Alfalfa: A Videoconferencing Protocol for Cellular Wireless Networks

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

Abstract

Alfalfa is an Internet videoconferencing protocol that works well over cellular wireless networks, where link speeds change dramatically with time, and current transport protocols build up lengthy in-network queues. Alfalfa does not use TCP-style reactive congestion control, but instead uses a stochastic model of the network path to forecast how many bytes are safe to send, while bounding the risk that its packets will be queued inside the network for too long. This forecast is used to govern the coded size of each frame of video, yielding a continuously-variable video compression ratio that achieves high utilization of the varying link without filling up network queues. In evaluations on traces from four commercial LTE and 3G networks, Alfalfa's rate control scheme, 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, it reduced delay by a factor of 7.2 while achieving 4.4× the bit rate, and compared with Apple's Facetime, it reduced delay by a factor of 8.7 with 1.9× the bit rate.

1 Introduction

Cellular wireless networks have become a dominant form 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 bit rates and occasional multi-second outages in one or both directions, especially when the user is mobile (see §2). 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 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 (and sometimes high) packet delays.

For an interactive application such as a videoconferencing program that requires both high throughput and low delay, these conditions are challenging to handle. If the application sends at too low a rate, it will waste the opportunity for higher-quality service when the cellular 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, causing unpleasant user-visible gaps in playback and destroying interactivity.

Furthermore, even if the transport protocol knew the varying rate of the network path exactly, a videoconferencing application would not benefit unless it could rapidly adapt the quality of its video encoding to meet the available rate. Unfortunately, this task is difficult and done poorly by existing videoconferencing applications. To match a time-varying rate, the application needs to control the coded size of each outgoing video frame. To help the client recover from loss, the application needs to ensure that transmitted video frames depend only on frames available to the receiver, and not frames that have been lost. Current video coder libraries generally don't expose this level of control to applications.

Our experiments with Microsoft's Skype, Google Hangout, and Apple Facetime running over traces from commercial 3G and LTE networks show the shortcomings of the transport protocols in use and the lack of fine-grained 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 in a jump and try again. They often create a large backlog of queued packets in the network, and when that happens, only after several seconds and a user-visible outage do they switch to a lower rate.

This paper presents *Alfalfa*, a videoconferencing protocol designed for variable-quality networks. Alfalfa makes two principal contributions. The first is a *predictive* rate control algorithm, which we call *Sprout* (§3). Sprout uses the receiver's observed packet interarrival 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. 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.

The second contribution is a video coder/decoder (codec) interface that can smoothly vary its quality on a frame-by-frame basis to match Sprout's network capacity forecasts, and can recover swiftly from losses (§4). Alfalfa treats compressed video not so much as a stream to be played back in order, but as a collection of "diffs" between arbitrary frames that may be marshaled to advance the receiver to the current video frame in the most efficient manner.

Alfalfa varies its frame rate and bit rate continuously, obeying the Sprout algorithm's estimate of how much data the network can bear without amassing a queue. Using Sprout's forecasts and the explicit modeling of delay, Alfalfa is able to use the available network rate even when that rate is highly variable.

Alfalfa represents an end-to-end, application-layer solution to the challenges presented by cellular networks: variable (and unpredictable) link speeds and delays caused by packets waiting to transit a wireless link with link-layer acknowledgments and retransmissions. We have implemented Alfalfa's codec interface on top of existing encoder and decoder implementations for MPEG-2 part 2 (H.262) video (§5), but our scheme is more generally applicable (see §4).

We conducted a trace-driven experimental evaluation (details in §6) 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 Alfalfa/Sprout to Skype, Hangout, Facetime, and several congestion control algorithms. The following table summarizes the average relative bit rate improvement and average reduction in self-inflicted delay[1] for Alfalfa/Sprout compared to the various other schemes, averaged over all four cellular networks:

App/protocol	Avg. speedup	Delay reduction (from avg. delay)
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	Same	2.8× (0.89 s)
CUBIC	1.1×	79× (25 s)

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

2 Context and Challenges

This section highlights the networking challenges in designing an adaptive videoconferencing application 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, 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[2]. The base station schedules data transmissions with each user based on fairness and channel quality. Typically, each user's device is scheduled for a fixed time slice over 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 packet 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 to delay is muted, although competing demand can affect the maximum throughput that any user may receive.

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" [12] 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 [19]).

The high delays in cellular wireless networks cannot simply be blamed on bufferbloat, because there is no single 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 be-

[1] This delay 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 §6.

