spurious retransmissions.

D. Performance of CMT vs. AppStripe

Figure 11(a) compares the time taken to transfer an 8MB file using CMT with the five retransmission policies, vs. using AppStripe. The x-axis represents different loss rates on Path 2. Each plotted value is the mean of at least 30 simulation runs. Overall, AppStripe ($\times$ in Figure 11(a)) performs worst, and CMT using any of the retransmission policies performs better than AppStripe; some policies better than others. At a 7% loss rate on Path 2, AppStripe takes 40.4 seconds to transfer an 8 MB file, whereas CMT using RTX-SAME or RTX-CWND takes 35.5 or 33.2 seconds, respectively. We first discuss the performance difference between CMT in general and AppStripe.

CMT using any retransmission policy performs better than AppStripe, particularly as the loss rate on Path 2 increases. Note that our AppStripe represents *the best possible performance* expected by an application that stripes data over multiple SCTP associations. AppStripe is an idealized case; CMT's performance gain over a practical AppStripe implementation would be even larger since a practical implementation has to optimally stripe data across paths that have different and changing delays and loss rates. Such striping may require information from the transport layer (such as current cwnd and RTT), that may not be readily available to the application.

CMT performs better than AppStripe for two reasons. First, and significant, CMT is more resilient to reverse path loss than AppStripe. CMT uses a single sequence space (TSN space, used for congestion control and loss detection and recovery) across an association's multiple paths, whereas AppStripe by design uses an independent sequence space per path. Since acks are cumulative, sharing of sequence spaces across paths helps a CMT sender receive ack info on either of the return paths. Thus, CMT effectively uses *both* return paths for communicating ack info to the sender, whereas each association in AppStripe cannot help the other "ack-wise". These results demonstrate the significant result that CMT's sharing of sequence space across paths is *an inherent benefit that performing load sharing at the transport layer has over performing it at the application layer*.

We emphasize that ack loss can cause throughput degradation, especially at higher loss rates. Ack loss can delay fast retransmissions by one or more RTTs, thus delaying cwnd increase. Increased ack loss can also increase the number of timeout retransmissions when the window is small (say during the initial part of an association, or after timeout recovery). These performance penalties add up over the lifetime of an association. (See Figures 6 and 7(a) in [24], for a demonstration of throughput degradation due to ack loss.)

Second, CMT gets faster overall cwnd growth than AppStripe in slow start (See Section III-C). As loss increases, the

