Cross-Layer Pacing for Predictably Low Latency

Andreas Schmidt*, Stefan Reif†, Pablo Gil Pereira*, Timo Hönig†,
Thorsten Herfet* Wolfgang Schröder-Preikschat†
*Saarland Informatics Campus {andreas.schmidt,gilpereira,herfet}@cs.uni-saarland.de
†Friedrich-Alexander University Erlangen-Nürnberg {reif,thoenig,wosch}@cs.fau.de

***Abstract*—Low-latency networking has become a major research domain as to the increasing demand for wireless applications. A predictably low *age of information* is key to such applications (i.e., augmented reality, smart factories, and edge computing infrastructures) as they need to operate on the latest data. Despite variable channel properties (i.e., non-deterministic interferences) it must be ensured that especially end-to-end communication operates with lowest latency that is possible.**

To enable low-latency networking we propose X-PACE, a cross-layer pacing scheme. X-PACE provides low-latency communication under consideration of dynamics from the network protocol layers up to the application level. Bottleneck detection and mitigation techniques of X-PACE provide predictability and improve the end-to-end communication latency. Our evaluation demonstrates that X-PACE reduces the end-to-end tail latency by up to 54 % and narrows the latency range by up to 91 %.

***Index Terms*—cross-layer optimization, low-latency networking, pacing, transport protocols**

I. INTRODUCTION

The ever increasing demand for mobile and wireless devices is triggered by a variety of novel application scenarios that accompany our everyday life. Fields of application such as augmented reality, virtual reality, and the construction of smart factories adapt wireless technologies that build upon cyber-physical systems and edge computing infrastructures [1]. At this, wireless network infrastructures are challenged to provide the necessary quality criteria, for example, reliable communication links in general and low-latency communication in particular. The latter is of special importance, as a low age of information is necessary for applications to operate on the latest data. The need for information freshness leads to the *motivation* of our work: applications need reliable low-latency communication systems that operate on channels that have unknown properties and that may change over time. Furthermore, such communication systems must be aware that mobile and wireless devices are resource-constrained in terms of processing capabilities and energy supply.

To achieve reliable low-latency communication over dynamic channels, we propose X-PACE, a cross-layer pacing scheme that spans from the lowest network protocol layers up to the application. It detects bottleneck components and defuses their impact on the end-to-end communication latency. This bottleneck can be in the network, but it can also be within a processing node due to limited computation capabilities or energy constraints. The benefits of X-PACE are the following:

The work is supported by the German Research Foundation (DFG) within SPP 1914 under grants HE 2584/4-1 and SCHR 603/15-1.

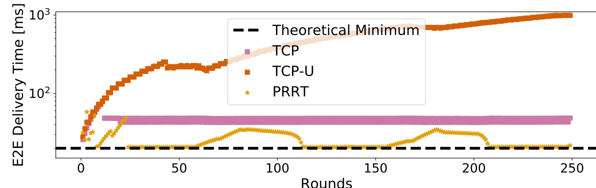


Fig. 1: X-PACE (in PRRT) significantly reduces the end-to-end latency at the application level compared to default TCP (TCP-U) and fine-tuned low-latency TCP

First, queueing delay is near zero when all processing steps are synchronized to the bottleneck rate. In consequence, units of work are not stalled in buffers. Second, it supports preparation tasks (e.g. memory preallocation) to aggressively minimize processing latencies. Third, fewer resources are required when the system adapts to the optimal processing speed. Knowing the rate at which the transport stack processes and transmits data allows the operating system and the physical computation platform to reduce clock-cycles (e.g. no polling) and to utilize low-power performance and sleep states of processors more efficiently. The consequence is a reduction in energy demand, which prolongs the lifetime of battery-driven devices.

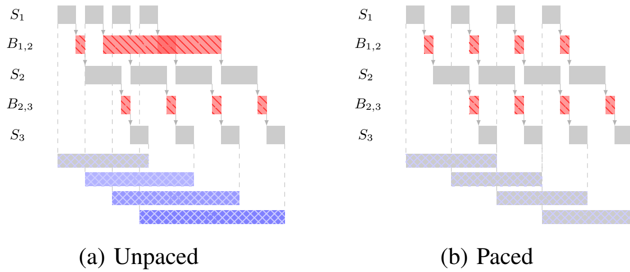
The *contributions* of this paper are the design and implementation of X-PACE. We demonstrate significant improvements in communication latencies and predictability, compared to the state of the art: in our empirical evaluation, X-PACE operates close to the network limit, and decreases the 99th percentile of the end-to-end latency at application level by 25 % (c.f. Fig. 1, 36.43 ms vs. 48.49 ms). We also present scenarios (cf. Sec. V-B2) in which even higher reductions of up to 54 % are achieved.