[2] Networking folklore says that per-user queues "don't scale", but cellular networks demonstrate that they are both practical and useful!

come 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 §6, we evaluate CoDel [20], a recent AQM technique, together with a modern TCP variant (CUBIC, which is the default in Linux), and found that on half of our tested network paths, CoDel slows down a bulk TCP transfer that has the link to itself, in one case by more than 40%. CoDel dramatically improved TCP's end-to-end delays compared with a gateway that indefinitely queues packets, but not as much as Sprout (even without active queue management). In some cases Sprout surpassed TCP-over-CoDel in throughput as well. 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.

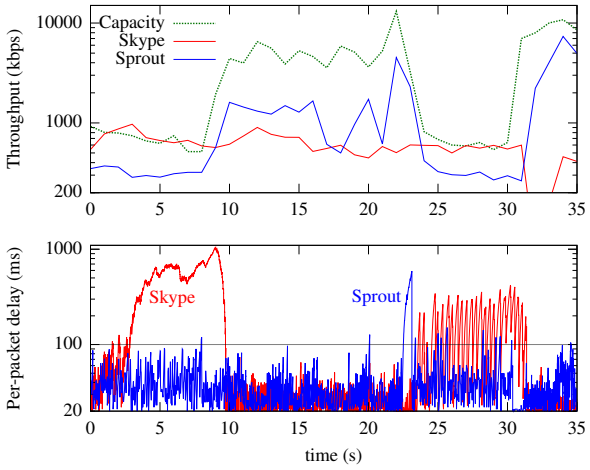
Alfalfa's improvements over existing videoconferencing systems are found when these queueing delays are self-imposed. This is typically the case when the application is running on a mobile device, because cellular network operators generally maintain a separate queue for each customer, a videoconferencing application is commonly the only significant network user running on a customer's device while in use, and the wireless link is typically the bottleneck. An important limitation of this approach is that in cases where these conditions don't hold—for example, if a competing application running on the same device, or with access to the same bottleneck queue, is sending aggressively—Alfalfa'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 videoconferencing applications do not handle these well. Figure 1 shows an illustrative section of our trace from the Verizon LTE uplink, whose capacity varied up and down by almost an order of magnitude within one second. From 3 to 9 seconds into the plot, and from 24 to 31 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 $t = 24$ seconds), but then repairs them.

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

Figure 1: Skype and Sprout on the Verizon LTE uplink trace. For Skype, even a small overshoot in throughput leads to a large standing queue. Sprout tries to keep each packet's delay less than 100 ms with 95% probability.



2.3 Challenges

A good videoconferencing system 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, and cope with asymmetric outages, and recover gracefully afterwards.

4. At the application layer, the quality of delivered video and audio must match the achievable link rate to provide the best possible experience to the user.

Our experimental results show that previous work (see §7) does not address these challenges satisfactorily. These methods are all *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 interarrival time 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. In addition, at the application layer, Alfalfa's finer-grained interface provides a tighter coupling with the rate control algorithm, enabling better application adaptation.

3 THE SPROUT CONTROL 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 ([12]), 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.[3]

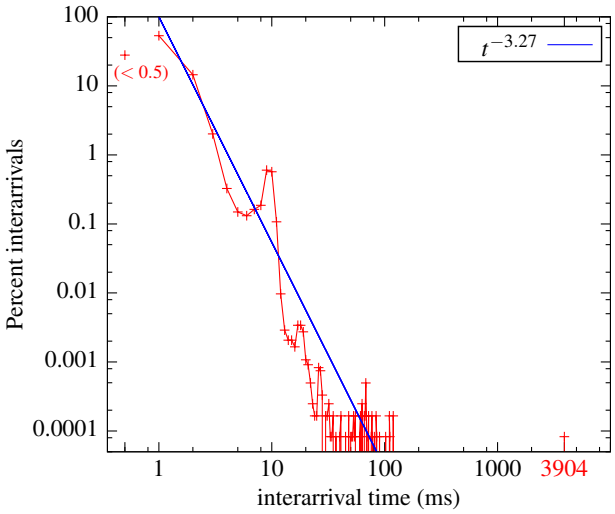
In our measurements, we found that estimating the capacity (by which we mean the maximum possible bit rate or throughput) of cellular links is challenging, because they do not have a directly observable rate *per se*. Even in the middle of the night, when average throughput is high and an LTE device 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 ([16]) 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.[4]

[3] 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.

[4] 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.

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



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.

