A Transport Layer Approach for Improving End-to-End Performance and Robustness Using Redundant Paths

Ming Zhang, Junwen Lai
Princeton University
{*mzhang, lai*}*@cs.princeton.edu*

Arvind Krishnamurthy
Yale University
arvind@cs.yale.edu

Larry Peterson, Randolph Wang
Princeton University
{*llp, rywang*}*@cs.princeton.edu*

Abstract

Recent work on Internet measurement and overlay networks has shown that redundant paths are common between pairs of hosts and that one can often achieve better end-to-end performance by adaptively choosing an alternate path [8, 27]. In this paper, we propose an end-to-end transport layer protocol, *mTCP*, which can aggregate the available bandwidth of those redundant paths in parallel. By striping one flow's packets across multiple paths, mTCP can not only obtain higher end-to-end throughput but also be more robust under path failures. When some paths fail, mTCP can continue sending packets on other paths, and the recovery process normally takes only a few seconds. Because mTCP could obtain an unfair share of bandwidth under shared congestion, we integrate a shared congestion detection mechanism into our system. It allows us to dynamically detect and suppress paths with shared congestion so as to alleviate the aggressiveness problem. mTCP can also passively monitor the performance of several paths in parallel and discover better paths than the path provided by the underlying routing infrastructure. We also propose a heuristic to find disjoint paths between pairs of nodes using traceroute. We have implemented our system on top of overlay networks and evaluated it in both PlanetLab and Emulab.

1 Introduction

Recent work on Internet measurement and overlay networks has shown that redundant paths are common between pairs of hosts [27]. One can often achieve better end-to-end performance by adaptively choosing an alternate path other than the direct Internet path [8]. At the same time, stub networks are increasingly resorting to multihoming to improve the reliability of their network connectivity [5]. The reliability is usually achieved by having sufficiently disjoint paths to the destinations of interest. Moreover, with the rapid growth of wireless coverage, mobile users can often have access to multiple communication channels simultaneously [17, 21]. All of the above means redundant paths are quite common between pairs of hosts.

Our goal is to design an end-to-end transport layer protocol (mTCP) which can not only aggregate the bandwidth of several paths concurrently but also enhance the robustness under path failures by taking advantage of redundant paths. Compared with the conventional single-path TCP flows, mTCP stripes a flow's packets across several paths. It can be viewed as a group of single-path subflows, with each subflow going through a separate path. A number of challenges arise when one attempts to develop a transport layer protocol that utilizes multiple paths simultaneously.

First, the traditional congestion control mechanism in TCP needs to be modified in order to fully exploit the aggregate bandwidth of the multiple paths used by mTCP. If one were to use the traditional congestion control mechanism on the whole flow (wherein the congestion window for the entire flow is halved for every packet loss), the mTCP connection would remain under-utilized in the following scenario. Assume that one of the sub-flows is using a heavily congested path. Severe packet losses on that path will result in repeated shrinkage of the window, resulting in sub-optimal use of the other sub-flows that remain uncongested. mTCP therefore needs to perform congestion control on each subflow independently so as to minimize the negative influence of congested subflows.

Second, paths may fail during data transmission. mTCP should not stall as long as there exists one operational path. It should be able to quickly detect failed paths and continue sending or retransmitting packets on live paths.

Third, when subflows of a mTCP flow share congested links, the whole flow can obtain an unfairly larger share of bandwidth than other single-path TCP flows since each subflow behaves as a single-path TCP flow. To alleviate this aggressiveness problem, we integrate a shared congestion detection mechanism into our system so as to identify and suppress subflows that traverse the same set of congested links.

Finally, there might exist many alternate paths between a pair of source and destination nodes. We want to select a small number of candidate paths for mTCP flows since it is impractical to use all the paths simultaneously. We use a heuristic to identify and select disjoint paths using traceroute. This can minimize the possibility of shared congestion and concurrent path failures.

We do note that mTCP provides performance benefits only when there are multiple paths that do not share congested physical links. Although in today's Internet, many congested or bottleneck links lie at the edge of the network and limit the performance benefits of mTCP, this is likely to change with the growing popularity of high speed Internet access. In [6], Akella, Seshan and Shaikh measured a diverse set of paths traversing Tier-1, Tier-2, Tier-3, and Tier-4 ISPs. They discovered that about 50% of the paths have bottleneck links that are located within ISPs or between neighboring ISPs. The available capacity of those bottleneck links are less than 50Mbps, well below the 100Mbps Ethernet speed that could be attained within local area networks. In such situations, mTCP is likely to find multiple paths that are not constrained by the bandwidths attainable at the edge of the network. Other scenarios where mTCP could find an useful set of disjoint paths to transmit data include multi-homed clients [5].

We also note that our focus is on improving the performance and robustness for large data transfers. Although most flows on the Internet are small, most of the traffic on the Internet is contributed by a small percentage of big flows [34, 13]. Therefore, improving the performance of such big flows is important. Additionally, while small flows might not necessarily benefit from throughput aggregation, they might still benefit from mTCP's robustness that would enable them to quickly detect and recover from path failures.

To the best of our knowledge, we are the first to implement and evaluate a transport layer protocol that can utilize redundant paths concurrently in real systems. We try to provide a comprehensive design that addresses the inter-related issues of sub-flow congestion control, unfair use of congested links, path selection, and recovery from path failures. We believe that it is beneficial to tackle all of these issues in a single tightly-coupled system. For instance, suboptimal decisions from the path selection mechanism could be corrected by a mechanism that detects the use of shared congested links. Alternately, shared congestions could be detected easily by monitoring TCP events (such as fast retransmits) without requiring separate probe messages. Furthermore, the system could quickly recover from failures by maintaining and transmitting along multiple paths. Finally, a mTCP flow can passively monitor the performance of several paths in parallel and estimate their available bandwidths. The bandwidth estimates are typically more accurate than the estimates provided by the underlying overlay routing mechanisms. This in turn can help select better paths.

The rest of the paper is organized as follows: Section 2 will describe related work. Section 3 will discuss the specific design problems of mTCP in detail. Section 4 briefly describes the implementation of our system. Section 5 demonstrates the results from experiments conducted on PlanetLab [25] and Emulab [2]. Finally, Section 6 concludes.

2 Related Work

The general idea of using multiple paths in a network to obtain better performance has been explored in a number of different research efforts. We briefly discuss how our work relates to previous research in this area.

One area of related work is the use of striping [32] or *inverse-multiplexing* in link-layer protocols to enhance the throughput by aggregating the bandwidth of different links. Adiseshu *et al* [4], Duncanson *et al* [12] and Snoeren [29] provide link-striping algorithms that address the issues of load-balancing over multiple paths and preserving in-order delivery of packets to the receiver. These efforts propose transparent use of link-level striping without requiring any changes to the upper layers of the protocol stack.

Another area of related work is the use of multiple paths by transport protocols to enhance reliability [10, 24, 20, 9]. Banerjea [10] proposed the use of redundant paths in his *dispersity routing* scheme to improve reliable packet delivery for real-time applications. Nguyen and Zakhor [24] also propose the use of multiple paths to reduce packet losses for delay-sensitive applications. They employ UDP streams to route data whose redundancy is enhanced through forward error correction techniques.

The most directly relevant related work is the use of multiple paths for improving the throughput or robustness of end-to-end connections. Several application-layer approaches have been proposed to improve throughput by opening multiple TCP sockets concurrently [7, 14, 19, 28], but the multiple TCP connections utilize the same physical path. These approaches obtain an unfair share of the throughput of congested links and seem to primarily benefit from increased window sizes over long-latency connections. SCTP [31] is a reliable transport protocol which supports multiple streams across different paths. However, it does not provide strict ordering across all the streams, and it cannot utilize the aggregate bandwidth on multiple paths as we do. The systems that are closest to what is described in this paper is R-MTP [21] and pTCP [16]. R-MTP provides bandwidth aggregation by striping packets across multiple paths based on bandwidth estimation. It estimates the available bandwidth by periodically probing the paths. As a result, its performance greatly relies on the accuracy of the estimation and the probing rate. It could suffer from bandwidth fluctuation as shown in [16]. pTCP uses multiple paths to transmit TCP streams and describes mechanisms for striping packets across the different paths. They however assume the existence of a separate mechanism that identifies what paths to use for their pTCP connections, and they also do not address the issues of recovering from path failures or obtaining an unfair share of the throughput of congested links if the paths are not disjoint.

Their study is also limited to simulations using *ns*[3].