II. BACKGROUND AND RELATED WORK

X-PACE aims at improving the end-to-end (E2E) latency at the application level which is defined as:

$$D_{E2E} = D_{prop} + D_{trans} + D_{proc} + D_{queue} \qquad (1)$$

This paper considers the propagation delay (D_{prop}) as a small static figure, which is common in networked cyber-physical systems. *Edge computing* [1] and similar approaches to reduce D_{prop} are complementary to our solution. The processing delay (D_{proc}) is caused by the operating system and application code, which can be reduced, for instance, by kernel bypassing [2]. The transmission delay (D_{trans}) is determined by a static payload size (for our use-case) and a varying data rate.



(a) Unpaced (b) Paced

Fig. 2: Pacing keeps the bottleneck throughput, but avoids excessive buffering latencies (marked as red striped) and increased E2E latencies (marked as blue hatched)

The major interest of this paper is in the queueing delay (D_{queue}), which is also the most harmful for low-latency applications. This delay quantifies the time spent by packets in queues along the path between application processes, which can be in-network queues, but also within system software and hardware, where increasing complexity causes an increase in layers with (often unmanaged) buffers [3]. Empirical evaluations show that "bufferbloat" can easily cause a round-trip time of a residential internet access to increase from 10 ms to 1 s or more. This is due to queues being large in order to compensate for bursty work loads. The root cause of bufferbloat is that applications are not aware of buffer sizes and the throughput provided by the different steps of the processing chain. The former requires coordinated management strategies (e.g. *active queue management*), the latter requires pacing.

Pacing has been intensively studied by the networking community with respect to TCP ([4], [5], [6]) and aims to evenly spread the transmission of a set of packets across the entire round trip time. Fig. 2 shows the processing of four work units through steps S_1 to S_3, where step S_2 is the bottleneck. Without pacing (cf. Fig. 2a), queues form at the bottleneck while the throughput stays the same. In contrast, pacing the first step to the pace of the bottleneck helps to keep the buffer empty (cf. Fig. 2b). By considering the bottleneck, both step S_1 and S_3 can slow down, to save resources.

III. Cross-Layer Pacing

Our proposed approach towards *cross-layer pacing* (X-Pace) exploits the TCP pacing idea and applies it to all system layers (i.e., network stack, operating system, and applications).

A. Pacing

Pacing is essential for networked systems to achieve predictably low latency by not filling buffers excessively [3] and avoiding harmful load spikes caused by operating system tasks [7]. In the following, we consider a *pace* ($P, [P] =$ second) as the time required to apply a certain step to a certain unit of data, e.g. encoding time per packet. A system is a processing and communication chain with n steps and associated paces $P^{(i)}$ ($i \in [0 : n-1]$). The *pace* size behaves similar to energy demand, i.e. lower is better (fast pace = low value, slow paces = high value). A system implements *pacing* if it ensures that each step is executed at a pace that considers the bottleneck pace $P^{(btl)}$ in the overall chain of processing steps. A system is *paced* if each step needs less (or equal) time to process a unit of work than a previous step.

B. Measuring and Communicating Paces

For synchronizing the paces across the different layers, the paces of all layers must be measured precisely. X-Pace measures the paces at run-time so that the executing platform is considered as well as disturbances caused by changes in system load or cross-traffic. As paces of each step differ by nature, e.g. system noise and fluctuations of network performance, portions of the pace caused by waiting for a pre- or succeeding step must be treated separately and filtered appropriately. Thereby, we calculate an *effective pace* (delay caused by the actions of the processing step itself – subtracting waiting times) and a *total pace* (the delay between two invocations of the processing step). Without this compensation, the system would not converge to a stable pace.

These measured paces must now be communicated throughout the system to allow adaptation. In any system, there exists at least one bottleneck pace $P^{(btl)}$ so that $\forall j \in [0 : n-1] : P^{(btl)} \geq P^{(j)}$, which is associated with the bottleneck step $S^{(btl)}$. If there are multiple bottlenecks, the last one (the one with the highest index) is considered in the following. Starting from the bottleneck, this information must be communicated in both directions (cf. the concrete implementation in Fig. 3).

