# Retransmission Policies For Concurrent Multipath Transfer Using SCTP Multihoming

**Janardhan R. Iyengar**
Protocol Engineering Lab
CIS Dept, University of Delaware
iyengar@cis.udel.edu

**Paul D. Amer**
Protocol Engineering Lab
CIS Dept, University of Delaware
amer@cis.udel.edu

**Randall Stewart**
Transport Technologies
Cisco Systems
rrs@cisco.com

*Abstract*— Concurrent Multipath Transfer (CMT) uses the Stream Control Transmission Protocol's (SCTP's) multihoming feature to distribute data across multiple end-to-end paths in a multihomed SCTP association. We propose five retransmission policies for CMT. We demonstrate the occurrence of spurious retransmissions in CMT with all of the five policies, and propose an amendment to the timeout retransmission mechanism to avoid spurious retransmissions. We also modify the Cwnd Update for CMT (CUC) algorithm to allow better cwnd growth in CMT with the different retransmission policies. We then evaluate the retransmission policies using ns-2 simulations, and discuss the distributions of traffic that result. We operate under the strong assumptions that the receiver's advertised window does not constrain the sender, and that the bottleneck queues on the end-to-end paths used in CMT are independent.

## I. INTRODUCTION

Multihoming among networked machines and devices is a technologically feasible and increasingly economical proposition. 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 can be expected to be the rule rather than the exception in the near future. When fault tolerance is crucial, multihoming will become an essential architectural design feature. Multihomed nodes may be simultaneously connected through multiple access technologies, and even multiple end-to-end paths to increase resilience to path failure. Multiple active interfaces suggest the simultaneous existence of multiple paths between the multihomed hosts. Previous work has proposed using these multiple paths between multihomed source and destination hosts through *Concurrent Multipath Transfer (CMT)* 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. Increasing economic feasibility and a desire for networked applications to be fault tolerant at an end-to-end

level, have brought multihoming within the purview of the transport layer. While concurrency can be arranged at the application layer, CMT at the transport layer is desirable since it has finest information about the end-to-end path(s). CMT at the application layer increases redundancy and room for error by requiring a separate implementation by each application programmer. Further, complexity at the transport-application interface will increase due to continuous information exchange between the transport and the application.

The Stream Control Transmission Protocol (SCTP) [9], [10] 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 an SCTP sender to send data to a multihomed receiver through different destination addresses. Simultaneous transfer of new data to multiple destination addresses is currently not allowed in SCTP due primarily to insufficient research. This research attempts to provide that needed work.

Previous work in CMT specified three algorithms for SCTP resulting in CMT$_{scd}$ - a protocol that uses SCTP's multihoming feature for *correctly* transferring data between multihomed end hosts using multiple separate end-to-end paths [8]. Since CMT$_{scd}$[1] concurrently transmits data to multiple destinations, the sender could send retransmissions to one of several destinations that are receiving new transmissions. In this paper, we propose and evaluate five retransmission policies for deciding which destination should be used. We present some modifications to SCTP and the Cwnd Update for CMT (CUC) algorithm [8] to prevent side-effects of the different retransmission policies. We also present a performance analysis of CMT vs. AppStripe (Application Striping), an ideal simulated application. In this evaluation, we operate under the strong assumptions that the receiver's advertised window does not constrain the sender, and that the bottleneck queues on the end-to-end paths used in CMT are independent of each other. Future work will relax these constraints.

Section II describes the different retransmission policies. Section III presents our simulation topology and describe our evaluation methodology. Section IV motivates and presents modifications to the Cwnd Update for CMT (CUC) algorithm (Section IV-A) and to SCTP's timeout recovery mechanism (Section IV-B). Section V presents our simulation results and a detailed analysis of the different retransmission policies.

[1]Henceforth, we will refer to "CMT$_{scd}$" as simply "CMT"

This section also presents a performance analysis of CMT vs. AppStripe. Section VI concludes the paper with notes on the current focus of our work in the larger framework of CMT.

