Enabling Users to Control their Internet

Ammar Tahir, Radhika Mittal
University of Illinois at Urbana-Champaign

Abstract

Access link from the ISP tends to be the bottleneck for many users. However, users today have no control over how the access bandwidth (which is under the ISP's control) is divided across their incoming flows. In this paper, we present a system, CRAB, that runs at the receiver's devices – home routers and endpoints – and enforces user-specified weights across the incoming flows, without any explicit support from the ISP or the senders. It involves a novel control loop that continuously estimates available downlink capacity and flow demands by observing the incoming traffic, computes the max-min weighted fair share rates for the flows using these estimates, and throttles the flows to the computed rates. The key challenge that CRAB must tackle is that the demand and capacity estimated by observing the incoming traffic at the receiver (after the bottleneck) is inherently ambiguous – CRAB's control loop is designed to effectively avoid and correct these ambiguities. We implement CRAB on a Linux machine and Linksys WRT3200ACM home router. Our evaluation, involving real-world flows, shows how CRAB can enforce user preferences to achieve 2× lower web page load times and 3× higher video quality than the status quo.

1 Introduction

This paper tackles a common and seemingly simple problem: how can users control how their Internet access link gets shared across their incoming flows? For instance, how can a user ensure that their Youtube video streaming is not impacted when the Dropbox app on their device starts downloading large files at the same time, and the two flows compete at the user's Internet access link?

At a glance, a plausible solution is to exploit the traffic shaping features provided in many home routers (e.g. mechanisms for weighted fair queuing or prioritization) [1, 15, 18]. However, these mechanisms are effective only when the bottleneck is at (and queues build up at) the home router – this happens when the bottleneck is the uplink from the router to the Internet Service Provider (ISP) for outgoing flows or the downlink from the router to the end-devices for the incoming flows [1]. Our work targets a different problem as illustrated in Figure 1, where the bottleneck for the incoming flows is the downlink from the ISP to the home router, and queues build up in the ISP. This is a common scenario [36, 50], with the Internet access bandwidth being governed by contractual agreements between an end-user and their ISP, and the median broadband download speed being less than 35Mbps in more than half the countries worldwide [3].

Existing literature provides us with two options for managing flow shares at the Internet access link, both of which are beyond the receiving user's control. The first option is to directly schedule or shape traffic at the bottleneck (e.g. via prioritization or weighted fairness) [17, 25, 43, 49]. However, the access bottleneck is controlled by the ISP. ISPs are unaware of user preferences and do not deploy any mechanisms that enable end users to configure how their traffic is scheduled and shaped at the access bottleneck. [1] The second option is for the senders to appropriately control the rate at which flows arrive at the bottleneck. For example, low-priority senders can use "scavenger" protocols that yield bandwidth more readily to higher priority flows [34, 38, 41, 48]. This is again outside the receiving user's control – it is up to the sender to use and configure such protocols.

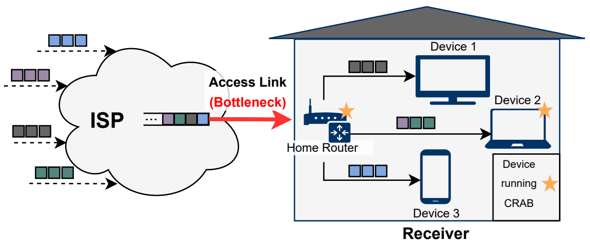


Figure 1: CRAB's Target Scenario. The user may own multiple devices, each downloading multiple flows over the Internet. These flows arrive at the user's home router via the access link from the ISP, from where they get routed to individual devices. The Internet access link is often the bottleneck for the incoming flows [3, 36].

We design a system, CRAB, [2] that enables users (receivers) to control how their Internet access bandwidth gets shared across their incoming flows without any explicit support from external entities (i.e. the ISPs and the senders). Here we use the term receiver to collectively refer to devices in the receiving domain that an end user can directly access and configure – these include end devices (phones and computers) as well as home routers (attached to the access link). More generally, CRAB provides a mechanism to control flow shares exclusively from a vantage point that is topologically placed *after* the bottleneck – where queues don't build up naturally, and where one has seemingly zero control.

CRAB allows a user (i) to configure the home router with weights across each end-device, and (ii) to optionally configure end-devices with weights across their incoming flows

[1] Although a few research proposals of this form exist [19, 24, 27, 35, 54], they have not been realized in practice, given the inherent difficulty of coordinating across multiple domains.

[2] For Customizable Receiver-driven Allocation of Bandwidth.

(defined based on application, web domain, etc). It then strives to throttle the incoming flows at the home router (grouped by destination device) and individually at the end devices (if enabled) to their respective *max-min weighted fair share rates*. CRAB's key challenge lies in correctly computing these rates after the bottleneck (as discussed below). While this after-the-bottleneck throttling cannot *directly* control how the access bandwidth is divided across the incoming flows, it signals the senders (which typically run some form of congestion controller [22, 26, 30, 32, 39, 46, 52]) to lower their sending rates to the throttled values, thus enabling the flows to eventually converge to their desired shares.

So what makes it difficult to compute the weighted fair share rates after the bottleneck? Given flow weights, computing the correct (max-min) weighted fair rates for each flow requires knowing the bottleneck link capacity and the flow demands. Once the absolute weighted share of a flow has been computed from the link capacity and flow weight, the max-min weighted fair rates can be computed by re-allocating any excess capacity, that is unused by flows with demands smaller than their absolute share, to other flows in the proportion of their weights. While the capacity and the flow demands are naturally available at the bottleneck, CRAB (placed after the bottleneck) must estimate them by observing the incoming traffic at the receiver. This introduces multiple challenges:

(1) It is not possible to distinguish whether the total traffic observed at the receiver is limited by the access link capacity or by the flows' cumulative demands – the latter would result in underestimating capacity.

(2) The arrival rate of a flow at the receiver depends on the rate at which it was served at the bottleneck. A flow that got a small share of bandwidth at the bottleneck (less than its weighted share or demand) could be wrongly perceived as having low demand.

(3) If the receiver incorrectly throttles flows to rates lower than their weighted shares (due to spuriously low capacity or demand estimates), the flows' sending rates (and their observed arrival rates at the receiver) would end up matching the throttled rates. As a result, the link capacity and demand estimates would stay unchanged and the system will not self-correct. Similar reasoning makes it difficult to adapt to an increase in link capacity and flow demands.

The centerpiece of CRAB is a control loop that is designed to tackle the above challenges (§3). It continuously loops between (i) measuring flow arrival rates (over timescales of hundreds of milliseconds) to estimate link capacity and flow demands, and (ii) re-computing and enforcing weighted fair-share rates based on these estimates. By waiting for rates measured over long enough timescales before reacting, CRAB avoids fast reaction to spuriously low demand estimates – it allows for the impact of any flow throttling to kick in, and for the sending (and observed) rates for the remaining flows to grow to their true demands (or weighted shares), before reallocating any unused capacity. When re-allocating capacity from a flow with low demand, CRAB leaves some headroom to detect growing demand, at which point it immediately reclaims all of the flow's re-allocated bandwidth, again allowing the flow to grow to its true demand or its weighted share. To self-correct capacity underestimation, it periodically probes for more bandwidth by explicitly increasing the total rate assigned to flows and checking for any consequent increase in observed rates.

CRAB runs the same logic at the home router and at the end-points, without requiring any explicit coordination among them CRAB at the home router enforces per-device shares based on estimated access link capacity and per-device demands. CRAB at the end-point independently adapts its capacity estimate to the per-device rate enforced by the home router and controls per-flow shares. When directly attached to the ISP's link (without a home router) or when the router and end-device are owned by different entities (e.g. in airports, cafes, etc), an end-device with CRAB enabled can self-sufficiently enforce its desired shares across its incoming flows.

CRAB's cautious re-allocation of unused capacity can leave the bottleneck link under-utilized at times. As we illustrate in §2, some amount of link under-utilization is inevitable when shaping traffic *after* the bottleneck (maximally utilizing the link would imply no flow gets throttled at the receiver, and consequently no impact on how the bottleneck bandwidth is shared). The link under-utilization with CRAB typically manifests as transient dips in the throughput for lower priority (throttled) flows, below their max-min weighted shares. This is a reasonable price to pay for better performance for higher priority traffic that is achieved by enforcing user-specified flow shares with CRAB.

We implement CRAB (§4) on a Linux endhost and a Linksys WRT3200ACM router. Our end-host implementation also includes hooks for classifying flows that are broadly defined by users based on applications and web domains, and involve cross-origin requests (e.g. to CDNs and ad networks). Our experiments involving real-world flows with different sender-side congestion controllers (YouTube videos, web browsing, and bulk downloads), show how CRAB comes close to achieving the desired weighted fair share rates. In particular, CRAB achieves 2.5-3$\times$ higher video quality and 2$\times$ lower web page load times in presence of lower-priority bulk flows than the status-quo (that cannot enforce desired preferences), with 10-20% decrease in overall link utilization.

2 Overview

CRAB enables the user to manage how their Internet access link (that is often the bottleneck for downloads [3, 36]) gets shared across the incoming flows. Figure 1 illustrates a typical target scenario. CRAB at the home router controls how the access bandwidth is shared across traffic destined to each end-device. CRAB at the end-device (if enabled) controls how its router-enforced bandwidth share is divided across its incoming flows. Note that all end devices need not run CRAB

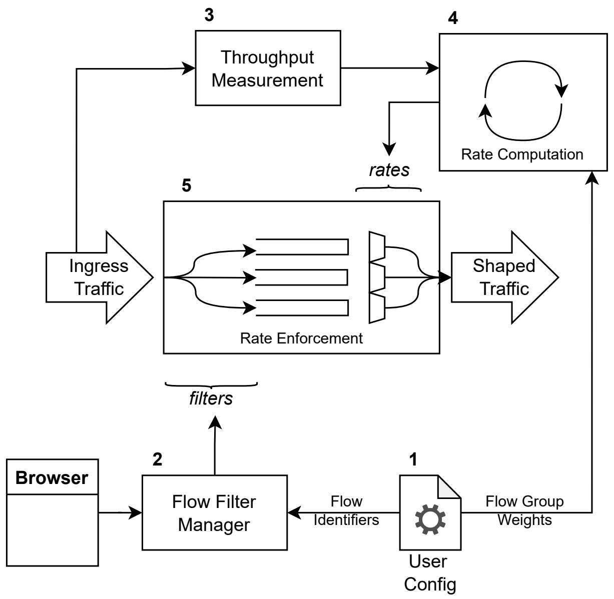


Figure 2: CRAB's high-level workflow (detailed in §2.1).

– only an end device that wishes to control the bandwidth sharing across its own flows needs to enable CRAB.

2.1 CRAB Framework

We provide an overview of the CRAB framework at the end-device (noting the small differences in CRAB's router design towards the end). CRAB sits in front of the ingress interface, where it intercepts and shapes the incoming traffic before forwarding it to the kernel's TCP/IP stack. Figure 2 shows the key elements in CRAB's architecture.

