Stochastic Forecasts Achieve High Throughput and Low Delay over Cellular Networks

Keith Winstein, Anirudh Sivaraman, and Hari Balakrishnan
MIT Computer Science and Artificial Intelligence Laboratory, Cambridge, Mass.
{keithw,anirudh,hari}@mit.edu

Abstract

Sprout is an end-to-end transport protocol for interactive applications that desire high throughput and low delay. Sprout works well over cellular wireless networks, where link speeds change dramatically with time, and current protocols build up multi-second queues in network gateways. Sprout does not use TCP-style reactive congestion control; instead the receiver observes the packet arrival times to infer the uncertain dynamics of the network path. This inference is used to forecast how many bytes may be sent by the sender, while bounding the risk that packets will be delayed inside the network for too long.

In evaluations on traces from four commercial LTE and 3G networks, Sprout, compared with Skype, reduced self-inflicted end-to-end delay by a factor of 7.9 and achieved 2.2× the transmitted bit rate on average. Compared with Google's Hangout, Sprout reduced delay by a factor of 7.2 while achieving 4.4× the bit rate, and compared with Apple's Facetime, Sprout reduced delay by a factor of 8.7 with 1.9× the bit rate.

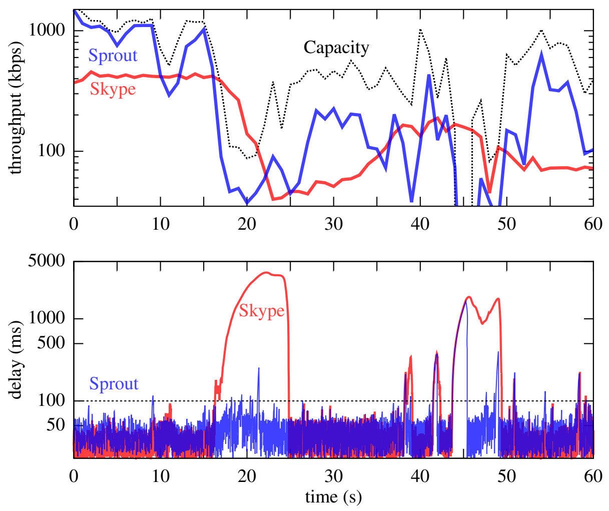
Although it is end-to-end, Sprout matched or outperformed TCP Cubic running over the CoDel active queue management algorithm, which requires changes to cellular carrier equipment to deploy. We also tested Sprout as a tunnel to carry competing interactive and bulk traffic (Skype and TCP Cubic), and found that Sprout was able to isolate client application flows from one another.

1 Introduction

Cellular wireless networks have become a dominant mode of Internet access. These mobile networks, which include LTE and 3G (UMTS and 1xEV-DO) services, present new challenges for network applications, because they behave differently from wireless LANs and from the Internet's traditional wired infrastructure.

Cellular wireless networks experience rapidly varying link rates and occasional multi-second outages in one or both directions, especially when the user is mobile. As a result, the time it takes to deliver a network-layer packet may vary significantly, and may include the effects of link-layer retransmissions. Moreover, these networks schedule transmissions after taking channel quality into account, and prefer to have packets waiting to be sent whenever a link is scheduled. They often achieve that goal by maintaining deep packet queues. The effect at the transport layer is that a stream of packets experiences widely varying packet delivery rates, as well as variable, sometimes multi-second, packet delays.

Figure 1: Skype and Sprout on the Verizon LTE downlink trace. For Skype, overshoots in throughput lead to large standing queues. Sprout tries to keep each packet's delay less than 100 ms with 95% probability.



For an interactive application such as a videoconferencing program that requires both high throughput and low delay, these conditions are challenging. If the application sends at too low a rate, it will waste the opportunity for higher-quality service when the link is doing well. But when the application sends too aggressively, it accumulates a queue of packets inside the network waiting to be transmitted across the cellular link, delaying subsequent packets. Such a queue can take several seconds to drain, destroying interactivity (see Figure 1).

Our experiments with Microsoft's Skype, Google's Hangout, and Apple's Facetime running over traces from commercial 3G and LTE networks show the shortcomings of the transport protocols in use and the lack of adaptation required for a good user experience. The transport protocols deal with rate variations in a reactive manner: they attempt to send at a particular rate, and if all goes well, they increase the rate and try again. They are slow to decrease their transmission rate when the link has deteriorated, and as a result they often create a large backlog of queued packets in the network. When that happens, only after several seconds and a user-visible outage do they switch to a lower rate.

This paper presents *Sprout*, a transport protocol designed for interactive applications on variable-quality networks. Sprout uses the receiver's observed packet arrival times as the primary signal to determine how the network path is doing, rather than the packet loss, round-trip time, or one-way delay. Moreover, instead of the traditional reactive approach where the sender's window or rate increases or decreases in response to a congestion signal, the Sprout receiver makes a short-term forecast (at times in the near future) of the bottleneck link rate using probabilistic inference. From this model, the receiver predicts how many bytes are likely to cross the link within several intervals in the near future with at least 95% probability. The sender uses this forecast to transmit its data, bounding the risk that the queuing delay will exceed some threshold, and maximizing the achieved throughput within that constraint.

We conducted a trace-driven experimental evaluation (details in §5) using data collected from four different commercial cellular networks (Verizon's LTE and 3G 1xEV-DO, AT&T's LTE, and T-Mobile's 3G UMTS). We compared Sprout with Skype, Hangout, Facetime, and several TCP congestion-control algorithms, running in both directions (uplink and downlink).

The following table summarizes the average relative throughput improvement and reduction in self-inflicted queueing delay[1] for Sprout compared with the various other schemes, averaged over all four cellular networks in both directions. Metrics where Sprout did not outperform the existing algorithm are highlighted in red:

App/protocol	Avg. speedup vs. Sprout	Delay reduction (from avg. delay)
Sprout	1.0×	1.0× (0.32 s)
Skype	2.2×	7.9× (2.52 s)
Hangout	4.4×	7.2× (2.28 s)
Facetime	1.9×	8.7× (2.75 s)
Compound	1.3×	4.8× (1.53 s)
TCP Vegas	1.1×	2.1× (0.67 s)
LEDBAT	1.0×	2.8× (0.89 s)
Cubic	0.91×	79× (25 s)
Cubic-CoDel	0.70×	1.6× (0.50 s)

Cubic-CoDel indicates TCP Cubic running over the CoDel queue-management algorithm [17], which would be implemented in the carrier's cellular equipment to be deployed on a downlink, and in the baseband modem or radio-interface driver of a cellular phone for an uplink.

We also evaluated a simplified version of Sprout, called Sprout-EWMA, that tracks the network bitrate with a simple exponentially-weighted moving average, rather than making a cautious forecast of future packet deliveries with 95% probability.

