Cross-layer Latency-Aware and -Predictable Data Communication

Dissertation

zur Erlangung des Grades des

Doktors der Ingenieurwissenschaften (Dr.-Ing.)

der Fakultät für Mathematik und Informatik

der Universität des Saarlandes

vorgelegt von

Andreas Schmidt

Saarbrücken, 2019

Tag des Kolloquiums:	07. April 2020
Dekan der Fakultät:	Prof. Dr. Thomas Schuster

Prüfungsausschuss:

Vorsitz:	Prof. Dr.-Ing. Holger Hermanns
Berichterstatter:	Prof. Dr.-Ing. Thorsten Herfet
	Prof. Dr.-Ing. Wolfgang Schröder-Preikschat
	Prof. Dr.-Ing. Klaus Wehrle
Beisitzer:	Dr.-Ing. Richard Membarth

Kurze Zusammenfassung

Cyber-physikalische Systeme werden immer relevanter für viele Aspekte des Alltages. Sie sind zunehmend verteilt und benötigen daher Netzwerktechnik zur koordinierten Erfüllung von Regelungsaufgaben. Um dies auf eine robuste und zuverlässige Art zu tun, ist Latenz-Bewusstsein und -Prädizierbarkeit auf allen Ebenen der Informations- und Kommunikationstechnik nötig. Diese Dissertation beschäftigt sich mit der Implementierung dieser zwei Latenz-Eigenschaften auf der Transport-Schicht, sodass Regelungsanwendungen deutlich besser unterstützt werden als es traditionelle Ansätze, wie TCP oder RTP, können. Hierzu wird das PRRT-Protokoll vorgestellt, inklusive seiner besonderen Eigenschaften (z.B. partiell zuverlässige, geordnete, rechtzeitige Auslieferung sowie Latenz-vermeidende Staukontrolle) und unkonventioneller API. Das Protokoll wird mit Hilfe von X-Lap evaluiert, welches speziell dafür entwickelt wurde Protokoll-Designer dabei zu unterstützen die Latenz-, Timing- und Energie-Eigenschaften von Protokollen zu verbessern. PRRT vermeidet Latenz-verursachenden *Bufferbloat* mit Hilfe von X-Pace, einer *Cross-Layer Pacing* Implementierung, die in dieser Arbeit präsentiert und mit Experimenten auf realen Internet-Pfaden evaluiert wird. Neben PRRT behandelt diese Arbeit *transparente Übertragungssegmentierung*, welche dazu dient dem TCP-basierten Transport individuelle Link-Latenzen bewusst zu machen und so die Vorhersagbarkeit der Ende-zu-Ende Latenz zu erhöhen.

Short Abstract

Cyber-physical systems are making their way into more aspects of everyday life. These systems are increasingly distributed and hence require networked communication to coordinatively fulfil control tasks. Providing this in a robust and resilient manner demands for latency-awareness and -predictability at all layers of the communication and computation stack. This thesis addresses how these two latency-related properties can be implemented at the transport layer to serve control applications in ways that traditional approaches such as TCP or RTP cannot. Thereto, the *Predictably Reliable Real-time Transport* (PRRT) protocol is presented, including its unique features (e.g. partially reliable, ordered, in-time delivery, and latency-avoiding congestion control) and unconventional APIs. This protocol has been intensively evaluated using the X-Lap toolkit that has been specifically developed to support protocol designers in improving latency, timing, and energy characteristics of protocols in a cross-layer, intra-host fashion. PRRT effectively circumvents latency-inducing bufferbloat using X-Pace, an implementation of the *cross-layer pacing* approach presented in this thesis. This is shown using experimental evaluations on real Internet paths. Apart from PRRT, this thesis presents means to make TCP-based transport aware of individual link latencies and increases the predictability of the end-to-end delays using *Transparent Transmission Segmentation*.

Abstract

Cyber-physical systems are making their way into more and more aspects of everyday life (e.g. self-driving vehicles in logistics or telemedicine for medical services). In contrast to pure information-processing systems, these systems come with strict requirements towards resilience but in particular also to reliable timing. Providing the latter is challenging, given that timing is mostly considered as a performance aspect of cyber systems. A further challenge is that these systems are increasingly distributed and hence require a network infrastructure to fulfil control tasks in a coordinated manner. As these systems are supposed to interoperate, protocols and standards should be open and accessible—which makes it straightforward to look at the Internet Architecture and its protocols.

Providing this interoperability in a robust and resilient manner demands for latency-awareness and -predictability at all layers of the information-processing and communication stack. This requires a detailed look at the fundamental network functions for CPS—including the trade-offs they pose with respect to latency and other desirable properties. This thesis addresses how both, latency-awareness and -predictability, can be implemented at the transport layer to serve control applications in ways that traditional approaches such as TCP or RTP cannot.

Thereto, the *Predictably Reliable Real-time Transport* (PRRT) protocol is presented, which comes with unique features and unconventional APIs compared to state-of-the-art transport layer protocols. Among these features is, for instance, the exposure of all *link characteristics* (latency, throughput, reliability) towards the application. These measurements are carried out by PRRT to provide error, congestion, as well as rate control and can be accessed via the socket object. In contrast to the fully reliable TCP, PRRT implements *partially reliable ordered in-time delivery* which allows to trade resilience for latency—enabling applications to explicitly state their requirements and let PRRT adapt to those requirements and the dynamic link characteristics. By providing this unconventional service, PRRT as well as the application become latency-aware, which leads to increased predictability in terms of timing.

The PRRT protocol has been intensively evaluated using the X-Lap toolkit that supports protocol stack analysis in a low-overhead, cross-layer, and intra-host fashion. Thereby, it supports protocol developers in improving latency, timing, and energy characteristics by changing the protocol implementation or choosing appropriate configurations. A further step towards reproducible network experiments is the *Network Experi-*

ment Automation Tool (NEAT), which leverages state-of-the-art tools from continuous development and integration, configuration management and network automation.

An unconventional approach to tackle the increased latencies due to bufferbloat is the novel *cross-layer pacing* approach presented in this thesis—and implemented in PRRT in the form of X-Pace. Thereto, processing and communication paces are measured at all steps of an end-to-end transport stack and afterwards exchanged between these steps so that the position and pace of the current bottleneck is known at runtime. This synchronization of pace information is continuously done and individual steps act upon this information to achieve near-zero queuing delay and avoid processing data too fast (which usually requires more resources). Using experimental evaluations on real Internet paths, it is shown how X-Pace allows PRRT to achieve predictable end-to-end delivery times that are close to the theoretical optimum.

Finally, this thesis investigates how TCP-based network transport can achieve increased latency-awareness and -predictability. This is done using the *Transparent Transmission Segmentation* (TTS) paradigm that deploys so-called *relays* in the network using Software-Defined Networking as well as Network Function Virtualization. Acting as termination points for the transport protocol, these systems allow a transport layer to act on granularities down to a single physical link, so that its operation can be fine-tuned to the link's unique characteristics. Various experiments show that indeed the TTS paradigm is able to reduce end-to-end flow completion times for different scenarios (e.g. a lossy last mile link or networks with high jitter) as well as increase its predictability.

Acknowledgments

First and foremost I want to thank Thorsten Herfet for providing just the right amount of guidance and support in the last four years. I will miss the intensive discussion we had during the various road trips that usually involved quite some bad weather. I want to thank wosch, not just for acting as a reviewer of this thesis, but also for providing a lot of insights, pragmatism and bits of wisdom during various meetings of the LARN project. I am grateful to Klaus Wehrle for acting as a reviewer and for having hosted us during the Aachen BarCamp in 2018, during which I realized how relevant the practical parts of this thesis are. Holger Hermanns has not only been an invariant during my graduate and doctoral University studies (signing the “Mathe-Preis” for my Abitur, acting as a reviewer in my qualifying exam, and now as the chairman for my doctoral examination), I have also enjoyed talking about our shared passion for an East Frisian Island whenever we met in public transport.