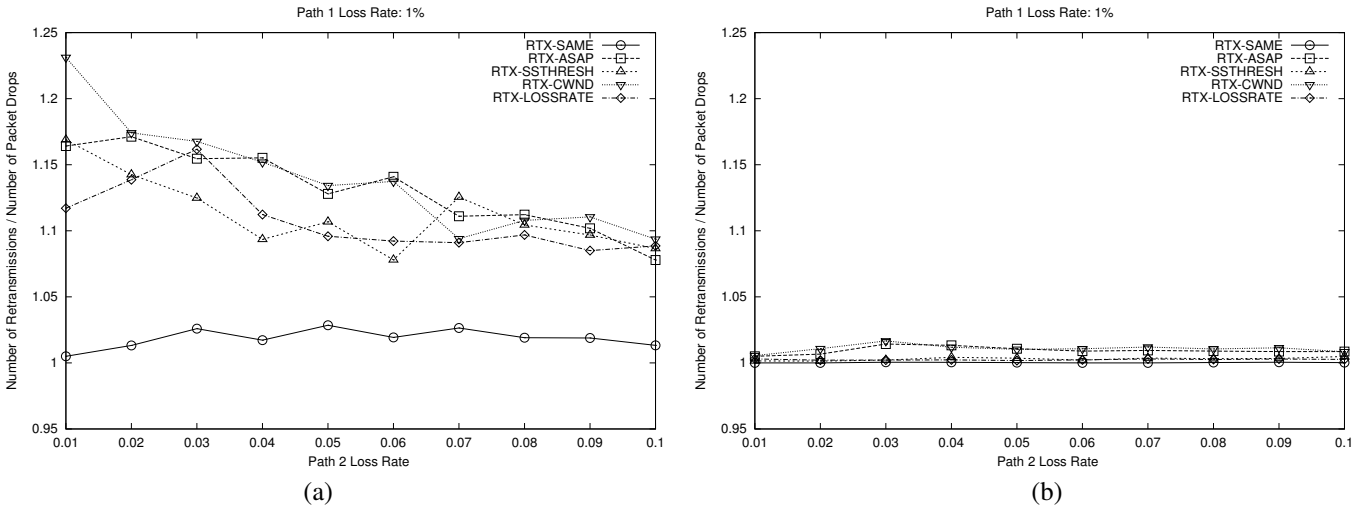

Fig. 10. Spurious retransmissions in CMT: (a) Without RTT heuristic (b) With RTT heuristic

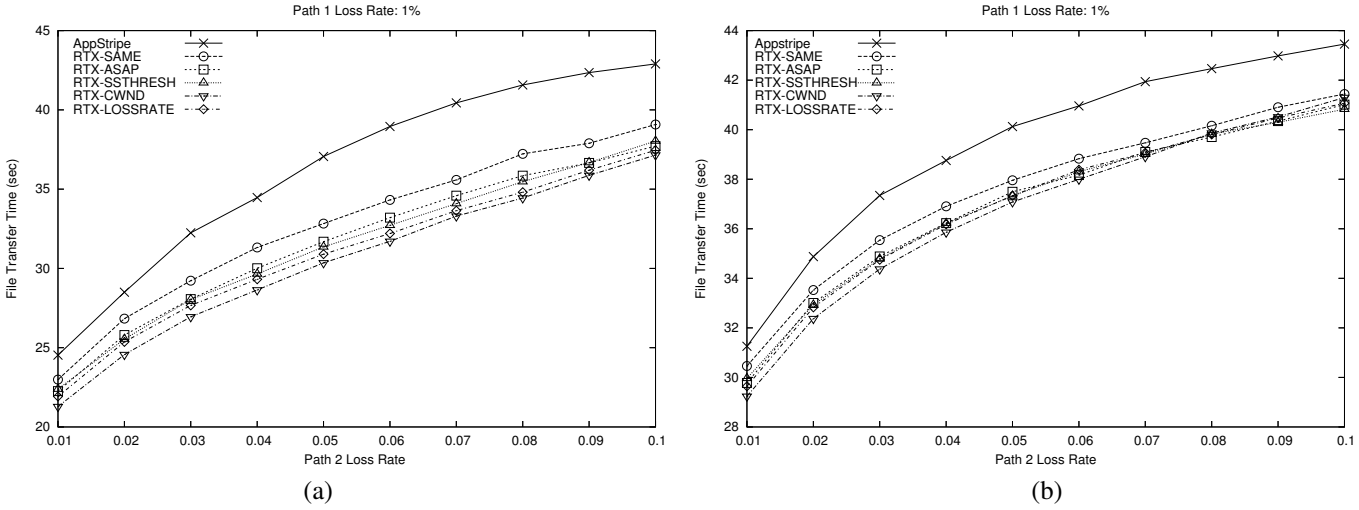

Fig. 11. Path 1 loss rate = 1%, performance of AppStripe vs. CMT with different policies, under (a) equal path delays (Path 1 = 45ms, Path 2 = 45ms), and (b) unequal path delays (Path 1 = 45 ms, Path 2 = 90 ms)

number of timeouts increases, and since slow start follows a timeout, the sender spends more time overall in slow start.

Extensive simulations with unequal path delays (results not included), show that unequal path delays do not impact the relative performance of AppStripe and CMT with the different policies. Figure 11(b) demonstrates this consistent behavior with unequal path delays of 45 ms on Path 1, and 90 ms on Path 2. Note that these results are consistent with Figure 11(a) which has equal delays of 45 ms on both paths.

E. Performance of different retransmission policies for CMT

Of the retransmission policies for CMT in Figures 11(a) and (b), RTX-SAME ($\ominus$) performs marginally but consistently worse than RTX-ASAP ($\square$), which in turn performs as well as the loss rate based policies - RTX-SSTHRESH ($\triangle$), RTX-CWND ($\bigtriangledown$), and RTX-LOSSRATE ($\Diamond$). While the performance difference between the retransmission policies in Figure 11 is not significant, these results use an 8MB receiver's buffer (rbuf) that does not constrain the sender - an unrealistic assumption which we will now drop [26].

