Design, implementation and evaluation of congestion control for multipath TCP

Damon Wischik, Costin Raiciu, Adam Greenhalgh, Mark Handley
University College London

ABSTRACT

Multipath TCP, as proposed by the IETF working group `mptcp`, allows a single data stream to be split across multiple paths. This has obvious benefits for reliability, and it can also lead to more efficient use of networked resources. We describe the design of a multipath congestion control algorithm, we implement it in Linux, and we evaluate it for multihomed servers, data centers and mobile clients. We show that some 'obvious' solutions for multipath congestion control can be harmful, but that our algorithm improves throughput and fairness compared to single-path TCP. Our algorithm is a drop-in replacement for TCP, and we believe it is safe to deploy.

1. INTRODUCTION

Multipath TCP, as proposed by the IETF working group `mptcp` [7], allows a single data stream to be split across multiple paths. This has obvious benefits for reliability—the connection can persist when a path fails. It can also have benefits for load balancing at multihomed servers and data centers, and for mobility, as we show below.

Multipath TCP also raises questions, some obvious and some subtle, about how network capacity should be shared efficiently and fairly between competing flows. This paper describes the design and implementation of a multipath congestion control algorithm that works robustly across a wide range of scenarios and that can be used as a drop-in replacement for TCP.

In §2 we propose a mechanism for windowed congestion control for multipath TCP, and then spell out the questions that led us to it. This section is presented as a walk through the design space signposted by pertinent examples and analysed by calculations and thought experiments. It is not an exhaustive survey of the design space, and we do not claim that our algorithm is optimal—to even define optimality would require a more advanced theoretical underpinning than we have yet developed. Some of the issues (§2.1–§2.3) have previously been raised in the literature on multipath congestion control, but not all have been solved. The others (§2.4–§2.5) are novel.

In §3–§5 we evaluate our algorithm in three application scenarios: multihomed Internet servers, data centers, and mobile devices. We do this by means of simulations with a high-speed custom packet-level simulator, and with testbed experiments on a Linux implementation. We show that multipath TCP is beneficial, as long as congestion control is done right. Naive solutions can be worse than single-path TCP.

In §6 we discuss what we learnt from implementing the protocol in Linux. There are hard questions about how to avoid deadlock at the receiver buffer when packets can arrive out of order, and about the datastream sequence space versus the subflow sequence spaces. But careful consideration of corner cases forced us to our specific implementation. In §7 we discuss related work on protocol design.

In this paper we will restrict our attention to end-to-end mechanisms for sharing capacity, specifically to modifications to TCP's congestion control algorithm. We will assume that each TCP flow has access to one or more paths, and it can control how much traffic to send on each path, but it cannot specify the paths themselves. For example, our Linux implementation uses multihoming at one or both ends to provide path choice, but it relies on the standard Internet routing mechanisms to determine what those paths are. Our reasons for these restrictions are (i) the IETF working group is working under the same restrictions, (ii) they lead to a readily deployable protocol, i.e. no modifications to the core of the Internet, and (iii) theoretical results indicate that inefficient outcomes may arise when both the end-systems and the core participate in balancing traffic [1].

2. THE DESIGN PROBLEM FOR MULTIPATH RESOURCE ALLOCATION

The basic window-based congestion control algorithm employed in TCP consists of additive increase behaviour when no loss is detected, and multiplicative decrease when a loss event is observed. In short:

ALGORITHM: REGULAR TCP

- Each ACK, increase the congestion window w by $1/w$, resulting in an increase of one packet per RTT.[1]
- Each loss, decrease w by $w/2$.

Additionally, at the start of a connection, an exponential increase is used, as it is immediately after a retransmission timeout. Newer versions of TCP [24, 9] have

[1]For simplicity, we express windows in this paper in packets, but real implementations usually maintain them in bytes.

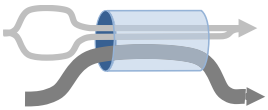


Figure 1: A scenario which shows the importance of weighting the aggressiveness of subflows.

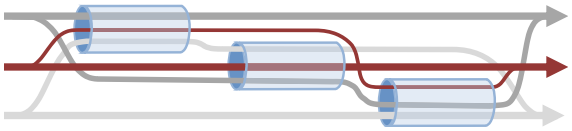


Figure 2: A scenario to illustrate the importance of choosing the less-congested path

faster behaviour when the network is underloaded; we believe our multipath enhancements can be straightforwardly applied to these versions, but it is a topic for further work.

The congestion control algorithm we propose is this:

ALGORITHM: MPTCP
A connection consists of set of subflows R, each of which may take a different route through the Internet. Each subflow $r \in R$ maintains its own congestion window w_r. An MPTCP sender stripes packets across these subflows as space in the subflow windows becomes available. The windows are adapted as follows:

- Each ACK on subflow r, for each subset $S \subseteq R$ that includes path r, compute

$$\frac{\max_{s \in S} w_s/\mathsf{RTT}_s^2}{\left(\sum_{s \in S} w_s/\mathsf{RTT}_s\right)^2}, \quad (1)$$

  then find the minimum over all such S, and increase w_r by that much. (The complexity of finding the minimum is linear in the number of paths, as we show in the appendix.)
- Each loss on subflow r, decrease the window w_r by $w_r/2$.

Here RTT_r is the round trip time as measured by subflow r. We use a smoothed RTT estimator, computed similarly to TCP.

In our implementation, we compute the increase parameter only when the congestion windows grow to accommodate one more packet, rather than every ACK on every subflow.

The following subsections explain how we arrived at this design. The basic question we set out to answer is how precisely to adapt the subflow windows of a multipath TCP so as to get the maximum performance possible, subject to the constraint of co-existing gracefully with existing TCP traffic.

2.1 Fairness at shared bottlenecks

The obvious question to ask is why not just run regular TCP congestion control on each subflow? Consider the scenario in Fig. 1. If multipath TCP ran regular TCP congestion control on both paths, then the multipath flow would obtain twice as much throughput as the single path flow (assuming all RTTs are equal). This is unfair. An obvious solution is to run a weighted version of TCP on each subflow, weighted so as to take some fixed fraction of the bandwidth that regular TCP would take. The weighted TCP proposed by [5] is not suitable for weights smaller than 0.5, so instead [11] consider the following algorithm, EWTCP.

ALGORITHM: EWTCP

- For each ACK on path r, increase window w_r by a/w_r.
- For each loss on path r, decrease window w_r by $w_r/2$.

Here w_r is the window size on path r, and $a = 1/\sqrt{n}$ where n is the number of paths.

Each subflow gets window size proportional to a^2 [11]. By choosing $a = 1/\sqrt{n}$, and assuming equal RTTs, the multipath flow gets the same throughput as a regular TCP at the bottleneck link. This is an appealingly simple mechanism in that it does not require any sort of explicit shared-bottleneck detection.

2.2 Choosing efficient paths

Athough EWTCP can be fair to regular TCP traffic, it would not make very efficient use of the network. Consider the somewhat contrived scenario in Fig.2, and suppose that the three links each have capacity 12Mb/s. If each flow split its traffic evenly across its two paths[2], then each subflow would get 4Mb/s hence each flow would get 8Mb/s. But if each flow used only the one-hop shortest path, it could get 12Mb/s. (In general, however, it is not efficient to always use only shortest paths, as the simulations in §4 of data center topologies show.)

A solution has been devised in the theoretical literature on congestion control, independently by [15] and [10]. The core idea is that a multipath flow should shift all its traffic onto the least-congested path. In a situation like Fig. 2 the two-hop paths will have higher drop probability than the one-hop paths, so applying the core idea will yield the efficient allocation. Surprisingly it

[2]In this topology EWTCP wouldn't actually split its traffic evenly, since the two-hop path traverses two bottleneck links and so experiences higher congestion. In fact, as TCP's throughput is inversely proportional to the square root of loss rate, EWTCP would end up sending approximately 3.5Mb/s on the two-hop path and 5Mb/s on the single-hop path, a total of 8.5Mb/s—slightly more than with an even split, but much less than with an optimal allocation.

turns out that this can be achieved (in theory) without any need to explicitly measure congestion[3]. Consider the following algorithm, called COUPLED[4]:

ALGORITHM: COUPLED

- For each ACK on path r, increase window w_r by $1/w_{\text{total}}$.
- For each loss on path r, decrease window w_r by $w_{\text{total}}/2$.