3 Design

The design of our system seeks to satisfy three goals. First, given several paths, mTCP should be able to make full use of the available bandwidth on those paths. Second, when mTCP uses paths with shared congested links, it should be able to alleviate the aggressiveness problem by suppressing some of the paths. Third, when some paths fail, mTCP should quickly detect and recover from the failures.

3.1 Transport Layer Protocol

mTCP provides the same semantics to applications as TCP. It preserves properties such as reliability and congestion control. Because mTCP uses several paths in parallel, it has to decide how to stripe packets across the paths and how to manage congestion control for each subflow.

3.1.1 Congestion Control

In mTCP, all subflows share the same send/receive buffer. Packets are assigned sequence numbers in the same way as in TCP. But it does congestion control independently on each subflow. Each subflow maintains a congestion window as in TCP. The congestion window changes independently as the subflow adapts to the network state. When there are no packet losses in the subflow, it linearly increases. Upon detecting packet losses, it is halved. When timeout occurs, it is reset to one and the subflow enters slow-start.

mTCP strives to keep all subflows independent from each other. Suppose we had used only one global congestion window for the entire flow that limits the total amount of outstanding data across all subflows. The packet losses on any one of the paths will cause the global congestion window to be halved. If one subflow happens to traverse a heavily congested path, it can keep the global congestion window small, and the other subflows will not be able to utilize the available bandwidth on other good paths. In fact, this can sometimes decrease the throughput of the whole flow to be even lower than that of a single-path TCP flow on a single good path. This phenomenon was also studied in [16].

3.1.2 Estimating Outstanding Packets

TCP uses *(sndnxt - snduna)* to estimate the number of outstanding packets in the network. (For convenience, we assume packets are of the same size and use packets instead of bytes for discussion.) Here *sndnxt* is the next packet to be sent and *snduna* is the next packet for which an ACK is expected. The difference should be no more than the congestion window (*cwnd*). In mTCP, since packets are striped across different paths, we need to keep track of how many outstanding packets are in each path to ensure that the number does not exceed the *cwnd* of that path.

Our mTCP is based on TCP SACK [22], which is an extension of TCP Reno. In Reno, the receiver only reports the greatest packet number that arrives in-order. But in mTCP, different paths have different latencies. Many packets can arrive at the receiver out-of-order. We want to accurately know which packets have been received, irrespective of whether or not they arrived in-order. This helps us to compute the number of outstanding packets on each path, which is essential for performing congestion control separately on each subflow. In SACK, the sender maintains a scoreboard data structure to keep track of which packets have or have not been received. An acknowledgment (ACK) packet may carry several SACK blocks, where each SACK block reports a non-contiguous set of packets that has been received. The first SACK block reports the most recently received packet and additional SACK blocks repeat the most recently reported SACK blocks. The SACK blocks allows the sender to identify what packets have been newly received irrespective of whether or not the data packets arrive in-order.

We augment the scoreboard data structure so that it records the path over which each packet is transmitted or retransmitted. For each $path_i$, we maintain a $pipe_i$ to represent the number of outstanding packets on $path_i$. $pipe_i$ is incremented by 1 when the sender either sends or retransmits a packet over $path_i$. It is decremented when an incoming ACK indicates that a packet previously sent on $path_i$ has been received. New packets are allowed to be sent over $path_i$ only when $pipe_i < cwnd_i$. Retransmitted packets require special handling. Suppose the original packet is sent over $path_i$ and the retransmitted packet is sent over $path_j$. When the retransmitted packet is ACKed, the sender decrements both $pipe_i$ and $pipe_j$ by 1, because it represents two packets having left the network: the original one on $path_i$, which is assumed to be lost, and the retransmitted one on $path_j$, which has been received. We want to emphasize that the original and retransmitted packets do not have to be sent over the same path. We will discuss this in more detail in Section 3.1.4. Finally, if $path_i$ times-out, $pipe_i$ will be reset to 0.

3.1.3 Fast Retransmit

Since mTCP sends packets along several paths with different latencies, packets can arrive at the receiver out-of-order. In traditional TCP, duplicate acknowledgments (*dupack*) are associated with packet losses, and *dupacks* typically trigger fast retransmits. In mTCP, since some of the *dupacks* are caused by packet reorderings, the system should be careful in using fast retransmits.

Although packets sent through different paths may be received out-of-order, packets within each subflow will still mostly arrive in-order. Each $path_i$ therefore maintains the following path-specific state: $snduna_i$, the next packet requiring an ACK over the given path, and *$dupack_i$*, the number of *dupacks* received for that path. If an incoming ACK indicates the receipt of a packet sent through $path_i$ and if that packet is $snduna_i$, this packet is considered to be in-order within that subflow. If that packet is greater than $snduna_i$, $dupack_i$ is incremented by 1. When $dupack_i$ reaches $dupthresh = 3$, $path_i$ will enter fast retransmit and fast recovery.

3.1.4 Sending Packets

mTCP separates the decisions of when to send a packet, which packet to send, and which path to use to send the packet. The sender is allowed to send a new packet when there exists at least one $path_i$ satisfying $pipe_i < cwnd_i$. The packet to send is usually determined by *sndnxt*, which represents the next packet to send as in TCP. But if there is a $path_i$ with packets to retransmit, *i.e.* $path_i$ is in fast recovery, the sender has to retransmit those packets inferred to be lost before sending any new data packets. Once again the scoreboard is consulted to determine whether there are any such packets that need to be retransmitted. Otherwise, a new data packet determined by *sndnxt* will be sent.

Next, the sender needs to decide the path over which the packet will be sent. There may be several candidate paths. We associate a score of $pipe_i/cwnd_i$ with each $path_i$. We choose the path with the minimum score. This form of proportional scheduling results in a fair striping of packets and avoids sending a burst of packets on one path.

Because mTCP separates the decisions about when to send, which to send and which path to use for sending, it has more flexibility in striping packets. By postponing the decision about which path to use until just before sending out the packet, it can quickly adapt to dynamic variations in path characteristics. If a path encounters congestion or fails, its *cwnd* will be reduced. mTCP does not have to wait for the re-opening of the *cwnd* on that path to retransmit the outstanding packets. It can instead retransmit the packets on other paths. We want to emphasize that, unlike the re-striping scheme used in pTCP [16], our scheme will not retransmit packets that have already been received, because we can precisely infer missing packets from the scoreboard data structure.

3.1.5 Single Reverse Path

In our design, despite the fact that data packets are striped over several paths, all ACKs return over the same path. There are two advantages of using a single path for conveying ACKs. First, it preserves the ACK ordering for all the subflows. If ACKs are conveyed through different paths, this may introduce ACK reorderings even within one subflow, which will increase the burstiness of the sender. Second, striping ACKs across different paths makes our system more complicated. The receiver has to maintain additional state about which ACKs are sent through which paths. We instead keep the receiver side as simple as possible, following the design principle of TCP.

The advantage of using multiple reverse paths is that it is more robust to congestion or failure. Note however that ACKs are typically small and are less likely to cause congestion than data packets. In addition to this, we try to avoid congestion by selecting the best path among all the candidate paths as the reverse path for sending ACKs. This will be discussed in Section 3.3. Recovering from failures of the single reverse path is a little more complicated. If we stripe ACKs across multiple paths, although the scheme is more robust to failures, we have to constantly suffer from ACK reorderings. Since failures generally will not occur very often, by using a single reverse path, we trade-off a performance degradation during failure recovery for better performance during normal operation. In Section 3.5.2 of our paper, we show that the failure recovery time of mTCP is reasonably small even though it uses a single reverse path.

3.1.6 Comparison with Multiple TCP sockets

We could have avoided much of mTCP's complexity by opening separate TCP sockets for each path and then striping packets at the application layer [28]. But this approach has limitations that can prevent us from efficiently aggregating the bandwidth of multiple paths. To fully utilize the bandwidth of multiple paths, we must ensure that the amount of data striped over a path is proportional to the bandwidth of that path. This can be very difficult to achieve in Internet-like settings where the path conditions could vary. Modifying TCP directly gives us more flexibility on striping data streams over multiple paths. We can decide, for each packet, an appropriate path the packet should traverse, and this decision is made just before the packet is sent out. This is especially useful for migrating the outstanding packets on bad paths to good paths when the quality of paths changes dynamically or during path failures. Striping at the application layer cannot adapt to changes in path quality responsively. The pTCP study [16] has shown in simulations that such a scheme cannot fully utilize multiple paths when the number of paths exceeds two.