Sprout and Sprout-EWMA represents different tradeoffs in their preference for throughput versus delay. As expected, Sprout-EWMA achieved greater throughput, but also greater delay, than Sprout. It outperformed TCP Cubic on both throughput and delay. Despite being end-to-end, Sprout-EWMA outperformed Cubic-over-CoDel on throughput and approached it on delay:

Protocol	Avg. speedup vs. Sprout-EWMA	Delay reduction (from avg. delay)
Sprout-EWMA	1.0×	1.0× (0.53 s)
Sprout	2.0×	0.60× (0.32 s)
Cubic	1.8×	48× (25 s)
Cubic-CoDel	1.3 ×	0.95× (0.50 s)

We also tested Sprout as a tunnel carrying competing traffic over a cellular network, with queue management performed at the tunnel endpoints based on the receiver's stochastic forecast about future link speeds. We found that Sprout could isolate interactive and bulk flows from one another, dramatically improving the performance of Skype when run at the same time as a TCP Cubic flow.

The source code for Sprout, our wireless network trace capture utility, and our trace-based network emulator is available at `http://alfalfa.mit.edu/`.

2 Context and Challenges

This section highlights the networking challenges in designing an adaptive transport protocol on cellular wireless networks. We discuss the queueing and scheduling mechanisms used in existing networks, present measurements of throughput and delay to illustrate the problems, and list the challenges.

2.1 Cellular Networks

At the link layer of a cellular wireless network, each device (user) experiences a different time-varying bit rate because of variations in the wireless channel; these variations are often exacerbated because of mobility. Bit rates are also usually different in the two directions of a link. One direction may experience an outage for a few seconds even when the other is functioning well. Variable link-layer bit rates cause the data rates at the transport layer to vary. In addition, as in other data networks, cross traffic caused by the arrival and departure of other users and their demands adds to the rate variability.

Most (in fact, all, to our knowledge) deployed cellular wireless networks enqueue each user's traffic in a separate queue. The base station schedules data transmissions taking both per-user (proportional) fairness and channel quality into consideration [3]. Typically, each user's device is scheduled for a fixed time slice over

[1]This metric expresses a lower bound on the amount of time necessary between a sender's input and receiver's output, so that the receiver can reconstruct more than 95% of the input signal. We define the metric more precisely in §5.

which a variable number of payload bits may be sent, depending on the channel conditions, and users are scheduled in roughly round-robin fashion. The isolation between users' queues means that the dominant factor in the end-to-end delay experienced by a user's packets is *self-interaction*, rather than cross traffic. If a user were to combine a high-throughput transfer and a delay-sensitive transfer, the commingling of these packets in the same queue would cause them to experience the same delay distributions. The impact of other users on delay is muted. However, competing demand can affect the throughput that a user receives.

Many cellular networks employ a non-trivial amount of packet buffering. For TCP congestion control with a small degree of statistical multiplexing, a good rule-of-thumb is that the buffering should not exceed the bandwidth-delay product of the connection. For cellular networks where the "bandwidth" may vary by two orders of magnitude within seconds, this guideline is not particularly useful. A "bufferbloated" [9] base station at one link rate may, within a short amount of time, be under-provisioned when the link rate suddenly increases, leading to extremely high IP-layer packet loss rates (this problem is observed in one provider [16]).

The high delays in cellular wireless networks cannot simply be blamed on bufferbloat, because there is no single buffer size that will always work. It is also not simply a question of using an appropriate Active Queue Management (AQM) scheme, because the difficulties in picking appropriate parameters are well-documented and become harder when the available rates change quickly, and such a scheme must be appropriate when applied to all applications, even if they desire bulk throughput. In §5, we evaluate CoDel [17], a recent AQM technique, together with a modern TCP variant (Cubic, which is the default in Linux), finding that on more than half of our tested network paths, CoDel slows down a bulk TCP transfer that has the link to itself.

By making changes—when possible—at endpoints instead of inside the network, diverse applications may have more freedom to choose their desired compromise between throughput and delay, compared with an AQM scheme that is applied uniformly to all flows.

Sprout is not a traditional congestion-control scheme, in that its focus is directed at adapting to varying link conditions, not to varying cross traffic that contends for the same bottleneck queue. Its improvements over existing schemes are found when queueing delays are imposed by one user's traffic. This is typically the case when the application is running on a mobile device, because cellular network operators generally maintain a separate queue for each customer, and the wireless link is typically the bottleneck. An important limitation of this approach is that in cases where these conditions don't hold, Sprout's traffic will experience the same delays as other flows.

2.2 Measurement Example

In our measurements, we recorded large swings in available throughput on mobile cellular links. Existing interactive transports do not handle these well. Figure 1 shows an illustrative section of our trace from the Verizon LTE downlink, whose capacity varied up and down by almost an order of magnitude within one second. From 15 to 25 seconds into the plot, and from 43 to 49 seconds, Skype overshoots the available link capacity, causing large standing queues that persist for several seconds, and leading to glitches or reduced interactivity for the users. By contrast, Sprout works to maximize the available throughput, while limiting the risk that any packet will wait in queue for more than 100 ms (dotted line). It also makes mistakes (e.g., it overshoots at $t = 43$ seconds), but then repairs them.

Network behavior like the above has motivated our development of Sprout and our efforts to deal explicitly with the uncertainty of future link speed variations.

2.3 Challenges

A good transport protocol for cellular wireless networks must overcome the following challenges:

1. It must cope with dramatic temporal variations in link rates.
2. It must avoid over-buffering and incurring high delays, but at the same time, if the rate were to increase, avoid under-utilization.
3. It must be able to handle outages without over-buffering, cope with asymmetric outages, and recover gracefully afterwards.

Our experimental results show that previous work (see §6) does not address these challenges satisfactorily. These methods are reactive, using packet losses, round-trip delays, and in some cases, one-way delays as the "signal" of how well the network is doing. In contrast, Sprout uses a different signal, the observed arrival times of packets at the receiver, over which it runs an inference procedure to make forecasts of future rates. We find that this method produces a good balance between throughput and delay under a wide range of conditions.

3 The Sprout Algorithm

Motivated by the varying capacity of cellular networks (as captured in Figure 1), we designed Sprout to compromise between two desires: achieving the highest possible throughput, while preventing packets from waiting too long in a network queue.

From the transport layer's perspective, a cellular network behaves differently from the Internet's traditional infrastructure in several ways. One is that endpoints can no longer rely on packet drops to learn of unacceptable congestion along a network path ([9]), even after delays reach ten seconds or more. We designed Sprout not to depend on packet drops for information about the available throughput and the fullness of in-network queues.

Another distinguishing feature of cellular links is that users are rarely subject to standing queues accumulated by other users, because a cellular carrier generally provisions a separate uplink and downlink queue for each device in a cell. In a network where two independent users share access to a queue feeding into a bottleneck link, one user can inflict delays on another. No end-to-end protocol can provide low-delay service when a network queue is already full of somebody else's packets. But when queueing delays are largely self-inflicted, an end-to-end approach may be possible.[2]

In our measurements, we found that estimating the capacity (by which we mean the maximum possible bit rate or throughput) of cellular links is challenging, because they do not have a directly observable rate *per se*. Even in the middle of the night, when average throughput is high and an LTE device may be completely alone in its cell, packet arrivals on a saturated link do not follow an observable isochronicity. This is a roadblock for packet-pair techniques ([13]) and other schemes to measure the available throughput.