First, it must be communicated to $S^{(btl-1)}$ (backward propagation), stating that data *cannot be processed* as fast as it could be provided. For $S^{(btl-1)}$, this means that it must run at a slower pace and take longer to complete the tasks. Consequently, $S^{(btl-1)}$ performs at an effective pace $P^{(btl-1)}$, which must in turn be communicated to $S^{(btl-2)}$.

Second, it may be communicated to $S^{(btl+1)}$ (forward propagation), stating that data *cannot be provided* as fast as it could be processed. For $S^{(btl+1)}$, this means that it can (a) run at a lower pace and completing tasks may take longer or (b) decide to keep its pace because finishing "fast" has benefits (e.g. low resources footprint or minimal E2E delivery time). For this reason, it holds that $P^{(btl+1)} \in [0 : P^{(btl)}]$, which must be propagated forward, so that other steps could also decide to take longer. This allows to optimize these steps with respect to other goals than throughput, e.g. energy demand.

In order to implement X-Pace, the measured and communicated paces must be considered to slow down specific steps. How to achieve this depends on the step itself, with respect to its ability to change its speed (some steps might always run as fast as possible) and the potential gains by this change (e.g. saving energy or buffer space).

IV. Implementation

To demonstrate the effectiveness of our proposed approach we have implemented X-Pace. The first part of this section describes the protocol to which we have added cross-layer pacing and the second part describes the specific adaptations of the transport protocol for cross-layer pacing in detail.

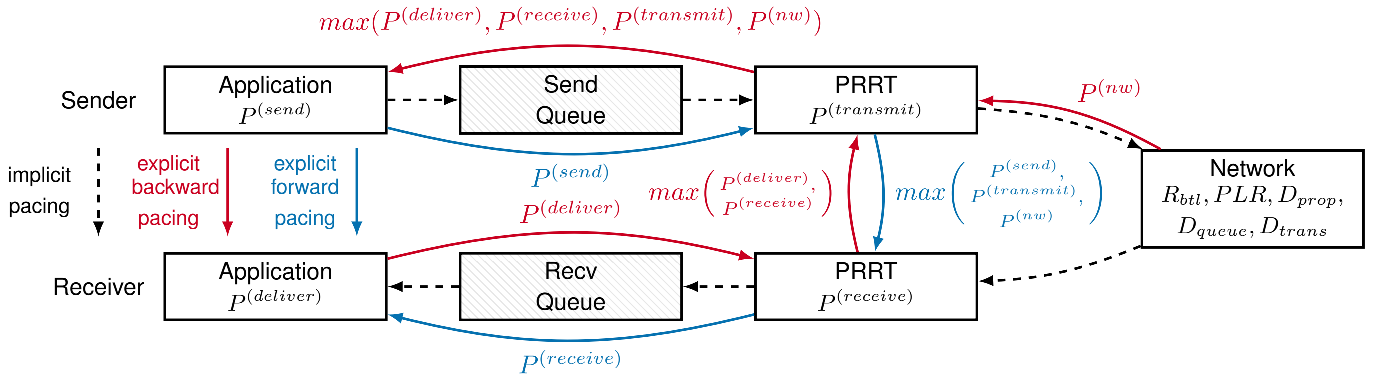


Fig. 3: X-PACE in PRRT communicates and adapts to paces across the entire system

A. Transport Protocol Design

The *Predictably Reliable Real-time Transport Protocol* (PRRT) [8] implements partial reliability and in-order delivery to allow statements about the timing characteristics of the transport. Thereby, it is able to incorporate application requirements (e.g. acceptable residual losses, maximum E2E latency) as well as network characteristics (e.g. loss rate, propagation delay) to parameterize its network functions. These include *error control* via hybrid ARQ and *congestion control* using a scheme similar to BBR [9]. Thus, it combines partial reliability with real-time characteristics.

The protocol supports bi-directional communication, but treats each direction independently. The sending part of a socket sends *data* and *redundancy* packets, while the receiving part selectively acknowledges these packets with *feedback* packets. Every piece of data that is passed to the stack from the application is tagged with a configurable expiry date, to state the tolerated latency.

For the purposes of the paper, we run PRRT on top of UDP in an IP-based environment to make it *internet-deployable* [10]. Technically, PRRT can run on any other lower-layer protocol that provides process multiplexing and best-effort delivery. The protocol and its implementation are published under an open source license[1].