3.2 Shared Congestion Detection

When mTCP uses paths that are not completely disjoint and if some of the shared physical links are congested, the whole mTCP flow will obtain more bandwidth than other single-path TCP flows along those congested links, since each of the subflows behaves as a TCP flow. mTCP tries to alleviate the aggressiveness problem by detecting shared congestion among its subflows and suppressing some of them. Previous work [26, 15, 18, 33] on shared congestion detection is based on the observation that if two single-path flows share congestion, packets from two flows traversing a congested link at about the same time are likely to be either dropped or delayed.

Rubenstein *et al.* [26] actively inject probing packets through the two paths to compute the correlation of packet losses or packet delays and thereby identify shared congestions.

We could certainly use one of the approaches mentioned above in our system since shared congestion detection is independent from other parts of the system. We however take a simpler approach based on the following observations. mTCP transmits a steady stream of packets through different paths. In this setting, there is no need to send probing packets. Instead, one can passively monitor the subflows by studying the behavior of the data packets.Furthermore, since individual packet drops will result in fast retransmits along the corresponding subflows, the sender can detect shared congestions by examining the correlations between the fast retransmit times of the subflows. Since data packets also double as probe packets and since there are a large number of data packets transmitted through a subflow, our passive monitoring strategy requires little overhead and generates a continuous stream of information resulting in fast detection of shared congestion.

3.2.1 Detecting Shared Congestion using Fast Retransmits

Let us focus on detecting shared congestion between a pair of subflows. For more than two subflows, we need to detect shared congestion between every pair of them. In the following discussion, we say that two subflows are *correlated* if the corresponding paths share congestion; otherwise, they are *independent*. We first assume that two paths have the same latency so that we do not have to worry about the time synchronization problem between them. Later, we will extend our algorithm so that it can deal with paths with different latencies.

Each time that a subflow enters fast retransmit, the sender records a timestamp in the subflow's list of fast retransmit events. After some time, we have two lists of timestamps, S and T, from two flows: $(s_1, s_2, ..., s_m)$ and $(t_1, t_2, ..., t_n)$. Each timestamp represents a fast retransmit event. Then we try to match a timestamp s_i in S with t_j in T. If $|s_i - t_j| < interval$, we call (s_i, t_j) a match. Intuitively, a match means the two subflows enter fast retransmit around the same time. This also means packets from the two flows are dropped at about the same time, so it is likely they share the same congested link. We define $match(S,T)$ to be the maximum number of pairs (s_i, t_j), such that s_i matches t_j. Please note that each s_i cannot be matched with multiple t_j. Finally, two subflows are considered to be correlated if:

$$ratio = \frac{Match(S,T)}{min(m,n)} > \delta$$

ratio is intended to identify what fraction of fast retransmits occur at about the same time in the two subflows. Since some of the fast retransmits are due to congestion on disjoint links, *ratio* reflects the level of shared congestion. We consider two subflows to be correlated when *ratio* is greater than some threshold δ.

Our method uses fast retransmits instead of individual packet losses to infer shared congestion. This is because when a data flow encounters congestion, there normally will be a burst of packet losses. All these losses are caused by one congestion period at some link. Therefore, a fast retransmit corresponds more directly to a congestion period, much more so than any individual packet loss. We would like to declare (s_i, t_j) to be a match only when packets from two subflows are dropped at one link during the same congestion period. So $interval$ cannot be too small, otherwise even if s_i and t_j occur in the same congestion period, the system will not detect the match. On the other hand, $interval$ cannot be too large, otherwise the system would consider (s_i, t_j) to be a match even when they are not due to shared congestion. Although the shared congestion detection may not work well under active queue management schemes, most routers on today's Internet use drop-tail queues, which lead to periods of bursty losses during congestion. In [35], the authors find that 95% of the duration of bursty losses are less than 220ms. So $interval$ should be on that time scale. We will study how to choose $interval$ and δ in more detail in Section 5.4.

3.2.2 Estimating Convergence Time

We need to emphasize that our goal is to suppress correlated subflows in order to alleviate the aggressiveness problem. We need to detect shared congestion as quickly as possible. Other efforts focus more on the accuracy of shared congestion detection, and they may take several hundred seconds to reach a decision. This does not work well for our purpose, because a mTCP flow could have ended before shared congestion is detected.

Our algorithm works as follows. After some number of fast retransmit events have been observed, we will check for shared congestion between the two subflows. If there is shared congestion, we can suppress one of them. Otherwise, we will wait until the occurrence of the next fast retransmit to check for shared congestion again. The question we now address is determining the number of fast retransmit events that we need to observe before we start checking for shared congestion.

We use a heuristic to estimate the probability of two fast retransmit events from two independent flows accidentally occurring within a small period of time. Suppose the fast retransmit events of two subflows, S and T, are completely independent when two subflows are independent, we compute the average interval of two consecutive fast retransmit events in S: $interval_s = \frac{now}{m}$, where *now* is the current time when shared congestion detection is invoked. $interval_t$ is computed in a similar way. Then we define $p = \frac{2 \times interval}{min(interval_s, interval_t)}$. Suppose $n \geq m$, we have $interval_t \leq interval_s$, and $interval = \frac{p}{2} \times interval_t$. For each s_i, if there exists a match t_j, s_i

must be in the $(t_j - interval, t_j + interval)$. Because we assume s_i and t_j are independent events, the probability that s_i matches some t_j is roughly p. So the total expected number of matches is roughly $E(Match(S,T)) = pm$. Because $min(m,n) = m$, we will misinterpret S and T to share congestion if $Match(S,T) > \delta m$. According to Chernoff bound [11]:

$$\zeta = Prob[(Match(S,T) > \delta m] < e^{-mD(\delta||p)},$$

where $D(\delta||p) = \delta \ln \frac{\delta}{p} + (1-\delta)\ln\frac{1-\delta}{1-p}$. So we need to wait for $m = -\frac{\ln\zeta}{D(\delta||p)}$ fast retransmit events to ensure that the probability of a false positive is less than ζ. We will see in Section 5.4 the convergence time is mostly within 15 seconds in our Emulab and PlanetLab experiments. We want to emphasize that even if a false positive does occur, it will only degrade a mTCP flow into a single-path flow.

This heuristic might encounter problems when $min(interval_s, interval_t) \leq 2 \times interval$. Because $interval$ is small (200ms in our experiments), this can only occur when a path is so heavily congested that fast retransmit happens almost every 400ms. The mTCP flows will try to suppress such paths, because using them will not bring much benefit. This is discussed in Section 3.4.1.

Finally, when two paths have different latencies, there is a time-lag, L, between them. We estimate L by shifting one sequence, say T, by dt in time and calculating $Match_{dt}(S,T)$ on sequences (s_1, s_2, ..., s_m) and ($t_1 + dt$, $t_2 + dt$, ..., $t_n + dt$) as described before. Because the L between two paths can be at most one RTT (where RTT is the larger round trip time of the two paths), we go through all possible value dt in $(-RTT, RTT)$ incrementally using some fundamental step x, then choose dt that maximizes $Match_{dt}(S,T)$ as L. This is similar to calculating the correlation between two signals.

3.3 Path Selection

In the previous sections, we assumed that flows have a number of candidate paths. Now we describe how to obtain such information. We use Resilient Overlay Networks (RON) [8] as our underlying routing layer. RON is an application-layer overlay.When mTCP starts, it queries RON to obtain multiple paths between a source-destination pair. For each pair, RON provides the direct Internet path and alternate single-hop indirect paths through other RON nodes. With a RON of n nodes, there are $m = n - 1$ paths between each pair. RON uses a score to represent the quality of each path based on latency, loss rate or throughput. RON can effectively bypass performance failure or path faults by using an alternate path with higher score. In the following, we only use the throughput score.

Since m can be large (greater than 10 in our experiments), mTCP will only select at most k (5 in our experiments) paths from them. A single-path flow will normally select the path with the best score, which we call the *RONpath*. mTCP could select the k best paths. But this simple strategy may select paths with many overlapping physical links. This leads to two disadvantages: First, paths are more likely to fail simultaneously, which is bad for the robustness. Second, paths are more likely to share congestion, which is bad for performance. To avoid these problems, we want to select sufficiently disjoint paths.

We use a heuristic based on traceroute to estimate the disjointness of paths. Using traceroute, we can obtain the IPs of the routers along a path and the latency of each physical link. Due to IP aliases, the same router might have different IPs in different paths. We use "Ally", a tool from Rocketfuel [30], to resolve IP aliases and assign a unique IP to each router. Although some routers may not respond to traceroute probes and the alias resolution may not be completely accurate, we only use the traceroute information as a hint to estimate path disjointness and eliminate many of the significantly overlapping paths. We also rely on the techniques described in Section 3.2 to further detect shared congestion.