I have been gifted to work in a great team at the telecommunications lab. Harini, Kelvin, and Tobias have been with me during my whole dissertation phase and I am very happy that we had this good atmosphere, while working towards a PhD. I want to thank Tobias for being a great room-mate and always jumping in when help is needed. Thanks to Kelvin for taking the first step and approaching me in our first TC1 tutorial, given by Christopher. These two, together with my thesis advisor Michael, are responsible that I have considered this journey that now led to me writing these lines. I want to thank Pablo for being a great master student and an even better colleague that is always curious, willing to help, and never afraid to ask questions. I have been honoured to work together with many students, who worked as student researchers or wrote their thesis with my advise—namely Philipp, Lukas, Kai, Marlene, Ashkan, and Sebastian. You have made me wiser by asking the right questions and your work has accelerated mine a lot. Without Frank, I would have probably never ventured to design Hands-On Networking, an endeavour that helped me grow and got me acquainted to many ambitious students. Lastly, I want to thank Zakaria and Diane for making non-scientific processes as smooth as possible and for taking care of any problem in a brisk manner.

I am also grateful for my collaborators at Friedrich-Alexander-University Erlangen-Nürnberg, Timo Hönig and Stefan Reif, as well as all the other peers I have got to know through the *DFG Priority Programme 1914 - “Cyber-Physical Networking”*. Without the SPP and all the involved people, the road to my dissertation would have been way more rocky.

I want to thank Oliver and Claudia for being another constant, for now over 15 and 20 years. While I feel a bit sad about no longer working or studying in close proximity to you, I am grateful for having you there for so long and providing a stability I can always return to. I am thankful for Agatha and her family, in particular my godchild Elias, for the time we spend together and the way you appreciate my presence. I also want to thank Sabrina for being both a mentor and very good friend at the same time; and still believing that I will once be the next Bill Gates. It makes me very happy that I am still in contact with my friends from my master programme, namely Felix, Kathrin, and Daniel. Having Maike, Claudius, and Aline as peers for intensive chats on the train between Scheidt and Limbach has made my daily commute more enjoyable.

I want to thank my friends from the *Deutsche Lebens-Rettungs-Gesellschaft*, namely Basti, Sascha, Udo, Jan, Seb, David, Bene, Elena, Michel, Sven, as well as many others from the Ortsgruppen Kirkel and Blieskastel. Despite the fact that free-time becomes more and more limited, I enjoy being together with you guys and girls, having fun and doing something with purpose at the same time. Thank you, Matthias, for all the guidance you gave me when I was just learning to teach swimming to kids and for the trust you had. Your deeds will never be forgotten.

I also want to mention a set of people that have made the last four years a great experience, by being nice neighbours, mentors, or chat and sports partners: Anna, Charlotte, Freddy, Dennis, Hanna, Michael, Michaela, Annika, Tanja & Uwe.

Thanks to my parents and grandparents, Marita, Hans-Peter, Helge, and Dieter for raising me up the way they did and enabling the way of life I currently live and love to live. Finally, I want to thank Louisa for the many lessons she taught me and for loving me, despite the fact that she thinks that almost all my jokes are not funny and that my work is—in large parts—senseless. I love you and therefore hope that at least my life outside work makes sense.

Contents

List of Figures

List of Tables

Nomenclature

ACK	Acknowledgement
AoI	Age-of-Information
ARQ	Automated Repeat reQuest
ASAP	As-soon-as-possible
BBR	Bottleneck-Bandwidth and Roundtrip Propagation Time
BDP	Bandwidth-Delay-Product
CCA	Congestion Control Algorithm
CDF	Cumulative Density Function
CPS	Cyber-physical System
DC	Data Center
DT	Delivery Time
E2E	End-to-End
FCT	Flow-Completion Time
FEC	Forward Error Coding
FPP	Fast Packet Processing
GE	Gilbert-Elliot (Model)
GPS	Global Positioning System
HARQ	Hybrid ARQ
I4.0	Industry 4.0
IETF	Internet Engineering Task Force

IO	Input/Output
IoP	Internet-of-Production
IoT	Internet-of-Things
IPC	Inter-Process-Communication
IPT	Inter-Packet Time
IQR	Inter-Quartile Range
ISP	Internet Service Provider
LARN	Latency- And Resilience-aware Networking
LTE	Long Term Evolution
MDP	Markov Decision Process
MTU	Maximum Transmission Unit
NAT	Network Address Translation
NCS	Networked Control System
NFV	Network Function Virtualization
NIC	Network Interface Controller
ON	Open Networking
OS	Operating System
OSI	Open Systems Interconnection
PEP	Performance-enhancing Proxy
POSIX	Portable Operating System Interface
PRRT	Predictably Reliable Real-time Transport
PTP	Precision Time Protocol
QoS	Quality-of-Service
QUIC	Quick UDP Internet Connections
RFC	Request for Comments
RNA	Reliable Networking Atom

RTT	Roundtrip Time
SACK	Selective ACK
SDG	Sustainable Development Goal
SDN	Software-Defined Networking
SGE	Simplified Gilbert-Elliot (Model)
STP	Spanning Tree Protocol
TCP	Transmission Control Protocol
TLS	Transport Layer Security
TSN	Time-Sensitive Networking
TTS	Transparent Transmission Segmentation
UDP	User Datagram Protocol
VM	Virtual Machine
VNF	Virtual Network Function
WCET	Worst-Case Execution Time

True happiness comes from the joy of deeds well done, the zest of creating things new.

Antoine de Saint-Exupéry

Chapter 1

Preface

We live in a time where digital computation and communication (i.e. cyber) systems are making their way into more and more aspects of everyday (physical) life. While this is straightforward for information-processing domains and services (e.g. knowledge work or governance), it is not that apparent for areas that involve control of physical processes[1] (e.g. self-driving vehicles or manufacturing facilities). These control processes are in many cases already involving digital electronic components, but are dwarfed by the apparent capabilities of soft- and hardware developed for communication and entertainment purposes. If we consider the Internet as the unified network infrastructure for any communication service that is accessed from nearly everywhere and everyone, it is straightforward to expect major gains for these control applications in leveraging Internet technology—in particular with respect to interoperability that further paves the way for the merging of cyber and physical worlds.

This rise in capabilities of digital technology is due to advances in manufacturing of computing hardware with high computational power, vast amounts of storage and memory, low energy footprint, and at a low cost[2]. Having access to this capable and affordable technology allows engineers to build highly sophisticated cyber systems. Lee [Lee17] argues that it is the symbiosis of these cyber systems with physical systems—leading to *Cyber-Physical Systems* (CPS)—that will allow new application types that help to advance humanity. In Lee's point of view, this is particularly the case because both computational as well as physical processes are (highly likely to be) fundamentally different and therefore complement each other in an invaluable manner. To get the terms clear, it should be noted that CPS are comprised of *embedded systems* (i.e. information processing systems integrated into products) and have a strong focus on the physical environment with respect to its quantities such as time, energy, and space [Mar11].

[1] This thesis considers physical processes as physical phenomena that gradually change along a (continuous) series of states.

[2] It should be noted that though the costs have reduced, there has also been an increase in demands that—to some extent—alleviate the efficiency gains. This can be considered as the computing incarnation of the *rebound* effect known from *energy economics*. This effect is originally a hypothesis by Jevons [Jev65] and received a recent treatment by the *Nature* journal [GKRW13]).

