Adaptive Data Transmission in the Cloud

Wenfei Wu *, Yizheng Chen *, Ramakrishnan Durairajan *, Dongchan Kim *, Ashok Anand † and Aditya Akella *
* UW-Madison, † Bell Labs, India

***Abstract*—Data centers provide resources for a broad range of services, such as web search, email, web sites, etc., each with different delay requirements. For example, web search should cater to users' requests quickly, while data backup has no special requirement on completion time. Different applications also introduce flows with very different properties (e.g., size and duration).**

The default method of transport in data centers, namely TCP, treats flows equally, forcing equal share of the bottleneck network bandwidth. This fairness property leads to poor outcomes for time-sensitive applications. A better solution is to allocate more bandwidth to time-sensitive applications. However, the state-of-the-art approaches that do this all require forklift changes to data center networking gear. In some cases, substantial changes need to be made to end-system stacks and applications as well.

In this paper, we argue that a simple modification to TCP can help better meet the requirements of latency-sensitive applications in the data center. No modification to end-systems, applications or networking gear is necessary. We motivate our Adaptive TCP (ATCP) design using measurements of real data center traffic. We analytically derive the parameters to use in our proposed modification to TCP. Finally, we use extensive simulations in NS2 to show the benefits of ATCP.

I. Introduction

Data centers serve as an infrastruture for providing essential resources to host a broad range of applications from web search, email, advertisement to data mining of user behavior and system log analysis [1]–[4]. However, different applications may come with different delay sensitivity requirements. While some background jobs such as backups do not necessitate completion in a timely fashion, online services usually impose stringent latency goals on response time in their service level agreements (SLAs) [5]. Many studies have shown that network transfer plays a key role in determining the completion times of these jobs [6]. For example, data shuffle in the reduce phase of Map-Reduce jobs is a well known bottleneck for the whole job. Due to the high operational cost, data center resources are shared and multiplexed by these different jobs. Thus, how network resources are shared has a crucial impact on the job performance and the ability to meet various latency requirements.

Existing proposals for sharing the network fall into three categories but each work has its own Achilles' heel. The first solution is to simply partition the cluster into two parts, and run delay sensitive jobs on one with dedicated network resource. However, this static partitioning scheme makes it impossible to do fine-grained resource sharing across clusters and thus is not efficient [7]. Another approach is to enforce network allocation by adding complex components into the end-system software and hypervisors. One example is SeaWall [8], which adds a bandwidth allocator between TCP/IP and the NIC. Indeed, this solution could divide network capacity based on the desired policy, but it comes with the expense of intricate modifications to end-system architecture (e.g., policer flow rate control mechanisms, schemes to ensure distributed convergence to pre-assigned weights etc.). More importantly, it also engenders key changes to the service model exposed to tenants, and therefore is not always feasible. Yet another way for sharing the network is to make explicit scheduling of network traffic. An example is Orchestra [6] which coordinates different data transfers with a global controller. Again, because jobs need to express their needs explicitly to the central controller, this method requires changes to the traditional service model. Depending on the transfer scheduler used, with the default being simple FIFO, this approach may result in poor utilization of the data center as a whole, as some of the afore-mentioned naive solutions.

The position we take in this paper is that an ideal usable and effective network sharing scheme that effectively helps meet application delay requirements: (1) discriminates network flows according to their job timing requirements; (2) share cluster resource very efficiently; (3) makes minimal changes to the end system software; changes should be simple to implement so that it is easy to reason about overall system behavior and performance; (4) does not modify tenant applications, in particular, the current service model the cloud exposes.

In this paper, we make two major contributions. First, we conduct a first-of-a-kind measurement study of the relationship between flow sizes and timing requirements using real data center traces. These measurements inform key aspects of the eventual design of our system; but, they are also interesting in their own right as they can influence other aspects of data center design that we don't focus on (e.g., traffic engineering). Our analysis shows that time-sensitive jobs usually come with small flows smaller than 10 MB, while the flow sizes of other jobs falls in the range that is larger than 10MB. Also, small flows often exist concurrently and share links with those large flows.

These observations motivate our Adaptive TCP (ATCP) design to meet the above four requirements, which forms the second contribution of this paper. We propose *Adaptive Transmission Control Protocol* (ATCP), a simple approach for network sharing. In this protocol, we solve three problems: how to precisely control flows' rate when they are contending, how much bandwidth to be allocated for various flows, and how to make the allocation to be flow agnostic. The basic idea is to modify the congestion control behavior in TCP and

perform *adaptive weighted fairness sharing among flows*. As is known, TCP allocates bandwidth equally among all flows and does not take job latency requirements into consideration. In order to distinguish flows with different timing targets, we count how many bytes a flow has delivered already. We dynamically tune a flow's weight such that it decreases as a flow transfers more data. In effect, we can prioritize small flows' bandwidth allocation and get them to complete faster than the larger flows that they are contending with. Our key insight is that only the additive increase behavior of TCP congestion control needs to be modified to realized the above form of weighted sharing. Therefore, our method makes as little a change as possible to the cloud infrastructure.

We introduce a weight-size function to derive a flow's weight according to the size of the data it has sent. The parameters in the weight-size function are the weight upper bound W_H, lower bound W_L and threshold T. We set up T by observing the empirical flow size distribution. We analyze different combinations of W_H and W_L and chose ones which produce the smallest value for the median completion time over real data center traces.

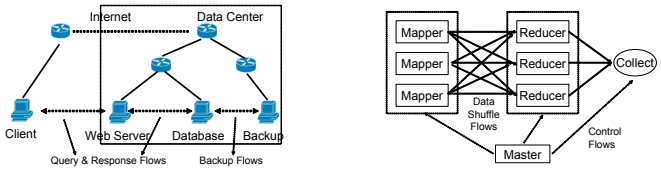
Based on extensive simulations using NS2, we find that ATCP benefits small flows significantly. Thus, delay-sensitive applications see the greatest improvements. We conduct trace-driven simulations in a chain topology and find that compared with TCP, more than 90% of flows benefited from ATCP and reduced their completion times. Small flows' ($<$ 100KB) average completion time is reduced by 10%; medium flows' (between 100KB and 10MB) completion time is reduced by 30%-40% on average; large flows' ($>$ 10MB) completion time is almost not influenced. We simulate a distributed application flow trace in the fattree topology and show that the benefit introduced by ATCP to small flows is comparable to DCTCP. Finally, we perform a simulation of MapReduce jobs. The result shows that by improving small flows completion time, the whole job's performance improves.