A note on language and terminology. Since an SCTP association allows multihomed source and destination endpoints, an SCTP source maintains per destination, a separate congestion window (cwnd), slow start threshold (ssthresh), retransmission timer, and roundtrip time (RTT) estimates among other state. Thus, 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 retransmission timer maintained for destination X at the sender. 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". SCTP acks carry cumulative and selective ack information and are called "SACKs". We use the terms "SACKs" and "acks" interchangeably.

## II. CMT RETRANSMISSION POLICIES

We present five retransmission policies for CMT. For most, a retransmission may be sent to a destination other than the one used for the original transmission. Previous work on retransmission policies for SCTP [5] shows that sending retransmissions to an alternate destination degrades performance severely. The primary reasons for the degradation in performance are related to lack of sufficient traffic on alternate paths. We point out to the reader that with CMT, data is concurrently sent on all paths, thus rendering the results in [5] not applicable. In CMT, a sender has much more information about the paths to the destination host than an SCTP sender (without CMT). A CMT sender uses this information to influence the selection of a destination for a retransmission. The five different retransmission policies for CMT are:

- **RTX-SAME** - Once a new data chunk is scheduled and sent to a destination, all retransmissions of the chunk thereafter are sent to the same destination (until the destination is deemed *inactive* due to failure [10]).
- **RTX-ASAP** - A retransmission of a data chunk is sent to any destination for which the sender has cwnd space available at the time the retransmission needs to be sent. If the sender has available cwnd space for multiple destinations, the destination is chosen randomly from among those available.
- **RTX-CWND** - A retransmission of a data chunk is sent to the destination for which the sender has the largest cwnd. A tie is broken by random selection.
- **RTX-SSTHRESH** - A retransmission of a data chunk is sent to the destination for which the sender has the largest ssthresh. A tie is broken by random selection.
- **RTX-LOSSRATE** - A retransmission of a data chunk is sent to the destination with the lowest loss rate path. If multiple destinations have the same loss rate, then a destination is selected randomly from among them. This policy is an ideal case since the loss rate is known, and not measured in the simulations.

Of the policies, RTX-SAME is simplest. RTX-ASAP is a "hot-potato" retransmission policy - the goal is to 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 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[2]. RTX-LOSSRATE uses information about loss rate provided by an "oracle" - information that RTX-CWND and RTX-SSTHRESH implicitly infer and use. This policy quantifies the impractical case where the sender knows the exact loss rate of the paths being used for CMT, and selects the path with the lowest loss rate to send all retransmissions.

We initially hypothesized that retransmission policies that take loss rate into account would outperform ones that do not, because sending retransmissions on a lower loss rate path would improve chances of a retransmission getting through. Consequently, performance would improve due to fewer timeouts, particularly those due to loss of retransmissions.

## III. METHODOLOGY

We evaluate the different retransmission policies using the University of Delaware's SCTP module for the ns-2 simulator [3], [6]. We have incorporated CMT and the different retransmission policies in the SCTP module. As a reference for performance, we use *AppStripe* - a hypothetical multihome-aware application that achieves the highest throughput achievable by an application that distributes data across multiple SCTP associations (see Figure 1). End-to-end load sharing is performed at the application layer by AppStripe, and at the transport layer by CMT [7].
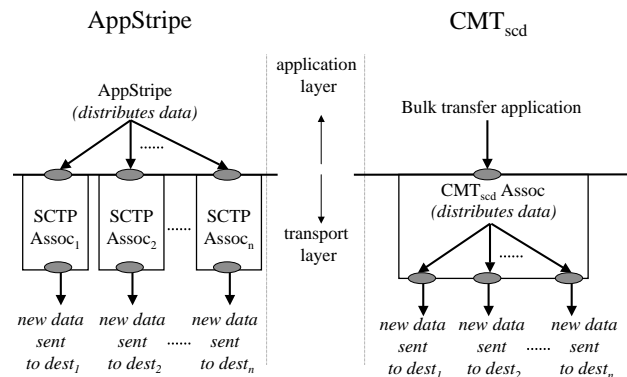


Fig. 1. Schematic - AppStripe and CMT

The simulation topology (see Figure 2) 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

---

[2]We assume that the reader is familiar with SCTP and TCP congestion control, the variables cwnd and ssthresh, and their dynamics [2], [10].

