practice

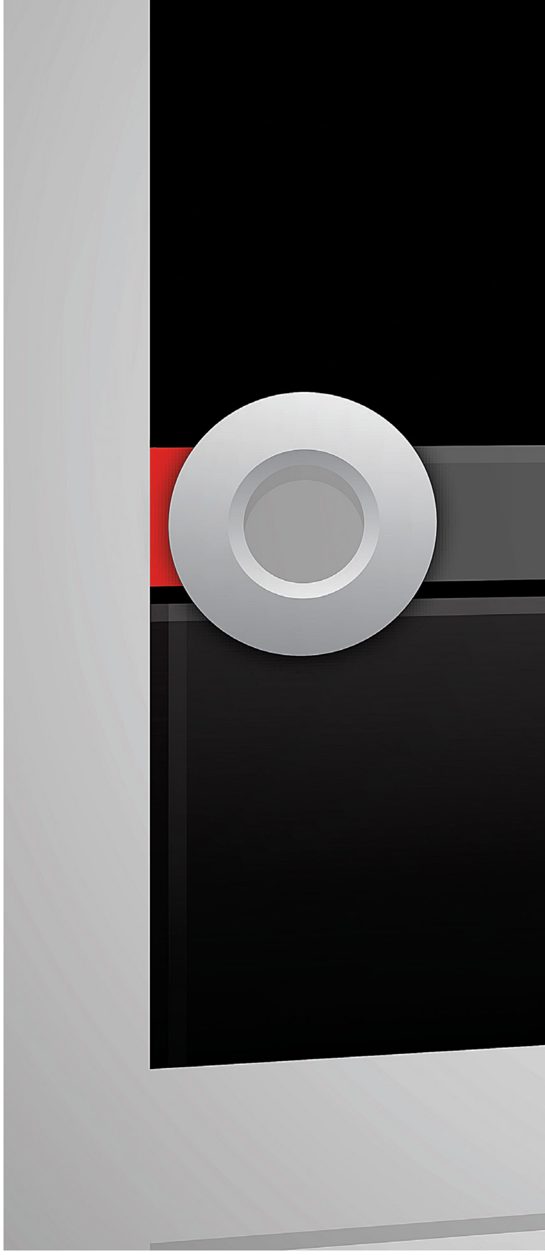
DOI:10.1145/2380656.2380671

Article development led by acmqueue
queue.acm.org

A proposal to improve the performance and availability of streaming video and other time-sensitive media.

BY AIMAN ERBAD AND CHARLES “BUCK” KRASIC

Sender-side Buffers and the Case for Multimedia Adaptation



THE INTERNET/WEB ARCHITECTURE has developed to the point where it is common for the most popular sites to operate at a virtually unlimited scale, and many sites now cater to hundreds of millions of unique users. Performance and availability are generally essential to attract and sustain such user-bases. As such, the network and server infrastructure plays a critical role in the fierce competition for users. Web pages should load in tens to a few hundred milliseconds at most. Similarly, sites strive to maintain *multiple nines* availability targets—for example, a site should be available to users 99.999% of the time over a one-year period.

Such stringent standards, however, are in practice carefully constructed to exclude problematic content such as video streaming and interactive virtual environments. If sites counted video-streaming rebuffer events against their availability, their multiple nines availability would vanish in a poof. Yet, one could argue that a video rebuffer just at the moment of the winning goal in the World Cup is a form of unavailability just as severe as, say, a failure that causes a social-network user to repeat some steps in composing a post. The goal of this article is to propose methods for adaptive transport that are ultimately aimed at bringing the performance and availability of streaming video and other time-sensitive media in line with that of traditional Web content. To do so we have developed an enhanced transport called Paceline.

IMAGE BY THOMAS PAJOT

As high-bandwidth low-latency multimedia applications such as videoconferencing, online games, and virtual reality applications become mainstream on the Internet, they will saturate the network, especially in mobile environments, much as other high-bandwidth (but high-latency) applications such as peer-to-peer file sharing do now. Videoconferencing, for example, is becoming a common service provided by social-networking sites (for example, Hangouts in Google). More than ever, real-time video streaming is becoming an integral component in many applications, such as live broadcasting of big events (for example, the Olympics, Super Bowl, and World Cup), distance learning, and on-demand video (Netflix). In addition, the global video-games industry was a $64 billion business in 2012, growing larger than the movie industry. Fast-paced large-scale games have high bandwidth requirements,[1] so they do not adhere to the old wisdom of network games having thin communication streams.

The combination of high bandwidth and low end-to-end latency is poorly supported in popular transports.[3,8] Paceline is an enhanced transport designed to support interactive, high-bandwidth applications. Contrary to conventional wisdom in multimedia transports, Paceline has not been implemented over User Datagram Protocol (UDP), nor does it propose changes to TCP. The deployment obstacles and duplication of effort faced by solutions that alter or replace TCP outweigh the challenges of mitigating its impairments. Instead, Paceline uses several innovative techniques to address TCP latency problems.

