Detecting if LTE is the Bottleneck with BurstTracker

Arjun Balasingam⋆, Manu Bansal⋆, Rakesh Misra†, Kanthi Nagaraj⋆,
Rahul Tandra†, Sachin Katti⋆, Aaron Schulman∞
⋆Stanford University   †Uhana Inc.   ∞UC San Diego

ABSTRACT

We present BurstTracker, the first tool that developers can use to detect if the LTE downlink is the bottleneck for their applications. BurstTracker is driven by our discovery that the proprietary LTE downlink schedulers running on LTE base stations allocate resources to users in a way that reveals if a user's downlink queue runs empty during a download. We demonstrate that BurstTracker works across Tier-1 cellular providers and across a variety of network conditions. We also present a case study that shows how application developers can use this tool in practice. Surprisingly, with BurstTracker, we find that the LTE downlink may not be the bottleneck for video streaming on several Tier-1 providers, even during peak hours at busy locations. Rather, transparent TCP middleboxes deployed by these providers lead to downlink underutilization, because they force Slow-Start Restart. With a simple workaround, we improve video streaming bitrate on busy LTE links by 35%.

CCS CONCEPTS

• **Networks** → **Application layer protocols**; **Network measurement**; **Mobile networks**.

KEYWORDS

Cellular; Bottleneck Detection; Scheduling; Middlebox

1 INTRODUCTION

When developers work on improving the performance of an application, they need to know where the bottlenecks are. For mobile applications, one of the key bottlenecks is the cellular network, namely, the LTE downlink from the base station to the user [12]. This is why the 3GPP[1] requires base stations to collect a metric that captures how often the downlink is the bottleneck for a user [1]. Unfortunately, this metric is only available for internal use by providers; it has never been available to developers[2].

In this paper, we present BurstTracker, a tool for mobile application developers to detect if the bottleneck is the LTE downlink—and thus out of the developers' control—or somewhere else that could be within the developers' control. BurstTracker identifies *bursts* within a prolonged transfer on the LTE downlink. Bursts are contiguous periods of a transfer during which the user's queue at the base station is nonempty; therefore, when an application has a burst, the downlink is the bottleneck.

The primary insight behind BurstTracker is the following: the LTE downlink scheduling algorithms running on base stations [4] allocate radio resources to users in a pattern that reveals when the user's downlink queue at the base station is nonempty. In each downlink time slot, schedulers prefer to allocate a large number of downlink resources to a small number of users. Therefore, a user can determine that its queue at the base station has run empty when the scheduler only allocates a small number of resources (just enough to empty out what remains in the user's queue). In Section 3.2.3, we show that this behavior exists across providers and network conditions.

BurstTracker is a client-side tool: it finds bursts in local traces of a user's downlink resource allocations (collected by MobileInsight or QXDM [25, 29, 32]). We demonstrate that BurstTracker can accurately detect the duration of bottleneck periods during a transfer with a median error of 7% by comparing against ground truth from a test base station.

To evaluate how useful BurstTracker is in practice, we investigated if LTE is the bottleneck for video streaming applications. We chose to study video streaming performance

---

[1]The *3rd Generation Partnership Project* (3GPP) is the standards organization behind LTE (4G) and its successor NR (5G).
[2]More in Section 2.2.

because it is an area that has received significant attention recently [10, 26, 34, 36–38]. In essence, we wanted to see if BurstTracker can identify a previously unknown issue affecting video streaming performance on LTE networks.

Surprisingly, BurstTracker indicated that often the LTE downlink was not the bottleneck, even when the link was congested. Instead, we found that three Tier-1 U.S. providers (AT&T, Verizon, and T-Mobile) appear to be operating transparent split-TCP middleboxes that impact the performance of video streaming. Specifically, the middleboxes of two providers (AT&T and Verizon) appear to be the bottleneck, by forcing Slow-Start to occur frequently during video streaming. This phenomenon is known as Slow-Start Restart (SSR) [2].

SSR is known to cause video streaming applications to underestimate available bandwidth on congested links [27]. In addition, we demonstrate that SSR forces the application to enter a negative feedback loop that leads to persistent resource underutilization. With only a simple workaround to disable SSR, we observed video streaming bitrate improve by up to 35%, even on busy LTE links.

In summary, we make the following contributions:

(1) We observe that the scheduling patterns of LTE base stations reveal the status of each user's downlink queue (Section 3). We also demonstrate that these scheduling patterns are not specific to a base station's operator (Verizon, AT&T, and T-Mobile) or traffic characteristics (unloaded and busy).

(2) We describe an algorithm for determining when the LTE link is the bottleneck, and we use it to implement a client-side LTE bottleneck estimator called Burst-Tracker (Section 3). We demonstrate that BurstTracker can estimate the time periods that the LTE downlink is the bottleneck with 93% accuracy.

(3) With BurstTracker, we discovered that many Tier-1 providers in the U.S. appear to operate middleboxes that perform SSR on HTTP and HTTPS flows (Sections 4 & 5).

(4) We observed that SSR causes video streaming applications to underutilize the LTE downlink due to a negative feedback loop in bitrate selection (Section 5).

2 MOTIVATION AND REQUIREMENTS

The inspiration for this work came from a surprising observation we made while streaming video on a smartphone (LG G3) connected to an LTE base station in a busy downtown location. We instrumented Google's ExoPlayer [13] to report the downlink throughput measured across each video segment while streaming a two-minute video. In this environment, the average throughput measured by ExoPlayer
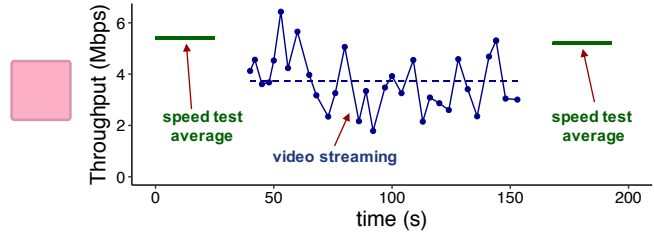
Figure 1: On a busy LTE network, we observed that a video stream had an average throughput of only 3.8 Mbps, while speed tests before and after averaged 5.4 and 5.2 Mbps

was 3.8 Mbps[3]. Fifteen seconds[4] before and after the video, we ran Ookla speed tests [28], and they reported 5.4 Mbps and 5.2 Mbps respectively (Figure 1).

This disparity in video and speed test throughputs left us wondering: if application developers are faced with this scenario, can they conclusively say whether or not the LTE downlink is the bottleneck for their application? In the above experiment, it is possible that the LTE downlink was indeed the bottleneck for ExoPlayer: there might have been a transient rise in congestion and/or drop in signal quality during the video session, or the speed test might have been given special treatment as was reported to occur in T-Mobile's network in 2014 [6]. Conversely, it is also possible that the LTE downlink was not the bottleneck: instead, the video server might have been congested, or middleboxes in the cellular provider's core network might have treated video differently than speed test (as was described by Kakhki et al. [21]).

Knowing whether or not the cellular downlink is the bottleneck is of paramount importance to mobile application developers; it guides them to where they should focus their development efforts. Like we described above, the fact that the application throughput is lower than the speed test throughput does not conclusively reveal if the cellular downlink is the bottleneck.

2.1 Requirement: knowing the status of per-device queues at the base station

Conclusively determining if the cellular downlink is the bottleneck for a user requires knowing if the downlink queue for the user[5] at the cellular base station remains nonempty during an application data transfer (e.g., downloading a video segment). Therefore, application developers need a metric that indicates *how often their users' queues at the base stations are nonempty*. The 3GPP specifications for LTE describe a

---

[3]This does not include any idle time between two consecutive segments.
[4]Cellular radio bearers usually time out after 10 seconds, so this pause ensures that a new radio bearer is used for each application instance.
[5]LTE base stations maintain a separate queue per user.

metric that captures this information called THROUGHPUT TIME [1]. LTE base station vendors typically implement this metric and make it available to providers.

2.1.1 Background: THROUGHPUT TIME.

The THROUGHPUT TIME of an application transfer is the number of milliseconds[6] where the queue for the user at the base station is *nonempty* for the entirety of each millisecond. More precisely, as illustrated in Figure 2, for every burst of data that arrives in a user's queue starting at Ⓐ— the millisecond where the data arrives at the base station—the base station computes the THROUGHPUT TIME of the burst as the time from the start of the millisecond where the burst is first transmitted Ⓑ to (but not including) the millisecond where the queue runs empty Ⓒ. If the downlink is not the bottleneck for the user, its queue will frequently run empty during a transfer; in other words, there will be multiple bursts of data for the same application-layer transfer. In this case, the THROUGHPUT TIME of the entire application transfer is the sum of THROUGHPUT TIMES of the individual bursts.

