Non-Intrusive Scheduling of TCP Flows

Urtzi Ayesta, Lionel Bertaux, Denis Carvin

Non-Intrusive Scheduling of TCP Flows

U. Ayesta[a,b,c,d], L. Bertaux[a,d], D. Carvin[a,d]
[a]CNRS, LAAS, 7 avenue du colonel Roche, 31400 Toulouse, France
[b]IKERBASQUE, Basque Foundation for Science, 48011 Bilbao, Spain
[c]UPV/EHU, Univ. of the Basque Country, 20018 Donostia, Spain
[d]Univ. de Toulouse, INP, INSA, LAAS, 31400 Toulouse, France

December 9, 2014

Abstract

We investigate how to build a non-intrusive scheduled TCP. For the flows of a given *origin-destination* pair, the objective is to schedule their TCP segments (according to some desired criteria) without modifying the network bandwidth-share used by these flows, which in turn ensures *friendliness* with respect to the rest of the network. We show that in order for a scheduling algorithm to be strictly non-intrusive, a sufficient and necessary condition is that the sender's and receiver's buffers are infinite. We then show that, under the additional condition that segments are neither lost or reordered, the number of active TCP flows can be minimized by size-based schedulers, and we propose a new scheduler `FAIR`, which guarantees that the transfer time of every TCP flow for the *origin-destination* pair is reduced. We develop `SCHED_TCP`, a user space implementation of our scheme in order to evaluate its performance on the Internet. Our experiments illustrate the non-intrusive property of `SCHED_TCP`, and also illustrate that the performance gain with `SCHED_TCP` can be considerable. Our scheme is scalable, and it could be incrementally deployed on the Internet improving the user experience on every *origin-destination* pair. The main application domain of our approach correspond to situations in which there are many TCP concurrent connections within the same *origin-destination* pair, this might happen as a consequence of HTTP 1.1, Web 2.0 applications using AJAX (Google Maps etc.), Split TCP, Parallel Sockets, and also with the use of ChromeBook's where the user accesses to all services through the same back-end server infrastructure.

1 Introduction

The last twenty years have witnessed a tremendous effort to understand and optimize the performance of congestion control algorithms over the Internet. Many variants of TCP have been proposed and studied, and the interaction of congestion control and buffer dynamics have also thoroughly been investigated. Without this list being exhaustive, researchers have looked at ways to improve the performance of TCP by modifying various aspects of the Additive Increase Multiplicative Decrease behavior of TCP, or by modifying the way packets are handled in buffers. An important stream of research, promoted by the advocates of the diffserv architecture, has been to look at how to improve the quality of experience by implementing service differentiation and priorities in the routers, see [11]. Several works have shown that the implementation in routers of

scheduling policies can improve the performance, particularly in the case of overload or close-to-saturation regimes, see for instance [33] and [10].

In spite of all these efforts, the actual number of changes and modifications that have been implemented in real protocols and systems is rather small. The main reason for this is that even though such proposals definitely improve the efficiency *locally*, there is little understanding as to what would be the impact if they were implemented *globally* at the scale of the Internet. An argument against the deployment of scheduling disciplines in routers is presented in [36], where the authors show that the performance in a network where each node implements a size-based scheduling can be extremely poor.

The above provides us with the motivation to undertake a different approach. We focus on a pair of *origin* and *destination* nodes, and we develop a scheme to implement scheduling policies on TCP flows in a *non-intrusive* and transparent way with respect to the rest of the network. In other words, the TCP segments of a given *origin-destination* are scheduled – according to some policy that can be freely chosen – without modifying the bandwidth share assigned by the network to this particular set of flows. This permits to build new schedulers that improve the performance for a set of flows sharing the same *origin-destination* pair, without having any harmful impact on the rest of the flows being transmitted over the Internet.

We now mention the main contributions of this paper. In the first main contribution we develop *gtcp*, a general congestion-control algorithm that can interchange the contents of segments from the flows within a given *origin-destination* pair in a non-intrusive way, that is, without modifying the bandwidth-share assigned to this set of flows. We also establish that having an infinite sender's and receiver's buffer is sufficient and necessary in order for *gtcp* to be *non-intrusive*. In the second main contribution, we apply classical results from scheduling theory to show that in the absence of packet losses and packet reordering, scheduling disciplines that favor TCP flows based on the *unsent amount of data* or *data already sent* minimize the transfer time and the number of on-going flows. We also develop `FAIR`, a scheduler that reduces the transfer time of each flow when there is no packet loss nor packet reordering on the network path. In our third main contribution we develop `SCHED_TCP`, a user space implementation of our scheme in order to evaluate its performance on the Internet. Our experiments illustrate the non-intrusive property of `SCHED_TCP`, and also illustrate that the performance gain with `SCHED_TCP` can be considerable. Our scheme is scalable, and it could be incrementally deployed on the Internet improving the user experience on every *origin-destination* pair.

It is important to highlight that all the above positive and beneficial results do not inflict any harm to the rest of the network, and that *gtcp* and `SCHED_TCP` can work with any TCP version like New Reno [20], CUBIC [23] or Compound [35]. These two aspects would permit an incremental deployment of `SCHED_TCP` over the entire Internet improving the performance of flows on every *origin-destination* pair. The main application domain of our approach correspond to situations in which there are many TCP concurrent connections within the same *origin-destination* pair, for instance this might happen as a consequence of HTTP 1.1 [18], Web 2.0 applications using AJAX (Google Maps etc.) [31], Split TCP [12], Parallel Sockets [34], and also with the use of ChromeBook's [2] where the user accesses to all services through the same back-end server infrastructure.

Our work shares similarities with the approaches of [27, 9, 33, 13], but there are two main differences. The first one is that our approach ensures that the bandwidth share taken by an *origin-*

destination pair remains unmodified with respect to what would happen under standard TCP, and the second one is that we do not necessarily require implementing a size-based scheduling policy. In our approach the congestion-control and scheduling features of TCP are decoupled, and for any *origin-destination* pair we are free to choose how flows are scheduled. Size-based scheduling is an appealing option since it enables to derive mathematical properties of the approach. However, other solutions are also equally viable, like giving priority to real-time, multimedia or interactive traffic over best-effort one, or by following an arbitrary order that would be optimized by the application layer.

The rest of the paper is organized as follows. In Section 2 we provide a brief overview of the related literature, in Section 3 we develop *gtcp* and in Section 4 we show how classical results from scheduling theory can be used to improve the performance of TCP flows. In Section 5 we present a discussion on the technical conditions needed to be non-intrusive, and Section 6 presents the results we have obtained with `SCHED_TCP`.

2 Related work

There are two main research bodies that are related to our proposal, one deals with modification and improvements of the TCP stack, and the other with the implementation of scheduling policies in IP networks. We briefly review some of the main results in these two areas.

The number of TCP modifications that have been proposed over the years is huge. Without aiming at being exhaustive, we mention some of the most significative ones: modification of the initial window [7], faster start proposals [26], handling more efficiently packets losses as done with ECN [28, 19], SACK [21] or RCP [15], or improving the estimation of the available bandwidth as done with TCP Westwood [17]. There have been dozens of new TCP proposed, and current Linux implementations deploy TCP CUBIC [23] whereas Windows implements TCP Compound [35]. These TCP versions only modify the congestion algorithm, but do not add new functionalities. Modern approach to Transport protocol design requires to split Transport layer [16] into sub-layers that can be shared between applications or between connexions. This approach has been followed by MPTCP [4] which enables to enhance available bandwidth by using all the network interfaces and paths between two hosts.