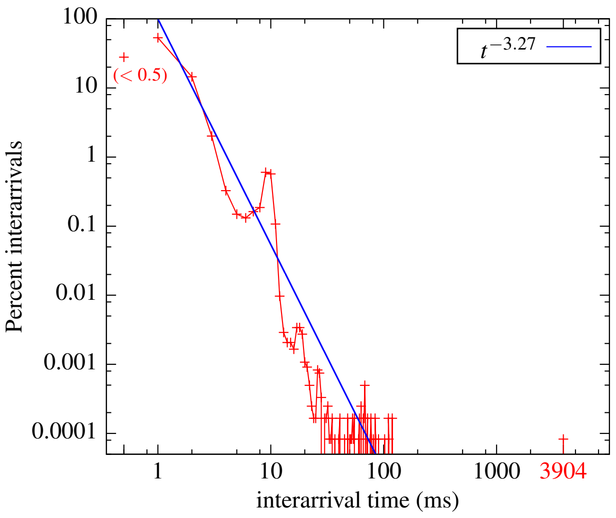
Figure 2 illustrates the *interarrival* distribution of 1.2 million MTU-sized packets received at a stationary cell phone whose downlink was saturated with these packets. For the vast majority of packet arrivals (the 99.99% that come within 20 ms of the previous packet), the distribution fits closely to a memoryless point process, or Poisson process, but with fat tails suggesting the impact of channel quality-dependent scheduling, the effect of other users, and channel outages, that yield interarrival times between 20 ms and as long as four seconds. Such a "switched" Poisson process produces a $1/f$ distribution, or *flicker noise*. The best fit is shown in the plot.[3]

A Poisson process has an underlying rate λ, which may be estimated by counting the number of bits that arrive in a long interval and dividing by the duration of the interval. In practice, however, the rate of these cellular links varies more rapidly than the averaging interval necessary to achieve an acceptable estimate.

Sprout needs to be able to estimate the link speed, both now and in the future, in order to predict how many packets it is safe to send without risking their waiting in a network queue for too long. An uncertain estimate of future link speed is worth more caution than a certain estimate, so we need to quantify our uncertainty as well as our best guess.

We therefore treat the problem in two parts. We model the link and estimate its behavior at any given time, preserving our full uncertainty. We then use the model to make forecasts about how many bytes the link will be willing to transmit from its queue in the near future. Most steps in this process can be precalculated at program startup, meaning that CPU usage (even at high throughputs) is less than 5% of a current Intel or AMD PC microprocessor. We have not tested Sprout on a CPU-constrained device or tried to optimize it fully.

Figure 2: Interarrival times on a Verizon LTE downlink, with receiver stationary, fit to a $1/f$ noise distribution.



3.1 Inferring the rate of a varying Poisson process

We model the link as a doubly-stochastic process, in which the underlying λ of the Poisson process itself varies in Brownian motion[4] with a noise power of σ (measured in units of packets per second per $\sqrt{\text{second}}$). In other words, if at time $t = 0$ the value of λ was known to be 137, then when $t = 1$ the probability distribution on λ is a normal distribution with mean 137 and standard deviation σ. The larger the value of σ, the more quickly our knowledge about λ becomes useless and the more cautious we have to be about inferring the link rate based on recent history.

Figure 3 illustrates this model. We refer to the Poisson process that dequeues and delivers packets as the service process, or packet-delivery process.

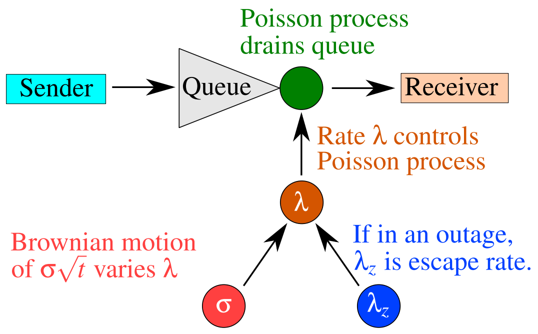
The model has one more behavior: if $\lambda = 0$ (an outage), it tends to stay in an outage. We expect the outage's

[2]An end-to-end approach may also be feasible if all sources run the same protocol, but we do not investigate that hypothesis in this paper.

[3]We can't say exactly why the distribution should have this shape, but physical processes could produce such a distribution. Cell phones experience fading, or random variation of their channel quality with time, and cell towers attempt to send packets when a phone is at the apex of its channel quality compared with a longer-term average.

[4]This is a Cox model of packet arrivals [5, 18].

Figure 3: Sprout's model of the network path. A Sprout session maintains this model separately in each direction.


Poisson process drains queue
Sender
Queue
Receiver
Rate λ controls Poisson process
λ
Brownian motion of $\sigma\sqrt{t}$ varies λ
If in an outage, λ_z is escape rate.
σ
λ_z

duration to follow an exponential distribution $\exp[-\lambda_z]$. We call λ_z the outage escape rate. This serves to match the behavior of real links, which do have "sticky" outages in our experience.

In our implementation of Sprout, σ and λ_z have fixed values that are the same for all runs and all networks. (σ = 200 MTU-sized packets per second per $\sqrt{\text{second}}$, and $\lambda_z = 1$.) These values were chosen based on preliminary empirical measurements, but the entire Sprout implementation including this model was frozen before we collected our measurement 3G and LTE traces and has not been tweaked to match them.

A more sophisticated system would allow σ and λ_z to vary slowly with time to better match more- or less-variable networks, Currently, the only parameter allowed to change with time, and the only one we need to infer in real time, is λ—the underlying, variable link rate.

To solve this inference problem tractably, Sprout discretizes the space of possible rates, λ, and assumes that:

- λ is one of 256 discrete values sampled uniformly from 0 to 1000 MTU-sized packets per second (11 Mbps; larger than the maximum rate we observed).
- At program startup, all values of λ are equally probable.
- An inference update procedure will run every 20 ms, known as a "tick". (We pick 20 ms for computational efficiency.)

By assuming an equal time between updates to the probability distribution, Sprout can precompute the normal distribution with standard deviation to match the Brownian motion per tick.

3.2 Evolution of the probability distribution on λ

Sprout maintains the probability distribution on λ in 256 floating-point values summing to unity. At every tick, Sprout does three things:

1. It *evolves* the probability distribution to the current time, by applying Brownian motion to each of the 255 values $\lambda \neq 0$. For $\lambda = 0$, we apply Brownian motion, but also use the outage escape rate to bias the evolution towards remaining at $\lambda = 0$.

2. It *observes* the number of bytes that actually came in during the most recent tick. This step multiplies each probability by the likelihood that a Poisson distribution with the corresponding rate would have produced the observed count during a tick. Suppose the duration of a tick is τ seconds (e.g., $\tau = 0.02$) and k bytes were observed during the tick. Then, Sprout updates the (non-normalized) estimate of the probabilities F:
$$F(x) \leftarrow \mathbb{P}_{\text{old}}(\lambda = x)\,\frac{(x\cdot\tau)^k}{k!}\exp[-x\cdot\tau].$$

3. It *normalizes* the 256 probabilities so that they sum to unity:
$$\mathbb{P}_{\text{new}}(\lambda = x) \leftarrow \frac{F(x)}{\sum_i F(i)}.$$

These steps constitute Bayesian updating of the probability distribution on the current value of λ.