For the rest of the paper, we refer to the following states of a user's queue at the base station (depicted in Figure 2):

ACTIVE millisecond: Queue is nonempty and traffic is being transmitted to the user.

CONTENTION millisecond: Queue is nonempty but traffic is not transmitted to the user because other users were assigned the resources.

LAST ACTIVE millisecond: Queue is nonempty at the start, but the remaining traffic is transmitted to the user and the queue runs empty.

UNDERFLOW millisecond: Queue is empty.

In this terminology, THROUGHPUT TIME is the count of ACTIVE and CONTENTION milliseconds in a time window while excluding all LAST ACTIVE and UNDERFLOW milliseconds.

2.1.2 THROUGHPUT TIME reveals if LTE is the bottleneck.

If mobile application developers had access to the THROUGHPUT TIMES corresponding to their application transfer time windows, they can conclusively determine if the cellular downlink is the bottleneck. Specifically, they can apply the following rule: if THROUGHPUT TIME closely matches the application transfer time (i.e., if the transfer time window largely consists of only ACTIVE and/or CONTENTION milliseconds), then the downlink is the bottleneck. On the other hand, if THROUGHPUT TIME is smaller than the application-level transfer time (i.e., the transfer time window consists of a significant number of LAST ACTIVE and/or UNDERFLOW milliseconds), then the downlink is not the bottleneck.

Moreover, when the cellular downlink is not the bottleneck, correlating fine-grained burst information (i.e., burst boundaries and lengths) with application-level information,
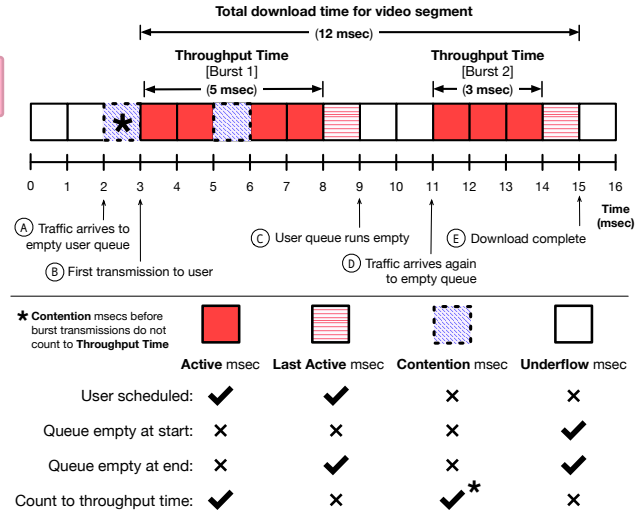
---

[6]LTE schedules transmissions in one millisecond time intervals.

Figure 2: THROUGHPUT TIME is the total number of ACTIVE and CONTENTION milliseconds. It measures the time during which a user's queue at the base station is nonempty.

can sometimes reveal exactly where the bottleneck is. We demonstrate an instance of this in our case study in Section 4.

It is important to note that queues at base stations are per-user, not per-application; therefore, THROUGHPUT TIME is easiest to interpret when users are only running one network-intensive application at a time. We expect this to be the common case for mobile devices. Although, even if there are multiple applications performing simultaneous transfers, THROUGHPUT TIME can still reveal if the downlink is *not* the bottleneck. However, if the downlink *is* the bottleneck, the developer will not be able to determine if the downlink would still be a bottleneck if only one application were running.

2.2 Reality: the status of base station queues is not available to developers

Base station vendors make THROUGHPUT TIME measurements available only to cellular providers in order to help them debug performance of their base stations on the field. Unfortunately, these measurements are not exposed to developers as it may *potentially* reveal the proprietary scheduling algorithms of the base stations. With device-based debugging tools [25, 29], an application developer can access logs from a user's modem to determine which milliseconds the device was scheduled to receive traffic (ACTIVE milliseconds). However, if a user was not scheduled in a millisecond, the developer does not know whether it was because its queue at the base station ran empty (UNDERFLOW millisecond), or because it simply lost its turn to other competing users (CONTENTION

millisecond). As a result, application developers today cannot disambiguate a bottleneck due to downlink congestion (the downlink is the bottleneck) from a bottleneck elsewhere in the network (the downlink is not the bottleneck).

3 ESTIMATING THROUGHPUT TIME

In this section, we describe BurstTracker, the first tool for estimating THROUGHPUT TIME with only client-side data, making it possible for *anyone*, including application developers, to know if a mobile application's performance is bottlenecked at the base station. We start by discussing the challenges that make it difficult to determine if the LTE downlink is a bottleneck, with only local information. Then we describe how cellular traces collected locally at a device have hidden information that enables developers to infer whether or not a device's queue at the base station is empty. We then present the implementation of BurstTracker, and evaluate its accuracy by comparing its THROUGHPUT TIME estimates with ground truth from a test base station.

3.1 Challenge: Classifying milliseconds

Recall that LTE downlink resources are scheduled in 1 millisecond transmission intervals. Also, recall that finding the burst boundaries during an application-level transfer requires knowledge of the device's queue status at the base station; specifically, the bursts correspond to the transmission intervals during which the queue is continuously nonempty. During consecutive scheduling intervals when a user is receiving traffic (ACTIVE milliseconds), the user knows for certain that its queue at the base station is nonempty, and therefore it is receiving a burst. However, for intervals where the user is not scheduled to receive traffic, a user cannot determine when the burst has ended. The problem is that a user does not know if their queue at the base station is empty (UNDERFLOW millisecond), or if it is nonempty but another user is being scheduled instead (CONTENTION millisecond).

Differentiating between UNDERFLOW milliseconds and CONTENTION milliseconds requires knowledge of the status of the user's queue at the base station. Without being able to distinguish between these categories, it is not possible to compute THROUGHPUT TIME, and therefore not possible to determine if the network bottleneck is at the base station.

3.2 Opportunity: Scheduling patterns

The primary contribution of BurstTracker is an algorithm for locally differentiating between a user's CONTENTION and UNDERFLOW milliseconds. This algorithm works by analyzing the information hidden in the patterns of resource assignments in ACTIVE milliseconds. These patterns reveal how a user can differentiate between the locally unobservable CONTENTION and UNDERFLOW milliseconds.

We developed this algorithm by observing the behavior of downlink schedulers running on Tier-1 U.S. provider base stations. Studying the behavior of LTE downlink schedulers is an untapped, but exciting opportunity to learn about the hidden state of a base station. Downlink scheduling is left out of the LTE specification (similar to how rate control is not specified in the WiFi specification [35]), leaving it up to each vendor to create their own scheduling algorithm, each of which may leak information about the state of a base station.

Our hypothesis was that the downlink scheduler behaves differently when it transitions from scheduling a user to not scheduling it—depending on the status of the user's queue at the base station. Restated, we suspected there are patterns in the way the base station's downlink scheduler allocates resources that reveal why the user was not scheduled to receive traffic in milliseconds following an ACTIVE period.

3.2.1 Schedulers assign one user per millisecond.

To test this hypothesis, we setup an LG G3 smartphone to run a persistent application-level transfer of a large file. The smartphone was connected to a macro-base station of a Tier-1 provider located in a populated metropolitan area. Simultaneously, we captured a trace of the LTE downlink control channel with an SDR-based LTE sniffer (similar to LTEye [23]). The sniffer allowed us to observe the downlink schedule across all users being served by a base station. Figure 3 shows an example of the downlink scheduler behavior that we observed. Our persistent transfer user is "User 1", and another user on the base station that was performing a competing transfer is "User 2".

Surprisingly, in nearly every millisecond, the base station allocates all of the radio resources to a single user. At first blush, it appears like the downlink scheduler may be overly simple: one of the unique features of LTE's OFDMA downlink is that multiple users can be scheduled in the same millisecond. Yet, the macro-base station only appears to use this feature for a small number of milliseconds.

3.2.2 The scheduler reveals the queue status.

The scheduler's preference to assign all of the resources to a single user makes it feasible to infer the state of a user's queue at the base station. Recall that a burst starts in the first transmission millisecond during which a user has traffic enqueued at the base station (ACTIVE millisecond). A burst continues for subsequent milliseconds (ACTIVE and CONTENTION milliseconds) until the user's queue runs empty (UNDERFLOW millisecond). This means that the difficulty lies in identifying the end of a burst, namely when the UNDERFLOW millisecond occurs.

