This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TMC.2018.2870156, IEEE Transactions on Mobile Computing

Janus: A Multi-TCP Framework for Application-Aware Optimization in Mobile Networks

Fan Zhou, *Student Member, IEEE,* David Choffnes, and Kaushik Chowdhury, *Member, IEEE*

Abstract—As the dominant protocol on the Internet, TCP has attracted significant attention and has been implemented in various ways, each of which optimizes for a single objective such as high throughput or low delay. However, in today's mobile networks that carry traffic from diverse types of flows, this approach may lead to misconfiguration of TCP congestion control algorithms and further degrade performance for many applications. In this paper, we propose Janus, a new transport-layer framework that automatically selects among existing congestion control variants to optimize traffic in accordance with application demands. Janus is easy to deploy because it reuses existing, well-tested congestion control implementations, and does not require any in-network or client-side changes. To explore the potential for this approach, we implement Janus in the Linux kernel and extensively evaluate its performance with both emulated and real Internet traffic. We show Janus outperforms alternative protocols by offering fast convergence times in response to changing network conditions, achieving 5-10X lower delay with comparable or higher throughput. Our approach also significantly improves user-perceived performance according to QoE metrics, with up to 5X fewer interruptions for video streaming applications and 2X faster page loading for web-browsing applications.

Index Terms—TCP, congestion control, QoE, transport layer, mobile application.

---

1 INTRODUCTION

User-perceived quality of experience (QoE) is a key factor that influences customer satisfaction. This directly impacts the consumer's preference for wireless service providers as well as the popularity of applications (apps) that run on smart-devices. Thus, a key challenge is how to meet QoE requirements for users continuously, given the tremendous diversity of apps and varying wireless bandwidth with mobility [1]. A variety of innovative congestion control protocols have been proposed for specific target scenarios to address these issues. Over 15 TCP variants are available in the latest Linux release [2], and the number is still growing. We argue that no single protocol can capture the collective needs of every possible app. In this work, we explore the effectiveness of automatically select and configure existing congestion control protocols to maximize the QoE based on the workload for each app that uses the network.

1.1 Motivation for Janus

Despite a large number of available options, only one congestion control protocol (usually chosen from either NewReno or Cubic on Linux) is active within the network protocol stack of a given server. This conservative choice was largely driven by two limitations of congestion control design:

- F.Zhou and K.Chowdhury are with the Department of Electrical and Computer Engineering, Northeastern University, Boston, MA, 02115 (E-mail: zhou.fan1@husky.neu.edu; krc@ece.neu.edu)
- D.Choffnes is with College of Computer and Information Science at Northeastern University (Email: choffnes@ccs.neu.edu)

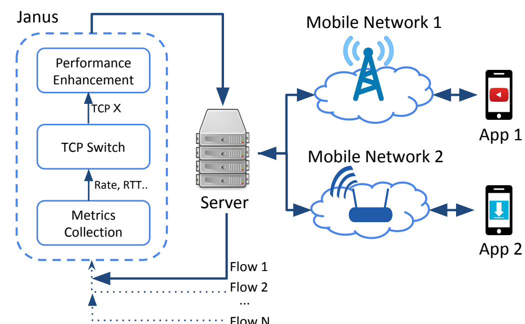

Fig. 1: Janus switches between aggressive / conservative TCP protocols in response to application demands and network conditions.

• *Unknown last-mile wireless access link:* Congestion control protocols are often designed for a specific network architecture and only work best in the scenario for which they were designed (see Sec. 2). Unfortunately for mobile networks, the choice of congestion control protocols is usually made at the server side, which is unaware of the actual *last mile* link condition at the client side. In today's content delivery network (CDN) architecture, a single server may provide service to thousands of mobile users, but there may also be large diversity in the wireless access links that range from traditional WiFi/LTE to 60 GHz mmWave [3]. Therefore, in the absence of knowledge about client-side wireless access conditions, servers are typically configured to use a single, standard congestion control protocol, even though it may lead to suboptimal performance.

• *Application-agnostic protocol selection:* A key goal of a congestion control protocol is to share the bandwidth fairly

1536-1233 (c) 2018 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TMC.2018.2870156, IEEE Transactions on Mobile Computing

among competing flows. However, this *flow-level* fairness ignores application-layer requirements and can result in inefficient resource allocation, or even drastically degrade QoE for an app when flows with conflicting demands compete for the network resources (See Sec. 3). Given this app-agnostic transport-layer design, some content providers focus on application layer enhancement approaches, e.g., YouTube incorporates adaptive bitrate (ABR) streaming to balance video quality and rebuffering rates. Google has deployed a new QUIC [4] protocol, which re-implements most of the transmission control functions in the application layer. However, such approaches require application-specific logic that are not always easy to port to new and emerging apps, and they can give rise to complex interactions between the application and transport layers [5].

Given the above limitations, we believe that the main issue is not the lack of alternative congestion control protocols best suited to different scenarios, but rather an automated mechanism to select the *right* one to maximize the QoE of the active apps on the client device. We design Janus to solve this problem: it is a new transport protocol selection and adaptation framework that enables *automatic* per-flow congestion control protocol selection and optimization according to the app requirements. As shown in Fig.1, Janus allows each app to specify its bandwidth requirement to provide the best-case QoE, and then selects the congestion control algorithm according to the so called *just aggressive enough* principle. This means that Janus will select an *aggressive* congestion control protocol only when the bandwidth requirement of the app is not met, and switch back to a *conservative* one otherwise. This improves overall network performance for two reasons. First it ensures that the QoE requirement of a given app is satisfied quickly, as long as there is enough bandwidth; second it prevents an app from being unnecessarily aggressive after its demand has been met, thus saving resources for other co-existing apps.

1.2 Design Parameters for Janus and Contributions

While the principles behind Janus are straightforward, we identified and addressed three key challenges to enable a flexible, efficient, and fair solution. The first challenge is how to capture the collective needs of various apps. Not all apps have specific bandwidth requirements. For example, delay sensitive apps (e.g., web browsing) need to minimize delay while some other apps may simply needs to maximize the link utilization (e.g., file downloading). The second challenge is how to seamlessly switch between congestion control protocols for each flow, in a way that satisfies multiple apps that are sharing the network with heterogeneous requirements. This is the central problem that Janus needs to solve. Third, Janus must operate when available bandwidth is not sufficient to satisfy the aggregate requirements of all apps. In this case, a fairness principle must be formulated to share the limited bandwidth resources among competing flows.

This paper makes three main contributions:

- We present Janus, a new transport framework that automatically selects different protocols for each flow according to specific app requirements. In this paper, we mainly focus on two representative delay- and loss-based congestion control protocols (Vegas and Cubic) to illustrates the design of Janus. In addition, we show in Sec.7, how to extend Janus with more TCP variants.
- We propose an adaptation policy for selecting and switching congestion control protocols in a way that quickly satisfies app requirements. We also analytically prove that our approach can ensure max-min fairness allocation given limited bandwidth.
- We implement Janus in the Linux kernel and experimentally evaluate its performance with traffic from both emulation and real-world apps. We show Janus provides excellent QoE, relative to other approaches, for practical situations. For example, it reduces rebuffering rates by up to 5X for a video steaming app and reduces page load times by up to 2X for web-browsing apps. It also shortens the queuing latency by up to 10X over Cubic and 5X over PCC, while attaining comparable or higher throughput.

2 Related Work

Many congestion control protocols [6]–[16], [18]–[28] have been proposed to optimize transport protocol performance. For example, Westwood+ [7] uses bandwidth estimation to increase TCP robustness with non-congestion packet loss in wireless networks; Hybla [6] scales cwnd faster in satellite networks with very long round-trip-delay (RTT); Cubic [12] modifies the classical additive increase, multiplicative decrease (AIMD) rule of TCP to quickly saturate the link in high bandwidth-delay product (BDP) networks. Table 1 summarizes the features of several existing congestion control protocols. However, these protocols do not adapt their congestion-control strategy as the network evolves over time, as the server is unable to estimate future last-mile wireless link selections.

Another line of work utilizes both the end-hosts' and in-network knowledge to make better congestion control decisions. For example, FCP [29] combines end-point control and explicit router feedback. [30] further suggests using end-point congestion control, arbitrary rate control and in-network packet prioritization all together to achieve faster flow convergence. However, such protocols present a high barrier to deployment since they require simultaneous support from both end-hosts and intermediate network switches. In comparison, Janus requires only modification at the server side and is effective even in incremental deployment.

Several protocol designs focus on learning how to achieve constant, optimal performance, despite changes in the network conditions. The key idea is to directly search for actions (e.g., increase/decrease cwnd or sending rate) that can maximize objectives (e.g, maximize throughput and minimize delay). Remy [31], [32] uses off-line training to obtain the optimal mapping between network conditions and the cwnd adjusting function. In contrast, the performance oriented congestion control (PCC) protocol [33] uses on-line learning to find the sending rate that can maximize the value of a *utility function* according to the feedback from the receiver in real time. Similar to Janus, PCC allows apps to specify their performance preference (throughput or RTT)

1536-1233 (c) 2018 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TMC.2018.2870156, IEEE Transactions on Mobile Computing

TABLE 1: Different network environments, underlying assumption on link properties and corresponding selected TCP variants

Networks	Underlying Assumptions	Congestion Control Protocols
Satellite	Very long RTT	Hybla [6]
Wireless links	Frequent non-congestion packet loss	Westwood+ [7], Veno [8], Jersey [9]
Sensor Network	Ad-hoc topology, limited power and transmission range	ESRT [10], CODA [11]
Modern Internet	High bandwidth and delay	Cubic [12], HTCP [13], Yeah [14], BBR [15]
Cognitive Radio	Dynamic Bandwidth availability	TP-CRAHN [16], TFRC-CR [17], CogNet [18]
Data Center	In-cast topology, stringent latency constrain	DCTCP [19], TIMELY [20], HULL [21], D^3 [22]
Cellular/WiFi	Highly varying network condition, Buffer-bloating	Sprout [23], CQIC [24], Verus [25]

by choosing a corresponding utility function. We compare Janus with PCC in Sec.8 and find Janus achieves better performance tradeoffs as it can support co-existing flows with unique or even conflicting requirements, while PCC can only optimize for a single, pre-selected performance metric.

Two recent efforts [34], [35] propose the concept of virtualized congestion control. The key idea is to *translate* one congestion control protocol into another by inspecting and modifying packet headers. One important goal of virtualized congestion control is to allow flows using different congestion control protocols (e.g, ECN vs. non-ECN) to share bandwidth fairly with each other. In contrast, Janus proposes a different approach of directly *switching* among congestion control variants, and is more flexible than protocol translation.