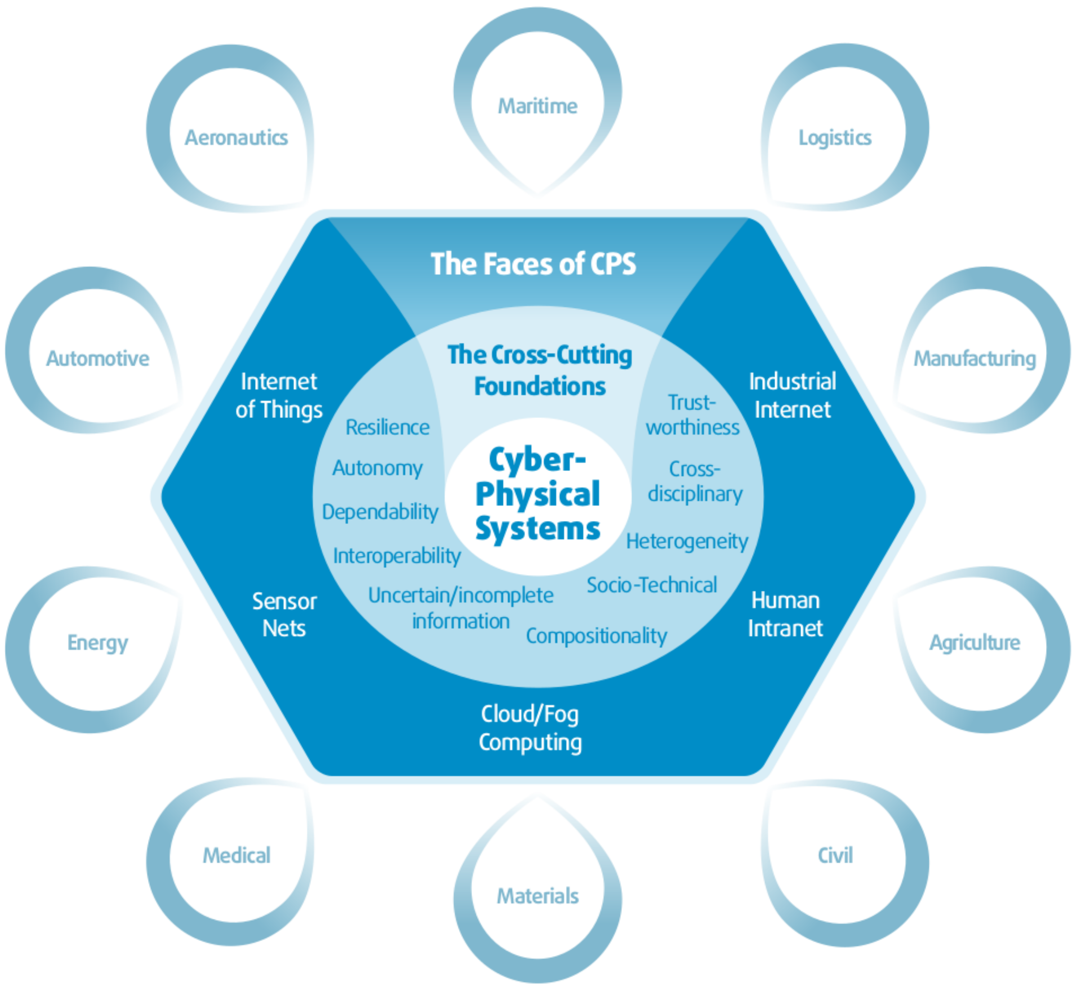


Figure 1.1: Distributed Cyber-Physical Systems enable new applications in various domains, but must fulfil strict requirements, such as dependability (CPS Summit [DS16]).

Considering the different application domains of CPS, as depicted in Figure 1.1 and described further in Chapter 2, there is enormous potential for coupling the cyber and physical worlds as tightly as possible, for instance in mobility/logistics (self-driving vehicles), manufacturing (Industry 4.0), smart spaces (Internet-of-Things), and medical services (telemedicine).

From an engineering perspective, the previously mentioned applications require tight integration of communication, control, and system design. By having a closer look, it is evident that any of the applications for CPS depend on communication networks that have properties, such as resilience and dependability, but also interoperability. The protocols and systems that are used in today's Internet cannot provide all of the properties that CPS require. Considering Clark [Cla18], it is evident that *the Internet of tomorrow* might be a different internet than *the Internet we use today*, in particular because the *Internet architecture* is separate from the implementation of its mechanisms and its actual deployment. In order to support CPS, both the implementation and deployment are bound to change.

With the current networked and distributed systems that leverage the Internet, one cannot answer any of the following questions reliably: How long does a remote operation or transmission take? How likely will it finish on time? Would an extra millisecond of delay be acceptable? How likely will a message arrive? How to deal with faults? How to increase the success rate of the applied measures? How to ensure proper (temporal and functional) operation? How can system layers be crossed to improve performance?

Answering these questions requires the application of cross-layer design paradigms and holistic analysis of all system layers, starting from the physical hardware, over the operating systems, through different communication systems, and up to the applications [Lee08, Mar11]. While briefly touching all of these layers, the main focus and contributions of this thesis are in the design and implementation of latency-aware and -predictable communication systems, with strong interconnections to the other layers.

1.1 Problem Statement

While the challenges for distributed CPS are manifold (e.g. achieving reliable timing, being fault-tolerant, and incorporating human behaviour), this thesis addresses three aspects that are related to networking technology and proposes solutions for all of them.

A1: Cross-layer Network Protocol Design and Implementation Looking at transport protocols that are common today, there is a gap between the services these provide and the requirements that CPS have towards the communication. In particular, these protocols (e.g. TCP [Pos81], QUIC [Goo18]) lack the capability to exchange and consider timing constraints of the application, for instance a maximum tolerable age-of-information to ensure reliable operations. Only by knowing about the applications' demands and the actual latencies caused by intermediate computation and communication steps, one is able to find suitable configurations or notify the application when demands cannot be fulfilled—allowing graceful and safe handling.

A2: Flow-level Optimization of Transmissions at the Network Core The channel characteristics of links on today's Internet are highly heterogeneous, as they are using different physical media, employ varying multiple access schemes and use different protocols. In the future, this heterogeneity is going to increase further with the use of new link technologies (e.g. 5G NewRadio [DPS18]). CPS are expected to leverage these links to achieve distributed control loops over long distances with high reliability. Using pure end-to-end approaches for network transport is bound to miss the potential of applying link- or segment-local optimizations for communication of CPS.

Figure 1.2 depicts a control scenario where the link from a cloud server (inside a datacenter) to an edge radio and the link from this radio to an edge device have dif-

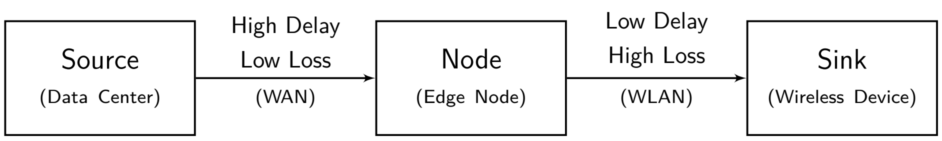


Figure 1.2: End-to-end (E2E) scenario with heterogeneous link characteristics that lead to suboptimal performance of network functions, e.g. error and congestion control.

ferent delay and loss characteristics. If we use a fully reliable transport such as TCP, there are multiple issues with this: First, losses on the last link cause retransmission with higher latency, as they have to traverse the first link again. Second, loss-based congestion control reduces the sending rate regularly, so that it is unlikely that one can fully use the maximum data rate of the first link and especially not the maximum data rate of the second link. The pure end-to-end operation [SRC84] prevents a link-level operation of network functions for a particular flow of data. Consequently, it is necessary to systematically investigate segmentation approaches that enable link-level flow optimization, which reduces latency as well as data overhead and increases utilization.

A3: Measurement and Control for Cross-layer Optimization Developing robust networked systems relies on measurement and analysis of communication parameters to optimize the transmission. With respect to the protocols with predictable reliability and latency, thorough knowledge about the channel characteristics, i.e. delay, data rate, and loss rate, is required. This is also true for applying segmentation as one can only find out about heterogeneity by constantly measuring the network and developing good models for current and future link parameters. Consequently, we need to employ data science approaches, algorithms, as well as statistical knowledge to tackle these challenges[3].