The diffserv architecture was described in [11], here each packet is classified into a limited number of traffic classes and each router on the network is configured to differentiate traffic based on its class. An example is [25] where interactive traffic is given priority over the rest. In order to preserve the quality of service, researchers have also advocated other solutions like admission control [24] or active queue management [6]. Authors of [13] have considered the scheduling of application traffic in computer clusters. Another stream of works have investigated the impact of giving priority in buffers to packets based on the size of the flows [27], [9] and [33]. The latter works draw from classical scheduling theory that giving preference to short tasks is beneficial in a single-server queue. Well-known results are that among *size-aware* policies, Shortest-Remaining Processing Time (`SRPT`) minimizes the number of tasks in the system [32], and that among *size-unaware* policies Least Attained Service (`LAS`) minimizes stochastically the number of tasks in the system when the service time distribution has a decreasing hazard-rate (DHR), see [30]. We refer to [37] for a survey with the most important scheduling results.

3 Non-intrusive scheduled TCP: decoupling of congestion control and scheduling

In this section, we show that it is possible to schedule the segments of a set of application flows sharing the same *origin-destination* route, without modifying the bandwidth share that would have been perceived using a standard TCP implementation. In the ensuing we will use the term *tcp* to refer to a standard TCP implementation, regardless of the specific TCP version. In other words, if $V^{tcp}(t)$ denotes the amount of traffic that has been injected in a given route by TCP up to time t, we will show that under the condition that the sender and receiver buffers are unbounded, a more general congestion-control *gtcp* can arbitrarily interchange the contents of the segments using a policy π, in such a way that $V^{gtcp_\pi}(t) = V^{tcp}(t)$, $\forall t$. The latter property implies that *gtcp* is non-intrusive with respect to the reference protocol *tcp*. This opens up the path to optimize the quality of service perceived by this set of flows, without incurring any harm on the other flows sharing the network. To build such a non-intrusive *gtcp*, we will need to be able to *emulate* at all times what would have been the state of every flows under *tcp*. In turn, this would enable *gtcp* to inject segments into the network at the *exact same times* as what *tcp* would have done. We note that naive solutions like combining all TCP flows into one single flow do not allow to achieve this goal.

In order for *gtcp* to be able to implement an arbitrary scheduling policy π, *gtcp* will need to keep two separate state representations for every flows: the *virtual* that keeps track of the state under *tcp* and the *real* one. Every time a packet is going to be sent (which depends on the *virtual* states), the scheduler π will choose which flow will really send its contents (this can depend on both the *real* and *virtual* states, and on other contextual information such as the QoS). In order to achieve our goal, we will show that is sufficient for *gtcp*, to encapsulate in all its segments, information on both the virtual TCP flow (which enables to keep track of the state of every *virtual* TCP flow) and on the real TCP (which enables to update the *real* state).

The rest of the section is as follows. In Subsection 3.1 we describe how a non-intrusive *gtcp* can be built for any scheduling policy π by describing in detail the algorithms that deal with the main events in a network, namely, arrival of application data (wri), reception of acknowledgement (ack), and retransmission timeout (rto). In Subsection 3.2 we state the formal result, establishing that if the sender and receiver's buffers are unbounded, it is possible to implement any scheduling policy π such that $V^{gtcp}(t) = V^{tcp}(t)$, $\forall t$.

3.1 Algorithmic framework : maintaining virtual queues

As depicted in the upper plane of Figure 1, TCP can be modeled as a multiple-server multiple-queue system, where each server is associated to a transmission queue (or queue). Our approach decouples queues and servers with a scheduling policy π and guarantees that servers' activities remain the same. In our convention, a service (or a sequence number) is the transmission of a segment, it starts with the first transmission attempt and is completed on the reception of its ack. The service can be preempted upon reception of three duplicate ack's, which will trigger π to choose a packet (not necessarily the same) for transmission. As a result, a server offers services in parallel, whose duration and order of completion depend on the network and on *tcp*. The total amount of traffic produced by the servers is captured by the function $V^{tcp}(t)$.

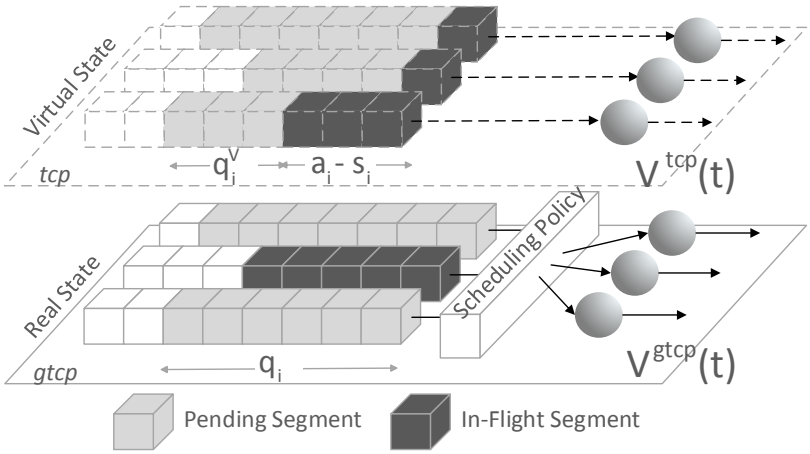

Figure 1: The two planes of *gtcp*. While in the underlying plane the second queue is prioritized, we maintain in the upper plane the virtual states of queues.

We consider an established state without any optional behavior. For the sake of clarity, we assume the size l of arrivals in queues to be multiples of the maximum segment size mss. As a consequence all the packets have the maximum segment size. This assumption is not really required in order to build *gtcp*, but it greatly simplifies the exposition, see Remark 1 for a discussion on how to handle variable segment sizes.

In *tcp*, the possibility for server i to transmit a new segment depends on its state variables which are :

- q_i: Size of transmission of queue i
- s_i: Highest in-flight sequence number
- a_i: Highest cumulatively acknowledged sequence number
- $cwnd_i$: Size of congestion window
- $rwnd_i$: Size of receiver window (remote host)

In order to be standard compliant [8], a new segment cannot be sent by server i if $cr_i = 0$ or $q_i = 0$. Here, cr_i denotes the transmission credit and is expressed as $cr_i = \min(cwnd_i, rwnd_i) + a_i - s_i$. With *gtcp*, we decouple queues from servers, as a consequence q_i will depend on π. As a substitute for q_i, we introduce q_i^v, the virtual size of queue i which keeps track of the value of q_i under *tcp*.

Server i evaluates its state to offer new services on the occurrence of events which are either (i) wri_i^l : l bytes arrive to queue i or (ii) $ack_{i,j}^{s,s'}$: segment s' of queue j is acknowledged with the service s of server i. Algorithm 1 details the actions to perform on the occurrence of the event wri_i^l.

On the arrival of l data in queue i (recall that l is proportional to mss), server i updates the size of its virtual queue q_i^v to compensate the arrival in its real queue q_i, and it offers new services up to its limitation based on its virtual queue. After each service, server i updates its

Algorithm 1 on_wri(i, l)

$q_i^v \Leftarrow q_i^v + l$ // $q_i \Leftarrow q_i + l$
while $cr_i > 0$ and $q_i^v > 0$ **do**
  serve(i, s_i)
  $s_i \Leftarrow s_i + mss$ // $cr_i \Leftarrow cr_i - mss$
  $q_i^v \Leftarrow q_i^v - mss$ // $q_j \Leftarrow q_j - mss$
end while

state as follows: it increments its sequence number (which decreases its transmission credit), and decrements the size of its virtual queue q_i^v. The latter means that server i virtually dequeues the data from queue i and consider that the served queue is virtually unchanged. The actions undertaken in the case where server i offers service s are given in pseudo-code in Algorithm 2.