The closest work to Janus is OpenTCP [36], where TCP parameters are optimized and adaptively selected among existing TCP variants according to the explicit network information collected in a software defined network (SDN). However, despite the conceptual similarities, there are two key differences between OpenTCP and Janus: (1) OpenTCP works for an SDN environment that allows the controller to get the global view and control of all network traffic, while Janus is designed for mobile network and only requires modification at the server side; (2) While OpenTCP mentions the idea of selecting TCP variants, the actual switching logic and procedures are not introduced. Our work demonstrates that doing so is not trivial, and we present a detailed design of protocol switching algorithms, with an implementation and thorough consideration of the diverse QoE requirements of different apps.

3 Preliminary Results & Problem Definition

The traditional goal of TCP is to share bandwidth fairly among flows. However, the *flow-level* fairness overlooks the actual requirement of various apps. As a result, a flow may behave aggressively and degrade the performance for other flows even when its own QoE requirement has been satisfied. We illustrate the problem with following two examples.

• *Inefficient resource allocation:* To demonstrate the problem of inefficient bandwidth allocation in classical TCP, we set up a video stream proxy server and play two YouTube videos on separate laptops, which were connected to the same WiFi hotspot with 12 Mbps downlink bandwidth. The video content is the same, but each laptop uses a different resolution (720P and 1440P). Theoretically, the total bandwidth is high enough to play both videos together smoothly. However, we find the 1440P video suffers from frequent stalls, and waits for an unacceptably long 30% of the total video time for re-buffering. The reason is shown in Fig. 2: the 1440P video flow only gets half of total bandwidth (6 Mbps, which is smaller than the 1440P bitrate) as TCP tries to allocate bandwidth evenly between two video flows. However, if we artificially limit the 720 P flow to occupy no more than its required share of 2 Mbps, there is no re-buffering for the both videos, since both video flows get the amount of bandwidth share to meet their QoE requirements.

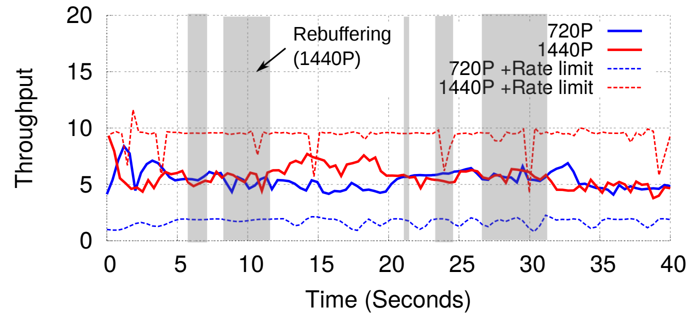

Fig. 2: Throughput of video streaming flows. Without rate limiting, TCP tries to allocate bandwidth evenly among two flows, which results in rebuffering for the 1440P video (shown as gray bars).

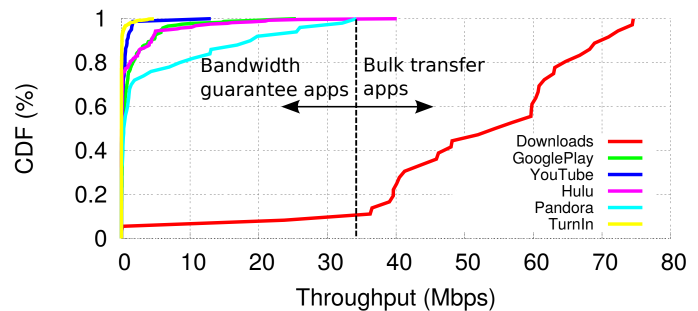

Fig. 3: Average throughput of bandwidth-intensive apps under ideal network conditions. Unlike *bulk transfer* apps, the *bandwidth guarantee* apps do not intend to fully utilize the bandwidth.

• *Managing conflicting requirements in flows:* It is well known that TCP cannot deal with the situation when co-existing flows have conflicting requirements, e.g, when a delay-sensitive flow and bulk transfer flow are competing for the same bottleneck link. To show this effect, we fetch the top 1000 most popular websites [37] automatically and measure the average RTT with/without a long-lived file downloading flow. From Fig.4, we see that 90% of web browsing flows have average RTT less than 100 ms without the file downloading flow, but this number reduces to only 10% in the presence of cross-traffic. This situation arises from a combined effect of choosing a loss-based TCP (Cubic used in this experiment) and its inherent aggressive nature that attempts to saturate the bottleneck link and its queues.

• *Lack of awareness of app QoE:* The main reason for the

1536-1233 (c) 2018 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TMC.2018.2870156, IEEE Transactions on Mobile Computing

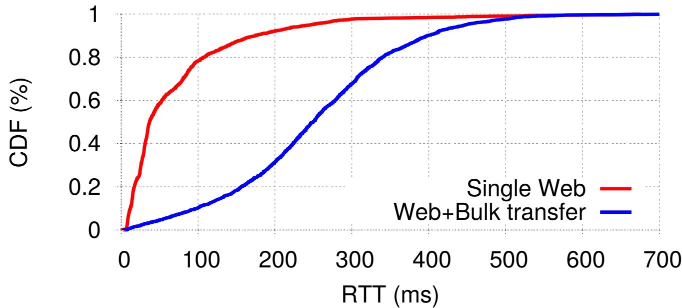

Fig. 4: RTT of web flow with/without background traffic.

TABLE 2: Apps classified by demands

Classes	Requirements	Examples
Bandwidth guarantee	Satisfy Target Rate	Video streaming
Delay sensitive	Minimize RTT	Web Viewing
Bulk transfer	Maximize link utilization	Large file transfer

above problem is that the *flow-level* fairness of TCP does not take actual QoE demands of an app into consideration. To solve this problem we first have to understand better the network requirements of various apps. We organize apps into three classes: *bandwidth guarantee* (e.g., streaming video), *delay sensitive* (e.g., Web traffic) and *bulk transfer* (e.g., file downloads) according to their QoE demands (Table.2). The idea behind the classification is straightforward: *bulk transfer* apps are work-conserving and need as high link utilization as possible, while *delay sensitive* apps need to minimize the round trip latency. For *bandwidth guarantee* apps, the QoE requirements can be met as long as the requested **target rates** are satisfied.

We measure the difference in throughput between *bulk transfer* and *bandwidth guarantee* apps by running six popular bandwidth-intensive apps individually on a Google Nexus 6 smartphone connected to a 100 Mbps WiFi network. Our setup ensures that the wireless link is not the bottleneck and the measured throughput reflects the actual bandwidth requirement for each app. Fig.3 shows a clear distinction between *bandwidth guarantee* and *bulk transfer* apps. While the *bulk transfer* app (`Download`) uses up to 80% of total bandwidth, *bandwidth guarantee* apps use less than 20%. We conclude that unlike *bulk transfer* apps, the *bandwidth guarantee* apps do not need to saturate the network to provide satisfactory QoE.

• *Summary:* The above demonstrations show that different kinds of apps have their own demands of network resources for satisfactory QoE. However, the *flow-level* fairness of traditional TCP will lead to suboptimal QoE as it can neither share bandwidth resources efficiently enough, nor resolve the conflict requirements of co-existing flows from different apps. This motivates our pursuit of a more flexible transport framework that can satisfy the unique requirements for each app by ensuring efficient, fair bandwidth allocation among competing flows.

4 JANUS OVERVIEW AND DESIGN

In Janus, each app to specify a request for network resources in terms of target rate, then Janus automatically configures congestion control protocols on a *per-flow* basis to ensure each app's objective can be satisfied. Fig.1 presents the three main components of Janus: (1) *metrics collection*, which gathers data regarding network conditions and app requirements. (2) *TCP switching engine*, which chooses the appropriate TCP variant for each flow given the metrics collected to ensure that app's objectives can be met; (3) *performance enhancement*, which tunes the default operation of the chosen TCP variant to further improve the performance.

4.1 Overview

At a high level, Janus works as follows: it checks the target rate R of an outgoing flow during the connection establishment stage. For bandwidth-guarantee flows, R simply represents the required bandwidth for the app to provide ideal QoE. We will introduce how to configure target rate later in Sec.7.2. If the flow is not associated with any target rate (e.g, bulk transfer or delay sensitive flow), Janus will first initialize one for the flow according to the current available bandwidth and then adjust it adaptively as network load varies (described in Sec.4.2).

Next, Janus continuously logs the throughput of the flow to detect whether the target rate has been satisfied. If so, Janus will select a conservative congestion control protocol to avoid overwhelming the network. Otherwise, Janus will choose an aggressive protocol to compete for more bandwidth until target rate is reached (Sec.4.3). Ideally, each flow should quickly converge its throughput to the target rate and then stay with the conservative TCP. However, selecting TCP variants naively cannot assure stable performance, especially when flows with distinct demands are sharing the bottleneck link. Therefore, Janus also uses proactive rate control to increase bandwidth sharing efficiency and tries to clear the bottleneck buffer during TCP switching to further minimize the queuing latency (Sec.4.4).

A key problem arises if the total network bandwidth is not enough to satisfy the requirements of all co-existing flows. This is common in mobile networks due to limited wireless capacity. To solve this problem, we develop an adaptive target rate adjustment algorithm to scale down each flow's target rate temporarily until the network condition is recovered. By lowering down each flow's expectation, Janus avoids the race to keep competing for bandwidth and allows the flows to converge to max-min fairness allocation under the limited resources. Sec. 5 and Sec. 6 introduces the algorithm and provides the theoretical analysis, respectively.

4.2 Metrics Collection

This component is responsible for: (1) measuring the network condition and flow performance; (2) reading or initializing target rate for each flow according to each apps' objective. The output of metrics collection will be fed into following blocks as reference to choose appropriate congestion control protocols and further optimize performance.

• **Network measurement:** We use throughput P and round-trip delay RTT as the two key metrics that influence the choice made by Janus. We carefully smooth the collected sample values to capture the meaningful trends within the network and yet avoid rapid fluctuations as follows.

Throughput P: Janus first calculates the sample throughput P_{smp} by dividing the bytes acknowledged in a time window by the window length. Then, an Exponential Weight

1536-1233 (c) 2018 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TMC.2018.2870156, IEEE Transactions on Mobile Computing

Moving Average (EWMA) filter $P_{avg} = \xi * P_{smp} + (1-\xi) * P$ is applied for smoothening. Through empirical measurements, we have observed that the values of $\xi = 1/8$ and one RTT as time window are preferred choices for this smoothing step. This setting is also used in bandwidth estimation techniques in other protocols [7].

RTT: Janus reads the round trip delay sample `tp->srtt` recorded in the TCP/IP stack of the Linux kernel. Note this `tp->srtt` has been previously passed through EWMA filter with $\xi = 1/8$ by the Linux kernel before feeding into Janus. However, we find that `tp->srtt` still suffers from rapid random fluctuations in wireless network. To accurately track the trends in RTT changes while filtering noise, we further pass the `tp->srtt` into a min filter (by taking minimum `tp->srtt` in a time window).