On top of TCP transport delays, large sender-side application buffers can accumulate and delay important data, exerting more influence over perceived quality. To have graceful failure modes for multimedia content in diverse environments (from gigabit broadband networks to congested wireless links), Paceline enables applications to adapt demands based on the available resources, and it favors important data. It supports quality adaptation based on the Priority-Progress model,[9] shown to be more stable in terms of packet delay and jitter for video streaming over TCP.[10]

End-to-End TCP Latency Analysis

Since interactivity and transport latency are a key focus here, let's look at the sources of TCP latency and set the context for Paceline. As depicted in Figure 1, end-to-end transport latency is commonly broken down into four components: *processing* delay, as a result of processing speed; *queuing* delays in nodes (hosts and network routers and switches); *transmission* delay resulting from the bit-rate of transmission; and *propagation* delays caused by physical distances. When one or more of those delays becomes large, interactivity (application-to-application message delivery) will suffer. Of these four latency components, *queuing delay* (inside TCP send buffers and network node queues) is the dominant cause of latency for high-bandwidth TCP applications. This is known as the bufferbloat problem.[7]

Processing delay is generally negligible because of fast CPUs and careful design of transport algorithms. Transmission delay will be bounded to

$$\mathtt{delay}_{\mathtt{transmit}} = \mathtt{mss/link\_rate}$$

assuming for the moment that ADUs (application data units) fit within transport segments up to an mss (maximum segment size). With common values of `link_rate` (Mbps or Gbps) and `mss` (for example, 1,500 byte), $\mathtt{delay}_{\mathtt{transmit}}$ will be a small value (for example, submillisecond). This leaves propagation delay and queuing delays as the dominant contributors to latency.

One-way propagation delay has lower bounds set by the laws of physics. Typical Internet path RTT (round-trip-time) values are in tens of milliseconds for intra-continental distances, or around 100 or 200 milliseconds for distances that cross oceans or traverse satellites. In addition, TCP provides reliability via retransmissions that can add extra queuing delay (multiples of the propagation delay) to the total. In the common case, however, TCP's fast retransmit mechanism should limit the retransmission-induced queuing delay to an RTT or two.

More importantly, TCP's socket buffer is often large enough that it can cause queuing delays in seconds. In many realistic conditions, the queuing delay specifically caused by the sender-side TCP socket buffer is the dominant portion of the total delay because of large kernel socket buffers employed by TCP implementations. For example, with a typical TCP send buffer size of 64KB, and a 300Kbps video stream, a full send buffer contributes 1,700ms of delay. To avoid unnecessary queuing delays, the kernel can be changed to dynamically tune the socket buffer size, bringing the end-to-end delay within two RTTs most of the time, while leaving TCP's congestion control unchanged.[8]

Paceline builds upon this idea, but is designed to avoid the need for kernel modifications. A user-level approach avoids the deployment obstacles of introducing new TCP implementations, deals gracefully with transparent proxies that can defeat an in-TCP-based approach, and allows a failover mechanism to reduce the worst-case latency when TCP becomes stalled in the case of back-to-back losses and retransmission timeouts.

Figure 1. Components of end-to-end latency.

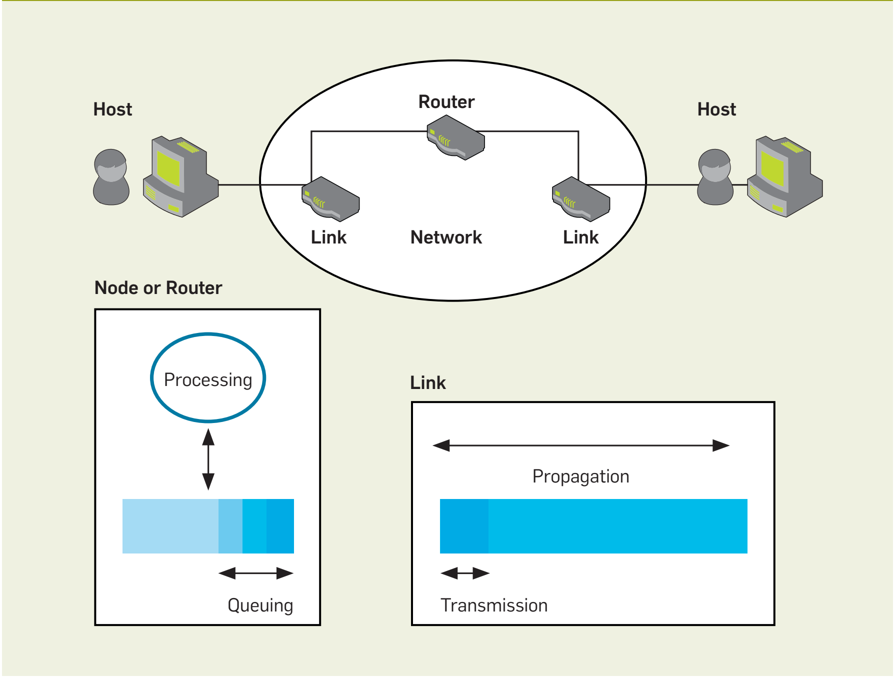
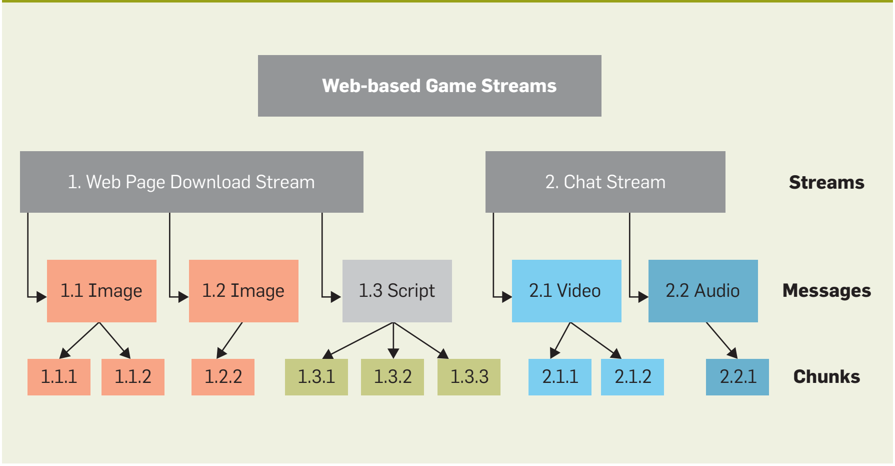


Figure 2. Streams of application data units.