1. User Interface. Our current prototype allows users to define flows based on three criteria: (i) all traffic destined to a specified application running on the end-device, (ii) all traffic associated with a specified web-domain, and (iii) all traffic originating from a specified source address. [3] The user can group multiple flows into a flow-group, and specify a weight for each flow-group. Users can also specify a weight for a default flow group, where traffic not classified in any other flow group is mapped. Henceforth, we use the terms flow and flow-group interchangeably.

2. Flow Filter Manager. It maps high-level flow identifiers (as specified by the user) into TCP/IP header fields that the system can use to classify packets as and when they arrive at the interface. Mapping a web-domain into header fields requires some inputs from the browser (we detail this in §4).

3. Throughput Measurement. CRAB sniffs the incoming traffic to measure (i) throughput of each flow group to estimate demands, and (ii) the cumulative throughput over all flows to estimate link capacity. The link capacity estimated by CRAB at the end-device corresponds to the device's bandwidth share as enforced by the home router.

4. Rate Computation. CRAB computes weighted fair share rates for each flow, based on user-specified weights and the capacity and demand estimates obtained from flow throughput measurements (as detailed in §3).

5. Rate Enforcement. CRAB uses the mappings from the flow filter manager to classify incoming traffic into user-defined flow groups. It puts the traffic for each group into separate queues, and throttles each queue to its computed weighted fair share rate. Since CRAB throttles traffic *after* the bottleneck, it cannot *directly* control how flows are scheduled at the bottleneck. However, it induces packet losses and queuing delay at the receiver, which signals the senders to adjust their rates to the throttled value, thus eventually achieving the desired bandwidth shares at the bottleneck.

We considered a few other alternatives for signalling sending rates from the receiver, e.g. by adjusting TCP receive windows. We decided to use throttling for rate enforcement because of its generality – all senders that run some form of congestion controller (either over TCP [22, 26, 30, 32, 52] or UDP [39, 46]) would naturally react to queue buildups and/or packet drops induced by throttling.

CRAB framework at the home router is same as that at the end-device. The only difference is that since users configure the home router with weights across each end-device (directly identified by the destination IP address), an explicit flow filter manager is not required.

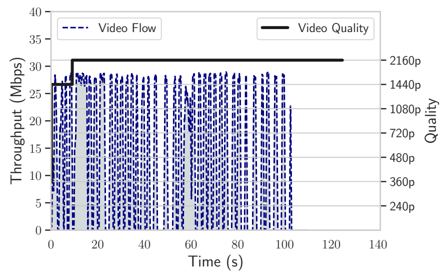
Note that, in principle, one could have managed flow-group shares directly at the home router, instead of the end-device. But then classifying the incoming traffic based on flow-groups at the router would have required explicit coordination with the applications running at the end-device in real-time (as explained in §4), which would have complicated system deployment. Our current division of functionality between the home-router and end-points requires no explicit coordination among them, which greatly simplifies CRAB's deployment and use, and extends CRAB's utility to other contexts beyond home users (as discussed in §7). It also reduces computational complexity at the router, with the router managing only per-device queues, and each device then managing the rates for their own flows.

2.2 Goals and Challenges

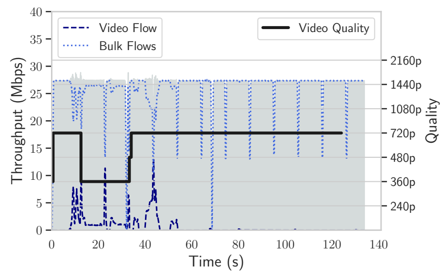
We use a series of experiments to illustrate some of the challenges that CRAB must tackle. We consider a scenario where a Linux end-host is directly attached to the access link from the ISP (without a home router). For repeatable experiments, we emulate an ISP-controlled access link by routing all traffic for the end-host via a router that mimics the ISP and throttles the traffic to 30Mbps. When we only stream a 4K YouTube video on the end-host, we find that the video quality stays at the maximum level (Figure 3a). We then stream the same video in presence of two long-lasting bulk downloads [4] over

[3]Future extensions of our system can support more criteria.

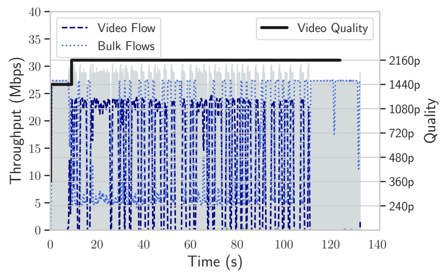
[4]Bulk downloads of games and movies are fairly common among users with limited bandwidth because of inaccessibility of high quality video streaming or cloud gaming [29, 31].



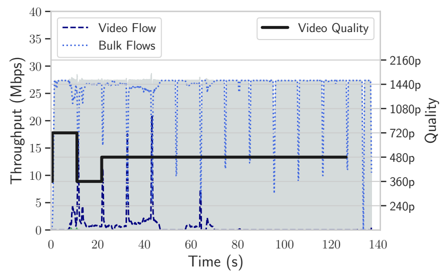
(a) Video plays at highest quality level alone.



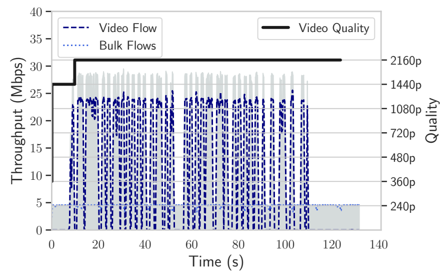
(b) Video quality suffers greatly in presence of bulk downloads.



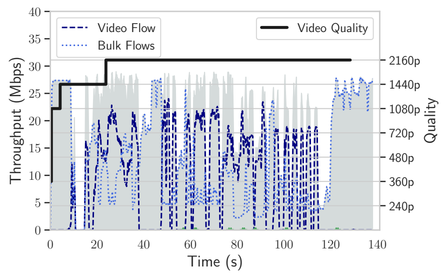
(c) WFQ at the bottleneck in 5:1 ensures bulk flows do not affect video quality.



(d) Instant reaction fully utilizes the link, but cannot enforce desired weights.



(e) Indefinitely throttling bulk flows results severely under-utilizes the link.



(f) CRAB ensures high video quality at cost of slight link underutilization

Figure 3: Video quality suffers in presence of bulk download flows, we look at different possible ways to ensure this does not happen. In all experiments, link bandwidth is set to 30 Mbps.

the Internet, and evaluate the results under different settings.

Status Quo. Today, there is no way for a user to enforce how their access bandwidth gets divided across their flows. We find that with the bulk downloads consuming a large share of the access bandwidth, the status-quo achieves very low video quality (Figure 3b).

This degradation in video quality is clearly undesirable and can be mitigated if the user can specify and enforce a higher weight (say 5×) for the video flow.

Impractical Ideal: WFQ at the bottleneck. We next model the ideal, but impractical scenario where the ISP enforces user preferences at the bottleneck via WFQ. For this, we configure the router in our setup (that mimics the ISP) to use weighted deficit round-robin (DRR) [49], with the video flow to bulk flows ratio set to 5:1. As shown in Figure 3c, the video flow is able to use its absolute share of 25Mbps, and achieve the highest video quality, while bulk downloads get at least 5Mbps. WFQ is *work-conserving* – whenever the video flow consumes less than its share of 25Mbps (e.g. when its playback buffer is full), the remaining capacity is used by the bulk downloads, keeping the link maximally utilized.

Impossible to mimic WFQ after the bottleneck. CRAB strives to mimic the ideal WFQ-enforced rates. However, while WFQ at the bottleneck can achieve both desired bandwidth shares and maximal link utilization, there is an inherent trade-off between the two when shaping traffic *after* the bottleneck. We highlight this trade-off by illustrating two extreme strawmen for rate computation at the receiver.

(i) Work-conserving re-allocation cannot enforce weighted fairness after the bottleneck. In order to maximally utilize the link, whenever the video flow's demand becomes less than its absolute share, any unused capacity should be explicitly reallocated to the bulk flows. It is natural to use the arrival rate of the video flow at the receiver as an estimate of its demand. However, if the video flow gets a small share of bandwidth at the bottleneck due to competing flows, it will have a low arrival rate at the receiver and will be incorrectly perceived as having low demand. We now evaluate the effects of instantaneously re-allocating unused capacity based on such spurious demand estimates.

For this, we configure the end-device to use Linux HTB (Hierarchical Token Bucket) [5] to throttle the incoming flows to their absolute shares (25Mbps for video and 5Mbps for bulk), and enable HTB's "bandwidth borrowing" feature which immediately re-allocates any capacity that is unused by a flow with a smaller arrival rate. Since the total arrival rate at the receiver is already capped by bottleneck link capacity, such instantaneous re-allocation induces no throttling – while this achieves maximal link utilization, it cannot enforce desired bandwidth shares and produces the same outcome as the status-quo (Figure 3d). For the setup in Figure 1, enabling WFQ for the incoming flows at the home router (after the bottleneck), will produce the same effect.

More generally, maintaining maximal link utilization is fundamentally at odds with CRAB's mechanism of enforcing desired rates. In order to signal any change in the sending rates, CRAB must throttle some flows at the receiver – the link under-utilization thus induced is what gives room to the remaining flows to grow to their true demands or absolute weighted shares.

(ii) No re-allocation leads to severe under-utilization. At the other extreme, indefinitely throttling bulk flows to their absolute weighted fair rates, oblivious of video flow demand estimates, allows the video flow to grow to its true demand and ensure high video quality, but decreases the link utilization by 46% compared to the status quo (Figure 3e).

CRAB achieves desired shares with high link utilization. CRAB navigates the above trade-off by re-allocating unused capacity more cautiously (at timescales of a few hundred milliseconds) – this allows the impact of any throttling to kick in, and for the flows to grow to their true demands or absolute shares, before the unused capacity is re-allocated. For flows consuming less than their absolute shares, CRAB provisions for detecting a growth in demand, upon which it immediately reclaims all lent out capacity. This cautious reallocation and aggressive reclamation may under-utilize the link at times, which is inevitable (as discussed above) and primarily affects the bulk flows. On the whole, as shown in Figure 3f, we find that CRAB can effectively enforce the desired shares while lowering the link utilization by only 19% with respect to the status quo. CRAB has 2× higher link utilization than the extreme alternative of no re-allocation or of the user explicitly pausing the bulk flow while the video lasts.

Other Challenges. CRAB was able to correctly estimate link capacity in the above experiments. However, link capacity estimation can be more challenging in other scenarios. In particular, if the total incoming traffic at the receiver is limited by flows' cumulative demand (as opposed to the access link capacity), it would result in underestimation of link capacity – CRAB should be able to self-correct its capacity estimate to ensure correct bandwidth shares if flows' demands increase. Link capacity could also vary over time – CRAB should be able to detect any changes and accordingly recompute weighted share rates. What makes such self-correction and adaptation particularly difficult is that once flows have been throttled to a spuriously low rate, their sending rates (and, consequently, their arrival rates and CRAB's capacity estimates) could stay stuck at the throttled values. CRAB handles this by explicitly probing for more bandwidth.