Algorithm 2 serve(i, s)

$j, s' \Leftarrow$ schedule() // implements π
$P_{i,j}^{s,s'} \Leftarrow$ tcp_head(i, s) | sched_head(j, s') | data()
send$(P_{i,j}^{s,s'})$ // generate $sen_{i,j}^{s,s'}$
set_timer$(P_{i,j}^{s,s'})$ // schedule rto_i^s
associate$((i, s), (j, s'))$
mark(j, s')

When server i is about to transmit a segment s, it first asks to the external function `schedule` which segment s' of which queue j it must serve. The function `schedule` implements the scheduling policy π specified by *gtcp*. Then server i builds a packet containing the service identification (i, s), the data identification (j, s') and the data. Both identifiers are associated and the segment of data is marked as *in-flight* for the scheduler. We denote the transmission event by $sen_{i,j}^{s,s'}$. Note here that policy *tcp* is the particular case of *gtcp* when the scheduler π assigns $(j, s') = (i, s)$.

On the receiver side, the counterpart of event $sen_{i,j}^{s,s'}$ is $rcv_{i,j}^{s,s'}$. When this event takes place, the receiver produces the acknowledgement event $ack_{i,j}^{s,s'}$ and advertises $rwnd_i$. Additionally, it keeps the association (i, s) and (j, s'). In the case of multiple reception of the same service (acknowledgment loss, poor RTT estimation...), the same association is immediately acknowledged. As we explained above, an $ack_{i,j}^{s,s'}$ event can trigger the transmission of a packet by server i. The reaction to this event is described in Algorithm 3.

On the reception of an *ack*, server i updates its congestion state. In particular, server i modifies the value of $cwnd_i, rwnd_i, a_i$ (and consequently cr_i) but also performs other actions like timers' cancelation, RTT estimation, etc. Since these actions depend on the version of TCP, we encapsulate them in the `up_cong_state` function which returns the acknowledgement type (number of duplication). If the acknowledgement is a third duplicate, a retransmission is triggered, which means that the service is preempted. Thus, we can serve another application and associate an additional segment to this service. If the acknowledgement is not a third duplicate, new services are offered with the `serve` function. In the case of a cumulative acknowledgement, the acknowledged *segment of queue* is released. All the other segments that were associated to the service are

Algorithm 3 on_ack(i, s, j, s')

```
ack_type ⇐ up_cong_state()
if ack_type = third_dupack then
  serve(i, s)
else
  if ack_type = cumulack then
    unmark_all_associations_of((i, s))
    remove_all_associations((i, s))
    free_tx_buf((j, s'))
  end if
  while cr_i > 0 and q_i^v > 0 do
    serve(i, s_i)
    s_i ⇐ s_i + mss          // cr_i ⇐ cr_i − mss
    q_i^v ⇐ q_i^v − mss      // q_j ⇐ q_j − mss
  end while
end if
```

unmarked and will be reconsidered by the scheduler for future transmission.

Finally, we have the rto_i^s event, which is a retransmission time-out. This even is processed by the sender like the detection of a third duplicate acknowledgement.

3.2 Formal statement

Based on this algorithmic framework, we have:

Proposition 1 *In any given sample-path, it is possible to adopt any arbitrary scheduling policy π such that $\forall t, \quad V^{gtcp}(t) = V^{tcp}(t)$ if and only if the sender and receiver buffers are not bounded.*

Proof. Let $\mathcal{T}^{tcp}(t) = \{t_i \in [-\infty, t[\quad i \in \mathbb{N}\}$ denote the set of event times up to time t under the standard *tcp*, with the convention that $\forall p < q,\ t_p \leq t_q$. Equivalently let $\mathcal{T}^{gtcp}(t)$ denote te set of event times with *gtcp*. To specify which event occurs at these instants, we define a function $E^{tcp} : \mathcal{T}^{tcp}(t) \to \mathcal{E}^{tcp}$, where $\mathcal{E}^{tcp}$ is the set of possible events that can occur at time t. The function E^{gtcp} is defined equivalently. To define $\mathcal{E}^{gtcp}$ we only consider the events that impact TCP's behavior, namely, $\mathcal{E}^{gtcp} = \{wri_j^l, ack_{i,j}^{s,s'}, rto_i^s, sen_{i,j}^{s,s'}, dat_{i,j}^{s,s'}\}$ where i, j are a queue and server identifier at time t, respectively, and $l, s, s' \in \mathbb{N}^3$. $\mathcal{E}^{tcp}$ is the same as $\mathcal{E}^{gtcp}$ with the restriction that necessarily $s = s'$ and $i = j$. It is easy to see that we have: $V^{gtcp}(t) = |\{t' \in \mathcal{T}^{gtcp}(t) \mid \exists (i, s), (j, s') \text{ s.t. } E^{gtcp}(t') = sen_{i,j}^{s,s'}\}| \times mss$, which states that the total amount of traffic injected up to time t equals the number of segments sent before t multiplied by their size. $V^{tcp}(t)$ is similarly defined.

By construction of Algorithms 1-3, for any policy π of *gtcp*, we have $\mathcal{T}^{gtcp}(t) = \mathcal{T}^{tcp}(t)$, and the following equivalence:

$$(\exists\ (i, s)\ \mid\ E^{tcp}(t') = sen_i^s)$$
$$\iff (\exists\ (i, s), (j, s')\ \mid\ E^{gtcp}(t') = sen_{i,j}^{s,s'}).$$

As a result, we have $V^{gtcp}(t) = V^{tcp}(t)$, $\forall t$, regardless of the policy π implemented by *gtcp*. For the above argument to hold the sender and receiver buffers must be infinite. If the sender buffer were finite, it is possible to find a sequence of TCP flows arrivals, such that under π the buffer overflows and segments are lost, whereas the *virtual* buffer remains small. The latter can happen on the sender's side for a low priority queue which is significantly less served than under *tcp*. On the receiver's side, it can happen for a priority queue which delivers a much larger amount of data than under *tcp*, the receiver window becoming possibly less than the congestion window. Such events will alter the future events in such a manner that their corresponding algorithm can not be called and it is no longer possible to preserve $\mathcal{T}^{gtcp}(t) = \mathcal{T}^{tcp}(t)$ from that moment on. ■

Remark 1 (On arrivals that are not a multiple of mss**)** *In this case we could encounter the situation where the size of a service is greater than the size of the prioritized queue. We would thus need to encapsulate several segments of different queues in the same service. The price of this situation is that it requires an enhanced mechanism to associate one service to a set (or several sets if retransmission) of queue segments that have different sizes.*

The unboundedness of the buffers is needed in order for *gtcp* to be *non-intrusive*, but is not required to implement an arbitrary scheduler in reality. This is investigated in Section 6, where we will show that `SCHED_TCP`, a user space implementation of *gtcp*, is qualitatively non-intrusive, (*i.e.* at a coarser time-scale than the one of the packet).

4 Scheduling: optimality of size-based policies

In Section 3 we have shown that the *congestion control* and *scheduling* aspects of TCP can be totally decoupled. As a consequence, a source can efficiently schedule TCP segments without any harmful impact on any other active data transmission in the network. The properties of the scheduling algorithm were hidden behind the function `schedule` used in Algorithms 2.