This paper is organized as follows. Section II provides motivating examples for the problem. In section III, we analyze the flow characteristics and flow relationships. In section IV, we propose the requirements to design an adaptive transmission control protocol in the cloud. In section V, we build the theoretical basis of flow rate control and scheduling, and design ATCP. In section VI, we describe our implementation. In section VII, we evaluate our ATCP and compare it with TCP and DCTCP [9] in various scenarios. We discuss related work in section VIII. Finally, we conclude in section IX.

II. Motivating Example

To motivate our changes on TCP, we describe below two kinds of applications: web services and a MapReduce distributed system, which are typical applications in current enterprise and university data centers. We measure their performance and should how our small changes improve them.

Web Services: A typical three-tier web service is shown in Figure 1(a). The query and response flows are usually small flows and they are delay-sensitive; the backup flows are usually



(a) Web Service (b) MapReduce

Fig. 1: Various Flows in Data Center Applications

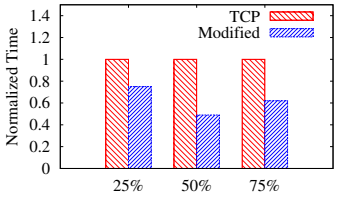
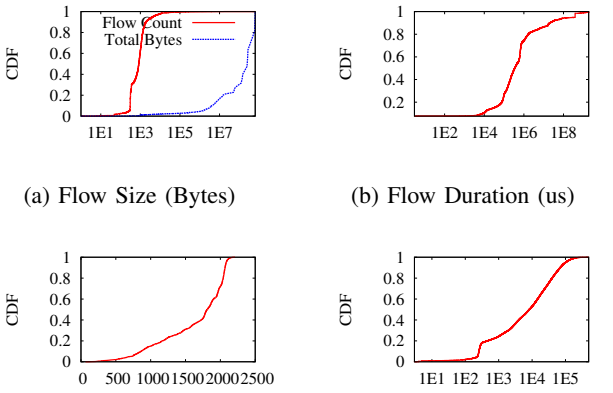


Fig. 2: Web Service Flow Completion Time

pretty large. We collected packet trace continuously for 12 hours over multiple days in a campus data center (serving the students and staff of a large US University). Inside the data center, a variety of services are running simultaneously, ranging from archival to distributed file systems, E-mail, web services (administrative sites and web portals for students and faculty), and even multicast video streams. Web service traffic and distributed file system traffic constitute most of the traffic, 60% and 40%, respectively.

We modify TCP so that when a small flow ($<$10MB) meet with a large ($>$10MB) flow, it get 3 times more bandwidth than the large flow (We use our mechanism in ATCP, which is described in Section V). We use NS2 to simulate the above trace and compare TCP with the above modification in terms of flow completion times (Figure 2). Compared with TCP, the modified TCP reduces median completion time by more than half, from 200ms to 80ms; and more than 90% of web flows benefit from this change. Only some large flows are influenced, but their completion time is increased negligibly (by less than 1%). Then We look into the web service traffic, which are time-sensitive. More than 90% of the web flows are smaller than 10MB and they benefit from this modification.

MapReduce: A typical MapReduce workload (Figure 1(b)) first distributes raw data blocks to several mappers, following which each mapper does some computation over the data. Subsequently, there is a shuffle phase in which results from mappers are sent to reducers in a many-to-many mode. After each reducer collects all corresponding results, there may be a final collector to fetch the result. Each phase is composed of parallel network transfers, especially the shuffle phase, which takes 33% of total job completion time in typical computation jobs in the cloud [6]. A key issue in MapReduce is that, if one of the transfers in the shuffle phase is delayed ("straggler"), the entire job is affected. MapReduce flows are usually mixed up with other background data flows. If they



(a) Flow Size (Bytes) (b) Flow Duration (us)

(c) Active Flows (d) Flow Interarrival Time(us)

Fig. 3: Data Center Flow Distributions

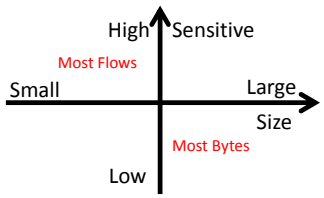


Fig. 4: Flow Size and Time Sensitivity

can grab bandwidth from such (delay-insensitive) background flows, the data transfer time is reduced, thus improving the MapReduce job performance.

The examples show that if small flows get some "help" when competing with large flows at a bottleneck link, they can complete more quickly and improve overall job performance. From applications like web services, most flows are small (less than 10MB) according to our measurements; while for distributed computations like MapReduce, a given job can be split into smaller components, and fine-grained tasks are easier to schedule and can benefit from our new transport protocol. In this way our new transport protocol can improve most delay-sensitive applications.

III. Flow Charateristics

To design Adaptive TCP, we leverage empirical observations of flow characteristics in a real-world data center. Unsurprisingly, we find flow distribution characteristics such as size and duration to be similar to measurements reported in [10], [11] and [12], but we repeat them here to provide a basis for the design choices in ATCP. A key difference is that we analyze the temporal relationship between flows such as overlap and arrival time interval. The observation results imply that small flows can take bandwidth away from large flows, and large flows will get the necessary compensation only after the completion of small flows.

The data center where we analyze traces has a canonical 2-Tier architecture, in which Middle-of-Rack switches are used to connect a row of 5 to 6 racks. Middle-of-Rack switches are connected by aggregation switches with an over-subscription factor of 2. In total, there are 500 servers and 22 network devices. To get the packet trace, we randomly selected a handful of locations and installed sniffers. Our collection spanned 12 hours over multiple days. According to our investigation and measurement (using the Bro application identification tool), the applications inside the data center are mainly web services (HTTP transactions, authentication services, custom applications) and distributed file system traffic.

First we examine the distributions of **flow size**. Flow size distribution in Figure III(a) indicates that 80% of the flows are smaller than 100KB in size and most of them are RPC requests and responses. Especially 99% of the flows are smaller than 10MB, and the flows between 100KB and 10MB are mainly web requests and response. The total bytes distribution shows most of the bytes are from the few large flows; especially over 80% of the bytes are sent by flows of size larger than 10MB. These flows are rare in amount; they are mainly introduced by activities like backup, virtual machine migration or large file transfers. We use **small**, **medium** and **large** to denote a flow of size [0, 100KB], [100KB, 10MB] and [10MB, ∞) respectively and use these terms in the following text.