Here w_{total} is the total window size across all subflows. We bound w_r to keep it non-negative; in our experiments we bound it to be $\geq$ 1pkt, but for the purpose of analysis it is easier to think of it as ≥ 0.

To get a feeling for the behaviour of this algorithm, we now derive an approximate throughput formula. Consider first the case that all paths have the same loss rate p. Each window w_r is made to increase on ACKs, and made to decrease on drops, and in equilibrium the increases and decreases must balance out, i.e. rate of ACKs $\times$ average increase per ACK must equal rate of drops $\times$ average decrease per drop, i.e.

$$\left(\frac{w_{\text{r}}}{\text{RTT}}(1-p)\right)\frac{1}{w_{\text{total}}} = \left(\frac{w_{\text{r}}}{\text{RTT}}p\right)\frac{w_{\text{total}}}{2}. \qquad (2)$$

Solving for w_{total} gives $w_{\text{total}} = \sqrt{2(1-p)/p} \approx \sqrt{2/p}$ (where the approximation is good if p is small). Note that when there is just one path then COUPLED reduces to regular TCP, and that the formula for w_{total} does not depend on the number of paths, hence COUPLED automatically solves the fairness problem in §2.1.

For the case that the loss rates are not all equal, let p_r be the loss rate on path r and let p_{min} be the minimum loss rate seen over all paths. The increase and decrease amounts are the same for all paths, but paths with higher p_r will see more decreases, hence the equilibrium window size on a path with $p_r > p_{\text{min}}$ is $w_r = 0$. In Fig.2, the two-hop paths go through two congested links, hence they will have higher loss rates than the one-hop paths, hence COUPLED makes the efficient choice of using only the one-hop paths.

An interesting consequence of moving traffic away from more congested paths is that loss rates across the whole network will tend to be balanced. See §3 for experiments which demonstrate this. Or consider the network shown in Fig.3, and assume all RTTs are equal.

[3]Of course it can also be achieved by explicitly measuring congestion as in [11], but this raises tricky measurement questions.

[4]COUPLED is adapted from [15, equation (21)] and [10, equation (14)], which propose a differential equation model for a rate-based multipath version of ScalableTCP [16]. We applied the concepts behind the equations to classic window-based TCP rather than to a rate-based version of ScalableTCP, and translated the differential equations into a congestion control algorithm.

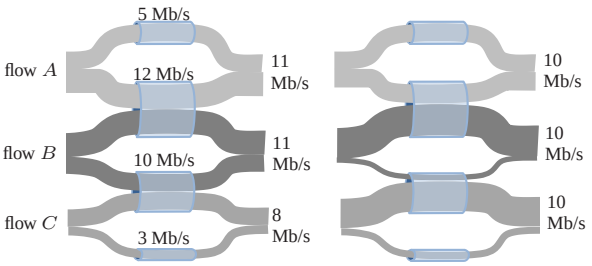


Figure 3: A scenario where EWTCP (left) does not equalize congestion or total throughput, whereas COUPLED (right) does.

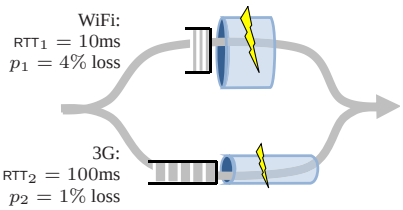


Figure 4: A scenario in which RTT and congestion mismatch can lead to low throughput.

Under EWTCP each link will be shared evenly between the subflows that use it, hence flow A gets throughputs 5 and 6 Mb/s, B gets 6 and 5 Mb/s, and C gets 5 and 3 Mb/s. Since TCP throughput is inversely related to drop probability, we deduce that the 3Mb/s link has the highest drop probability and the 12Mb/s link the lowest. For COUPLED, we can calculate the throughput on each subflow by using two facts: that a flow uses a path only if that path has the lowest loss rate p_{min} among its available paths, and that a flow's total throughput is proportional to $\sqrt{2/p_{\text{min}}}$; the only outcome consistent with these facts is for all four links to have the same loss rate, and for all flows to get the same throughput, namely 10Mb/s.

In this scenario the rule "only use a path if that path has lowest drop probability among available paths" leads to balanced congestion and balanced total throughput. In some scenarios, these may be desirable goals *per se*. Even when they are not the primary goals, they are still useful as a test: a multipath congestion control algorithm that does not balance congestion in Fig.3 is unlikely to make the efficient path choice in Fig.2.

2.3 Problems with RTT mismatch

Both EWTCP and COUPLED have problems when the RTTs are unequal. This is demonstrated by experiments in §5. To understand the issue, consider the scenario of a wireless client with two interfaces shown in Fig.4: the 3G path typically uses large buffers, resulting in long delays and low drop rates, whereas the wifi path might have smaller delays and higher drop rate. As

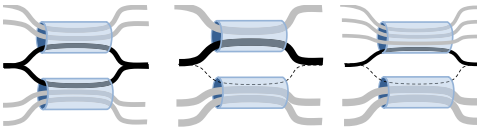


Figure 5: A scenario where multipath TCP might get 'trapped' into using a less desirable path.

a simple approximation, take the drop rates to be fixed (though in practice, e.g. in the experiments in §5, the drop rate will also depend on the sender's data rate). Also, take the throughput of a single-path TCP to be $\sqrt{2/p}/\text{RTT}$ pkt/s. Then

- A single-path WiFi flow would get 707 pkt/s, and a single-path 3G flow would get 141 pkt/s.
- EWTCP is half as aggressive as single-path TCP on each path, so it will get total throughput $(707 + 141)/2 = 424$ pkt/s.
- COUPLED will send all its traffic on the less congested path, on which it will gets the same window size as single-path TCP, so it will get total throughput 141 pkt/s.[5]

Both EWTCP and COUPLED are undesirable to a user considering whether to adopt multipath TCP.

One solution is to switch from window-based control to rate-based control; the rate-based equations [15, 10] that inspired COUPLED do not suffer from RTT mismatch. But this would be a drastic change to the Internet's congestion control architecture, a change whose time has not yet come. Instead, we have a practical suggestion for window-based control, which we describe in §2.5. First though we describe another problem with COUPLED and our remedy.

2.4 Adapting to load changes

It turns out there is another pitfall with COUPLED, which shows itself even when all subflows have the same RTT. Consider the scenario in Fig. 5. Initially there are two single-path TCPs on each link, and one multipath TCP able to use both links. It should end up balancing itself evenly across the two links, since if it were uneven then one link would be more congested than the other and COUPLED would shift some of its traffic onto the less congested. Suppose now that one of the flows on the top link terminates, so the top link is less congested, hence the multipath TCP flow moves all its traffic onto the top link. But then it is 'trapped': no matter how much extra congestion there is on the top link, the the multipath TCP flow is not using the bottom link, so it gets no ACKs on the bottom link, so COUPLED is unable to increase the window size on the bottom subflow. The same problem is demonstrated in experiments in §3.

We can conclude that the simple rule "Only use the least congested paths" needs to be balanced by an opposing consideration, "Always keep sufficient traffic on other paths, as a probe, so that you can quickly discover when they improve." In fact, our implementation of COUPLED keeps window sizes $\geq$ 1pkt, so it always has some probe traffic. And the theoretical works [15, equation (11)] and [10, equation (14)] that inspired COUPLED also have a parameter that controls the amount of probing; the theory says that with infinitesimal probing one can asymptotically (after a long enough time, and with enough flows) achieve fair and efficient allocations.

But we found in experiments that if there is too little probe traffic then feedback about congestion is too infrequent for the flow to discover changes in a reasonable time. Noisy feedback (random packet drops) makes it even harder to get a quick reliable signal. As a compromise, we propose the following.

ALGORITHM: SEMICOUPLED

- For each ACK on path r, increase window w_r by a/w_{total}.
- For each loss on path r, decrease window w_r by $w_r/2$.

Here a is a constant which controls the aggressiveness, discussed below.