1) Network Measurement: PRRT aims to be broadly applicable and not to depend on guarantees of a lower layer, yet these can improve its performance. Thereto, it measures the round-trip time (similar to NTP) in cooperation with the congestion control process described in Sec. IV-A2. Additionally, throughput is measured by a dedicated packet tracking facility, following a recent IETF draft for delivery rate estimation [11]. Finally, the protocol keeps track of incoming packets and calculates loss statistics on-the-fly (i.e., loss rates and correlation).

2) Congestion Control: Our implementation of congestion control follows the IETF draft on BBR [12], the related Linux kernel source code [13], and the recent Google presentations at IETF [14]. The algorithm builds a model of the current bandwidth-delay product (BDP) by measuring the *bottleneck data rate* as well as the *round-trip propagation delay* and controlling the *congestion window* and the *pacing rate*. Ideally, the window is the BDP and the rate is the bottleneck rate [15]. As networks change, flows compete and measurements are imprecise, BBR uses a control loop that modulates the pacing rate by cycling through different gain values, in order to measure the current values and adjust the sending rate (e.g. to drain queues or to probe for data rate). Several adaptions of BBR were required to account for the conceptual differences between PRRT and TCP (e.g. datagram-oriented service).

[1] `http://prrt.larn.systems`

B. Measuring Paces in PRRT

During operation, PRRT keeps track of paces at different layers (cf. Fig. 3), which is the basis for X-PACE. In the following, we assume that the packet size L is fixed, while the pace in which a step is executed P is variable. The motivation are e.g. video streaming or sensor applications where a lower frame size or sampling rate are preferred over significantly larger end-to-end delays caused by queuing.

1) Pace Filter: For keeping track of a pace, a windowed maximum filter is maintained. The motivation behind this design is that the window size can be configured to be as long as the pace is expected to stay the same. A maximum filter design is used so that the worst case paces are considered.

2) Sender & Receiver Application Pace: An application sends or receives messages with an average size of $L, [L] =$ Byte and a frequency of $f, [f] = \mathrm{Hz}$ (i.e., a pace of $P = 1/f$). In principle, paces $P^{(send)}$ and $P^{(deliver)}$ can be retrieved by tracking calls to `send()/recv()` on the PRRT socket, an approach similar to [16]. With each call, the current time is stored to compute the frequency at which messages are to be sent / received. Hence, the application data rate is $R = L \cdot f, [R] = \mathrm{bps}$. For the receiver pace, we compensate for waiting times incurred by the preceding step.

3) Transport Protocol Paces: Both sides of the stack spend time processing data before it is available to the next step. At the sender, this is the time between accepting a packet from the application and sending it to the network. At the receiver, this is the time between receiving a packet from the network and storing it for delivery to the application. In both

cases, the code to execute all the encoding, error correction and additional functions happens in a sequential manner, which allows to track the time these functions take to be executed on the current system (yielding $P^{(transmit)}$ and $P^{(receive)}$).

4) Network Pace: The pace at which a network can transmit packets depends on the bottleneck data rate $R_{btl}, [R_{btl}] = \text{bps}$ of a network path and the size of the transmitted packets L. The packet size is known from the application pace measurements, and the bottleneck data rate is measured by PRRT as described in Sec. IV-A2. Consequently, the network pace is $P^{(nw)} = \frac{L}{R_{btl}}$, which is also the transmission delay per packet.

C. Controlling Paces in PRRT

With pace samples for all layers, the system synchronizes to the slowest pace. Thereto, each step takes part in both the forward- and backward-propagation of paces, by computing the maximum of its own pace and the pace it received from its following step to backward-propagate this information. The same is done for the reverse direction to forward-propagate the paces. The pace is controlled at two locations: The transmitting step of the protocol, with automatic pacing, as well as the sending application, using a cross-layer interface.

1) Automatic Pacing: Within the protocol itself, there is (currently) a single location where the pace is enforced. The PRRT socket can pace packets, by inserting additional delay between `send()`s if the receiver or network is the bottleneck.