• **Target rate initialization:** For bandwidth guarantee flows, the target rate R should be configured beforehand since it is decided by app's throughput demand. Therefore, Janus can read R directly during the connection establishment time.

Unlike bandwidth guarantee flows, bulk transfer and delay sensitive flows do not have any specific bandwidth requirements. In this case, Janus starts a flow with TCP Vegas and waits for two additional RTTs after the slow start stage finishes. Then it initializes the target rate as the current measured instantaneous throughput ($R = P_{smp}$). Note that the slow start phase of Vegas is conservative, i.e., it finishes as soon as packets start accumulating in the network. The newly joined flow can only fill packets in an under-utilized link. Therefore, the current throughput represents the *available bandwidth* that the new joining flow can use. We choose to wait two RTTs because flow throughput needs a short duration to ramp up after slow start.

Allowing a new flow to utilize available bandwidth yields several benefits. It prevents new flows from starving any bandwidth guarantee flows. It ensures high link utilization for bulk transfer flows, since it makes sure all available bandwidth is utilized. Finally, it prevents a large number of packets from accumulating in the bottleneck buffer, thus reducing queuing latency for delay sensitive flows.

Available bandwidth may change due to varying network conditions such as load, link capacity, etc. Janus uses the target rate adjustment algorithm (described in Sec.5) to reactively tune the target rate for bulk transfer and delay sensitive flows so to match the varying available bandwidth.

4.3 TCP Switching Engine

Congestion control algorithms can be loosely categorized into delay-based and loss-based. Delay-based TCP is considered more *conservative*, as it interprets an increasing delay as a sign of congestion and slows the sending rate down accordingly. On the other hand, loss-based TCP is more *aggressive*, as it aims to saturate the bottleneck link until packets begin to drop because of buffer overflow. Therefore, loss-based TCP usually achieves higher link utilization at the cost of increased queuing delay, while delay-based TCP manages to keep queuing delay stable, though sometimes failing to attain high throughput.

Janus balances the advantages of these two congestion control strategies by switching between a (i) *conservative stage* (i.e., choosing the TCP Vegas [38]) and (ii) *aggressive stage* (i.e., choosing the TCP Cubic [12]) based on the measured throughput and the target rate from the metrics collection block. We choose these two protocols because Vegas is a classical delay-base congestion control protocol and Cubic is one of the default congestion control protocols in Linux. We also discuss how to extend Janus with other TCP variants in Sec.7.

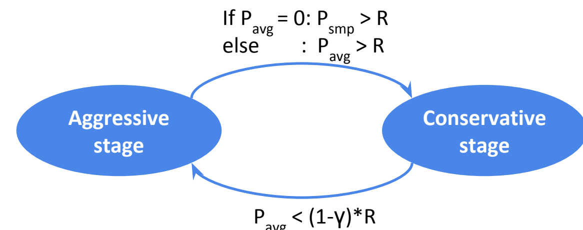

Fig. 5: Two-stage TCP transition diagram.

The stage transition logic is demonstrated in Fig.5: Janus enters the *conservative stage* and switches to Cubic whenever the target rate is satisfied, otherwise, it enters in the *aggressive stage* to compete for more bandwidth. While the switching basic logic is straightforward, we note:

(1) To avoid rapid oscillation between two stages, Janus uses a protection parameter γ that allows this switch to the *aggressive stage* only when $P_{avg} < (1-\gamma) * R$.

(2) Janus uses sample throughput P_{smp} instead of P_{avg} to decide when to transit to the *conservative stage* for a new joining flows, before the first *conservative stage*. This is because the initial value of the P_{avg} is zero when this flow starts, and it takes few RTTs for the smoothed P_{avg} to ramp up until it captures the actual throughput. After the flow enters into the conservative stage, Janus initializes P_{avg} to the current instantaneous throughput and uses P_{avg} to decide the state transition.

An example demonstrating the two-stage transition algorithm is given in Fig.6. We set the bottleneck bandwidth to be $10\,\text{Mbps}$ and start a flow with $8\,\text{Mbps}$ target rate. When the flow just joins in the network, Janus starts with TCP Cubic (*aggressive stage*) to quickly grab bandwidth. After its target rate R has been satisfied (around $0.7\,\text{s}$, $P_{smp} > R$), Janus switches to TCP Vegas (*conservative stage*) to keep queuing delay stable. During bursty cross-traffic ($6-8\,\text{s}$), the throughput drops below $(1-\gamma) * R$ (because of EWMA smoothing, it takes about 1s for P to drop to bottom after cross-traffic leaves), Janus returns to the *aggressive stage* until its throughput recovers (around $10\,\text{s}$, $P_{avg} > R$).

From the Fig.6, we notice that Janus increases `cwnd` faster than the cubic function during *aggressive stage* (in this example, `cwnd` increases by 1.5X in eight RTTs). We find that this is because of an undocumented rule in Cubic's Linux implementation which specifies that `cwnd` is increased by 5% every RTT before the first packet loss. This highlights the need to carefully limit aggression in Janus to avoid starving competing flows. We further discuss the influence of the duration of the *aggressive stage* later in Sec. 5.

4.4 Performance Enhancement

While above simple TCP switching algorithm satisfies the requirement of a single flow, more complex situations may arise when flows with various demands are competing for one bottleneck link. In this section, we discuss the

1536-1233 (c) 2018 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

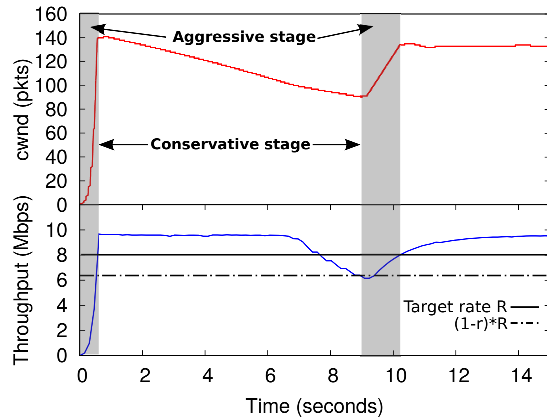

Fig. 6: An example of two-stage TCP switching. Top figure shows the `cwnd` dynamics and bottom figure shows the varying throughput.

enhancements to the basic Janus switching algorithm with two additional mechanisms that (i) speed up convergence and increase bandwidth allocation efficiency by performing proactive rate control, and (ii) minimize the self-inflicted queuing delay during state transition by clearing the bottleneck buffer.

• **Rate control:** Classical TCP constantly probes for available bandwidth and then shares it equally among concurrent flows. Janus increases the overall efficiency in bandwidth utilization by actively controlling the sending rate of *bandwidth guarantee* flows, thus preventing such flows from grabbing resources greater than their demands. This *rate control* is used in the first *aggressive stage* and the subsequent *conservative stage*. Janus does not modify the code of any existing congestion control algorithms; rather it limits the sending rate through (i) slow start threshold setting `snd_ssthresh` and (ii) maximum congestion window `cwnd_clamp` present in the Linux kernel.

Aggressive stage: For a *bandwidth guarantee* flow, Janus initially chooses Cubic with the purpose of satisfying the target rate R as soon as possible. It overwrites the `snd_ssthresh` to $R * D_p$, where D_p is the round-trip propagation delay (we introduce how to estimate D_p later in this section). Therefore, $R * D_p$ is simply the number of outstanding packets required by the flow to reach its target rate. After `cwnd` reaches `snd_ssthresh`, if the target rate is still not satisfied, Janus continues the normal congestion avoidance operation. By resetting the `snd_ssthresh`, Janus significantly improves the convergence speed by making sure that the newly joined flow's target rate can be reached as soon as possible, regardless of the end-to-end propagation delay of the connection.

Conservative stage: During the *conservative stage*, Janus limits the throughput P of a *bandwidth guarantee* flow to $(1+\varepsilon) * R$, where ε decides the extra bandwidth potentially obtained by each flow. For this, Janus sets the maximum `cwnd` to $(1+\varepsilon) * R * RTT$ when $P_{avg} > (1+\varepsilon) * R$, and removes this limit when $P_{avg} < R$. This ensures that the throughput of the *bandwidth guarantee* flow stays between R and $(1+\varepsilon) * R$, thus preventing the flow from trying to compete for more bandwidth than it demands.

• **Buffer clearing:** For *delay sensitive* apps, the goal of minimizing queuing delay cannot be achieved in isolation by a given end-device. This is because different flows may have accumulated packets in the network, making it impossible for other flows to further reduce delay. Therefore, Janus forces every flow to minimize the *self-inflicted* queuing delay. Unfortunately, following the *aggressive stage*, flows accumulate many packets in the bottleneck buffer. As Vegas can only reduce one outstanding packet in every RTT duration, it takes a while for Vegas to compensate for the increased queuing delay incurred during the *aggressive stage* (Fig.6, $0.7-9$ s). Furthermore, Vegas may regard the extra queuing delay as part of the link latency and not attempt to reduce outstanding packets at all (after 10 s).

To solve this problem, Janus cuts the `cwnd` to $P_{avg} * D_p$ every time before switching to the *conservative stage* so that the flow's own outstanding packets is just large enough to maintain its current throughput. This allows the depletion of packets in the bottleneck buffer, thus minimizing the queuing latency during the transition.

• **Estimating propagation delay:** The success of both rate control and buffer clearing requires accurate estimation of propagation delay D_p. Janus regards the minimum RTT observed (`base_rtt`) during the connection lifetime as estimated propagation delay. However, it is well known that delay based TCP like Vegas suffers from the so called $re-routing$ problem [39]. This happens when link latency varies because of changing routes or user mobility. To address this issue, Janus updates D_p when it enters a new *conservative stage* as:

$$D_p = RTT_{min} - \frac{Q_{avg}}{P_{avg}} \tag{1}$$

RTT_{min} is the minimum RTT sample taken at the start of a *conservative stage*. Q_{avg} is the average queued packets a conservative TCP will try to maintain in the bottleneck link buffer. Janus estimates D_P by subtracting RTT_{min} from $\frac{Q_{avg}}{P_{avg}}$, where the latter is the extra delay caused by average queuing packets of a conservative TCP. As we show in Sec.8, this allows Janus to accurately track the propagation delay despite the varying link latency in a mobile network.

5 Adaptive Target Rate Adjustment

When the cumulative bandwidth demands of all active flows exceeds the available network capacity, one or more flows will stay in the *aggressive stage* because the required target rate cannot be satisfied. This starts a race to consume bandwidth that forces other flows to also switch to Cubic, leading to higher queuing delay and packet loss rate. In this section, we introduce an adaptive target rate adjusting algorithm to solve this problem. The key idea is that Janus intelligently lowers the target rate of the flows given the limited bandwidth so that network can still converge. We start with the goal of adjusting the target rate, then describe the algorithm and configuration of the parameters in detail.