By our measurement, we find that most time-sensitive applications like web service have smaller size, while the large flows are usually not time-sensitive, as is shown in Figure 4.

From Figure III(b) about the **flow duration**, we find that in our data center, 80% of the flows are less than 10 seconds long, but there are still some flows that last for more than 100s. Most of the long-duration flows are large flows (larger than 1GB). Although the link capacity is 1Gbps, the flow's sending rate is constrained by contention from other flows.

In Figure III(c), we present the distribution of the number of **active flows** within a one second bin at one edge switch. In over 90% of the time instances, the number of active flows per edge switches is between 1000 and 2000, and these flows are not uniformly distributed on each link. On some "hot spot" links there are tens of flows, and the competition between them causes high utilization of the corresponding link.

Flow **interarrival time** is presented in Figure III(d). We observe that 80% of the flow's interval times were between 400us and 40ms. Given that 20% of the long-duration flows last longer than 10s, large flows will coexist with many small flows.

We also look into the **temporal relationship** between flows. Most switches maintain queues at in ports and out ports and all these queues typically share the same memory; so the contention between flows is not only restricted on links, but also on switch buffers. We separate large flows from small and medium flows, and calculate the percentage of time during which large flow exists over the total measurement time, which is over 95%. Then we look into each small or medium flow to see whether it is overlapping with one or more large flows; we find that over 90% of them flows have durations overlap with one or more large flows.

The above measurement and analysis has three implications. First, in a data center a variety of applications introduce flows of different properties. Most flows are mice flows, they are small in size, but are majority in quantity. Elephant flows are small in quantity, but they contribute the majority the total bytes. Second, as applications fill in the data center capacity, flows end up competing which results in bottlenecked links and switch buffers; most of these resources (link capacity and buffers) are taken by large flows. Third, most small flows and large flows coexist and compete with each other in the network; with large flows taking most of the network resources, small flows suffer. If small flows "borrow" some bandwidth from large flows, they would complete more quickly; while large flows can get time compensation after the small flows complete. The improvements to small flows can enhance a variety of different applications.

IV. Requirements

From the above examples and trace analysis, we identify the requirements of Adaptive TCP. Requirements such as **high network utilization, low latency and scalability** are common in network design. For our special motivation of reducing average completion time and ease to deploy, an ideal solution should also have the following properties.

Shorter average completion time: Small flows should complete more quickly, while large flows should be not influenced. According to our measurement, most time-intensive applications such as web services and distributed computations are transferring data under a certain size. If we can reduce the completion time of small flows, we will significantly improve application performance.

Flow agnostic: Operators do not need to know whether a flow is small or large. The rate allocation should be adaptive to the flow size automatically. One existing approach to distinguish small flow from large flow is to judge them by IP, port number in some historical records and deploy QoS accordingly. But we argue that this is not a reasonable way. With the rapid increase of applications, configurations for flows will become a large burden for the operator; in addition, it is hard to distinguish large flows and small flows in the same application such as FTP. Other approaches to adaptively allocate bandwidth to flows, such as D^3 [5] and D2TCP [13], are not flow agnostic.

No changes to network devices: Recently router-based flow rate control protocols such as D^3 [5] are proposed. In these protocols, routers need to perform some computations to allocate bandwidth to each flow. With so many commodity switches being deployed in data centers already, we believe an edge- and software- based solution is more reasonable [14], [15].

V. Adaptive TCP

In this section, we theoretically prove that assigning weight to the TCP additive increase lead to precise flow rate control a flow's rate and preferring small flows to large flows benefits small flows without influence on large ones. Then we design Adaptive TCP (ATCP) which make size-adaptive bandwidth allocation automatic.

A. Flow Rate Control

In networks, each intermediate router maintains virtual output queues at each input port and an output queue at each output port; these queues share the switch memory [16]. Packets that arrive are served in first-in-first-out (FIFO) order in switches, and they are dropped when the switch buffer overflows which accounts for packet loss [17], [18]. A TCP flow begins with the slow start phase, during which its congestion window increases by 1 segment after each ACK. Duplicate ACKs (usually 3) indicate packet loss in the network, leading the congestion window to be halved. After the first loss, TCP gets into the congestion avoidance phase where the congestion window is increased by 1 segment every round trip time (RTT) and still halved in the case of the next packet loss (3 duplicate ACKs). This scheme is called additive increase multiplicative decrease (AIMD). The size of the data that a sender can send is at most equal to the congestion window size; therefore the sending rate is the congestion window size divided by RTT. RTT does not change too much in a flow's duration, so the congestion window determines the sending rate. In TCP all flows follow the same AIMD scheme, thus they equally share the network and achieve max-min fairness.

In the AIMD, we define a **weight** to each flow. A flow with weight a increases its congestion window by a segment each RTT in the congestion avoidance phase. Then the contending flows' sending rate can be precisely controlled by the following theorem.

Theorem 1: In congestion avoidance phase, if two contending flows additive increase rate ratio is $a : b$ and the multiplicative decrease is the same (decreased by a half when congested), these two flows' sending rate ratio converge to $a : b$.

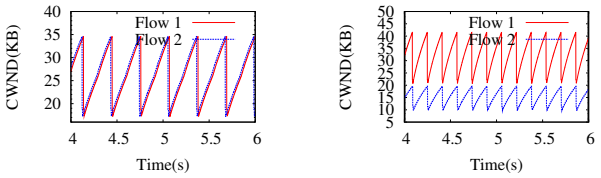
Assumptions: (1) There are only 2 flows competing with each other. (2) The flow durations are long enough for them to converge to the final allocation ratio. (3) When the network is congested, the intermediate router starts to drop packets belonging to both the flows. (4) Both flows have the same total delay and RTT on their paths.

Symbols and Terms: (1) T is the total delay on the links of the path and T' is the RTT, (2) B is the total size of all switch buffers in the network, (3) On-path links have the same bandwidth capacity C, (4) a is flow 1's weight, and b is flow 2's weight, (5) flow 1 and flow 2 converge to sending rate R_1 and R_2 finally.