SEMICOUPLED tries to keep a moderate amount of traffic on each path while still having a bias in favour of the less congested paths. For example, suppose a SEMICOUPLED flow is using three paths, two with drop probability 1% and a third with drop probability 5%. We can calculate equilibrium window sizes by a balance argument similar to (2); when $1 - p_r \approx 1$ the window sizes are

$$w_r \approx \sqrt{2a}\frac{1/p_r}{\sqrt{\sum_s 1/p_s}}.$$

In three-path example, the flow will put 45% of its weight on each of the less congested path and 10% on the more congested path. This is intermediate between EWTCP (33% on each path) and COUPLED (0% on the more congested path).

To achieve fairness in scenarios like Fig.1, one can fairly simply tune the a parameter. For more complicated scenarios like Fig.4, we need a more rigorous definition of fairness, which we now propose.

2.5 Compensating for RTT mismatch

In order to reason about bias and fairness in a principled way, we propose the following two requirements for multipath congestion control:

[5]The 'proportion manager' in the multipath algorithm of [11] will also move all the traffic onto the less congested path, with the same outcome.

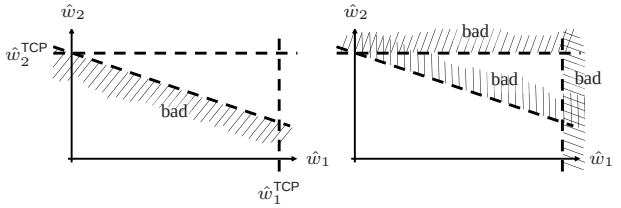


Figure 6: Fairness constraints for a two-path flow. Constraint (3) on the left, constraints (4) on the right.

- A multipath flow should give a connection at least as much throughput as it would get with single-path TCP on the best of its paths. This ensures there is an incentive for deploying multipath.
- A multipath flow should take no more capacity on any path or collection of paths than if it was a single-path TCP flow using the best of those paths. This guarantees it will not unduly harm other flows at a bottleneck link, no matter what combination of paths passes through that link.

In mathematical notation, suppose the set of available paths is R, let $\hat{w}_r$ be the equilibrium window obtained by multipath TCP on path r, and let $\hat{w}_r^{\text{TCP}}$ be the equilibrium window that would be obtained by a single-path TCP experiencing path r's loss rate. We shall require

$$\sum_{r\in R} \frac{\hat{w}_r}{\mathsf{RTT}_r} \geq \max_{r\in R} \frac{\hat{w}_r^{\text{TCP}}}{\mathsf{RTT}_r} \tag{3}$$

$$\sum_{r\in S} \frac{\hat{w}_r}{\mathsf{RTT}_r} \leq \max_{r\in S} \frac{\hat{w}_r^{\text{TCP}}}{\mathsf{RTT}_r} \quad \text{for all } S \subseteq R. \tag{4}$$

These constraints are illustrated, for a two-path flow, in Fig.6. The left hand figure illustrates (3), namely that $(\hat{w}_1, \hat{w}_2)$ should lie on or above the diagonal line. The exact slope of the diagonal is dictated by the ratio of RTTs, and here we have chosen them so that $\hat{w}_2^{\text{TCP}}/\mathsf{RTT}_2 > \hat{w}_1^{\text{TCP}}/\mathsf{RTT}_1$. The right hand figure illustrates the three constraints in (4). The constraint for $S = \{\text{path}_1\}$ says to pick a point on or left of the vertical line. The constraint for $S = \{\text{path}_2\}$ says to pick a point on or below the horizontal line. The joint bottleneck constraint ($S = \{\text{path}_1, \text{path}_2\}$) says to pick a point on or below the diagonal line. Clearly the only way to satisfy both (3) & (4) is to pick some point on the diagonal, inside the box; any such point is fair. (Separately, the considerations in §2.2 say we should prefer the less-congested path, and in this figure $\hat{w}_1^{\text{TCP}} > \hat{w}_2^{\text{TCP}}$ hence the loss rates satisfy $p_1 < p_2$, hence we should prefer the right hand side of the diagonal line.)

The following algorithm, a modification of SEMICOUPLED, satisfies our two fairness requirements, when the flow has two paths available. EWTCP can also be fixed with a similar modification. The experiments in §5 show that the modification works.

ALGORITHM

- Each ACK on subflow r, increase the window w_r by $\min(a/w_{\text{total}}, 1/w_r)$.
- Each loss on subflow r, decrease the window w_r by $w_r/2$.

Here

$$a = \hat{w}_{\text{total}} \frac{\max_r \hat{w}_r/\mathsf{RTT}_r^2}{(\sum_r \hat{w}_r/\mathsf{RTT}_r)^2}, \tag{5}$$

w_r is the current window size on path r and $\hat{w}_r$ is the equilibrium window size on path r, and similarly for w_{total} and $\hat{w}_{\text{total}}$.

The increase and decrease rules are similar to SEMICOUPLED, so the algorithm prefers less-congested paths. The difference is that the window increase is capped at $1/w_r$, which ensures that the multipath flow can take no more capacity on either path than a single-path TCP flow would, i.e. it ensures we are inside the horizontal and vertical constraints in Fig.6.

The parameter a controls the aggressiveness. Clearly if a is very large then the two flows act like two independent flows hence the equilibrium windows will be at the top right of the box in Fig.6. On the other hand if a is very small then the flows will be stuck at the bottom left of the box. As we said, the two fairness goals require that we exactly hit the diagonal line. The question is how to find a to achieve this.

We can calculate a from the balance equations. At equilibrium, the window increases and decreases balance out on each path, hence

$$(1-p_r)\min\Big(\frac{a}{\hat{w}_{\text{total}}}, \frac{1}{\hat{w}_r}\Big) = p_r \frac{\hat{w}_r}{2}.$$

Making the approximation that p_r is small enough that $1 - p_r \approx 1$, and writing it in terms of $\hat{w}_r^{\text{TCP}} = \sqrt{2/p_r}$,

$$\max\Big(\hat{w}_r, \frac{\hat{w}_{\text{total}}\hat{w}_r}{a}\Big) = \hat{w}_r^{\text{TCP}}. \tag{6}$$

By simultaneously solving (3) (with the inequality replaced by equality) and (6), we arrive at (5).

Our final MPTCP algorithm, specified at the beginning of §2, is a generalization of the above algorithm to an arbitrary number of paths. The proof that it satisfies (3)–(4) is in the appendix. The formula (5) technically requires $\hat{w}_r$, the equilibrium window size, whereas in our final algorithm we have used the instantaneous window size instead. The experiments described below indicate that this does not cause problems.

Trying too hard to be fair? Our fairness goals say "take no more than a single-path TCP". At first sight this seems overly restrictive. For example, consider a single-path user with a 14.4Mb/s WiFi access link, who

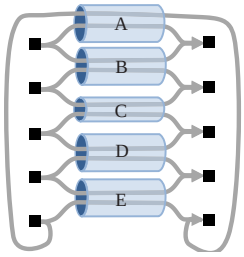


Figure 7: Torus topology. We adjust the capacity of link C, and test how well congestion is balanced.

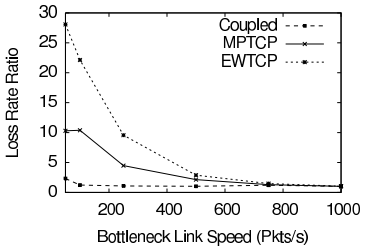


Figure 8: Effect of changing the capacity of link C on the ratio of loss rates p_C/p_A. All other links have capacity 1000pkt/s.

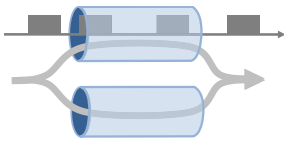


Figure 9: Bursty CBR traffic on the top link requires quick response by the multipath flow.

then adds a 2Mb/s 3G access link. Shouldn't this user now get 16.4Mb/s, and doesn't the fairness goal dictate 14.4Mb/s?

We describe tests of this scenario, and others like it, in §5. MPTCP does in fact give throughput equal to the sum of access link bandwidths, when there is no competing traffic. When there is competing traffic on the access links, the answer is different.

To understand what's going on, note that our precise fairness goals say "take no more than would be obtained by a single-path TCP *experiencing the same loss rate*". Suppose there is no competing traffic on either link, and the user only takes 14.4Mb/s. Then one or other of the two access links is underutilized, so it has no loss, and a hypothetical single-path TCP with no loss should get very high throughput, so the fairness goal allows MPTCP to increase its throughput. The system will only reach equilibrium once both access links are fully utilized. See §5 for further experimental results, including scenarios with competing traffic on the access links.