5.1 Goal of target rate adjustment

Target rate adjusting aims to re-allocate the limited bandwidth to flows by striking a balance between efficiency and fairness. Janus uses the max-min fairness (MMF) allocation principle [40], which implies: (1) flows with lower demand

1536-1233 (c) 2018 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TMC.2018.2870156, IEEE Transactions on Mobile Computing

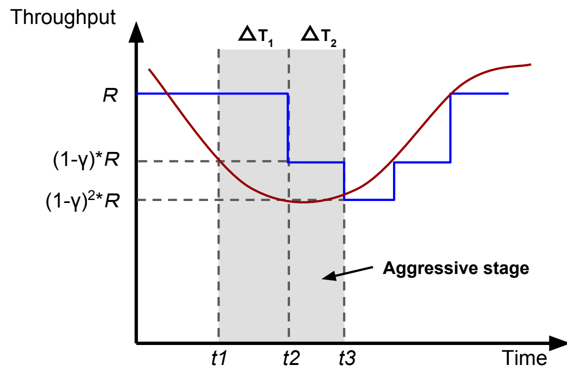

Fig. 7: An example of target rate adjustment algorithm: The flow enters the *aggressive stage* at $t1$, and reduces target rate twice at $t2$ and $t3$, respectively. Then it leaves the *aggressive stage* because its throughput exceeds the adjusted target rate. Finally, in the *conservative stage* the flow gradually recovers its throughput as well as target rate.

have higher priority in bandwidth allocation. This prevents a single flow with large target rate from overwhelming the network; (2) each flow only gets the resources it needs, thus guaranteeing efficient utilization of bandwidth; (3) equally allocating spare bandwidth to those flows whose demands are not met. The easiest way to achieve this goal is to compute the MMF allocation and assign the rate appropriately for each flow. However, this approach requires an unacceptable centralized control. Hence, we next introduce a lightweight algorithm to achieve the MMF bandwidth allocation by adjusting target rate in a distributed manner.

5.2 Algorithm for adaptive target rate adjustment

A Janus flow tunes its own target rate using only local information with Alg.1. Intuitively, if a flow stays in *aggressive stage* for too long, and queuing delay is increased during this stage, then it implies that (1) the bottleneck link is saturated, and (2) the current bandwidth is not enough to support the flow's target rate. In this case, the flow should reduce its target rate. Otherwise, if a flow is in the *conservative stage* with increasing throughput, then the flow may raise its target rate as surplus bandwidth is available.

A rigorous proof of MMF allocation with Alg.1 is provided in Sec.6. Here we explain the key steps of the algorithm: Janus measures the time elapsed $time_diff$ since entering the *aggressive stage* (line 3). If the flow's throughput recovers to its original target rate R in interval ΔT ($time_diff < \Delta T$), Janus leaves the *aggressive stage* without reducing R. Otherwise, Janus checks whether the bottleneck link is saturated by evaluating if RTT has increased over threshold $\sigma(RTT)$, where σ denotes the standard deviation (line 4). If so, Janus reduces its target rate to $(1-\gamma)*R'$, where R' is its current target rate and γ is the same margin parameter introduced in Sec. 4.3 (lines 9-10). Notice that the decrease in the adjusted target rate is bounded by minimum target rate R_{min}. The above iteration is repeated until the throughput of a given flow exceeds its adjusted target rate, concluding the *aggressive stage*.

In *conservative stage*, if the throughput of flow increases beyond $R'/(1-\gamma)$, then Janus raises its current target rate correspondingly (lines 13-15). For *bandwidth guarantee* flows,

Algorithm 1: Adaptive Target Rate Adjusting

Input: P: throughput; R: original target rate
Output: R': adjusted target rate
1 $R' = R$; $\Delta T = 0$; $prev_adjust_time =$ `Now time`; $prev_rtt = RTT$;
2 **while** *in aggressive stage* **do**
3 $time_diff$ = `Now time` - $prev_adjust_time$
4 rtt_diff = RTT - $prev_rtt$
5 **if** $\Delta T = 0$ **then**
6 $\Delta T = \tau * \frac{\gamma}{1-\gamma} * RTT$
7 **else**
8 $\Delta T = 2 * RTT$
9 **if** $time_diff > \Delta T$ *and* $rtt_diff > \sigma($`RTT`$)$ **then**
10 $R' = max\{(1-\gamma)*R', R_{min}\}$
11 $prev_adjust_time =$ `Now time`
12 $prev_rtt = RTT$
13 **while** *in conservative stage* **do**
14 **if** $P > R'/(1-\gamma)$ *and* $R' \neq R$ **then**
15 $R' = R'/(1-\gamma)$
16 **return** R'

the increase of R' is upper bounded by its original target rate R. However, there is no such limitation for *bulk transfer* flows since its original target rate $R = 0$. This allows bulk transfer flows to keep increasing R' until full utilization of the bandwidth. An example showing the entire process of reducing/recovering target rate is given in Fig.7.

Finally, the time interval ΔT greatly influences Janus's performance. A large ΔT allows a flow to stay longer in the *aggressive stage* and may potentially degrade the performance of competing flows. On the other hand, a small ΔT may unnecessarily reduce the target rate. Hence, we set ΔT in an adaptive manner, where the first $\Delta T_1 = \tau * \frac{\gamma}{1-\gamma} * RTT$ and the following $\Delta T_i = 2 * RTT$ ($i \geq 2$). Here τ is a scaling factor controlling the duration of ΔT_1. We will discuss the ideal setting of τ in Sec.6 and Sec.7. If after ΔT_1 the target rate is not yet satisfied, Janus reduces the interval to $2 * RTT$. This avoids hurting the performance of other flows by forcing a departure from the *aggressive stage* as soon as possible.

6 Max-Min Convergence Analysis

In this section, we analytically prove that multiple Janus flows converge to MMF bandwidth allocation using the distributed target rate adjusting algorithm under assumptions of limited network bandwidth. We note that this proof is not limited to Vegas and Cubic; rather it holds true for general delay-based and loss-based congestion control protocols.

• **Hypothesis statement:** Consider N Janus flows sharing a single bottleneck link with capacity C. Packets arriving at the bottleneck router are served using the first in first out (FIFO) principle. $\vec{R} = \{R_1...R_N\}$ is the target rate vector, where R_i denotes the requested target rate for flow i. Assuming network capacity is less than the total demand of all flows, i.e., ($\sum_{i=1}^{N} R_i > C$), Alg.1 results in MMF allocation of bandwidth.

We prove this hypothesis using a simplified version of Janus with only the *conservative stage*. We then extend the

1536-1233 (c) 2018 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TMC.2018.2870156, IEEE Transactions on Mobile Computing

analysis for the full-featured Janus, and show that the additional *aggressive stage* does not affect the MMF allocation.

6.1 Simplified Janus with only conservative stage

Consider the throughput vector $\vec{P} = \{P_1 ... P_N\}$ and the target rate vector $\vec{R'} = \{R'_1 ... R'_N\}$ after the system converges, where P_i and R'_i are throughput and adjusted target rate of flow i, respectively. We start with a simplified Janus with only the *conservative stage*. Therefore, if a flow's throughput drops below $(1-\gamma)*R$, it will reset its target rate directly to $(1-\gamma)*R$ without switching to TCP Cubic. Additionally, the rate control technique requires the flow to limit its throughput under $(1+\varepsilon)*R$.

The proof of MMF allocation for Janus is based on the property that multiple delay-based flows converge to fair bandwidth allocation, which is a basic requirement for any congestion control protocol (e.g., see [39] for the proof for Vegas).Therefore, without considering *rate control*, each Janus flow converges to the fair share C/N in a finite amount of time. However, C/N may be larger then the actual target rates of some flows. Assume flow i has target rate $R_i < C/N$. In this case, the *rate control* limits its rate to $[R_i, (1+\varepsilon)*R_i)$. Then, other flows start to compete for the bandwidth spared by flow i. This process repeats until the remaining bandwidth can no longer be fairly distributed to meet the target rate of the left-over flows.

After the system converges, we separate flows into two different sets: Φ_1 with satisfied demands and Φ_2 that includes the rest. The throughput of flows in these two sets are: $\forall i \in \Phi_1$: $P_i \in [R_i, (1+\varepsilon)*R_i)$, $\forall j \in \Phi_2$: $P_j \in [\frac{C-(1+\varepsilon)*\sum_{i\in\Phi_1} R_i}{\|\Phi_2\|}, \frac{C-\sum_{i\in\Phi_1} R_i}{\|\Phi_2\|})$. Specifically, as $\varepsilon \to 0$, $\forall i \in \Phi_1$: $P_i \to R_i$; $\forall j \in \Phi_2$: $P_j \to \frac{C-\sum_{i\in\Phi_1} R_i}{\|\Phi_2\|}$.

Conversely, we can derive the target rate vector R' from the throughput vector P'. Since the throughput of an unsatisfied flow must lie between $[(1-\gamma)*R', R'/(1-\gamma)]$ after the system converges, we have $\forall i \in \Phi_1$: $R'_i = R_i$; $\forall j \in \Phi_2$: $R'_j \in [(1-\alpha)*P_j, P_j/(1-\alpha)]$, where $P_j \to \frac{C-\sum_{i\in\Phi_1} R_i}{\|\Phi_2\|}$ as $\varepsilon \to 0$.

Intuitively, this means that (1) flows in Φ_1 receive a share just equal to their demand, and hence, they maintain their original target rate; (2) flows in Φ_2 get the fair share of the residual bandwidth but must reduce their target rate correspondingly. Also, since only flows with target rate less than fair share will be limited, we have $\forall i \in \Phi_1, \forall j \in \Phi_2, R_i < R_j$. As all three MMF principles are satisfied (Sec.5.1), the final bandwidth allocation must follow MMF.

6.2 Full-featured Janus

We now extend the analysis to show that the complete Janus protocol (with the *aggressive stage*) does not deviate from the MMF allocation. We first introduce following lemma:

Lemma 1: If the bottleneck link is fully utilized, full-featured Janus flows cannot leave the aggressive stage without reducing their target rate within ΔT_1 if $\tau < \frac{1}{\alpha}$. α is the increment of `cwnd` *in every RTT of the aggressive TCP.*

Proof: We need to prove that a flow can only utilize residual bandwidth after needs of all flows have been considered, and it cannot steal bandwidth from competing flows during the *aggressive stage*. Assume flow i's throughput drops to $P_i = (1-\gamma)*R_i$. This triggers an *aggressive stage* and Janus switches to Cubic for flow i. As a result, flow i can leave the *aggressive stage* without reducing target rate only if its throughput P recovers back to R_i within ΔT_1. We prove that this is impossible if the bottleneck link is fully utilized.