A user knows how many resources it was allocated in ACTIVE milliseconds. It is exactly this information—the number of resources scheduled during an ACTIVE millisecond
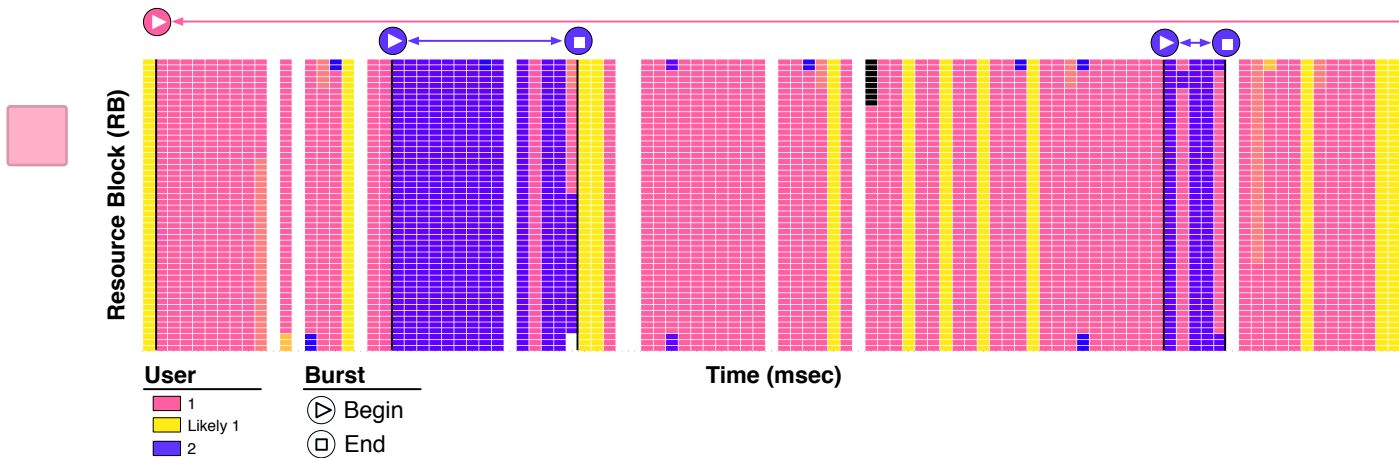
Figure 3: The base station downlink scheduling pattern indicates the status of the user's downlink queue at the base station. In this experiment, user 1 is performing a persistent download of a large file, and user 2 is another user being served by the same base station.

of a burst—that reveals the status of a user's queue at the base station. Recall that the base station prefers to assign all resources in a millisecond to a single user. Therefore, in a given millisecond, if the scheduler allocates the user fewer than the maximum available resources and does not schedule that user any resources in the next millisecond, it is likely because there was not enough traffic enqueued at the base station to completely fill all resources in that last millisecond. So, the millisecond following the partially allocated ACTIVE millisecond must be an UNDERFLOW millisecond, indicating the end of a burst. Therefore, the other locally unobservable milliseconds in the middle of a burst are CONTENTION milliseconds.

In summary, we discovered a way to infer the state of a user's burst by inferring CONTENTION and UNDERFLOW milliseconds using only that user's own resource allocation patterns (and without information from the base station and other users connected to the network). BurstTracker's rules for identifying the beginning and end of a burst from a user's local trace of its resource allocations in ACTIVE milliseconds are as follows:

- **Begin burst:** ACTIVE millisecond with more than 90% of resources allocated to the user.
- **End burst:** Unobservable millisecond following an ACTIVE millisecond with less than 40% of resources allocated to the user.

For example, we marked the burst boundaries for the two competing users in Figure 3. The downlink scheduler starts both of User 2's bursts by allocating 100% of the resources to it, and ends both bursts by allocating less than 40% of the resources. Our persistent transfer does not encounter an
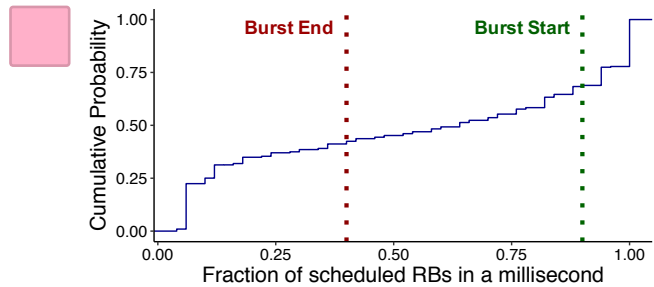

Figure 4: Selecting burst start and end thresholds, based on 60,000 msec of scheduling data. Bursts start in milliseconds where the user is scheduled *most* of the resources; bursts end in milliseconds where a user is scheduled *very few* resources.

UNDERFLOW millisecond, so its burst continues through the entire duration shown in Figure 3[7].

We selected burst start and end thresholds based on the resource allocation patterns of the base stations we tested. Figure 4 shows a CDF of the fraction of resources scheduled to each user in each millisecond. We obtained this data with an LTE sniffer, over a period of 1 minute (60,000 msecs) within a peak usage hour of a Tier-1 provider's macro-base station in a downtown area. The distribution of fractional resource assignment is bimodal: the scheduler prefers to

---

[7] All LTE sniffer traces can have errors. The yellow user is likely User 1 because its ID# is only one bit off of User 1's. Additionally, the idle msecs are unlikely to have actually been idle; they were likely control channel decode failures. Bui et al. observed that the only open-source sniffer, LTEye, failed to identify the correct user for up to 20% of the radio resources traced [7].
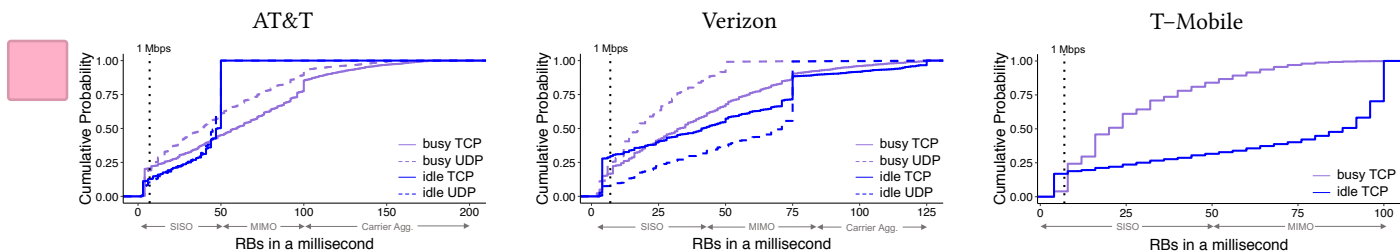
Figure 5: LTE schedulers prefer to schedule slow transfers (1 Mbps) in bursts that consume many radio resources.

assign users either *most* or *very few* of the resources to a particular user in any given millisecond.

Since a burst can be denoted as a pair of start and end milliseconds, a good choice of thresholds should account for roughly similar fractions of ACTIVE milliseconds. In this experiment, starting a burst at a millisecond during which a user was scheduled > 90% of resources accounts for 35% of ACTIVE milliseconds. Ending a burst at a millisecond during which a user was scheduled < 40% of resources accounts for 40% of ACTIVE milliseconds. The middle 25% may correspond to decoding errors from the LTE sniffer, or the users' traffic might have only been enough to create a single-millisecond burst. It may also be possible that in certain uncommon conditions, the scheduler will split resources in a millisecond between users. We later verify against ground-truth data that the accuracy of BurstTracker is not sensitive to the precise values of these thresholds.

3.2.3 Compatibility across providers and workloads.
Next, we set out to evaluate if BurstTracker can be applied to all Tier-1 U.S. providers. Our algorithm is based on the assumption that LTE downlink schedulers prefer to allocate a large number of resources to a small number of users in each millisecond. Therefore, we designed an experiment to test if a provider exhibits this behavior, even across transport protocols and load conditions. Our hypothesis was, if we perform a slow transfer on the downlink, then the scheduler will prefer to allocate a large number of resources over a short duration, rather than a small number of resources over a long duration.

To test this hypothesis, we initiated a series of five one minute slow transfers (1 Mbps) over *iperf* from a server on AWS to a client on a Google Pixel smartphone. During these transfers, we collected the client's downlink resource allocation trace with MobileInsight [25]. We repeated this for all three Tier-1 U.S. providers, in two locations with a range of load conditions. The first location was a busy cell at the "cell phone waiting lot" of a major airport during the peak flight arrival time, and the second location was an idle cell near an empty parking lot in the middle of the night (1 a.m.). We also performed the transfers with both TCP and UDP (except on T-Mobile because their NAT blocks UDP), to see if the transport protocol affects the scheduler's behavior.