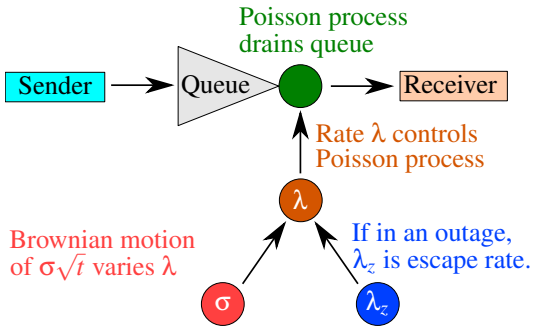
We therefore treat the problem in two parts. We model the link and estimate its behavior at any given time, preserving our full uncertainty. We then use the model to make forecasts about how many bytes the link will be willing to transmit from its queue in the near future. Most steps in this process 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.

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[5] 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

[5] This is a Cox model of packet arrivals [7, 22].

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



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 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 root-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 means multiplying each probability by the likelihood that a Poisson distribution with the corresponding rate would have produced the observed count during a tick. If k bytes were observed, we update

$$\mathbb{P}_{\text{new}}(\lambda = x) \leftarrow \mathbb{P}_{\text{old}}(\lambda = x) \cdot \frac{x^k}{k!} \exp[-x].$$

3. It *normalizes* the 256 probabilities so that they sum to unity.

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 (e.g., a coded video frame), 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 small packets (e.g., containing audio data) 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 interactive videoconferencing we would like to bound the risk of a packet's getting queued for more than 100 ms (a conventional standard for interactivity) 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.[6]

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

Sprout's sender uses the most recent forecast it has received 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. Upon receipt of the forecast, it timestamps it and estimates the current queue occupancy, based on the difference between the number of bytes it has sent so far and "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—unless this would cause an underflow.

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"—it is bytes that we expect to be cleared from the queue within 100 ms, even taking into account the current contents of the queue. This is the window size that is returned to the rest of Alfalfa.

4 A Framework for Adaptive Video

We treat the videoconferencing problem as one of state synchronization (as in Mosh [28]), and we assume that the sender's goal is to advance the receiver to the current frame of video as quickly as possible. Periodically an audio encoder may produce packets of compressed audio, which should be able to reach the receiver swiftly without sitting in a sender-side or network queue. We do not explicitly try to produce smooth video with regularly-spaced frames, which requires buffering and extra delay. The assumption is that the receiver wants our video and audio in sync and with less than 100 ms of latency.[7]

Existing video encoder interfaces generally expose a stream abstraction to the application. The application sets the encoder to a particular target bit rate, and provides raw frames, which the encoder turns into a coded video stream for sending to the receiver. Generally the first frame in such a stream will be "intra-coded," meaning without reference to other frames. Such frames tend to be larger than "predictively-coded" frames, or P-frames, which depend on one or more previous frames in order to be correctly decoded.[8] Subsequent frames in a videoconferencing stream will generally be P-frames.[9] The receiver provides this octet stream to the decoder, which produces a sequence of reconstructed frames. If part of a coded frame is missing, the decoder does its best to render the parts of the frame that it can, and meanwhile notifies the sender that it has suffered a "slice loss" or "picture loss" [21]. The encoder will reply with a "repair" that usually means sending an intra-frame or part of an

[6] Most of these steps can also be precalculated, so the only work at runtime is to take a weighted sum over each λ.

[7] We have noticed that existing applications, especially Google Hangout, make a different choice: they buffer video to produce smooth playback, with the consequence that video can be delayed by a few hundred milliseconds relative to received audio. In future work, we plan to measure video quality and delay end-to-end.

[8] These techniques are common to DPCM-based video compression methods, which includes the popular formats MPEG-2 (H.262), H.263, MPEG-4 part 2, MPEG-4 part 10 (H.264), VP8, VC-1, and the forthcoming H.265.

[9] Streams for videoconferencing generally are optimized for low delay and do not use bilinearly-predicted frames, or B-frames, which can depend on future as well as past frames.

intra-frame.[10] The application generally cannot change to a new bit rate without risking that the encoder will reinitialize the video stream and send a large intra-coded frame.

Such techniques are commonly used to code high-quality video for videoconferencing applications. The stream abstraction's simplicity is useful because it allows the application to treat the codec like a black box and to substitute new codecs when more sophisticated compression methods are developed, and this interface is generally the only interface envisioned by compressed video specifications and supported by codec implementations.