After alias resolution, suppose we have the IPs of two paths $X = (x_0, x_1, ..., x_m)$ and $Y = (y_0, y_1, ..., y_n)$. Let L be the set of overlapping links of X and Y, we define the overlapping between X and Y as: $Overlapping(X,Y) = \sum_{l \in L} latency(l)$. An alternative is to use the size of L to quantify the degree of overlap. We use latency instead because we hope to distinguish among different types of link. Most nodes on PlanetLab are connected through ethernet links to backbones. Those ethernet links usually have smaller latency than backbone links. Because the sharing of the local ethernet links are almost unavoidable, we focus on finding disjoint paths that traverse different backbone links. By using link latencies, $Overlapping(X,Y)$ will be mostly determined by the shared backbone links instead of ethernet links. This argument might not be true if nodes are connected through modem or wireless links that have high latency. Using traceroute to find disjoint paths is only suitable for small-scale overlay networks. As the number of nodes increases, we need a more scalable way to discover disjoint paths. In [23], Nakao, Peterson and Bavier propose to use BGP information to find disjoint Autonomous System (AS) paths, which incur little cost. Although disjoint AS paths are not as fine-grained as disjoint router-level paths, it would greatly simplify disjoint path search by providing a small set of promising candidate paths which we can further verify using traceroute.

Finally, we estimate the disjointness of X and Y by:

$$Disjoint(X,Y) = 1 - \frac{Overlapping(X,Y)}{Min(Latency(X), Latency(Y))}$$

We say that X and Y are disjoint if $Disjoint(X,Y) > \beta$. Using the disjointness metric between each pair of paths, we select at most k paths from m paths using a greedy algorithm as follows: (1) Initialize the set of selected paths to be empty. (2) Pick the path with the highest score from the set of m paths and check if it is disjoint from all the previously selected paths. (3) If so, select this path, otherwise pick the path

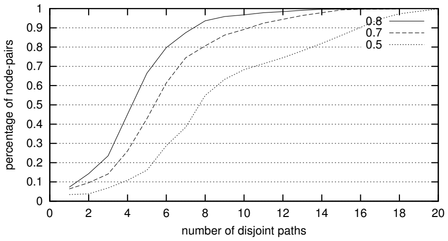

Figure 1: CDF of number of disjoint paths between node-pairs

with the next highest score and repeat step (2) until we find k paths or we have tried all m paths. The first selected forward path and the reverse path will always be the *RON path*, which is optimized for throughput in RON.

Figure 1 plots the cumulative distribution function (CDF) of the number of disjoint paths between 630 node pairs based on traceroute among 36 PlanetLab nodes that are used in our experiments. When β decreases, the number of disjoint paths between node-pairs increases. We want a β such that there are sufficient number of disjoint paths which we can choose from while eliminating most significantly overlapping paths. When $\beta = 0.5$ (the value used in our experiments), 90% of node-pairs have more than 4 disjoint paths but less than 16 disjoint paths. If we use a larger β, many node pairs will not have enough candidate disjoint paths.

3.4 Path Management

3.4.1 Path Suppression

In mTCP, a subflow f_i on $path_i$ may be suppressed because of one of three reasons: First, f_i shares congestion with another subflow f_j and its throughput $T(f_i)$ is lower than $T(f_j)$. This is because we do not want mTCP to be too aggressive to other single-path TCP flows. Second, suppose f_j has the highest throughput among all the subflows and if $T(f_i) < \frac{T(f_j)}{\omega}$ (for some constant $\omega > 1$), then f_i may be suppressed. This is because $path_i$ is too poor and using it does not bring much benefit. Third, $path_i$ fails.

We define a family of mTCP flows, called MP_d flows. An MP_d flow will try to use at least d ($d \geq 1$) paths, which means we will not suppress any path because of shared congestion when the number of paths being used is less than or equal to d. The value of d is a tradeoff between performance/robustness and friendliness. With a larger d, mTCP can obtain more bandwidth because it uses more paths. And it is more reliable because the probability that d paths fail simultaneously gets smaller as d increases. But it can be more aggressive to single-path flows under shared congestion. The aggressiveness problem can be alleviated by suppressing some subflows. But when there are only d subflows, no subflow would be suppressed. The actual value of d should be decided by the application. Applications that want higher performance and more reliability should choose a larger d. Applications that care more about friendliness should choose a smaller d. In our experiments, we choose $d = 1$ to demonstrate how much performance improvement mTCP can obtain without being too aggressive to other TCP flows.

3.4.2 Path Addition

An MP_d flow can dynamically add new paths because of two reasons: First, some paths that are not being used become better than those paths that are being used. Second, it is using less than d paths because some paths were suppressed. mTCP will periodically update the information about all the paths by querying RON. If an unused path has a much higher score than a path being used, it can start using the new path. Then it runs the path suppression algorithm on all the paths to suppress any paths with shared congestion. By doing this, mTCP can gradually replace bad paths with good ones. This is especially useful for long-lived flows.

3.5 Path Failure Detection and Recovery

3.5.1 Failure Detection

mTCP may encounter path failures during transmission. If all the paths fail simultaneously, we call it a *fatal path failure*, otherwise we call it a *partial path failure*. We will focus on partial failures in this section. To recover from fatal failures, mTCP relies on the routing layer to establish new paths just like single-path flows.

When a path fails, the data packets sent over it will no longer be acknowledged (ACKed) because the packets have been dropped. We maintain one failure detection timer, $timer_i$, for each $path_i$. When a data packet sent over $path_i$ is ACKed, $timer_i$ is reset. $path_i$ is considered to have failed when $timer_i$ expires.

We need to decide a timeout value I_i for $timer_i$. On one hand, we want a small I_i so that failures can be detected quickly. On the other hand, I_i cannot be too small, otherwise it may misinterpret a good path to have failed. The retransmission timeout (RTO_i) provides a good base for computing I_i. First, during timeout, the sender will go into idling and no packets will be ACKed in that period. So I_i should be at least greater than RTO_i. Second, several consecutive timeouts means either the path has failed or it is heavily congested. In either case, we would like to abandon $path_i$. So we choose $I_i = \chi RTO_i$. Here χ reflects how many consecutive retransmission timeouts mTCP is willing to tolerate before it consider a path to have failed. In our experiments, we choose $\chi = 2$, because we have observed that consecutive retransmission timeouts rarely occur on good paths. We should emphasize that even if a good path is misinterpreted as a failed one, it will only degrade the performance of mTCP to that of a single-path flow in the worst case. The path addition technique described in Section 3.4.2 allows us to reclaim a path if it had been previously misinterpreted to be a failed path.

3.5.2 Failure Recovery

We now describe how to recover from failure after $timer_i$ expires. Since all ACKs return over the same path, we call that path a *primary* path. The other paths are *auxiliary* paths. We need to distinguish between primary and auxiliary path failures. When an auxiliary $path_i$ fails, the sender will mark $path_i$ as failed and retransmit the outstanding packets of $path_i$ over other paths. When a primary path fails, the situation is more complicated. Because all the ACKs are lost, it may appear to the sender that all paths have failed. To deal with this problem, sender records the time π_i when $path_i$ is detected to have failed. Suppose at time *now*, the primary $path_p$ is also detected to have failed and let the timeout of $timer_p$ be I_p. We know that $path_p$ must have failed at some point between $now - I_p$ and *now*. For an auxiliary $path_i$, if $now - I_p \leq \pi_i$, its failure is possibly due to the failure of $path_p$. In this case, we will change the status of $path_i$ to be active and the status of $path_p$ to be failed. After doing this for all the paths, the sender starts to send packets over all active paths. During this period, these data packets serve as "probing" packets that solicit ACKs from the receiver. All timers are stopped to prevent any auxiliary path from being misinterpreted as failed due to the lack of an active primary path during this period. The receiver will also detect the primary path failure because it no longer receives any data packets over that path. Then it elects a new primary path and sends ACKs along that path in response to those "probing" packets from sender. It chooses the best path (based on the path score in RON) among all the active paths to be the new primary path. Later, when the sender receives the ACKs and knows that a new primary path has been elected, it restarts all the timers and proceeds as normal.