The objective of this section is to elaborate on how *size-based scheduling* and *priority policies* can improve the performance perceived by flows using the same *origin-destination* pair. In particular, we will discuss how these disciplines can reduce the transfer time of flows and also the number of on-going flows. Throughout this section we assume that all the data of a given flow is available to be *scheduled*. If an application transmits data over the same TCP connection at discontinuous time intervals, for the purposes of this section each of these transmissions can be seen as a *separate* flow.

4.1 Equivalence with a queue with time-varying capacity

Our scheduling problem is different from the classical problem studied in the literature. In the classical problem, tasks receive reliable service "instantaneously" and depart as soon as their required service time has been served, whereas in our setting, segments are scheduled on transmission epochs, and a while later, when the *ack* is received, we consider the information to be successfully transmitted.

In order to establish optimality results, we need to assume the absence of segment losses, and also of segment reordering. The results of Section 6 indicate that, in the presence of segment losses

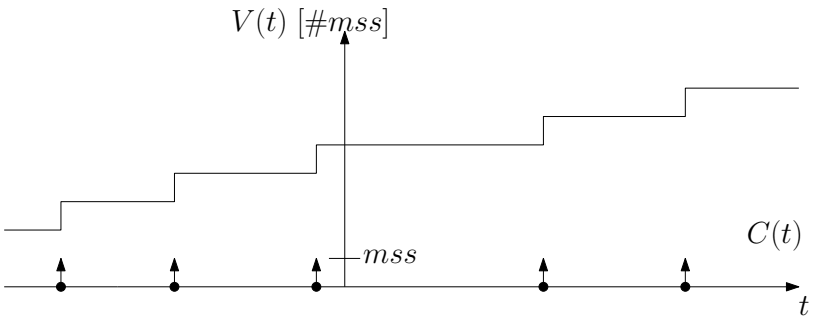

Figure 2: Function $C(t)$ that captures the instantaneous data transmission. The black dots represent time epochs in which a segment is transmitted.

and segment reordering, size-based scheduling will still provide a significant performance gain, see also Remark 2. Since neither data segments nor *ack*'s are reordered, we can equivalently assume that a segment is successfully transmitted as soon as it departs from the sender's buffer, and we can thus say that a flow departs the system as soon as its last data segment is sent. Let $\mathcal{N}^\pi(t)$ denote the set of flows that have unsent segments at time t and let $N^\pi(t) = |\mathcal{N}^\pi(t)|$ denote the number of flows present at time t. The assumption that segments are neither lost nor reordered directly implies that minimizing $N^\pi(t)$ is equivalent to minimizing the number of active flows measured at the reception of *ack*'s.

As explained in Section 3, our scheme enables that the total number of transmitted segments at time t, $V^{gtcp}(t)$, is independent of how segments are scheduled by the scheduler π implemented in *gtcp*. Let t_i, $i = 1, 2, \ldots$, denote the transmission epoch of the i-th segment under *tcp*. It will be proven useful to consider a capacity function $C(t) = mss \times \sum_{i:t_i \leq t} \delta(t - t_i)$, where $\delta(\cdot)$ is the Dirac delta function. Figure 2 depicts the function $C(t)$ in a particular example. The function $C(t)$ can be seen as time-varying capacity function, and it captures the *instantaneous* rate at which data is being transmitted, that is, at time t data is instantaneously transmitted at rate $C(t)$. It is easy to see that for all t, $\int_{-\infty}^{t} C(s)\mathrm{d}s = V^{tcp}(t)$, so under the function $C(t)$, a segment is *instantaneously* transmitted at times t_i, $i = 1, 2, \ldots,$.

The problem of determining an optimal scheduling algorithm π can now be cast as a problem of optimal scheduling in a single-server queue with a time-varying service capacity $C(t)$. As we will show in Subsection 4.3, this allows us to use known results from the literature to establish optimality results. Note that we have dropped the dependency of $C(t)$ on π to highlight that, as shown in Proposition 1, we can implement a scheduler π while preserving the bandwidth share at all time epochs, i.e., $V^{gtcp}(t) = V^{tcp}(t)$, $\forall t$. This result is critical in order to compare the performance of scheduling disciplines. We will now introduce the main size-based disciplines that we will need in the rest of the section.

4.2 Scheduling disciplines

The discipline implemented by *gtcp*, denoted by π, specifies which flow is served at time t. A selected flow will receive instantaneous service at rate $C(t)$. Since $C(t)$ is formed by Dirac delta functions, this implies that if $t \neq t_i$, then the rate is 0, and that if $t = t_i$, a segment with

mss amount of data will be sent. Depending on the information available to the scheduler, two important classes of disciplines are: (i) *size-aware* if the required service times of flows is known and (ii) *age-based* if the amount of received service is known (but not the required service time). The notions of *required service time* and *received service* commonly used in the scheduling literature must be here understood as *amount of segments a flow must transmit* and *amount of segments already transmitted by a flow*, respectively. Well-known disciplines are:

- Least Attained Service (`LAS`) is an *age-based* discipline which serves the flow that has received the least amount of service, which as explained above, in our terminology means the TCP flow that has transmitted the least number of segments.
- Shortest-Remaining-Processing-Time (`SRPT`) is a *size-aware* discipline that serves the flow with the smallest remaining service time.
- Fair TCP (`FAIR`) is a *size-aware* discipline, which serves the flow that will finish next under *tcp*.

(`FAIR`) is inspired by the Fair Sojourn Protocol introduced in [22]. In the absence of segment losses and segment reordering, if the scheduler is *size-aware*, and if *gtcp* knows the behavior of *tcp*, it is possible for *gtcp* the scheduler to infer *exactly* which would be the next flow under *tcp*.

It will also be useful to consider *priority policies* that apply a priority among different type of flows. Here a type might refer for instance to the application: web, VoIP, P2P etc.

4.3 Optimality results

Proposition 2 below summarizes our main theoretical results that illustrate how size-based scheduling improves the performance of the flows in a given *origin-destination* pair. We will need the notion of *hazard-rate*. Let $\mathbb{P}(S = k)$ denote the probability that a flow requires to transmit k segments. The hazard rate of a distribution is then given by $h(k) := \frac{\mathbb{P}(S=k)}{\mathbb{P}(S \geq k)}$, and when $h(k)$ decreases in k we say that the distribution is of type *decreasing hazard rate* (DHR). Flow size distributions on the Internet are commonly modeled with distributions of type DHR, see for example [14].

Proposition 2 *Under the conditions of Proposition 1, if segments are neither lost nor reordered, we have:*

- *If the scheduler is* `SRPT`*, then* $\{N^{\mathtt{SRPT}}(t)\}_{t\geq 0} \leq \{N^{gtcp}(t)\}_{t\geq 0}$*, where gtcp can implement any arbitrary* size-aware *discipline.*
- *If the service time distribution is DHR, then* $\mathbb{P}(N^{\mathtt{LAS}}(t) > k) \leq \mathbb{P}(N^{gtcp}(t) > k)$*,* $\forall k$*, where gtcp can implement any arbitrary* age-based *discipline.*
- *If the scheduler is* `FAIR`*, then* $T_i^{\mathtt{FAIR}} \leq T_i^{tcp}$*, where* T_i *denotes the transfer time of the i-th flow.*
- *If flows are classified into types, then* $\mathbb{E}(N^{\mu\text{-}rule}) \leq \mathbb{E}(N^{gtcp})$*, where* μ*-rule denotes the priority discipline that gives full service to the type (present in the system) with smallest mean number of segments.*