Proof: The bandwidth-delay product is $C \times T$ and so the total byte capacity in the network is $C \times T + B$, which is denoted by

$$S = C \times T.$$

As two flows increase their own congestion window, the network experiences congestion. Assume the two flows' congestion windows are w_1 and w_1' when the network is congested for the 1st time, w_2 and w_2' for the 2nd time,..., w_i and w_i' the



(a) TCP (b) Weighted TCP

Fig. 5: Congestion Window Changes in the 2-flow Scenario

i-th time. Then we have

$$w_j + w'_j = S, \forall j.$$

When the network is congested, according to the mulitplicative decrease the two flows' congestion windows are halved into $0.5w_j$ and $0.5w'_j$ respectively. In the following additive increase, their windows increase in a ratio of $a : b$ until the remain $0.5S$ network capacity is divided. Then the next congestion occurs, at this time

$$w_{j+1} = 0.5 \times w_j + 0.5 \times S \times \frac{a}{a+b}$$

$$w'_{j+1} = 0.5 \times w'_j + 0.5 \times S \times \frac{b}{a+b}.$$

With this recursive equation,

$$\begin{aligned} w_{j+1} - \tfrac{a}{a+b}S &= 0.5(w_j - \tfrac{a}{a+b}S) \\ &= \ldots = 0.5^j(w_1 - \tfrac{a}{a+b}S) \end{aligned}$$

let $i \to \infty$, we have

$$w = lim_{i\to\infty} w_i = S \times \frac{a}{a+b}$$

$$w' = lim_{i\to\infty} w'_i = S \times \frac{b}{a+b}.$$

So sending rates of flow 1 and flow 2 are

$$R_1 = \frac{S}{T'} \times \frac{a}{a+b}, R_2 = \frac{S}{T'} \times \frac{b}{a+b}.$$

and

$$R_1 : R_2 = a : b.$$

Simulation: We simulate the 2-flow scenario. The network topology is a 3-node chain topology, the link capacity is 100Mbps and the delay on links is 50us. The two flows that have the same source and destination. We measure and plot the congestion window as a function of time in Figure 5.

In both TCP and weighted TCP, the 2 flows converge to a congestion avoidance state quickly. In TCP each of the 2 flows increases its congestion window by one segment size every RTT, and the window is halved in case of packet loss. In rate control mode, we set the weights of the 2 flows to be 1 and 2 respectively. In the final converged state, flow 1's congestion window increases twice as fast as flow 2's. When the network is congested, both windows are halved. In converged state, the ratio of both flows' window sizes equals the ratio of weight at any time.

The assumption (2) holds for most of the flows. When a new flow join the network, it contend with existing flows which probably take most of the link capacity. Assume the first packet drop happens at the rate of hundreds of Mbps and the RTT is hundreds of microseconds, then the congestion windows is in the order of tens of KB, the data that is already sent is also in this order. The assumption (4) does not always hold, which causes occasionally deviations from the theoretical result. But when the network is congested, and both flows are sending at least hundreds of packets per second, it is of high possibility that both flows' packets are dropped. The simulation (Figure 5) also shows that in most cases that the network is congested, both flows' congestion window are decreased by a half.

B. Small Flow First Scheduling

With weighted TCP, we can control flows' sending rates precisely. When multiple flows contend about the bandwidth, the bandwidth allocation actually is a job scheduling problem. We propose that, by small flow first scheduling, we can reduce the average completion time.

Theorem 2: If a large flow with duration $[s_1, t_1]$ and a small flow with duration $[s_2, t_2]$ share the same network bandwidth, and if $s_1 < s_2 < t_2 < t_1$, then by allocating more bandwidth to the small flow, the average completion time of the two flows reduces.

Symbols and Terms: (1) the large flow has size S_1 and the small S_2, (2) the shared link bandwidth on the path is C, (3) in TCP, the small flow sends at rate R_2; when assigning more bandwidth to the small flow, it has sending rate of R'_2, (4) the durations of the large flow and the small one are $[s'_1, t'_1]$ and $[s'_2, t'_2]$ when assigning more bandwidth to the small flow.

Proof: With the same trace in two cases, we have

$$s'_1 = s_1, s'_2 = s_2.$$

Assigning more bandwidth in the 2nd case, we have

$$R'_2 > R_2.$$

Then for the small flow

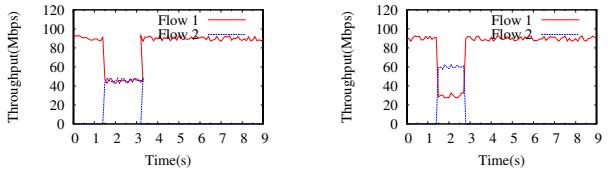
$$t'_2 - s'_2 = \frac{S_2}{R'_2} < \frac{S_2}{R_2} = t_2 - s_2,$$

completion time decreases. Consider the time when the last bytes of both flows is sent, we have

$$t'_1 - s'_1 = \frac{S_1 + S_2}{C} = t_1 - s_1$$

For the large flow, the completion time is the same. So the average completion time decreases.

By the measurement in Section III, we observe that most small flows start and finish in the duration of a large flow. Only a very small number of small flow partially overlap with a large flow. So we conclude that most small flows benefit in their completion time. If a large flow partially overlap with a small flow, the overlapping period is less than a small flow's duration, which is 2 orders of magnitude smaller than its original completion time, thus it is neglectable.



(a) TCP (b) Weight TCP

Fig. 6: Throughput in the 2-Flow Scenario

Simulation: We still simulate the 2-flow scenario, with the two flow of size 100MB and 10MB on the same path. We set the large flow with weight 1 and small flow with 2 in weighted TCP, and also simulate the same flows with TCP. The throughput of both flows is shown in Figure 6.

In TCP, the large flow starts first, then followed by the small flow. According to TCP's fairness, both sending rates finally converge to half of the link capacity. In weighted TCP, the expected bandwidth allocation ratio is 2:1, which is exactly shown in Figure 6(b), and the completion time also decreases from 1.8s to 1.2s.

C. ATCP Design

The design of ATCP is based on TCP rate control and flow scheduling in the previous sections. In ATCP, we first add a sent-data counter in the flow socket structure. It counts bytes as the flow sends data. Then we introduce a weight-size function, which takes the sent-data size as input and gives a weight as output. Finally, we change the additive increase in TCP by making the increased size proportional to the weight.