Typically, RTO_i is one second, therefore I_i is two seconds. The total detection and recovery time will be between two and three seconds in most cases. The interruption due to partial path failures will be fairly short. Furthermore, partial path failure does not cause mTCP to stall since packets will continue to be transmitted through active paths. Since mTCP uses several paths concurrently and since it typically employs disjoint paths, the probability of fatal path failures is much lower than that of single-path failure. mTCP is therefore more robust than single-path flows.

4 Implementation

Our system is implemented at the user-level and is composed of a Portable User-Level TCP/IP stack (PULTI) and an overlay router/forwarder modified from RON. RON is an application-layer overlay on top of the Internet. PULTI and RON run in two separate processes. We modified RON so that it can communicate with PULTI using UDP sockets and export the multiple paths between a source-destination pair. The whole system does not require any root privilege, and hence it can be easily deployed on shared distributed platforms such as PlanetLab. Currently, it runs on Linux, NetBSD and FreeBSD. Note that although our system in build on top of overlay networks, it will work in a non-overlay setting as well. The only requirement of our system is that there exists an underlying routing system that can provide multiple paths between a source-destination pair and let us select paths for sending packets. RON is just one of such systems. Our system will also work in other Internet settings, such as multihomed hosts.

PULTI is a full user-level TCP/IP stack based on FreeBSD 4.6.2. We extract the network-related code from the kernel source and wrap it with some basic kernel environment support, such as timing, synchronization and memory allocation. We do not modify any network-related code. Because FreeBSD 4.6 does not support SACK, we also add SACK-related code in PULTI which is required by our system. OS dependent information is hidden by device drivers. With different device drivers, PULTI can send or receive through a UDP socket, IP_QUEUE in Linux or a divert socket in FreeBSD. PULTI provides standard socket interface and supports multiple applications through multithreading. It can query RON to learn about multiple paths between a source-destination pair. The mTCP code only affects a few files in PULTI. It can be easily moved into FreeBSD kernel.

5 Evaluation

5.1 Methodology

In this section, we validate our protocol in both emulation and real-world deployment. The emulations are run on Emulab [2], which is a time- and space-shared network emulator. Emulab consists of several hundred PCs, which can be configured to emulate different network scenarios. Users can specify parameters such as packet loss rate, latency, and bandwidth. While an experiment is running, the experiment gets exclusive use of the assigned machines. While Emulab provides a controlled environment for our experiments, we further conduct experiments on PlanetLab, a wide-area distributed testbed for running large-scale network services [25]. The experiments on the PlanetLab allow us to study our protocol for Internet settings, where latency, bandwidth and background traffic are more realistic and unpredictable.

5.2 Utilizing Multiple Independent Paths

In this experiment, we study whether mTCP can obtain the total available bandwidth over multiple independent paths. We use the topology in Figure 2 on Emulab. Because each PC in Emulab has four Ethernet cards, each node can have at most four links. There are six endhosts (H_i) and ten routers (R_j). RON is running on the six endhosts to construct an overlay network. All routers have drop-tail queues. The source and destination nodes are H_0 and H_5 respectively.

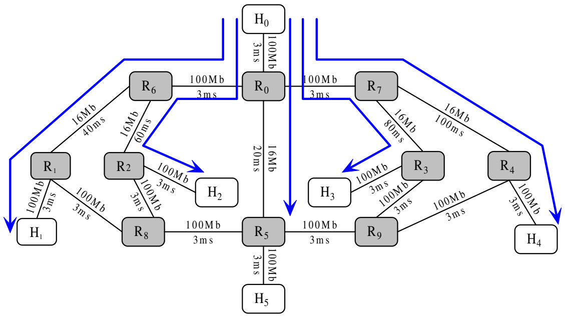

Figure 2: Topology of multiple independent paths on Emulab

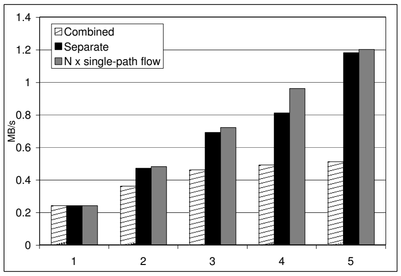

Figure 3: Throughput of mTCP flows with combined or separate congestion control as number of paths increases from 1 to 5

Each of the remaining endhosts provides an alternate path. For example, we can use H_1 to construct an alternate path $(H_0, R_0, R_6, R_1, H_1, R_1, R_8, R_5, H_5)$. So the topology contains five independent paths, which include one direct path and four alternate paths. We use the direct path as the reverse path for ACKs. The capacity of all the paths is 16Mbps and their RTTs vary from 52-147ms. The figure annotates each link with its corresponding bandwidth and latency. The arrows represent background flows. We use Iperf [1] to generate 25 TCP and 25 1Mbps UDP flows as background traffic, with 5 TCP and 5 UDP flows on each path. Each experiment runs for 40 seconds and the results are obtained by averaging three runs.

Figure 3 shows the results when the number of paths used by mTCP increases from 1 to 5. In this figure, "combined" represents mTCP flows with congestion control performed on the entire flow, "separate" represents regular mTCP flows with congestion control performed separately on each subflow, and "NxSingle-path flow" is the throughput of a single-path flow on one path multiplied by the number of paths. Because each path has the same available bandwidth, "NxSingle-path" throughput represents the ideal throughput of a mTCP flow. The results verify that mTCP can effectively aggregate the available bandwidth on multiple independent paths. The results also show that higher throughput can be achieved only when congestion control is performed for each subflow separately.

Path	Intermediate node	RTT(ms)
0	direct path	80.165
1	planetlab1.nbgisp.com	112.503
2	planet2.berkeley.intel-research.net	71.639
3	planet2.pittsburgh.intel-research.net	96.641
4	planet2.seattle.intel-research.net	90.305

Table 1: Independent paths between Princeton and Berkeley nodes on PlanetLab.

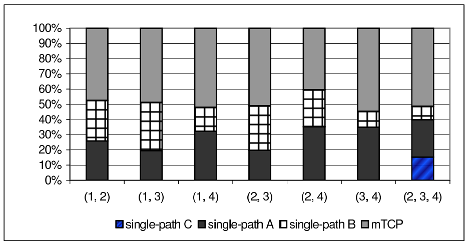

Figure 4: Throughput percentage of individual flows

We conduct similar experiments on PlanetLab. We use one node in Princeton and one node in Berkeley as source and destination nodes. As shown in Table 1, the four Intel nodes serve as intermediate nodes for the alternate paths. We only use the four alternate paths in this experiment, because they do not share any congestion links. To verify this, we examined the traceroute data to find that any pair of the alternate paths only share the initial and final hops, which are unavoidable. The capacity of these two links are 100Mbps, which is far greater than the total throughput of the single-path TCP flows on these four paths. Therefore, we conclude that the initial and final hops are not congested and the four alternate paths are independent.

Each experiment measures the throughput of flows lasting for 60 seconds. The average throughput of three runs is reported. For convenience, we use $T(i)$ to denote the throughput of a single-path flow on $path_i$. Similarly, $T(i, j)$ denotes the throughput of a mTCP flow using $path_i$ and $path_j$. In Figure 4, (i,j) on the x-axis means $path_i$ and $path_j$ are used in that experiment. We first run single-path flows on $path_i$ and $path_j$ respectively, then run a mTCP flow on both paths simultaneously. The corresponding column compares the percentage that the throughput of an individual flow, $T(i)$, $T(j)$ or $T(i, j)$, contributes to the total throughput of these flows. Ideally, we expect $T(i, j) = T(i) + T(j)$, so the percentage of $T(i, j)$ should be around 50%. With the exception of the experiment involving $path_2$ and $path_4$, which suffered from unexpected bandwidth variations, the rest of the experiments indeed provide the expected throughputs. The last column in Figure 4 shows the result of the experiment using $path_2$, $path_3$, and $path_4$. Again, the net throughput of $T(2, 3, 4)$ is close to the sum of $T(2)$, $T(3)$ and $T(4)$. We have conducted experiments between different source-destination pairs on PlanetLab. The results are similar. We omit them due to space constraints.

Path	Intermediate node	RTT(ms)
0	direct path	80.165
1	planetlab02.cs.washington.edu	102.890

Table 2: Paths used in the failure recovery experiment.