Data Service Model: Not All Data Is Born Equal

In diverse environments, demands often exceed available bandwidth, leading to large sender-side queues. Queues can introduce head-of-line blocking (a delay that occurs when a line of packets is held up by the first packet) and hinder perceived quality if all the data items are treated equally and processed in a FIFO (first-in first-out) order. Minimizing the amount of data committed to TCP socket buffers reduces TCP sender-side queuing delays and pushes the sender-side buffers up the stack to the layer above TCP (Paceline in our design). Here, we describe the transport service model in Paceline with the necessary quality adaptation mechanisms to manage the sender-side buffers based on the Priority-Progress model.

Paceline provides a reliable message-based service model, chosen because low latency is a primary goal and messages provide a natural explicit means for the application to inform the transport about latency preferences, as well as representing an ADU. Paceline's programming interface allows the application to specify message importance on a per-message basis, and Paceline delivers messages in order of importance. The ability to queue messages ahead of time is essential to achieve high bandwidth, but the ability to prioritize messages is necessary to prevent head-of-line blocking between messages of different importance levels and the attending loss of responsiveness.

Unlike the byte-stream service model, Paceline allows the sender to cancel a pending message. This feature is motivated by the goal of responsiveness because the old data will slow down new messages and waste bandwidth. In conjunction with congestion control, cancellation is used by the application to adapt the rate of message delivery to the underlying network conditions. Informed cancellation maintains reliable delivery semantics while allowing applications to cancel stale messages. This provides an alternative to random dropping of messages (for example, UDP) under congestion. At the receiver, Paceline passes messages directly to the application. The application needs to handle out-of-order delivery and missing data introduced by message priority and cancellation.

In Figure 2, each ADU such as a video frame or a Web image uses a Paceline message. Each message is part of a full-duplex transport instance referred to as a *stream* (for example, video stream or Web document download stream). Similar to SPDY,[2] an application-layer transport protocol designed for minimal latency, Paceline supports multi-streaming, which multiplexes concurrent streams on top of a single TCP channel. Applications can perform the following operations on streams: creation, sending a message, canceling a message, and deletion. Streams are decoupled from the underlying TCP channels since all streams with the same host address and port number are multiplexed over the same TCP channel. A *channel* is the underlying communication primitive, identified by the host address and port number.

Like SPDY, Paceline efficiently multiplexes small Web transactions on top of a single TCP channel using short-lived streams (that is, those with few messages). More importantly, its innovative fairness design allows timely communication across concurrent long-lived streams such as videoconferencing or games. Different high-bandwidth streams can share a link and have varying requirements in terms of latency and high-level application-quality metrics (for example, frame rate). For example, a game transfers several kinds of streams such as player status updates, player video coordination chats, advertisements, and game control messages. The frequency of advertisements might be relaxed if necessary to help ensure player updates are sent promptly. Similarly, a distance-learning session can transfer voice, video, and slides from different users as separate streams multiplexed over the same TCP channel and can have different quality metrics.

To help illustrate Paceline's service model, Figure 3 contains a pseudo code example of the logic that an adaptive real-time application might use, in this case an adaptive videoconferencing client. The client calls the `send_video_frame` function to send a video-frame message. This function sends the message with an importance specified using an application-specific utility measure, reflecting the relative importance of individual frames to perceived quality. If congestion control restricts the rate of the stream, the client will cancel messages of low importance when their utility has expired, while messages of high importance will be sent. For messages of equal importance, Paceline breaks the tie according to position. Paceline's service model provides a clean interface for rate adaptation to match application demands with network conditions, instead of committing messages to the network transport (that is, socket buffer) and then suffering from transport queuing delays.

To benefit from Paceline's data-service model, applications have to develop domain-specific adaptation policies. High-definition videoconferencing, for example, has two dimensions for adaptation: spatial and temporal quality. For each ADU, the application calculates an importance value to estimate its contribution to video quality. Each ADU represents a video layer and uses a Paceline message.

In addition to these two quality dimensions, higher-level indicators can be incorporated, such as the active tab,

Figure 3. Adaptive videoconferencing client.

```
send_video_frame (player, stream, frame) {
        /* Set message data and length */
        msg_init.data             = frame.data;
        msg_init.length           = frame.data_len;

        /* Set message importance */
        msg_init.importance       = get_importance(frame);
        msg_init.virtual_time     = get_virtual_time(frame);
        msg_init.sent             = video_frame_sent;

        /* Sending a frame with cancellation */
        stream.msg_create(msg_init, &frame.msg_handle);
        stream.msg_write(frame.msg_handle);
        frame.expire_event = expire_video_frame;
        add_timer(frame.deadline,
                frame.expire_event);
}
expire_video_frame (frame, stream) {
        stream.msg_cancel(frame.msg_handle);
}
video_frame_sent (player, frame) {
        cancel_timer(player,frame.expire_event);
}
```

mouse clicks, and position of scroll bar to derive the adaptation policies. Similarly, FPS (first-person shooter) games have limited upload bandwidth to send frequent updates to all game players, especially in epic fights with a large number of players concentrated in one area. Games can reduce the bandwidth requirements using criteria such as proximity, recency, and aim, because players are most likely interested in nearby players or those with whom they have recently interacted.

Paceline is implemented as a user-level library and is layered above standard TCP implementations. As depicted in Figure 4, Paceline's architecture consists of two layers: stream and channel. The *stream layer* manages the application-message queue in Paceline and ensures low latency for data with more influence over quality by enabling adaptation between messages in one stream and across streams. This layer consists of two subsystems: the message framing and fragmentation; and the stream fairness. The channel layer handles the TCP low-level sender-side delays. It consists of the latency controller and the connection manager.

Framing and Fragmentation: Which Chunk to Send?