The weight is large at the beginning, and then decreases as sent-data size increases. So small flows' weight is relatively high in their duration; a large flow only sends the first few bytes with a high weight and the remaining bytes are sent with a low weight. When a small flow competes with a large flow, it is quite possible that the small flow has a higher weight than the large flow, so that small flow can get more bandwidth.

ATCP design satisfies the requirements in Section IV, ATCP uses AIMD scheme, so the network is still fully utilized. By setting small flows' a higher weight, ATCP guarantees that small flows get more bandwidth and complete more quickly. By counting sent bytes, ATCP does not need flow size information from the application layer; thus it is flow agnostic. All the changes are in the protocol stack, so ATCP avoids hardware device changes.

The **weight-size function** is the key of the adaptive rate control. ATCP start from the requirements and design the weight-size function:

- All flows achieve high network utilization. So the weight-size function is always positive.
- The more a flow sends, the less competitive it is. So the weight-size function is decreasing, but not necessary to be strict.
- Small flows are more or at least not less competitive than large flows. So in the duration of a small flow, its weight is no smaller than a large one. $W(s) = max(w)$, for $s < T_1$, where T_1 is the threshold of small flows. $W(s)$ is a constant when $s < T_1$ because if it is strictly monotone decreasing, a late-start large flow can have larger weight than an early-start small flow in their overlapping period, which degrades the small flow's throughput.
- ATCP should have neglectable influence on large flows. For several contending large flows, they should have the same behaviors with TCP, thus their weight should be the same. So $W(s) = c$, for $s > T_2$, where T_2 is a threshold which means the flow has sent sufficient volume of traffic. $W(s)$ is a constant when $s > T_2$, because if it is strictly monotone decreasing, the late-start large flows will dominate the early-started ones.

To satisfy the principles above, the weight-size function should have the following format

$$W = \begin{cases} W_H & \text{if } s \le T_1 \\ W'(s) & \text{if } T_1 < s \le T_2 \\ W_L & \text{if } s > T_2 \end{cases},$$

where the $W'(s)$ is a monotone decreasing function from W_H to W_L at the range $[T_1, T_2]$. There are multiple choices of the function $W'(s)$, such as exponentially decreasing, or linear decreasing. But considering the order of magnitude of small flows and large flows, we go on simplify the weight-size function to a two-segment constant function.

As we discussed in Section III, 80% bytes are sent by flows that are larger than 10MB, so we define the small flow threshold T_1 to be 10MB. We want the large flows behave like TCP, and only when $s > T_2$, the large flows has a fixed weight and max-min fairness among them. So T_2 should not be too large and the interval $[T_1, T_2]$ should only be a small portion of the whole flow size. We assume it to be 2 orders of magnitude smaller. Consider the typical large flow size are hundreds of megabytes, T_2 is in [1MB, 10MB]. Then T_2 is simplified to be equal to T_1. In our real trace simulation, even we set $T_1 \neq T_2$ and different $W'(s)$, we find very small portion of flows overlapping with large flows in $[T_1, T_2]$ and the format of $W'(s)$ really does not make too much difference. So the weight-size function is finally set to be:

$$W = \begin{cases} W_H & \text{if } s \le T \\ W_L & \text{otherwise} \end{cases}.$$

By this weight-size function, for all flows that are smaller than T, their data will be sent by the highest weight W_H. When competing with a large flow, if the large flow is sending in weight W_H, its performance in ATCP is no worse than in TCP; if the large flow is sending in weight W_L, the small flow will be more aggressive. And comparing the magnitude of T and data size of a large flow, the latter is the most common case in the network, thus most small flows benefit.

D. Discussions

ATCP does not lead to large flow starvation. ATCP allocates more bandwidth to small flows than large flows, thus small

flows completes more quickly and leave more time to large flows as compensation. One may argue that if small flows come one by one which makes the large flow contends with small flows throughout its life, it always gets lower bandwidth in ATCP than in TCP. We claim that this comparison is unfair, because in ATCP, small flows complete more quickly, and if they come one by one, there are actually more frequent small flows in ATCP. To make the comparison fair, we fix the flow trace with the same flow arrival time stamps and sizes, If the network sends data by the best effort and links are always close-to-fully-utilized, in a fixed period, the network sends a certain amount of bytes. In this amount, the total size of all small flows is fixed, and all the left bandwidth are used by large flows, which follows max-min fairness among them in both TCP and ATCP. So the large flows' completion time is not influenced. Simulation results also verify that large flows are not influenced.

There are many TCP variants now, such as Tahoe, Reno, new Reno, Cubic [19], etc., and the IETF TCPM working group [20] also develop various extensions of TCP to adjust to different scenarios. However, all these TCP variants follow AIMD, our mechanism can be used to modify them to be adaptive in the cloud.

ATCP does not achieve application-level fairness. If an application starts multiple connections to speed data transfer up, it gets more bandwidth than the application with less connections. However, this is not solved by TCP either. ATCP can work with other mechanisms that control fairness, such as fairness queuing or QoS. In this case, the flows in the same queue can still reduce their average completion time by ATCP.

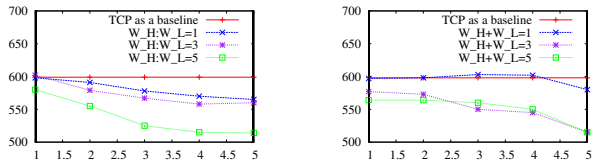
The weight-size function can be in other formats. For example, it can be refreshed periodically to adjust to some periodical bursty flows. Recently there are some flow deadline-aware designs such as D^3 [5] and D2TCP [13]. We can also achieve this by changing the weight-size function to be weight-time function. For example, we let the weight function increase with time first, and after it misses the deadline, the weight is decreased to a small value.

A flow's data transfer time includes propagation delay, queuing delay and transmission time. Propagation delay equals [sum of links' length] over [light speed in the links], queuing delay is the sum of [queue length in each switch] over [link bandwidth], and transmission time equals [flow size] over [sending rate]. ATCP actually decreases the transmission time by assigning a larger sending rate. In the data center, if the data size is too small, the queuing delay and propagation delay dominate the total transfer time, the improvement is limited. While if the transmission time dominates the total time, the improvement is more significant. In this case the flow size is usually 100KB-10MB according to our simulation.