Figure 12(a) shows the time taken for a CMT sender to transfer an 8MB file when the rbuf is set to 64KB, using the five retransmission policies. RTX-SAME is the simplest to implement, but performs worst. The performance difference between RTX-SAME and other policies increases as the loss rate on Path 2 increases. RTX-ASAP performs better than RTX-SAME, but still worse than RTX-LOSSRATE, RTX-SSTHRESH and RTX-CWND. The three loss rate based policies perform equally.

Figure 12(b) shows the number of retransmission timeouts experienced when using the different policies. This figure shows that performance improvement in using RTX-LOSSRATE, RTX-CWND, and RTX-SSTHRESH is due to the reduced number of timeouts. A lost transmission may be recovered via a fast retransmission, but a lost fast retransmission can be recovered only through a timeout. RTX-SAME does not consider loss rate in choosing a retransmission destination and consequently experiences the largest number of timeouts due to increased loss of retransmissions.

RTX-ASAP does not consider loss rate, and performs better than RTX-SAME. This improved performance with RTX-ASAP is attributed to cwnd space availability on both

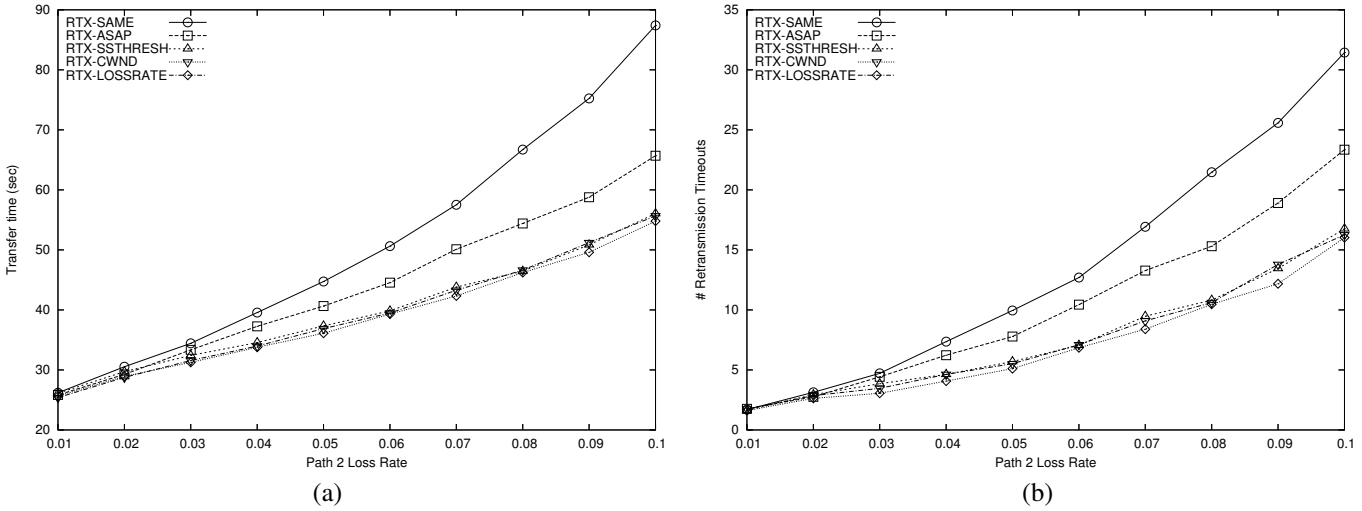

Fig. 12. rbuf = 64KB, and Path 1 loss rate = 1%: (a) CMT time to transfer 8MB file, (b) Retransmission timeouts for CMT with different policies