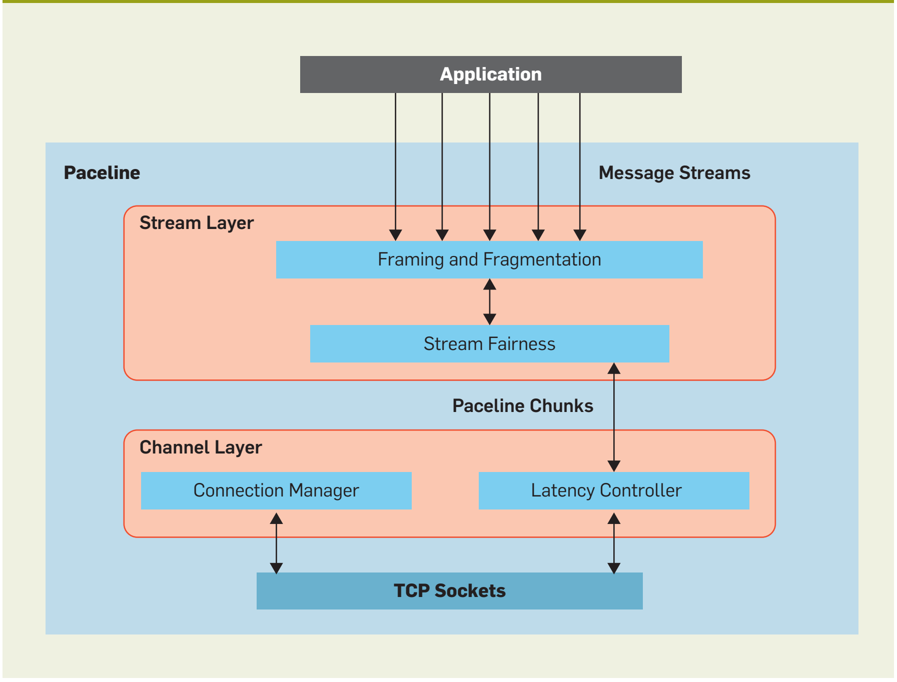
Fragmentation is the first of several techniques that improve transport latency. Paceline allows application-level messages of arbitrary size. To decouple transmission delay of potentially large application messages from lower-level queuing delays, the data-transfer mechanism supports sender-side fragmentation of application messages into Paceline chunks, and receiver-side reassembly of chunks back into the original application messages. Chunks are bounded to a small size, typically a fraction of TCP's maximum segment size.

Paceline includes application-level message queues. Unlike lower-level queues that operate in FIFO order, Paceline's message queues are based on priority, so that chunks of newly arrived important messages may quickly preempt older less-important ones. Therefore, chunks of messages with high importance are released to the network faster and observe minimal queuing inside Paceline, as well as minimum application-level transmission delay. Cancellation allows the application to abort a low-importance message if its overall transmission delay is too large.

Stream Fairness: From Which Stream?

Paceline messages (and chunks) are part of a full-duplex stream. Each stream in Paceline has a separate priority queue with chunks ready to be sent.

Figure 4. Paceline architecture.



For fair and timely communication across concurrent streams, Paceline supports two notions of fairness: quality and resource fairness. Resource fairness guarantees fair bandwidth across streams, while quality fairness ensures fair application-level quality. Quality is specified in generic terms but derived from the application level; examples are the frames per second in videoconferencing or the updates per second in online games.

While FIFO or round-robin policies are simple ways of multiplexing data of different streams over the underlying channel, timeliness necessitates a better notion of fairness among concurrent streams, especially when bandwidth is limited. Paceline implemented a fair sharing policy among *active* streams that has data inspired by weighted fair queuing. Each stream has a cumulative *virtual time*, an increasing counter quantifying the resources a stream (messages) has used since it was created. Active streams are organized in order of their virtual time, and chunks are sent from the stream with the minimum virtual time. The important factor regulating how streams are multiplexed is how their virtual times are initialized and adjusted.

Virtual time initialization is based on two rules:

▸ **Rule 1 (Fair Start).** When a stream is created its virtual time is set to the minimum virtual time of all active streams to ensure that existing streams do not starve until the new stream catches up in virtual time. If no active streams exist, the virtual time of the newly entered stream is set to the maximum time of all idle streams, or zero if this is the only stream.

▸ **Rule 2 (Use It Or Lose It).** If a stream becomes active after being idle, the stream's virtual time is set to the maximum of its virtual time and the minimum virtual time of all active streams. This guarantees that no stream, while idle, can save its share of resources for future use.

When a stream transmits a message over the underlying channel, its virtual time is updated according to the virtual times of messages sent, which are based on the application fairness criteria. Conventional resource fairness increments the virtual time based on the size of each message transmitted by the stream. On the other hand, quality fairness increments the virtual time based on an indicator of the application quality of experience, such as the frame rate. By scaling virtual times of streams with different factors, Paceline can allocate different shares to different streams, providing weighted fair sharing.

Latency Controller: How Many Chunks to Send?

To give applications more agility in adapting data delivery, Paceline reduces the amount of committed data in TCP's outgoing buffer and keeps data in its own message queues. The latency controller monitors the progress of the underlying TCP flow and regulates the rate of application data (chunks) delivered to the sender-side TCP. The goal of this controller is to send chunks into TCP fast enough to allow the congestion control to claim the flow's fair share of available bandwidth, but not so fast as to cause an unnecessary amount of FIFO queuing to accumulate in TCP's outgoing socket buffer.

Paceline's controller regulates the writing of application data to TCP in a way that dynamically matches the buffer fill level to a value close to the size of TCP's congestion window (`cwnd`)—namely, `cwnd+3×MSS`. This design implements at the user level the same strategy that was implemented inside the kernel in a previous study and was shown to strike the best balance between latency and throughput.[8]

Paceline has two distinct schemes to estimate the `cwnd`: kernel-assisted and the purely user-level approach. Each has specific advantages. The kernel-assisted scheme, called the PaceK controller, uses information directly from the kernel TCP via the socket API. While this scheme is simple and effective, it requires information that only some implementations of TCP make available. Thus, the PaceK controller is not fully portable. Also, transparent proxies in the network path would likely defeat the PaceK controller's ability to regulate queuing delay, as the TCP socket buffers in the proxies operate independently and can easily become points of major queuing delay if they precede the path bottleneck.