However, on a mobile cellular network, available bytes (as estimated by Sprout) may vary quickly, on a frame-by-frame basis. A system like Alfalfa whose priority is low latency (for audio as well as video data) needs control over the coded length of each P-frame. A network video application also would benefit from explicit control over the dependencies of each P-frame, in order to work well on lossy links and recover swiftly from outages.

We propose that codec implementers expose a richer interface, which we believe will allow higher-quality mobile videoconferencing. Instead of treating the video stream itself as opaque, the codec provides a "video state" object that represents one frame of video and its associated coder state, and supports a "diff" and "patch" method. The application can ask for a "diff" between any two states, of a requested length, and receive a bit string that conveys a decoder from one state to another at the quality permitted by the requested length. At the receiver, the corresponding "patch" method allows the application to reconstruct transmitted frames. Finally, the state object exposes a "quality" method calculating the fidelity of a reconstructed frame compared with the original. We refer to this as an "explicit-state" interface to video compression and decompression.

This interface gives Alfalfa the freedom to make tradeoffs among encoding choices based on knowledge about the network path. For example, Sprout may report that 10 kilobytes are available in the network path without risking an unacceptable self-inflicted delay. But if Alfalfa's state synchronization protocol has not yet received an acknowledgment to its most recently transmitted frame, should Alfalfa transmit a 10 kilobyte diff that uses, as its predicate, the most recently transmitted frame (which may never arrive)? Or should it transmit a diff relative to an older frame that the receiver has acknowledged?

If the "quality" is not markedly different, the latter is preferable, because it allows the receiver to definitely reconstruct the new frame, even if the previous frame has been lost. Such decisions are best made by Alfalfa, not the codec. Similarly, to recover from outages, Alfalfa bases its diffs on frames it knows the receiver already has. Alfalfa never needs to send an intra-coded frame.[11]

We refer to a "video state" object rather than a frame, because current video coding methods maintain a large amount of global state as they run, in addition to predicate frames themselves. For example, MPEG-2 maintains two quantization tables (expressing the relative importance of spatial frequency components) that persist frame-to-frame, as well as summary information like the overall resolution and bit depth. H.264 and VP8 are considerably more complicated and evolve rich entropy coding state from frame to frame.

Unfortunately, coded video specifications generally do not clearly separate out "per-frame" state and global state, because such specifications do not envision an explicit-state interface to video coding. It requires knowledge of the internals of a video compression method to convert a traditional encoder to our "explicit state" interface. We have implemented the interface for the MPEG-2 part 2 (H.262) coded video format, which is an older and simpler method than VP8 or MPEG-4 part 10 (H.264).

5 IMPLEMENTATION

Alfalfa's networking protocol is the State Synchronization Protocol (SSP), which we previously used to develop Mosh [28]. SSP works to synchronize objects over a network path, as long as the objects support a "diff" and "patch" interface. The major modification we made to SSP to support Alfalfa is that in Mosh, "diff" and "patch" are round-trip operations, so the protocol is lossless. In Alfalfa, the "diffs" are lossy and the available length for a diff varies according to Sprout's forecast window size.

Our "patch" implementation is built around the `libmpeg2` routines for decoding an MPEG-2 picture with prerequisites in place. In the case of MPEG-2, these are one or two previously-transmitted frames, two quantization tables, and summary information about the stream (e.g. resolution). Around these routines, we built a C++ library for marshaling the prerequisites, and firing off threads that can decode pieces of a frame and its prerequisites in parallel.

Our "diff" implementation is built around the corresponding routines in `mpeg2enc` to code a picture given these same prerequisites. For simplicity and at the cost of video quality, we simply use the MPEG-2 default quantization tables and do not allow them to change. We modified `mpeg2enc`'s two-pass rate controller to adjust

[10] Some video coding schemes, including H.264 and VP8, have the ability to reference two or more predicate frames at the receiver, meaning that a coder can sometimes perform a "repair" by simply referencing an earlier frame, rather than sending a new intra-coded frame.

[11] At program startup, both sides have an implicit frame 0 that is an all-black frame. Subsequently frames are predictively coded from this frame.

the quality of quantization for each section of the frame based on the overall bit budget supplied by Sprout.[12]

Currently, Alfalfa simply sends opaque "audio" packets isochronously at 20 Hz and does not decode or play them. We intend to fill these packets with data from a constant-bit-rate compressed audio stream, such as the Opus audio codec reference implementation ([26]).