simulation evaluation provides insight into the fundamental differences between the retransmission policies, and their performance in a constrained environment. The loss rate on Path 1 is maintained at 1%, and on Path 2 is varied from 1 to 10%. The loss rates apply to both forward and reverse paths - a loss rate of 1% means a forward path loss rate of 1%, and a reverse path loss rate of 1%.
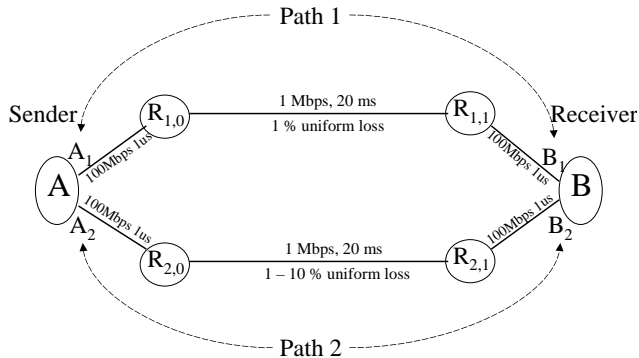


Fig. 2.   Simulation topology used for evaluation

## IV. MODIFICATIONS TO PROTOCOL MECHANISMS

### A. CUCv2: Modified CUC Algorithm

SCTP's current design principle (as with TCP) assumes that a SACK's cumulative ack (cum ack), which tracks the latest Transmission Sequence Number (TSN)[3] received in-order at the receiver, applies to an entire association and 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 when CMT is not considered.

Since CMT uses multiple destinations simultaneously, cwnd growth in CMT demands tracking the latest TSN received in-order *per-destination* since TSNs may be arbitrarily distributed across destinations depending on the scheduling algorithm. This per-destination ordering information is not present in a SACK. A sender must infer cum ack per destination, possibly through SACKs and history information in the retransmission queue - the Cwnd Update for CMT (CUC) algorithm achieves this end [8]. The CUC algorithm enables correct cwnd updates in the face of increased reordering due to CMT.

The CUC 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. The CUC algorithm assumes that retransmissions are sent to the same destination as the original transmission, and so the per-destination pseudo-cumack 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 transmission, the current CUC algorithm cannot

---

[3]TSN serves the the same function in SCTP as the sequence number does in TCP.

faithfully track the left edge on either destination. We propose a modification to the CUC algorithm to permit the different retransmission policies. The modified CUC algorithm, named CUCv2 (CUC version 2) is shown in Figure 3.

The crux of the modification is in recognizing that of the TSNs outstanding on a destination, a distinction can be made - the ones that have been retransmitted, and those that have not been retransmitted. The CUCv2 algorithm maintains two left edges for these two sets of TSNs - *exp-pseudo-cumack* and *exp-rtx-pseudo-cumack*. Whenever either of the left edges moves, a cwnd update is triggered. In CUCv2 (see Figure 3), lines 2(iv), 2(v), 3(iv) and 3(v) have been added, and lines 3(ii) and 4 have been modified from the CUC algorithm [8].

### B. Spurious Timeout Retransmissions

When a timeout occurs, an SCTP sender is expected to bundle as many of the earliest TSNs outstanding on the destination for which the timeout occurred as can fit in a packet and send the packet. As per RFC2960, if more TSNs are outstanding on that destination, these TSNs "should be marked for retransmission and sent as soon as cwnd allows (normally when a SACK arrives)". The cwnd is also reset to 1 Maximum Segment Size (MSS) for the destination on which a timeout occurs, allowing only one MSS sized packet in flight. Thus, the next TSN(s) marked for retransmission can be sent only when a SACK arrives for the TSN(s) retransmitted first.