Figure 5 shows the results of these experiments as CDFs of the number of resources allocated to the client during the milliseconds that they received traffic (ACTIVE milliseconds). Interestingly, across all providers and protocols, even when the cell is idle, schedulers schedule the slow transfers in large bursts. On AT&T, more than 50% of these bursts consumed 100% of the Single-Input Single-Out (SISO) resources of the base station. On Verizon and T-Mobile, more than 50% of the idle cell bursts used so many resources that they were sent on several Multiple Input Multiple Output (MIMO) channels. For the loaded cells, the distribution of allocations in each millisecond also tended to have more than 50% of scheduled milliseconds consuming half or more of the SISO resources, but also they were utilizing MIMO as well as Carrier Aggregation (bonding of multiple LTE channels). We did not observe a significant difference between the scheduling behavior for TCP and UDP traffic.

In summary, even for the slowest flows, the LTE downlink schedulers of all Tier-1 U.S. providers prefer to assign most resources to a user in each millisecond, confirming our hypothesis. Different providers may require different thresholds, but the thresholds we selected in the previous section are already quite conservative—many resources are required to start a burst, but only less than half are required to end it.

Why do downlink schedulers prefer bursts? Schedulers likely prefer to allocate a small number of users in each millisecond for two reasons. First, bursting saves spectrum resources by reducing control channel overhead. Scheduling an additional user in a timeslot increases the control channel overhead because it requires transmitting additional control information for each additional user [8]. Specifically, the LTE control channel occupies between 7% and 21% of the downlink radio resources in each millisecond [23]. Second, bursting improves a user device's battery life by increasing the time a user's radio can enter a low-power mode. Specifically, transmitting downlink traffic every millisecond in a fraction of the available resources requires users to continuously operate their radios with their receive circuitry powered on [16].

3.3 Implementation

BurstTracker's implementation consists of a set of scripts that estimate THROUGHPUT TIME by analyzing an LTE modem's MAC-layer traces. The code is open source and can be found at https://github.com/arjunvb/bursttracker. In this section, we describe the implementation in detail.

An LTE device's modem has a debug interface that records a trace of resource allocation during ACTIVE milliseconds. BurstTracker uses the following information from each ACTIVE millisecond in these traces: (a) the global index identifying the current millisecond at the base station, (b) the number of radio resources allocated to the user in that millisecond, and (c) the timestamp in the device's clock domain, so base station scheduler information can be aligned in time with the application-level transfer logs.

BurstTracker provides two modes of operation for application developers. In the *offline* mode, an application developer runs their application, and simultaneously collects the LTE modem's resource allocation traces with QXDM [29] or MobileInsight [25]. Then the developer runs BurstTracker with these traces as input to identify if LTE is the bottleneck link, and also to produce fine-grained estimates of the bandwidth offered by the base station to the client device. The offline implementation consists of a set of MATLAB and R scripts.

In the *online* mode, the application developer runs BurstTracker in real-time, while running their application and collecting MAC-layer traces with MobileInsight [25][8]. This can be useful during field tests, where the developer might be interested in testing particular scenarios and environments where their mobile application is underperforming. The online implementation is a Python script that processes the modem traces in real-time using the MobileInsight API.

If the developer uses QXDM to capture physical layer logs, then the mobile device running the application must be connected over USB to a host computer, which runs all of the required software. If the developer uses MobileInsight, then BurstTracker can run entirely the mobile device. However, a current limitation of MobileInsight is that the mobile device must be rooted, but this is unlikely to be a problem for an application developer.

3.4 Evaluation

In this section, we evaluate how accurately BurstTracker is able to locally estimate THROUGHPUT TIME. First, we perform a rigorous evaluation of BurstTracker's accuracy by comparing its estimates of THROUGHPUT TIME to ground truth measurements from a test base station. We also evaluate BurstTracker's estimates against a baseline estimator, which takes as input the same ACTIVE milliseconds traces as

---

[8]The QXDM software interface does not expose real-time logs of MAC-layer messages.

BurstTracker, but does not infer CONTENTION milliseconds when estimating THROUGHPUT TIME.

We find that BurstTracker is able to accurately estimate ground truth THROUGHPUT TIME measurements from a test base station under challenging conditions; this significantly increases our confidence in the soundness of the algorithm.

Experiment setup. We evaluate the accuracy of BurstTracker's THROUGHPUT TIME estimates with two common mobile workloads: (1) large file downloads, and (2) HTTP video streams. Each of these high-throughput workloads will put different aspects of BurstTracker to the test. A *20 MB File download* will produce one large burst that will only end when the download ends. This workload tests if BurstTracker ends bursts too early, especially when there is congestion. A *2 min DASH video stream (10 bitrates from 200 kbps−12 Mbps)* will produce many short bursts that will end when each segment download ends, and the burst sizes will change over time depending on the network congestion and the segment sizes selected by the ABR algorithm. This workload tests how accurately BurstTracker detects the beginning and end of the many short bursts.

We ran the workloads on a LG G3 smartphone, and simultaneously collected THROUGHPUT TIME estimates from Burst-Tracker and ground-truth THROUGHPUT TIME measurements. Both workloads were implemented within a JavaScript client and ran within a Google Chrome browser; this allowed us to collect the same client-side metrics in order to compare the performance of both workloads. BurstTracker's THROUGHPUT TIME estimates were generated by offline scripts with QXDM traces as input (described in Section 3.3).

We conducted 100 runs of each of the workloads on a single test base station, from which we were provided ground-truth THROUGHPUT TIME data. Our experiment setup was stationary in order to ensure that the device remained connected to the same base station. We also interleaved the workloads so they experienced a similar distribution of network conditions.

We evaluated the accuracy of BurstTracker in both typical and adverse network conditions. To test typical network conditions, we ran experiments with mid-range signal strength and sporadic competing users. To create adverse conditions, we reduced the signal strength at the smartphone by placing it inside a metal container. We also introduced competing file downloads from five other smartphones to saturate the base station's downlink. Across all network conditions that we tested, the throughput available to the workloads ranged from 2−12.5 Mbps.

3.4.1 *BurstTracker is accurate.* In Table 1, we present the 25th, 50th (median), and 75th percentiles of the Absolute Percent Error of BurstTracker estimates when compared with the ground-truth. BurstTracker performs consistently

Application	Throughput Time (% of application time)			Absolute Error (%)					
				BurstTracker			Active Time		
	25 pct	50 pct	75 pct	25 pct	50 pct	75 pct	25 pct	50 pct	75 pct
File Download	83.3	94.1	98.4	4.3	7.2	12.1	50.8	60.2	66.9
Video Streaming	62.3	73.4	82.2	3.1	6.9	15.2	54.6	59.9	65.9

Table 1: BurstTracker locally estimates Throughput Time accurately for file download and video streaming—two applications with different kinds of burst patterns (i.e. video streaming is more "bursty" than file download). BurstTracker estimates are evaluated against ground-truth measurements of Throughput Time.

well for both file download and video streaming, achieving a Median Absolute Percent Error (MAPE) of 7%. This indicates that BurstTracker is robust to applications of varying burstiness, quantified by the fraction of application transfer time during which the downlink is the bottleneck. We also found that the baseline estimator, Active Time, performs poorly, with a MAPE of ~60% under the same conditions. The improvement in error over the baseline Active Time lies in BurstTracker's ability to: (i) count time slots where the user had a nonempty queue, but was *not* scheduled for transmission, and (ii) discount time slots where the user was scheduled for transmission, but had a queue that ran empty.

3.5 Is BurstTracker future proof?

Will BurstTracker be needed in future generations of mobile networks? We expect that in New Radio (the 5G successor to 4G LTE) and beyond, the need to detect if the radio link is the bottleneck for an application will remain. Bandwidth-intensive applications such as augmented reality, virtual reality, and autonomous vehicles will continue to drive application developers to investigate cellular bottlenecks.

Will the BurstTracker algorithm continue to work in future generations of cellular networks? The core insight of BurstTracker is that it is possible to infer if the radio link is the bottleneck by observing how the base station allocates resources to its users. We believe that this core insight will likely continue to remain true in future generations. However, we acknowledge that the algorithm that we outlined in Section 3.2 is specific to LTE scheduling.