Contemporary video compression schemes, such as H.264 and VP8, are more complex and will require more effort to translate to an explicit-state framework, but they generally provide roughly twice the coding efficiency of MPEG-2. We have not quantitatively evaluated Alfalfa's end-to-end video performance or that of competing schemes. For the remainder of this paper, we focus on evaluating Alfalfa's rate control by itself, using achieved bit rate as a proxy for video quality.

5.1 Experimental Testbed

We use trace-driven emulation to evaluate Alfalfa/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 (variable) conditions.

5.1.1 Saturator

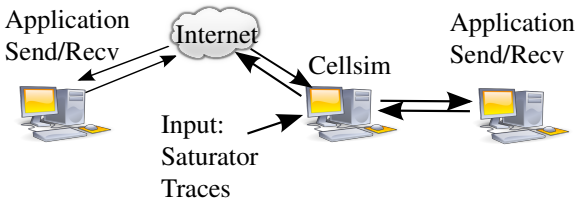
Our strategy is to characterize the behavior of a cellular network by saturating its uplink and downlink at the same time with MTU-sized packets, so that neither queue goes empty. We record the times that packets actually cross the link, and we treat these as the ground truth representing all the times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

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 Internet connection with a low-delay ($<$ 20 ms) 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.)

[12] One subtlety is that MPEG-2 encoders prefer to have both the original, high-quality version of the prerequisite frames as well as the reconstructed version, which is the only version actually available to the decoder. Nonetheless, by allowing the encoder to have reference to the *original* version of the frame, the task of motion estimation on low-quality video becomes more stable. For simplicity at the cost of coding efficiency, Alfalfa simply provides the reconstructed version as the original version.

Figure 4: Block diagram of Cellsim



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 whose uplink and downlink are saturated, and a second cell phone used only for short feedback packets and kept otherwise unloaded. In our experiments, we kept this "feedback phone" on Verizon's LTE network, which provided satisfactory performance (generally about 20 ms delay back to the server).

5.1.2 Cellsim

We then replay the traces in a network emulator, known as Cellsim (Figure 4). 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.[13]

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. One computer was connected directly to the Internet (via gigabit Ethernet). The second computer was connected to the same gigabit Ethernet switch via Cellsim.

We also added stochastic packet losses to Cellsim to

[13] 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.

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 [19]). Cellsim also includes an optional implementation of CoDel, based on the psuedocode in [20].

6 EVALUATION

This section presents our experimental results obtained as described in §5.1. We start by describing the trace data, and then motivate and define the two main metrics, bit rate and self-inflicted delay. We then compare Alfalfa with Skype, Facetime, and Hangout, focusing on how the different rate control algorithms used by these systems affect the metrics of interest. Next, we compare Sprout with delay-based congestion control algorithms: TCP Vegas, Compound TCP, and LEDBAT. We also prepared a modified version of Sprout that eliminates the cautious packet-delivery forecast in place of an exponentially-weighted moving average of observed throughput, and compare this with bulk-transfer TCP (CUBIC), with and without CoDel AQM. We also measured Sprout's performance in the presence of two packet loss rates.

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.

6.1 Wireless Network Traces

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. 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. Later, we replayed the traces to emulate the same network conditions for pairs of computers running each protocol.[14]

6.2 Metrics

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

The *bit rate* is the total number of bits transmitted by an application (measured as the sum of Ethernet frame sizes), divided by the duration of the experiment. We use this as a measurement of video quality or bulk throughput available. Although it may be preferable to report a perceptual measurement of video quality (such as PSNR or SSIM), this is challenging in the context of a videoconferencing application, which does not preserve a one-to-one correspondence between transmitted and received frames of video. Given the significant advances that have been made in video compression, compression schemes are generally able to make productive use of each bit available to them.

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, or in practice for each millisecond. The 95% percentile of this value (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."