The purely user-level controller in Paceline is called PaceA. Unlike the PaceK controller, it uses only the common TCP socket API available on all major operating systems. Thus, PaceA is more portable (no need to extend or modify kernels), easier to deploy (for example, in relation to firewalls), and avoids problems that result from intermediate proxies. At the user level the value of `cwnd` is not available so the primary goal of PaceA is to derive `cwnd`', an estimate of TCP's `cwnd`. PaceA uses application-level acknowledgments (P-ACKs) to measure latency and bandwidth and estimate `cwnd`' as the `latency × bandwidth`. More information about the design of PaceA is available elsewhere.[5]

Connection Manager: Which TCP Socket to Use?

In Paceline, the latency controller is the basic technique for limiting TCP latency. In our experiments, however, the distribution of latencies across messages retained a prominent tail, and there was a wide gap (for example, more than a factor of eight) between median and worst-case latencies. We diagnosed the worst-case latencies through a combination of instrumentation in Paceline and packet-trace analysis. Under heavy congestion, TCP can experience back-to-back losses leading to one or more retransmission timeouts. Our diagnosis confirmed that the worst-case latencies were correlated with such episodes of exponential back-off. Similar problems are observed when testing Paceline over wireless links with poor signal strength. To reduce their impact and introduce a ceiling on worst-case performance, Paceline includes a failover mechanism to supplement its basic latency-limiting mechanisms.

Paceline's failover is analogous to the scenario where the user presses the stop/reload buttons in a Web browser upon encountering slow response. Automated failover may sound quite radical, but our evaluation shows that our implementation achieves significant reductions in worst-case latencies while preserving bandwidth fairness. Automated failover resembles removing exponential back-off from TCP, shown to be safe in previous work.

The connection manager maintains a number of back-up TCP sockets and implements failover in a manner that is fully transparent to the application. We switch to a new TCP socket when a threshold is reached. The threshold setting is subject to a trade-off between latency and fairness since replacement channels start in TCP slow-start, possibly resulting in underutilization of the network. The failover threshold is set dynamically using an equation that resembles TCP's RTO (retransmission timeout). A safety margin was added to the failover factor to reduce the number of false positives caused by noise in measurements outside the kernel.

We evaluated Paceline experimentally within a network-emulation testbed. Our measurements are compared against two points of reference: TCP, used to quantify the improvements resulting from Paceline; and SST (Structured Stream Transport),[6] implemented over UDP with no transport buffer queuing delays. SST provides a rich service model including reliable messaging and congestion control, and it includes the full range of capabilities one might expect from any realistic clean-slate replacement for TCP. Thus, we use SST to approximate a best-case reference point against which to compare Paceline.

In TCP mode, our application still uses the service API of Paceline, but the latency controller is disabled; hence, we send data via TCP as fast as it will allow.

Our network set-up uses the common dumbbell topology, where a set of servers on a LAN connect through a single-bandwidth-delay-constrained link, emulating a congested WAN intra-continental path, to a set of clients on a remote LAN. For the WAN path, we emulate a 30ms RTT delay with a bandwidth limit of 16Mbps or 12Mbps. The WAN bottleneck uses drop-tail queuing with a queue size of twice the bandwidth-delay product of the WAN adding 60ms when the bottleneck becomes congested. The experiments are set up to reflect rather harsh congested conditions, where the bottleneck WAN link is persistently saturated. This is the range in which TCP's performance leaves a lot to be desired.

Transport Performance Summary

Paceline improves upon TCP's weaknesses and maintains its strengths.

Figure 5. Latency threshold versus temporal quality.