Let W denote the flow i's `cwnd` at the point of entering the *aggressive stage*, which results in $W = P_i * RTT$. Since during the *aggressive stage*, `cwnd` is increased by a factor of α every RTT (Sec.4.3). So after ΔT_1, `cwnd` increases to:

$$W' = W * (1+\alpha)^{\Delta T_1/RTT} \tag{2}$$

Using Taylor expansion and ignoring higher-degree components, W' simplifies to:

$$W' = W * (1 + \alpha * \frac{\Delta T_1}{RTT}) \tag{3}$$

Inserting $\Delta T_1 = \tau * \frac{\gamma}{1-\gamma} * RTT$ (from Alg. 1), we get:

$$W' = W * (1 + \alpha\tau * \frac{\gamma}{1-\gamma}) \tag{4}$$

Therefore, if $\tau < \frac{1}{\alpha}$, then $W' < W/(1-\gamma)$. Also, note that the current RTT' is larger than RTT because of the aggressive behavior of flow i. So, after ΔT_1, the throughput of flow i will be bounded by:

$$P'_i = W'/RTT' < \frac{W}{(1-\gamma)*RTT} = \frac{P_i}{1-\gamma}, \tag{5}$$

The main takeaway is that if we set τ less than the inverse of the increment `cwnd` every RTT during *aggressive stage*, then flow $i's$ throughput cannot increase to $P_i/(1-\gamma) = R_i$. Thus, it leaves the aggressive stage without reducing target rate.

Lemma 1 explains why the MMF allocation property is still valid. Only flows in Φ_2 may potentially enter the *aggressive stage* because they have unsatisfied target rate and must compete for the residual bandwidth. However, *Lemma 1* indicates that they cannot maintain high target rate during the *aggressive stage*. At the end, their throughput and target rate will converge to the range specified in Sec. 6.1.

Finally, we point out that minimum target rate R_{min} will influence the MMF bandwidth allocation. If a flow sets R_{min} too large (e.g., equal with R), then it could maintain its own high target rate by forcing down other flows to stay within lower target rate. Therefore, R_{min} should be small enough to ensure that the throughput of all flows can fall in the range specified by their MMF share. Specifically, R_{min} must satisfy: $\forall j \in \Phi_2$, $R_{min} < P_j/(1-\alpha)$, where $P_j \to \frac{C-\sum_{i\in\Phi_1} R_i}{\|\Phi_2\|}$ as $\varepsilon \to 0$ to maintain the MMF allocation.

7 Janus Implementation

We implement Janus as an kernel module in the TCP/IP stack based on Linux 4.2.6. In this section, we briefly describe the high-level context for our implementation. We also discuss the parameter configuration and the extension to other congestion control protocols.

1536-1233 (c) 2018 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TMC.2018.2870156, IEEE Transactions on Mobile Computing

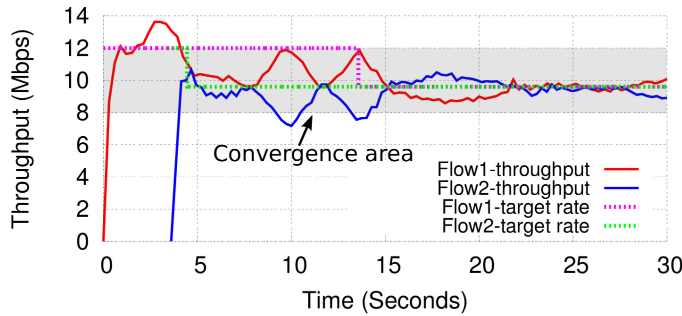

Fig. 8: The convergence of Janus-*reno* flows.

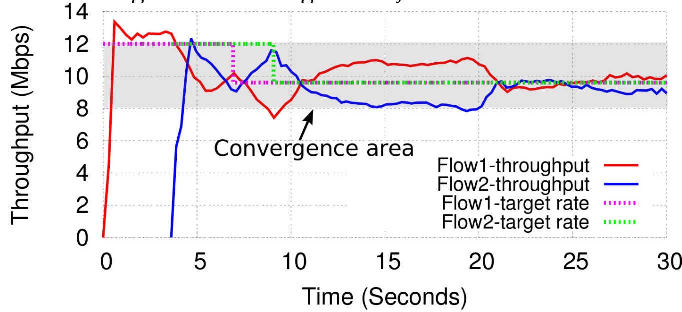

Fig. 9: The convergence of Janus-*htcp* flows.

7.1 Implementation overview

Our implementation only requires minimum modification of the kernel (∼20 lines of code). Once receiving a new packet, Janus reads the TCP socket information `tcp_sock` associated with current flow, and then feeds the `tcp_sock` into the metrics collections module to update the estimation of the throughput and RTT.

TCP switching is achieved by calling the `tcp_set_congestion_control` function defined in `tcp_cong.cc` without interrupting the active TCP connection. We modify the Linux kernel to allow Janus get access to this function by adding the latter into the exported kernel symbols. Janus automatically falls back to the default TCP (NewReno or Cubic) if the switching fails (e.g., the selected TCP does not exist in the current kernel).

• **Janus API:** Janus accepts the desired target rate R (and optional minimum target rate R_{min}) from the application layer as input and configures the congestion control protocols according to Sec. 4.3. As described in Sec.3, target rate represents the desired bandwidth for the app to provide satisfactory QoE to users, e.g., the target and minimum rate for watching 1080P@60fps YouTube video is 18 Mbps and 9 Mbps, respectively [41]. We add two socket options `SO_TARGET_RATE` and `SO_MIN_TARGET_RATE` to the TCP/IP stack of Linux kernel so that R and R_{min} can be specified with the kernel function `setsockopt` at the time of instantiating a connection socket. We provide a `config` interface file to enable Janus to support unmodified apps seamlessly. In the `config` file, the tuple <*source port:*R/R_{min}> can be specified and fed to Janus.

7.2 Parameter configuration

Janus requires 4 parameters to be configured: (1): γ, ε: margin parameter of throughput used in TCP switching and rate control respectively; (3) Q_{avg}: average queued packets maintained by conservative TCP, which is used in propagation delay estimation; (4) τ: the scaling parameter used in target rate adjusting.

• *Configure* γ, ε: These two parameters are protocol-independent, i.e, they do not vary with different selected TCP variants. We empirically set $\gamma = 0.2$ and $\varepsilon = 0.1$, which is also the default setting in following evaluation.

• *Configure* Q_{avg}: The conservative TCP usually maintains a certain number of packets in the bottleneck buffer to make sure the link is fully utilized. Q_{avg} is ideally set as the default queued packets of conservative TCP. For example, Vegas bounds the buffer size between 2 and 6 packets, so Q_{avg} is 4.

• *Configure* τ: As we proved in Sec.6, the minimal setting of τ should the the $\frac{1}{\alpha}$, where α denotes the `cwnd` increment of every RTT with aggressive TCP. For example, we set τ as 20 if Cubic is chosen since `cwnd` is increased by 5% every RTT during *aggressive stage*. Also, we recommend $\tau = 20$ when it is difficult to decide the accurate increment of `cwnd` for other TCP variants (as long as the selected TCP is less aggressive than Cubic). The main reason is that τ balances the aggressiveness of Janus. The larger τ is, the longer Janus stays in the *aggressive stage*, which can negatively influence other flows. However, setting τ too small may force Janus to unnecessarily reduce the target rate. Setting $\tau = 20$ and $\gamma = 0.2$ means that Janus stays in *aggressive stage* for *at least* 5 RTTs before reducing the target rate.

7.3 Extension to other TCP variants

Janus supports other TCP implementations in the server kernel through a *plug-and-play* mode, beyond Vagas and Cubic, which were selected to illustrate its operation in this paper. We provide two additional examples of Janus with the widely used TCP NewReno (Janus-*reno*) [39] and H-TCP (Janus-*htcp*) [13], designed for high-speed and high-latency networks.

To show that Janus still converges with these new loss based TCP variants, we let two flows with target rate $R = 12$ Mbps compete for the bottleneck link with 20 Mbps bandwidth. We use the default parameter settings as introduced in Sec.7.2 ($\gamma = 0.2$, $\varepsilon = 0.1$, $Q_{avg} = 4$ and $\tau = 20$). As shown in Fig.8 and Fig.9, both Janus-*reno* and Janus-*htcp* adjust the target rate for each flow and converge to the fair share of the bandwidth. However, it takes longer for Janus-*reno* to converge because NewReno increases `cwnd` slower than H-TCP in the *aggressive stage*. Besides, note the conservative TCP (Vegas) did not allocate the bandwidth exactly in a fair share (10 - 20 seconds). However, Janus ensures the throughput of these two flows remains within the convergence range of $[(1-\gamma)*R, (1+\gamma)*R]$. In fact, when flow 2 drops out at 20 seconds (Fig.9), it quickly switches to H-TCP and gets back the fair share of the bandwidth within 2 seconds. This experiment not only shows that Janus can be easily extended to include more TCP variants, but also indicates that the Janus actually improves the stability and fairness over single TCP configurations.

8 Experimental Evaluation

We implement Janus and evaluate its performance on a testbed with traffic from emulated workloads as well as real world apps. We wish to show that Janus:

1536-1233 (c) 2018 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TMC.2018.2870156, IEEE Transactions on Mobile Computing

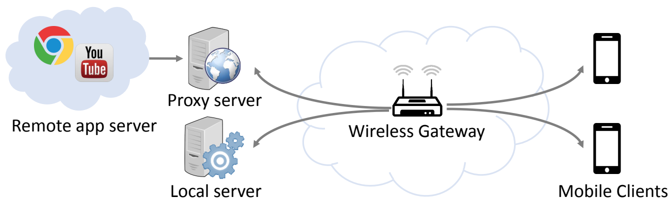

Fig. 10: Our experimental setup includes a local server that generates emulated traffic and a proxy used to redirect Internet traffic. The topology represents a typical dumbbell-like network where multiple flows compete for the same bottleneck link.

1) efficiently shares network resources while satisfying various demands for *bandwidth guarantee* flows, (Sec.8.1);
2) minimizes queuing latency for delay sensitive flows without sacrificing the throughput of background bulk transfer flows (Sec.8.2);
3) achieves high and fair link utilization for *bulk transfer* flows when used as the default congestion control algorithm (Sec.8.3);

In addition, we show how Janus is *robust* against varying link latency and *fair* when competing with heterogeneous TCP flows in Sec.8.4 and Sec.8.5, respectively.

Testbed description: As shown in Fig.10, our testbed contains a local server connected to a wireless gateway router (both running Linux Ubuntu 14.04) via a $1\,\mathrm{Gbps}$ Ethernet link. We use the Linux Traffic Controller tool [42] on the wireless gateway for controling the bandwidth, link latency and buffer size. Thus the wireless link is the bottleneck of the connection between servers and clients. The experimental topology represents a classical dumbbell-like network that widely used in previous research [31], where multiple flows compete for the same bottleneck link.