One important practical difficulty concerns how to deal with the situation where the queue is underflowing because the sender has not sent enough. To the receiver, this case is indistinguishable from an outage of the service process, because in either case the receiver doesn't get any packets.

We use two techniques to solve this problem. First, in each outgoing packet, the sender marks its expected "time-to-next" outgoing packet. For a flight of several packets, the time-to-next will be zero for all but the last packet. When the receiver's most recently-received packet has a nonzero time-to-next, it skips the "observation" process described above until this timer expires. Thus, this "time-to-next" marking allows the receiver to avoid mistakenly observing that zero packets were deliverable during the most recent tick, when in truth the queue is simply empty.

Second, the sender sends regular heartbeat packets when idle to help the receiver learn that it is not in an outage. Even one tiny packet does much to dispel this ambiguity.

3.3 Making the packet delivery forecast

Given a probability distribution on λ, Sprout's receiver would like to predict how much data it will be safe for the sender to send without risking that packets will be stuck in the queue for too long. No forecast can be absolutely safe, but for typical interactive applications we would like to bound the risk of a packet's getting queued for longer than the sender's tolerance to be less than 5%.

To do this, Sprout calculates a *packet delivery forecast*: a cautious estimate, at the 5th percentile, of how many bytes will arrive at its receiver during the next eight ticks, or 160 ms.

It does this by *evolving* the probability distribution forward (without observation) to each of the eight ticks in the forecast. At each tick, Sprout sums over each λ to find the probability distribution of the cumulative number of packets that will have been drained by that point in time. We take the 5th percentile of this distribution as the cautious forecast for each tick. Most of these steps can also be precalculated, so the only work at runtime is to take a weighted sum over each λ.

3.4 The control protocol

The Sprout receiver sends a new forecast to the sender by piggybacking it onto its own outgoing packets.

In addition to the predicted packet deliveries, the forecast also contains a count of the total number of bytes the receiver has received so far in the connection or has written off as lost. This total helps the sender estimate how many bytes are in the queue (by subtracting it from its own count of bytes that have been sent).

In order to help the receiver calculate this number and detect losses quickly, the sender includes two fields in every outgoing packet: a sequence number that counts the number of bytes sent so far, and a "throwaway number" that specifies the sequence number offset of the *most recent* sent packet that was sent more than 10 ms prior.

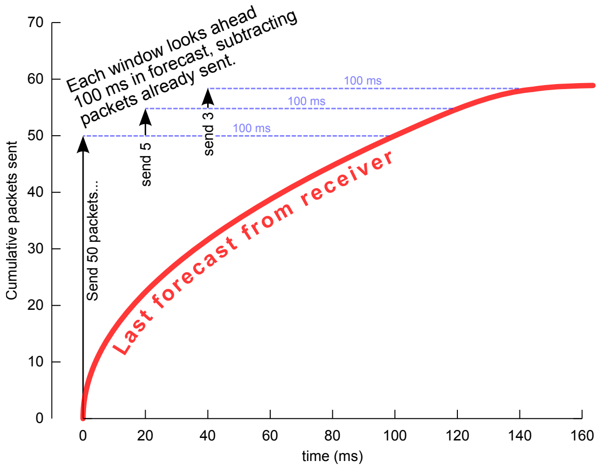
The assumption underlying this method is that while the network may reorder packets, it will not reorder two packets that were sent more than 10 ms apart. Thus, once the receiver actually gets a packet from the sender, it can mark all bytes (up to the sequence number of the first packet sent within 10 ms) as received or lost, and only keep track of more recent packets.

3.5 Using the forecast

The Sprout sender uses the most recent forecast it has obtained from the receiver to calculate a window size—the number of bytes it may safely transmit, while ensuring that every packet has 95% probability of clearing the queue within 100 ms (a conventional standard for interactivity). Upon receipt of the forecast, the sender timestamps it and estimates the current queue occupancy, based on the difference between the number of bytes it has sent so far and the "received-or-lost" sequence number in the forecast.

The sender maintains its estimate of queue occupancy going forward. For every byte it sends, it increments the estimate. Every time it advances into a new tick of the 8-tick forecast, it decrements the estimate by the amount of the forecast, bounding the estimate below at zero packets.

Figure 4: Calculating the window sizes from the forecast. The forecast represents the receiver's estimate of a lower bound (with 95% probability) on the cumulative number of packets that will be delivered over time.



To calculate a window size that is safe for the application to send, Sprout looks ahead five ticks (100 ms) into the forecast's future, and counts the number of bytes expected to be drained from the queue over that time. Then it subtracts the current queue occupancy estimate. Anything left over is "safe to send"—bytes that we expect to be cleared from the queue within 100 ms, even taking into account the queue's current contents. This evolving window size governs how much the application may transmit. Figure 4 illustrates this process schematically.

As time passes, the sender may look ahead further and further into the forecast (until it reaches 160 ms), even without receiving an update from the receiver. In this manner, Sprout combines elements of pacing with window-based flow control.

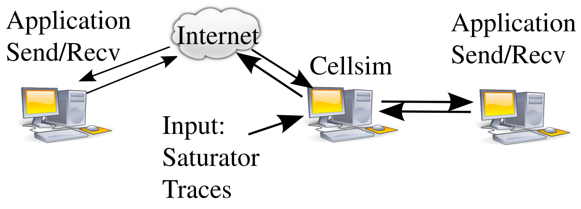
4 Experimental Testbed

We use trace-driven emulation to evaluate Sprout and compare it with other applications and protocols under reproducible network conditions. Our goal is to capture the variability of cellular networks in our experiments and to evaluate each scheme under the same set of time-varying conditions.

4.1 Saturator

Our strategy is to characterize the behavior of a cellular network by saturating its uplink and downlink at the same time with MTU-sized packets, so that neither queue goes empty. We record the times that packets actually cross the link, and we treat these as the ground truth representing all the times that packets *could* cross the link as long as a sender maintains a backlogged queue.

Figure 5: Block diagram of Cellsim



Because even TCP does not reliably keep highly variable links saturated, we developed our own tool. The Saturator runs on a laptop tethered to a cell phone (which can be used while in a car) and on a server that has a good, low-delay ($<$ 20 ms) Internet path to the cellular carrier.

The sender keeps a window of N packets in flight to the receiver, and adjusts N in order to keep the observed RTT greater than 750 ms (but less than 3000 ms). The theory of operation is that if packets are seeing more than 750 ms of queueing delay, the link is not starving for offered load. (We do not exceed 3000 ms of delay because the cellular network may start throttling or dropping packets.)

There is a challenge in running this system in two directions at once (uplink and downlink), because if both links are backlogged by multiple seconds, feedback arrives too slowly to reliably keep both links saturated. Thus, the Saturator laptop is actually connected to *two* cell phones. One acts as the "device under test," and its uplink and downlink are saturated. The second cell phone is used only for short feedback packets and is otherwise kept unloaded. In our experiments, the "feedback phone" was on Verizon's LTE network, which provided satisfactory performance: generally about 20 ms delay back to the server at MIT.

We collected data from four commercial cellular networks: Verizon Wireless's LTE and 3G (1xEV-DO / eHRPD) services, AT&T's LTE service, and T-Mobile's 3G (UMTS) service.[5] We drove around the greater Boston area at rush hour and in the evening while recording the timings of packet arrivals from each network, gathering about 17 minutes of data from each. Because the traces were collected at different times and places, the measurements cannot be used to compare different commercial services head-to-head.