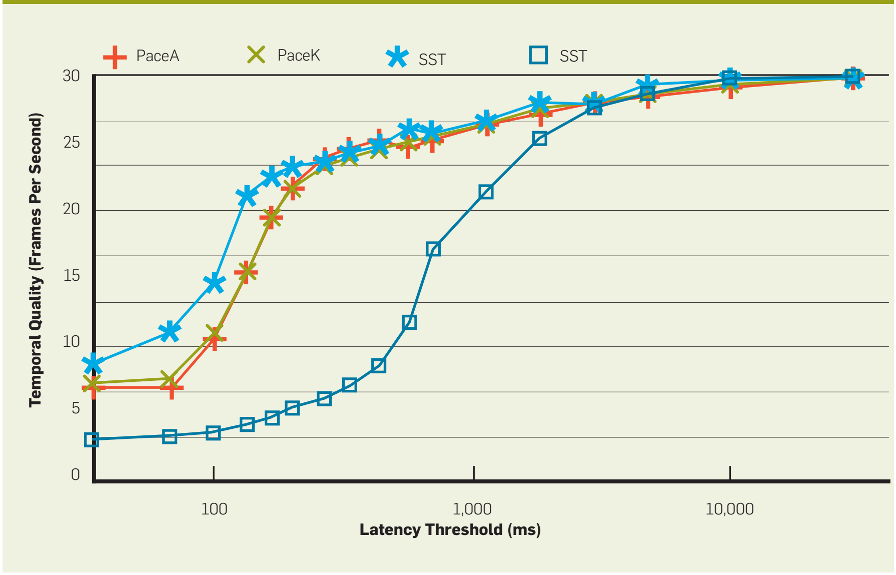
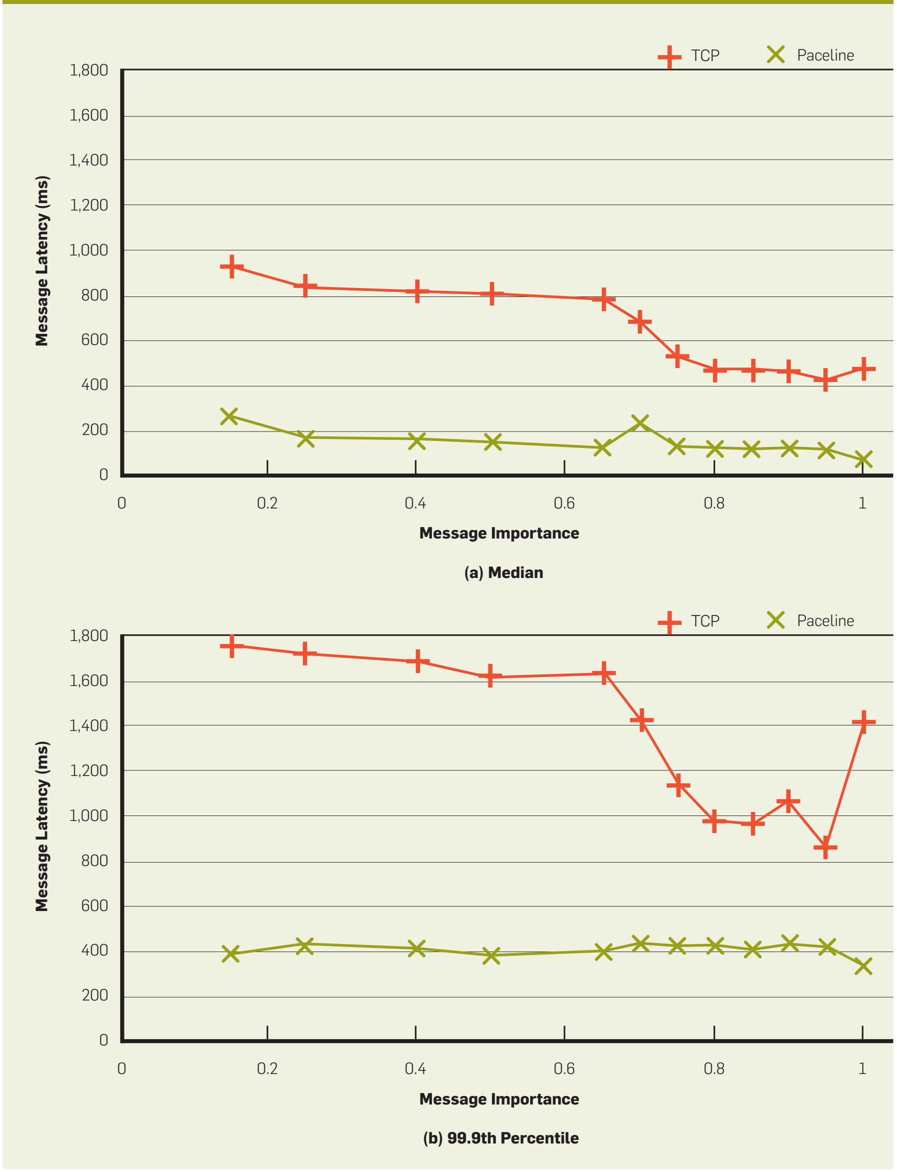


Figure 6. End-to-end message latency based on message importance.



The low-level transport performance compares the transport latency, utilization, and fairness of Paceline with that of TCP. Paceline reduces the median, the 99.9%, and worst-case end-to-end latency by a factor of three to four times. On the other hand, Paceline has similar bandwidth fairness to TCP, high network utilization, and reasonable wire overhead. Paceline is incrementally deployable on the Internet since it shares bandwidth fairly with TCP flows while retaining all latency improvements. (For detailed transport performance, see Erbad et al.[5])

Application-Level Performance: Does Adaptation Work?

Here, we evaluate the performance of adaptive applications in terms of application-level quality metrics. The evaluation sheds light on the trade-off between average quality and interactivity, and then shows the message latency in Paceline with respect to assigned importance.

Quality and interactivity trade-off. One of the main issues to consider is the nature of the trade-off between overall multimedia quality and interactivity—better interactivity (lower latency) generally comes at the expense of video quality (for example, spatial detail). The following experiments fix the number of flows to eight videos (4Mbps each, extremely congested link) but vary the level of interactivity using a configuration parameter of the *latency threshold* (the amount of time each ADU is given before it expires and gets canceled by the sender) on outgoing messages.

To quantify video performance close to the level of user experience, we measure the frame rate in fps (frames per second). Adaptation in the videoconferencing test application prioritizes ADUs according to two dimensions of video quality: temporal quality (frame rate) and spatial quality, measured with PSNR (peak signal-to-noise ratio) of frames. The video format is scalable so each video frame consists of eight ADUs with one base spatial-layer ADU and seven (progressive) enhancement spatial-layer ADUs. The default adaptation policy is biased toward temporal quality. That is, as the bit rate of a video stream drops, spatial enhancement

ADUs are dropped; and when the spatial quality nears minimum, then further reductions in bit rate will cause dropping of base ADUs, which will result in dropping entire frames (lower temporal quality). Therefore, in the congested settings used for testing, the temporal quality (that is, frame rate) is the quality measure.

Figure 5 shows the average frame rate as the latency threshold is varied. Notice that on the rightmost side of the graph with the highest latency thresholds (tens of seconds), all transports achieve full temporal quality of the video (30fps). The temporal quality when using TCP drops much more rapidly moving leftward (with lower latency thresholds). Even though TCP delivers high throughput, the high transport latency with TCP causes frequent head-of-line delays blocking between low-importance ADUs (spatial enhancements) and high-importance ADUs (base layers). This translates to dropped frames and a much lower fps rate.