4 CASE STUDY: VIDEO STREAMING

We demonstrate the power of BurstTracker by using it to investigate and explain the gap between speed test throughput and video streaming performance (following up with the experiment in Section 2). We focus on video streaming because it is one of the most widely deployed and thoroughly understood mobile applications. With BurstTracker, we were able to uncover a new, previously-unidentified bottleneck affecting video streams on U.S. cellular networks.

We start the investigation with the following question: is the bottleneck the LTE downlink, or somewhere upstream?

As we discussed previously, this question is difficult for developers to answer with confidence. Without a metric like Throughput Time, which captures how frequently an application's queue at the base station is nonempty, application developers do not know if the poor performance an application experiences is outside of the developer's control (e.g., congestion), or something within the developer's control (e.g., the application not requesting enough data).

BurstTracker proves instrumental in explaining the gap between speed test and video streaming in three ways. First, BurstTracker conclusively distinguishes between the two sides of the base station to reveal that the source of the problem is not the radio network, but somewhere upstream. This eliminates uncertainty about the behavior of base station schedulers that sometimes deters application developers from investigating further. Second, BurstTracker exposes a pattern of growing burst lengths during video segment downloads, which points to TCP Slow-Start Restart (SSR) as the root cause. And third, BurstTracker verifies that queue underruns are eliminated with connection settings designed to bypass middleboxes, indicating that a transparent middlebox in the cellular network is likely responsible for forcing SSR.

In this section, we demonstrate that BurstTracker's ability to track the status of a user's queue at the base station is the key to determining the root cause of the gap between speedtest and video streaming; namely, that it is not due to the LTE link. Additionally, we show that developers can solve this problem with a simple change to the application, which will lead to significant improvements in the performance of mobile video streaming.

4.1 Experiment setup

To run controlled experiments that exposed all of the relevant client and server-side metrics for this case study, we set up an AWS machine running an HTTP Apache2 server. We ran a nodejs server on our AWS instance to record the various client-side metrics that were being POSTed by the client device. We used the same DASH video and 20 MB file download setup described in Section 3.4. We instrumented the JavaScript-based DASH client to record the start time, end time, and volume downloaded for each video segment

streamed by the player. For the file download, we collected client-side metrics by implementing a JavaScript client—identical to the one streaming the DASH video—to fetch the file via a basic `XMLHTTPRequest`. We instrumented the client to record the timestamps at which the first and last bytes of the file are downloaded, as well as the total volume downloaded. We disabled TCP Slow-Start Restart (SSR) on our server[9], following the HTTP/2 deployment recommendations [17], and because SSR impacts the performance of video streaming [27].

Our physical setup consisted of a stationary LG G3 Android smartphone on AT&T's network and locked to one LTE band. We ran the video and file download benchmark applications in Google Chrome. In addition to obtaining the client-side metrics described above, we also collected radio-layer logs from the phone's modem using QXDM [29], and ran our offline implementation of BurstTracker to compute THROUGHPUT TIME estimates for every application transfer.

4.2 Is LTE the bottleneck?

Using the instrumentation described in Section 4.1, we now investigate the root cause for the poor video streaming performance that we observed in the motivating experiment (Figure 1). We first ask the following question: was the video application bottlenecked by congestion at the base station, or was there some other inefficiency that selectively inhibited the performance of video but not the speed test?

Link demand metric. In order to compare how often the downlink is a bottleneck across different applications and network conditions, we normalize THROUGHPUT TIME to create a new metric that we call *link demand*. Link demand is the ratio of THROUGHPUT TIME to the duration of an application transfer. Intuitively, the link demand indicates whether there is congestion at the base station, or whether there is a bottleneck elsewhere. When the link demand is close to 100%, the application must have kept the queue at the base station nonempty most of the time, indicating that congestion at the base station was the bottleneck. Conversely, when the link demand is low, we can conclude that the queue at the base station frequently ran empty over the duration of the session, suggesting that there is a bottleneck elsewhere.

We compute the link demand for both file download and video streaming applications. BurstTracker estimates THROUGHPUT TIME, and we use client-side timestamps recorded during each application-level transfer to calculate the transfer duration. Note that for video streaming, we only consider the total time spent downloading video segments, and omit any idle periods spent between segment downloads, because these periods of inactivity are intentionally introduced by

---

[9]This can be done by setting `net.ipv4.tcp_slow_start_after_idle` to `0` on a Linux kernel.
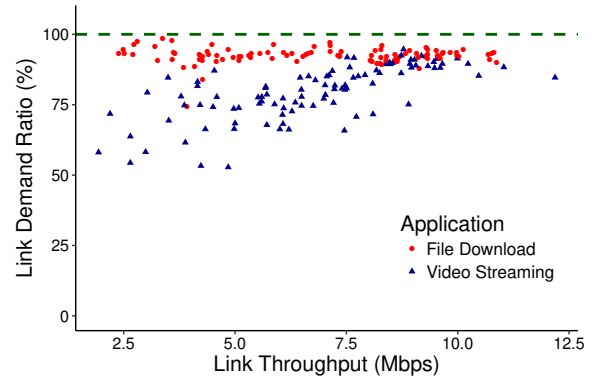
Figure 6: While file downloads consistently keep their queues nonempty, video streaming applications keep their queues nonempty only 50–75% of the time.

the application. By ignoring these inter-segment idle times, the video streaming transfer effectively becomes a stream of continuous smaller file downloads, and hence, it can be be compared with a larger file download.

Link throughput metric. We compute the *link throughput* as the ratio of application volume and THROUGHPUT TIME. This metric tells us how much bandwidth the base station was offering the application, and provides a measure of the severity of the congestion during the application transfer.

4.2.1 Experiment and results. To investigate the disparity we identified between file download and video streaming performance, we conducted 100 experiment runs on a base station of a Tier-1 U.S. provider in a downtown area at different times of day (morning, lunch hours, late night). Each run consisted of the following sequence of events: download a 20 MB file, wait 15 seconds, stream a DASH video. For each run, we use BurstTracker to estimate the link demand for the file download and the video stream. A video stream is a series of file downloads, so it is natural to expect that a video stream would register a similar link demand as a large file download.

Surprisingly, we find that there is a significant gap in utilization between the file download and video transfers. In Figure 6, we plot the link demand for each application transfer against the link throughput observed for that particular run. Over a wide range of throughput conditions (2–12 Mbps), file downloads spend 95% of their time in bursts, while video streaming sees a much larger variance, keeping their queues nonempty 65–70% of the time, on average. Additionally, the disparity becomes more significant as the link congestion intensifies (lower link throughputs), suggesting that video streaming applications are less efficient under more congested settings.

(a) TCP sender throughput

(b) Time series of Resource Block (RB) allocation for a single DASH segment
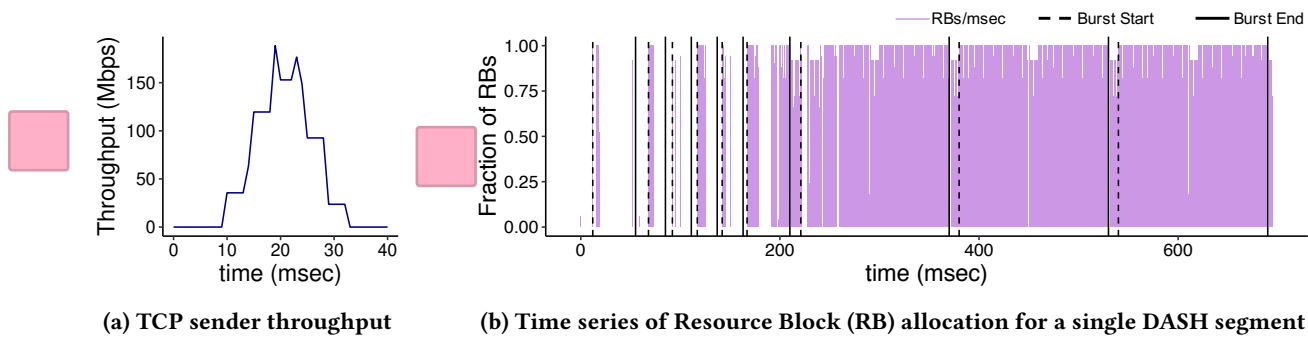
Figure 7: Data releases from the server's TCP sender within 25 ms, implying a throughput of more than 100 Mbps. However, the client receives the data over 700 ms, indicating that the bottleneck is not at the video server.

Our findings indicate that the queue frequently runs empty at the base station for a video stream, while it rarely does for a large file download. Further, the 25–30% difference in link demand approximately matches the difference in application throughputs that we observed in the original motivating scenario. This explains a majority of the performance gap. We can also conclude that the gap is not caused by the LTE downlink, but rather by another inefficiency that prevents the video application from enqueuing at the base station. This points us upstream from the base station to continue our investigation.