For the device under test, the Verizon LTE and 1xEV-DO (3G) traces used a Samsung Galaxy Nexus smartphone running Android 4.0. The AT&T trace used a Samsung Galaxy S3 smartphone running Android 4.0. The T-Mobile trace used a Samsung Nexus S smartphone running Android 4.1.

[5]We also attempted a measurement on Sprint's 3G (1xEV-DO) service, but the results contained several lengthy outages and were not further analyzed.

Figure 6: Software versions tested

Program	Version	OS	Endpoints
Skype	5.10.0.116	Windows 7	Core i7 PC
Hangout	as of 9/2012	Windows 7	Core i7 PC
Facetime	2.0 (1070)	OS X 10.8.1	MB Pro (2.3 GHz i7), MB Air (1.8 GHz i5)
TCP Cubic	in Linux 3.2.0		Core i7 PC
TCP Vegas	in Linux 3.2.0		Core i7 PC
LEDBAT	in μTP	Linux 3.2.0	Core i7 PC
Compound TCP	in Windows 7		Core i7 PC

4.2 Cellsim

We then replay the traces in a network emulator, which we call Cellsim (Figure 5). It runs on a PC and takes in packets on two Ethernet interfaces, delays them for a configurable amount of time (the propagation delay), and adds them to the tail of a queue. Cellsim releases packets from the head of the queue to the other network interface according to the same trace that was previously recorded by Saturator. If a scheduled packet delivery occurs while the queue is empty, nothing happens and the opportunity to delivery a packet is wasted.[6]

Empirically, we measure a one-way delay of about 20 ms in each direction on our cellular links (by sending a single packet in one direction on the uplink or downlink back to a desktop with good Internet service). All our experiments are done with this propagation delay, or in other words a 40 ms minimum RTT.

Cellsim serves as a transparent Ethernet bridge for a Mac or PC under test. A second computer (which runs the other end of the connection) is connected directly to the Internet. Cellsim and the second computer receive their Internet service from the same gigabit Ethernet switch.

We tested the latest (September 2012) real-time implementations of all the applications and protocols (Skype, Facetime, etc.) running on separate late-model Macs or PCs (Figure 6).

We also added stochastic packet losses to Cellsim to study Sprout's loss resilience. Here, Cellsim drops packets from the tail of the queue according to a specified random drop rate. This approach emulates, in a coarse manner, cellular networks that do not have deep packet buffers (e.g., Clearwire, as reported in [16]). Cellsim also includes an optional implementation of CoDel, based on the pseudocode in [17].

4.3 SproutTunnel

We implemented a UDP tunnel that uses Sprout to carry arbitrary traffic (e.g. TCP, videoconferencing protocols) across a cellular link between a mobile user and a well-connected host, which acts as a relay for the user's Inter-

[6]This accounting is done on a per-byte basis. If the queue contains 15 100-byte packets, they will all be released when the trace records delivery of a single 1500-byte (MTU-sized) packet.

net traffic. SproutTunnel provides each flow with the abstraction of a low-delay connection, without modifying carrier equipment. It does this by separating each flow into its own queue, and filling up the Sprout window in round-robin fashion among the flows that have pending data.

The total queue length of all flows is limited to the receiver's most recent estimate of the number of packets that can be delivered over the life of the forecast. When the queue lengths exceed this value, the tunnel endpoints drop packets from the head of the longest queue. This algorithm serves as a dynamic traffic-shaping or active-queue-management scheme that adjusts the amount of buffering to the predicted channel conditions.

5 Evaluation

This section presents our experimental results obtained using the testbed described in §4. We start by motivating and defining the two main metrics: throughput and self-inflicted delay. We then compare Sprout with Skype, Facetime, and Hangout, focusing on how the different rate control algorithms used by these systems affect the metrics of interest. We compare against the delay-triggered congestion control algorithms TCP Vegas and LEDBAT, as well as the default TCP in Linux, Cubic, which does not use delay as a congestion signal, and Compound TCP, the default in some versions of Windows.

We also evaluate a simplified version of Sprout, called Sprout-EWMA, that eliminates the cautious packet-delivery forecast in favor of an exponentially-weighted moving average of observed throughput. We compare both versions of Sprout with a queue-management technique that must be deployed on network infrastructure. We also measure Sprout's performance in the presence of packet loss.

Finally, we evaluate the performance of competing flows (TCP Cubic and Skype) running over the Verizon LTE downlink, with and without SproutTunnel.

The implementation of Sprout (including the tuning parameters $\sigma = 200$ and $\lambda_z = 1$) was frozen before collecting the network traces, and has not been tweaked.

5.1 Metrics

We are interested in performance metrics appropriate for a real-time interactive application. In our evaluation, we report the *average throughput* achieved and the 95th-percentile *self-inflicted delay* incurred by each protocol, based on measurement at the Cellsim.

The *throughput* is the total number of bits received by an application, divided by the duration of the experiment. We use this as a measure of bulk transfer rate.

The *self-inflicted delay* is a lower bound on the end-to-end delay that must be experienced between a sender and receiver, given observed network behavior. We define it as follows: At any point in time, we find the most recently-sent packet to have arrived at the receiver. The amount of time since this packet was sent is a lower bound on the instantaneous delay that must exist between the sender's input and receiver's output in order to avoid a gap in playback or other glitch at this moment. We calculate this instantaneous delay for each moment in time. The 95th percentile of this function (taken over the entire trace) is the amount of delay that must exist between the input and output so that the receiver can recover 95% of the input signal by the time of playback. We refer to this as "95% end-to-end delay."

For a given trace, there is a lower limit on the 95% end-to-end delay that can be achieved even by an omniscient protocol: one that sends packets timed to arrive exactly when the network is ready to dequeue and transmit a packet. This "omniscient" protocol will achieve 100% of the available throughput of the link and its packets will never sit in a queue. Even so, the "omniscient" protocol will have fluctuations in its 95% end-to-end delay, because the link may have delivery outages. If the link does not deliver any packets for 5 seconds, there must be at least 5 seconds of end-to-end delay to avoid a glitch, no matter how smart the protocol is.[7]

The difference between the 95% end-to-end delay measured for a particular protocol and for an "omniscient" one is known as the *self-inflicted delay*. This is the appropriate figure to assess a real-time interactive protocol's ability to compromise between throughput and the delay experienced by users.