We detail CRAB's control loop, i.e. its re-allocation, reclamation, and bandwidth estimation logic in §3.

The specific control loop we describe in this paper is one way of using CRAB's framework for controlling flow shares at the receiver. There can be alternative ways of using our framework while complying with our observation that some amount of link under-utilization is inherently needed to control flow shares from the receiver. For instance, we can directly throttle the *cumulative* rate of all incoming traffic at the receiver to a value lower than the overall link capacity that CRAB estimates (via its throughput observation and bandwidth probing logic). This creates an artificial bottleneck at the receiver where we can enforce the desired scheduling policy (prioritization, weighted fair queuing, etc) across the flow classes maintained by CRAB. While effective at controlling flow shares, such an approach would be more sensitive to precise bandwidth estimation and may lead to unnecessary wastage of bandwidth due to consistent link under-utilization. We evaluate this alternative in §5.4.

2.3 CRAB's Scope

CRAB's scope is limited in the following key ways:

1. CRAB relies on the fact that flows have a sender-side rate control mechanism that is responsive to throttling. This holds for most of the traffic on the Internet that either uses TCP's congestion control mechanism or runs adaptive rate control over UDP [39]. CRAB is not effective in scenarios where a flow is not responsive to throttling.

2. CRAB cannot directly control the fine-grained queuing behavior at the bottleneck. It can only influence the rates achieved by different flows over long-enough timescales (a few hundred milliseconds), as it requires the senders to react to the throttled rates or the increased room for growth in rates. This allows CRAB to effectively control bandwidth shares across long-lived flow groups, e.g. video streaming, conferencing, web browsing sessions, bulk downloads, etc. However, CRAB cannot effectively control the queuing delay experienced at the bottleneck by short intermittent downloads that terminate before CRAB gets a chance to react (e.g. a flow group comprising of only interactive chats). Though CRAB cannot actively help such flows, it will not hurt them either.

3. CRAB can only actively control how a user's incoming flows share their common bottleneck (at the ISP's access link or at the access router). If a flow is bottlenecked elsewhere (e.g. at the sender's uplink), it is simply perceived by CRAB as having a lower demand at the shared access bottleneck, and CRAB accordingly reallocates the access link capacity unused by this flow across the remaining flows.

4. Since CRAB must react at slow timescales (of hundreds of ms) to build correct capacity and demand estimates, it is not a good fit for volatile cellular networks where link capacity changes by large magnitudes at much smaller timescales due to high mobility, hand-overs, etc [23, 37, 51, 57]. In comparison, we found broadband connectivity and home WiFi networks to be significantly more stable (see Appendix B).

3 CRAB Control Loop

We now describe CRAB's control loop that continuously iterates between (i) measuring flow throughput to estimate capacity and demand, and (ii) computing and applying new per-flow rates. Table 1 lists different attributes that CRAB maintains for each flow and uses for rate computations.

3.1 Throughput Measurement

Two parameters govern the granularity of our throughput measurement: the observation period (t) and the number of

Flow Attribute	Description
weight	The weight assigned by the user
observed_tpt	The measured throughput of the flow (its arrival rate at the ingress)
true_bw	The absolute weighted fair share of the flow computed from estimated link capacity and flow weight
lent_bw	The unused bandwidth the flow lends out to other flows
borrowed_bw	The amount of bandwidth the flow borrowed from other flows
assigned_bw	The rate assigned to a flow (set to $true\_bw + borrowed\_bw$)
saturating (bool)	set to true if the flow's demand is potentially higher than its assigned bandwidth
non_saturating (bool)	set to true if the flow's demand is smaller than its assigned bandwidth
growing (bool)	set to true if the flow needs to reclaim its lent bandwidth

Table 1: Attributes of a flow in CRAB

observations (n). In each observation, CRAB measures the throughput (or arrival rate) for each flow over time t (i.e. number of bytes received in t time divided by t). It makes n such observations and picks the maximum value as the observed throughput of a flow. We use the max filter instead of mean or median to capture bursts which are common for applications like web browsing and video streaming.

The values of t and n govern how long we wait before making any changes to flow rates. A very small value of t would result in inaccurate throughput measurements, whereas a very high value can mask spikes in demand. [5] A very small n will not give flows enough time to adjust to new rates skewing demand estimates, and a very large n would induce much slower reactions to changes in demands and capacity. In practice, $n \times t$ should be as high as a few RTTs to allow the senders enough time to react. We find that setting t to 200ms and n to 5 works well across different scenarios. We evaluate the impact of these parameter settings in §5.6.

After every throughput measurement (over $n \times t$ s), CRAB sets following flags of each flow f:

Growing: A flow is determined to be growing if f had previously lent out bandwidth but its observed throughput indicates an increase in its demand (i.e. it is using more than what it was using earlier). $f.growing = (f.lent\_bw > 0)$ **and** $(f.observed\_tpt \geq f.assigned\_bw - f.lent\_bw)$.

Saturating: If the flow f is either growing or it is utilizing almost all of its assigned bandwidth, i.e., $f.saturating = f.growing$ **or** $(f.observed\_tpt + \delta \geq f.assigned\_bw)$. Here, δ masks noise in throughput observations, and is set to $max(0.1 \times f.observed\_tpt, 0.25Mbps)$. Note that we consider all growing flows to be saturating, but a saturating flow (that is simply utilizing all of its assigned bandwidth) may not necessarily be growing.

Non-saturating: If f is under-utilizing its assigned bandwidth (after subtracting its lent out bandwidth): $f.non\_saturating = (f,observed\_tpt + \delta) < (f.assigned\_bw - f.lent\_bw)$.

[5]The value of t should be at least as high as the inter-arrival time between multiple consecutive 64KB chunks to correctly compute throughput (with TSO/LRO enabled, packets arrive in bursts of 64KB).

3.2 Rate Computation Overview

Figure 4 shows a simplified state diagram for CRAB's control loop. Followed by a throughput measurement, we take one of the four actions in the exact priority order.

(1) If there exists any growing flow, we do reclamation for it (i.e. reclaim any bandwidth it has lent out to other flows).

(2) Otherwise, if there is at least one non-saturating and one saturating flow, we reallocate (or lend out) bandwidth unused by non-saturating flows to saturating flows.

(3) If observed throughput has dropped and there does not exist any saturating flow, we decrease the bandwidth estimate and divide it between flows according to their weights.

(4) In all other cases, we try to probe for more bandwidth.

We return to the throughput measurement after each action. The following sub-sections describe these actions and their triggers in more detail.

3.3 Reallocation

Reallocation takes place if there is at least one non-saturating flow (which can lend out bandwidth) and at least one saturating flow (which can potentially utilize this lent bandwidth).

CRAB first computes the bandwidth each flow f can lend:

$$f.lent\_bw = f.assigned\_bw - f.observed\_tpt - headroom$$

We keep a small headroom (set to 0.25Mbps) to enable detecting when the flow needs to grow back (§5.6 evaluates the impact of this parameter).

If $f.lent\_bw > f.borrowed\_bw$, this means that the flow can no longer make use of the bandwidth it has previously borrowed and instead has extra unused bandwidth to lend. In this case, CRAB subtracts $f.borrowed\_bw$ from $f.lent\_bw$ and resets $f.borrowed\_bw$ to zero. CRAB computes the global excess (unused) bandwidth by summing up the lent bandwidth across all such flows. It then resets the assigned bandwidth for each flow f to $f.true\_bw$, after which it reallocates the excess bandwidth across all flows in proportion to their weights until their demands are satisfied, accordingly updating $f.borrowed\_bw$ and $f.assigned\_bw$ (set to $f.true\_bw + f.borrowed\_bw$) for each flow. The algorithm for this redivision is given in AppendixA.

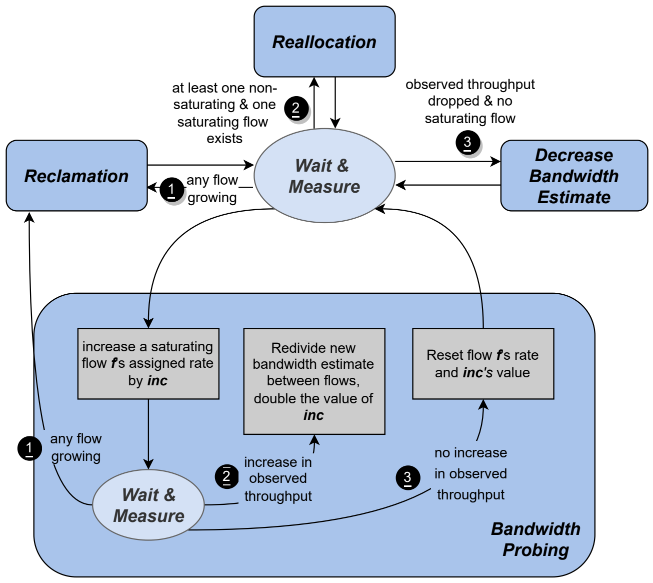

Reallocation
at least one non-saturating & one saturating flow exists
2
observed throughput dropped & no saturating flow
3
Reclamation
Wait & Measure
Decrease Bandwidth Estimate
1
any flow growing
increase a saturating flow f's assigned rate by inc
Redivide new bandwidth estimate between flows, double the value of inc
Reset flow f's rate and inc's value
1
any flow growing
2
increase in observed throughput
3
no increase in observed throughput
Wait & Measure
Bandwidth Probing

Figure 4: State diagram of CRAB's control loop.

3.4 Reclamation

Notice that we do not decrease the assigned bandwidth of the non-saturating flow during reallocation. This, combined with the lent bandwidth headroom, ensures that CRAB can detect growth in a flow's demand, and classify such a flow as *growing* (as noted in §3.1). To enable faster reclamation, CRAB terminates the throughput measurement sooner than $n \times t$ s, once it detects that any flow f is growing. It reclaims all bandwidth lent out by the flow f by setting $f.lent\_bw = 0$. It reduces the global excess if $f.lent\_bw$ (before updating to 0) was greater than $f.borrowed\_bw$, and accordingly recomputes how the updated excess bandwidth is redivided across flows (using the same logic from §3.3). Note that when redividing excess bandwidth, we consider a growing flow to be a saturating flow because it can potentially utilize more bandwidth as its demand is still unknown.

3.5 Bandwidth Estimation

CRAB keeps track of estimated bandwidth ($estimated\_bw$) based on the overall observed throughput across all flows ($total\_observed\_tpt$). We now discuss CRAB's mechanism for detecting a change (increase or decrease) in capacity. This is required for (i) correcting spuriously low capacity estimates caused by limited demand, and (ii) adapting to potential capacity variations (e.g. due to change in an end-device's share of bandwidth triggered by changes in another device's demands).