A timeout retransmission can occur in SCTP (as in TCP) due to one of several reasons. One of the reasons is loss of the fast retransmission of a TSN[4]. Consider Figure 4[5]. When a timeout occurs due to loss of a fast retransmission, it is quite likely that some TSNs that were just sent to the destination on which the timeout occurred are awaiting SACKs (in the Figure, TSNs Y+2 and Y+3). These TSNs get incorrectly marked for retransmission on timeout. With the different retransmission policies in CMT, the retransmissions may be sent to a different destination than the original transmission; if cwnd space on a destination is available, possibly due to receipt of a SACK on that destination, TSNs marked for retransmission may be sent to that destination. In Figure 4, incorrect retransmissions of TSNs Y+2 and Y+3 are sent to destination $B_1$, on receipt of SACKs freeing up cwnd space for destination $B_1$. This problem is exacerbated in CMT, as shown through this illustration, due to the possibility of sending data (including retransmissions) to multiple destinations concurrently.

We studied the occurrence of such spurious retransmissions with the different retransmission policies in CMT. The simulation topology used is the one described in Section III. Figure 5(a) shows the number of retransmissions relative to the number of actual packet drops at the router in our simulations. Ideally, the number of retransmissions should be exactly equal

---

[4]The Multiple Fast Retransmit (MFR) algorithm allows recovery using fast retransmission multiple times on the same TSN [4]. The MFR algorithm has not been ported to CMT, and so the only recovery mechanism possible from the loss of a fast retransmission currently in CMT, as is the case currently in SCTP and TCP, is a timeout recovery.

[5]This figure illustrates the point, and may have discrepancies in scale and/or details which may be overlooked.

**At beginning of an association [Sender side behavior]:**
 ∀ destinations d, reset
   $d.find\_pseudo\_cumack$ = TRUE;
   $d.find\_rtx\_pseudo\_cumack$ = TRUE;
**On receipt of a SACK [Sender side behavior]:**
 1) ∀ 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;
    (iv) set $d_c.find\_rtx\_pseudo\_cumack$ = TRUE;
    (v) set $d_c.new\_rtx\_pseudo\_cumack$ = 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
     **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 according to [9], [10];

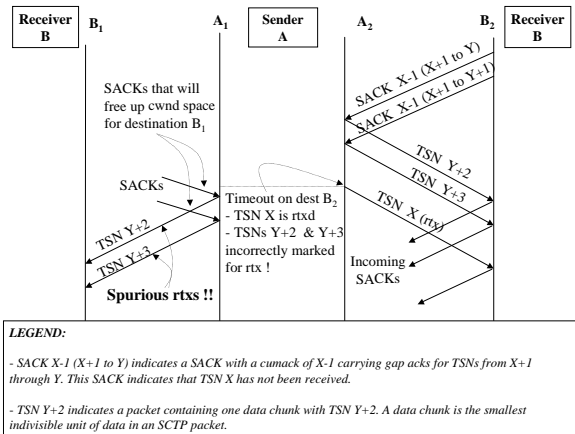Fig. 3. CUCv2 Algorithm - Modified Cwnd Update for CMT (CUC) Algorithm



Fig. 4. Example of spurious retransmissions after timeout in CMT

to the number of packet drops at the router; the curves should be straight lines at 1 on the Y-axis. The figure shows that spurious retransmissions occur commonly in CMT with the different retransmission policies.