VI. Implementation

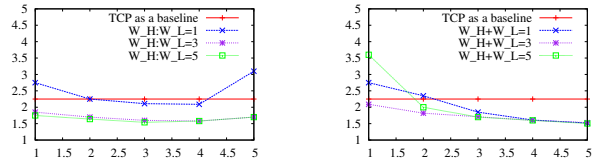
We implement ATCP in NS2 to estimate parameters and evaluate the trace from data center and other benchmarks. In NS2's socket data structure, we add the parameter W_H, W_L and T as members, and we also add the counter to the socket to record sent-data size which is set to be 0 initially. When the socket sends data, the size of sent data is added to the counter. In congestion avoidance state, when the congestion window CWND is increased by some value $adder$, we increase it by $adder \times weight()$, where $weight()$ is computed by the parameters and counter. We add APIs to set the parameters in the socket data structure.

The changed code is less than 100 lines, but it improves TCP's performance significantly.



(a) Time(ms) - $[W_H + W_L]$ (b) Time(ms) - $[W_H : W_L]$

Fig. 7: Median Completion Time of Small Flows



(a) Time(s) - $[W_H + W_L]$ (b) Time(s) - $[W_H : W_L]$

Fig. 8: Median Completion Time of Medium Flows

VII. Evaluation

In this section, we first use real trace simulation to discuss how to set the parameters W_H and W_L, and then evaluate ATCP performance in various scenarios such as different topologies and traces. Finally, we compare application performance in TCP, DCTCP [9] and ATCP.

A. Parameter Setting

In weight-size function, the parameters are W_H, W_L and threshold T. We set up the T by observing the flow size distribution in Figure III. The gap between small and medium (<10MB) flows and large (>10MB) flows is very obvious, so we set the parameter T to be 10MB.

We build a 4-hop chain topology with 100Mbps capacity and 50us latency on each link. We use the measured trace and try combinations of $W_H + W_L$ (fixed-sum mode) and $W_H : W_L$ (proportional mode). We simulated with $W_H + W_L$ and $W_H : W_L$ ranging from 1 to 5. We measure the flow completion time. The results are shown in Figure 8 and Figure 7, which are the median of completion times for medium flows and small flows.

Compared with small flows, medium flows have more significant improvement. In the best case, with $W_H = 3$ and $W_L = 1$, median completion time for medium flow is reduce by 40% from 2.3s to 1.4s. In all cases with $W_H : W_L > 2$,

median completion time is reduced by 20% to 40%. While small flows' median completion time is reduced by at most 15%. In most cases, the reduction is about 10%. Since the flow completion time is composed by propagation delay, queuing delay and transmission time. Small flows' propagation delay dominates the total transfer time, so allocating more bandwidth only helps a bit; but medium flows' transmitting time(size/rate) dominates the total transfer time, so allocating more bandwidth improves medium flow's performance more.

$\mathbf{W_H : W_L}$ plays a key role in bandwidth allocation. From figure 8(b) the larger the ratio is, the smaller the completion time is. The completion time reduces rapidly as $W_H : W_L$ varies from 1 to 4, then trend slows down. This can be explained by the 2-flow example, suppose the weight ratio is R, the link capacity is C and the medium flow size is S, then the bandwidth allocated to medium flow is

$$C \times \frac{R}{R+1},$$

so the completion time is

$$\frac{S}{C} \times (1 + \frac{1}{R}),$$

whose curve is decreasing rapidly first, then slows down.

$\mathbf{W_H + W_L}$ has influence on the oscillation of total throughput. In the 2-flow example, $(W_H + W_L) \times SegmentSize$ is the increment of sum of all congestion windows in each RTT. With a large $W_H + W_L$, the total congestion window exceeds the networks capacity quickly, and packet loss happens frequently, which leads to throughput oscillation. So as $W_H + W_L$ increases from 1 to 5, the completion time decreases first, then increases, which matches Figure 8(a).

B. ATCP Performance

We first simulate ATCP on a **chain topology** to evaluate its effectiveness and robustness in various situations. We still use the flow traces from the measurement and 4-hop chain topology. In all these evaluations, we set $W_H = 3$ and $W_L = 1$.

We introduce flow **deadlines** from D_3 [5]. Applications in data center usually have deadlines, which constrained by the service's acceptive response. Only flows that complete in their deadline is meaningful [5]. For example, in some distributed computations such as web search, the sub task that cannot complete before a certain deadline is abandoned. We introduce deadline for our small flows, we take the deadline of 30ms from [5]. We look into the result get in previous section with $W_H = 3$ and $W_L = 1$, the results is that with ATCP, 84.6% small flows meet with their deadlines, while only 77% small flows meeting with their deadline in TCP.

ATCP is robust in case of **dense flows**. We increase the density of flows in three ways, dense large flows, dense non-large flows, and dense all flows, the way we increase flow density is to double the corresponding (large, non-large, and all) flow number in a fixed time. As we expected, with the increase in completion time, which is measured in Table I

TABLE I: Completion Time - Flow Density

Setting	Median Completion Time (Small/Medium Flows)	
	TCP	ATCP
Non-dense flows	625ms/2.3s	543ms/1.4s
dense non-large flows	763ms/2.5s	680ms/1.6s
dense large flows	750ms/2.8s	680ms/1.8s
dense all flows	813ms/2.9s	707ms/1.8s

TABLE II: Completion Time with Disjoint Path

Setting	Median Completion Time (Small/Medium Flows)	
	TCP	ATCP
Large flows on the chain	562ms/2.0s	489ms/1.3s
Small flows on the chain	688ms/2.5s	598ms/1.6s

the links become more congested, so the throughput for each flow decreases. In all cases, ATCP reduces completion time compared with TCP, small flows' median completion time get 10%-13% reduction at the median and medium flows is about 30%-40%. In dense non-large flow case, the improvement is a bit better, which is 39% than that in dense large flow case in which the reduction is 36% for medium flow; because in dense non-large flow case, non-large flows take more bandwidth from the large flows.

Flows always have **disjoint paths**, and only some links on the path may be shared. We simulate two scenarios where large flows and small flows partially share their path. We use a 5-node chain topology. In the first case, large flows transfer data through the whole chain, and small and medium flows takes one of the links from the chain. In the second case, each large flow takes one link respectively, while small and medium flow pass the whole chain. we still use trace from the measurement. The completion time at both cases with different parameters are shown in Table II.