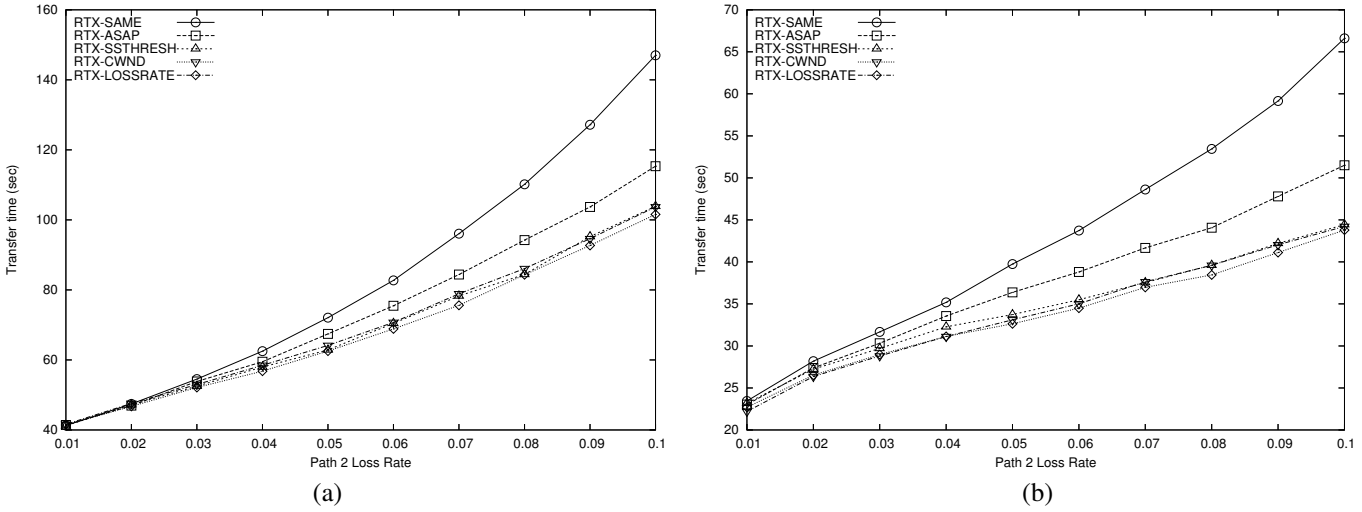

Fig. 13. Path 1 loss rate = 1%, CMT time to transfer 8MB file using: (a) rbuf=32KB, (b) rbuf=128KB

destinations most of the times a retransmission is triggered - (i) one retransmission is normally allowed to be sent to the destination that has just suffered loss, and (ii) the ack that triggers a retransmission (in case of fast retransmission) may have created cwnd space for the other destination. From (i) and (ii), RTX-ASAP has cwnd space availability on both destinations to send a retransmission. Consequently, RTX-ASAP randomly chooses a destination causing a reduction in timeouts over RTX-SAME which pins its TSNs to the same destination. The three loss rate based policies effectively choose the better destination to redirect retransmissions to, and thus show fewer timeouts than RTX-ASAP.

Figures 13(a) and (b) show performance of the retransmission policies with rbuf sizes of 32KB and 128KB respectively. Together with Figure 12(a), we can see that the smaller the rbuf, the more important the choice of retransmission policy. These results show that *a retransmission policy that considers loss outperforms policies that do not, particularly in the practical reality where rbuf is constrained.*

Figures 12 and 13 show that rbuf size has a strong impact on CMT performance. When CMT is used over paths with different loss rates, a constrained rbuf that is shared within an association causes performance degradation due to *rbuf blocking*. Degradation increases with a reduction in rbuf size, and/or an increase in the number of timeouts [26], [27]. Using loss rate based policies alleviates rbuf blocking since the number of timeouts is reduced. From Figure 12(a) and 13, as rbuf size decreases, rbuf blocking increases, and loss rate based policies perform increasingly better than the other policies. (See [26], [27] for an extensive discussion of rbuf blocking.)

Figures 12 and 13 suggest that any retransmission policy that takes loss rate into account will likely improve load distribution for both new transmissions and retransmissions. Retransmissions will be redirected to a lower loss rate path, avoiding inactive timeout recovery periods, and allowing new transmissions to be sent on the higher loss rate path, thus maintaining a flow of data on both paths. Policies that take loss rate into account avoid repeated retransmissions and timeouts - thus also improving the timeliness of data.

Of three loss rate based policies, the practical ones to implement are RTX-CWND and RTX-SSTHRESH. Both perform equally under all conditions considered. *Of these two policies,*

we arbitrarily select RTX-SSTHRESH as the recommended retransmission policy for CMT.