2) Cross-Layer Interface: The application can leverage the X-PACE using three different approaches. First, our system provides an interface for the application to query the current bottleneck data rate. Using this information, a *pacing-aware application* can fine-tune its parameters (e.g. sampling rate, sensor resolution) to adjust to the bottleneck. Second, our system *detects* when an application is too fast and *reacts* by delaying the execution of these calls, to enforce the correct timing of send and receive operations. This approach is transparent to the application in functional terms—existing legacy (i.e., pacing-unaware) applications need no modifications. Third, the network stack *monitors* the application behavior and *predicts* the next send or receive operation. The operating system utilizes this estimation to schedule application processes at the right moment in time. This approach is semi-transparent to the application—it makes specific assumptions, in particular periodic behavior. Such an application behavior is generally detectable by monitoring the timing of function calls [16]. Hence, the next send and receive calls will be issued exactly when the next package can be processed.

V. EVALUATION

We compare the implementation of X-PACE in PRRT with state-of-the-art protocols to empirically demonstrate that X-PACE detects and adapts to bottlenecks at any layer.

A. Methodology

The evaluation compares PRRT and TCP, as to evaluate different bottleneck locations in a system of two communicating hosts. The figures show time series of measured latencies and give *cumulative density functions* (CDFs) to present their distribution. We present 99th percentiles to show that X-PACE can reduce tail latencies and give inter-percentile ranges between 1st and 99th percentile to show that we also improve the latency predictability by reducing the variation or *jitter*.

1) Testbed Setup: The evaluation setup comprises two physical hosts running OpenvSwitch and execute the test application as a Docker container. The hosts are connected via a direct 1 Gbps Ethernet link, but most experiments employ `netem` traffic shapers to make the link the bottleneck or control its delay characteristic. PTPv2 is used to synchronize system clocks to get reliable time samples. This synchronization happens out-of-band to avoid interferences caused by queues formed during the evaluations. The same control path is also used to trigger the evaluations using SSH. The hosts run Ubuntu 16.04 and Linux kernel 4.15, incorporating a recent version of the BBR congestion control algorithm for TCP.

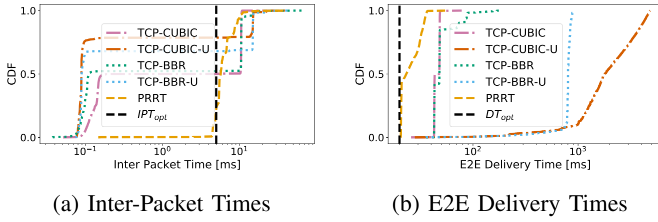
2) Measurement Application: The evaluations use a minimalistic application capturing the E2E delivery time (DT) a packet takes from the sender to the receiver application and the inter-packet time (IPT) in the sender application. The application layer protocol is identical for all transport layer implementations and sends fixed size messages, which are common for a control application in cyber-physical systems, containing the timestamp and the round number. The client captures the current time, composes a message and calls `send()`. This call can block for both TCP and PRRT, implementing the cross-layer interface mentioned in Sec. IV-C2. The receiver takes the packet out of the socket and uses the current time to get the DT. On the sender side, the time differences between calls to `send()` yield the IPT.

3) Parameter & Protocol Tuning: TCP sockets and several kernel options are tuned, to allow a fair comparison of TCP and PRRT. The sending queue size of PRRT is one packet, while the TCP socket option `SO_SNDBUF`[2] is set to scenario-dependent values. The receive buffer in PRRT is not size-limited, but packets that are not delivered to the application in time (100 ms for all scenarios) expire. `SO_RCVBUF` is tuned, so that flow-control can be used to cause pacing through backpressure in some scenarios. `TCP_NODELAY` and `TCP_QUICKACK` are set on the sender to avoid aggregation-induced latencies. TCP timestamps and SACK are disabled, while the `low_latency` option is enabled. Finally, `write()` to TCP socket is used to set the `PSH` flag, telling the end-host to deliver it immediately.

The PRRT pace filter window (cf. Sec. IV-B1) is set to 2 s in our evaluations, as observed system and network dynamics in our testbed were relatively stable during periods of this length.

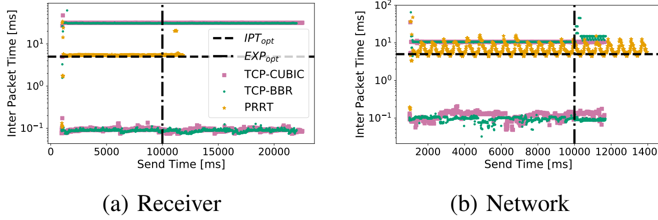
Fig. 4a shows how the options and buffer-limits impact the ability of TCP to pace packets. The sending and receiving application run as fast as possible, and the network is the bottleneck. The ideal IPT is 5 ms, which is also the network pace in this scenario, with 1000 B packets sent over a 1.6 Mbps

[2] `SNDBUF` in TCP limits the amount of unacked data in-flight. The queue in PRRT is only to decouple application `send()` and link layer `transmit()`.