We propose a heuristic to avoid these spurious retransmissions. Our solution assumes that a timeout cannot be triggered on a TSN until the TSN has been outstanding for at least one Round-Trip Time (RTT). Thus, if a timeout is triggered, TSNs which are sent within an RTT in the past are not marked for retransmission. We use an average measure of the RTT for this purpose - the Smoothed RTT (SRTT) which is maintained at the 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 5(b) shows how the application of this heuristic drastically reduces the number of spurious retransmissions.
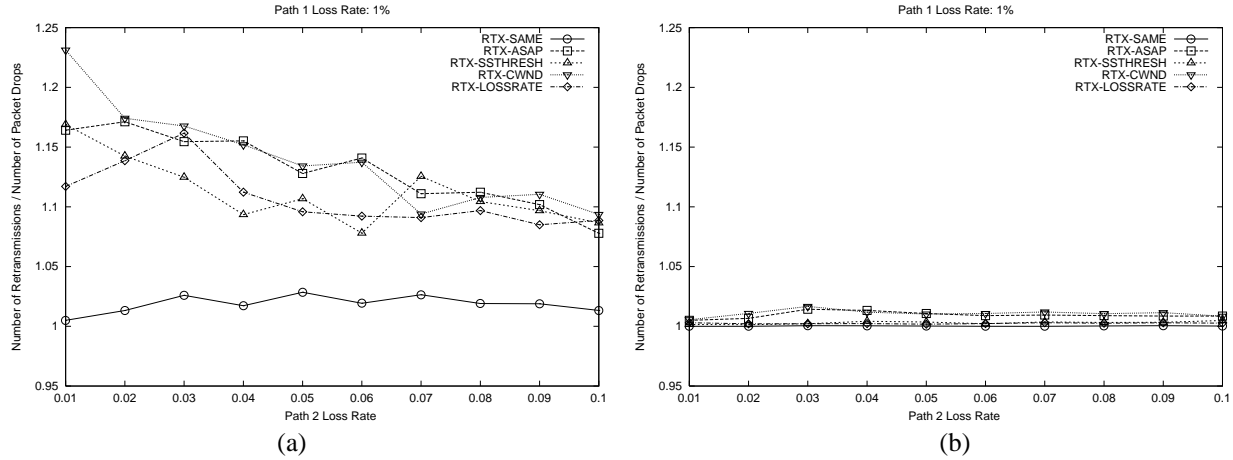
Fig. 5. Spurious retransmissions in CMT: (a) Without SRTT heuristic (b) With SRTT heuristic

## V. EVALUATION OF RETRANSMISSION POLICIES

Figure 6 shows the time taken to transfer an 8MB file using CMT with the five retransmission policies, and using AppStripe. The x-axis represents different loss rates on Path 2. These results are averaged over 30 simulation runs per point. Overall, AppStripe (× in Figure 6) performs worst, RTX-SAME (⊖) performs better than AppStripe, RTX-ASAP (□) performs better than RTX-SAME, and almost as good as the best performing group - RTX-SSTHRESH (△), RTX-CWND (▽) and RTX-LOSSRATE (◇). We now discuss these performance differences.
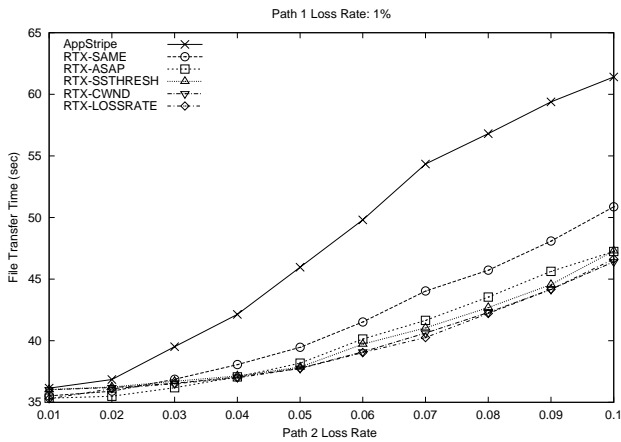


Fig. 6. Performance comparison of AppStripe and CMT with different retransmission policies

### A. Performance of CMT vs. AppStripe

CMT using any of the retransmission policies performs better than, if not the same as, AppStripe. As the loss rate on Path 2 increases, CMT performs increasingly better for two reasons. First, CMT gets better overall cwnd growth than AppStripe in the slow start phase when delayed acks are used [8]. This phenomenon allows the CMT sender to increase its overall cwnd faster than AppStripe during the slow start phase. Note that the overall cwnd increase is faster, yet TCP-friendly with CMT. As the loss rate increases, number of timeouts increases, and since the sender enters slow start after each timeout, the sender spends more time overall in slow start. This phenomenon explains part of the improvement observed with CMT under high loss rates.

Second, 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 the multiple paths used in the association, whereas AppStripe by its design must use a separate SCTP association per path, and hence uses a separate sequence space per path. Since SCTP acks are cumulative, sharing of sequence spaces across paths helps an SCTP 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 SCTP association in AppStripe uses a single return path for sending acks. For instance, if an ack is lost on one return path, traffic on the other path will cause the CMT receiver to respond with acks on the other return path. These cumulative acks carry the information that was lost due to the ack loss. Since AppStripe uses an independent SCTP association on each path, acks for one association cannot help acks on the other association. This phenomenon is solely due to the fact that CMT shares sequence space across the paths.