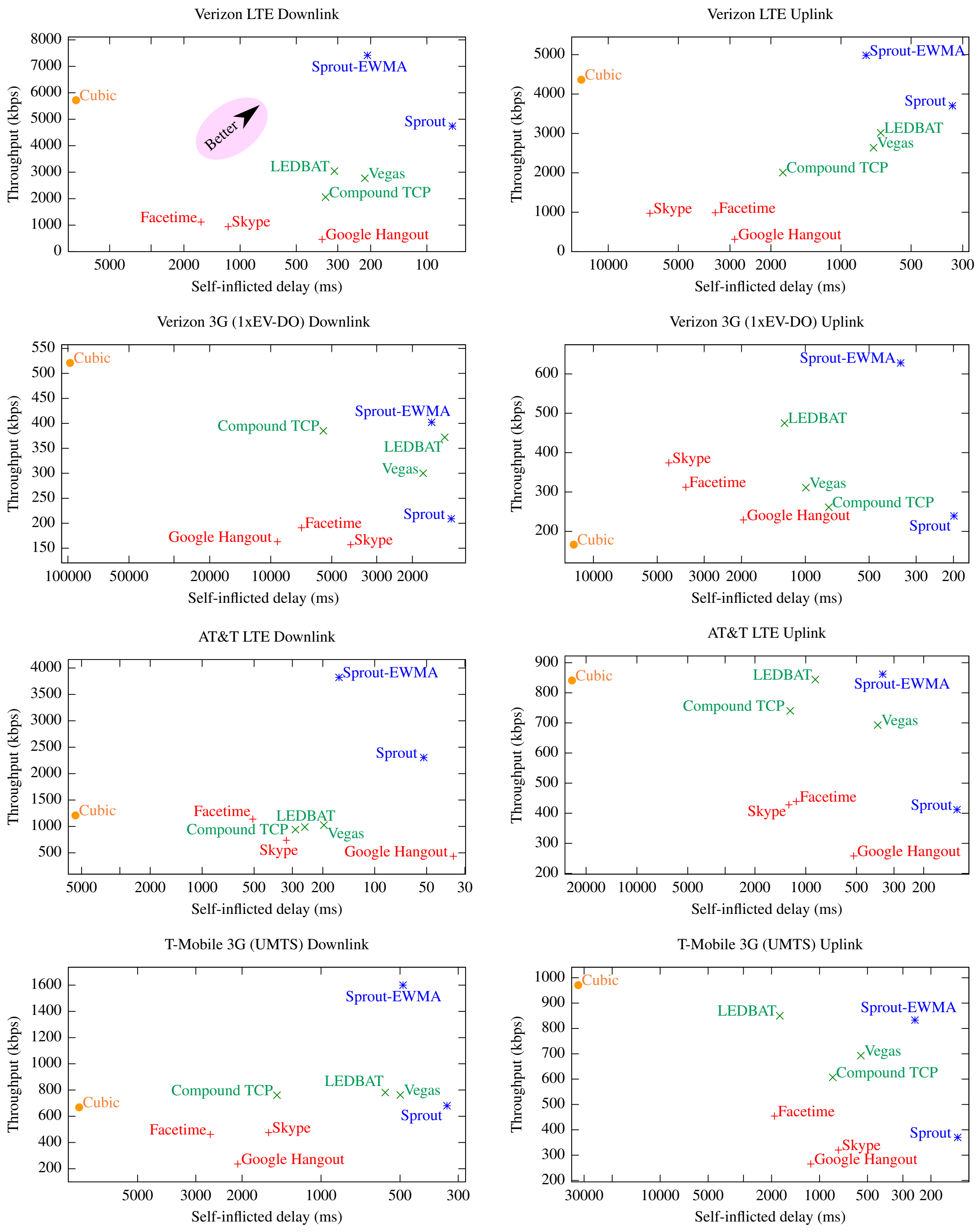
To reduce startup effects when measuring the average throughput and self-inflicted delay from an application, we skip the first minute of each application's run.

5.2 Comparative performance

Figure 7 presents the results of our trace-driven experiments for each transport protocol. The figure shows eight charts, one for each of the four measured networks, and for each data transfer direction (downlink and uplink). On each chart, we plot one point per application or protocol, corresponding to its measured throughput and self-inflicted delay combination. For interactive applications, high throughput and low delay (up and to the right) are the desirable properties. The table in the introduction shows the average of these results, taken over all the measured networks and directions, in terms of the average relative throughput gain and delay reduction achieved by Sprout.

[7]If packets are not reordered by the network, the definition becomes simpler. At each instant that a packet arrives, the end-to-end delay is equal to the delay experienced by that packet. Starting from this value, the end-to-end delay increases linearly at a rate of 1 s/s, until the next packet arrives. The 95th percentile of this function is the 95% end-to-end delay.

Figure 7: Throughput and delay of each protocol over the traced cellular links. Better results are up and to the right.



We found that Sprout had the lowest, or close to the lowest, delay across each of the eight links. On average delay, Sprout was lower than every other protocol. On average throughput, Sprout outperformed every other protocol except for Sprout-EWMA and TCP Cubic.

We also observe that Skype, Facetime, and Google Hangout all have lower throughput and higher delay than the TCP congestion-control algorithms. We believe this is because they do not react to rate increases and decreases quickly enough, perhaps because they are unable to change the encoding rapidly, or unwilling for perceptual reasons.[8] By continuing to send when the network has dramatically slowed, these programs induce high delays that destroy interactivity.

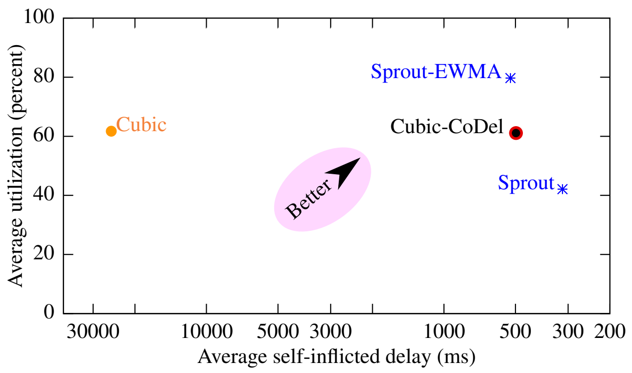
5.3 Benefits of forecasting

Sprout differs from the other approaches in two significant ways: first, it uses the packet arrival process at the receiver as the "signal" for its control algorithm (as opposed to one-way delays as in LEDBAT or packet losses or round-trip delays in other protocols), and second, it models the arrivals as a flicker-noise process to perform Bayesian inference on the underlying rate. A natural question that arises is what the benefits of Sprout's forecasting are. To answer this question, we developed a simple variant of Sprout, which we call *Sprout-EWMA*. Sprout-EWMA uses the packet arrival times, but rather than do any inference with that data, simply passes them through an exponentially-weighted moving average (EWMA) to produce an evolving smoothed rate estimate. Instead of a cautious "95%-certain" forecast, Sprout-EWMA simply predicts that the link will continue at that speed for the next eight ticks. The rest of the protocol is the same as Sprout.

The Sprout-EWMA results in the eight charts in Figure 7 show how this protocol performs. First, it outperforms all the methods in throughput, including recent TCPs such as Compound TCP and Cubic. These results also highlight the role of cautious forecasting: the self-inflicted delay is significantly lower for Sprout compared with Sprout-EWMA. TCP Vegas also achieves lower delay on average than Sprout-EWMA. The reason is that an EWMA is a low-pass filter, which does not immediately respond to sudden rate reductions or outages (the tails seen in Figure 2). Though these occur with low probability, when they do occur, queues build up and take a significant amount of time to dissipate. Sprout's forecasts provide a conservative trade-off between throughput and delay: keeping delays low, but missing legitimate opportunities to send packets, preferring to avoid the risk of filling up queues. Because the resulting throughput is relatively high, we believe it is a good choice for interactive applications. An application that is interested only in high throughput with less of an emphasis on low delay may prefer Sprout-EWMA.