V. Summary and Discussion

We identified three negative side-effects of introducing CMT with SCTP, and proposed algorithms to avoid these side-effects. We compared CMT against AppStripe, an idealized data striping application, and showed that *a shared sequence space in CMT improves performance and increases resilience to reverse path loss*. We also presented and evaluated five retransmission policies for CMT. Our results reveal that a retransmission policy that considers loss rate performs better than one that does not, particularly in the practical reality where rbuf is constrained. *We recommend the RTX-SSTHRESH retransmission policy for CMT.*

CMT also inherently adds to SCTP's fault tolerance, which is a major motivation for, and benefit of multihoming. An SCTP sender gathers information about paths to alternate destination addresses through explicit probes. Since explicit probes are infrequent, a sender has inadequate information and consequently, is unable to make an informed decision about which destination to use when the primary destination becomes unreachable. A CMT sender avoids this problem because data sent concurrently on all paths act as frequent implicit probes, reflecting current conditions of paths to all destinations. This information will better assist a CMT sender in detecting and responding to network failures.

Alternative design

Another approach to accomplishing CMT would be to define a separate sequence space per destination. This solution simplifies some issues, but also introduces its own complications.

- What sequence number is used for a packet that is retransmitted to a destination other than the original? What happens to the sequence number used for the original destination (is it reused, or is it discarded thereby introducing a gap?) Any solution will likely require additional reliable signaling between sender and receiver.
- During association closure, the final sequence number must be agreed upon by sender and receiver to ensure complete reliable transfer. Introducing multiple sequence number spaces complicates this issue.
- Several mechanisms are understood with a single sequence space, for example, reneging. Managing per destination sequence numbering for these mechanisms requires careful examination.
- Separating sequence spaces causes separation of ack info per path. This separation cannot provide CMT's increased resilience to reverse path loss and reverse path failure as shown in Section IV-D.

We believe that the complexities introduced by such a design outweigh the benefits.

CMT in other environments

Small file transfers: Small file transfers (web transfers) suffer from the problem that they are more prone to timeouts because the number of packets in the transfer may be insufficient to trigger a fast retransmission. We have not tested CMT behavior with small files, but note the following. Spreading a small file transfer over multiple paths further decreases the ability to have fast retransmit, and thus will decrease expected throughput. At the same time, using the aggregated bandwidth of multiple paths should tend to increase throughput. We suspect that the performance degradation due to timeouts may dominate with small file transfers using CMT.

Failure scenarios: SCTP uses k consecutive timeouts as an indication of failure (recommended value of k is 6 [2]). CMT's failure detection/response mechanisms and latency are currently the same as those of SCTP. In the presence of failure, we observed that CMT's behavior is the same as that of SCTP with a failed primary path (brief transmission periods followed by long silence periods). We believe that further optimization is possible to improve CMT performance during failures, and is part of our future work.

Future work

Several items can be pursued in the future: (i) Our assumption of independent paths is a strong one. To drop this assumption, we plan to employ an end-to-end bottleneck detection technique in CMT [8]–[11]. (ii) Since CMT sends data to all receiver destinations, a CMT sender has more accurate information about paths to a receiver than SCTP does. We believe this information can be leveraged for improving failure detection and response latency. (iii) CMT may increase the end-to-end delay seen by an application due to increased reordering at a sender. We plan to study CMT's impact on this delay, and mechanisms to mitigate it. (iv) TCP aware load balancers at the network layer, which make sure that packets belonging to one connection take the same path, are commonplace. A performance comparison of CMT against such load balancing would be interesting. (v) With the RTX-SSTHRESH policy, retransmissions are all sent to the lower loss rate path. It would be interesting to investigate if this policy creates oscillatory behavior, as the extra retransmissions may increase the loss rate of the lower loss rate path.

VI. Related Work

A. Load Balancing at the Application Layer

Several applications [28], [29] use multiple TCP connections to increase throughput in high bandwidth networks. These applications load balance over the same path to a receiver, whereas CMT distributes data over multiple independent paths.

Content Networks [30] provide an infrastructure for *connection level load balancing* at the granularity of TCP connections. Connection level load balancing is useful for short TCP connections such as web requests and responses, but can be suboptimal for long bulk data transfers, where the server is constrained to a single path throughout the transfer. CMT provides load balancing *within* a transport connection.