To support our analysis, Figure 7(a) shows the time taken to transfer an 8MB file, when delayed acks are turned OFF (one ack is sent per packet received) and the topology is modified to have loss only on the forward path - there is no reverse path loss. These results (and also the rest of the results in Figure 7) are averaged over 30 simulation runs per point. This graph shows very little performance difference between CMT with RTX-SAME policy and AppStripe, thus supporting our analysis of the performance difference between AppStripe
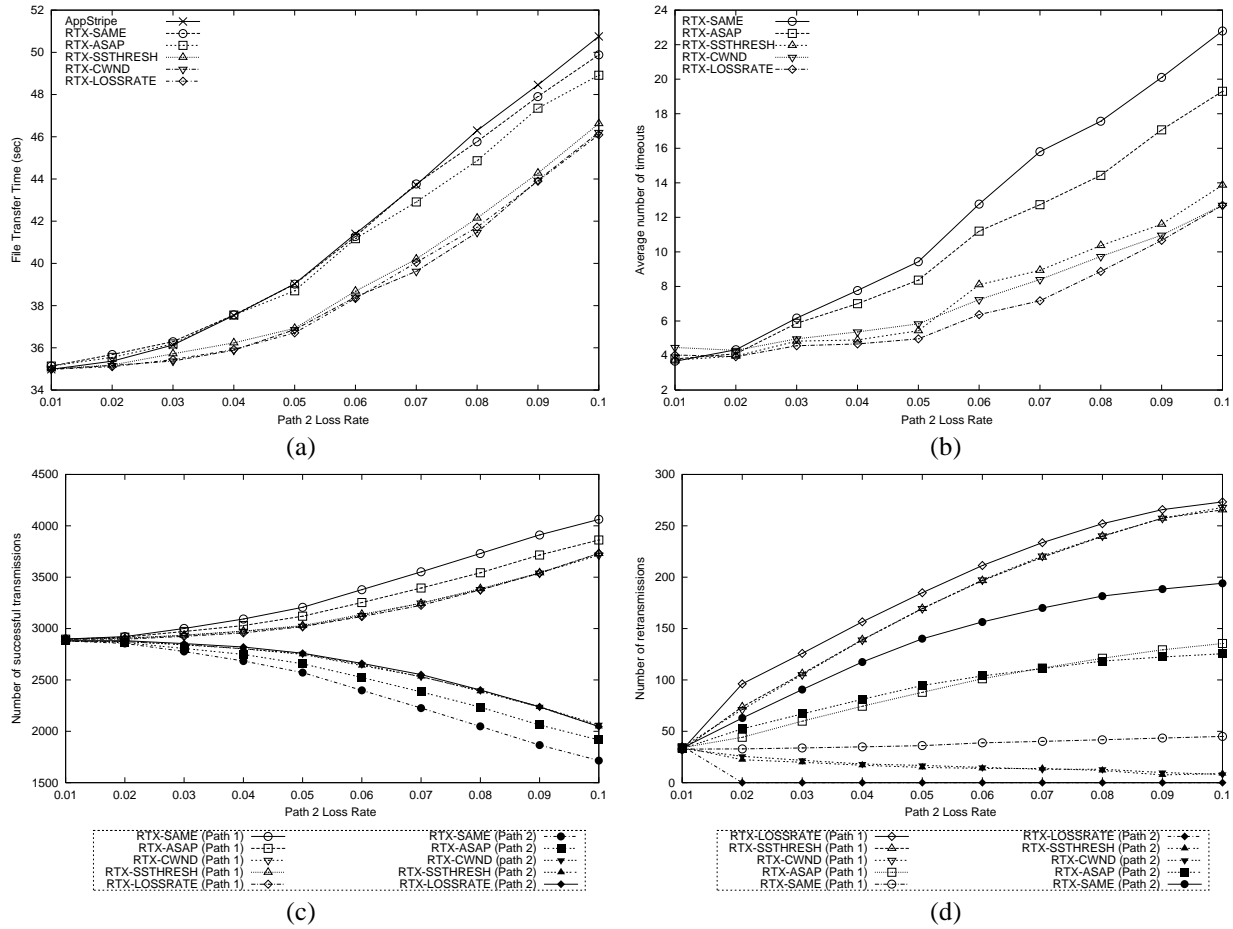
Fig. 7. With Path 1 loss rate = 1%: (a) AppStripe and CMT: Performance with no reverse path loss and no delayed acks, (b) Number of retransmission timeouts for CMT with different retransmission policies, (c) Distribution of successful transmissions for CMT with different retransmission policies, (d) Distribution of retransmissions for CMT with different retransmission policies