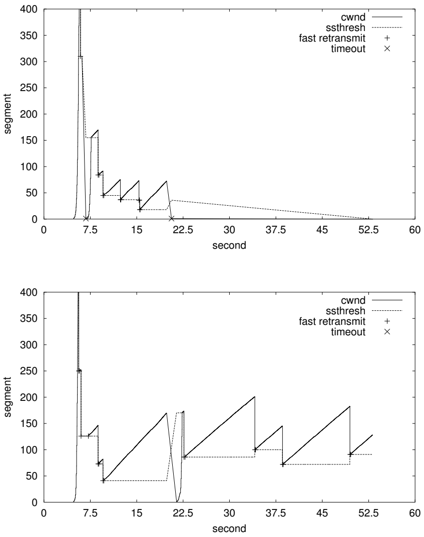

Figure 5: *cwnd* of primary/auxiliary paths, primary fails

5.3 Recovering from Partial Path Failures

Now we will study whether mTCP can quickly recover from partial path failures using experiments on PlanetLab. Because path failures on the Internet are unpredictable, we intentionally introduce failures by killing the appropriate RON agent. The source and destination nodes are still the Princeton and Berkeley nodes. The paths are shown in Table 2.

The two graphs in Figure 5 show how the congestion window (*cwnd*) of the primary and auxiliary paths changes over time. As shown in the first graph in Figure 5, the primary path fails at about 20s. It is quickly detected so that the *cwnd* of the subflow on this path is reduced to 0. At the same time, the *cwnd* of the subflow on the auxiliary path also decreases to 0, because the auxiliary path was misinterpreted to have failed (as explained in Section 3.5.2). But a few seconds later, the subflow on the auxiliary path recovers from this false decision by restoring its *cwnd* to the previous value with slow start. Finally the auxiliary path becomes the new primary path and the whole flow proceeds using only one path. The behavior of mTCP during auxiliary path failures is similar, and we omit the corresponding results.

The total recovery time of mTCP during partial path failures is only about 3s, which is negligible for most applications. In contrast, a TCP flow will completely stall when its path fails, and it typically takes about 18s for RON to establish a new path. RON is optimized for quickly recovering from path failures. On wide area network that uses BGP to detect failures, recovery could take several minutes. Hence, mTCP is more responsive and robust than single-path flows.

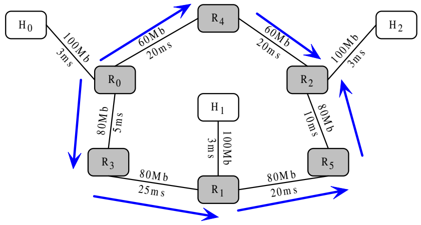

Figure 6: Two independent paths used in shared congestion detection

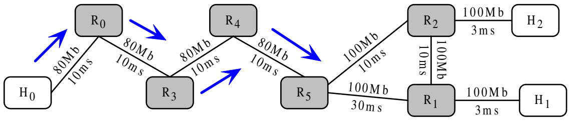

Figure 7: Two paths that completely share congestion

5.4 Detecting Shared Congestion

In this section, we will evaluate shared congestion detection. We first use experiments on Emulab to study the behavior of our algorithm with different parameters in a controlled environment. Then we further validate it using experiments on PlanetLab. The topologies for the Emulab experiments are shown in Figures 6 and 7. Between the source node H_0 and the destination node H_2, there is one direct path and one alternate path through the intermediate node H_1.

In Figure 6, The two paths only share the initial and final hops with link capacities of 100Mbps. We generate 12 TCP flows and 18 1Mbps UDP flows as background traffic, with 2 TCP flows and 3 UDP flows on each link between each pair of neighboring routers. With this scheme, we ensure that congestion only occurs on the links between pairs of routers and not on the links between endhosts and routers. As a result, the two paths (H_0, H_2) and (H_0, H_1, H_2) are independent.

In Figure 7, The two paths share the four links between H_0 and R_5. We generate 8 TCP flows and 8 1Mbps UDP flows as background traffic, with 2 TCP and 2 UDP flows on each of the four shared links. By doing this, we ensure that congestion only occurs on the four shared links. As a result, paths (H_0, H_2) and (H_0, H_1, H_2) share congested links.

We run mTCP flows for 300s using the two paths in Figure 7. The results in Figure 8 compare the estimated *ratio* of shared congestion with different *interval* values of 5ms, 10ms, 25ms, 50ms, 100ms, 200ms, and 400ms. Each data point represents the average of five runs. As *interval* increases from 5 to 100ms, *ratio* increases quickly from 0.4 to 0.8 as expected. When *interval* increases beyond 100ms, *ratio* only increases slightly. When *interval* is 400ms, *ratio* reaches 0.96. The ideal *ratio* is 1 because the two paths share all the congestion.

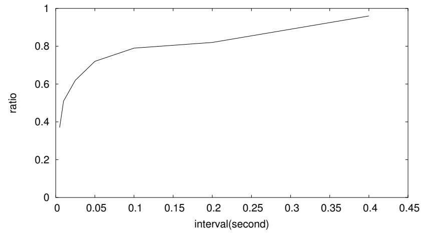

Figure 8: On two paths with shared congestion, *ratio* increases as *interval* increases

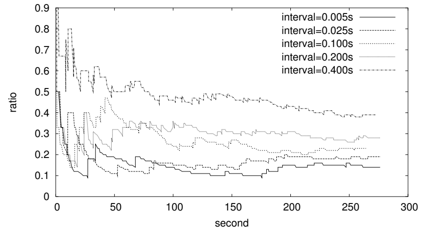

Figure 9: On two independent paths, *ratio* decreases faster when *interval* is smaller

We next run mTCP flows for 300s on the two paths in Figure 6. The results are shown in Figure 9, which plots *ratio* over time for different *interval* values of 5ms, 25ms, 100ms, 200ms and 400ms. As explained in Section 3.2, a smaller *interval* will lead to smaller estimated values of *ratio*. At the end of the experiments, *ratio* drops quickly from 0.39 to 0.28 when *interval* decreases from 400 to 200ms. When *interval* deceases further, *ratio* drops more slowly until it reaches 0.14 when $interval = 5ms$. The ideal *ratio* is 0 because the 2 paths are independent. We also notice that the *ratio* curve for a smaller *interval* value decreases faster than that for a larger *interval*.

According to the above experiment results, an *interval* value between 100 and 200ms seems to balance the goal of minimizing both false negatives and false positives. Consequently, the *ratio* threshold δ should fall between 0.3 and 0.8. If it is less than 0.3, it is very likely to cause false positives when the *interval* is 200ms. If it is greater than 0.8, it can easily cause false negatives when the *interval* is 100ms. By setting $interval = 200ms$ and $\delta = 0.5$, we successfully detect shared congestion between the two paths in all five runs for the topology in Figure 7. For the topology in Figure 6, no shared congestion is detected and the two paths are determined to be independent as expected.

Next, we go on to evaluate the shared congestion detection on PlanetLab. As explained in Section 3.2, by setting *interval* to be no less than the congestion period during which bursty losses occur, we can avoid false negatives. In [35], the authors find that 95% of the duration of bursty losses on the Internet are very short-lived (less than 220ms). By choosing an *interval* around that value, we should be able to avoid most false negatives. At the same time, the average time between consecutive fast retransmits is mostly on the order of several seconds or more, much greater than 220ms. (Otherwise, mTCP will suppress such path because the path is too lossy.) Therefore, this *interval* value will also allow us to avoid most false positives, as long as we wait for enough number of fast retransmits. In the following experiments, we report the results using $interval = 200ms$ and $\delta = 0.5$.

Path	Run 1	Run 2	Run 3
1 2	No	No	No
1 3	No	No	No
1 4	No	No	No
2 3	No	No	No
2 4	N/A	Yes	No
3 4	No	No	No

Table 3: Shared congestion detection for independent paths.

Path	Intermediate node	RTT(ms)
0	direct path	80.165
1	planetlab2.cs.duke.edu	96.138
2	planetlab2.cs.cornell.edu	100.382
3	vn2.cs.wustl.edu	92.267

Table 4: Paths with shared congestion on PlanetLab.

We first need to choose paths such that we can be reasonably sure as to whether they share congestion or not. Then, we can compare the measured results with the expected results. We conduct two sets of experiments. The mTCP flow is running on a pair of paths for 60 seconds in each experiment. As explained in Section 3.2, the probability of false positive decreases very fast as the number of fast retransmit increases. We find that a 60 second period is long enough for our algorithm to converge. Each experiment is repeated three times. We use the Princeton and Berkeley nodes as source and destination in all experiments, but we choose different pairs of paths in different sets of experiments.