B. Load Balancing at the Transport Layer

Load balancing is desirable at the transport layer since it has the most accurate information about end-to-end path(s). CMT uses loss and delay information for redirection of retransmissions - such decisions are best made in the transport layer. We believe that load balancing at the application layer

increases code redundancy and room for error by requiring independent implementations in each application.

Hsieh et al. [31] propose *pTCP (parallel TCP)* which provides an infrastructure for data striping within the transport layer. pTCP has two components - Striped connection Manager (SM) and TCP-virtual (TCP-v). The TCP-v's are separate connections that are managed by the SM. TCP-v probes the path and performs congestion control and loss detection/recovery, while the SM decides which data is sent on which TCP-v. This decoupling of functionality avoids some pitfalls of application layer approaches, and allows for intelligent scheduling of transmissions and retransmissions. A significant issue with pTCP is its complexity. As the authors note, maintenance of multiple Transmission Control Blocks at a sender can be a resource sink [31]. Implementation is also complex, since pTCP replicates transport layer functionality such as connection establishment/teardown and checksum calculations. Further, pTCP has several unresolved issues. If both sender and receiver are multihomed with two IP addresses each, pTCP does not address how a sender decides on which sender-receiver pairs to establish TCP connections - a complex problem. Plugging transport protocols into pTCP also requires non-trivial modifications to the transport protocols themselves. CMT, on the other hand, modifies SCTP, a transport protocol which has built-in mechanisms for multihoming.

mTCP [32], an effort parallel with ours, implements a transport layer solution to aggregate bandwidth across multiple end-to-end paths. mTCP, like CMT, uses a single sequence space across paths. mTCP significantly modifies TCP to use multiple paths provided by an overlay network (RON [33]), and also employs mechanisms to handle reordering side-effects. mTCP also uses a shared bottleneck detection mechanism to detect and respond to shared bottlenecks, but [32] lacks extensive testing of the proposed method. RON is assumed as the underlying routing layer, and is required for obtaining multiple paths; that is, mTCP cannot be used on an arbitrary IP network. On the other hand, CMT leverages SCTP's multihoming mechanisms and can be used on any IP network. mTCP also uses a single reverse path for ack traffic, thereby requiring additional mechanisms to detect failure of the single ack path, and causing performance degradation during failure.

Al et al. [34] suggest ideas for *load sharing* that requires additional metadata in the SCTP PDUs. We believe that the SCTP (and TCP-SACK) PDUs already contain sufficient information for the data sender to infer the per-path ordering information that [34] explicitly codes as metadata. [34] fails to suggest modified procedures for mechanisms which are immediately affected, such as initialization of the per-path sequence numbers, association initialization and shutdown procedures with multiple sequence numbering schemes, and response to reneging by a receiver. We have also seen that sharing sequence number space across paths improves performance whereas [34] uses a separate sequence number space per path, and will therefore not see CMT's performance benefits. Further, [34] assumes that the rbuf does not constrain a sender which is unrealistic in practice.

Argyriou et al. [35] provide techniques for *bandwidth aggregation* with SCTP, but do not present and analyze their protocol modifications to SCTP. The modified fast retransmission algorithm is simplistic and assumes information that is not available to an SCTP receiver. For instance, the implicit assumption that a receiver will be able to differentiate a packet loss from reordering is unrealistic. [35] also ignores the impact of a bounded rbuf.

C. Load Balancing at the Network Layer

Phatak and Goff [36] propose distributing data at the network layer transparent to the higher layers using IP-in-IP encapsulation. The authors identify conditions under which this mechanism avoids incorrect retransmission timeouts. The proposed solutions *assume* end-to-end delays are dominated by fixed transmission delay, and do not apply to propagation delay dominated paths, or paths with dynamically changing bandwidths and delays. CMT's algorithms do not require such assumptions, and will operate under dynamic and propagation delay dominated conditions.