Decrease in Bandwidth. A drop in $total\_observed\_tpt$ can happen due to two reasons – either the total bandwidth has dropped, or a flow's demand has decreased. Bandwidth estimate should not be decreased in the latter case – reallocating the bandwidth now unused by the flow can increase observed throughput. Since there is no way to tell these two scenarios apart, we first let reallocation try to fix things before we reduce $estimated\_bw$. More specifically, as long as there is a saturating flow (that can potentially use more bandwidth), CRAB keeps trying to reallocate excess capacity. If no flow can be classified as saturating and $total\_observed\_tpt$ remains lower than $estimated\_bw$, [6] it can assume that the bandwidth has dropped and reduces $estimated\_bw$ to the $total\_observed\_tpt$. This assumption can still be incorrect because it is possible that it is not the bandwidth, but the demand for all the flows that have actually decreased. However, in such a case, decreasing $estimated\_bw$ does not hurt the flows, and later when flows grow back, we can re-estimate bandwidth through bandwidth probing as discussed ahead.

After updating $estimated\_bw$, CRAB resets global excess bandwidth to zero and assigns each flow its absolute weighted share of the new bandwidth estimate. This eradicates the effect of erroneous reallocations that happen before decreasing the bandwidth estimate. Correct reallocation (if needed) can then take place in subsequent iterations of the control loop.

Increase in Bandwidth. Detecting an increase in bandwidth is particularly tricky because CRAB itself limits the arrival rates of flows by throttling them. Thus, it needs to explicitly probe for more bandwidth. To do so, CRAB increases the assigned rate of a saturating flow and then waits to take a throughput measurement. If it detects any significant increase in $total\_observed\_tpt$, [7] it updates $estimated\_bw$ to $total\_observed\_tpt$. It accordingly computes the absolute weighted share for each flow (setting it as the flow's true and assigned bandwidth). It then accordingly increases the global excess bandwidth and redivides the bandwidth across all flows in proportion to their weights until their demands are satisfied (as in §3.3).

The above requires careful consideration of two aspects – which saturating flow should we select for bandwidth probing and what should the increment value be. We select a new saturating flow in a round-robin fashion in each bandwidth probing round, such that any one flow does not get an advantage over the other. We calculate the increment in proportion of $estimated\_bw$ i.e. $increment = inc \times estimated\_bw$. To prevent large disruptions in flow shares, we start off with a small value of inc (0.125), but every time bandwidth probing results in an increase in estimated bandwidth, we double the value of inc. When probing does not result in an increase, we reset the increment to its starting value. We continuously keep probing for bandwidth until this happens. The only time we terminate bandwidth probing prematurely is if we detect a flow to be growing, in which case we skip to reclamation.

Bootstrapping. CRAB bootstraps by calculating weighted fair share rates of flows based on an arbitrary initial estimate of bandwidth. By keeping this estimate large, we can avoid doing

[6]To be robust against minor throughput changes, the precise condition we check for is $total\_observed\_tpt < 0.9 \times estimated\_bw$.

[7]If the increase in $total\_observed\_tpt$ is greater than $\max(0.1 \times total\_observed\_tpt, 1\text{Mbps})$

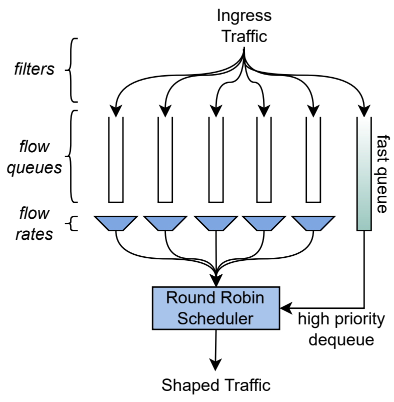


Figure 5: Linux's HTB scheduler.

bandwidth probing at startup time. Once the first measurement interval is over, CRAB is able to fix this estimate. CRAB can maintain a historical average estimate of bandwidth in the persistent state to feed as an initial value for more efficient bootstrapping.

3.6 CRAB's Router Control Loop

CRAB runs the same control loop at the home router, except for one change: each throughput measurement takes $3n$ observations, so the length of throughput measurement is $3n \times t$ s. This ensures that CRAB in end-devices is able to adjust their flow rates to per-device rate changes made by the home router, before the router's next measurement.

4 System Implementation

We implement CRAB's end-host logic on a 2.4GHz 8-core Ubuntu 20.04 machine with Linux 5.11 kernel, and its home-router logic on a 1.8GHz dual-core Linksys WRT3200ACM router running Linux-based OpenWRT firmware. We start with discussing the end-host implementation (§4.1-§4.5), and then discuss router implementation (§4.6).

4.1 CRAB's Placement

The inbound traffic arrives at an ethernet (eth) or wireless (wlan) interface. Since Linux does not have rich options to shape ingress traffic, we redirect the inbound traffic to an intermediate function block (ifb) [6] interface, where CRAB can shape traffic using Linux TC [10] (§4.3). The shaped traffic then gets picked up by the receiver's TCP/IP stack for further (normal) processing.

4.2 Throughput Measurement

We measure the flow throughputs (or arrival rates) by using scapy [16] to sniff and record the incoming traffic at the original ingress interface (eth or wlan).

4.3 Rate Enforcement

We enforce the computed weighted fair share rates for each flow group at the ifb interface using Linux's HTB (Hierarchical Token Bucket) scheduler [5].

```
Flow Groups : {
      Video Streaming: {
            flow identifiers: ["app>netflix", "web>youtube.com", "web>hulu.com"],
            weight: 5
      },
      Work: {
            flow identifiers: ["app>dropbox", "ip>1.2.3.4"]
            weight: 5
      },
      Default:{
            weight: 1
      }
}
```

Figure 6: An Example of a CRAB Config File.

Primer on HTB. Figure 5 shows the basic working of HTB. HTB allows a user to classify traffic into different classes (based on filters defined by packet header fields such as IP address, protocol type, TCP/UDP ports, etc) and specify different rates for throttling each class. If an incoming packet cannot be classified into a class defined by the filter rules, it is put into a special queue called the *fast queue*. Each class consists of a FIFO queue (to buffer packets) and a token bucket filter. Tokens are added to the bucket at the specified rate. If the amount of tokens in the bucket is greater than or equal to the size of the head packet in the queue, then the packet is dequeued. Otherwise, the queue blocks. If the queue is full, new incoming packets for that class are dropped.

A round-robin scheduler moves between classes to dequeue packets. If a class cannot dequeue a packet because it does not have enough tokens, the scheduler moves to the next class without blocking. The scheduler prioritizes dequeuing from the fast queue before any of the HTB classes.

HTB supports work-conserving traffic shaping by allowing unused tokens to be borrowed by other classes – we do not enable this feature in CRAB for reasons discussed in §2.2, and use the re-allocation logic described in §3 instead.

HTB in CRAB. We maintain a class for each flow group. The flow filter manager (detailed in §4.5) installs the filter rules for classifying incoming packets into their respective classes. We set the throttling rate for each class to the weighted fair share rate of its respective flow-group (as computed by CRAB's control loop), and accordingly adjust the queue size. [8]

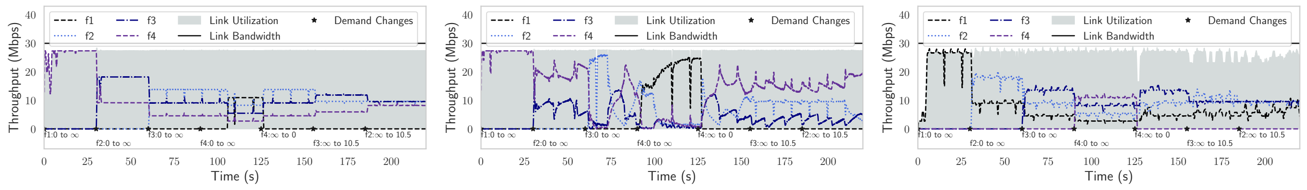
4.4 User Interface

The user specifies their preferences in a config file. Figure 6 shows a sample config file. It contains different flow groups, where each flow group consists of a list of flow identifiers and a weight associated with the flow group. Our current implementation allows a user to give three kinds of flow identifiers – *ip* (for traffic coming from the specified IP address), *app* (for traffic destined to the specified application), and *web* (for traffic destined for webpages from the specified web domain).

4.5 Flow Filter Management

Flow filter manager (FFM) maps user-specified high-level flow identifiers into packet header fields that can be used for

[8]Queue size is adjusted to 2BDP = 2 × rate × RTT, where we assume RTT to be 50ms.



(a) WFQ@bottleneck **(b)** Status Quo **(c)** CRAB@end-host

Figure 7: Weighted sharing of 30 Mbps bottleneck between 4 flows in a ratio of 4:3:2:1, where f1 has a weight of 1 and f4 has a weight of 4. The bottom part of each graph shows flow demand changes in Mbps, where ∞ means unknown demand.

filtering packets at HTB. When we encounter a packet from an unknown *source* IP address (that does not correspond to an installed filter rule), we copy the packet header to FFM. FFM determines the mapping for the packet (as detailed below) and installs a new filter rule for it. Note that FFM installs filter rules asynchronously, and we do not block traffic while this happens. Instead, the unclassified packet is put (and served) in HTB's fast queue. Once FFM installs the corresponding rule, it is used to correctly classify future packets from that flow. This ensures that if the unclassified packets are of a potentially important flow, their service is not degraded. Typically, FFM is able to install new filter rules fast enough, such that only the first couple of packets for an unclassified flow land up in the fast queue. FFM installs filter rules as follows:

(i) Source IP Mapping: If the source IP address of a packet arriving at the ingress matches with an *ip* field in the config file, we install the corresponding filter simply based on that.

(ii) Application: If the source IP is not found in the config file, we use psutil [7] to reverse map the destination port on the packet header to find which application has opened that port. If the application is specified in any flow group in the config file, we install a filter that maps traffic from the packet's source IP address to this flow group. If the application does not match any identifier in the config file, we map it to the default flow group (unless the app is the web browser, which we handle as a special case as discussed below)

(iii) Web Domain: Mapping packets to the web pages (identified by web domains) is tricky for multiple reasons:

(a) Web pages from the same web domain may be hosted on several different machines.

(b) Web pages make many cross-origin requests e.g. to CDNs and ad servers. Since these requests are often dynamic (e.g. due to load balancing in CDNs or real-time bidding in ad networks), it is not possible to pre-populate a list of IP addresses a webpage from a certain web domain would access.

(c) Packet headers do not carry any information about which web domain the packet belongs to. The packet payload of HTTPS traffic, which does carry some information, is encrypted at the ingress (where CRAB sniffs) and is decrypted only at the browser.

Thus, if the application using the destination port is a web browser, FFM needs more context from the browser to correctly classify the packet. We built a Google Chrome plugin that provides this context. As soon as the browser starts receiving an HTTPS response, the plugin prepares and sends a message to FFM that contains the source IP address of the response, and the URL of the webpage that initiated the corresponding HTTPS request. Using this mapping, the FFM can extract the web domain from the URL and find its match in the config file. If there is a match, we install a filter for the source IP address, mapping it to the corresponding flow group. Otherwise, we map it to the default flow group.