4.3 Is the server the bottleneck?

After ruling out the LTE downlink as the source of video streaming underutilization, we suspect that either the application or the TCP sender on the server is the bottleneck. This is because the link between the server and the base station is unlikely to be the bottleneck at the moderate throughput we are observing.

In order to isolate the behavior of the server-side, we conducted controlled experiments on an idle macro-base station. Specifically, we ran the DASH video stream on a macro-base station at 1 A.M. local time, when the base station is likely to be idle (i.e., not serving competing traffic). This factors out congestion on the cellular downlink, which we have already concluded is not responsible for poor video performance. During the experiment, we collected `tcpdump` traces at the server and client to measure the dynamics at the transport layer. Simultaneously, we collected QXDM traces to observe the resource allocation patterns. We processed these traces with BurstTracker to identify the burst boundaries.

Figure 7a plots TCP throughput at the server (from `tcpdump`) for a single video segment downloaded during a DASH video, and Figure 7b shows a time series of LTE resource allocation (and bursts from BurstTracker) for the same segment. Each bar corresponds to a millisecond and the height of the bar corresponds to the fraction of resources scheduled in

that millisecond. Burst start and end boundaries, identified by BurstTracker, are marked with dashed and solid lines, respectively.

A comparison of the two time series charts shows that data gets released from the server very quickly at very high effective throughput, while it takes much longer to be received by the client. This indicates that the application or the server (TCP sender) is not limiting the flow of data. This mismatch between the two time series suggests that the transport between the server and base station is the root cause.

4.4 It is Slow-Start Restart

The mismatch between the timespan over which the server transmits data and the duration over which the client receives data is revealing: if data drains out from the server faster than it drains into the client, then it must end up in a mid-network queue. However, the base station queue runs empty frequently, as we discussed in Section 4.2, so it must be queuing upstream. At the same time, the base station backhaul link is faster than the rate at which the mid-network link is draining, as indicated by the throughput of the file download (and speed test).

To make sense of these observations, we took a closer look at the data. Figure 7b reveals that the duration of bursts increases over the entire video segment transfer. This indicates that the video application's queue tends to run empty at the *beginning* of the segment transfer, and as the download progresses, the application's queue at the base station remains nonempty for much longer.

The increase in the duration of bursts contained within a single video segment transfer is characteristic of TCP Slow-Start. However, such behavior is not expected in our setup (after the first segment), because we disabled SSR on the server. Usually, the sender's TCP window clarifies TCP behavior; however, in this case, the server's TCP window shows no relation with the client-side observations.

Following this lead, we investigate the TCP throughput over time measured by running `tcpdump` on the client for three video consecutive DASH video segments (Figure 8a). We observe that the throughput grows exponentially, from a very low value to the perceived network capacity, for each segment download. The behavior appears to be SSR for every video segment.

4.5 SSR is being added by a TCP middlebox

After ruling out the LTE downlink and server as the bottlenecks, we dig deeper to identify the cause of the SSR effect. We investigate the only remaining possibility—the link between the server and the base station—however unlikely it may seem. We consider the possibility of a middlebox in the cellular network. Prior work [39] suggested that middleboxes are typically deployed as transparent split-TCP proxies that terminate the TCP connection, process or shape the traffic, and accordingly deliver the data to the client. The presence of such a middlebox would explain both the mismatch between server and client TCP traces, and the presence of SSR despite it being disabled on the server. With a split-TCP proxy, the effective sender to the client is the middlebox, which may have SSR enabled.

Transparent middleboxes are difficult or impossible to detect by directly probing. However, they can be detected indirectly. Moreover, they can be bypassed by using non-standard ports, as they typically only act on HTTP and HTTPS traffic [39], which is delivered over TCP ports 80 and 443. If changing the port eliminates the SSR behavior, it is a strong indicator of the middlebox adding SSR, and therefore the middlebox is the bottleneck.

To confirm that a cellular middlebox is responsible for forcing SSR on video segment downloads, we repeat the DASH video segment downloads conducted on an idle cell, but this time using port 7777, which is unreserved. In Figure 8b, we plot the TCP throughput traces from `tcpdump` on the client. Indeed, we do not see the growth in throughput characteristic of SSR. Each chunk download starts at throughput close to the stable capacity, which is proportional to the cwnd it learned from the last packet sent over the connection. Importantly, the cwnd does not appear to reset to its initial value in response to the idle time between chunk downloads. Therefore, it is evident that the video segments downloaded on port 7777 were not subject to SSR.

In summary, using BurstTracker, we have shown that a cellular middlebox is indeed responsible for forcing SSR on the application transfer. We reached this conclusion by first using the link demand computed with BurstTracker to rule out the cellular downlink as the bottleneck for mobile video streaming; then, we used BurstTracker along with other data sources (e.g., `tcpdump` traces at the server and client) to
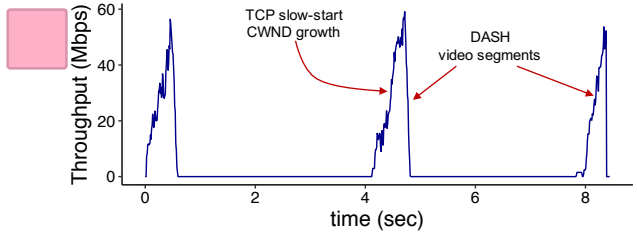
(a) TCP throughput for 3 video segments over port 80
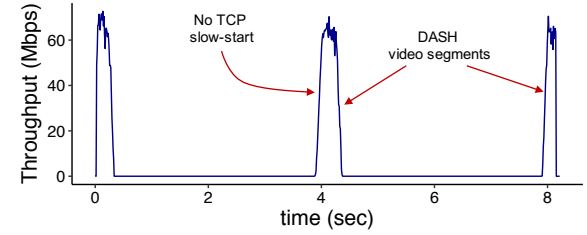
(b) TCP throughput for 3 video segments over port 7777

Figure 8: A transparent TCP proxy forces SSR on port 80, but not over an unreserved port (e.g. 7777).

pinpoint a split-TCP proxy adding SSR, as the root cause of the performance gap (identified in Figure 6). SSR is especially detrimental to video streaming on congested links, because it forces ABR algorithms to select lower bitrates (smaller video segment sizes). Consequently, the smaller segment downloads spend a significant portion of time in the Slow-Start phase, where their throughputs are artificially limited.

5 SSR MIDDLEBOXES

In Section 4, we discovered that a Tier-1 carrier had an SSR-forcing middlebox in its network. In this section, we first explore how prevalent SSR middleboxes are on all U.S. Tier-1 cellular carriers. Then, we characterize the impact of SSR on mobile video streaming.

5.1 Middleboxes are widely deployed

We compared the performance of DASH video streaming on three Tier-1 U.S. carriers—AT&T, T-Mobile, and Verizon—served over HTTP (port 80), HTTPS (port 443), and an unreserved port (port 7777). We used the same DASH video setup described in Section 4.1, and repeat 25 times for each combination of port and carrier, amounting to a total of 225 data points. To ensure that the video streams being compared were subject to similar network conditions, we downloaded the video over ports 80, 443, and 7777 in succession. We conducted these experiments during peak hours (lunch hours) in a densely-populated metropolitan area, on a congested base station belonging to each of the carriers. We focused

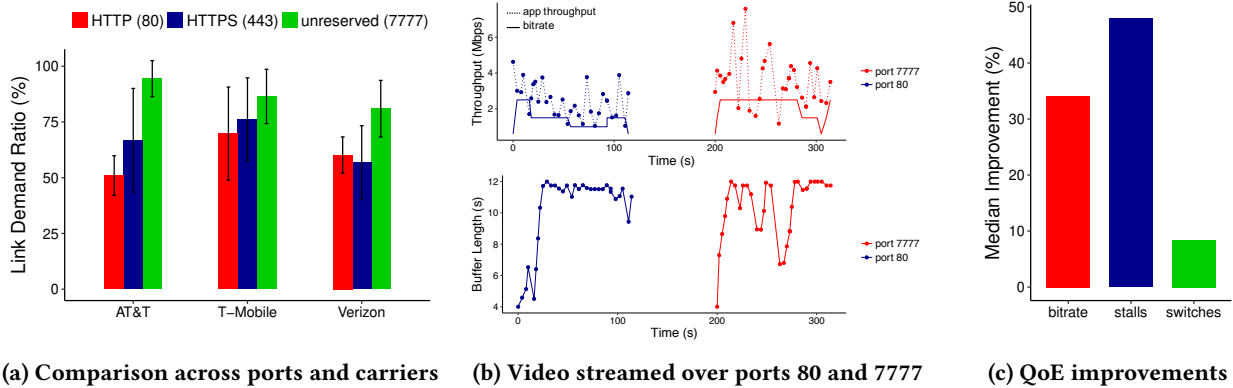(a) Comparison across ports and carriers (b) Video streamed over ports 80 and 7777 (c) QoE improvements

Figure 9: SSR at the middlebox impacts video streamed over HTTP (port 80) and HTTPS (port 443) on AT&T and Verizon. Bypassing the middlebox on all three providers (port 7777) yields 40% more link utilization. On AT&T's network, we observed that streaming video without SSR at the middlebox improves the video bitrate by 35% on average.

on congested links because our results in Figure 6 indicated that the video streaming link demand was especially low for highly congested links (with sub-8 Mbps throughput). During these experiments, we simultaneously ran BurstTracker to observe the link demand ratio.