1.2 Research Question

If we consider the layers of which digital computing systems are composed (cf. Figure 1.3), it can be seen that while the (in this abstract model two) *lower layers* expose predictable and reliable timing, i.e. the digital circuits or wires, the higher layers cannot achieve the same due to abstractions that ignore timing [Lee08]. The *highest layer* (i.e. the *intent* of one or more users) in contrast has a very differentiated notion of requirements with respect to latency, e.g. the consumption of a live video stream or the usage of a control application has a maximum tolerable latency for each unit of data. In fact, the *middle layers* (marked in light grey) are mostly *agnostic* of latency that their operation induces. This leads to a situation where the latency that the higher layers face is not predictable and it gets harder for these to be aware of the latencies lower layers cause.

[3]cf. Andreas Schmidt: “Network Traffic and Infrastructure Analysis in Software Defined Networks”, Master Thesis, Saarland Informatics Campus, 2015

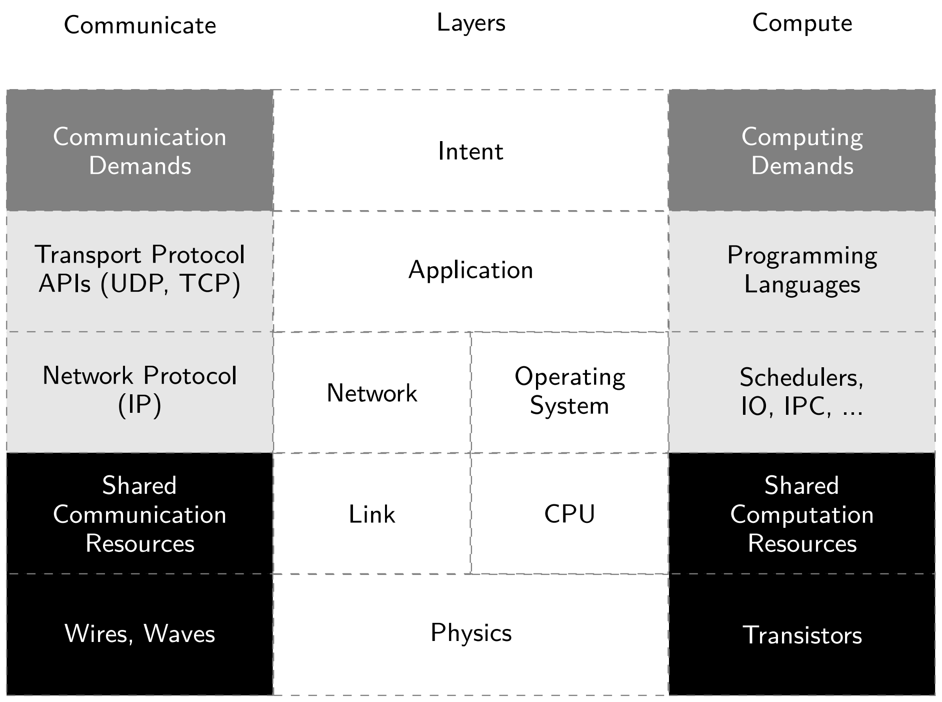


Figure 1.3: Both communication and computation parts of systems have layered architectures. With respect to latency, there are layers that have *reliable and predictable* latency (black), that are *aware* of latency (dark grey), and which are often *agnostic* of latency (light grey).

The research question of this thesis is how we can bridge this gap for *communication* systems to provide both latency-awareness and -predictability beyond existing knowledge and solutions in the field—with a particular focus on the transport layer. These two properties are defined as follows:

Latency-awareness A layer is *latency-aware* when it knows (a) what latency characteristics it requires from the layer below and (b) what latency it causes to the layer above. Furthermore, a layer should ideally *share* information about its own latency with the layer above, which can then adapt.

Latency-predictability As soon as awareness is established, a further step is to make the latency of a layer *predictable* and also share this confidence about the timing with the adjacent layers. A reduction of the latency on an absolute scale could also be achieved, but it should be ensured that having a predictable latency is always preferred over having a low latency.

The *compute* half of Figure 1.3 is also relevant, because transport layers are effectively executed by operating systems, which must not be agnostic of the latency induced by certain operations in order to allow CPS operations. Due to tight interconnection, it is a necessity to jointly analyse both sides, computing and communication. Therefore, this thesis elaborates on computing system aspects wherever necessary while having a clear focus on the communication aspects.

Besides communication and computation, *control* is the third component that we find in *cybernetics* [Wie48], which is going to be further elaborated on in Chapter 2 and is represented in Figure 1.3 by the *intent*. To give a concrete example, a user of a self-driving car has the intent (and strong demand) to not crash into anything, which, together with the physics of driving, govern what she expects from the computing as well as communication system.

In summary, we investigate ways to properly treat latency as a *functional aspect* in CPS, in contrast to traditional information processing where it is only a non-functional, performance-related aspect [LS17].

1.3 Contributions

With respect to this research question, this thesis contributes the following theoretical considerations, empirical observations, and practical implementations to achieve reliable latency-awareness and -predictability in transport layers of CPS. These contributions are essential artefacts of the *Latency- And Resilience-aware Networking* (LARN) project[4], as they allow a tight cooperation of the networking and operating system domains to support CPS applications.

Predictably Low Latency and Latency-awareness in Transport Protocols First and foremost, this thesis describes the *Predictably Reliable Real-time Transport* (PRRT) protocol, a straightforward to use, openly available[5] transport protocol that can replace existing UDP or TCP solutions. PRRT (as described in Chapter 4) is (a) *channel-aware* with respect to all parameters (latency, throughput, reliability), (b) *system-aware* with respect to processing latencies that are incurred by the communication-related computations as well as the application itself, and (c) *application-aware* in that it allows the application to explicitly state its demand with respect to maximum age-of-information. This holistic awareness is leveraged in decisions about the configuration of hybrid error control, loss-avoiding congestion control, as well as cross-layer packet pacing—leading to predictably low latencies and hence reliable timing.

Cross-layer Latency and Timing Analysis While designing PRRT to provide predictable latency to control applications, we have created the tool X-Lap, which allows

[4]http://larn.systems (accessed August 15, 2019)
[5]http://prrt.larn.systems (accessed August 15, 2019)

cross-layer latency and timing analysis. In particular, X-LAP uses a minimally invasive run-time capturing system to take timestamps at various steps within the communication stack to properly quantify latencies. This allows to identify the sources of latency and jitter across end-hosts and among the layers in the stack. X-LAP and its extension Δelta thereby close a gap that existing network- or system-only profilers (e.g. `wireshark` and `gprof`) leave open. Thereto, X-LAP serves as a tool to validate predictable and reliable timing and provides empirical measurements to further guide the engineering process.

Cross-layer Pacing Even though it is a well-known fact in manufacturing that one can achieve just-in-time processing when bottlenecks are known and communicated, this approach is not yet widely used in communication and information-processing systems. This thesis makes the case for a thorough capturing and synchronization of processing latencies across all the steps that are involved in process-to-process communication. Leveraging this awareness, sub-systems which are not the bottleneck can take measures to avoid "waste"[6], leading to lower latencies due to empty buffers, fewer losses due to avoiding bursts, and lower energy demand by slower execution—achieving savings with respect to the time, memory, or energy resources. This cross-layer pacing approach X-PACE is designed as well as implemented (as an extension to PRRT) and able to outperform TCP variants that are optimised for low latency. Thereby, cross-layer pacing increases predictability of latencies and increase overall reliability by avoiding losses.

Transparent Transmission Segmentation Accompanying these end-host-centric contributions of this thesis, a further look is taken at *transparent transmission segmentation* (TTS) approaches that allow to localize network functions for improved performance. A thorough description of the segmentation problem is given, considering the demands we have towards this technology, the dimensions it affects, as well as the network functions that are involved. The approach is formalized, allowing exemplary theoretical considerations with respect to flow control that are later validated empirically. In order to do this, a complete *relaying* approach is presented that leverages software-defined networking for deployment in open networks and presents a virtual networking function in the form of two different relay implementations that are able to segment transmissions. Further positive effects of the segmentation are measured and analysed, while more light is shed on limitations of the segmentation approach—in particular when TTS achieves too little gain to compensate for the overhead it induces.