We use four Android HTC Desire 610 phones running Android 4.4 and two Dell laptops running Ubuntu 14.04 as the clients. The server generates multiple flows for different end-clients through a traffic generator that we have developed. In addition, to study the effect of Janus on real-world apps, we deploy a proxy to redirect the Internet traffic from the remote app servers (such as YouTube and Chrome).

Competing protocols: In our evaluation, we mainly use Cubic as the base-line protocol as it is one of the most most widely used congestion control protocols (e.g., Google also uses Cubic in QUIC at the time of writing). We also compare Janus with PCC [33]: a state-of-art performance oriented congestion control protocol that is built atop UDP. Similar to Janus, PCC also aims to satisfy app-specific demands by selecting different utility functions that optimize either throughput or RTT. However, the selection of the utility function is predefined and cannot be changed during runtime. In the following evaluation, we refer to the PCC with throughput maximization utility function as PCC-*tp* and RTT minimization as PCC-*rtt*.

8.1 Satisfying target rates for bandwidth guarantee flows

We first study the convergence of Janus under varying network loads and show that our protocol can satisfy the bandwidth demands for each flow quickly. Then we validate our model in Sec.6 by showing that the Janus flows converge to MMF allocation under limited bandwidth.

8.1.1 Convergence studies

We schedule four *bandwidth guarantee* flows with different target rates R $(10, 8, 6, 4\,\mathrm{Mbps})$ that enter and leave the network successively, while limiting the bottleneck bandwidth to $25\,\mathrm{Mbps}$. The connection RTT is $60\,\mathrm{ms}$ and the bottleneck link buffer size is 200 packets (same RTT and buffer size setting is used unless otherwise stated). We use the *satisfaction degree* as the performance metric, which is defined as received throughput divided by target rate (P/R). Intuitively, satisfaction degree < 1 means the flow demand has not yet been satisfied. In contrast, satisfaction degree larger than one means the flow has received a larger share than it needs, which is also undesirable as it may hurt the performance for competing flows. Note that for flows without specified bandwidth requirements (Cubic and PCC), the satisfaction degree is simply the received throughput divided by its fair share of total bandwidth.

As shown in Fig.11(a), Janus allows each flow to grab the needed bandwidth and reach the satisfaction degree if there is residual capacity. After the last flow comes in at $32\,\mathrm{s}$, we see only flows 3 ($R = 6\,\mathrm{Mbps}$) and 4 ($R = 4\,\mathrm{Mbps}$) can be satisfied. Note the fair allocation for each flow is 6.25 Mbps. The adaptive target rate adjustment algorithm guarantees that only flows that request bandwidth less than their fair share can be satisfied given limited resources. By comparison, TCP Cubic (Fig.11(b)) tries to allocate bandwidth evenly among all the flows but suffers from longer convergence time (almost 20 seconds). Lastly, in PCC, the first flow takes over 70% of the total bandwidth, leaving only 30% for other flows (Fig.11(c)), thus lead to large deviation in the satisfaction degree.

8.1.2 Analytical model validation

To validate our model in Sec.6, we start the same four *bandwidth guarantee* flows sequentially with a synchronized departure at $60\,\mathrm{s}$, and plot results for two different bottleneck bandwidths- $25\,\mathrm{Mbps}$ and $20\,\mathrm{Mbps}$. Fig. 12 shows that the observed average throughput closely matches with the analytic MMF bandwidth (Sec. 6). We notice flows with target rate equal to the MMF share (e.g., $4\,\mathrm{Mbps}$) have slightly higher throughput than the theoretical value. This is because Janus attempts to achieve an extra $\varepsilon * R$ bandwidth (Sec.4.4) after its target rate has been reached.

8.1.3 Comparison with Google's TCP for YouTube

To demonstrate the QoE improvement for real-world apps, we play the same 5-minute YouTube video simultaneously at different resolutions ($1080\,\mathrm{P}$ and $1440\,\mathrm{P}$, with target rates $4\,\mathrm{Mbps}$ and $10\,\mathrm{Mbps}$ respectively). This represents the scenario of two *bandwidth guarantee* flows sharing one bottleneck link ($15\,\mathrm{Mbps}$). We count the number of rebuffering events, total rebuffering duration (seconds) and the start up latency (seconds) for the $1440\,\mathrm{P}$ flow with (1) Janus (2) Cubic, and (3) the default TCP protocol that Google runs on the origin server (we omit the $1080\,\mathrm{P}$ result here because there is no rebuffering observed). To ensure a fair comparison, we confirm that the extra round trip delay between the Google frontend and our video streaming proxy is less than $2\,\mathrm{ms}$.

1536-1233 (c) 2018 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TMC.2018.2870156, IEEE Transactions on Mobile Computing

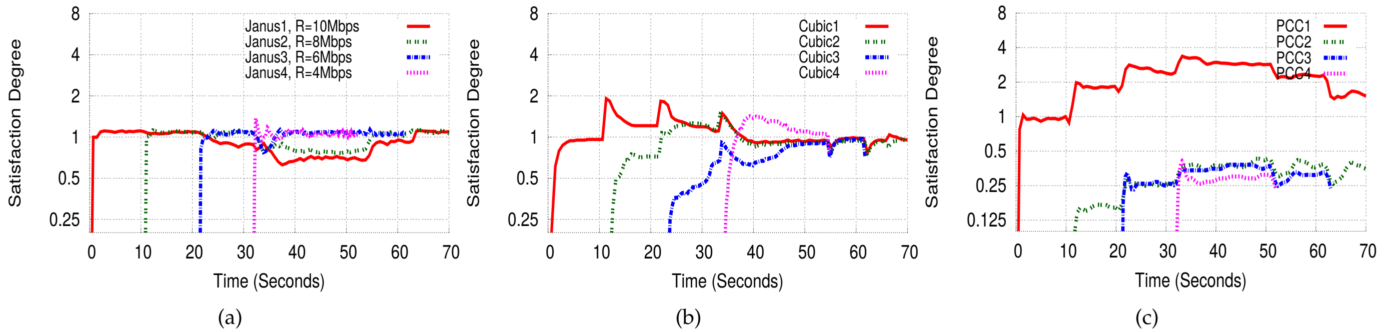

Fig. 11: Convergence dynamics of satisfaction degree (throughput/target rate, ideal value is 1) for four flows with (a) Janus (b) Cubic and (c) PCC-*tp*. Janus allow flows to quickly grab bandwidth according to the target rate before the bottleneck link is saturated and achieves the fastest convergence time when compared with Cubic and PCC.

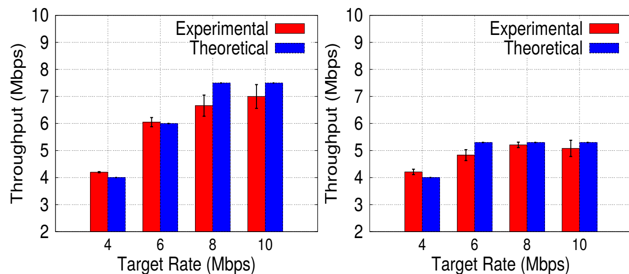

Fig. 12: Flows converge to MMF allocation with bandwidth 25 Mbps (left) and 20 Mbps (right).

TABLE 3: The QoE metrics for 1440P YouTube video with Janus and alternative protocols.

Protocol	Rebuffering number	Rebuffering duration	Startup latency
Janus	6	4.2	3.6
Cubic	21 (3.5X)	31.9 (7.6X)	3.5 (0.98X)
Unmodified	12 (2X)	22.3 (5.3X)	2.7 (0.75X)

As shown in Table. 3, Janus greatly improves the QoE for 1440 P video compared with Cubic and interestingly, even Google's own default protocol. It achieves upto 5X reduction in number of rebuffering events and 7.6X lowering of the re-buffering duration. The improvement occurs when Janus allocates more bandwidth for the 1440 P flow with explicit target rate setting. Note that this does not sacrifice the QoE for 1080 P video since there is no rebuffering observed. This implies users can get satisfactory QoE by sharing bandwidth more efficiently, which is an important design criterion for Janus (Sec. 3).

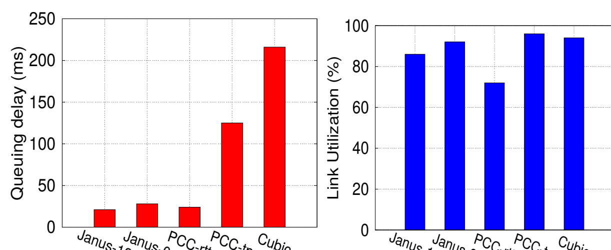

Fig. 13: 95th percentile queuing delay of emulated web traffic (left) and average throughput for background traffic (right).

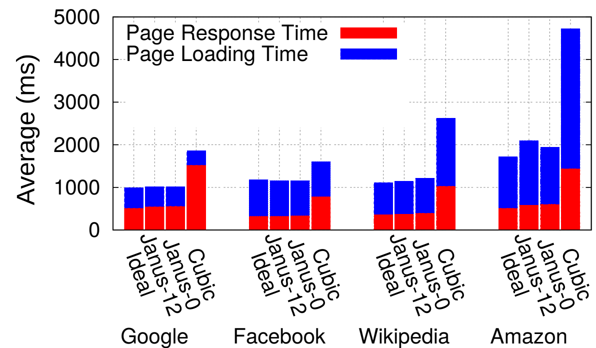

Fig. 14: Average page load time for four popular websites without/with background traffic.

8.2 Minimize latency for delay-sensitive flows

In this section, we study the reduction in queuing latency for *delay sensitive* flows with Janus and its effects on QoE for web-browsing applications.

8.2.1 Reducing queuing latency

We emulate a web workload by sending short, bursty flows of 100 KB every two seconds. Meanwhile, we compare the 95th-percentile queuing delay for emulated web flows in presence of a background long lived flow with five protocol settings: Janus-12 (*bandwidth guarantee* flow with 12 Mbps target rate), Janus-0 (*bulk transfer* flow without any target rate), PCC-*tp*, PCC-*rtt* and Cubic. The bottleneck link bandwidth is set to 15 Mbps.

From Fig.13(left), we see that Janus achieves almost an order of magnitude lower queuing latency than that of Cubic. Additionally, this does not sacrifice the performance of background traffic as shown in Fig.13(right), where the target rate of bandwidth guarantee flow is satisfied and the bulk transfer flow achieves the over 90% of the link utilization. This indicates that Janus solves the conflicting requirements problem (see Sec. 3) by preventing bulk transfer flows from overwhelming the bottleneck buffer.