FFM may not be able to correctly classify packets if the relevant packet header fields are encrypted (as in the case of VPNs). In such cases, application integration similar to the plugin we built for Google Chrome can help remove FFM's dependency on encrypted packet headers and enable the classification of non-encrypted fields. Note that DNS encryption does not affect FFM, as it does not rely on DNS packets.

4.6 Home Router Implementation

Similar to our end-host implementation, CRAB at the router sniffs incoming traffic (using tcpdump [12]) at the ingress (eth) interface, and redirects the traffic to the ifb interface. It classifies the traffic based on the destination IP address at the ifb interface and enforces the per-destination rates computed by CRAB's control loop using Linux HTB.

5 Evaluation

We now evaluate the following:

• CRAB's ability to adapt to changes in flow demands and link capacity in synthetic scenarios involving real-world bulk flows (§5.1).

• Performance (QoE) improvement enabled by CRAB for real-world video streaming (§5.2) and web browsing (§5.3), when competing with bulk downloads.

• An alternative way of using CRAB's framework to enforce user preferences, and the trade-offs involved (§5.4).

• The need for CRAB's home router logic with multiple active devices in the user's domain (§5.5).

• The impact of changing CRAB's key parameters, i.e. throughput observation length and lending headroom (§5.6).

• CRAB's robustness to diverse traffic characteristics and its overheads (summarized in §5.7, and detailed in the appendix).

We use the same setup as in §2.2, that models a single end-host directly attached to the ISP's link (for repeatable experiments, we emulate a 30Mbps access link by throttling traffic at our home router). The only exception is §5.5, where we extend the home router logic to implement CRAB after the throttle point.

Unless otherwise specified, we compare CRAB with two baselines: (i) ideal WFQ (implemented at the access link emulated by our router), and (ii) status quo (i.e. no traffic shaping). We also conducted experiments using HTB with bandwidth borrowing *after* the bottleneck – since this produces almost exactly the same outcome as the status quo (as discussed in §2.2), we omit presenting those results.

Our workloads span real-world flows with a diverse set of sender-side rate control mechanisms: (i) bulk download flows that likely use TCP Cubic [30], (ii) YouTube video streaming that uses BBR [11, 22] along with an adaptive bitrate (ABR) algorithm for adapting video quality, and (iii) web page-loads over Google Chrome that potentially use a mix of BBR [22, 33] and Cubic [30] over QUIC [33, 39] and TCP [9].

5.1 CRAB in action

We design synthetic scenarios to visualize CRAB's fine-grained reaction to changes in flow demands and/or link bandwidth, using real-world flows that download large Linux images from different servers. We configure each download as a separate flow group with different weights. Each flow individually has a demand higher than 30 Mbps as it is a backlogged flow with no server-side bottleneck.

Testing Reallocation and Reclamation. We test a scenario with 4 flows sharing a 30 Mbps link, with a desired sharing ratio of 4:3:2:1 between them. We emulate dynamic flow demands by shaping traffic at two interfaces in the home router. The first interface throttles rates of individual flows (to emulate flow demands limited by low sending rates or other upstream bottlenecks). The second interface then cumulatively throttles all the traffic to 30 Mbps, emulating an ISP's access link (as mentioned before).

Figure 7a shows the flow shares when we do WFQ at the ISP, which sets a perfect, but impractical baseline. Flow 1, which has the lowest weight of 1, starts at 0 seconds. Because there is no other active flow, it gets to utilize the entire link bandwidth. Flows 2, 3, and 4 become active after every 30 seconds respectively, and at each point, the link is shared in the proportion of active flows' weights. At 125 seconds, flow 4 stops, and link bandwidth is redivided between the remaining three flows in the proportion to their weights. At about 155 seconds, flow 3's demand drops to 10.5 Mbps, and its remaining share is taken up by flow 1 and flow 2. At 185 seconds, flow 2's demand also drops to 10.5 Mbps, at which point flow 1 gets all the remaining unused share.

Figure 7b shows how the flows share the link arbitrarily without any shaping with the status quo.

Figure 7c shows that, on the whole, CRAB is able to imitate

[9]We can only guess the protocols used by different content providers based on public knowledge.

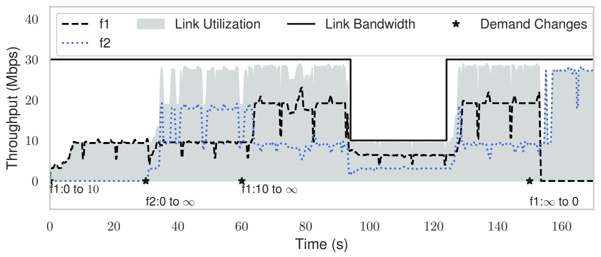


Figure 8: 2 flows sharing a link in 2:1 ratio with CRAB. Flow 1's demand and link bandwidth vary over time.

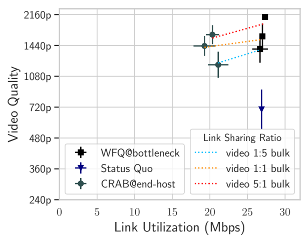
the ideal WFQ baseline very closely despite being at the other side of the bottleneck (although there are some transient, and inevitable, dips in link utilization).

Testing Bandwidth Estimation. In Figure 8, we evaluate CRAB's reallocation and reclamation, in addition to bandwidth estimation due to varying link bandwidth. We have two flows configured to share a 30Mbps link in the ratio of 2:1. CRAB builds a spuriously low estimate of bandwidth when flow 1 (with demand limited to 10Mbps) starts at 0 seconds. CRAB is able to probe for more bandwidth once flow 2 (with a demand more than 20Mbps) starts at 30s. It reallocates the unused bandwidth of flow 1 to flow 2 as well. Flow 1's demand then increases at 60 seconds, CRAB reclaims its lent bandwidth and the link bandwidth is correctly shared in a ratio of 2:1 between flows 1 and 2 respectively. Then the link bandwidth drops to 10Mbps at 90 seconds, CRAB detects this change and adjusts the flows' rates to 6.66Mbps and 3.33Mbps respectively. When bandwidth increases again to 30Mbps at 125 seconds, CRAB is able to probe for more bandwidth and divide it according to the flows' weights. Finally, when flow 1 stops at 155 seconds, its bandwidth is reallocated to flow 2. Appendix D presents a similar experiment, except that instead of starting a single flow f2 at 30s with a demand of more than 20Mbps, we start 3 new flows, each with a demand of 10Mbps.

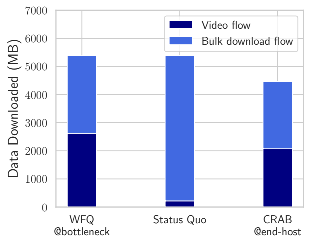
5.2 Video Streaming

We repeat the experiment in §2.2 with 7 different Youtube videos of varying playtime [10], competing with bulk download. Figure 9a reports the average video quality and link utilization across all experiments. We record the quality of each video over time using Youtube's API [13]. We then calculate the average video quality for each video by averaging the video quality weighted by the amount of time played at that quality. We calculate the average link utilization as the sum of data received during the video playback, divided by playback time. For CRAB and WFQ, we also try different weight assignments between video flow and bulk downloads, 5:1, 1:1, and 1:5 respectively. Figure 9a shows how CRAB maintains comparable video quality to WFQ for each weight assignment setting (i.e. within [92-94]% of WFQ), but with [15-20]% lower link utilization compared to status quo. The

[10]shortest video is 1 minute, while longest is 10 minutes



(a) Average video quality vs Average throughput



(b) Distribution of downloaded data

Figure 9: Video streaming in presence of bulk downloads.

video quality achieved by status quo is worse than that with CRAB even with weighted sharing of 1:5 between video and bulk flows (this indicates that the video flow gets less than 17% of bandwidth share with the default status quo). [11]. Figure 9b shows the cumulative amount of video and bulk flow data downloaded across all videos for the experiment with a weighted sharing of 5:1 between video and bulk download. With CRAB and WFQ, video flow consumes almost half of the data which translates to much higher video quality. In comparison, with status-quo, video amounts to only 3% of total downloaded data.

5.3 Web Browsing

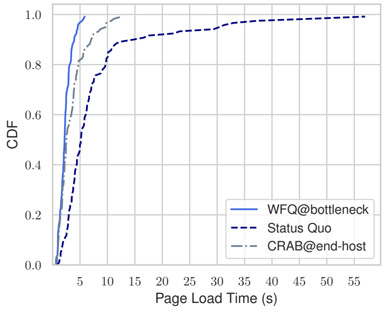
In this experiment, we show how CRAB helps improve web page load times despite background download flows. We emulate a user's browsing behavior by visiting 125 webpages (around 300 MBs of data) in total from 4 popular web domains (facebook.com, google.com, bbc.com, yahoo.com) in different sessions of browsing using Selenium [8] [12]. We separate each session by a Poisson inter-session time determined with a mean of 60 seconds. Within each browsing session, we separate each web page's download by a Poisson distribution with a mean of 5s to emulate a user's page read-time. We download two competing large files from two different servers. We throttle the access downlink to 10 Mbps at the router for these experiments and configure weights in the ratio of 7:3 between web traffic and bulk downloads. To fairly compare link utilization, we run the experiment for each baseline for the same amount of time. Figure 10a shows the CDF of page load times with CRAB, ideal WFQ, and status quo. The median page load time with CRAB is $2\times$ smaller than with the status-quo and is within 15% of ideal WFQ. Figure 10b shows that CRAB under-utilizes the link by about 13%.

5.4 Alternative way of using CRAB's framework

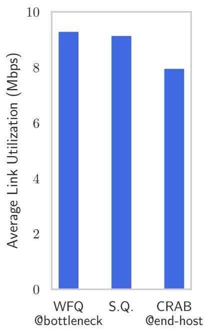
We now evaluate the alternative mechanism for using CRAB's framework (referred to earlier in §2), where we directly throttle the cumulative rate of all flows arriving at the ingress to a value lower than the overall link capacity that CRAB

[11]Lower video throughput translated to lower resolution in all cases, and we did not notice any re-buffering events.

[12]Selenim allows us to automate webpage loads and user clicks.



(a) Page Load Times



(b) Link Utilization

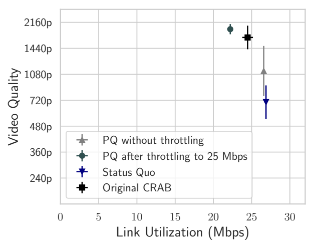
Figure 10: Page load times vs link utilization for CRAB, WFQ and Status quo

estimates (via its throughput observation and slightly modified bandwidth probing logic), and then enforce the desired scheduling policy on the artificial bottleneck that gets created. To understand the trade-offs involved with this approach, we evaluate it under two different scenarios. We emulate (and assume) a static link capacity of 30Mbps, and do not implement bandwidth estimation for the alternative approach for simplicity. We also disable bandwidth estimation in the original CRAB implementation for a fairer comparison.