3. BALANCING CONGESTION AT A MULTIHOMED SERVER

In §3–§5 we investigate the behaviour of multipath TCP in three different scenarios: a multihomed Internet server, a data center, and a mobile client. Our aim in this paper is to produce one multipath algorithm that works robustly across a wide range of scenarios. These three scenarios will showcase all the design decisions dicussed in §2—though not all the design decisions are important in every one of the scenarios.

The first scenario is a multihomed Internet server. Multihoming of important servers has become ubiquitous over the last decade; no company reliant on network access for their business can afford to be dependent on a single upstream network. However, balancing traffic across these links is difficult, as evidenced by the hoops operators jump through using BGP techniques such as prefix splitting and AS prepending. Such techniques are coarse-grained, very slow, and a stress to the global routing system. In this section we will show that multipath transport can balance congestion, even when only a minority of flows are multipath-capable.

We will first demonstrate congestion balancing in a simple simulation, to illustrate the design discussion in §2 and to compare MPTCP to EWTCP and COUPLED. In the static scenario COUPLED is better than MPTCP is better than EWTCP, and in the dynamic scenario the order is reversed—but in each case MPTCP is close to the best, so it seems to be a reasonable compromise. We will then validate our findings with a result from an experimental testbed running our Linux implementation.

Static load balancing simulation. First we shall investigate load balancing in a stable environment of long-lived flows, testing the predictions in §2.2. Fig.7 shows a scenario with five bottleneck links arranged in a torus, each used by two multipath flows. All paths have equal RTT of 100ms, and the buffers are one bandwidth-delay product. We will adjust the capacity of link C. When the capacity of link C is reduced then it will become more congested, so the two flows using it should shift their traffic towards B and D, so those links become more congested, so there is a knock-on effect and the other flows should shift their traffic onto links A and E. With perfect balancing, the end result should be equal congestion on all links.

Fig.8 plots the imbalance in congestion as a function of the capacity of link C. When all links have equal capacity ($C = 1000$pkt/s) then congestion is of course perfectly balanced for all the algorithms. When link C is smaller, the imbalance is greater. COUPLED is very good at balancing congestion, EWTCP is bad, and MPTCP is in between. We also find that balanced congestion results in better fairness between total flow rates: when link C has capacity 100 pkt/s then Jain's fairness index is 0.99 for the flow rates with COUPLED, 0.986 for MPTCP and 0.92 for EWTCP.

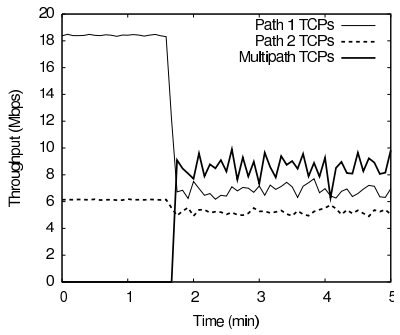


Figure 10: Server load balancing with MPTCP

Dynamic load balancing simulation. Next we illustrate the problem with dynamic load described in §2.4. We ran a simulation with two links as in Fig.9, both of capacity 100Mb/s and buffer 50 packets, and one multipath flow where each path has a 10ms RTT. On the top link there is an additional bursty CBR flow which sends at 100Mb/s for a random duration of mean 10ms, then is quiet for a random duration of mean 100ms. The multipath flow ought to use only the bottom link when the CBR flow is present, and it ought to quickly take up both links when the CBR flow is absent. We reasoned that COUPLED would do badly, and the throughputs we obtain confirm this. In Mb/s, they are

	top link	bottom link
EWTCP	85	100
MPTCP	83	99.8
COUPLED	55	99.4

We have found similar problems in a wide range of different scenarios. The exact numbers depend on how quickly congestion levels change, and in this illustration we have chosen particularly abrupt changes. One might expect similarly abrupt changes for a mobile devices when coverage on one radio interface is suddenly lost and then recovers.

Server load balancing experiment. We next give results from an experimental testbed that show our Linux implementation of MPTCP balancing congestion, validating the simulations we have just presented.

We first ran a server dual-homed with two 100Mb/s links and a number of client machines. We used dummynet to add 10ms of latency to simulate a wide-area scenario. We ran 5 client machines connecting to the server on link 1 and 15 on link 2, both using long-lived flows of Linux 2.6 NewReno TCP. The first minute of Fig.10 shows the throughput that is achieved—clearly there is more congestion on link 2. Then we started 10 multipath flows able to use both links. Perfect load balancing would require these new flows to shift completely to link 1. This is not perfectly achieved, but nonetheless multipath helps significantly to balance load, despite constituting only $1/3$ the total number of flows. The figure only shows MPTCP; COUPLED was similar and EWTCP was slightly worse as it pushed more traffic onto link 2.

Our second experiment used the same topology. On link 1 we generated Poisson arrivals of TCP flows with rate alternating between 10/s (light load) and 60/s (heavy load), with file sizes drawn from a Pareto distribution with mean 200kB. On link 2 we ran a single long-lived TCP flow. We also ran three multipath flows, one for each multipath algorithm. Their average throughputs were 61Mb/s for MPTCP, 54Mb/s for COUPLED, and 47Mb/s for EWTCP. In heavy load EWTCP did worst because it did not move as much traffic onto the less congested path. In light load COUPLED did worst because bursts of traffic on link 1 pushed it onto link 2, where it remained 'trapped' even after link 1 cleared up.

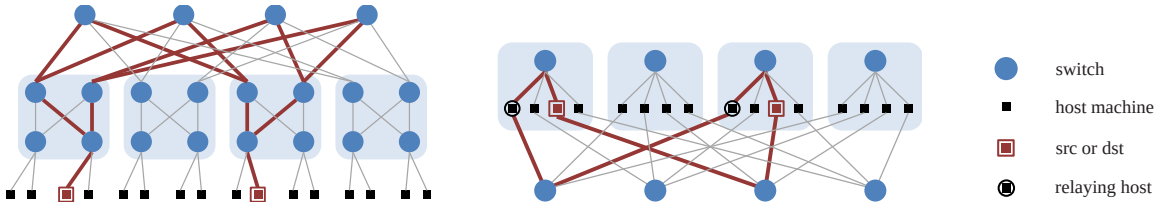
4. EFFICIENT ROUTING IN DATA CENTERS

Growth in cloud applications from companies such as Google, Microsoft and Amazon has resulted in huge data centers in which significant amounts of traffic are shifted between machines, rather than just out to the Internet. To support this, researchers have proposed new architectures with much denser interconnects than have traditionally been implemented. Two such proposals, FatTree [2] and BCube [8], are illustrated in Fig.11. The density of interconnects means that there are many possible paths between any pair of machines. The challenge is: how can we ensure that the load is efficiently distributed, no matter the traffic pattern?

One obvious benefit of any sort of multipath TCP in data centers is that it can alleviate bottlenecks at the host NICs. For example in BCube, Fig.11(b), if the core is lightly loaded and a host has a single large flow then it makes sense to use both available interfaces.

Multipath TCP is also beneficial when the network core is the bottleneck. To show this, we compared multipath TCP to single-path TCP with Equal Cost Multipath (ECMP), which we simulated by making each TCP source pick one of the shortest-hop paths at random. We ran packet-level simulations of FatTree with 128 single-interface hosts and 80 eight-port switches, and for each pair of hosts we selected 8 paths at random to use for multipath. (Our reason for choosing 8 paths is discussed below.) We also simulated BCube with 125 three-interface hosts and 25 five-port switches, and for each pair of hosts we selected 3 edge-disjoint paths according to the BCube routing algorithm, choosing the intermediate nodes at random when the algorithm needed a choice. All links were 100Mb/s.

We simulated three traffic patterns, all consisting of



(a) FatTree (b) BCube

Figure 11: Two proposed data center topologies. The bold lines show multiple paths between the source and destination.