[8]We found that the complexity of the video signal did not seem to affect these programs' transmitted throughputs. On fast network paths, Skype uses up to 5 Mbps even when the image is static.

Figure 8: Average utilization and delay of each scheme. Utilization is the average fraction of the cellular link's maximum capacity that the scheme achieved.



5.4 Comparison with in-network changes

We compared Sprout's end-to-end inference approach against an in-network deployment of active queue management. We added the CoDel AQM algorithm [17] to Cellsim's uplink and downlink queue, to simulate what would happen if a cellular carrier installed this algorithm inside its base stations and in the baseband modems or radio-interface drivers on a cellular phone.

The average results are shown in Figure 8. Averaged across the eight cellular links, CoDel dramatically reduces the delay incurred by Cubic, at little cost to throughput.

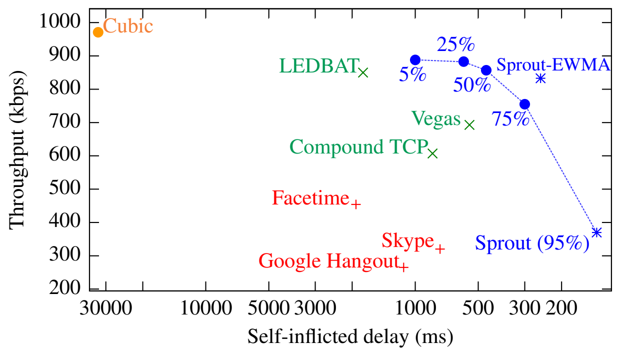
Although it is purely end-to-end, Sprout's delays are even lower than Cubic-over-CoDel. However, this comes at a cost to throughput. (Figures are given in the table in the introduction.) Sprout-EWMA achieves within 6% of the same delay as Cubic-over-CoDel, with 30% more throughput.

Rather than embed a single throughput-delay trade-off into the network (e.g. by installing CoDel on carrier infrastructure), we believe it makes architectural sense to provide endpoints and applications with such control when possible. Users should be able to decide which throughput-delay compromise they prefer. In this setting, it appears achievable to match or even exceed CoDel's performance without modifying gateways.

5.5 Effect of confidence parameter

The Sprout receiver makes forecasts of a lower bound on how many packets will be delivered with at least 95% probability. We explored the effect of lowering this confidence parameter to express a greater willingness that

Figure 9: Lowering the forecast's confidence parameter allows greater throughput at the cost of more delay. Results on the T-Mobile 3G (UMTS) uplink:



packets be queued for longer than the sender's 100 ms tolerance.

Results on one network path are shown in Figure 9. The different confidence parameters trace out a curve of achievable throughput-delay tradeoffs. As expected, decreasing the amount of caution in the forecast allows the sender to achieve greater throughput, but at the cost of more delay. Interestingly, although Sprout achieves higher throughput and lower delay than Sprout-EWMA by varying the confidence parameter, it never achieves both at the same time. Why this is—and whether Sprout's stochastic model can be further improved to beat Sprout-EWMA simultaneously on both metrics—will need to be the subject of further study.

5.6 Loss resilience

The cellular networks we experimented with all exhibited low packet loss rates, but that will not be true in general. To investigate the loss resilience of Sprout, we used the traces collected from one network (Verizon LTE) and simulated Bernoulli packet losses (tail drop) with two different packet loss probabilities, 5% and 10% (in each direction). The results are shown in the table below:

Protocol	Throughput (kbps)	Delay (ms)
Downlink		
Sprout	4741	73
Sprout-5%	3971	60
Sprout-10%	2768	58
Uplink		
Sprout	3703	332
Sprout-5%	2598	378
Sprout-10%	1163	314

As expected, the throughput does diminish in the face of packet loss, but Sprout continues to provide good throughput even at high loss rates. (TCP, which interprets loss as a congestion signal, generally suffers unacceptable slowdowns in the face of 10% each-way packet loss.) These results demonstrate that Sprout is relatively resilient to packet loss.

5.7 Sprout as a tunnel for competing traffic

We tested whether SproutTunnel, used as a tunnel over the cellular link to a well-connected relay, can successfully isolate bulk-transfer downloads from interactive applications.

We ran two flows: a TCP Cubic bulk transfer (download only) and a two-way Skype videoconference, using the Linux version of Skype.

We compared the situation of these two flows running directly over the emulated Verizon LTE link, versus running them through SproutTunnel over the same link. The experiments lasted about ten minutes each.[9]

	Direct	via Sprout	Change
Cubic throughput	8336 kbps	3776 kbps	−55%
Skype throughput	78 kbps	490 kbps	+528%
Skype 95% delay	6.0 s	0.17 s	−97%

The results suggest that interactive applications can be greatly aided by having their traffic run through Sprout along with bulk transfers. Without Sprout to mediate, Cubic squeezes out Skype and builds up huge delays. However, Sprout's conservatism about delay also imposes a substantial penalty to Cubic's throughput.

6 Related Work

End-to-end algorithms. Traditional congestion-control algorithms generally do not simultaneously achieve high utilization and low delay over paths with high rate variations. Early TCP variants such as Tahoe and Reno [10] do not explicitly adapt to delay (other than from ACK clocking), and require an appropriate buffer size for good performance. TCP Vegas [4], FAST [12], and Compound TCP [20] incorporate round-trip delay explicitly, but the adaptation is reactive and does not directly involve the receiver's observed rate.

LEDBAT [19] (and TCP Nice [21]) share our goals of high throughput without introducing long delays, but LEDBAT does not perform as well as Sprout. We believe this is because of its choice of congestion signal (one-way delay) and the absence of forecasting. Some recent work proposes TCP receiver modifications to combat bufferbloat in 3G/4G wireless networks [11]. Schemes such as "TCP-friendly" equation-based rate control [7] and binomial congestion control [1] exhibit slower transmission rate variations than TCP, and in principle could introduce lower delay, but perform poorly in the face of sudden rate changes [2].

[9]In each run, Skype ended the video portion of the call once and was restarted manually.

Google has proposed a congestion-control scheme [15] for the WebRTC system that uses an arrival-time filter at the receiver, along with other congestion signals, to decide whether a real-time flow should increase, decrease, or hold its current bit rate. We plan to investigate this system and assess it on the same metrics as the other schemes in our evaluation.

Active queue management. Active queue management schemes such as RED [8] and its variants, BLUE [6], AVQ [14], etc., drop or mark packets using local indications of upcoming congestion at a bottleneck queue, with the idea that endpoints react to these signals before queues grow significantly. Over the past several years, it has proven difficult to automatically configure the parameters used in these algorithms. To alleviate this shortcoming, CoDel [17] changes the signal of congestion from queue length to the delay experienced by packets in a queue, with a view toward controlling that delay, especially in networks with deep queues ("bufferbloat" [9]).