1.4 Outline

This thesis is composed of two parts: Chapters 2 to 4 lay the *foundation* to and set the context of this work. Within Chapter 4, the thesis smoothly transitions into the *contribution* part in Chapters 5 to 7. The chapters are structured as follows.

[6]This follows the Toyota Production System [Ohn88] terminology, where waste or *muda* is any human or natural resource and even time that one could have spared.

First, Chapter 2 introduces cyber-physical systems, their application domains, and why it is not straightforward to fulfil their communication needs with existing technology. Particular attention is paid to distributed, networked CPS where reliable timing of the communication is essential. In this context, the *Reliable Network Atom* (RNA) and the OpenNetworking testbed are introduced as demonstrators for building communication stacks for CPS using off-the-shelf components paired with unconventional transport layer and operating system approaches. Chapter 3 holistically investigates latency aspects in distributed CPS, showing where individual latencies are caused, how systems are made aware of them and how latencies can be decreased as well as how the latency-predictability can be increased. With the knowledge about the composition of end-to-end latency, several transport layer functions are revisited regarding their impact on certain sub-latencies. Transport layer protocols are essential components that are required to bring latency-awareness and predictability to communicating CPS.

The PRRT protocol that is introduced in Chapter 4 provides such a CPS-enabled transport layer. PRRT is thoroughly analysed for its latency characteristics by X-Lap, a tool that is presented in Chapter 5. This chapter also considers reproducibility of experiments, which is essential when building reliable systems. Chapter 6 presents X-Pace, which allows to determine a system's bottleneck, communicate this information and allows the other non-bottleneck components to adapt. Thereto, a brief overview of the history of packet pacing is given, together with a general formulation of the cross-layer pacing process. At the network core, transparent transmission segmentation, as described in Chapter 7, is able to reduce latencies and increase predictability. This claim is proven by providing a practical segmentation solution for TCP and evaluating it in comparison to unmodified end-to-end communication.

The final Chapter 8 concludes the thesis by summarizing the contributions and giving an outlook to further directions of research, which add additional capabilities to the transport layer to be able to support CPS.

The machine does not isolate man from the great problems of nature but plunges him more deeply into them.

Antoine de Saint-Exupéry

Chapter 2

Networking in Distributed Cyber-Physical Systems

While the field of networking and telecommunications is well established—with the first documented networks consisting of fire signals in ancient Greece—demands have changed over the centuries and new network infrastructures have been created over and over again. The most prominent one today is the Internet, providing global access to a shared infrastructure for information flows that are fast and large in volume—independent of the actual service or content that they provide.

It is still a common (mis-)conception that the Internet, and the technologies it is composed of, are not suitable for services that are time-critical and require high reliability. In fact, such a generalization is inadequate and whether Internet technologies are used does not imply anything on its characteristics with respect to timing, reliability, or communication capacity. As Clark [Cla18] points out, *the Internet* is only the current choice of architecture and there are various other ways to build *an internet*[1] that will be considered in the future and can change how it operates. Instead of providing specific performance guarantees, the use of Internet protocols, software, and hardware allows *interoperability*—the core goal of the design of the Internet. This can be seen by a recent trend towards the use of Internet technology in the manufacturing industries[2]; a field that has for most of its time refused to use these due to the assumed insufficiencies.

Communication and networking that fosters *interoperability* is guaranteed to stay an essential building block of technology to solve current and future societal challenges. However, the course of the Internet's success does not stop at solving challenges in the domains of information or knowledge exchange, but continues to make its way to domains that *change the world* in a more literal way—through establishing control loops with physical processes.

[1]The capitalization and definiteness of the articles is in line with Clark and expresses that we can build inter-networks in many ways but currently have a single, though changing, implementation.

[2]Often referred to as the (Industrial) Internet-of-Things (https://www.iiconsortium.org/, accessed July 24, 2019) or Industry 4.0.

2.1 Cyber-Physical Systems

The prefix *cyber*, which is often associated with digital (and sometimes conceived of as parallel) worlds, has been originally coined by the American mathematician Norbert Wiener in his classic work *Cybernetics or Control and Communication in the Animal and the Machine* [Wie48]. At the core of this is the combination of communication, computation, and control based on closed-loop feedback to continuously measure and steer physical processes. Though his thoughts at this time could not consider digital computers or the Internet we know today, he already presents the theoretical ingredients and concepts that are at the core of the technology that we build and envision today.

From cybernetics, a domain of science, we get to the engineering term *cyber-physical systems*, which has been coined by Helen Gill in 2006 at the National Science Foundation in the US and can be defined as:

Definition 1 (Cyber-Physical System) *Cyber-Physical Systems (CPS) are integrations of computation with physical processes. [LS17]*

While this is apparently an extremely general definition, it captures the essence of CPS and how these differ from pure information processing (e.g. for financial or entertainment applications). Other (non-contradictory) definitions exist that characterize CPS as "a confluence of embedded systems, real-time systems, distributed sensor systems and controls" [RLSS10] or "[integrations of] software-intensive embedded systems with (potentially global) information systems" [DS16].

2.1.1 Applications of Cyber-Physical Systems

With this definition, and in particular the differentiation from information-processing, we can identify domains in which CPS are already essential or will be essential in the future. Unsurprisingly, these application domains are manifold and in most cases already include electronic components and software (i.e. cyber parts). But despite these circumstances, these contexts do not (yet) leverage the full potential of a holistic approach where cyber and physical components are at equal parts.

This upcoming evolution of technical systems is widely considered to improve operations in these application domains with respect to different metrics and objectives. As this is an engineering—and therefore creative—task[3], it is up to the humans who create (or are told to create) it to balance these demands with what is technically possible. A contemporary approach would be to consider the United Nations Sustainable Development Goals[4] (SDG) in order to achieve a sustainable use of technology that serves

[3]See [Lee17] for a detailed treatment of the difference between science and engineering.

[4]https://sustainabledevelopment.un.org/?menu=1300 (accessed July 23, 2019)

the society. The following provides a highly abridged list of relevant aspects and gives clues on how we can advance humankind through enabling more sophisticated CPS:

Environmental Footprint & Climate Action (SDG 13) Even though the environmental footprint metric (e.g. CO2 emissions or non-renewable resource usage) is central to all domains of human life and has to be considered more thoroughly, there are primary domains where CPS can reduce environment-unfriendly operations (leading to higher *efficiency*) or even avoid them completely (leading to better *sufficiency*). These improvements are possible even though CPS themselves increase the footprint of individual technology artefacts (e.g. through the use of rare materials to *host* the cyber components) but require careful investigation and system design. Promising domains are logistics (e.g. through improved scheduling or platooning [AAGJ10]), individual mobility (and its absence due to telepresence [Ste92]), as well as structures & buildings (e.g. through improved air conditioning [Sno03]). Finally, the advent of *affordable and clean energy* (SDG 7) can be achieved through the implementation of distributed energy grids [FMXY11] that are able to tackle the challenges that renewable, intermittent energy sources pose—eventually providing a sufficient and sustainable amount of energy to everyone.

Human Health and Well-Being (SDG 3) Due to their high reliability constraints and extensive certification procedures [Sta17], medical systems and devices can usually not keep up with the speed of innovation and often appear *antique* when compared with, for instance, state-of-the-art consumer technology. This situation can only change when newly created technology is robust and its reliability can be certified in a straightforward and brisk way. Furthermore, *medical operations* can also benefit from the use of real-time information systems as well as remote control. Telemedicine is expected to increase the range and frequency at which surgeons can operate and the use of advanced robotics is going to also reduce the invasiveness of treatments. In addition to these clinical applications, the fields of assisted living as well as (automated) medial & technical emergency services can benefit from advanced CPS technology. Lastly, through employing improved traffic control and safety systems as well as safe control of critical infrastructures, human lives can be saved by avoiding accidents or reducing their impact.