Proof. The proofs are applications of known results to the system with instantaneous transfer time $C(t)$, see Appendix A for more details. ■

By a straightforward application of Little's Law, we then obtain the following corollary:

Corollary 1 `SRPT`, `LAS`, `FAIR`, *μ-rule minimize the mean transfer time of flows in each of the settings described in Proposition 2.*

Remark 2 (On the necessity of absence of segment losses and reordering) *These assumptions are required to establish the optimality results of Proposition 2. To explain this, consider the case of* `SRPT` *and that at a given time there are two flows, one with $R_1(t) = 1$ and the other one $R_2(t) = 2$.* `SRPT` *will schedule flow 1, but if this segment is lost or reordered, flow 2 might receive the ack's earlier and as a consequence finish its transmission earlier. This would imply that* `SRPT` *is not optimal in the sample-path sense as stated in Proposition 2. For* `LAS` *we can construct similar counterexamples. Regarding* `FAIR`*, if there are losses or reordering it is not possible to predict exactly which flow will finish next. As a consequence, it will no longer hold that* `FAIR` *reduces the transmission time of all flows. Nevertheless, the results of Section 6 indicate that in the presence of segment reordering and losses, the performance gain of* `SRPT`*,* `LAS` *and* `FAIR` *with respect to tcp is considerable.*

5 On the technical conditions

This section presents a discussion on the impact of various of the conditions mentioned in Propositions 1 and 2, namely, (i) schedulability of traffic, (ii) unbounded buffers, (iii) size of segments and (iv) absence of segment losses and reordering.

(i) Schedulability of traffic. The possibility to schedule transmission queues only arises when several queues sharing the same route are non-empty at the same time. In order to obtain an idea of the amount of traffic that satisfies this assumption, we have analyzed thirty minutes of TCP/IPv4 traffic captured from Seattle to Chicago in March 2014 on an OC192 backbone link of a Tier1 ISP [1]. Since it is not possible to infer the state of a TCP transmission queue in the Internet, we approximate the number of non-empty queues by the number of active flows. A flow is considered active from the time of its first data segment until the time of its last one, and we say that two flows compete on a route if their activity period overlap. Table 1 summarizes our results, where we represent the number of routes that have competing flows, the total number of competing flows, and the total data volume carried by competing flows. The main conclusion is that even though

Table 1: Proportion of schedulable traffic

	Total	Shared/Competing	Ratio
Routes (number)	1788796	102981	5.7%
Flows (number)	5155554	937033	18.1%
Flows (volumes in KB)	261769672	152530580	58.2%

only 18.1% of the flows compete, they carry 58.2% of the total data volume. This illustrates that,

in addition to applications mentioned in the introduction, the transmission of big data files can be a domain of interest for our scheme, and in particular we can think of cloud deployment, database replication and distributed analysis as considered in [13].
(ii) Unbounded buffers. On the sender side, infinite buffers are only necessary for low priority queues and can be emulated by a blocking call on the corresponding socket. Thus, if low priority applications are able to wait like in the case of file transfers, we can consider that the sender buffer is infinite. On the receiver side, it is reasonable to consider that the buffer is infinite when congestion window is always less than the receiver window. The latter naturally happens when the congestion is in the network and not on the receiving host. For instance, the approach of [29], in which the receiver's window is dynamically adjusted, would ensure that the receiver window is always larger than the congestion window.
(iii) Size of segments. In Remark 1 we explained that it was possible to handle different segment sizes at the price of adding a layer of complexity to our scheme. Using the publicly available statistics from CAIDA [3], in Figure 3 we depict the cumulative distribution function of TCP/IPv4 segment sizes during years 2002 and 2014. We observe that most of the segments correspond to

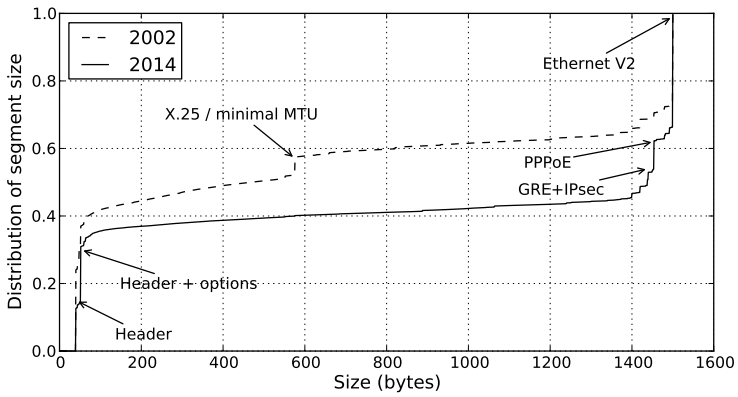

Figure 3: Distribution of TCP/IPv4 segment sizes in 2002 and in 2014 on all probing points of CAIDA

a Maximum Transmission Unit (MTU) of an Internet Technology, and that at least 60% of data segments are full-sized segments.
(iv) Absence of segment losses and reordering. These assumption was needed in order the optimality results of Proposition 2 to hold. In the presence of segments losses and reordering, it is not possible to prove any optimality result, but we nevertheless expect that *gtcp* can provide an important performance gain over *tcp*. We also note that if ECN [19] were deployed, the optimality results of Proposition 2 would hold even with losses. From the losses point of view, the worst-case scenario is the loss of the last *ack* for a flow. In such a case, a server would detect the loss with the *rto* event and all the data sent up to this event would not have been from a priority flow.

6 Implementation and Experimental Results

We present an implementation of *gtcp* along with experimental results for different scheduling policies.

6.1 SCHED_TCP, an implementation of *gtcp*

A congestion control algorithm can be implemented either by relying on an existing Transport protocol or by deploying a new one (as a version of TCP for example). From an implementation point of view, the easiest way to obtain a congestion control *gtcp* that mimics Algorithm 1 of Section 3 is to maintain as many active connections as *tcp* would have, letting each of these connections use *tcp* congestion control. In turn, the latter guarantees that we can replicate the event generation of *tcp*, and we can thus obtain $V^{gtcp}(t) = V^{tcp}(t)$, $\forall t$. The shared scheduling layer of *gtcp* can be deployed on top of these *tcp* connections as an intermediate layer between applications and connections. This approach follows the current trend in Transport protocol design [16], which we already mentioned in Section 2.

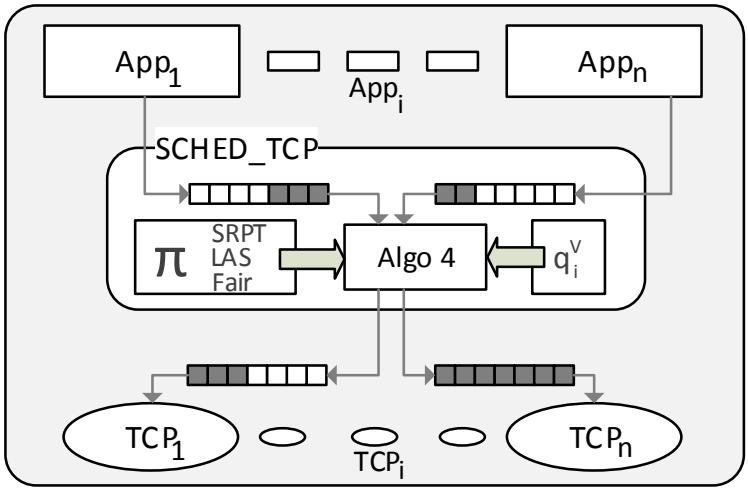

Figure 4: SCHED_TCP implementation with n applications and n TCP connections.