Our results are shown in Figure 9a. Each bar shows the average link demand for a particular combination of port and cellular carrier. For all three Tier-1 U.S. carriers, videos served over HTTP and HTTPS achieve lower link demand ratios (50–70% average utilization) compared to their counterparts streamed over port 7777 (80+% average utilization). This experiment reveals that split-TCP middleboxes are likely deployed on all major U.S. carriers. We manually inspected tcpdump traces taken during these experiments to determine what the middleboxes are doing to throttle the traffic. We observed that the AT&T and Verizon middleboxes appear to be forcing SSR (we only saw this on AT&T in Section 4). The T-Mobile middlebox appeared to be throttling video streams the standard ports, but it did not look like SSR.

5.2 SSR is detrimental to video streaming

Next, we explore the impact of SSR on the bitrates selected by mobile video streaming algorithms. We show how bypassing cellular middleboxes not only improves link demand, but also allows ABR algorithms to stream at higher bitrates for the same link conditions. We repeated a similar set of experiments described in Section 5.1. In addition to the link demand and application throughput metrics, we also collected video streaming performance metrics, namely bitrate and stalls. We conducted 20 runs of a DASH video on port 80 followed by a DASH video streamed on port 7777, all on a congested AT&T base station during peak hours. We show

the client-side metrics from a single run of this experiment in Figure 9b. Figure 9c shows a summary over all 20 runs.

On the top pane of Figure 9b, we show the bitrate the ABR algorithm selected for each segment (solid line) along with the application throughput measurements (dotted line). On the lower pane, we plot the length of the video buffer on the client device. The video streamed over port 7777 is able to sustain a higher bitrate on average, because its ABR algorithm observes high application throughput from the previously downloaded video segments. In contrast, the video streamed on port 80 measures a lower throughput in each consecutive segment, due to a feedback loop from SSR. As the ABR lowers the bitrate, the video segments are smaller in size, and therefore downloads spend an increasing proportion of their time in Slow-Start, reducing their throughputs. This effect is especially pronounced at the low throughputs characteristic of a congested cellular link. The flow on port 7777 also makes better use of its buffer. The buffer rarely fills to capacity (lower pane of Figure 9b), allowing the video to sustain the bitrate of 2.5 Mbps for much longer than the port 80 video stream which always keeps its buffer full.

In Figure 9c, we quantify the overall improvements in Quality-of-Experience (QoE) that we observe after eliminating the effects of SSR introduced by the middlebox. We plot the average percent improvement in QoE across all 20 runs. We observe a median of 35% improvement in video quality, 50% shorter stall durations, and 10% fewer switches in bitrate. Notably, the improvement in bitrate aligns closely with the ~35–40% capacity gap observed for AT&T (Figure 9a).

5.3 A simple way to disable SSR

These experiments demonstrate that middleboxes can be by-passed by moving video streams to operate over an uncommon TCP port (e.g., 7777). However, this is not a deployable solution, because it would require significant modifications to both clients and server infrastructure [14].

We devised a simple way to that requires a minor modification to existing video streaming client software. Our solution comes from the observation that Slow-Start is restarted when a TCP connection goes idle for a duration exceeding a timeout period. Therefore to avoid restarting Slow-Start, we modified the video client to perform frequent micro-fetches (i.e., download 1-byte files) whenever the application is idle between downloading video segments. We verified that this "chatty" video streaming application effectively disables middlebox-forced SSR.

6 RELATED WORK

LTE capacity estimation

Systems such as LoadSense [9], CLAW [37], and piStream [36] have demonstrated that it is possible for a user to passively and locally determine the total downlink resources allocated to all users on an LTE base station. These systems provide limited bottleneck detection capability during an application transfer. Specifically, if they detect that the base station has unallocated resources, they will accurately conclude that LTE is not the bottleneck. However, if all of the resources are allocated, for instance when there are competing users, these systems will not reveal if LTE is the bottleneck. This is because they can not differentiate between the following two reasons why a user is not receiving traffic in a particular millisecond: (a) the user's queue at the base station is empty so there is no traffic to send (LTE downlink is not the bottleneck), and (b) the user's queue is non-empty, but other users are being scheduled ahead of the user (LTE downlink is the bottleneck). BurstTracker can infer the state of the user's queue at the base station, so it works even in the scenario where multiple users are receiving traffic.

Detecting if the LTE downlink is the bottleneck has been considered so difficult that several proposals suggested adding explicit indications of where the bottleneck is into network traffic. For example, engineers from Google, Nokia, and Vasona Networks proposed adding a TCP option where LTE base stations can specify the bandwidth available for a user [19]. Also, Xu et al. [38] suggested injecting timing and sequence information about network socket calls into packets in order to improve estimates of cellular network performance. BurstTracker does not require any modifications to network traffic in order to detect if LTE is the bottleneck.

Network bottleneck estimation

BurstTracker detects whether or not a *specific* link—the cellular downlink—is the bottleneck link. This is different than estimating the bandwidth of the bottleneck link, which is one of the primary goals of congestion control algorithms [5, 18, 22]. However, it is similar to the goal of per-link Internet measurement tools such as pathchar [11]. These tools can observe the bandwidth and latency at each link between a source and destination. However, they are active measurement tools, so they require transmitting significant additional traffic. Therefore, they can not be used to detect if an application's traffic is bottlenecked by the LTE downlink.

QProbe [3] is an active probing tool that can determine if there is congestion at the LTE downlink, or elsewhere in the network. Similar to BurstTracker, QProbe is driven by the insight that a base station's scheduler produces bursty transfers. Unlike BurstTracker, QProbe requires active probing (with TTL-limited load packets). Therefore, users can run QProbe before and after an application transfer, but not during one. QProbe has the same limitation as speed tests: it can not determine if an application has poor network performance because LTE is the bottleneck. Instead, QProbe can determine how often there is more congestion on the LTE downlink, than there is elsewhere in the network.

Sundaresan et al. [31] proposed techniques to detect bottlenecks in home networks, but their approach relies on steady inter-arrival times that do not appear in OFDMA-based cellular networks.

Transport protocols can be optimized for performance on LTE networks [15]. BurstTracker can be helpful during the testing and development of these protocols to ensure that they are making full use of the LTE downlink.

Application-layer measurements

Prior work has used application-layer throughput measurements to estimate future network performance [20, 30, 34]. BurstTracker can complement these approaches by providing real-time, accurate measurements of available cellular downlink capacity.

Cellular middleboxes

BurstTracker complements existing tools to characterize cellular middleboxes. NetPiculet [33] is a network monitoring tool that exposes the NAT and firewall policies affecting mobile applications on cellular networks. Through a series of measurements, this work explores how these policies affect the performance, energy consumption, and security of mobile applications. Xu et al. [39] and Li et al. [24] present automated techniques for discovering policies enforced by transparent TCP proxies. By adding BurstTracker's bottleneck detection to these tools, we believe that these tools

can discover more middlebox policies that affect application performance, particularly in congested networks. For example, we confirmed that the LTE downlink was not the bottleneck for video streams on congested links. This lead us to investigate middlebox-forced TCP behavior, and discover that middleboxes are forcing SSR (Section 4).

7 CONCLUSION

The key insight behind BurstTracker is that a base station's hidden state (e.g., queue length) can be estimated based on patterns in the output of their proprietary downlink scheduling algorithms. We believe that the behavior of these schedulers may be a new opportunity to reveal other hidden information that may be useful to application developers as well as researchers.