We now compare the performance of Janus and PCC. We find that PCC-*rtt* achieves almost the same queuing delay as Janus, but it does so by sacrificing the throughput of competing flows (as shown in Fig.13(right)). With PCC-*rtt* the background traffic utilizes only about 73% of the link capacity. In contrast, while PCC-*tp* achieves high throughput, the queuing delay is nearly 5X greater than that of Janus. In summary, PCC requires explicit tradeoffs between

1536-1233 (c) 2018 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TMC.2018.2870156, IEEE Transactions on Mobile Computing

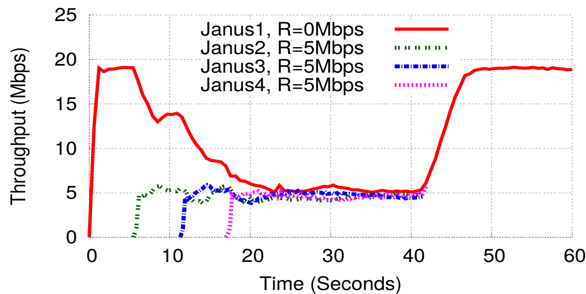

Fig. 15: Convergence dynamics of one *bulk transfer* and three *bandwidth guarantee* flows.

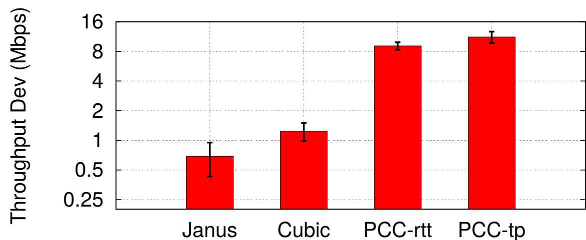

Fig. 16: Throughput deviation of four flows with different protocols.

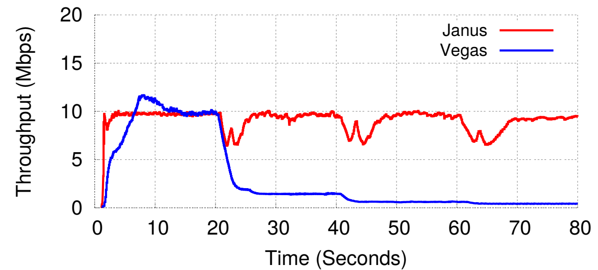

Fig. 17: Janus increases robustness to varying propagation delay (increases from 40ms to 100ms during the connection) when compared to Vegas.

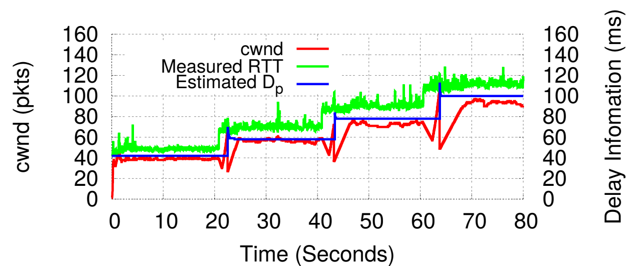

Fig. 18: Variation of cwnd and delay of Janus when link latency increases (40 - 100ms). Janus accurately estimates the propagation delay D_p.

throughput and RTT, while Janus can satisfy various QoE requirements for different apps *simultaneously*.

8.2.2 *Improving page loading time*

We use web browsing to demonstrate QoE improvement for delay-sensitive apps using Janus. We implement an Android app that opens websites and collects Page Response Time (PRT, defined as the time until client receives last byte of response from server) and Page Load Time (PLT, defined as the time until entire page is loaded) via the Navigation Timing API [43]. We then use a Janus proxy to host four websites (Google, Facebook, Wikipedia, Amazon), which represent different types of network services (search, social, wiki and shopping). We compared the average PRT and PLT in presence of a background traffic with the same protocol setting as in the previous experiment (except PCC, since PCC can only be used to serve self-generated traffic).

As shown in Fig.14, Janus achieves almost ideal PRT and PLT as if there is no background traffic. This is because Janus tries to minimize queuing latency, which directly lowers the PRT and PLT for web pages with small objects. By comparison, Cubic incurs large queues at the bottleneck link, and leads to more than 2X larger PRT and PLT.

8.3 Link utilization with bulk transfer flows

In Sec.8.2, we showed that Janus can ensure high link utilization for a single bulk transfer flow in the presence of short, delay-sensitive flows. We now study two more scenarios with: (1) a mixed environment of bulk-transfer and bandwidth-guarantee flows; (2) all bulk transfer flows competing for the same bottleneck link.

8.3.1 *Mixed classes of flows*

We first evaluate whether a bulk transfer flow can coexist fairly with bandwidth guarantee flows. As shown in Fig.15, after one bulk transfer flow fully saturates the bottleneck link, we start three bandwidth-guarantee flows ($R = 5\,\mathrm{Mbps}$) in succession. Note that the initial bulk transfer flow relinquishes bandwidth for the subsequent bandwidth-guarantee flows, until they reach fair share of the total bandwidth (from 20- to 40s). Moreover, after the latter flows leave the network, the bulk transfer flow fully saturates the link again. This validates that Janus automatically achieves fair yet efficient resource allocation on a need-basis.

8.3.2 *All bulk transfer flows*

Without a target rate, Janus should reduce to classical congestion control that faily allocates bandwidth among all flows. To verify this, we start four bulk transfer flows that last for 60 seconds. We then measure the throughput deviation ΔTp, computed as the difference between the largest throughput and smallest throughput achieved by these four flows. Intuitively, ΔTp should be small (ideally 0) if the flows converge fast to the even bandwidth allocation. We compare ΔTp between alternative protocols (Fig.16) and show that Janus achieves better performance: the throughput deviation ΔTp is about 45% less than that of Cubic. PCC has much larger deviation, as shown in Sec.8.1.

8.4 Robustness against varying link latency

Vegas does not perform well when the link delay changes during a connection (Sec.4.4). Since Janus also uses Vegas, we would like to evaluate if Janus will suffer from the same problem. This is important since link latencies vary frequently in a mobile network.

We start two Janus flows (target rate $10\,\mathrm{Mbps}$) sharing a $20\,\mathrm{Mbps}$ bottleneck link, and increase the D_p by 20ms (from 40 ms) every 20 seconds. Then we plot the real-time throughput of one flow in Fig. 17 (another flow shows nearly identical performance). While the throughput of the

1536-1233 (c) 2018 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TMC.2018.2870156, IEEE Transactions on Mobile Computing

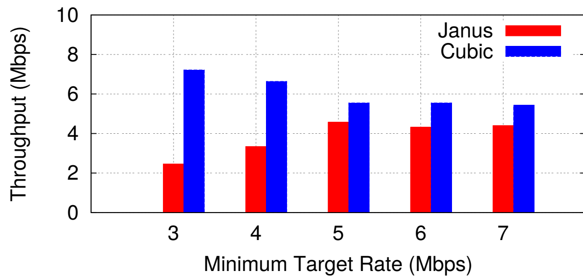

Fig. 19: Throughput comparison between Cubic and competing Janus with different minimum target rates.

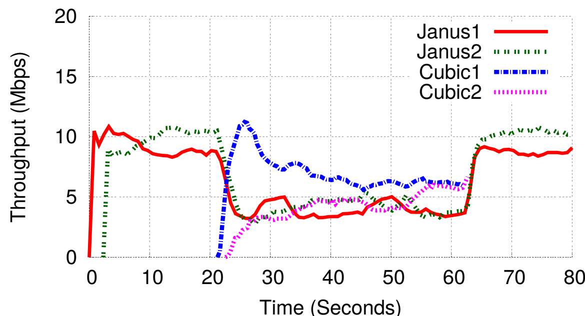

Fig. 20: Convergence dynamics of two Janus flows and two Cubic flows. Janus keeps its throughput around minimum target rate (5 Mbps) and is friendly to competing Cubic flows.

flow first drops due to the increasing propagation delay, Janus quickly recovers its 10 Mbps bandwidth allocation. Repeating the same experiments with Vegas shows that both flows are starved, just as the link latency starts to increase.

To explain the robustness of Janus, we plot the `cwnd` variation and estimated propagation delay D_p in Fig.18. Link latency increases trigger the following processes: First Janus switches to the *aggressive stage* and increases `cwnd` exponentially when the flow throughput drops below the bound $R * (1 - r)$. After flow throughput recovers back to R, Janus clears the queued packets by cutting `cwnd` down to $P_{avg} * D_p$ (Sec.4.4) before switching back to Vegas again. Here, the estimated D_p is outdated, so Janus further reduces the `cwnd` driving the throughput proportionately lower, which forces Janus into the *aggressive stage*. Finally, Janus updates its estimation of D_p, and the above cycle is repeated once again with updated D_p, allowing Janus to converge the `cwnd` at a higher level.

8.5 Fairness with other TCP

To study whether Janus can co-exist with flows of other TCP variants, we consider competing Cubic flows. We start one Janus flow and a Cubic flow simultaneously and measure their respective average throughput. We see in Fig.20 that the throughput for Janus flows is within a small fraction of the Cubic flows, and R_{min} approaches the fair share of total bandwidth. This implies that despite the fact that delay-based protocols cannot share bandwidth evenly with loss-based protocols given their inherent conservative behavior, the minimum target rate addresses this limitation for Janus flows. The key reason is that Janus adopts a *tit-for-tat* competition strategy: it will first lower its target rate, in an attempt to share resources with competing flows when the network is saturated. However if the competing flow continually overwhelms the network, Janus is forced into *aggressive stage* to assure at least R_{min} is satisfied.

To take a closer look at the dynamics of convergence with heterogeneous flows, we start two Janus flows ($R_{min} = 5\,\text{Mbps}$) and two Cubic flows consecutively. As shown in Fig. 20, Janus maintains its throughput around $5\,\text{Mbps}$ given the presence of Cubic flows, and increases back to $10\,\text{Mbps}$ as soon as the Cubic flows leave the network. Thus, Janus can co-exist nicely with different congestion control protocols as long as the minimum target rate is satisfied.

9 Conclusion

Given the tremendous diversity of mobile apps and wireless access technologues, manually configuring the TCP network stack for each scenario is inflexible, error-prune and usually leads to suboptimal performance. Janus instead enables automatic, *per-flow* TCP selection and configuration. Our approach is lightweight and has a low barrier to deployment because (i) it does not need any modification to existing congestion control implementations, and (ii) it does not depend on feedback from lower layers or in-network devices. We showed, through theoretical analysis as well as a proof-of-concept system implementation, that Janus not only improves QoE for various apps by seamlessly switching between different congestion control variants, but also enforces max-min fairness bandwidth allocation among flows given limited network resources.

Acknowledgments

This work is supported by the Office of Naval Research under grant N000141612651.