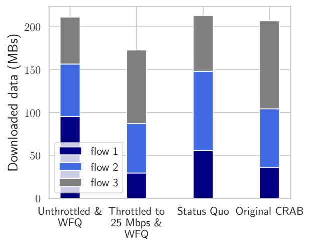
(a) We first consider the scenario from §5.2, where a YouTube video competes with bulk download on a bottleneck link with a capacity 30Mbps. We prioritize the video flow at the receiver without throttling the flows in one case, and after throttling the incoming flows to a cumulative rate of 25Mbps in the other. We compare these strategies with the status-quo (that does not enforce user preferences) and the original CRAB design. Figure 11a shows the results. Prioritizing the video flow without throttling cannot enforce user preferences very effectively (for reasons discussed in §2). However, prioritizing the video flow after throttling the incoming traffic to a rate of 25Mbps (which is lower than the link capacity) is effective. It results in slightly higher video quality but slightly lower link utilization than the original CRAB design. [13]

(b) We next evaluated a scenario where the 30Mbps bandwidth is to be divided across three backlogged flows in the ratio 1:2:3. We now apply weighted fair queuing at the receiver without throttling, and after throttling the incoming traffic to 25Mbps, and compare the outcomes with original CRAB and the status-quo (Figure 11b). Again, we observe that the desired shares could not be effectively enforced without throttling the flows to a rate lower than the link capacity. WFQ applied after throttling at 25Mbps was able to enforce the desired flow shares similar to the original CRAB. However, the original design achieved 19% higher link utilization

[13]The difference in video quality potentially stems from the difference in scheduling policy – strict prioritization vs 5:1 weighted fair sharing with original CRAB.



(a)



(b)

Figure 11: Creating chokepoint by throttling to less than known link capacity **(a)** helps control traffic (e.g. by using priority queues) **(b)** but results in avoidable bandwidth wastage.

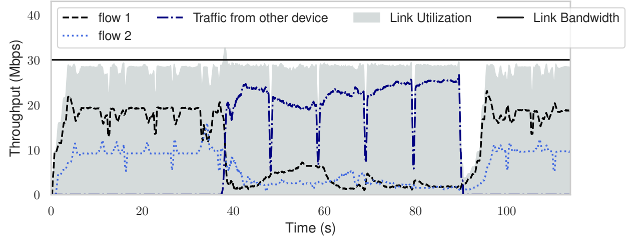
(which was very close to the status quo).

These results highlight that the original CRAB design *strives* to achieve maximal link utilization. The link underutilization is transient and less notable when flow demands are stable (as in the second scenario), and is more notable when flow demands vary due to more frequent re-allocation and reclamation (as in the first scenario). In contrast, the alternative approach of throttling the cumulative rate of incoming flows will consistently suffer from lower link utilization by design. The amount of link underutilization can be reduced by reducing the gap between the throttling rate and the link capacity, but this would also impact how effectively user preferences get enforced (e.g. resulting in lower video quality for the first scenario). This makes it difficult to correctly configure the cumulative throttling rate, especially as link capacity varies or is estimated imprecisely. Nonetheless, this alternative design effectively demonstrates the potential of using CRAB's framework in more than one way.

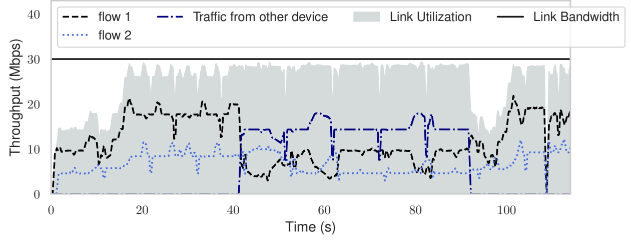
5.5 Multiple end-devices need CRAB at home-router

In this section, we show the need for CRAB at the home router to ensure proper enforcement of bandwidth shares when there are multiple devices actively using the Internet in the user's domain. We connect two machines (M1 and M2) to the home router. M1 runs CRAB to enforce 2:1 weights between two bulk download flows, while M2 does not run CRAB. Initially, we just have two flows from M1 sharing the bottleneck link in the 2:1 ratio enforced by CRAB. When the flow from M2 starts at around 40 seconds, in absence of CRAB support at the home router, it ends up stealing M1's bandwidth share (as shown in Figure 12a). When CRAB at M1 throttles its lower weight flow, the bandwidth yielded by this flow at the access link is taken up by the flow from M2, instead of the other higher-weighted flow at M1. [14] With CRAB enabled at the home-router, CRAB at an individual device can correctly control how its router-enforced bandwidth share is divided between its flows (as shown in Figure 12b). Thus, in case of multiple devices sharing the home Internet connection, it is important to enable CRAB at the home router to enforce

[14]Note that sender side protocols to yield bandwidth [41, 48] would suffer from a similar issue.



(a) Without CRAB@Router



(b) With CRAB@Router

Figure 12: With multiple active devices, CRAB at the home router is required to ensure correct working of CRAB at the end-host.

bandwidth shares across different devices, and to prevent the devices from stealing bandwidth from one another.

5.6 Impact of CRAB's Parameters

The value of n (number of observations) $\times\, t$ (observation interval) determines how long we spend in estimating throughput, before making a change in assigned rates. Figure 13 shows the effect of changing it from its default value of $(5 \times 0.2s)$ to higher $(10 \times 0.3s)$ and lower $(5 \times 0.1s)$ value for the video streaming experiment from §5.2. Higher value of $n \times t$ means we are much slower in our reactions – we reallocate late which improves video quality (very slightly) but at the cost of greater link under-utilization. In contrast, a smaller value of $n \times t$ implies quicker decisions – we have slightly better link utilization, but video quality also slightly drops. If we keep making observation length smaller, it would boil down to doing instantaneous reallocation similar to bandwidth borrowing with HTB (which, like status-quo, can maintain high link utilization, but cannot enforce bandwidth shares).

We also experimented with changing CRAB's lending headroom parameter from its default value of 0.25Mbps to higher (0.5Mbps) and lower (0.05Mbps) values. This had no significant impact on CRAB's performance – we present detailed results in Appendix E.

5.7 Other Results

We briefly summarize some of our other results, providing the details in the appendix:

• CRAB is quite robust to differences in RTTs and congestion control algorithms across flows, and it scales well with the increasing number of flow groups (Appendix C).

• CRAB has a negligible impact on packet delay and forwarding rates. It has a CPU utilization of 10.74% on a 2.4GHz

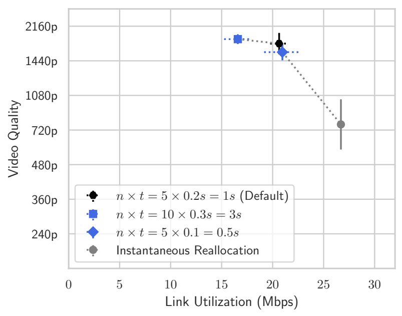


Figure 13: Effect of observation length ($n \times t$) on video quality vs link utilization.

8-core machine, which largely stems from the throughput measurement module (Appendix F). This is because our current implementation uses Scapy [16], a Python-based packet sniffer. Using a more efficient sniffer (e.g. libtins [4]) would reduce this overhead.

6 Related Work

There have been a number of proposals for enabling differentiated services (in the form of weighted fair sharing or prioritization) at network switches [17, 21, 25, 49, 53, 56]. However, these policies must be applied at the bottleneck, which is controlled by the ISP and not the users. There exist proposals that allow a user to send their preferences to the ISP [19, 24, 27, 35, 54] which are difficult to deploy in practice. CRAB allows the user to control the access bottleneck *without seeking any support from the ISP*.

There exist mechanisms for the device to control the uplink bandwidth usage when sending data [5, 44, 45], e.g. prioritizing latency-sensitive uploads over file backups [14] – the bottleneck occurs at the user device in these cases. Another category of work allows the end user to configure their home routers to do traffic prioritization [18, 40, 47], assuming that the bottleneck is at the wireless link in the home network. CRAB tackles the harder problem of controlling downlink bandwidth usage by shaping traffic *after* the bottleneck (that is likely to occur at the access link from the ISP), and naturally helps in scenarios where the bottleneck is at the home-router.

With bottlenecks at the ISP, it can even be challenging to do sender-side traffic prioritization. Bundler [20] solves this problem in context of site-to-site traffic by estimating the bottleneck rate in the ISP and enforcing that rate at the sender. This shifts the bottleneck at the sender's site instead of the ISP, which lets the sender enforce its desired scheduling policies. CRAB enforces desired bandwidth shares solely from the receiving domain, without seeking any explicit coordination with the senders.

Receiver-driven protocols [28, 42, 55] provide a receiver with greater control over their downlink bandwidth, by letting them explicitly dictate the sending rates. Some senders can also use bandwidth-yielding protocols (e.g. [41, 48]), if they know their flow has a relatively lower priority. However, the onus of using these receiver driven or yielding protocols is on the senders – a receiver can use these protocols only if the senders also support them. CRAB allows receivers to unilaterally control their access bandwidth shares.

7 Conclusion and Discussion

This paper presents CRAB, a system that enables end-user to unilaterally control how their Internet access bandwidth is shared across their incoming flows. In particular, we show how home users can exploit CRAB to enforce their preferences and achieve better performance for their video and web flows. Our source code is publicly available. [15] Our work opens up several interesting future directions:

Theoretical analysis of performance. Formal characterization of CRAB's performance, e.g. by analyzing the upper-bound on link utilization for effective enforcement of user-specified shares under different scenarios, can inform future designs for improved performance.

Other deployment modes. CRAB does not require any explicit coordination among the home router and the endpoints. This extends CRAB's utility to scenarios where multiple users share a common Internet connection, e.g. in coffee shops, enterprises, airports, etc. The domain owners can advertise their use of CRAB at the access routers for enforcing fairness across users (they can also use other scheduling mechanisms at the routers [1, 15, 18] if it is known that the bottleneck is at the downlink from the router to the end-devices). Each user can then use CRAB at the endpoint to independently control how their share of bandwidth is divided across their flows.

Setting Flow Weights. It might be difficult for users to set the appropriate weight for a flow group that CRAB requires as an input. Future work can explore how to design a more intuitive user interface. For instance, we can auto-classify incoming flows across broad categories (video streaming vs browsing vs downloads, etc), and then automate weight assignments based on coarse-grained user preferences across these categories and learned estimates of bandwidth requirements for different flows. Such bandwidth requirements are already known for many standard applications, e.g. video streaming [2, 9]. CRAB can also ship with some default configurations for popular traffic classes, which can be further customized by users according to their needs.

Support for phones. We currently implement CRAB on a Linux PC. We plan on porting our system to Android phones.

8 Acknowledgements

We would like to thank our shepherd, Srikanth Kandula, and the anonymous NSDI reviewers for their insightful comments. We would also like to thank Sachin Ashok, Hari Balakrishnan, Brighten Godfrey, Akshay Narayan, Aurojit Panda, Scott Shenker, Deepak Vasisht, and Tianyin Xu, for their helpful feedback on the paper. This work was supported by Intel, Facebook, and AG NIFA under grant 2021-67021-34418.