Several proposals exist for *multipath routing* - routing packets from a source to a destination network over multiple paths. However, different paths are likely to exhibit different RTTs, thus introducing packet reordering. TCP's performance degrades in the presence of increased reordering. To enable optimal load balancing at intermediate routers without affecting end-to-end TCP performance, modifications to TCP have also been proposed [15]–[17], [37]. These proposals augment and/or modify TCP's congestion control mechanisms to cope with reordering introduced by network layer load balancing; the burden of actually using multiple paths in the network is left to the intermediate routers.

In the Internet, the end user has knowledge of, and control over, only the multihomed end hosts, not the intermediate routers. In such cases the end host cannot dictate or govern use of multiple paths in the network. But the end host can use multiple end-to-end paths available to the host [38], thus motivating CMT at the transport layer.

ACKNOWLEDGMENTS

We sincerely thank the editor and the anonymous reviewers for their careful reviews and useful suggestions.

DISCLAIMER

The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Army Research Laboratory or the U.S. Government.

REFERENCES

[1] R. Stewart, I. Arias-Rodriguez, K. Poon, A. Caro, and M. Tuexen, "Stream Control Transmission Protocol Specification Errata and Issues," draft-ietf-tsvwg-sctpimpguide-16.txt, Oct. 2005, (work in progress).

[2] R. Stewart, Q. Xie, K. Morneault, C. Sharp, H. Schwarzbauer, T. Taylor, I. Rytina, M. Kalla, L. Zhang, and V. Paxson, "Stream Control Transmission Protocol," RFC2960, Oct. 2000.

[3] E. Kohler, M. Handley, and S. Floyd, "Designing DCCP: Congestion Control Without Reliability," Tech. Rep., ICIR, 2004.

[4] S. Floyd, J. Mahdavi, M. Mathis, and M. Podolsky, "An Extension to the Selective Acknowledgement (SACK) Option for TCP," RFC2883, IETF, July 2000.

[5] M. Allman, V. Paxson, and W. Stevens, "TCP Congestion Control," RFC2581, IETF, Apr. 1999.

[6] K. D. Gradisching and M. Tuexen, "Signalling Transport Over IP-based Networks using IETF Standards," in *International Workshop on Design of Reliable Comm Networks (DRCN)*, Budapest, Oct. 2001.

[7] "Future combat systems website," www.globalsecurity.org/military/systems/ground/fcs.htm.

[8] M. S. Kim, T. Kim, Y. Shin, S. S. Lam, and E. J. Powers, "A Wavelet-based Approach to Detect Shared Congestion," in *ACM SIGCOMM*, Oregon, Aug. 2004.

[9] D. Rubenstein, J. Kurose, and D. Towsley, "Detecting Shared Congestion of Flows Via End-to-End Measurement," *IEEE/ACM Transactions on Networking*, vol. 10, no. 3, June 2002.

[10] D. Katabi, I. Bazzi, and X. Yang, "A Passive Approach for Detecting Shared Bottlenecks," in *ICCCN*, Arizona, Oct. 2001.

[11] K. Harfoush, A. Bestavros, and J. Byers, "Robust Identification of Shared Losses Using End-to-End Unicast Probes," in *ICNP 2000*, Osaka, Oct. 2000.

[12] A. Akella, S. Seshan, and H. Balakrishnan, "The Impact of False Sharing on Shared Congestion Management," in *ICNP*, Georgia, Nov. 2003.

[13] UC Berkeley, LBL, USC/ISI, and Xerox Parc, "ns-2 documentation and software," Version 2.1b8, 2001, www.isi.edu/nsnam/ns.

[14] A. Caro and J. Iyengar, "ns-2 SCTP module," Version 3.2, December 2002, http://pel.cis.udel.edu.

[15] E. Blanton and M. Allman, "On Making TCP More Robust to Packet Reordering," *ACM Computer Comm. Review*, vol. 32, no. 1, Jan. 2002.

[16] M. Zhang, B. Karp, S. Floyd, and L. Peterson, "RR-TCP: A Reordering-Robust TCP with DSACK," in *ICNP*, Georgia, Nov. 2003.

[17] S. Bohacek, J. Hespanha, J. Lee, C. Lim, and K. Obraczka, "TCP-PR: TCP for Persistent Packet Reordering," in *ICDCS*, Rhode Island, May 2003.