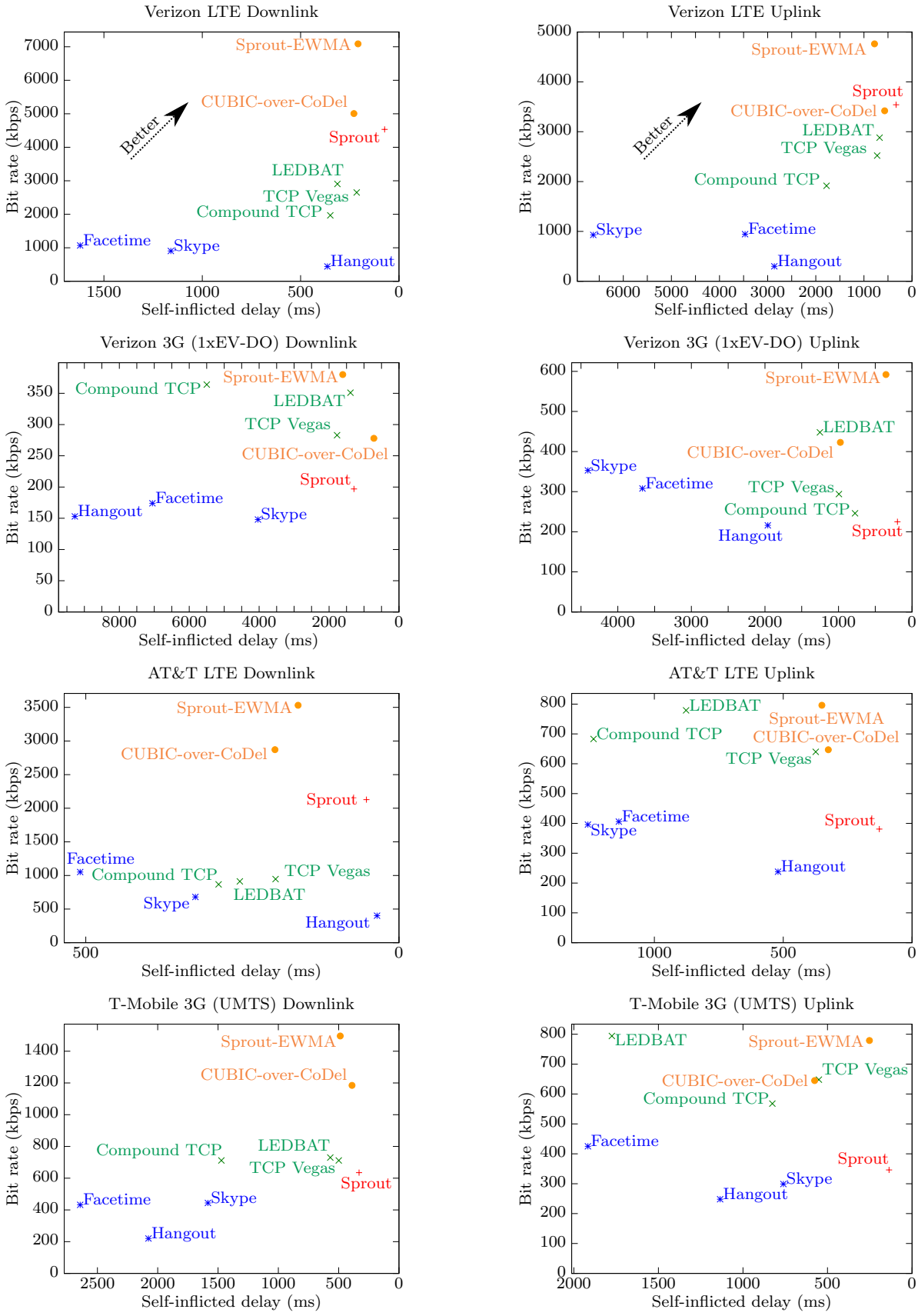
For a given trace, there is a lower limit on the 95% end-to-end delay that can be achieved even by an ideal protocol: one that sends packets timed to arrive exactly when the network is ready to dequeue and transmit a packet. This "ideal" protocol will achieve 100% of the available throughput of the link and its packets will never sit in a queue. Even so, the "ideal" protocol will have nonzero 95% end-to-end delay, because the link has nonzero propagation delay and 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.)

The difference between the 95% end-to-end delay measured from a particular protocol and the minimum possible figure is known as the *self-inflicted delay*. This is the appropriate figure to assess a real-time interactive protocol's ability to compromise between aggregate throughput and the end-to-end delay experienced by users, normalized to a particular quality of the received signal.

6.3 Comparative Performance

Page 10 presents the results of our trace-driven experiments. 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 bit rate and self-inflicted delay combination. For a videoconferencing application, a high bit rate and low delay are the desirable properties. (The table in the introduction shows a subset of these results, averaged

[14] Because the traces were collected at different times and places, the measurements cannot be used to compare different commercial services head-to-head.