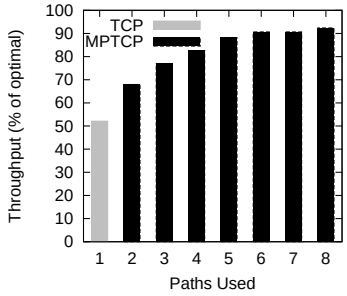


Figure 12: Multipath needs 8 paths to get good utilization in FatTree

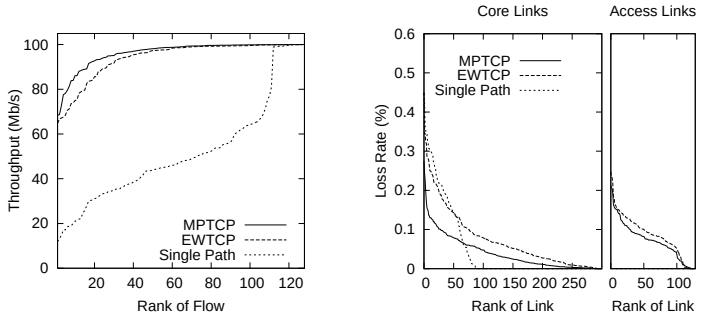


Figure 13: Distribution of throughput and loss rate, in 128-node FatTree

long-lived flows. TP1 is a random permutation where each host opens a flow to a single destination chosen uniformly at random, such that each host has a single incoming flow. For FatTree, this is the least amount of traffic that can fully utilize the network and is a good test for overall utilization. In TP2 each host opens 12 flows to 12 destinations; in FatTree the destinations are chosen at random, while in BCube they are the host's neighbours in the three levels. This mimics the locality of communication of writes in a distributed filesystem, where replicas of a block may be placed close to each other in the physical topology in order to allow higher throughput. We are using a high number of replicas as a stress-test of locality. Finally, TP3 is a sparse traffic pattern: 30% of the hosts open one flow to a single destination chosen uniformly at random.

FatTree simulations. The per-host throughputs obtained in FatTree in Mb/s, are:

	TP1	TP2	TP3
SINGLE-PATH	51	94	60
EWTCP	92	92.5	99
MPTCP	95	97	99

These figures show that for all three traffic patterns, both EWTCP and MPTCP have enough path diversity to 'find' nearly all the capacity in the network, as we can see from the fact that they get close to full utilization of the machine's 100Mb/s interface card. Fig.12 shows the throughput achieved as a function of paths used, for MPTCP under TP1—we have found that 8 is enough to get 90% utilization, in simulations across a range of traffic matrices and with thousands of hosts.

Average throughput figures do not give the full picture. Fig.13 shows the distribution of throughput on each flow, and of loss rate on each link, obtained by the three algorithms, for traffic pattern TP1. We see that MPTCP does a better job of allocating throughput fairly than EWTCP, for the reasons discussed in §2.2 and §3. Fairness matters for many datacenter distributed computations that farm processing out to many nodes and are limited by the response time of the slowest node. We also see that MPTCP does a better job of balancing congestion.

BCube simulations. The per-host throughputs obtained in BCube, in Mb/s, are:

	TP1	TP2	TP3
SINGLE-PATH	64.5	297	78
EWTCP	84	229	139
MPTCP	86.5	272	135

These throughput figures reflect three different phenomena. First, both multipath algorithms allow a host to use all three of its interfaces whereas single-path TCP can use only one, so they allow higher throughput. This is clearest in the sparse traffic pattern TP3, where the network core is underloaded. Second, BCube paths may have different hop counts, hence they are likely to traverse different numbers of bottlenecks, so some paths will be more congested than others. As discussed in §2.2, an efficient multipath algorithm should shift its

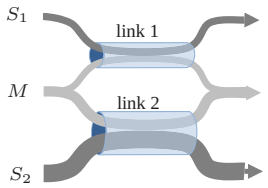


Figure 14: A multipath flow competing against two single-path flows

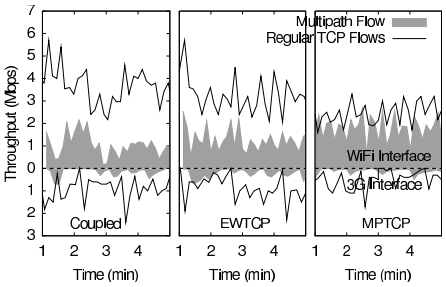


Figure 15: Multipath TCP throughput compared to single-path, where link 1 is WiFi and link 2 is 3G.

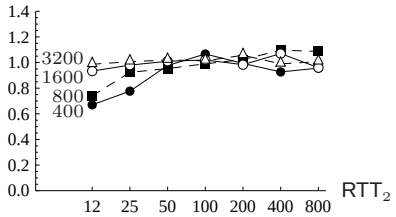


Figure 16: The ratio of flow M's throughput to the better of flow S_1 and S_2, as we vary link 2 in Fig.14.

traffic away from congestion, and EWTCP does not do this hence it tends to get worse throughput than MPTCP. This is especially clear in TP2, and not noticeable in TP3 where the core has little congestion. Third, even MPTCP does not move *all* its traffic away from the most congested path, for the reasons discussed in §2.4, so when the least-congested paths happen to all be shortest-hop paths then shortest-hop single-path TCP will do better. This is what happened in TP2. (Of course it is not always true that the least congested paths are all shortest-hop paths, so shortest-hop single-path TCP does poorly in other cases.)

In summary, MPTCP performs well across a wide range of traffic patterns. In some cases EWTCP achieves throughput as good as MPTCP, and in other cases it falls short. Even when its average throughput is as good as MPTCP it is less fair.

We have compared multipath TCP to single-path TCP, assuming that the single path is chosen at random from the shortest-hop paths available. Randomization goes some way towards balancing traffic, but it is likely to cause some congestion hotspots. An alternative solution for balancing traffic is to use a centralized scheduler which monitors large flows and solves an optimization problem to calculate good routes for them [3]. We have found that, in order to get comparable performance to MPTCP, one may need to re-run the scheduler as often as every 100ms [22] which raises serious scalability concerns. However, the exact numbers depend on the dynamics of the traffic matrix.

5. MULTIPATH WIRELESS CLIENT

Modern mobile phones and devices such as Nokia's N900 have multiple wireless interfaces such as WiFi and 3G, yet only one of them is used for data at any given time. With more and more applications requiring Internet access, from email to navigation, multipath can improve mobile users' experience by allowing simultaneous use of both interfaces. This shields the user from the inherently variable connectivity of wireless networks.

3G and WiFi have quite different link characteristics. WiFi provides much higher throughput and short RTTs, but in our tests its performance was very variable with quite high loss rates, because there was significant interference in the 2.4GHz band. 3G tends to vary over longer timescales, and we found it to be overbuffered leading to RTTs of well over a second. These differences provide a good test of the fairness goals and RTT compensation algorithm developed in §2.5. The experiments we describe here show that MPTCP gives users at least as much throughput as single-path users, and that the other multipath algorithms we have described do worse.

Single-flow experiment. Our first experiments use a laptop equipped with a 3G USB interface and a 802.11 network adapter, running our Linux implementation of MPTCP. The laptop was placed in the same room as the WiFi basestation, and 3G reception was good. The laptop did not move, so the path characteristics were reasonably static. We ran 15 tests of 20 seconds each: 5 with single-path TCP on WiFi, 5 with single-path TCP on 3G, and 5 with MPTCP. The average throughputs (with standard deviations) were 14.4 (0.2), 2.1 (0.2) and 17.3 (0.7) Mb/s respectively. As we would wish, the MPTCP user gets bandwidth roughly equal to the sum of the bandwidths of the access links.

Competing-flows experiment. We repeated the experiment, but now with competing single-path TCP flows on each of the paths, as in Fig.14. In order to showcase our algorithm for RTT compensation we repeated the experiment but replacing MPTCP first with EWTCP and then with COUPLED. The former does not have any RTT compensation built in, although the technique we used for MPTCP could be adapted. For the latter, we do not know how to build in RTT compensation.

Fig.15 shows the total throughput obtained by each of the three flows over the course of 5 minutes, one plot for each of the three multipath algorithms. The top half

of the figure shows the bandwidth achieved on the WiFi path, the bottom half shows (inverted) the throughput on the 3G path, and the range of the grey area extending into both halves shows the throughput the multipath algorithms achieved on both paths.