The implementation of *gtcp* used for experiments is a scheduling layer between applications and legacy TCP connections illustrated by Figure 4. This user space program is coded in C on Linux 3.2 and referred to as SCHED_TCP.

SCHED_TCP distributes application flows among TCP connections at the sender side and re-arranges them at the receiver's side. The scheduling component π is in charge of selecting the prioritized flow, while the main component encapsulates data in a scheduling header that identifies the selected flow and forward them to TCP. Algorithm 4 is an implemented solution inspired from Algorithm 1 and 2, where the `schedule` function refers to the scheduling policy π being implemented. When a TCP connection is ready to accept new data in its sending buffer, it checks the state of its virtual queue and chooses the highest priority data flow depending on the scheduling policy π. In the state depicted by Figure 4, the connection TCP_1 has space in its sending buffer

and is thus ready to accept new data coming from the application chosen by the scheduling policy π. The `schedule` function of Algorithm 4 implements *tcp*, `SRPT`, `LAS` and `FAIR` policies described in Section 4.3.

Algorithm 4 On_connection_ready(i)

```
if q_i^v > 0 then
    j, s' ⇐ schedule()                  // implements π
    P_j^{s'} ⇐ sched_head(j, s') | data()
    enqueue(i,P_j^{s'})                 // equivalent to sen_{i,j}^{s,s'}
    q_i^v ⇐ q_i^v − mss                 // q_j ⇐ q_j − mss
end if
```

Compared to the theoretical behavior of *gtcp*, `SCHED_TCP` has some limitations due to the use of legacy TCP connections. Since transmission credit is managed by TCP, it is not possible to preempt a low priority segment once it has been queued in the socket i.-e. parts of transmission buffers are not shared among flows. More specifically, the scheduling discipline π of `SCHED_TCP` can choose the amount of data given to TCP connections while *gtcp* allows to choose the exact segment sent on the network. Despite this difference, the following section shows that `SCHED_TCP` provides a significant performance gain over *tcp*.

6.2 Experimental Results and comparison of policies

In this sub-section we present measurements done on a real network with `SCHED_TCP` deployed on a *origin-destination* pair consisting of two desktop computers (see Figure 5). The origin is located in our research lab and the destination is in a home network with poor Internet connection (bandwidth equivalent to 350kBps). We note that in a real network is impossible to compare the performance of two protocols under the *same* traffic conditions. In order to counteract the non-replicability of the experiments, we repeat each experiment 100 times and then take an average.

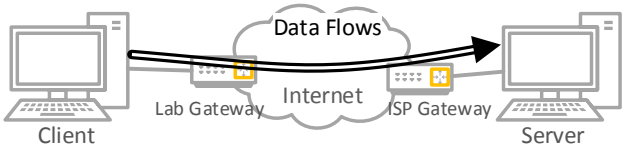

Figure 5: Testbed used for the first experiment.

Non-intrusiveness. In Figure 6 we depict the average throughput of flows for *tcp* and `SRPT` policies. We generate three flows at predefined starting times and as explained above, we repeat the experiment 100 times. In the top figure all three flows are handled by *tcp*, whereas in the figure below `SCHED_TCP` handles the flows *long* and *short*, and *tcp* handles the flow *sample*. Upon arrival at $t = 5s$, `SCHED_TCP` gives full priority to *short*, and as a consequence *short* finishes its transmission around $t = 11s$. On the other hand, *long* finishes its transmission at around $t = 25s$ in both cases. The throughput obtained by *sample* at any time is similar in both cases, which illustrates the non-intrusiveness of `SCHED_TCP`.

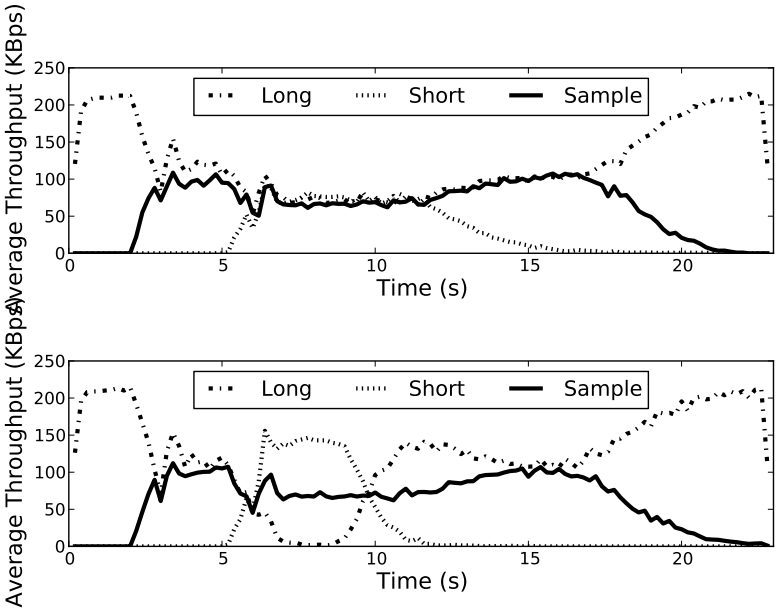

Figure 6: Average throughput of 2 flows scheduled with *tcp* (top) and with `SRPT` (bottom) that compete with 1 *sample* flow with *tcp* policy.

Performance of size-based disciplines. In order to compare the performance of the scheduling policies of Section 4.3, we now consider a more realistic traffic scenario experimented on the production network of our lab (still a real network, but less bandwidth limitations). New flows are generated according to a Poisson process of rate λ. The flow size is assumed to follow a Pareto distribution, that is, if S denotes the size of a flow, we consider that $\mathbb{P}(S > x) = \frac{1}{(1+cx)^{\alpha}}$, $\forall x \geq 0$. Pareto distribution is of DHR type, and has been commonly used to describe the file size distribution on the Internet. We set $\alpha = 3$ and $c = 10^{-7}$, which gives an average file size $\mathbb{E}(S) = 5MB$. The capacity of the link is $C = 12MBps$, and the segment size 1448B. The value of λ is obtained by fixing the overall load in the system to 0.9, that is, $\frac{\lambda \mathbb{E}(S) \cdot 1448}{C} = 0.9$. The results we report come from 100 independent experiments.

Figure 7 shows the distribution of the number of active connections. We observe that `SRPT` and `FAIR` minimize the number of active connections, and that even though `LAS` is better than *tcp*, its performance is worse that than `SRPT` and `FAIR`. The latter shows that the fact of being *size-aware*, gives a significant advantage to both `SRPT` and `FAIR`.

Figure 8 shows the distribution of transfer time per flow. As in the case of the number of connections, *size-aware* policies `SRPT` and `FAIR` provide a significant performance gain. It is noteworthy to mention that `SRPT` and `FAIR` outperform `LAS` and *tcp* regardless of the file transfer duration. We also see that `LAS` improves the transfer time of short flows. However, for flows that last more than 6 seconds, `LAS` is less efficient than *tcp*. This is expected since `LAS` gives priority to new flows at the expense of older ones.