References

[1] Cisco, "Cisco visual networking index: Global mobile data traffic forecast update," Tech. Rep., 2016.
[2] "The linux kernel archives," https://www.kernel.org/.
[3] Y. Zhu, Z. Zhang, Z. Marzi, C. Nelson, U. Madhow, B. Y. Zhao, and H. Zheng, "Demystifying 60ghz outdoor picocells," in *Proceedings of the 20th annual international conference on Mobile computing and networking*. ACM, 2014, pp. 5–16.
[4] "Quic, a multiplexed stream transport over udp," https://www.chromium.org/quic.
[5] T.-Y. Huang, N. Handigol, B. Heller, N. McKeown, and R. Johari, "Confused, timid, and unstable: picking a video streaming rate is hard," in *Proceedings of the 2012 ACM conference on Internet measurement conference*. ACM, 2012, pp. 225–238.
[6] C. Caini and R. Firrincieli, "Tcp hybla: a tcp enhancement for heterogeneous networks," *International journal of satellite communications and networking*, vol. 22, no. 5, pp. 547–566, 2004.
[7] S. Mascolo, C. Casetti, M. Gerla, M. Y. Sanadidi, and R. Wang, "Tcp westwood: Bandwidth estimation for enhanced transport over wireless links," in *Proceedings of the 7th annual international conference on Mobile computing and networking*. ACM, 2001, pp. 287–297.
[8] C. P. Fu and S. C. Liew, "Tcp veno: Tcp enhancement for transmission over wireless access networks," *IEEE Journal on selected areas in communications*, vol. 21, no. 2, pp. 216–228, 2003.
[9] K. Xu, Y. Tian, and N. Ansari, "Tcp-jersey for wireless ip communications," *IEEE Journal on Selected Areas in Communications*, vol. 22, no. 4, pp. 747–756, 2004.
[10] Y. Sankarasubramaniam, Ö. B. Akan, and I. F. Akyildiz, "Esrt: event-to-sink reliable transport in wireless sensor networks," in *Proceedings of the 4th ACM international symposium on Mobile ad hoc networking & computing*. ACM, 2003, pp. 177–188.

1536-1233 (c) 2018 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TMC.2018.2870156, IEEE Transactions on Mobile Computing

[11] C.-Y. Wan, S. B. Eisenman, and A. T. Campbell, "Coda: congestion detection and avoidance in sensor networks," in *Proceedings of the 1st international conference on Embedded networked sensor systems*. ACM, 2003, pp. 266–279.
[12] S. Ha, I. Rhee, and L. Xu, "Cubic: a new tcp-friendly high-speed tcp variant," *ACM SIGOPS Operating Systems Review*, vol. 42, no. 5, pp. 64–74, 2008.
[13] D. Leith and R. Shorten, "H-tcp: Tcp for high-speed and long-distance networks," in *Proceedings of PFLDnet*, vol. 2004, 2004.
[14] A. Baiocchi, A. P. Castellani, and F. Vacirca, "Yeah-tcp: yet another highspeed tcp," in *Proc. PFLDnet*, vol. 7, 2007, pp. 37–42.
[15] N. Cardwell, Y. Cheng, C. S. Gunn, S. H. Yeganeh, and V. Jacobson, "Bbr: Congestion-based congestion control," *Queue*, vol. 14, no. 5, p. 50, 2016.
[16] K. R. Chowdhury, M. Di Felice, and I. F. Akyildiz, "Tp-crahn: A transport protocol for cognitive radio ad-hoc networks," in *INFOCOM 2009, IEEE*. IEEE, 2009, pp. 2482–2490.
[17] A. K. Al-Ali and K. Chowdhury, "Tfrc-cr: An equation-based transport protocol for cognitive radio networks," *Ad Hoc Networks*, vol. 11, no. 6, pp. 1836–1847, 2013.
[18] D. Sarkar and H. Narayan, "Transport layer protocols for cognitive networks," in *INFOCOM IEEE Conference on Computer Communications Workshops, 2010*. IEEE, 2010, pp. 1–6.
[19] M. Alizadeh, A. Greenberg, D. A. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan, "Data center tcp (dctcp)," in *ACM SIGCOMM computer communication review*, vol. 40, no. 4. ACM, 2010, pp. 63–74.
[20] R. Mittal, N. Dukkipati, E. Blem, H. Wassel, M. Ghobadi, A. Vahdat, Y. Wang, D. Wetherall, D. Zats *et al.*, "Timely: Rtt-based congestion control for the datacenter," in *ACM SIGCOMM Computer Communication Review*, vol. 45, no. 4. ACM, 2015, pp. 537–550.
[21] M. Alizadeh, A. Kabbani, T. Edsall, B. Prabhakar, A. Vahdat, and M. Yasuda, "Less is more: trading a little bandwidth for ultra-low latency in the data center," in *Presented as part of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*, 2012, pp. 253–266.
[22] C. Wilson, H. Ballani, T. Karagiannis, and A. Rowtron, "Better never than late: Meeting deadlines in datacenter networks," in *ACM SIGCOMM Computer Communication Review*, vol. 41, no. 4. ACM, 2011, pp. 50–61.
[23] K. Winstein, A. Sivaraman, and H. Balakrishnan, "Stochastic forecasts achieve high throughput and low delay over cellular networks," in *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, 2013, pp. 459–471.
[24] F. Lu, H. Du, A. Jain, G. M. Voelker, A. C. Snoeren, and A. Terzis, "Cqic: Revisiting cross-layer congestion control for cellular networks," in *Proceedings of the 16th International Workshop on Mobile Computing Systems and Applications*. ACM, 2015, pp. 45–50.
[25] Y. Zaki, T. Pötsch, J. Chen, L. Subramanian, and C. Görg, "Adaptive congestion control for unpredictable cellular networks," in *ACM SIGCOMM Computer Communication Review*, vol. 45, no. 4. ACM, 2015, pp. 509–522.
[26] V. Jacobson, "Congestion avoidance and control," in *ACM SIGCOMM computer communication review*, vol. 18, no. 4. ACM, 1988, pp. 314–329.
[27] D. X. Wei, C. Jin, S. H. Low, and S. Hegde, "Fast tcp: motivation, architecture, algorithms, performance," *IEEE/ACM Transactions on Networking (ToN)*, vol. 14, no. 6, pp. 1246–1259, 2006.
[28] K. Tan, J. Song, Q. Zhang, and M. Sridharan, "A compound tcp approach for high-speed and long distance networks," in *Proceedings-IEEE INFOCOM*, 2006.
[29] D. Han, R. Grandl, A. Akella, and S. Seshan, "Fcp: a flexible transport framework for accommodating diversity," vol. 43, no. 4. ACM, 2013, pp. 135–146.
[30] A. Munir, G. Baig, S. M. Irteza, I. A. Qazi, A. X. Liu, and F. R. Dogar, "Friends, not foes: synthesizing existing transport strategies for data center networks," in *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 4. ACM, 2014, pp. 491–502.
[31] K. Winstein and H. Balakrishnan, "Tcp ex machina: computer-generated congestion control," in *ACM SIGCOMM Computer Communication Review*, vol. 43, no. 4. ACM, 2013, pp. 123–134.
[32] A. Sivaraman, K. Winstein, P. Thaker, and H. Balakrishnan, "An experimental study of the learnability of congestion control," in *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 4. ACM, 2014, pp. 479–490.
[33] M. Dong, Q. Li, D. Zarchy, P. B. Godfrey, and M. Schapira, "Pcc: Re-architecting congestion control for consistent high performance," in *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, 2015, pp. 395–408.
[34] B. Cronkite-Ratcliff, A. Bergman, S. Vargaftik, M. Ravi, N. McKeown, I. Abraham, and I. Keslassy, "Virtualized congestion control," in *Proceedings of the 2016 conference on ACM SIGCOMM 2016 Conference*. ACM, 2016, pp. 230–243.
[35] K. He, E. Rozner, K. Agarwal, Y. J. Gu, W. Felter, J. Carter, and A. Akella, "Ac/dc tcp: Virtual congestion control enforcement for datacenter networks," in *Proceedings of the 2016 conference on ACM SIGCOMM 2016 Conference*. ACM, 2016, pp. 244–257.
[36] M. Ghobadi, S. H. Yeganeh, and Y. Ganjali, "Rethinking end-to-end congestion control in software-defined networks," in *Proceedings of the 11th ACM Workshop on Hot Topics in Networks*. ACM, 2012, pp. 61–66.
[37] GTmetrix, "The top 1,000 sites on the internet," Tech. Rep. [Online]. Available: https://gtmetrix.com/top1000.html
[38] L. S. Brakmo and L. L. Peterson, "Tcp vegas: End to end congestion avoidance on a global internet," *IEEE Journal on selected Areas in communications*, vol. 13, no. 8, pp. 1465–1480, 1995.
[39] J. Mo, R. J. La, V. Anantharam, and J. Walrand, "Analysis and comparison of tcp reno and vegas," in *INFOCOM*, vol. 3. IEEE, 1999, pp. 1556–1563.
[40] I. Marsic, *Computer networks: Performance and quality of service*. Rutgers University, 2010.
[41] "Youtube help : Live encoder settings, bitrates, and resolutions," https://support.google.com/youtube/answer/2853702?hl=en.
[42] "Linux traffic controller," https://linux.die.net/man/8/tc.
[43] "Navigation timing api," https://www.w3.org/TR/navigation-timing/.

Fan Zhou is a PhD Student in the ECE Department at Northeastern University. He is currently working in the Next Generation Networks and Systems Lab under the supervision of Prof. Kaushik Chowdhury. He received B.S. and M.S. degree from Hohai University (2011) and Beijing University of Posts and Telecommunications (2014). His research interests spread across different layers of wireless networking, with a specific focus on the design and implementation of high performance data transfer architecture.

David Choffnes is an assistant professor in the College of Computer and Information Science at Northeastern University. His research is primarily in the areas of distributed systems and networking, focusing on mobile systems, privacy, and security. He is a co-author of three textbooks, and his research has been supported by the NSF, Google, the Data Transparency Lab, M-Lab, and a Computing Innovations Fellowship.

Kaushik Roy Chowdhury (M'09-SM'15) received the M.S. degree from the University of Cincinnati in 2006, and the Ph.D. degree from the Georgia Institute of Technology in 2009. He was an Assistant Professor from 2009 to 2015 at Northeastern University, where he is now an Associate Professor with the Electrical and Computer Engineering Department. He was a winner of the Presidential Early Career Award for Scientists and Engineers (PECASE) in 2017, ONR Director of Research Early Career Award in 2016 and the NSF CAREER Award in 2015. His current research interests include dynamic spectrum access, wireless RF energy harvesting and IoT and in the area of intra/on-body communication.

1536-1233 (c) 2018 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.