and CMT. These results demonstrate that sharing sequence space across paths is more beneficial than maintaining separate sequence spaces per path [1].

### B. Performance of different retransmission policies for CMT

Of the retransmission policies used for CMT, 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. The improvement in using RTX-LOSSRATE, RTX-CWND, or RTX-SSTHRESH reinforces our intuition - taking path loss rate into consideration while deciding the retransmission destination improves the chances of a retransmission getting through, and improves overall performance. RTX-ASAP performs as good as the best performing policies - note that RTX-ASAP does not explicitly consider loss rate in its decision, but sends the retransmissions as soon as possible, to any destination that has available cwnd space.

Figure 7(b) shows the number of timeouts with the different policies. RTX-SAME has the most and RTX-LOSSRATE,

RTX-CWND and RTX-SSTHRESH have the fewest. RTX-ASAP figures between these policies in number of timeouts. Policies that take loss rate into consideration do best at avoiding timeouts. The number of timeouts is much larger with RTX-SAME as compared to any of RTX-LOSSRATE, RTX-CWND or RTX-SSTHRESH. Thus, the number of timeouts, and hence the amount of time spent in timeout recovery for RTX-SAME explains its poorer performance.

Figure 7(c) shows the distribution of successful transmissions, and Figure 7(d) shows the distribution of retransmissions over the two paths. The numbers of transmissions or retransmissions sent over Path 1 are shown with hollow shapes (⊖), and those sent over Path 2 are shown in corresponding solids (●).

Figure 7(c) shows that of the policies, RTX-SAME gets most of its data through to the destination via Path 1 (the lower loss rate path), and the loss rate based policies divide the successful transmission more evenly. RTX-ASAP is roughly midway between RTX-SAME and the group of RTX-LOSSRATE, RTX-SSTHRESH and RTX-CWND. Figure 7(d)

shows that of the policies, RTX-LOSSRATE as expected always sends the retransmissions via Path 1 (the lower loss rate path) except when the loss rates are the same for both paths at 1%, where the retransmission distribution is even. RTX-SSTHRESH and RTX-CWND send most retransmissions via Path 1, while some retransmissions are still sent via Path 2 (the higher loss rate path). On the other hand, RTX-SAME sends most of its retransmissions via Path 2. The reason for this difference is that RTX-SSTHRESH and RTX-CWND divert retransmissions for most of the losses occurring on *both* paths to Path 1, whereas RTX-SAME maintains the retransmissions for losses occurring on Path 2 to the same path. RTX-ASAP distributes the retransmissions fairly evenly between the two paths. This behavior with RTX-ASAP is because most of the times that a retransmission has to be sent, cwnd space is available for both destinations for the retransmission, thus resulting in a random selection of the retransmission destination. To better explain this behavior, we point out that SCTP allows sending one MTU worth of retransmission to the destination which has suffered a loss, even if the cwnd for that destination has no space. Thus, immediately after a cwnd reduction for a destination due to loss, the sender still has space for one MTU of retransmission that can be sent to that destination. This extra space creates multiple choices for the sender during most retransmissions thus resulting in random selection of the retransmission destination most of the times.

Figure 7(c) shows that the amount of data that gets through to the destination via Path 2 (the higher loss rate path) is much lesser with RTX-SAME as compared to the other policies. At the same time, Figure 7(d) shows that the number of retransmissions that are sent via Path 2 is higher than with the other policies. This higher number of retransmissions on Path 2 given that a lesser number of packets actually get through on Path 2 indicates that the same TSNs get dropped repeatedly.