(a) Inter-Packet Times (b) E2E Delivery Times

Fig. 4: Measurements for PRRT, as well as optimized and unoptimized (-U) variants of TCP

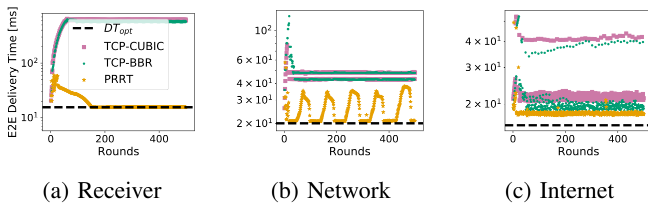


(a) Receiver (b) Network

Fig. 5: Inter-packet times for different bottlenecks

link. With a one-way propagation delay of 15 ms, this relates to a BDP of 6000 B. In the optimized scenario the SNDBUF and RCVBUF both are set to $3 \times BDP$ to allow for congestion control to work, but also put a limit on the amount of data in the TCP queues. The optimizations lead to nearly bimodal distributions for all TCP variants, with CUBIC [17] and BBR performing only slightly different. In comparison with TCP, PRRT narrows the distribution significantly with 4.40 ms and 12.79 ms as 1st and 99th percentiles. The distribution for PRRT is skewed towards slow paces, caused by our conservative approach to avoid filled buffers as much as possible.

Fig. 4b shows the E2E delivery times with the ideal line at 20 ms (15 ms one-trip propagation delay and 5 ms network pace / transmission delay). PRRT can approach this bound, but periodically deviates from it (cf. Fig. 1). This is caused by the probes for a higher share of the data rate by increasing the pacing rate, hence filling buffers. With a static link and no contention, one could get rid of this oscillation, but if congestion control should be part of the protocol this cannot be circumvented. The TCP series also show different traits of the congestion control algorithms, depending on their parameters. The unoptimized variants fill the buffers to a significant level, causing seconds instead of milliseconds of E2E delay, whereas the optimized versions perform constantly at around 45 ms.



(a) Receiver (b) Network (c) Internet

Fig. 6: E2E Delivery Times [ms] for different bottlenecks

TABLE I: Bottleneck scenario parameters

Scenario	Receiver	Network	Sender	Internet
R_{nw}	16 Mbps	1.6 Mbps	16 Mbps	10 Mbps
P_{nw}	0.5 ms	5 ms	0.5 ms	0.8 ms
D_{nw}	15 ms	15 ms	15 ms	15 ms
BDP	60 kB	6 kB	60 kB	37.5 kB
$SNDBUF$	$1\times$ BDP	$3\times$ BDP	$1\times$ BDP	$3\times$ BDP
$RCVBUF$	6 kB	$3\times$ BDP	$1\times$ BDP	$3\times$ BDP

B. Scenarios

To validate X-PACE, inter-packet time (IPT) and the E2E application layer delivery time (DT) are compared for different bottleneck locations in the system. Theoretical optimal values are based on the network parameters, for DT, IPT and experiment duration (EXP), respectively.

$$DT_{opt} = D_{prop} + P^{(nw)} \tag{2}$$

$$IPT_{opt} = max\{P^{(send)}, P^{(nw)}, P^{(deliver)}\} \tag{3}$$

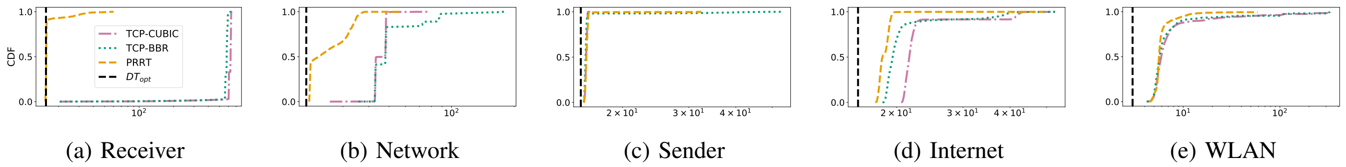
$$EXP_{opt} = IPT_{opt} \cdot rounds \tag{4}$$

The following diagrams include these values (as dashed black lines) and give a baseline to compare PRRT as well as different TCP variants against it.