In the first set of experiments, we use the four alternate paths in Table 1, where we know that all these paths are independent. The results are in Table 3. The first column shows the pairs of paths used by the mTCP flows. The remaining three columns show the results. A *No* means two paths are independent, a *Yes* means they share congestion, and *N/A* means one of the subflows is suppressed because its throughput is much lower than the other subflow before the end of the experiment. All the results in Table 1 conform to our expectation except the one false positive for using $path_2$ and $path_4$. As explained before, a false positive will only degrade the performance of the mTCP flow to that of a single-path flow.

The second set of experiments use the paths in Table 4. From traceroute, we know the underlying physical links of any pair of these paths are mostly overlapping, so they should share congested links. The results are shown in Table 5. The first column gives the pairs of paths used in the experiments.

Path	Run 1	Run 2	Run 3	Average
0 1	7.000	9.975	4.266	7.080
0 2	4.276	3.223	6.011	4.503
0 3	6.847	3.263	14.214	8.108
1 2	12.184	8.906	16.804	12.631
1 3	4.478	10.101	13.131	9.237
2 3	12.380	9.873	17.845	13.366

Table 5: Shared congestion detection for correlated flows.

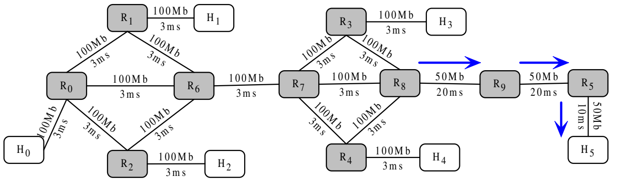

Figure 10: All paths share congestion in this topology

The following three columns give the time in seconds when shared congestion is detected in each run. The last column gives the average detection time. Shared congestion is correctly detected in all cases.

Unlike other shared congestion detection algorithms, our algorithm seeks to minimize the detection time while maintaining a low false positive rate. In the second set of experiments, shared congestion is correctly detected mostly within 15 seconds. At the same time, such early decisions do not cause too many false positives in the first set of experiments.

5.5 Alleviating Aggressiveness with Path Suppression

In this section, we demonstrate mTCP can be more friendly to other single-path flows by suppressing its subflows that share congestion. We construct the topology of Figure 10 on Emulab. The source and destination nodes are H_0 and H_5. There are one direct path and four alternate paths provided by the remaining four endhosts. Their RTTs are from 124ms to 133ms and they share the three links between R_8 and H_5. We generate 12 1Mbps UDP flows as background traffic, with 4 UDP flows on each of the three shared links. By doing this, we ensure that all five paths share congestion. Each experiment runs for 300 seconds and the results are obtained by averaging three runs.

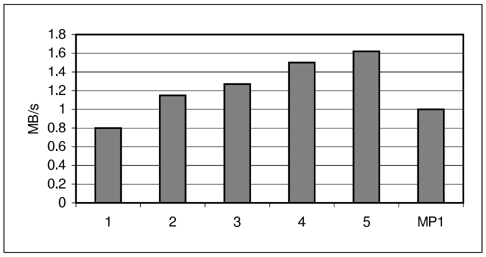

Figure 11: MP_1 flows are less aggressive than other mTCP flows

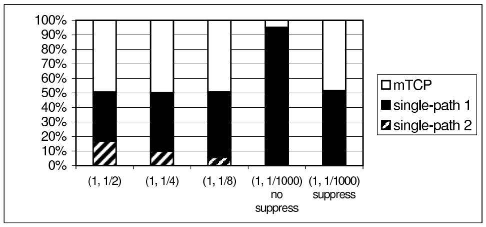

Figure 12: Path suppression helps avoid using bad paths.

In Figure 11, the first five columns give the throughput of the flows when the number of paths being used increases from one to five. The first column is the throughput of single-path TCP flows. Under shared congestion, the mTCP flows become more aggressive as they use more paths. The sixth column shows the throughput of the mTCP flow with path suppression. Although it uses five paths in the beginning, it quickly detects shared congestion and suppresses all but one path. So its throughput is very close to that of a single-path flow and less aggressive than the flow using all five paths without suppression.

5.6 Suppressing Bad Paths

In this experiment, we demonstrate that mTCP can effectively aggregate the bandwidth of multiple paths with sufficiently differing characteristic, and path suppression can help avoid the penalty from using bad paths. We use the same topology as in Figure 2. The bandwidth of direct path is still 16Mbps. But the bandwidth of four alternate paths is 1/2, 1/4, 1/8 and 1/1000 of the bandwidth of the direct path. In Figure 12, $(1, 1/n)$ on the x-axis means the direct path and alternate path with 1/n bandwidth are used in that experiment. We first run a single-path flow on each path respectively, then run a mTCP flow on both paths. The corresponding column compares the percentage that the throughput of an individual flow contributes to the total throughput of these flows. Ideally, the throughput percentage of mTCP flows should be 50%. Figure 12 shows mTCP can efficiently utilize the aggregate bandwidth of two paths even when one path has only 1/8 the bandwidth of the other path. The mTCP flows in the last 2 columns use the direct path and the alternate path with 1/1000 bandwidth. Such a scenario could occur when a path becomes heavily congested or even temporarily fails. Using such bad paths can bring no benefit but impair the performance of the whole flow. Because most packets are lost along that path, it persistently causes timeouts. While packets can still be sent over the other path for some time, the flow will finally stall when the send/receive buffer is exhausted. As explained in Section 3.4.1, mTCP will suppress the paths with too low a throughput to avoid such penalty. (We choose $\omega = 10$ in

Host Name	Host Name
planetlab2.millennium.berkeley.edu	planetlab2.postel.org
planetlab02.cs.washington.edu	planetlab2.lcs.mit.edu
planetlab-2.cs.princeton.edu	planetlab2.cs.ucla.edu
planetlab2.cs.uchicago.edu	planet.cc.gt.atl.ga.us
planetlab2.cs.duke.edu	pl2.cs.utk.edu

Table 6: The 10 endhosts used in the experiments that compare mTCP with single-path flows.

our experiments.) This is confirmed by the last two columns which represent the throughput of mTCP flows with and without suppression.

5.7 Comparing with Single-Path Flows

We are going to compare three types of flows: single-path flows using direct Internet path (INET), single-path flows using *RON path* optimized for bandwidth (RON) and MP_1 flows. MP_1 flows will use multiple paths when there is no shared congestion. We use MP_1 flows to demonstrate how much performance improvement mTCP can obtain without being too aggressive to other TCP flows. Table 6 shows the 10 nodes that serve as endhosts in an overlay network for this experiment. (We actually use a total of 24 nodes to form the overlay network, with the remaining 14 nodes only serving the role of packet forwarders.) For each source-destination pair, we transfer data for 40 seconds using each of the three types of flows. Each experiment is repeated three times and we report the average throughput.

The available bandwidth of the paths between the pairs of endhosts can be very high, because nine of them are connected to Internet2. We bypass those pairs with very high available bandwidth on the corresponding direct paths because: First, these paths are between pairs of nodes that exhibit shared congestion/bottleneck at the initial and/or final hops. Second, the bandwidth-delay products of the paths between such pairs of nodes are very large. The maximum send/receive buffer size of our user-level TCP implementation is 1MB and is not large enough to utilize the bandwidth on other alternate paths besides the direct path. We estimate the available bandwidth of a direct path between a source-destination pair by running a TCP flow for 10 seconds. If the measured throughput is less than 12 Mbps, we will use that pair for our experiments. Among the 90 pairs, we got 15 pairs that satisfy the above condition. We want to emphasize that we are not trying to study the popularity of independent paths with distinct points of congestion between node-pairs on the Internet; such topic has been studied by others [6].Instead, we focus on demonstrating that mTCP can achieve better performance by taking advantage of such redundant paths.

Among the 15 pairs, MP_1 flows achieve significantly higher throughput in 6 pairs, as shown in Figure 13. They achieve 33% to more than a factor of 60 better performance than single-path flows. We have to mention that MP_1 flows only try to aggregate the available bandwidth on multiple paths when there is no shared congestion. Other MP_d ($d \geq 2$) flows would obtain better performance, but they are potentially more aggressive.