Process Efficiency, Cost-Efficiency, and Adaptability (SDG 9) In the domains of manufacturing and process control, major inefficiencies lie in the *interfaces* between involved partners and domains. While the former is often due to information that is lost between suppliers due to missing (or abundance of incompatible) standards, the latter is caused by the still-present divide between information technology and operations technology[5]. This divide is not just a cause to inefficiencies in the production process,

[5]Consider e.g. https://ics.sans.org/media/IT-OT-Convergence-NexDefense-Whitepaper.pdf for a security-focused view on the issue (accessed November 29, 2019).

but also an impediment to achieving higher product quality [LNPV18]. Furthermore, it harms adaptability and makes production facilities often highly static and inflexible.

Overcoming this divide is envisioned by several efforts, for instance *Industry 4.0* (I4.0) [LFK^{+}14], the *Industrial Internet-of-Things* (IIoT) [DXHL14, JBM^{+}17], or the *Internet-of-Production* (IoP) [PGH^{+}19]. Embracing these ideas and enabling them through advanced CPS is considered to reduce overhead at the interfaces, improve cost-effectiveness, and enable adaptability. When used in an appropriate and responsible way, this can help to secure and increase prosperity[6].

2.1.2 Challenges and Traits of Cyber-Physical Systems

With these applications in mind, it becomes evident that there is a multitude of challenges with today's technology that must be solved in order to enable implementations (cf. [BA07, Lee08, Sta17]). This is particularly relevant because CPS have significantly higher demands with respect to safety and reliability in comparison with general purpose information processing (cf. [Mar11], p. 10).

Timing The most important challenge is that the notion of *time* is not present in most of the computing and communication technology that is available today—a stark contrast to the fact that CPS are inherently concurrent and reliable timing is essential for correct operation. In information-processing software, timing is mostly considered as a non-functional aspect (i.e. a *performance* measure) and not as a functional aspect (i.e. a *correctness* measure). In spite of digital circuitry being extremely reliable and having reliable timing, the abstractions we use today do not capture timing in an appropriate way [Lee08]. This deficiency of the abstractions is present at most layers above the digital circuits, e.g. instruction set architectures, operating systems including concurrency primitives, as well as programming languages[7].

For the networking regime, the same is true as there are established (e.g. wavelength-division-multiplexing) and upcoming (e.g. *Time-Sensitive Networking* (TSN)[8]) solutions that provide predictable timing but do not use appropriate abstractions to *forward* this information up the stack (i.e. towards the application). In the top-down direction, it is still the case that network applications (e.g. messaging servers) are not able to explicitly state timing constraints—even though they might well have some and could share them. This is especially problematic as "passage of time is inexorable" [Lee08]. Due to this fact, CPS that come with software that does not properly consider timing are destined to only be *best-effort systems*, where reliable operation is a coincidence and not a guarantee with high confidence. Furthermore, CPS typically operate outside of

[6]It should not just lead to more production and consumption, as has been the case with many inventions that boost efficiency.

[7]A programming language approach to solve this has been presented in [SG10], but as of today there are no general purpose and widely used programming languages with temporal semantics.

[8]The work group was formerly known as the audio/video bridging task group: http://www.ieee802.org/1/pages/tsn.html (accessed August 14, 2019)

controlled environments and are faced with unpredictable, unexpected conditions and must handle partial failures.

Applications of today also have certain expectations about the *reliability* of subsystems (including the communication) and would be able to share them if the programming abstractions would allow to do so. Sharing these expectations allow the subsystems to optimize their operations accordingly and striving to fulfil them. This means that new ways to express, measure, and validate *reliable timing* must be developed that are fundamentally different from state-of-the-art approaches.

Reliability Reliable timing alone does not make a system reliable, and it is interesting to see how the demands and expectations of the general public towards software systems are fundamentally different between pure information processing and embedded systems. With respect to Internet-based multimedia services, people got used to the fact that video streams stall, rebuffer, or switch quality, while such a behaviour would be unacceptable for the traditional TV broadcast service. While people got used to rebooting their computer if something does not work as expected, they would certainly return their car (or insulin pump) if they were supposed to do the same. Therefore, technology has to evolve to fulfil the expectations users have towards embedded systems right from day one—which means that many domains (e.g. manufacturing, avionics) cannot do a partial migration to modern (hard- as well as software) technology while the technology is not yet reliable enough. With CPS, this is even more challenging as the environments they operate in are often harsh and full of noise, interference, and other effects that hinder a reliable real-time communication.

In this spirit, [Lee08] argues that components at all layers should be made as reliable as possible; if infeasible, the next higher layer should compensate and create robustness.

Feasibility Another central argument of [Lee08] is that even though the current software and technology cannot support advanced CPS, it is not a fundamental contradiction that software *similar* to the current one can do so. While the layer below software (i.e. digital circuits) is extremely reliable, software is in many cases not reliable because the employed abstractions do not properly capture timing and reliability constraints. The same holds for networks, which are built on increasingly reliable circuitry and physical modulation schemes, but lose this reliability as soon as they provide best-effort services, like on most of the Internet. Therefore, it is essential to the analysis and implementation of CPS to create good latency-aware models and abstractions[9] that allow to have a high confidence that their proven formal properties hold in the physical realizations.

Further Challenges There are certainly more challenges to CPS than the three that have been mentioned previously. One of them is the *increasing number of devices* which raises the density of systems that must be concurrently operated and controlled. Having

[9][LS17] follows the aphorism "all models are wrong, but some are useful", which is commonly attributed to George Box.

these many devices, which are often also mobile, requires that they have an adequately *low energy demand* that can be fulfilled in a sustainable manner and without regularly imposing the need for charging. The vast number of devices also come with increased integration into human-centered perimeters, which brings two additional challenges. First, extensions of models are needed to incorporate human actions, which is particularly challenging due to the non-linearities *human behaviour* exhibits. Second, maintaining *privacy* is crucial in the presence of an unprecedented number of sensors that continuously measure the surroundings and could enable privacy-violating derivations.

While all these additional challenges are certainly important and must be considered by CPS designers and engineers, they are not the major focus of this thesis. Instead, we focus on interoperable and predictably reliable real-time communication infrastructures that are a building block to CPS that fulfil all of the mentioned requirements.

2.2 Distributed Cyber-Physical Systems

Many of the applications that were mentioned in Section 2.1.1 include means of networking and telecommunication. So even though [Wie48] already considered communication as an essential component of cybernetics, most control designs have simplistic communication models and most network solutions abstract control applications in a simplistic way: as processes that want to communicate. While no control system can exist without communication (i.e. actor, sensor, and controller must exchange information), for most of the time communication has been neglected as it did not require explicit treatment. In local scenarios, with information travelling across short distances and highly reliable media (e.g. wires), the major challenge for CPS has been to compensate for latencies caused by the sensing, acting and control processes themselves, not the communication. In contrast, the control systems that are envisioned today should cover large physical distances and traverse multiple potentially unreliable communication media so that communication aspects are central to fulfilling the control task. As delay and jitter can severely impact the quality of control [BPZ00, BA07, CSL18], their *communication-induced* causes have to be treated carefully in a distributed scenario.

In control engineering, these systems are considered as *networked control systems*, of which we give one definition here (cf. [HVLA+07]):

Definition 2 (Networked Control System) *A Networked Control System (NCS) is a control system in which sensors, actuators, and controllers are connected by means of a network or other shared medium.*

Even though this term emphasizes the role of networking, most of the research about CPS carried out over the first decade after the rise of the term NCS did not put a major focus on the networking challenges—apart from treatment of the lowest layer in the OSI model[10]. Networking research in contrast has been majorly focused on providing high throughput over flexible packet-switched networks, considering *latency* primarily as a performance parameter[11] of the lower layers (physical and link layer), but not as a reliability metric for the transport layer. More details on the recent shift in research from increasing throughput to decreasing latency are going to be presented in Chapter 3.