[15] https://projectcrab.web.illinois.edu.

References

[1] How qos improves performance? https://bit.ly/3f7IdXO.

[2] Internet connection speed recommendations. https://help.netflix.com/en/node/306.

[3] Internet speed around the world. https://www.speedtest.net/global-index.

[4] Libtins. http://libtins.github.io/benchmark/.

[5] Linux hierarchical token buckets. http://luxik.cdi.cz/~devik/qos/htb/.

[6] networking:ifb [wiki]. https://wiki.linuxfoundation.org/networking/ifb.

[7] Psutil. https://pypi.org/project/psutil/.

[8] Selenium with python¶. https://selenium-python.readthedocs.io/.

[9] System requirements - youtube help. https://support.google.com/youtube/answer/78358?hl=en.

[10] tc(8) - linux manual page. https://man7.org/linux/man-pages/man8/tc.8.html.

[11] Tcp bbr congestion control comes to gcp – your internet just got faster | google cloud blog. https://bit.ly/3qRKCsv.

[12] Tcpdump/libcap. https://www.tcpdump.org/.

[13] Youtube data api. https://developers.google.com/youtube/v3.

[14] 5 benefits and 3 drawbacks of using cloud storage for your baas offering. https://bit.ly/3qQSBGe, Mar 2018.

[15] Re: R6700v2 - where is downstream bandwidth control? https://bit.ly/3BYMAha, Dec 2018.

[16] Philippe Biondi and the Scapy community. Scapy. https://scapy.net/.

[17] Sj Blake, David Black, Mark Carlson, Elwyn Davies, Zheng Wang, and Walter Weiss. An Architecture for Differentiated Services. RFC 2475, 1998.

[18] Ilker Nadi Bozkurt and Theophilus Benson. Contextual router: Advancing experience oriented networking to the home. In *The Symposium on SDN research*, 2016.

[19] Ilker Nadi Bozkurt, Yilun Zhou, and Theophilus Benson. Dynamic prioritization of traffic in home networks. In *CoNEXT Student Workshop*, 2015.

[20] Frank Cangialosi, Akshay Narayan, Prateesh Goyal, Radhika Mittal, Mohammad Alizadeh, and Hari Balakrishnan. Site-to-site internet traffic control. In *EuroSys*, 2021.

[21] Zhiruo Cao and E.W. Zegura. Utility max-min: an application-oriented bandwidth allocation scheme. In *IEEE INFOCOM*, 1999.

[22] Neal Cardwell, Yuchung Cheng, C. Stephen Gunn, Soheil Hassas Yeganeh, and Van Jacobson. Bbr: Congestion-based congestion control. *ACM Queue*, 2016.

[23] Pedro Casas, Michael Seufert, Florian Wamser, Bruno Gardlo, Andreas Sackl, and Raimund Schatz. Next to you: Monitoring quality of experience in cellular networks from the end-devices. *IEEE Transactions on Network and Service Management*, 2016.

[24] Saoussen Chaabnia and Aref Meddeb. Slicing aware qos/qoe in software defined smart home network. In *IEEE/IFIP Network Operations and Management Symposium*, 2018.

[25] Alan Demers, Srinivasan Keshav, and Scott Shenker. Analysis and Simulation of a Fair Queueing Algorithm. *ACM SIGCOMM CCR*, 1989.

[26] Sally Floyd, Tom Henderson, and Andrei Gurtov. The newreno modification to tcp's fast recovery algorithm. 1999.

[27] Hassan Habibi Gharakheili, Jacob Bass, Luke Exton, and Vijay Sivaraman. Personalizing the home network experience using cloud-based sdn. In *IEEE International Symposium on a World of Wireless, Mobile and Multimedia Networks*, 2014.

[28] Rajarshi Gupta, Mike Chen, Steven McCanne, and Jean Walrand. A receiver-driven transport protocol for the web. *Telecommunication Systems*, 2002.

[29] Gabe Gurwin. Should you stick with console gaming, or make the jump into the cloud? https://bit.ly/3LxrHfX, Oct 2019.

[30] Sangtae Ha, Injong Rhee, and Lisong Xu. Cubic: a new tcp-friendly high-speed tcp variant. *ACM SIGOPS Operating Systems Review*, 2008.

[31] Ada Ivanova. Streaming vs. downloading: Which one should you use? https://bit.ly/3QWgWox, Aug 2022.

[32] Van Jacobson. Congestion avoidance and control. *ACM SIGCOMM CCR*, 1988.

[33] Matt Joras, Matt Joras, and Yang Chi. How facebook is bringing quic to billions. https://bit.ly/3UoGRZ5, Apr 2022.

[34] Ravi Kokku. {TCP} nice: A mechanism for background transfers. In *OSDI*, 2002.

[35] Himal Kumar, Hassan Habibi Gharakheili, and Vijay Sivaraman. User control of quality of experience in home networks using sdn. In *IEEE International Conference on Advanced Networks and Telecommunications Systems (ANTS)*, 2013.

[36] Ralf Kundel, Joerg Wallerich, Wilfried Maas, Leonhard Nobach, Boris Koldehofe, and Ralf Steinmetz. Queueing at the telco service edge: Requirements, challenges and opportunities. In *Workshop on Buffer Sizing*, 2019.

[37] Eymen Kurdoglu, Yong Liu, Yao Wang, Yongfang Shi, ChenChen Gu, and Jing Lyu. Real-time bandwidth prediction and rate adaptation for video calls over cellular networks. In *International Conference on Multimedia Systems*, 2016.

[38] Aleksandar Kuzmanovic and Edward W Knightly. Tcp-lp: A distributed algorithm for low priority data transfer. In *IEEE INFOCOM*, 2003.

[39] Adam Langley, Alistair Riddoch, Alyssa Wilk, Antonio Vicente, Charles Krasic, Dan Zhang, Fan Yang, Fedor Kouranov, Ian Swett, Janardhan Iyengar, Jeff Bailey, Jeremy Dorfman, Jim Roskind, Joanna Kulik, Patrik Westin, Raman Tenneti, Robbie Shade, Ryan Hamilton, Victor Vasiliev, Wan-Teh Chang, and Zhongyi Shi. The quic transport protocol: Design and internet-scale deployment. In *ACM SIGCOMM*, 2017.

[40] Jake Martin and Nick Feamster. User-driven dynamic traffic prioritization for home networks. In *ACM SIGCOMM workshop on Measurements up the stack*, 2012.

[41] Tong Meng, Neta Rozen Schiff, P Brighten Godfrey, and Michael Schapira. Pcc proteus: Scavenger transport and beyond. In *ACM SIGCOMM*, 2020.

[42] Venkata Padmanabhan. Coordinating congestion management and bandwidth sharing for heterogeneous data streams. In *NOSSDAV*, 1999.

[43] Abhay K Parekh and Robert G Gallager. A generalized processor sharing approach to flow control in integrated services networks: The multiple node case. *IEEE/ACM Transactions on Networking*, 1994.

[44] Sivasankar Radhakrishnan, Yilong Geng, Vimalkumar Jeyakumar, Abdul Kabbani, George Porter, and Amin Vahdat. {SENIC}: Scalable {NIC} for end-host rate limiting. In *USENIX NSDI*, 2014.

[45] Ahmed Saeed, Nandita Dukkipati, Vytautas Valancius, Vinh The Lam, Carlo Contavalli, and Amin Vahdat. Carousel: Scalable traffic shaping at end hosts. In *ACM SIGCOMM*, 2017.

[46] Henning Schulzrinne, Stephen Casner, Ron Frederick, and Van Jacobson. Rtp: A transport protocol for real-time applications, 1996.

[47] M Said Seddiki, Muhammad Shahbaz, Sean Donovan, Sarthak Grover, Miseon Park, Nick Feamster, and Ye-Qiong Song. Flowqos: Qos for the rest of us. In *HotSDN*, 2014.

[48] Sea Shalunov, Greg Hazel, Janardhan Iyengar, and Mirja Kuehlewind. Low extra delay background transport (ledbat). In *RFC 6817*, 2012.

[49] Madhavapeddi Shreedhar and George Varghese. Efficient fair queuing using deficit round-robin. *IEEE/ACM Transactions on Networking*, 1996.

[50] Srikanth Sundaresan, Nick Feamster, and Renata Teixeira. Home network or access link? locating last-mile downstream throughput bottlenecks. In *International Conference on Passive and Active Network Measurement*, 2016.

[51] Keith Winstein, Anirudh Sivaraman, and Hari Balakrishnan. Stochastic forecasts achieve high throughput and low delay over cellular networks. In *USENIX NSDI*, 2013.

[52] Gary R Wright and W Richard Stevens. *TCP/IP Illustrated, Volume 2 (paperback): The Implementation*. Addison-Wesley Professional, 1995.

[53] Haikel Yaiche, Ravi Mazumdar, and Catherine Rosenberg. A game theoretic framework for bandwidth allocation and pricing in broadband networks. *IEEE/ACM Transactions on Networking*, 2000.

[54] Yiannis Yiakoumis, Sachin Katti, Te-Yuan Huang, Nick McKeown, Kok-Kiong Yap, and Ramesh Johari. Putting home users in charge of their network. In *ACM Conference on Ubiquitous Computing*, 2012.

[55] Yasir Zaki, Thomas Pötsch, Jay Chen, Lakshminarayanan Subramanian, and Carmelita Görg. Adaptive congestion control for unpredictable cellular networks. In *ACM SIGCOMM*, 2015.

[56] Lixia Zhang. Virtual Clock: A New Traffic Control Algorithm for Packet Switching Networks. *ACM SIGCOMM CCR*, 1990.

[57] Xuan Kelvin Zou, Jeffrey Erman, Vijay Gopalakrishnan, Emir Halepovic, Rittwik Jana, Xin Jin, Jennifer Rexford,

and Rakesh K Sinha. Can accurate predictions improve video streaming in cellular networks? In *HotMobile*, 2015.

Appendix

A Pseudocode for Redivision of Excess Bandwidth

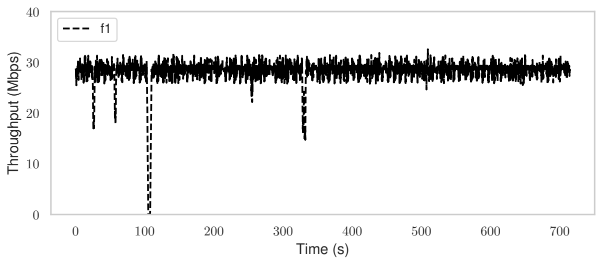
We first calculate the demand of each flow based on the amount of bandwidth it lends out. Then excess is divided based on this demand. If a flow's demand is fulfilled with bandwidth less than its share of excess, we can redivide this residual excess share between other flows.