The figure shows that only MPTCP gives the multipath flow a fair total throughput, i.e. approximately as good as the better of the single-path competing flows, which in this case is the WiFi flow. The pictures are somewhat choppy: it seems that the WiFi basestation is underbuffered, hence the TCP sawtooth leads to peaks and troughs in throughput as measured at the receiver; it also seems the 3G link has bursts of high speed, perhaps triggered by buffer buildup. Despite these experimental vicissitudes, the long-run averages show that MPTCP does a much better job of getting fair total throughput. The long-run average throughputs in Mb/s, over 5 minutes of each setup, are:

	multipath	TCP-WiFi	TCP-3G
EWTCP	1.66	3.11	1.20
COUPLED	1.41	3.49	0.97
MPTCP	2.21	2.56	0.65

These numbers match the predictions in §2.3. COUPLED sends all its traffic on the less congested path so it often chooses to send on the 3G path and hardly uses the WiFi path. EWTCP splits its traffic so it gets the average of WiFi and 3G throughput. Only MPTCP gets close to the correct total throughput. The shortfall (2.21Mb/s for MPTCP compared to 2.56Mb/s for the best single-path TCP) may be due to difficulty in adapting to the rapidly changing 3G link speed; we continue to investigate how quickly multipath TCP should adapt to changes in congestion.

Simulations. In order to test RTT compensation across a wider range of scenarios, we simulated the topology in Fig.14 with two wired links, with capacities $C_1 = 250$pkt/s and $C_2 = 500$pkt/s, and propagation delays $\mathsf{RTT}_1 = 500$ms and $\mathsf{RTT}_2 = 50$ms. At first sight we might expect each flow to get 250pkt/s. The simulation outcome is very different: flow S_1 gets 130pkt/s, flow S_2 gets 315pkt/s and flow M gets 305pkt/s; the drop probabilities are $p_1 = 0.22\%$ and $p_2 = 0.28\%$. After some thought we realize this outcome is very nearly what we designed the algorithm to achieve. As discussed in §2.5, flow M says 'What would a single-path TCP get on path 2, based on the current loss rate? I should get at least as much!' and decides its throughput should be around 315pkt/s. It doesn't say 'What would a single-path TCP get on path 2 if I used only path 2?' which would give the answer 250pkt/s. The issue is that the multipath flow does not take account of how its actions would affect drop probabilities when it decides on its fair rate. It is difficult to see any practical alternative.

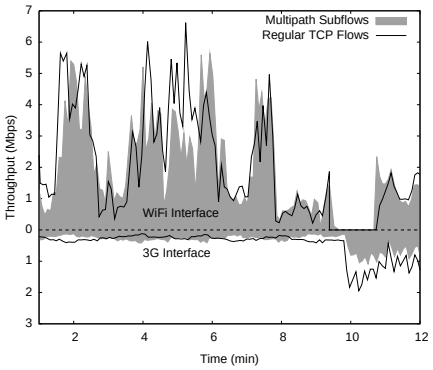


Figure 17: Throughput of multipath and regular TCP running simultaneously over 3G and WiFi. The 3G graph is shown inverted, so the total multipath throughput (the grey area) can be seen clearly.

And nonetheless, the outcome in this case is still better for both S_1 and M than if flow M used only link 1, and it is better for both S_2 and M than if flow M used only link 2.

We repeated the experiment, but with $C_1 = 400$pkt/s, $\mathsf{RTT}_1 = 100$ms, and a range of values of C_2 (shown as labels in Fig.16) and RTT_2 (the horizontal axis). Flow M aims to do as well as the better of flows S_1 and S_2. Fig.16 shows it is within a few percent of this goal in all cases except where the bandwidth delay product on link 2 is very small; in such cases there are problems due to timeouts. Over all of these scenarios, flow M always gets better throughput by using multipath than if it used just the better of the two links; the average improvement is 15%.

Mobile experiment. Having shown that our RTT compensation algorithm works in a rather testing wireless environment, we now wish to see how MPTCP performs when the client is mobile and both 3G and WiFi connectivity are intermittent. We use the same laptop and server as in the static experiment, but now the laptop user moves between floors of the building. The building has reasonable WiFi coverage on most floors but not on the staircases. 3G coverage is acceptable but is sometimes heavily congested by other users.

The experiment starts with one TCP running over the 3G interface and one over WiFi, both downloading data from an otherwise idle university server. A multipath flow then starts, using both interfaces, downloading data from the same server. Fig.17 shows the throughputs over each link (each point is a 5s average). Again, WiFi is shown above the dashed line, 3G is shown inverted below the dashed line, and the total throughput of the multipath flow can be clearly seen from the vertical range of the gray region.

During the experiment the subject moves around the building. For the first 9 minutes the 3G path has less congestion, so MPTCP would prefer to send its traffic on that route. But it also wants to get as much throughput as the higher-throughput path, in this case WiFi. The fairness algorithm prevents it from sending this much traffic on the 3G path, so as not to out-compete other single path TCPs that might be using 3G, and so the remainder is sent on WiFi. At 9 minutes the subject walks downstairs to go to a coffee machine. On the stairwell there is no WiFi coverage, but 3G coverage is better, so MPTCP adapts and takes advantage. When the subject leaves the stairwell, a new WiFi basestation is acquired, and multipath quickly takes advantage of it. This single trace shows the robustness advantage of multipath TCP, and it also shows that it does a good job of utilizing different links simultaneously without harming competing traffic on those links.

6. PROTOCOL IMPLEMENTATION

Although this paper primarily focuses on the congestion control dynamics of MPTCP, the protocol changes to TCP needed to implement multipath can be quite subtle. In particular, we must to be careful to avoid deadlock in a number of scenarios, especially relating to buffer management and flow control. In fact we discovered there is little choice in many aspects of the design. There are also many tricky issues regarding middleboxes which further constrain the design, not described here. A more complete exposition of these constraints can be found in [21], and our protocol is precisely described in the current `mptcp` draft [7].

Subflow establishment. Our implementation of MPTCP requires both client and server to have multipath extensions. A TCP option in the SYN packets of the first subflow is used to negotiate the use of multipath if both ends support it, otherwise they fall back to regular TCP behavior. After this, additional subflows can be initiated; a TCP option in the SYN packets of the new subflows allows the recipient to tie the subflow into the existing connection. We rely on multiple interfaces or multiple IP addresses to obtain different paths; we have not yet studied the question of when additional paths should be started.

Loss Detection and Stream Reassembly. Rgeular TCP uses a single sequence space for both loss detection and reassembly of the application data stream. With MPTCP, loss is a subflow issue, but the application data stream spans all subflows. To accomplish both goals using a single sequence space, the sequence space would need to be striped across the subflows. To detect loss, the receiver would then need to use selective acknowledgments and the sender would need to keep a scoreboard of which packets were sent on each subflow. Retransmitting packets on a different subflow creates an ambiguity, but the real problem is middleboxes that are unaware of MPTCP traffic. For example, the *pf*[19] firewall can re-write TCP sequence numbers to improve the randomness of the initial sequence number. If only one of the subflows passes through such a firewall, the receiver cannot reliably reconstruct the data stream.

To avoid such issues, we separated the two roles of sequence numbers. The sequence numbers and cumulative ack in the TCP header are per-subflow, allowing efficient loss detection and fast retransmission. Then to permit reliable stream reassembly, an additional data sequence number is added stating where in the application data stream the payload should be placed.

Flow Control. TCP's flow control is implemented via the combination of the receive window field and the acknowledgment field in the TCP packet header. The receive window indicates the number of bytes beyond the acknowledged sequence number that the receiver can buffer. The sender is not permitted to send more than this amount of additional data.

Multipath TCP also needs to implement flow control, although packets now arrive over multiple subflows. Two choices seem feasible:

- separate buffer pools are maintained at the receiver for each subflow, and their occupancy is signalled relative to the subflow sequence space using the receive window field.
- a single buffer pool is maintained at the receiver, and its occupancy is signalled relative to the data sequence space using the receive window field.

Unfortunately the former suffers from potential deadlock. Suppose subflow 1 stalls due to an outage, but subflow 2's receive buffer fills up. The packets received from subflow 2 cannot be passed to the application because a packet from subflow 1 is still missing, but there is no space in subflow 2's receive window to resend the packet from subflow 1 that is missing. To avoid this we use a single shared buffer; all subflows report the receive window relative to the last consecutively received data in the data sequence space.