In order to overcome this, there have recently been a number of endeavours to bring together the disciplines of control, network, and systems engineering at an equal level. First, the priority programme 1914 "Cyber-Physical Networking"[12] has been established, which is funded by the German Science Foundation (DFG) since 2015, with the second project phase starting at the end of 2019 after the first ran from end of 2016 to 2019. Second, the *International Symposium on Networked Cyber-Physical Systems* (NetCPS) took place in September 2016, where several initial ideas on *Cyber-Physical Networking* [SH16a] have been presented, which have been implemented in the aftermath and are described in the remainder of this thesis. Finally, there have been several presentations and workshops on this topic at major networking conferences, hosted by the ACM SIGCOMM and IEEE ComSoc, as well as control engineering communities.

In order to stay in line with the historical development of the term cybernetics (which by definition includes communication), it is therefore canonical to talk about *distributed cyber-physical systems*. This term stresses that the system is either distributed or multiple systems cooperate[13]. In either case, communication and (inter-)networking are essential to be able to control the physical world. For simplicity, the rest of this thesis uses the term CPS, but always having in mind those distributed instantiations where it is essential to have predictably reliable networking solutions to allow robust control.

2.2.1 The Internet as a Universal Communication System

Considering the increasing importance of the networking components in CPS, it is straightforward to look at the Internet—the most universal and widely distributed communication system humankind has created so far. This *universality* has been achieved by a clear separation of concerns and assigning functions to multiple independent but well-defined layers[14] that are stacked together. This approach has been codified in the ISO/OSI model[15], a cornerstone of the Internet's success. Keeping the network core as simple as possible and putting complex applications in the end-systems has been a further contributor to this universality [SRC84]. This way, it became tremendously simple

[10]For instance within the priority programme 1305 "Control Theory of Digitally Networked Dynamic Systems" (https://gepris.dfg.de/gepris/projekt/29058575, accessed September 6, 2019), funded by the DFG, which dealt in particular with wireless networking and the control theory to be developed to handle these dynamic communication systems.

[11]The same hazardous paradigm that was mentioned before with respect to software dynamics.

[12]https://spp1914.de (accessed August 15, 2019)

[13]These alternatives are more philosophical and subject to the bounds we put on a system.

[14]Or "models of models [...] of things" [Lee17].

[15]https://www.itu.int/ITU-T/recommendations/rec.aspx?rec=2820 (accessed April 17, 2019)

to create applications that involve information exchange *on-top* of the Internet. The *wide distribution and scaling* has been enabled by the design goal of the creator's of the Internet to *keep things as simple as possible*—as complexity is considered as a major impediment to scaling [BM02].

In summary, it would be straightforward to conclude that the Internet as it is today is an appropriate solution to connect parts of distributed CPS. The following section shows why it is not (yet) and why it should only be considered as a good starting point to enable distributed CPS that are interoperable. In the spirit of Clark [Cla18], this is a perfectly valid approach to take *the Internet* and gradually transform it into a different internet—a process that is happening while you read this line.

2.2.2 Interoperable CPS using Internet Technologies

The increasing grade of digitalization of all domains of daily life and business, mainly due to some form of symbiosis with the Internet, makes it natural for CPS to also employ this communication network instead of creating a proprietary one. If we consider future applications such as self-driving cars, it is evident that no market player is going to establish a network infrastructure of its own but will instead at some point interface with the Internet[16]. Nevertheless, the access technologies vary, with fibre-optic links, cellular services (e.g. 5G New Radio), local wireless networking (e.g. 802.11p) or low-power wide range networking (e.g. LoRaWAN) representing a highly abridged list. In addition, the approaches for the placement of the computing part of these applications are not yet clear, which is also discussed in the edge and fog computing domains [BMZA12, SCZ$^+$16]. Both the network access technology as well as the placement of computing functions influence the system performance dramatically and can make solutions infeasible if they cannot provide sufficient latency and resilience properties.

In addition to these considerations, RFC3439 [BM02] discusses several aspects of the design philosophy behind the Internet that can become obstacles for CPS:

- While the end-to-end argument [SRC84] demands that there should be no *state* in the network, the actual implementations of today indeed have state (e.g. routes), which has a direct impact on the timing and reliability.

- Large scale systems exhibit *non-linearities*, *amplification effects*, and *resonances* that do not show in small scales—and can be a major impediment when implementing reliable CPS.

- *Synchronization*, which is often essential for predictable operation, can be harmful as faults occur synchronously and lead to amplification or cascading failures.

[16]It should be noted that some non-ISPs (e.g. BMW, BASF) are pushing to get dedicated frequencies by the German Bundesnetzagentur (https://www.spiegel.de/international/business/a-1257798.html, accessed July 24, 2019). Nevertheless, interoperability is key and having physical layer independence does not free from interfacing with the Internet for collaborative applications.

- The (horizontal or vertical) *coupling* is problematic for the same reason.
- *Layering* (at runtime) is considered harmful, as it implies potentially duplicate work and too many layers can lead to higher complexity, which counteracts the simplicity principle.

Therefore we can conclude that as of today, the Internet and most of its technologies are not ready for demanding CPS applications. In particular, networking technologies are missing *time- and resilience-awareness* that are essential for distributed CPS. We expect distributed CPS to be large, complex and require scaling—but also to be interoperable. In consequence of the need for interoperability, the solutions shown in this thesis adhere to the concepts governing Internet design, in particular the E2E principle (cf. Chapters 4 and 6). However, this thesis also incorporates evolutionary approaches that overcome weaknesses caused by strict application of the E2E principle (cf. Chapter 7). In summary, we are confident that it is possible but challenging to provide communication and networking technology for CPS that is reliable, exposes reliable timing, and fosters interoperability.

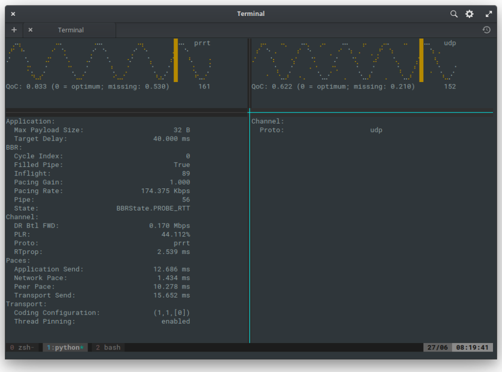
2.3 Reliable Networking Atoms (RNA)

As a first step, the *reliable networking atom*[17] (RNA) has been developed within the LARN project[18] and serves as a suitable component for composing interoperable CPS. The RNA in general comprises two components: (1) a recent Linux version with system-level adaptations to improve performance of the communication stack and (2) a recent version of the PRRT protocol (cf. Chapter 4) to allow process-to-process communication. The RNA comes in several forms and serves as a constrained evaluation platform to empirically test the approaches developed in this thesis in real-life CPS applications.