1) Local Testbed: For all bottleneck scenarios, the configuration is so that the bottleneck pace is 5 ms, packet payloads are 1000 B and other parameters are as in Tab. I.

For the first scenario, we add a delay of $5\,\text{ms} \pm 20\,\%$ between `recv()` calls so that the receiving application is the bottleneck. RCVBUF is set to the 6 kB "BDP" of the receiving application, based on the channel RTT and the receiving applications rate. Thereby, flow control in TCP causes backpressure on the sender, implementing naive pacing. The pace of the sending application is only bounded by the processing speed, but is throttled down by X-PACE. Fig. 5a shows the IPTs of the different protocols during the evaluation, PRRT meets the optimal pace of 5 ms after a short startup of about 200 ms. The overall time until experiment completion for PRRT (11.85 s) is close to optimum (EXP_{opt} = 10.0 s), but far away for TCP CUBIC (22.61 s) and TCP BBR (22.53 s). The reason is that both TCP variants have IPTs that are either significantly higher (30 ms) or lower (0.1 ms) than the optimum of 5 ms. The DTs are shown in Fig. 6a, where PRRT needs a startup phase until the optimal DT of 15 ms is met for the rest of the experiment. Hence, PRRT avoids queues as much as possible, in contrast to TCP, which builds queues that lead to a constant delay of around 60 ms (cf. CDF in Fig. 7a).

The second scenario has a bottleneck at the network level (e.g. due to varying link characteristics) that must be communicated to the sender. Buffers are chosen so that congestion control works and potentially causes filling buffers if more than one BDP is in-flight. Fig. 5b shows that PRRT performs close to the optimum, while the IPTs oscillate in contrast to the receiver bottleneck evaluation. This is due to the BBR part of PRRT probing for a higher data rate using a faster pace for a short period. Fig. 6b also shows this oscillation, but with respect to the DTs. The faster pace causes the bottleneck



(a) Receiver (b) Network (c) Sender (d) Internet (e) WLAN

Fig. 7: CDFs for E2E Delivery Times [ms] for different bottlenecks

queue to fill, leading to increased delivery times. Nevertheless, PRRT achieves significantly lower DTs as queues are caused only by probing and not during normal operation, as with TCP. This is also visible in Fig. 7b, where TCP operates between 40 and 50 ms, while PRRT is way closer to the optimum.

The third scenario places the bottleneck in the sender by adding a delay of 5 ms $\pm$ 20 % between `send()` calls. Both protocols behave optimally as the system is inherently paced. Fig. 7c shows both TCP variants and PRRT are now achieving near-optimum DTs. This proves that our optimization of TCP by means of kernel options leads to an "as-fast-as-possible" forwarding without buffers and increased latencies.

2) Internet as a Network Bottleneck: We also evaluate PRRT and TCP on an Internet link across a straight-line distance of 20 km, using a residential access as one side and our testbed as the other[3]. The one-way delay is measured by ping and the sustainable data rate by iperf3. Fig. 6c and 7d show that TCP BBR achieves lower latencies than TCP CUBIC, which is also what the BBR authors intended. Notably, PRRT does not perform as close to the optimum as in previous experiments. This gap could be due to the precision of the time-synchronization between the hosts that both use NTP as a reference. The unavoidable imprecision in the clock synchronization affects all protocols and might lead to a constant systematic error on the absolute timescale. Still, the CDF of PRRT shows that latencies are more predictable and stable compared to the heavy tailed TCP variants. PRRT reduces the 99th percentile by up to 54 % (PRRT: 19.61 ms, TCP-CUBIC: 42.74 ms) as well as the range between the 1st and 99th percentile by up to 91 % (PRRT: 1.92 ms, TCP-CUBIC: 22.02 ms). This effect is smaller than in the testbed evaluations, but shows that PRRT outperforms TCP under conditions faced on Internet links.

3) WLAN as a Network Bottleneck: Finally, we also evaluate our approach in an 802.11ac wireless LAN, composed of an access point and two nodes. Fig. 7e shows the results of this evaluation in terms of E2E delivery times. While the curves are closer to each other, PRRT using X-PACE is able to achieve a smaller distribution (range between 1st and 99th percentile is 10.70 ms for PRRT and 304.54 ms for TCP-BBR).