Algorithm 1 Rediviving Excess Bandwidth between all flows

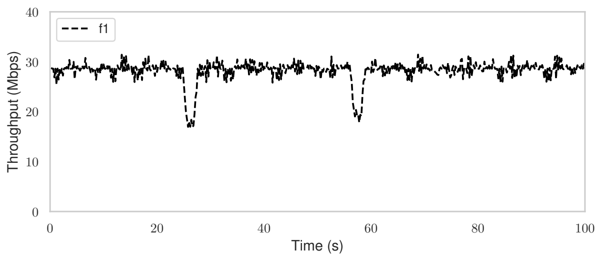
```
1:  procedure REDIVIDE(excess)
2:      // First we calculate demand of each flow based on
        the bandwidth it lends out
3:      for f in flows do
4:          if f.lent_bw > 0 then
5:              f.demand ← f.true_bw + f.borrowed_bw -
                f.lended_bw
6:          else
7:              f.demand ← ∞
8:          f.assigned_bw ← f.true_bw
9:          if f.demand > f.assigned_bw then
10:             f.lent_bw ← 0
11:             f.borrowed_bw ← 0
12:     // Based on the calculated demand, we divide excess
        between all flows. When a flow's demand is met, its
        residual excess is again divided between other flows.
13:     while excess > 0 do
14:         residual_excess = 0
15:         for f in flows do
16:             if f.demand > f.assigned_bw then
17:                 excess_share ← excess × (f.weight
                    /weight_sum)
18:                 f.assigned_bw ← f.assigned_bw + ex-
                    cess_share
19:                 f.borrowed_bw ← f.borrowed_bw + ex-
                    cess_share
20:                 if f.assigned_bw > f.demand then
21:                     residual_excess ← residual_excess +
                        (f.assigned_bw - f.demand)
22:                     f.lent_bw ← f.lent_bw +
                        (f.assigned_bw - f.demand)
23:         excess ← residual_excess
```

B Stability of Wifi Connection

Cellular networks are known to be highly unstable due to factors like high mobility and handovers. Wifi connections are relatively more stable. We evaluated this by sending IPerf data over UDP at a fixed rate of 30Mpbs to a Linux machine via a WiFi router. We measured the throughput over 200ms granularity at the ingress of the Linux end-host using tcpdump. Figure 14 shows the results. The observed throughput was largely stable with minor fluctuations around 30Mbps and only a handful of dips.



(a)



(b) Zoomed into first 100 seconds.

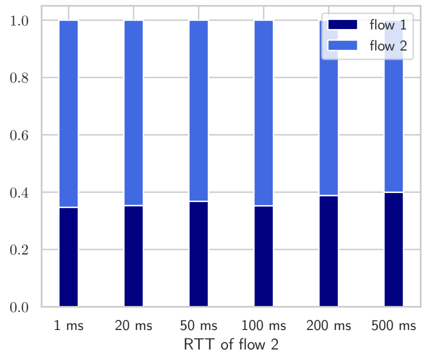
Figure 14: Throughput of a 30 Mbps flow over Wifi measured in 200ms intervals.

C Robustness to Different Traffic Characteristics

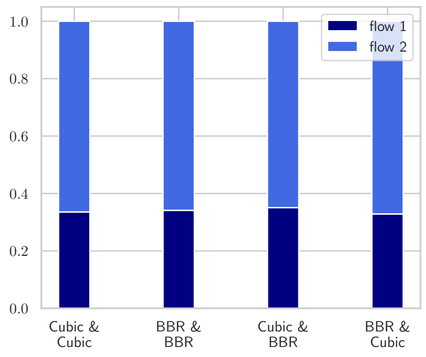
We now evaluate CRAB's performance under diverse traffic characteristics – flows with different RTTs, using different congestion controllers, and varying the number of flow groups. For these experiments, we generated iPerf flows with different characteristics using a local server, which then arrived at our receiver side setup used in our other experiments so far.

In the first experiment, we vary the RTT of flows by adding artificial delay in packet delivery using Linux tc at the server that generates flows. We start three backlogged flows sharing a 30 Mbps link in a 1:2:3 ratio. We fix the delay of the first flow (with weight 1) and the third flow (with weight 3) to 1ms and 50ms respectively, and vary the delay of the second flow (with weight 2) from 1ms to 500ms. We stop the third flow after 30 seconds and continue to run the other two flows until 100 seconds. We then study the effect of different RTTs for the first two flows as CRAB redivides the third flow's share between them in a 1:2 ratio. As shown in figure 15a, CRAB is pretty robust to the difference in RTTs. The slight mismatch in flow shares seen with an extremely high RTT difference of 200-500ms stems from the natural RTT unfairness that occasionally manifests in CRAB during the bandwidth probing phase when both flows share the bandwidth increment in a non-isolated manner.

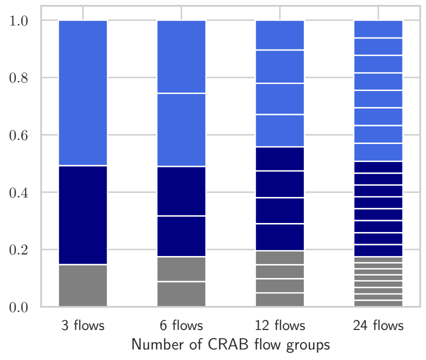
We use a similar setup as above for our second experiment, except that the flows now have the same RTTs (20ms), but use different congestion control algorithms. The third flow uses



(a) Varying RTT difference



(b) Different congestion control algorithms for both flows



(c) Number of active CRAB flow groups

Figure 15: CRAB maintains weighted sharing despite different characteristics of flows.

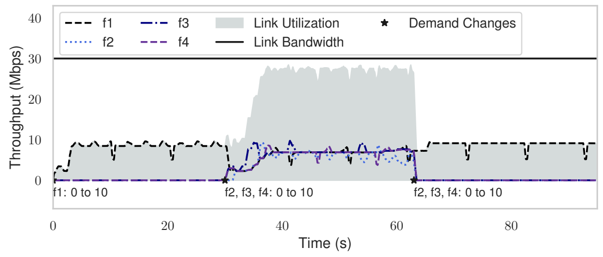


Figure 16: CRAB is able to estimate link capacity even with finite demand flows.

TCP Cubic and stops after 30 seconds. We vary the congestion control algorithms used by the other two flows as shown in figure 15b and observe how that impacts their flow shares. We find that CRAB's enforcement of weighted fair shares is robust to different congestion controllers. **[New]** CRAB is unaffected by the unfairness that could manifest because of RTT and congestion controller difference because it reacts at super-RTT time scales thus forcing underlying flows to adhere to throttled rates.

In the last experiment, we test CRAB's robustness as we increase the number of flow groups from 3 to 24. In the first run, we have three flows sharing a 60 Mbps link in a 3:2:1 ratio. In the next run, we double the number of flows associated with each weight, and so on. Figure 15c shows the bandwidth share received by each flow, with different colors indicating flows with different weights. We find that CRAB can effectively tackle a large number of flows. **[new]** As long as flows are large enough to react to CRAB, any number of flows can be handled by it. The only breaking point may be when a flow group consists of a large number of short-lived flows which finish before reacting to CRAB's throttling. However, such a case is unlikely to exist in our target scenario of a home network.

D Bandwidth Probing with Limited Demand Flows

Extending on our discussion in §5.1, here we evaluate the scenario when we do not have a convenient infinite demand flow to rely on for bandwidth probing. CRAB is still able to quickly probe for bandwidth by alternating between different finite demand flows for bandwidth probing. Figure 16 shows a scenario where we initially have one flow with a demand of 10 Mbps, at 30 seconds, 3 new flows each with a 10 Mbps demand start. Since their cumulative demand is more than 30 Mbps, the bandwidth probing algorithm is able to estimate link capacity by alternatively picking a flow for bandwidth probing and dividing capacity equally between them.

E CRAB's Sensitivity to Lent Bandwidth Headroom

The lent bandwidth headroom ensures that a flow has some room in the link to send at least a few packets so CRAB can detect it to be growing and reclaim for it. When the bandwidth of a flow is detected to be exceeding this headroom, CRAB quickly reclaims for it. Figure 17 shows CRAB's sensitivity to this parameter through the video experiment discussed in §5.2. Overall, CRAB is not very sensitive to this parameter, but

A larger value of headroom ensures better guarantees on early detection for reclamation, thus, slightly better video quality. However, overprovisioning may result in under-utilization, especially if we have a much higher number of flow groups. This effect can be avoided easily by having a cap on the collective headroom of all flow groups combined. A smaller value of headroom may not guard very well against pressure from other flows, which may result in CRAB not being able to detect flow growth in time and therefore slightly worse video quality. Another hidden effect that deteriorates link utilization in case of small headroom is spurious reclamations. Small values of headroom are not able to mask minor fluctuations and noise, which results in spurious reclamation, as a result,

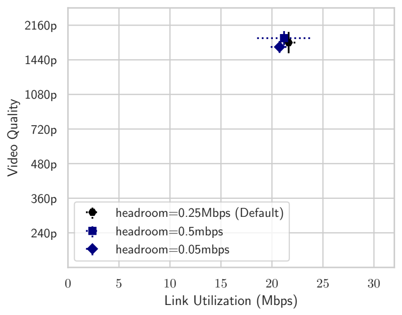


Figure 17: Effect of lent bandwidth headroom on video quality vs link utilization.

	With CRAB	No CRAB
Throughput	28.93 Mbps	28.98 Mbps
Delay	0.94 ms	0.88 ms
CPU Usage@end-host	10.74%	N/A

Table 2: Overheads of CRAB

we see slight link under-utilization. Overall CRAB is not very sensitive to this parameter.

F Overhead of CRAB

We evaluate CRAB's overhead by measuring the throughput and delay of a single bulk download flow with and without CRAB. To measure throughput, we record the flow's rate at ifb's egress (i.e. after shaping) with CRAB, and at eth0's ingress (as the raw arrival rate) without CRAB. We measure processing delay by recording the difference between timestamps for when a packet arrives at the eth0 and when its acknowledgment passes through eth0. Since CRAB's components are placed after the eth0 interface on the path of ingress traffic, this calculation captures any extra delay inflicted by CRAB. Table 2 shows that CRAB does not induce any significant overhead (the throughput remains largely unchanged, and the processing delay increases by only 0.06ms (i.e. 6.8% over baseline).

We also measure the CPU utilization of all CRAB threads during the experiment using Linux utility top. On a 2.4GHz 8-core machine, CRAB has an overall utilization of 10.34%. Almost all of the CPU usage stems from the throughput measurement thread of CRAB due to traffic sniffing. This is because Scapy, the Python-based packet sniffing library we use, copies the entire packet even though we just need access to a few packet header fields. The corresponding CPU overhead at the home router, which uses tcpdump for sniffing, is 16.65% on two cores at 1.8GHz. Writing a custom sniffer for CRAB that copies only a few packet header fields can potentially reduce the CPU overhead. We are working on shifting our throughput measurement module to a faster packet sniffing library like libtins [4].