In Figure 9 we depict the transfer time for every simulated flow. A dot in (x, y) means that a flow of size x has finished its transfer at time y. We observe that as the flow size increases, the

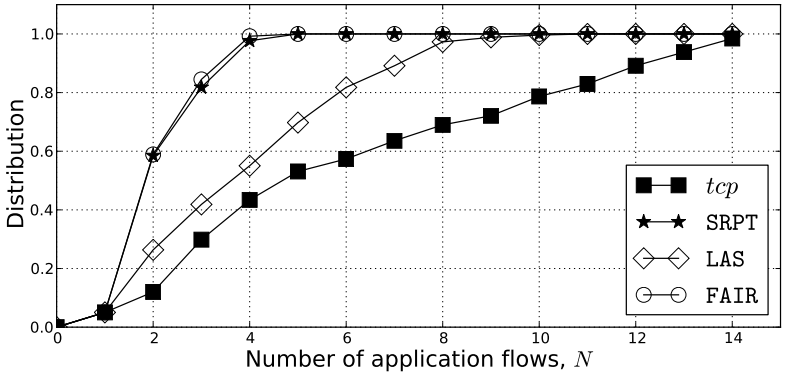

Figure 7: Distribution of the number of active connections (regardless of size) with *tcp*, `SRPT`, `LAS` and `FAIR` policies.

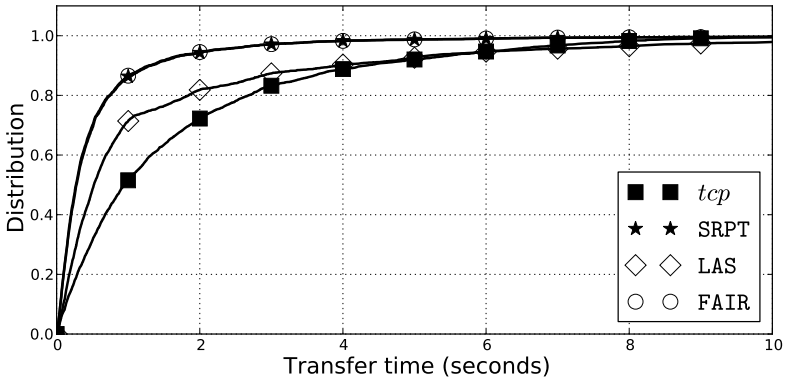

Figure 8: Distribution of flow transfer time with different scheduling policies.

dispersion of the transfer time increases under `LAS` and *tcp*, and that the dispersion stays much smaller under `SRPT` and `FAIR`. This means that *size-aware* are not only better in terms of transfer time, but that they also increase the *predictability* of the performance, that is, all flows of the same size will have a similar transfer time.

Table 2 shows various statistics on the transfer time. Taking *tcp* as a reference, we show the percentage of flows that finish *faster* and *slower*. We see that with `SRPT` and `FAIR`, more than 88% of the flows finish their transfer *earlier* than with *tcp*, and that only 3% of the flows do finish *later*. `LAS` reduces the transfer time for nearly 60% of the flows, whereas the performance is *worse* for 25% of the flows. We also give the *average gain* and loss, which refer to the average reduction *(increase)* of the transfer time among flows that finish earlier *(later)*. The overall conclusion is that `SRPT` and `FAIR` provide a similar performance gain, and that being *size-aware* provides a significant advantage to both `SRPT` and `FAIR`.

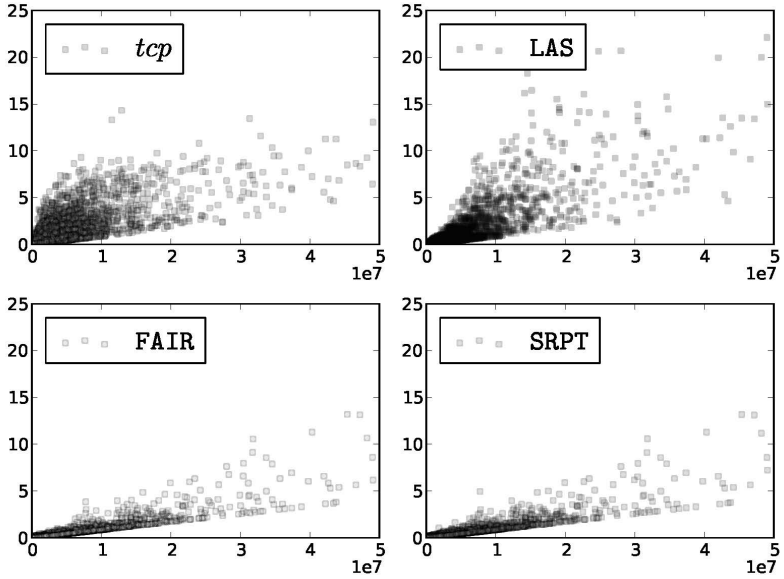

Figure 9: Transfer time in seconds against flow size in MB. A dot in (x, y) means that a flow of size x has finished its transfer at time y.

Table 2: Statistics on transfer time of different scheduling policies.

	Faster	*Slower*	*Equal*	$\overline{Gain}$	$\overline{Loss}$
LAS	61%	25.6%	13.2%	-0.57s	+314ms
SRPT	88.8%	3.2%	8%	-1.11s	+28ms
FAIR	89.6%	2.4%	7.9%	-1.13s	+33ms

7 Conclusions and future work

The main objective of this paper is to show that the *congestion control* and *scheduling* features of TCP can be efficiently decoupled. We have introduced *gtcp*, a new protocol that permits to implement any scheduling policy among the flows sharing any given route, while preserving the overall impact of these flows on the network. The experiments with SCHED_TCP on a real network with have shown that the performance gain can be significant, and that this gain comes at no cost for other flows.

We have focused on *size-based* scheduling disciplines. This choice allowed us to derive optimality results for *gtcp*, but the scheduler of *gtcp* need not be restricted to this particular class of policies. For instance, one could consider giving priority to a certain type of traffic, like interactive traffic, web, email or Skype, over the rest. More generally, the end user could define his own scheduler π, aiming at maximizing his own particular notion of quality-of-service.

SCHED_TCP could be incrementally deployed, improving the performance of the flows of any given route, and without harming the performance of any other flow on the network. It could be

implemented on the end-hosts, or between two proxies within the network in order to *aggregate* traffic.

References

[1] The CAIDA UCSD anonymized internet traces 2014, equinix-chicago, 03-20-2014. "http://www.caida.org/data/passive/passive_2014_dataset.xml".

[2] Chromebook. http://en.wikipedia.org/wiki/Chromebook.

[3] The web site of the cooperative association for internet data analysis (caida). http://www.caida.org/.

[4] M. Handley ands O. Bonaventure A. Ford, C. Raiciu. TCP extensions for multipath operation with multiple addresses. RFC6824, January 2014.

[5] S. Aalto and U. Ayesta. SRPT applied to bandwidth-sharing networks. *Annals of Operations Research*, 170:3–19, 2009.

[6] R. Adams. Active queue management: A survey. *Communications Surveys Tutorials, IEEE*, 15(3):1425–1476, Third 2013.

[7] M. Allman, S. Floyd, and C. Partridge. Increasing TCP's initial window. RFC3390, October 2002.

[8] M. Allman and V. Paxson. TCP congestion control. RFC5681, 2009.

[9] K.E. Avrachenkov, U. Ayesta, P. Brown, and E. Nyberg. Differentiation between short and long TCP flows: Predictability of the response time. In *Proceedings of INFOCOM*, 2004.

[10] E. Biersack, B. Schroeder, and G. Urvoy-Keller. Scheduling in practice. *SIGMETRICS Perform. Eval. Rev.*, 34(4):21–28, March 2007.

[11] S. Blake, D. BlaD. Black. Carlson, E. Davies, Z. Wang, and W. Weiss. An architecture for differentiated services. RFC2475, February 1998.

[12] J. Border, M. Kojo, J. Griner, G. Montenegro, and Z. Shelby. Performance enhancing proxies intended to mitigate link-related degradations. RFC 3135 (Request For Comments, 2001.