[18] J. Iyengar, K. Shah, P. Amer, and R. Stewart, "Concurrent Multipath Transfer Using SCTP Multihoming," in *SPECTS*, California, July 2004.

[19] J. Iyengar, A. Caro, P. Amer, G. Heinz, and R. Stewart, "Making SCTP More Robust to Changeover," in *SPECTS*, Montreal, July 2003.

[20] R. Ludwig and R. Katz, "The Eifel Algorithm: Making TCP Robust Against Spurious Retransmissions," *ACM Computer Communications Review*, vol. 30, no. 21, pp. 30–36, Jan. 2000.

[21] S. Ladha, S. Baucke, R. Ludwig, and P. Amer, "On Making SCTP Robust to Spurious Retransmissions," *ACM Computer Communication Review*, vol. 34, no. 2, pp. 123–135, Apr. 2004.

[22] N. Jani and Krishna Kant, "SCTP Performance in Data Center Environments," Tech. Rep., Intel Corporation, 2005.

[23] J. Iyengar, *End-to-end Load Balancing using Transport Layer Multihoming*, Ph.D. thesis, CISC Dept, University of Delaware, (in progress).

[24] J. Iyengar, P. Amer, and R. Stewart, "Retransmission Policies For Concurrent Multipath Transfer Using SCTP Multihoming," in *IEEE ICON*, Singapore, Nov. 2004.

[25] A. Caro, P. Amer, and R. Stewart, "Retrans Policies for Multihomed Transport Protocols," *Computer Communications*, (to appear).

[26] J. Iyengar, P. Amer, and R. Stewart, "Receive Buffer Blocking In Concurrent Multipath Transport," Tech Report TR2005-10, CIS Dept, University of Delaware, Jan. 2005.

[27] J. Iyengar, P. Amer, and R. Stewart, "Receive Buffer Blocking In Concurrent Multipath Transport," in *GLOBECOM*, Missouri, Nov. 2005.

[28] T. Hacker and B. Athey, "The End-to-End Performance Effects of Parallel TCP Sockets on a Lossy Wide-Area Network," in *IEEE IPDPS*, Florida, Apr. 2002.

[29] H. Sivakumar, S. Bailey, and R. Grossman, "PSockets: The Case For Application-Level Network Striping For Data Inttensive Applications Using High Speed Wide Area Networks," in *IEEE Supercomputing (SC)*, Texas, Nov. 2000.

[30] M. Day, B. Cain, G. Tomlinson, and P. Rzewski, "A Model For Content Internetworking (CDI)," RFC3466, IETF, Feb. 2003.

[31] H.Y. Hsieh and R. Sivakumar, "A Transport Layer Approach for Achieving Aggregate Bandwidths on Multihomed Mobile Hosts," in *MOBICOM*, Georgia, Sept. 2002.

[32] M. Zhang, J. Lai, A. Krishnamurthy, L. Peterson, and R. Wang, "A Transport Layer Approach for Improving End-to-End Performance and Robustness Using Redundant Paths," in *USENIX*, June 2004.

[33] D. Andersen, H. Balakrishnan, and R. Morris M. Kaashoek, "Resilient Overlay Networks," in *18th ACM Symposium on Operating Systems Principles (SOSP 2001)*, Banff, Oct. 2001.

[34] A. Abd El Al, T. Saadawi, and M. Lee, "LS-SCTP: A Bandwidth Aggregation Technique For Stream Control Transmission Protocol," *Computer Communications*, vol. 27, no. 10, 2004.

[35] A. Argyriou and V. Madisetti, "Bandwidth Aggregation With SCTP," in *IEEE GLOBECOM 2003*, California, Dec. 2003.

[36] D. S. Phatak and T. Goff, "A Novel Mechanism for Data Streaming Across Multiple IP Links for Improving Throughput and Reliability in Mobile Environments," in *INFOCOM*, New York, June 2002.

[37] M. Gerla, S. S. Lee, and G. Pau, "TCP Westwood Simulation Studies in Multiple-Path Cases," in *SPECTS*, California, July 2002.

[38] R. Teixeira, K. Marzullo, S. Savage, and G.M. Voelker, "In Search of Path Diversity in ISP Networks," in *USENIX/ACM Internet Measurement Conference*, Florida, Oct. 2003.