RTX-SAME suffers a large number of timeouts (as seen in Figure 7(b)), but these timeouts are dominated by repeated timeouts on the same TSNs. These timeouts cause a sender to wait for a retransmission timer to expire for destination $B_2$, thus keeping Path 2 idle for a significant portion of the transfer. This behavior causes the sender using the RTX-SAME policy to schedule more new transmissions on Path 1 (the lower loss rate path), thus distributing the load of new transmissions automatically with a bias towards the lower loss rate path. This automatic shift of load towards the lesser loss rate path may benefit the RTX-SAME policy, but the higher number of timeouts, and hence greater time spent waiting for retransmission timer expiration and in timeout recovery causes degradation in overall performance.

Figure 7(d) shows RTX-ASAP distributing the retransmission load fairly evenly between the paths. This distribution explains why RTX-ASAP sees fewer timeouts than RTX-SAME, but more timeouts than RTX-LOSSRATE, RTX-CWND and RTX-SSTHRESH. The fewer number of timeouts seen by RTX-ASAP improves its performance over RTX-SAME.

## VI. CONCLUSION AND FUTURE WORK

In this paper, we presented and evaluated several retransmission policies for CMT. Our overall analysis reveals that RTX-LOSSRATE, RTX-SSTHRESH and RTX-CWND are clear winners. RTX-SAME performs the worst in terms of both total time taken for a transfer, and number of retransmission timeouts. RTX-ASAP performs better than RTX-SAME and about the same or worse than RTX-LOSSRATE, RTX-SSTHRESH and RTX-CWND, which are the best performing.

Though further investigation is needed to recommend a general retransmission policy, our investigation reveals that any retransmission policy that takes loss rate into account will likely improve load distribution for both new transmissions and retransmissions. Policies that take loss rate into account avoid repeated retransmissions and timeouts - thus improving the timeliness of data. We believe that with a constrained receiver window and/or with path failure, a policy that cannot avoid repeated retransmissions will suffer.

In the larger framework of CMT, we are currently working on sharing mechanisms for the receiver's advertised window when the window is limited. We have so far assumed that the receiver's advertised window does not constrain the sender; we will now relax this constraint. We propose to evaluate the different retransmission policies in the face of a constrained receiver's advertised window. In the future, we will evaluate these different policies in the presence of path failures.

### 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] A. Abd El Al, T. Saadawi, and M. Lee. LS-SCTP: A Bandwidth Aggregation Technique For Stream Control Transmission Protocol. *Computer Communications*, 27(10), 2004.

[2] M. Allman, V. Paxson, and W. Stevens. TCP Congestion Control. RFC2581, Internet Engineering Task Force (IETF), April 1999.

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

[4] A. Caro, P. Amer, J. Iyengar, and R. Stewart. Retransmission Policies with Transport Layer Multihoming. In *ICON 2003*, Sydney, Australia, September 2003.

[5] A. Caro, P. Amer, and R. Stewart. Transport Layer Multihoming for Fault Tolerance in FCS Networks. In *MILCOM 2003*, Boston, MA, October 2003.

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

[7] J. Iyengar, P. Amer, and R. Stewart. Concurrent Multipath Transfer Using Transport Layer Multihoming: Performance Under Varying Bandwidth Proportions. Tech report, CIS Dept., University of Delaware, October 2004.

[8] J. Iyengar, K. Shah, P. Amer, and R. Stewart. Concurrent Multipath Transfer Using SCTP Multihoming. In *SPECTS 2004*, San Jose, California, July 2004.

[9] R. Stewart, L. Ong, I. Arias-Rodriguez, K. Poon, A. Caro, and M. Tuexen. Stream Control Transmission Protocol (SCTP) Implementer's Guide. draft-ietf-tsvwg-sctpimpguide-10.txt, Internet Draft (work in progress), Internet Engineering Task Force (IETF), November 2003.

[10] 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, Internet Engineering Task Force (IETF), October 2000.