Even though we designed BurstTracker with the objective of simply determining if the cellular downlink is the bottleneck for applications, we were able to use BurstTracker to discover a significant issue that affects video streaming in most Tier-1 U.S. cellular networks and also identify the cause of the issues (namely, middlebox-forced Slow-Start Restart). Based on BurstTracker's findings, we also believe that there is an easy fix for this issue—transferring small packets periodically between larger application transfers. We believe that a tool like BurstTracker opens up possibilities into discovering other similar inefficiencies that may be plaguing user experience on cellular networks, and we look forward to the community (and us) trying using BurstTracker to investigate the performance of more high-throughput applications like interactive video and virtual/augmented reality streaming.

ACKNOWLEDGMENTS

REFERENCES

[1] 3rd Generation Partnership Project (3GPP). Telecommunication management; Key Performance Indicators (KPI) for Evolved Universal Terrestrial Radio Access Network (E-UTRAN): Definitions. https://www.3gpp.org/DynaReport/32450.htm.

[2] M. Allman and V. Paxson. TCP congestion control. RFC 5681, IETF, September 2009.

[3] N. Baranasuriya, V. Navda, V. N. Padmanabhan, and S. Gilbert. QProbe: locating the bottleneck in cellular communication. In *Proc. ACM Conference on emerging Networking EXperiments and Technologies (CoNEXT)*. ACM, 2015.

[4] A. Biernacki and K. Tutschku. Comparative performance study of LTE downlink schedulers. *Wireless Personal Communications*, 74, 2014.

[5] J.-C. Bolot. Characterizing end-to-end packet delay and loss in the Internet. *Journal of High Speed Networks*, 2(3):305–323, July 1993.

[6] J. Brokdin. T-Mobile forced to stop hiding slow speeds from throttled customers. https://arstechnica.com/information-technology/2014/11/t-mobile-forced-to-stop-hiding-slow-speeds-from-throttled-customers/.

[7] N. Bui and J. Widmer. OWL: a reliable online watcher for LTE control channel measurements. In *Proc. Workshop on All Things Cellular: Operations, Applications and Challenges*, 2016.

[8] F. Capozzi, G. Piro, L. A. Grieco, G. Boggia, and P. Camarda. Downlink packet scheduling in LTE cellular networks: Key design issues and a survey. *IEEE Communications Surveys Tutorials*, 15(2):678–700, 2013.

[9] A. Chakraborty, V. Navda, V. N. Padmanabhan, and R. Ramjee. Coordinating cellular background transfers using LoadSense. In *Proc. ACM Conference on Mobile Computing and Networking (MobiCom)*, 2013.

[10] M. C. Chan and R. Ramjee. TCP/IP performance over 3G wireless links with rate and delay variation. In *Proc. ACM Conference on Mobile Computing and Networking (MobiCom)*, 2002.

[11] A. B. Downey. Using pathchar to estimate internet link characteristics. In *Proc. ACM SIGCOMM*, 1999.

[12] A. Ghosh, R. Ratasuk, B. Mondal, N. Mangalvedhe, and T. Thomas. LTE-Advanced: Next-generation wireless broadband technology [invited paper]. *IEEE Wireless Communications*, 17(3):10–22, June 2010.

[13] Google. Google ExoPlayer. https://github.com/google/ExoPlayer.

[14] Google. HTTPS encryption on the web. https://transparencyreport.google.com/https/overview?hl=en.

[15] P. Goyal, M. Alizadeh, and H. Balakrishnan. Rethinking congestion control for cellular networks. In *Proc. Workshop on Hot Topics in Networks (HotNets)*, 2017.

[16] J. Huang, F. Qian, A. Gerber, Z. M. Mao, S. Sen, and O. Spatscheck. A close examination of performance and power characteristics of 4G LTE networks. In *Proc. ACM Conference on Mobile Systems, Applications, and Services (MobiSys)*, 2012.

[17] IETF HTTP Working Group. HTTP/2 specifications. https://github.com/http2/http2-spec/wiki/Ops.

[18] V. Jacobson. Congestion avoidance and control. In *Proc. ACM SIGCOMM*, 1988.

[19] A. Jain, A. Terzis, H. Flinck, N. Sprecher, S. Arunachalam, K. Smith, V. Devarapalli, and R. B. Yanai. Mobile throughput guidance inband signaling protocol. Internet-draft, IETF, 2017.

[20] J. Jiang, V. Sekar, and H. Zhang. Improving fairness, efficiency, and stability in HTTP-based adaptive video streaming with FESTIVE. In *Proc. ACM Conference on emerging Networking EXperiments and Technologies (CoNEXT)*, 2012.

[21] A. M. Kakhki, A. Razaghpanah, A. Li, H. Koo, R. Golani, D. Choffnes, P. Gill, and A. Mislove. Identifying traffic differentiation in mobile networks. In *Proc. ACM Internet Measurement Conference (IMC)*, 2015.

[22] S. Keshav. A control-theoretic approach to flow control. In *Proc. ACM SIGCOMM*, 1991.

[23] S. Kumar, E. Hamed, D. Katabi, and L. E. Li. LTE Radio analytics made easy and accessible. In *Proc. ACM SIGCOMM*, 2014.

[24] F. Li, A. Razaghpanah, A. M. Kakhki, A. A. Niaki, D. Choffnes, P. Gill, and A. Mislove. lib•erate,(n): A library for exposing (traffic-classification) rules and avoiding them efficiently. In *Proc. ACM Internet Measurement Conference (IMC)*, 2017.

[25] Y. Li, C. Peng, Z. Yuan, J. Li, H. Deng, and T. Wang. MobileInsight: Extracting and analyzing cellular network information on smartphones. In *Proc. ACM Conference on Mobile Computing and Networking (MobiCom)*, 2016.

[26] 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 *Proc. International Workshop on Mobile Computing Systems and Applications (HotMobile)*, 2015.

[27] H. Mao, R. Netravali, and M. Alizadeh. Neural adaptive video streaming with Pensieve. In *Proc. ACM SIGCOMM*, 2017.

[28] Ookla. Speedtest by Ookla. http://www.speedtest.net/.

[29] Qualcomm. eXtensible diagnostic monitor. https://tinyurl.com/yc4e9dcy.

[30] Y. Sun, X. Yin, J. Jiang, V. Sekar, F. Lin, N. Wang, T. Liu, and B. Sinopoli. CS2P: Improving video bitrate selection and adaptation with data-driven throughput prediction. In *Proc. ACM SIGCOMM*, 2016.

[31] S. Sundaresan, N. Feamster, and R. Teixeira. Locating throughput bottlenecks in home networks. In *Proc. ACM SIGCOMM*, 2014.

[32] N. Vallina-Rodriguez, A. Auçinas, M. Almeida, Y. Grunenberger, K. Papagiannaki, and J. Crowcroft. RILAnalyzer: A comprehensive 3G monitor on your phone. In *Proc. ACM Internet Measurement Conference (IMC)*, 2013.

[33] Z. Wang, Z. Qian, Q. Xu, Z. Mao, and M. Zhang. An untold story of middleboxes in cellular networks. In *Proc. ACM SIGCOMM*, 2011.

[34] K. Winstein, A. Sivaraman, and H. Balakrishnan. Stochastic forecasts achieve high throughput and low delay over cellular networks. In *Proc. Symposium on Networked Systems Design and Implementation (NSDI)*, 2013.

[35] S. H. Wong, H. Yang, S. Lu, and V. Bharghavan. Robust rate adaptation for 802.11 wireless networks. In *Proc. ACM Conference on Mobile Computing and Networking (MobiCom)*, 2006.

[36] X. Xie, X. Zhang, S. Kumar, and L. E. Li. piStream: Physical layer informed adaptive video streaming over LTE. In *Proc. ACM Conference on Mobile Computing and Networking (MobiCom)*, 2015.

[37] X. Xie, X. Zhang, and S. Zhu. Accelerating mobile web loading using cellular link information. In *Proc. ACM Conference on Mobile Systems, Applications, and Services (MobiSys)*, 2017.

[38] Q. Xu, S. Mehrotra, Z. M. Mao, and J. Li. PROTEUS: Network performance forecast for real-time, interactive mobile applications. In *Proc. ACM Conference on Mobile Systems, Applications, and Services (MobiSys)*, 2013.

[39] X. Xu, Y. Jiang, T. Flach, E. Katz-Bassett, D. Choffnes, and R. Govindan. Investigating transparent web proxies in cellular networks. In *Proc. Passive and Active Measurement Conference (PAM)*, 2015.