In both cases, ATCP works better. Even when flows share different links, non-large flows still get more bandwidth when they compete with large flows. In the case that large flows take the whole chain, the performance is a bit better than that in the case that non-large flows take the whole chain. If the large flow takes the whole chain, as long as one of the link on its path is congested, the long flow will decrease the sending rate at the whole chain, which will give more opportunity for non-large flow on other links.

We also simulate a **web service** application in a **tree**

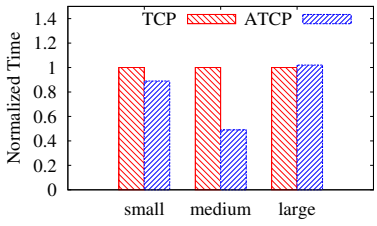
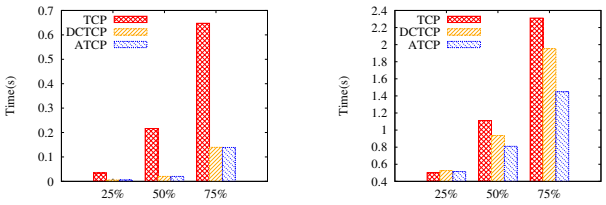


Fig. 9: Median Web Service Completion Time

TABLE III: Web Service in a Tree

Completion Time	TCP	ATCP
median of small flows	102ms	80ms
median of medium flows	1.7s	1.2s



(a) Small Flows (b) Medium Flows

Fig. 10: Completion Time in Fattree

topology like Figure II(a), we choose a one node as a web server and one node as a database. The database server backups data by transmitting data to another server, which is a large flow. We also add some other background flows, with the distribution of Figure III. Then we simulate some non-large flows from between web server and a client and between web server and database. Results in Table III show that there are about 5%-10% improvement on completion time for small flows, about 50% for medium flows, and almost no influences on large flows.

C. ATCP v.s. DCTCP

We compare ATCP with DCTCP. We choose DCTCP because both of them are flow agnostic, and we do not need deadlines from the applications.

We perform our simulations on a network with **fattree** [21] topology. We maintain the full bisection bandwidth in a tree topology to simulate the rich paths in fattree [21] topology. There are 4 racks with each rack having upto 10 machines. Each of these machines connects to the top-of-rack (ToR) switch via 1 Gbps link. ToR switches are connected to a core router via 10 Gbps link. One server works as an aggregator in a distributed application, and all other servers send small responses of size in [1KB, 10MB] to the aggregator. Then we inject a background flow for the core switch to the aggregator. We compare TCP, ATCP and DCTCP in terms of the flows' median completion time.

In Figure 10, small flows' median completion time is 210ms, 20ms and 21ms in TCP, DCTCP and ATCP respectively; and medium flows'median is 1.1s, 0.93s and 0.81s. Both ATCP and DCTCP dominate TCP; ATCP allocates more bandwidth to small flows to reduce their transmission time; DCTCP maintains smaller switch queue length to reduce their queuing delay. In DCTCP and ATCP, the small flows completion times are near each other. But medium flows has better performance in ATCP than DCTCP. Because DCTCP reduce the queuing time in the network, and this saving has an upper bound (queue length over bandwidth); but ATCP reduce the transmission time (data size over sending rate), so that the larger the data size is, the more a flow benefits.

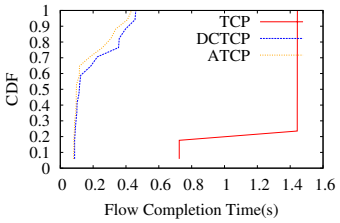


Fig. 11: MapReduce Shuffle Flows' Completion Time CDF

Referring to the **deadline** discussion in [5], we set a deadline of 30ms for the flows smaller than 100KB. In our simulation, the percentage of flows whose deadlines are satisfied are 23%, 62% and 61% in TCP, DCTCP and ATCP respectively. ATCP improves TCP a lot and is comparable with DCTCP. In our simulation, we use rather dense flows compared with D^3 [5], and 38% more flows are satisfied. This portion is larger than the result in D^3. We believe ATCP is comparable with D^3 in terms of the service deadline.

Finally, we look into an application's performance. We take **MapReduce** as an example. According to [22], the straggler in the shuffle phase influences the application's whole performance. With the same topology and background flow setting, we deploy TCP, ATCP and DCTCP for a MapReduce simulation. We collect the completion times of all shuffle flows. Figure 11 displays the completion time CDF of a hadoop job's shuffle flows in different protocols. The ATCP curve is the leftmost, thus is the most efficient protocol for shuffle flows. By running MapReduce several times, ATCP and DCTCP performs better than TCP in terms of average completion time. ATCP reduce the average completion time by 61%, and DCTCP by 59%. ATCP is even better than DCTCP, because in our MapReduce simulation, we simulate a large data sorting application, and set the data block size to be 8MB. In this size, ATCP flows benefit more than DCTCP.

The latest completion time determines the application's completion time. ATCP reduces the data shuffle time by 33% and total time by 10.5%; DCTCP reduce these them by 25% and 7%. ATCP's small-flow-preferred mechanism improves distributed application's performance in the end. This MapReduce simulation also implies that the MapReduce application can be configured to suits the new transport layer protocol.

VIII. Related Work

There are vast literatures on TCP congestion control. However, our idea that ATCP combines rate control, flow scheduling and cloud adaptiveness is more or less different from existing works.

MulTCP [23] and AIMD(a,b) TCP [24] all proposes that by changing additive increase rate the TCP flow's throughput and loss ratio changes. But they only study single flow's throughput, and observe multiple flows' performance by simulation. In ATCP, we provide a theoretical proof about the precise bandwidth allocation when flows contending.

Rai et al. propose size-based flow scheduling algorithms like Shortest Job First(SJF) , Shortest Remaining Processing Time (SRPT) and Least Attained Service (LAS) in [25], They use simulation to show that these scheduling algorithms reduce job completion time. But they do not mention how to control flow rate and their solution is not flow agnostic.