Verizon LTE Downlink
Bit rate (kbps)
Self-inflicted delay (ms)
Better
Sprout-EWMA
CUBIC-over-CoDel
Sprout
LEDBAT
TCP Vegas
Compound TCP
Facetime
Skype
Hangout
Verizon LTE Uplink
Verizon 3G (1xEV-DO) Downlink
Verizon 3G (1xEV-DO) Uplink
AT&T LTE Downlink
AT&T LTE Uplink
T-Mobile 3G (UMTS) Downlink
T-Mobile 3G (UMTS) Uplink

over all the measured networks and directions, in terms of the average relative bit rate gain and average delay reduction achieved by Alfalfa/Sprout.)

We found that Sprout has the lowest (or close to the lowest) self-inflicted delay in all eight networks. These delays are often lower than CUBIC-over-CoDel, an AQM scheme, which is a strong result for an end-to-end scheme (we don't show CUBIC results because the delays absent AQM exceed 25 seconds). These results also show that Sprout's first priority is controlling delay, even if this entails some under-utilization. (The next subsection presents a Sprout variant that is less conservative.)

The second observation is that Skype, Facetime, and Hangout all have lower throughput and higher delay than the transport protocols. We believe this is because they don't react to rate increases or decreases quickly enough, perhaps because they are unable to (or don't want to, for perceptual reasons) change the encoding rapidly.[15] By continuing to send when the network has dramatically slowed, these programs induce high delays that destroy interactivity.

6.4 Benefits of Forecasting

Sprout differs from other approaches in two significant ways: first, it uses the packet inter-arrival time distribution 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 interarrivals 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 develop a simple variant of Sprout, which we term *Sprout-EWMA*. Sprout-EWMA uses the packet interarrivals, but rather than do any sophisticated inference with that data, simply passes it through an exponential weighted moving average (EWMA) to produce an evolving smoothed rate estimate. Instead of a cautious "95%-certain" forecast, Sprout 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" points in the eight charts in Figure 6.3 show how this protocol performs compared with the other schemes. First, it out-performs all the methods in bit rate, including modern TCPs such as Compound TCP and CUBIC (both with and without CoDel).These results also clearly highlight the role of cautious forecasting: the self-inflicted delay is significantly lower for Sprout compared with Sprout-EWMA (whose delay is comparable to a network with CoDel). 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 bit rate and delay: keeping delays low, but not taking advantage of legitimate opportunities to send packets, preferring to avoid the risk of filling up queues. But because the resulting throughput is high enough, we believe it is a good choice for interactive videoconferencing applications. We also believe that an application that is interested only in high throughput and not low delay would use Sprout-EWMA. We believe that it makes architectural sense to provide end points and applications with such control when possible, rather than embed a single policy within the routers in the network.

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

6.5 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	Bit rate (kbps)	Delay (ms)
Downlink		
Sprout	4533	73
Sprout-5%	3796	60
Sprout-10%	2646	58
Uplink		
Sprout	3540	332
Sprout-5%	2484	378
Sprout-10%	1112	314

As expected, the throughput does reduce in the face of packet loss, but the protocol continues to provide a good bit rate even at high loss rates. These results demonstrate that Sprout is resilient to packet losses.

7 Related Work

End-to-end algorithms. Previously developed congestion control algorithms don't simultaneously achieve high utilization and low delay over paths with high rate variations. Early TCP variants such as Tahoe and Reno [13] do not explicitly adapt to delay (other than from ACK clocking), and require an appropriate buffer size for good performance. TCP Vegas [4], FAST [15], and Compound TCP [25] incorporate round-trip delay explicitly, but the adaptation is reactive and does not directly involve the receiver's observed rate.

LEDBAT [18] (and TCP Nice [27]) share our goals of high throughput without introducing long delays, but

LEDBAT does not perform as well as Sprout 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 [14]. We note that it is not enough to reactively throttle the receiver window when delays rise, because one needs mechanisms to predict rate increases and outages as well. Schemes such as "TCP-friendly" equation-based rate control [10] and binomial congestion control [2] exhibit slower transmission rate variations than TCP, and in principle could introduce lower delay, but they have a hard time performing well in the face of sudden rate changes [3].

Active queue management. Active queue management schemes such as RED [11] and its variants, BLUE [9], AVQ [17], etc. drop or mark packets using local indications of upcoming congestion at a bottleneck queue, with the idea that end points 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 [20] changes the sign 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" [12]).

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 many places where queues may build up (on the cards, in IP-layer queues, near the USB interface in tethering mode, etc.), so one would need to deploy CoDel at all these locations, which could be difficult. That said, 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, which assumes that a single delay is right for all traffic.