The trends exhibited by SST and Paceline are very similar. Recall that SST's implementation completely avoids transport queuing delays. Comparing temporal qualities of Paceline and SST, we see that Paceline also eliminates most TCP sender-side queuing delays. The knees of the Paceline and SST curves in the 100ms–200ms zone indicate that even in this heavily congested network, it is possible for an application such as videoconferencing to keep within the zone of reasonable interactivity with a modest impact on quality. On the other hand, using TCP as the transport results in quality not increasing substantially until well over the 500ms point, which is probably not acceptable for comfortable interaction.

Importance effects on latency. Up to this point, Paceline was shown to be within the zone of responsiveness similar to clean-slate protocols such as SST. This section investigates the effects of importance on message latency. Messages are spread into buckets according to their importance, and the one-way end-to-end latency of the delivered messages is measured in each bucket. Figure 6a presents the median latency, while Figure 6b is the 99.9th percentile latency.

As shown in Figure 6, both TCP (with adaptation) and Paceline have lower median and worst-case latency for important data, with an improvement of more than a factor of two over less-important data. Since TCP commits messages in the kernel send buffer, TCP flows have higher overall latency with a median that is well above the expected latency (275ms). Paceline, on the other hand, keeps the median latency very close to the one-way delay (75ms) for more important data. Paceline also has consistent 99.9th percentile latency due to failover, which is close to 400ms for all messages. The 99.9th percentile latency in TCP is above a second for the majority of messages, reaching almost 1.8 seconds in some cases.

We evaluated quality fairness across streams in Paceline. Video quality is defined by the temporal quality (fps). Figure 7 plots the frame rate of three videos over time. The videos (transferred over three streams) display with identical quality (in terms of frame rates) that changes based on network conditions. It is interesting to note that streams were allocated different bandwidth shares in the same period to achieve equal quality.

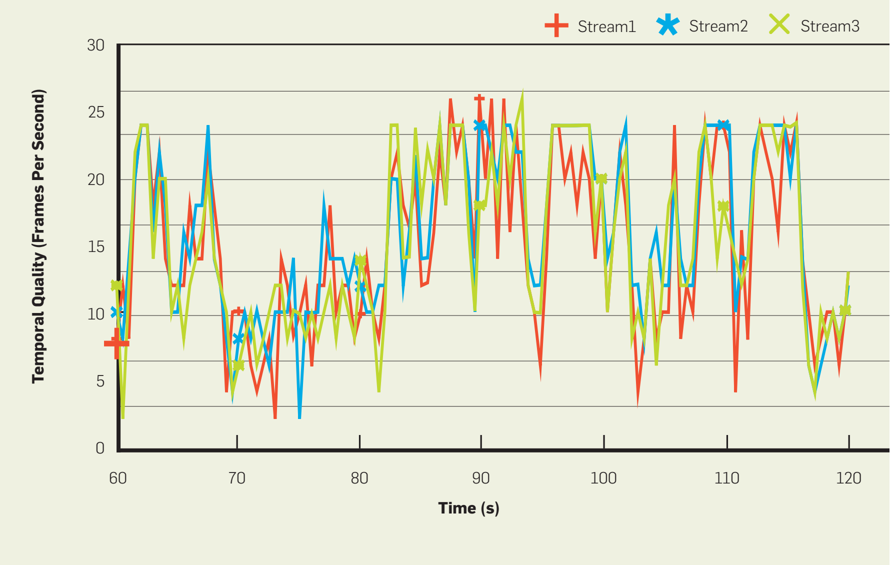
Quality fairness in the Paceline model is completely controlled by application-quality metrics. We provide applications with the notion of importance to control adaptation within streams. Weights and virtual time, on the other hand, specify importance across stream boundaries.

Limitations and Future Work

The Paceline implementation is written in C, and we have been mindful of performance and efficiency from the start. Using QStream, a complete end-to-end implementation of adaptive video streaming, was helpful because the application provided visual and quantitative feedback directly connected to each performance change. Later Paceline was used in research on massive scale gaming, which revealed performance weaknesses not apparent in the video setting. Prominent among these, certain elements of game traffic (state updates) involve very high volumes of small messages, and keeping processing overhead down in this setting is a challenge, particularly in terms of taxing dynamic memory allocators. We have a design to reduce the Paceline memory allocation to at most one per application-level message.

Current transport protocols such as SPDY embrace all-SSL-all-the-time methodology, partly motivated by security but also motivated to mitigate myriad problems caused by middle boxes that are intolerant (intentionally and not) to new protocols. In hindsight, it would have been useful to think of SSL integration from the early stages of Paceline's design. Paceline can perform SSL negotiation at the channel level, amortizing the cost of the initial negotiation. We also need to ensure that encryption happens only when messages are written to the socket in order to avoid canceling encrypted data.

Figure 7. Quality fairness policy: Temporal quality



The Minion architecture of Nowlan et al.[12] has several features that would be beneficial and complementary to Paceline's strengths, such as its SSL strategy and wire-compatible changes to TCP that allow unordered receive.

Paceline reduces sender-side queuing delay by writing the minimum amount to the TCP socket buffer. Recently, the bufferbloat problem has received greatly renewed general attention and interest. The recent algorithm work by Kathleen Nichols and Van Jacobson is a promising step since it is easy to implement and needs no configuration.[11] What is still missing, however, is true end-to-end evaluation that quantifies the total combined effect of refinements to TCP, ECN (Explicit Congestion Notification), AQM (Active Queue Management), and adaptive multimedia.

Underlying our interest is what we believe remains an open question: Is there really a line between applications that needs to break from TCP for interactivity reasons? This is a line that has been moving steadily over time.