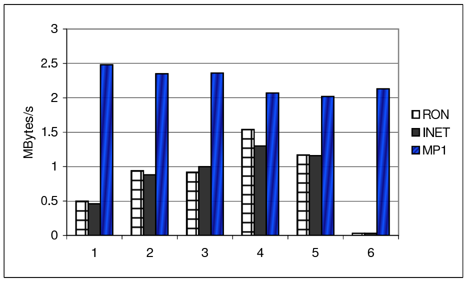

Figure 13: mTCP flows achieve better throughput than single-path flows

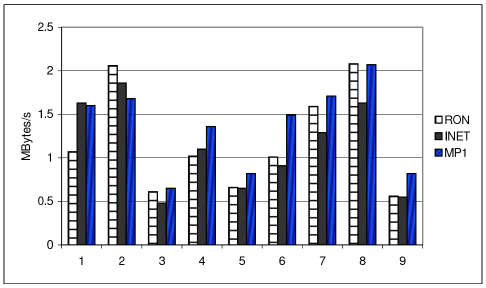

Figure 14: Throughput of mTCP and single-path flows is comparable

The performance improvement of mTCP does not solely come from bandwidth aggregation on multiple paths, it is also because mTCP can help select better paths than those provided by the routing layer, such as the direct path or the *RON path* optimized for throughput. RON estimates the available bandwidth of a path using $score = \frac{\sqrt{1.5}}{rtt\sqrt{p}}$. Here p is the packet loss rate and rtt is the round trip time, both of which are obtained by active probing. Although it can help RON distinguish paths with significant performance difference and select better alternate path, this estimate may not be accurate; a path with high score may actually have low available bandwidth [8]. In mTCP, the sender can monitor the performance of several paths in parallel. The throughput of each subflow provides a fairly good estimate of the available bandwidth on that path. This does not require any active probing because the data packets serve as probing packets. This can help mTCP discover and utilize better paths than the suboptimal *RON path* or direct path. We examined the paths in those 6 pairs and found that MP_1 flows do take paths different from either the direct paths or the *RON paths*. The achieved throughput of on those paths are higher than that of direct path or *RON path*.

Figure 14 shows the results of the remaining 9 pairs. By examining the paths, we find that all MP_1 flows degrade to single-path flows because of shared congestion, and the RON/INET/MP_1 flows all take the same single path for the whole transfer. Hence, the throughput of MP_1 flows should be comparable to that of INET/RON flows, as shown in Figure 14. In three pairs, MP_1 flows obtain slightly lower performance than RON/INET flows, this is because different types of flows are run sequentially and there is minor fluctuations in the available bandwidth of a path over time.

6 Conclusions

In this paper, we present mTCP, a transport layer protocol, for improving end-to-end throughput and robustness. mTCP can efficiently aggregate the available bandwidth on several paths in parallel. To address the aggressiveness of mTCP during shared congestion, we integrate a shared congestion detection mechanism into our system so that correlated subflows can be suppressed. mTCP flows are more robust to path failures than TCP flows, because they will not stall even when some paths fail. The failure detection time is within several seconds. We also propose a heuristic to find disjoint paths based on traceroute. We have implemented our system on top of overlay networks and evaluated it on PlanetLab and Emulab.

References

[1] http://dast.nlanr.net/projects/iperf/.

[2] http://www.emulab.net.

[3] http://www.isi.edu/nsnam/ns.

[4] H. Adiseshu, G. M. Parulkar, and G. Varghese. A reliable and scalable striping protocol. In *Proceedings of ACM SIGCOMM*, 1996.

[5] A. Akella, B. Maggs, S. Seshan, A. Shaikh, and R. Sitaraman. A measurement-based ananlysis of multihoming. In *Proceedings of ACM SIGCOMM*, Aug. 2003.

[6] A. Akella, S. Seshan, and A. Shaikh. An empirical evaluation of wide-area Internet bottlenecks. In *Proceedings of ACM Internet measurement conference*, Oct. 2003.

[7] M. Allman, H. Kruse, and S. Ostermann. An application-level solution to TCP's satellite inefficiencies. In *Proceedings of WOSBIS*, Nov. 1996.

[8] D. Andersen, H. Balakrishnan, F. Kaashoek, and R. Morris. Resilient overlay networks. *Proceedings of ACM SOSP*, Oct. 2001.

[9] J. Apostolopoulos, T. Wong, W. Tan, and S. Wee. On multiple description streaming with content delivery networks. In *Proceedings of IEEE INFOCOM*, 2002.

[10] A. Banerjea. Simulation study of the capacity effects of dispersity routing for fault tolerant realtime channels. *Proceedings of ACM SIGCOMM*, Aug. 1996.

[11] B. Chazelle. The discrepancy method: randomness and complexity. *Cambridge University Press*, 2000.

[12] J. Duncanson. Inverse multiplexing. In *IEEE Communications Magazine*, volume 32, pages 34–41, 1994.

[13] A. Feldmann, A. Greenberg, C. Lund, N. Reingold, J. Rexford, and F. True. Deriving traffic demands for operational IP networks: methodology and experience. *Proceedings of ACM SIGCOMM*, Aug. 2000.

[14] T. Hacker, B. Athey, and B. Noble. The end-to-end performance effects of parallel TCP sockets on a lossy wide-area network. In *Proc. of IPDPS*, 2002.

[15] K. Harfoush, A. Bestavros, and J. Byers. Robust identification of shared losses using end-to-end unicast probes. *Proceedings of IEEE ICNP*, Oct. 2000.

[16] H. Hsieh and R. Sivakumar. ptcp: An end-to-end transport layer protocol for striped connections. In *Proceedings of IEEE ICNP*, 2002.

[17] H. Hsieh and R. Sivakumar. A transport layer approach for achieving aggregate bandwidths on multi-homed mobile hosts. In *Proceedings of ACM MOBICOM*, 2002.

[18] D. Katabi, I. Bazzi, and X. Yang. An information theoretic approach for shared bottleneck inference based on end-to-end measurements. In *Class Project, MIT Laboratory for Computer Science*, 1999.

[19] J. Lee, D. Gunter, B. Tierney, B. Allcock, J. Bester, J. Bresnahan, and S. Tuecke. Applied techniques for high bandwidth data transfers across wide area networks. In *Proceedings of CHEP*, Sept. 2001.

[20] Y. Liang, E.G.Steinbach, and B. Girod. Real-time voice communication over the Internet using packet path diversity. In *Proceedings of ACM Multimedia*, 2001.

[21] L. Magalhaes and R. Kravets. Transport level mechanisms for bandwidth aggregation on mobile hosts. In *Proceedings of ICNP*, Nov. 2001.

[22] M. Mathis, J. Mahdavi, S. Floyd, and A. Romanow. TCP selective acknowledgement options. *RFC 2018*, Oct. 1996.

[23] A. Nakao, L. Peterson, and A. Bavier. A routing underlay for overlay networks. In *Proceedings of ACM SIGCOMM*, Aug. 2003.

[24] T. Nguyen and A. Zakhor. Path diversity with forward error correction (pdf) system for packet switched networks. In *Proceedings of IEEE INFOCOM*, 2003.

[25] L. Peterson, T. Anderson, D. Culler, and T. Roscoe. A blueprint for introducing disruptive technology into the Internet. *Proceedings of ACM HOTNET*, Oct. 2002.

[26] D. Rubenstein, J. Kurose, and D. Towsley. Detecting shared congestion of flows via end-to-end measurement. *Proceedings of ACM SIGMETRICS*, June 2000.

[27] S. Savage, A. Collins, and E. Hoffman. The end-to-end effects of Internet path selection. *Proceedings of ACM SIGCOMM*, Aug. 1999.

[28] H. Sivakumar, S. Bailey, and R. L. Grossman. PSockets: The case for application-level network striping for data intensive applications using high speed wide area networks. In *Supercomputing*, 2000.

[29] A. Snoeren. Adaptive inverse multiplexing for widearea wireless networks. In *Proc. of IEEE Conference on Global Communications*, 1999.

[30] N. Spring, R. Mahajan, and D. Wetherall. Measuring ISP topologies with rocketfuel. *Proceedings of ACM SIGCOMM*, Aug. 2002.

[31] 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. In *RFC 2960*, Oct. 2000.

[32] B. Traw and J. Smith. Striping within the network subsystem. In *IEEE Network*, volume 9, pages 22–32, 1995.

[33] O. Younis and S. Fahmy. On efficient on-line grouping of flows with shared bottlenecks at loaded servers. In *Proceedings of IEEE ICNP*, Nov. 2002.

[34] Y. Zhang, L. Breslau, V. Paxson, and S. Shenker. On the characteristics and origins of Internet flow rates. *Proceedings of ACM SIGCOMM*, Aug. 2002.

[35] Y. Zhang, N. Duffield, V. Paxson, and S. Shenkar. On the constancy of Internet path properties. *Proceedings of Internet Measurement Workshop*, Nov. 2001.