Does the data cumulative ack then need to be explicit, or can it be inferred from subflow acks by keeping track of which data corresponds to which subflow sequence numbers?

Consider the following scenario: a receiver has sufficient buffering for two packets[6]. In accordance with the receive window, the sender sends two packets; data segment 1 is sent on subflow 1 with subflow sequence num-

[6]The same issue occurs with larger buffers

ber 10, and data segment 2 is sent on subflow 2 with subflow sequence number 20. The receiver acknowledges the packets using subflow sequence numbers only; the sender will infer which data is being acknowledged. Initially, the inferred cumulative ack is 0.

i. In the Ack for 10, the receiver acks data 1 in order, but the receiving application has not yet read the data, so relative to 1, the receive window is closed to 1 packet.
ii. In the Ack for 20, the receiver acks data 2 in order. As the application still has not read, relative to 2 the receive window is now zero.
iii. Unfortunately the acks are reordered simply because the RTT on path 2 is shorter than that on path 1, a common event. The sender receives the Ack for 20, infers that 2 has been received but 1 has not. The data cumulative ack is therefore still 0.
iv. When the ack for 10 arrives, the receiver infers that 1 and 2 have been received, so the data cumulative ack is now 2. The receive window indicated is 1 packet, relative to the inferred cumulative ack of 2. Thus the sender can send packet 3. Unfortunately, the receiver cannot buffer 3 and must drop it.

In general, the problem is that although it is possible to infer a data cumulative ack from the subflow acks, it is not possible to reliably infer the trailing edge of the receive window. The result is either missed sending opportunities or dropped packets. This is not a corner case; it will occur whenever RTTs differ so as to cause the acks to arrive in a different order from that in which they were sent.

To avoid this problem (and some others related to middleboxes) we add an explicit data acknowledgment field in addition to the subflow acknowledgment field in the TCP header.

Encoding. How should be data sequence numbers and data acknowledgments be encoded in TCP packets? Two mechanisms seemed feasible: carry them in TCP options or embed them in the payload using an SSL-like chunking mechanism. For data sequence numbers there is no compelling reason to choose one or the other, but for data acknowledgements the situation is more complex.

For the sake of concreteness, let us assume that a hypothetical payload encoding uses a chunked TLV structure, and that a data ack is contained in its own chunk, interleaved with data chunks flowing in the same direction. As data acks are now part of the data stream, they are subject to congestion control and flow control. This can lead to potential deadlock scenarios.

Consider a scenario where A's receive buffer is full because the application has not read the data, but A's application wishes to send data to B whose receive buffer is empty. This might occur for example when B is pipelining requests to A, and A now needs to send the response to an earlier request to B before reading the next request.

A sends its data, B stores it locally, and wants to send the data ACK, but can't do so: flow control imposed by A's receive window stops him. Because no data acks are received from B, A cannot free its send buffer, so this fills up and blocks the sending application on A. The connection is now deadlocked. A's application will only read when it has finished sending data to B, but it cannot do so because his send buffer is full. The send buffer can only empty when A receives an data ack from B, but B cannot send a data ack until A's application reads. This is a classic deadlock cycle.

In general, flow control of acks seems to be dangerous. Our implementation conveys data acks using TCP options to avoid this and similar issues. Given this choice, we also encode data sequence numbers in TCP options.

7. RELATED WORK

There has been a good deal of work on building multipath transport protocols [13, 27, 18, 12, 14, 6, 23, 7]. Most of this work focuses on the protocol mechanisms needed to implement multipath transmission, with key goals being robustness to long term path failures and to short term variations in conditions on the paths. The main issues are what we discussed in §6: how to split sequence numbers across paths (i.e. whether to use one sequence space for all subflows or one per subflow with an extra connection-level sequence number), how to do flow control (subflow, connection level or both), how to ack, and so forth. Our protocol design in §6 has drawn on this literature.

However, the main focus of this paper is congestion control not protocol design. In most existing proposals, the problem of shared bottlenecks (§2.1) is considered but the other issues in §2 are not. Let us highlight the congestion control characteristics of these proposals.

pTCP [12], CMT over SCTP[14] and M/TCP [23] use uncoupled congestion control on each path, and are not fair to competing single-path traffic in the general case.

mTCP [27] also performs uncoupled congestion control on each path. In an attempt to detect shared congestion at bottlenecks it computes the correlation between fast retransmit intervals on different subflows. It is not clear how robust this detector is.

R-MTP [18] targets wireless links: it probes the bandwidth available periodically for each subflow and adjusts the rates accordingly. To detect congestion it uses packet interarrival times and jitter, and infers mounting congestion when it observes increased jitter. This only works when wireless links are the bottleneck.

The work in [11] is based on using EWTCP with different weights on each path, and adapting the weights to

achieve the outcomes described in §2.1–§2.2. It does not address the problems identified in §2.3–§2.5, and in particular it has problems coping with heterogenous RTTs.

Network layer multipath. ECMP[25] achieves load balancing at the flow level, without the involvement of end-systems. It sends all packets from a given flow along the same route in order that end-systems should not see any packet re-ordering. ECMP and multipath TCP complement each other. Multipath TCP can use ECMP to get different paths through the network without having multihomed endpoints. Different subflows of the same multipath connection will have different five-tuples (at least one port will differ) and will likely hash onto a different path with ECMP. This interaction can be readily used in data centers, where multiple paths are available and ECMP is widely used.

Horizon [20] is a system for load balancing at the network layer, for wireless mesh networks. Horizon network nodes maintain congestion state and estimated delay for each possible path towards the destination; hop-by-hop backpressure is applied to achieve near-optimal throughput, and the delay estimates let it avoid re-ordering. Theoretical work suggests that inefficient outcomes may arise when both the end-systems and the network participate in balancing traffic [1].

Application layer multipath. BitTorrent [4] is an example of application layer multipath. Different chunks of the same file are downloaded from different peers to increase throughput. BitTorrent works at chunk granularity, and only optimizes for throughput, downloading more chunks from faster servers. Essentially BitTorrent is behaving in a similar way to uncoupled multipath congestion control, albeit with the paths having different endpoints. While uncoupled congestion control does not balance flow rates, it nevertheless achieves some degree of load balancing when we take into account flow sizes [17, 26], by virtue of the fact that the less congested subflow gets higher throughput and therefore fewer bytes are put on the more congested subflow.

8. CONCLUSIONS & FUTURE WORK

We have demonstrated a working multipath congestion control algorithm. It brings immediate practical benefits: in §5 we saw it seamlessly balance traffic over 3G and WiFi radio links, as signal strength faded in and out. It is safe to use: the fairness mechanism from §2.5 ensures that it does not harm other traffic, and that there is always an incentive to turn it on because its aggregate throughput is at least as good as would be achieved on the best of its available paths. It should be beneficial to the operation of the Internet, since it selects efficient paths and balances congestion, as described in §2.2 and demonstrated in §3, at least in so far as it can given topological constraints and the requirements of fairness.

We believe our multipath congestion control algorithm is safe to deploy, either as part of the IETF's efforts to standardize Multipath TCP[7] or with SCTP, and it will perform well. This is timely, as the rise of multipath-capable smart phones and similar devices has made it crucial to find a good way to use multiple interfaces more effectively. Currently such devices use heuristics to periodically choose the best interface, terminating existing connections and re-establishing new ones each time a switch is made. Combined with a transport protocol such as Multipath TCP or SCTP, our congestion control mechanism avoids the need to make such binary decisions, but instead allows continuous and rapid rebalancing on short timescales as wireless conditions change.

Our congestion control scheme is designed to be compatible with existing TCP behavior. However, existing TCP has well-known limitations when coping with long high-speed paths. To this end, Microsoft incorporate Compound TCP[24] in Vista and Windows 7, although it is not enabled by default, and recent Linux kernels use Cubic TCP[9]. We believe that Compound TCP should be a very good match for our congestion control algorithm. Compound TCP kicks in when a link is underutilized to rapidly fill the pipe, but it falls back to NewReno-like behavior once a queue starts to build. Such a delay-based mechanism would be complementary to the work described in this paper, but would further improve a multipath TCP's ability to switch to a previously congested path that suddenly has spare capacity. We intend to investigate this in future work.

9. REFERENCES

[1] D. Acemoglu, R. Johari, and A. Ozdaglar. Partially optimal routing. *IEEE Journal of selected areas in communications*, 2007.