Lessons Learned

Introducing a new transport layer is challenging because the designer must address several critical issues. First, the transport layer has to improve performance significantly for the target applications in order to justify the extra effort. Second, the enhancements should not negatively affect other traffic types; we adhere to the vision that the Internet should remain a general-purpose infrastructure for a vast array of applications. Finally, the performance improvements at the transport level in terms of latency, fairness, and utilization need to translate to quality improvements at the application level. Paceline has shown positive results in all these aspects.

We tested Paceline extensively using videoconferencing scenarios over WAN settings. Paceline was also used in a small cloud-based game prototype to scale the wide area communication of an Epic-scale game scenario.[13] Both of these applications show significant improvements in multimedia quality because of the reductions in TCP sender-side delays and the use of adaptation in Paceline.

To ensure we have a general-purpose transport supporting a wide range of applications, we tested using a network traffic generator simulating different HTTP Web traffic flavors (for example, HTTP1.0 and HTTP1.1, with and without pipelining). The experiment tested Web document downloads while varying the number of objects per page and the page size. Paceline improved the bandwidth utilization using the multistreaming feature as we increased the number of objects as a result of the automatic pipelining of small transactions on top of the underlying channel.

We evaluated Paceline using transport and application-quality metrics. Transport-level metrics were necessary during the early stages of developing Paceline algorithms, such as the latency rate controller and failover criteria. It was essential at that stage to ensure Paceline did improve latency without reducing fairness and network utilization. As the transport became more mature, however, we needed to ensure these low-level improvements translated into quality improvements for video and other multimedia applications. Verifying application-quality improvements is the limitation of the majority of newly proposed transports.

Finally, effective use of the network required application-level knowledge of quality and importance measures, along with careful tracking of messages to avoid wasting bandwidth. Quality adaptation is an essential transport feature. Paceline enables applications to scale quality with the available bandwidth and to favor data with more influence over quality. Adaptive multimedia applications can provide graceful failure modes with different quality levels instead of rebuffering when bandwidth is limited.

The Paceline implementation is part of the QStream video-streaming system, which is open source and may be downloaded from http://qstream.org. More implementation and evaluation details can be found in an earlier article on the topic.[4]

Acknowledgments

Thanks to colleagues at the Network, Systems, and Security lab at the University of British Columbia for their feedback on previous drafts of the article. Special thanks to Mahdi Tayarani Najaran for his significant contributions to the design and implementation of Paceline. Finally, thanks to Terry Coatta for helping to improve both the style and content of this article. C

Related articles on queue.acm.org

Four Billion Little Brothers?: Privacy, mobile phones, and ubiquitous data collection
Katie Shilton
http://queue.acm.org/detail.cfm?id=1597790

VoIP: What is it Good for?
Sudhir R. Ahuja and Robert En
http://queue.acm.org/detail.cfm?id=1028897

Data in Flight
Julian Hyde
http://queue.acm.org/detail.cfm?id=1667562

References

1. Bharambe, A., Douceur, J. R., Lorch, J.R., Moscibroda, T., Pang, J., Seshan, S. and Zhuang, X. Donnybrook: Enabling large-scale, high-speed, peer-to-peer games. In *Proceedings of the ACM SIGCOMM Conference on Data Communication*, (2008), 389–400.
2. The Chromium Projects. SPDY: An experimental protocol for a faster Web, (2011); http://www.chromium.org/spdy/spdy-whitepaper.
3. Erbad, A. Real-time support for interactive multimedia applications (2012); http://hdl.handle.net/2429/42878.
4. Erbad, A., Hutchinson, N.C. and Krasic, C. DOHA: Scalable real-time Web applications through adaptive concurrent execution. In *Proceedings of the 21st International Conference on World Wide Web*, (2012), 161–170.
5. Erbad, A., Tayarani Najaran, M. and Krasic, C. Paceline: latency management through adaptive output. In *Proceedings of the First Annual ACM SIGMM Conference on Multimedia Systems*, (2010), 181–192.
6. Ford, B. Structured streams: a new transport abstraction. In *Proceedings of the Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, (2007), 361–372.
7. Gettys, J. and Nichols, K. Bufferbloat: Dark buffers in the Internet. *Queue 9*, 11 (2011), 40:40–40:54.
8. Goel, A., Krasic, C. and Walpole, J. Low-latency adaptive streaming over TCP. *ACM Transactions on Multimedia Computing, Communications, and Applications 4*, 3 (2008), 1–20.
9. Krasic, C. A framework for quality-adaptive media streaming: Encode once–stream anywhere. Ph.D. thesis. AAI3119036 (2004).
10. Kuschnig, R., Kofler, I. and Hellwagner, H. An evaluation of TCP-based rate-control algorithms for adaptive Internet streaming of H.264/SVC. In *Proceedings of the First Annual ACM SIGMM Conference on Multimedia Systems*, (2010), 157–168.
11. Nichols, K. and Jacobson, V. Controlling queue delay. *Queue 10*, 5 (2012), 20:20–20:34.
12. Nowlan, M., Tiwari, N., Iyengar, J., Amin, S. and Ford, B. Fitting square pegs through round pipes: unordered delivery wire compatible with TCP and TLS. In *Proceedings of the 9th Conference on Networked Systems Design and Implementation*, (2012).
13. Tayarani Najaran, M. and Krasic, C. Scaling online games with adaptive interest management in the cloud. In *Proceedings of the 9th Annual Workshop on Network and Systems Support for Games*, (2010), 9:1–9:6.

Aiman Erbad recently joined the computer science and engineering department at Qatar University as an assistant professor after completing his Ph.D. at the University of British Columbia. His research interests include Web architecture, networking, real-time multimedia, ubiquitous computing, and concurrency support.

Charles "Buck" Krasic works on large-scale processing of video at Google's YouTube. Prior to joining YouTube, he was a professor in the department of computer science at the University of British Columbia. His research led to the development of QStream. Other research interests include multimedia, operating systems, networking, and distributed systems.