VI. CONCLUSION

This paper has presented X-PACE, an approach to measure the current bottleneck pace of a networked system, communicate it across layers and between nodes, and enforce the correct pace everywhere. Our evaluation shows that X-PACE keeps the communication latency near the network and system limit (down to 1.24× the theoretical minimum), and it reduces jitter significantly (up to 11.4×), compared to TCP. Therefore, we strongly recommend that other transport protocol implementations consider cross-layer pacing as a mechanism to achieve predictably low age of information. Future work will explore the possible energy savings when computers slow down to adapt to the bottleneck pace, as well as how the system scales with the number of processing steps.

[3]The traceroute reveals that the actual covered distance is much higher and includes multiple hops at Internet Exchange Points.

REFERENCES

[1] W. Shi and S. Dustdar, "The promise of edge computing," *IEEE Computer*, vol. 49, no. 5, pp. 78–81, May 2016.

[2] S. Peter, J. Li, I. Zhang, D. R. K. Ports, D. Woos, A. Krishnamurthy, T. Anderson, and T. Roscoe, "Arrakis: The operating system is the control plane," in *Proc. of the 11th Symposium on Operating Systems Design and Implementation (OSDI'14)*, 2014, pp. 1–16.

[3] J. Gettys and K. Nichols, "Bufferbloat: Dark buffers in the Internet," *ACM Queue*, vol. 9, no. 11, pp. 40:40–40:54, Nov. 2011.

[4] L. Zhang, S. Shenker, and D. D. Clark, "Observations on the dynamics of a congestion control algorithm: The effects of two-way traffic," *ACM SIGCOMM Computer Comm. Review*, vol. 21, no. 4, pp. 133–147, 1991.

[5] A. Aggarwal, S. Savage, and T. Anderson, "Understanding the performance of TCP pacing," in *Proc. of the IEEE INFOCOM*, vol. 3. IEEE, 2000, pp. 1157–1165.

[6] M. Ghobadi and Y. Ganjali, "TCP pacing in data center networks," in *Proc. of the 21st Annual Symposium on High-Performance Interconnects (HOTI'13)*. IEEE, 2013, pp. 25–32.

[7] S. Rumble, D. Ongaro, R. Stutsman, M. Rosenblum, and J. Ousterhout, "It's time for low latency," in *Proc. of the 13th Conference on Hot Topics in Operating Systems (HotOS'11)*, 2011, pp. 1–5.

[8] M. Gorius, "Adaptive Delay-constrained Internet Media Transport," Ph.D. dissertation, Saarland University, 2012.

[9] N. Cardwell, Y. Cheng, C. S. Gunn, S. H. Yeganeh, and V. Jacobson, "BBR: Congestion-based congestion control," *ACM Queue*, vol. 14, no. 5, pp. 50:20–50:53, Dec. 2016.

[10] S. McQuistin, C. Perkins, and M. Fayed, "TCP goes to hollywood," in *Proc. of the 26th Int. Workshop on Network and Operating Systems Support for Digital Audio and Video (NOSSDAV'16)*. ACM, 2016, p. 5.

[11] Y. Cheng, N. Cardwell, S. H. Yeganeh, and V. Jacobson, "Delivery rate estimation - IETF DRAFT," 2017.

[12] N. Cardwell, Y. Cheng, S. H. Yeganeh, and V. Jacobson, "BBR congestion control - IETF DRAFT," 2017.

[13] "Linux kernel repository: TCP BBR," https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/log/net/ipv4/tcp_bbr.c.

[14] N. Cardwell, Y. Cheng, C. S. Gunn, S. H. Yeganeh, I. Swett, J. Iyengar, V. Vasiliev, P. Jha, Y. Seung, K. Yang, M. Mathis, and V. Jacobson, "BBR Congestion Control Work at Google IETF 102 Update," https://www.youtube.com/watch?v=LdjavTiMrs0&t=1h10m3s, 2018.

[15] L. Kleinrock, "Internet congestion control using the power metric: Keep the pipe just full, but no fuller," *Ad Hoc Net.*, vol. 80, pp. 142–157, 2018.

[16] T. Cucinotta, F. Checconi, L. Abeni, and L. Palopoli, "Self-tuning schedulers for legacy real-time applications," in *Proc. of the 5th European Conf. on Computer Systems (EuroSys'10)*. ACM, 2010, pp. 55–68.

[17] S. Ha, I. Rhee, and L. Xu, "CUBIC : A new TCP-friendly high-speed TCP variant," *ACM SIGOPS Operating Systems Review*, vol. 42, no. 5, pp. 64–74, 2008.