[2] M. Al-Fares, A. Loukissas, and A. Vahdat. A scalable, commodity data center network architecture. In *Proc. SIGCOMM*, 2008.

[3] M. Al-Fares, S. Radhakrishnan, B. Raghavan, N. Huang, and A. Vahdat. Hedera: Dynamic flow scheduling for data center networks. In *Proc. NSDI*, 2010.

[4] B. Cohen. Incentives build robustness in BitTorrent. In *Workshop on econonomics of peer-to-peer systems*, 2003.

[5] J. Crowcroft and P. Oechslin. Differentiated end-to-end Internet services using a weighted proportional fair sharing TCP. *CCR*, 1998.

[6] Y. Dong, D. Wang, N. Pissinou, and J. Wang. Multi-path load balancing in transport layer. In *Proc. 3rd EuroNGI Conference on Next Generation Internet Networks*, 2007.

[7] A. Ford, C. Raiciu, and M. Handley. TCP extensions for multipath operation with multiple addresses, Oct 2010. IETF draft (work in progress).

[8] C. Guo, G. Lu, D. Li, H. Wu, X. Zhang, Y. Shi, C. Tian, Y. Zhang, and S. Lu. Bcube: a high performance,

server-centric network architecture for modular data centers. In *Proc. SIGCOMM*, 2009.
[9] S. Ha, I. Rhee, and L. Xu. CUBIC: a new TCP-friendly high-speed TCP variant. *SIGOPS Oper. Syst. Rev.*, 42(5), 2008.
[10] H. Han, S. Shakkottai, C. V. Hollot, R. Srikant, and D. Towsley. Multi-path TCP: a joint congestion control and routing scheme to exploit path diversity in the Internet. *IEEE/ACM Trans. Networking*, 14(6), 2006.
[11] M. Honda, Y. Nishida, L. Eggert, P. Sarolahti, and H. Tokuda. Multipath Congestion Control for Shared Bottleneck. In *Proc. PFLDNeT workshop*, May 2009.
[12] H.-Y. Hsieh and R. Sivakumar. A transport layer approach for achieving aggregate bandwidths on multi-homed mobile hosts. In *Proc. MobiCom '02*, pages 83–94, New York, NY, USA, 2002. ACM.
[13] C. Huitema. Multi-homed TCP. Internet draft, IETF, 1995.
[14] J. R. Iyengar, P. D. Amer, and R. Stewart. Concurrent multipath transfer using SCTP multihoming over independent end-to-end paths. *IEEE/ACM Trans. Netw.*, 14(5):951–964, 2006.
[15] F. Kelly and T. Voice. Stability of end-to-end algorithms for joint routing and rate control. *CCR*, 35(2), Apr. 2005.
[16] T. Kelly. Scalable TCP: improving performance in highspeed wide area networks. *SIGCOMM Comput. Commun. Rev.*, 33(2):83–91, 2003.
[17] P. Key, L. Massoulié, and D. Towsley. Path selection and multipath congestion control. In *Proc. IEEE Infocom*, May 2007. Also appeared in proceedings of IEEE ICASSP 2007.
[18] L. Magalhaes and R. Kravets. Transport level mechanisms for bandwidth aggregation on mobile hosts. *ICNP*, page 0165, 2001.
[19] *PF: the OpenBSD Packet Filter*. OpenBSD 4.7, `www.openbsd.org/faq/pf`, retrieved Sep 2010.
[20] B. Radunović, C. Gkantsidis, D. Gunawardena, and P. Key. Horizon: balancing TCP over multiple paths in wireless mesh network. In *Proc. MobiCom '08*, 2008.
[21] C. Raiciu, M. Handley, and A. Ford. Multipath TCP design decisions. Work in progress, `www.cs.ucl.ac.uk/staff/C.Raiciu/files/mtcp-design.pdf`, 2009.
[22] C. Raiciu, C. Pluntke, S. Barre, A. Greenhalgh, D. Wischik, and M. Handley. Data center networking with multipath TCP. In *Hotnets*, 2010.
[23] K. Rojviboonchai and H. Aida. An evaluation of multi-path transmission control protocol (M/TCP) with robust acknowledgement schemes. *IEICE Trans. Communications*, 2004.
[24] K. Tan, J. Song, Q. Zhang, and M. Sridharan. A Compound TCP approach for high-speed and long distance networks. In *Proc. IEEE INFOCOM 2006*, pages 1–12, April 2006.
[25] D. Thaler and C. Hopps. Multipath Issues in Unicast and Multicast Next-Hop Selection. RFC 2991 (Informational), Nov. 2000.
[26] B. Wang, W. Wei, J. Kurose, D. Towsley, K. R. Pattipati, Z. Guo, and Z. Peng. Application-layer multipath data transfer via TCP: schemes and performance tradeoffs. *Performance Evaluation*, 64(9–12), 2007.
[27] M. Zhang, J. Lai, A. Krishnamurthy, L. Peterson, and R. Wang. A transport layer approach for improving end-to-end performance and robustness using redundant paths. In *Proc USENIX '04*, 2004.

Appendix

We now prove that the equilibrium window sizes of MPTCP satisfy the fairness goals in §2.5. The rough intuition is that if we use SEMICOUPLED from §2.4, and additionally ensure (4), then the set of bottlenecked paths increases as a increases. The proof involves identifying the order in which paths become bottlenecked, to permit an analysis similar to §2.5.

First define

$$i(S) = \frac{\max_{r\in S} \sqrt{\hat{w}_r}/\mathsf{RTT}_r}{\sum_{r\in S} \hat{w}_r/\mathsf{RTT}_r}$$

and assume for convenience that the window sizes are kept in the order

$$\frac{\sqrt{\hat{w}_1}}{\mathsf{RTT}_1} \le \frac{\sqrt{\hat{w}_2}}{\mathsf{RTT}_2} \le \cdots \frac{\sqrt{\hat{w}_n}}{\mathsf{RTT}_n}.$$

Note that with this ordering, the equilibrium window increase (1) reduces to

$$\min_{S\subseteq R: r\in S} \frac{\hat{w}_{\max(S)}/\mathsf{RTT}^2_{\max(S)}}{\left(\sum_{s\in S} w_s/\mathsf{RTT}_s\right)^2} = \min_{r\le u\le n} \frac{\hat{w}_u/\mathsf{RTT}_u^2}{\left(\sum_{t\le u} w_t/\mathsf{RTT}_t\right)^2}$$

i.e. it can be computed with a linear search not a combinatorial search.

At equilibrium, assuming drop probabilities are small so $1 - p_r \approx 1$, the window sizes satisfy the balance equations

$$\min_{S: r\in S} i(S)^2 = p_r \hat{w}_r/2 \quad \text{for each } r \in R.$$

Rearranging this, and writing it in terms of $\hat{w}_r = \sqrt{2/p_r}$,

$$\hat{w}_r^{\mathsf{TCP}} = \sqrt{\hat{w}_r} \max_{S: r\in S} 1/i(S). \tag{7}$$

Now take any $T \subseteq R$. Rearranging the definition of $i(T)$, and applying some simple algebra, and substituting in (7),

$$\sum_{r\in T} \frac{\hat{w}_r}{\mathsf{RTT}_r} = \max_{r\in T} \frac{1}{\mathsf{RTT}_r} \sqrt{\hat{w}_r}/i(T) \le \max_{r\in T} \frac{1}{\mathsf{RTT}_r} \sqrt{\hat{w}_r} \max_{S: r\in S} 1/i(S) = \max_{r\in T} \frac{\hat{w}_r^{\mathsf{TCP}}}{\mathsf{RTT}_r}.$$

Since T was arbitrary, this proves we satisfy (4).

To prove (3), applying (7) at $r = n$ in conjunction with the ordering on window sizes, we get

$$\frac{\hat{w}_n^{\mathsf{TCP}}}{\mathsf{RTT}_n} = \sum_r \frac{\hat{w}_r}{\mathsf{RTT}_r}.$$

One can also show that for all r, $\hat{w}_r^{\mathsf{TCP}}/\mathsf{RTT}_r \le \hat{w}_n^{\mathsf{TCP}}/\mathsf{RTT}_n$; the proof is by induction on r starting at $r = n$, and is omitted. These two facts imply (3).