Our results show that Sprout largely holds its own with CoDel over challenging wireless conditions without requiring any gateway modifications. It is important to note that network paths in practice have several places where queues may build up (in LTE infrastructure, in baseband modems, in IP-layer queues, near the USB interface in tethering mode, etc.), so one may need to deploy CoDel at all these locations, which could be difficult. However, in networks where there is a lower degree of isolation between queues than the cellular networks we study, CoDel may be the right approach to controlling delay while providing good throughput, but it is a "one-size-fits-all" method that assumes that a single throughput-delay tradeoff is right for all traffic.

7 Limitations and Future Work

Although our results are encouraging, there are several limitations to our work. First, as noted in §2 and §3, an end-to-end system like Sprout cannot control delays when the bottleneck link includes competing traffic that shares the same queue. If a device uses traditional TCP outside of Sprout, the incurred queueing delay—seen by Sprout and every flow—will be substantial.

Sprout is not a traditional congestion-control protocol, in that it is designed to adapt to varying link conditions, not varying cross traffic. In a cellular link where users have their own queues on the base station, interactive performance will likely be best when the user runs bulk and interactive traffic *inside* Sprout (e.g. using Sprout-Tunnel), not alongside Sprout. We have not evaluated the performance of multiple Sprouts sharing a queue.

The accuracy of Sprout's forecasts depends on whether the application is providing offered load sufficient to saturate the link. For applications that switch intermittently on and off, or don't desire high throughput, the transient behavior of Sprout's forecasts (e.g. ramp-up time) becomes more important. We did not evaluate any non-saturating applications in this paper or attempt to measure or optimize Sprout's startup time from idle.

Finally, we have tested Sprout only in trace-based emulation of eight cellular links recorded in the Boston area in 2012. Although Sprout's model was frozen before data were collected and was not "tuned" in response to any particular network, we cannot know how generalizable Sprout's algorithm is without more real-world testing.

In future work, we are eager to explore different stochastic network models, including ones trained on empirical variations in cellular link speed, to see whether it is possible to perform much better than Sprout if a protocol has more accurate forecasts. We think it will be worthwhile to collect enough traces to compile a standardized benchmark of cellular link behavior, over which one could evaluate any new transport protocol.

8 Conclusion

This paper presented Sprout, a transport protocol for real-time interactive applications over Internet paths that traverse cellular wireless networks. Sprout improves on the performance of current approaches by modeling varying networks explicitly. Sprout has two interesting ideas: the use of packet arrival times as a congestion signal, and the use of probabilistic inference to make a cautious forecast of packet deliveries, which the sender uses to pace its transmissions. Our experiments show that forecasting is important to controlling delay, providing an end-to-end rate control algorithm that can react at time scales shorter than a round-trip time.

Our experiments conducted on traces from four commercial cellular networks show many-fold reductions in delay, and increases in throughput, over Skype, Facetime, and Hangout, as well as over Cubic, Compound TCP, Vegas, and LEDBAT. Although Sprout is an end-to-end scheme, in this setting it matched or exceeded the performance of Cubic-over-CoDel, which requires modifications to network infrastructure to be deployed.

9 Acknowledgments

We thank Shuo Deng, Jonathan Perry, Katrina LaCurts, Andrew McGregor, Tim Shepard, Dave Täht, Michael Welzl, Hannes Tschofenig, and the anonymous reviewers for helpful discussion and feedback. This work was supported in part by NSF grant 1040072. KW was supported by the Claude E. Shannon Research Assistantship. We thank the members of the MIT Center for Wireless Networks and Mobile Computing, including Amazon.com, Cisco, Google, Intel, Mediatek, Microsoft, ST Microelectronics, and Telefonica, for their support.

REFERENCES

[1] D. Bansal and H. Balakrishnan. Binomial Congestion Control Algorithms. In *INFOCOM*, 2001.

[2] D. Bansal, H. Balakrishnan, S. Floyd, and S. Shenker. Dynamic Behavior of Slowly-Responsive Congestion Control Algorithms. In *SIGCOMM*, 2001.

[3] P. Bender, P. Black, M. Grob, R. Padovani, N. Sindhushyana, and A. Viterbi. A bandwidth efficient high speed wireless data service for nomadic users. *IEEE Communications Magazine*, July 2000.

[4] L. S. Brakmo, S. W. O'Malley, and L. L. Peterson. TCP Vegas: New Techniques for Congestion Detection and Avoidance. In *SIGCOMM*, 1994.

[5] D. Cox. Long-range dependence: A review. In *H.A. David and H.T. David, editors, Statistics: An Appraisal*, pages 55–74. Iowa State University Press, 1984.

[6] W. Feng, K. Shin, D. Kandlur, and D. Saha. The BLUE active queue management algorithms. *IEEE/ACM Trans. on Networking*, Aug. 2002.

[7] S. Floyd, M. Handley, J. Padhye, and J. Widmer. Equation-Based Congestion Control for Unicast Applications. In *SIGCOMM*, 2000.

[8] S. Floyd and V. Jacobson. Random Early Detection Gateways for Congestion Avoidance. *IEEE/ACM Trans. on Networking*, 1(4), Aug. 1993.

[9] J. Gettys and K. Nichols. Bufferbloat: Dark buffers in the internet. *Queue*, 9(11):40:40–40:54, Nov. 2011.

[10] V. Jacobson. Congestion Avoidance and Control. In *SIGCOMM*, 1988.

[11] H. Jiang, Y. Wang, K. Lee, and I. Rhee. Tackling bufferbloat in 3g/4g mobile networks. *NCSU Technical report*, March 2012.

[12] C. Jin, D. Wei, and S. Low. Fast tcp: motivation, architecture, algorithms, performance. In *INFOCOM 2004. Twenty-third AnnualJoint Conference of the IEEE Computer and Communications Societies*, volume 4, pages 2490 – 2501 vol.4, march 2004.

[13] S. Keshav. Packet-Pair Flow Control. *IEEE/ACM Trans. on Networking*, Feb. 1995.

[14] S. Kunniyur and R. Srikanth. Analysis and design of an adaptive virtual queue (AVQ) algorithm for active queue management. In *SIGCOMM*, 2001.

[15] H. Lundin, S. Holmer, and H. Alvestrand. A Google Congestion Control Algorithm for Real-Time Communication on the World Wide Web. http://tools.ietf.org/html/draft-alvestrand-rtcweb-congestion-03, 2012.

[16] R. Mahajan, J. Padhye, S. Agarwal, and B. Zill. High performance vehicular connectivity with opportunistic erasure coding. In *USENIX*, 2012.

[17] K. Nichols and V. Jacobson. Controlling queue delay. *ACM Queue*, 10(5), May 2012.

[18] V. Paxson and S. Floyd. Wide-Area Traffic: The Failure of Poisson Modeling. *IEEE/ACM Trans. on Networking*, 3(3):226–244, June 1995.

[19] S. Shalunov, G. Hazel, J. Iyengar, and M. Kuehlewind. Low Extra Delay Background Transport (LEDBAT), 2012. IETF RFC 6817.

[20] K. Tan, J. Song, Q. Zhang, and M. Sridharan. A Compound TCP Approach for High-Speed and Long Distance Networks. In *INFOCOM*, 2006.

[21] A. Venkataramani, R. Kokku, and M. Dahlin. TCP Nice: a mechanism for background transfers. *SIGOPS Oper. Syst. Rev.*, 36(SI):329–343, Dec. 2002.