Gorinsky et al. provide the theoretical proof [26] that their Shortest Fair Sojourn (SFS), Optimistic Fair Sojourn Protocol (OFSP) and Shortest Fair Sojourn (SFS) scheduling policies are fair without starving any flows. We use their proof techniques to proof our small flow preferred scheduling lead to smaller average completion time.

DCTCP [9] uses ECN to notify the end-host of congestion in the network, which the end-host then uses to modulate its congestion window. DCTCP also reduces the completion time by reducing router's queue length. However, it does not explicitly help short flows like our approach does.

In D^3 [5], the endhosts encode deadline requirements within packet headers using which the intermediate router computes flows' bandwidth. The scheduling algorithm tries to satisfy as many flows' deadlines as possible. However, this approach changes router hardware as well as applications. ATCP only makes small changes to endhosts. The trade-off is that ATCP cannot provide explicit deadline guarantees; but as our results show, it can improve the performance a significant fraction of short flows compared to status quo.

D2TCP [13] uses both ECN flag and deadline knowledge to adjust the congestion window, so that D2TCP can solve both bursty fan-in problem and assign larger bandwidth to the flows near deadlines. However, D2TCP still takes applications' deadlines as input to compute congestion window changes, which is not flow agnostic.

Seawall [8] uses weighted TCP to control sending rate, but the authors introduce a different granularity for flow control (VMs, or entity). Seawall requires tremendous changes to host software. Also, it is non-trivial to modify applications to provide weights. We note that ATCP can be complimentary to Seawall. It can be viewed as a way to do dynamic bandwidth allocation among flows corresponding to the same network entity.

QoS is another way to allocate bandwidth to flows; it maintains priority queues in the switches. However, typical approaches need operator involvement and configuration for each flow. And there are not sufficient amount of queues for various applications.

IX. Conclusion

In this paper, we propose a new variant of TCP for clouds named Adaptive TCP. Our work is motivated by the fact that TCP's fairness does not differentiate among delay-sensitivity of applications, while existing solutions require draconian changes to applications, infrastructures and the service models. Based on measurements that delay-sensitive applications typically use short flows, and that the flows often co-exist with large flows, our scheme is designed to "steal" bandwidth from large flows over time and reallocate to small ones, and to "compensate" large flows by more transfer time. We achieve this via simple adjustment of TCP's additive increase parameter as a function of data sent, requiring minimal changes to the cloud software infrastruce, leaving applications and hardware unmodified. Simulations based on real data center traces show that ATCP reduces small flow completion time significantly and does not influence large flows. As a result the performance of time-sensitive applications is improved.

References

[1] D. Beaver, S. Kumar, and et al., "Finding a needle in haystack: Facebook's photo storage." *OSDI*, 2010.

[2] J. Dean, S. Ghemawat, and et al., "Mapreduce: Simplified data processing on large clusters." *USENIX OSDI*, 2010.

[3] G. DeCandia, D. Hastorun, and et al., "Dynamo: amazon's highly available key-value store." *SIGOPS*, 2007.

[4] M. Isard, M. Budiu, and et al., "Dryad: Distributed data-parallel programs from sequentical building blocks." *EuroSys*, 2007.

[5] C. Wilson and H. Ballani, "Better never than late: Meeting deadlines in datacenter networks," *SIGCOMM*, 2011.

[6] M. Chowdhury, M. Zaharia, and et al., "Managing data transfers in computer clusters with orchestra," *SIGCOMM*, 2011.

[7] B. Hindman, A. Konwinski, A. G. Matei Zaharia, and et al., "Mesos: A platform for fine-grained resource sharing in the data center," *NSDI*, 2011.

[8] A. Shieh and S. Kandula, "Sharing the data center network," *SIGCOMM*, 2011.

[9] M. Alizadeh and A. Greenberg, "Data center tcp (dctcp)," *SIGCOMM*, 2010.

[10] S. Kandula, S. Sengupta, A. Greenberg, P. Patel, and R. Chaiken, "The nature of datacenter traffic: Measurements and analysis," *IMC*, 2009.

[11] A. Greenberg, J. R. Hamilton, and et al., "Vl2: A scalable and flexible data center network," *SIGCOMM*, 2009.

[12] T. Benson, A. Akella, and D. A. Maltz, "Network traffic characteristics of data centers in the wild," *IMC*, 2010.

[13] B. Vamanan, J. Hasan, and T. Vijaykumar, "Deadline-aware datacenter tcp (d2tcp)," *SIGCOMM*, 2012.

[14] N. Dukkipati, "Rcp: Congestion control to make flows complete quickly," *PhD thesis, Standford University*, 2006.

[15] D. Katabi, M. Handley, and C. Rohrs, "Congestion control for high bandwidth-delay product networks," *SIGCOMM*, 2002.

[16] N. McKeown, "A fast switched backplane for a gigabit switched router," *White Paper*.

[17] S. Floyd and V. Jacobson, "Random early detection gateways for congestion avoidance," *IEEE/ACM Transations on Networking*, 1994.

[18] Y. Gu, D. Towsley, C. Hollot, and H. Zhang, "Congestion control for small buffer high bandwidth networks," *INFOCOM*, 2007.

[19] I. R. S. Ha and L. Xu, "Cubic: A new tcp-friendly high-speed tcp variant," *SIGOPS-OSR*, 2008.

[20] "www.ietf.org/wg/tcpm/."

[21] R. N. Mysore, A. Pamboris, N. Farrington, N. Huang, P. Miri, S. Radhakrishnan, V. Subramanya, and A. Vahdat, "Portland: A scalable fault-tolerant layer 2 data center network fabric," *SIGCOMM*, 2009.

[22] G. Ananthanarayanan, S. Kandula, and et al., "Reining in the outliers in map-reduce clusters using mantri," *OSDI*, 2010.

[23] P. Gevros, F. Risso, and P. Kirstein, "Analysis of a method for differential tcp service," *GLOBCOM*, 1999.

[24] S. Floyd, M. Handley, and J. Padhye, "A comparison of equation-based and aimd congestion control," 2000.

[25] I. A. Rai, E. W. Biersack, and G. Urvoy-Keller, "Size-based scheduling to improve the performance of short tcp flows," *IEEE Network*, vol. 19, pp. 12–17, 2005 Jan-Feb.

[26] S. Gorinsky, E. J. Friedman, S. Henderson, and C. Jechlitschek, "Efficient fair algorithms for message communication," 2007.