[13] M. Chowdhury, M. Zaharia, J. Ma, M.J. Jordan, and I. Stoica. Managing data transfers in computer clusters with orchestra. *SIGCOMM Comput. Commun. Rev.*, 41(4):98–109, August 2011.

[14] M. Crovella, M. Taqqu, and A. Bestavros. A practical guide to heavy tails. chapter Heavy-tailed Probability Distributions in the World Wide Web, pages 3–25. Birkhauser Boston Inc., Cambridge, MA, USA, 1998.

[15] N. Dukkipati, M. Mathis, Y. Cheng, and M. Ghobadi. Proportional rate reduction for TCP. In *IMC*, pages 155–170, 2011.

[16] B.Ford et al. Breaking up the transport logjam. In *HOTNETS'08*.

[17] M. Saverio et al. TCP westwood: Bandwidth estimation for enhanced transport over wireless links. In *ACM MOBICOM 2001*.

[18] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext transfer protocol – http/1.1. RFC 2616 (Request For Comments, 1999.

[19] S. Floyd. TCP and explicit congestion notification. *ACM Computer Communication Review*, 24(5):10–23, 1994.

[20] S. Floyd and T. Henderson. The newreno modification to TCP's fast recovery algorithm. RFC2582, April 1999.

[21] S. Floyd, J. Mahdavi, M. Mathis, and M. Podolsky. An extension to the selective acknowledgement (sack) option for tcp. RFC2883, July 2000.

[22] E. Friedman and S. Henderson. Fairness and efficiency in processor sharing protocols to minimize sojourn times. *Proceedings of ACM SIGMETRICS*, pages 229–337, 2003.

[23] S. Ha, I. Rhee, and L. Xu. CUBIC: a new TCP-friendly high-speed TCP variant. *Operating Systems Review*, 42(5):64–74, 2008.

[24] L. Massoulié and J. Roberts. Arguments in favour of admission control for tcp flows. In *Proceedings of ITC-19*, pages 33–44, 1999.

[25] W. Noureddine and F. Tobagi. Improving the performance of interactive TCP applications using service differentiation. In *Proceedings of IEEE INFOCOM*, pages 31–40, 2002.

[26] V. Padmanabhan and R. Katz. TCP fast start: A technique for speed-ing up web transfers. In *Proceedings of IEEE GLOBECOM*, 1998.

[27] I. Rai, G. Urvoy-Keller, and E. Biersack. LAS scheduling approach to avoid bandwidth hogging in heterogeneous TCP networks. In *Proceedings of IEEE HSNMC*, 2004.

[28] K.K. Ramakrishnan, S. Floyd, and D. Black. The Addition of Explicit Congestion Notification (ECN) to IP. RFC3168, September 2001.

[29] P. Ravis, J. Manish, and C. Dovrolis. Socket buffer auto-sizing for high-performance data transfers. *J. of Grid Computing*, 1:361–376, 2003.

[30] R. Righter and J.G. Shanthikumar. Scheduling multiclass single server queueing systems to stochastically maximize the number of successful departures. *PEIS*, 3:323–334, 1989.

[31] F. Schneider, S. Agarwal, T. Alpcan, and A. Feldmann. The new web: Characterizing ajax traffic. volume 4979 of *Lecture Notes in Computer Science*, pages 31–40. Springer Berlin, 2008.

[32] L.E. Schrage. A proof of the optimality of the shortest remaining processing time discipline. *Operations Research*, 16(3):687–690, 1968.

[33] B. Schroeder and M. Harchol-Balter. Web servers under overload: How scheduling can help. In *18th ITC 2003*, 2003.

[34] H. Sivakumar, S. Bailey, and R. L. Grossman. Psockets: The case for application-level network striping for data intensive applications using high speed wide area networks. In *Proceedings of the 2000 ACM/IEEE Conference on Supercomputing*, 2000.

[35] K. Tan, J. Song, Q. Zhang, and M. Sridharan. A Compound TCP approach for high-speed and long distance networks. In *Proceedings of IEEE INFOCOM*, 2006.

[36] I.M. Verloop, S.C. Borst, and R. Núñez-Queija. Stability of size-based scheduling disciplines in resource-sharing networks. *Performance Evaluation*, 62:247–262, 2005.

[37] G. Weiss. A tutorial in stochastic scheduling. In J.K. Lenstra P. Chretienne, E.G. Coffman and Z. Liu, editors, *Scheduling Theory and Its Applications*. Wiley, 1995.

Appendix A: Proof of Proposition 2

The proofs are applications of known results to the *modified* system with instantaneous transfer time $C(t)$, and we provide a sketch. To be specific let us consider the case of SRPT. Since SRPT is *size-aware*, the scheduler has precise information of how many segments each flow needs to transmit in total. Let us assume that *gtcp* implements a scheduler π, and let $\mathbf{R}^{gtcp}(t) = (R_n^{gtcp}(t); n \in \{1, 2, \ldots\})$ denote the *ordered* vector of unsent segments at time t, with the first element $R_1^{gtcp}(t)$ being the largest number of unsent segments, and let $R^{gtcp}(t)$ denote the total number of unsent segments at time t. Since the total number of sent segments $V^{gtcp}(t) = V^{tcp}(t)$ is independent of the scheduling discipline π deployed by *gtcp*, it follows directly that the total number of unsent segments $R^{gtcp}(t) = R^{tcp}(t)$ is also independent of π.

We can now invoke [5, Proposition 1], which holds for arbitrary capacity functions $C(t)$, to show that SRPT preserves over time a relation on the cumulative remaining amount of unsent segments. We have that for any π that *gtcp* may implement, and $k \in \{1, 2, \ldots\}$

$$\sum_{n=k}^{\infty} R_n^{\mathtt{SRPT}}(t) \leq \sum_{n=k}^{\infty} R_n^{gtcp}(t). \tag{1}$$

We can now give a direct argument to show the optimality of SRPT. Let us assume that $N^{gtcp}(t) = n$. Then, by (1) we have $\sum_{j=n+1}^{\infty} R_j^{\mathtt{SRPT}}(t) \leq \sum_{j=n+1}^{\infty} R_j^{gtcp}(t) = 0$, implying that $N^{\mathtt{SRPT}}(t) \leq n = N^{gtcp}(t)$.

The proof of LAS is a direct adaptation of [30, Theorem 2.1]. The DHR assumption intuitively implies that the more service a flow receives, the larger its expected remaining service time is. This explains why the policy LAS is optimal, since by serving flows that have receive less amount of service, the scheduler "guesses" correctly which flow is more likely to finish the transmission earlier. The proof in [30, Theorem 2.1] uses an interchange argument to show that any policy that does not follow LAS in a decision epoch is necessarily sub-optimal.

The result of FAIR is by construction, and is a direct application of [22]. Under FAIR all flows will finish their transmission earlier than what they would under *tcp*, except the flow that transmits the last segment before an inactivity period which will finish at exactly the same moment. To see this we observe that under FAIR, the selected flow will transmit at much higher rate than under *tcp*. In fact, under FAIR, the selected flow gets to transmit in all the next available transmission epochs until it finishes the transmission. It is then trivial to observe that all flows will finish transmission earlier than what they would under *tcp*. An exception is the last flow, when $R^{\mathtt{FAIR}}(t) = 1$, this implies that there is only one flow active, and we then necessarily have $R^{tcp}(t) = 1$. This last flow will finish its transmission exactly at the same time under FAIR and *tcp*.