Adaptive applications. In terms of application-layer adaptation, Alfalfa's new interface to video codec libraries enables finer-grained adaptation compared with other systems. Our use is an example of Application-Level Framing (ALF) [6], but unlike previous ALF-inspired applications, the adaptation is at a finer grain. Alfalfa does not just pick a suitable application data unit (ADU) from pre-determined choices, but instead constructs data frames dynamically using knowledge of previous receptions as well as the forecasted network conditions. The ALF principle has been used in a variety of media delivery protocols and architectural proposals, including RTP [24], MBone applications [5], the Congestion Manager API [1], etc. Much of the previous work on ALF-inspired protocols has focused on handling packet losses [23, 8], whereas in cellular networks the main problem is coping with rapid rate variations.

8 Limitations and Future Work

Although our results are encouraging, there are some limitations to our work. First, as noted in §2 and §3, an end-to-end system like Alfalfa cannot control delays when the bottleneck link includes competing traffic sharing the same queue. Even in a cellular network, if a device uses Alfalfa and TCP at the same time, the delays would be considerably worse—but we have not yet evaluated how Alfalfa "plays with" competing traffic that has access to the same bottleneck queue. We leave to future work a better understanding of the *need* for any form of active queue management to achieve good delay-throughput trade-offs, speculating here that a purely end-to-end approach may in fact work.

Second, we have not evaluated end-to-end video performance or demonstrated that Alfalfa's explicit-state interface to video codecs produces gains commensurate with its difficulty. Finally, Alfalfa does not currently include an audio codec and simply sends zeros that represent constant bit-rate audio; this will have to be completed before the application is useful.

9 Conclusion

This paper presented Alfalfa, a protocol for videoconferencing over Internet paths traversing cellular wireless networks. Current approaches perform significantly worse than Alfalfa over these networks because they do not provide the right balance between bit rate and self-inflicted delay when the available rate varies and packets experience variable latencies. At the heart of Alfalfa is Sprout, a predictive rate control protocol, which has two interesting ideas: the use of packet interarrivals as a congestion signal, and the use of Bayesian inference to make a forecast of the bottleneck link rate in the near future, which the sender uses to pace its transmissions. Our experiments show that forecasting is key to controlling delay, providing (for perhaps the first time, to our knowledge), an end-to-end rate control algorithm that is able to react at time scales shorter than a round-trip time. Alfalfa also implements a novel codec interface that allows a closer match between frame transmissions at the application and the available transmission rate from Sprout.

Our experiments conducted on traces from four commercial cellular networks show significant delay reductions and (often) bit rate gains over Skype, Facetime, and Hangout, as well as over CUBIC, Compound TCP, Vegas, LEDBAT, and CUBIC-over-CoDel.

REFERENCES

[1] H. Balakrishnan, H. S. Rahul, and S. Seshan. An integrated congestion management architecture for internet hosts. In *SIGCOMM*, 1999.

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

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

[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] S. Casner and S. Deering. First IETF Internet audiocast. *SIGCOMM CCR*, 22(3):92–97, July 1992.

[6] D. Clark and D. Tennenhouse. Architectural Considerations for a New Generation of Protocols. In *SIGCOMM*, 1990.

[7] 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.

[8] N. Feamster and H. Balakrishnan. Packet Loss Recovery for Streaming Video. In *12th International Packet Video Workshop*, Pittsburgh, PA, April 2002.

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

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

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

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

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

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

[15] 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.

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

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

[18] Low extra delay background transport. http://tools.ietf.org/html/draft-ietf-ledbat-congestion-09.

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

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

[21] J. Ott, S. Wenger, N. Sato, C. Burmeister, and J. Rey. Extended RTP Profile for Real-time Transport Control Protocol (RTCP)-Based Feedback (RTP/AVPF), 2006. IETF RFC 4585.

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

[23] C. Perkins, O. Hodson, and V. Hardman. A survey of packet loss recovery techniques for streaming audio. *Netwrk. Mag. of Global Internetwkg.*, 12(5):40–48, Sept. 1998.

[24] H. Schulzrinne, S. Casner, R. Frederick, and V. Jacobson. RTP: A Transport Protocol for Real-Time Applications, 2003. IETF RFC 3550.

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

[26] J. Valin, K. Vos, and T. Terriberry. Definition of the Opus Audio Codec, 2012. IETF RFC 6716.

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

[28] K. Winstein and H. Balakrishnan. Mosh: An Interactive Remote Shell for Mobile Clients. In *USENIX Annual Technical Conference*, Boston, MA, June 2012.