2.3.1 Sine Control App

In order to evaluate the control performance when using the RNA, the *Sine application* that is depicted in Figure 2.1a has been developed. In this control system, the *plant* works on signals in the form of a three-tuple (t_c, v_c, t_p), where t_c and t_p are the timestamps of the control message at the controller (when it was sent) and the plant (when it was received). The controller derives its actions from a sinus function, so that $v_c = sin(k \cdot t_c)$, and k scales the sine to achieve an appealing visualization. To achieve a visual representation of how good the communication systems works, these samples are plotted at (t_p, v_c) in yellow. White dots resemble the *expected samples* that are derived by knowing the control function at the plant. The figure clearly shows how the PRRT variant can reliably reproduce the sine at the plant, while the UDP variant

[17]http://rna.larn.systems (accessed July 2, 2019)
[18]http://larn.systems (accessed July 2, 2019)



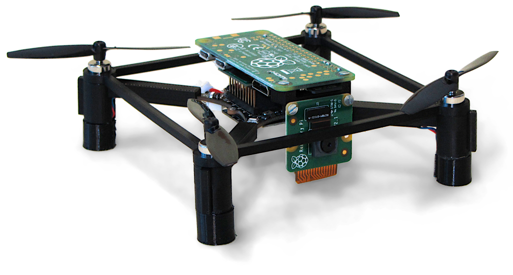
```
prrt
QoC: 0.033 (0 = optimum; missing: 0.530)        161

Application:
  Max Payload Size:                          32 B
  Target Delay:                          40.000 ms
BBR:
  Cycle Index:                                  0
  Filled Pipe:                               True
  Inflight:                                    89
  Pacing Gain:                              1.000
  Pacing Rate:                       174.375 Kbps
  Pipe:                                        56
  State:                      BBRState.PROBE_RTT
Channel:
  DR Btl FWD:                          0.170 Mbps
  PLR:                                    44.112%
  Proto:                                     prrt
  RTprop:                                2.539 ms
Paces:
  Application Send:                     12.686 ms
  Network Pace:                          1.434 ms
  Peer Pace:                            10.278 ms
  Transport Send:                       15.652 ms
Transport:
  Coding Configuration:                 (1,1,[0])
  Thread Pinning:                         enabled
```

```
udp
QoC: 0.622 (0 = optimum; missing: 0.210)        152

Channel:
  Proto:                   udp
```

(a) The *Sine App* resembles a minimalistic control application where the controller transmits a control signal (i.e. the sine) and the plant monitors how reliably and timely this signal arrives.



(b) In its *Drone* form, the RNA can be used to stream mobile video and it is also envisioned to act as a bridge to allow control of the drone via off-the-shelf wireless technologies such as WiFi.

Figure 2.1: The RNA is a building block for distributed CPS.

shows scattering. For this scenario, a synthetic 1 ms delay with 1 ms of jitter is used. The PRRT controller also exhibits various runtime parameters that yield insights into its current configuration as well as channel characteristics—information that is exposed to any application that uses PRRT sockets. This application serves as a minimalistic demonstrator to highlight the latency-awareness and -predictability that can be achieved by PRRT—a essential quality that is needed for communication in CPS.

2.3.2 Drone

A more challenging incarnation of the RNA is the drone depicted in Figure 2.1b. The drone is a highly customized CrazyFlie 2.1 provided by BitCraze[19], which is equipped with a Raspberry Pi Zero W that allows to run the RNA software (on Raspbian) and interface with the drone platform. The drone is used for two scenarios: (a) PRRT is used to reliably stream live video over WLAN from the platform to a pilot, and (b) PRRT over WLAN is used to control the drone—instead of using the proprietary CrazyRadio link. The latter is still under development.

The point here is not to demonstrate that drones can be controlled and video can be streamed, but instead to show how this can be achieved using open, off-the-shelf solutions (such as Linux and WiFi) in combination with a straightforward use of PRRT—all in a reliable and latency-aware manner. This brings the core interoperability design principle of Internet protocols to the highly demanding domain of CPS.

[19]https://www.bitcraze.io/ (accessed July 2, 2019)

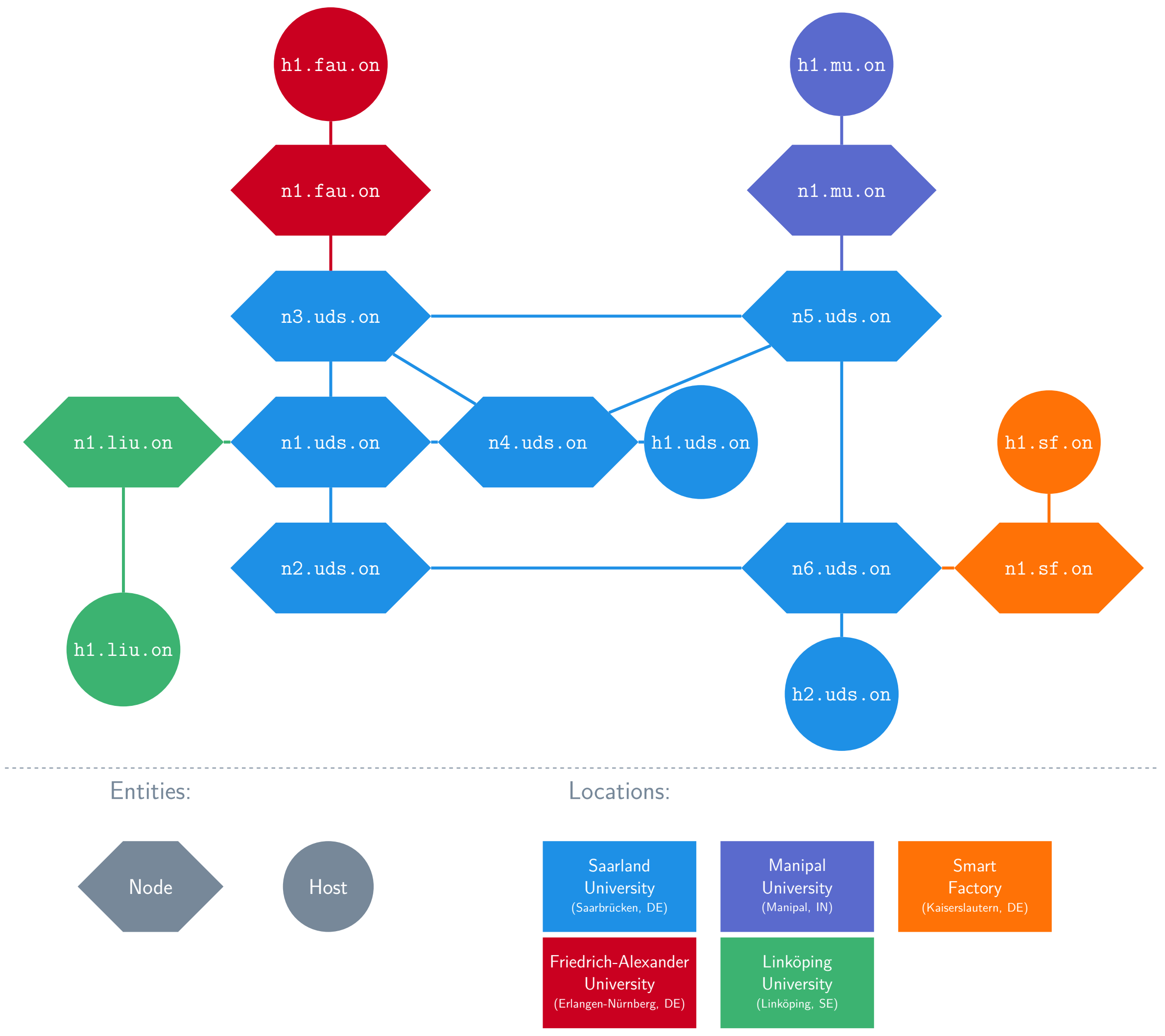

h1.fau.on
h1.mu.on
n1.fau.on
n1.mu.on
n3.uds.on
n5.uds.on
n1.liu.on
n1.uds.on
n4.uds.on
h1.uds.on
h1.sf.on
n2.uds.on
n6.uds.on
n1.sf.on
h1.liu.on
h2.uds.on
Entities:
Node
Host
Locations:
Saarland University (Saarbrücken, DE)
Manipal University (Manipal, IN)
Smart Factory (Kaiserslautern, DE)
Friedrich-Alexander University (Erlangen-Nürnberg, DE)
Linköping University (Linköping, SE)

Figure 2.2: The OpenNetworking testbed at Saarland Informatics Campus spans across multiple remote locations in addition to the core network at `uds`.

2.4 The OpenNetworking Testbed

The evaluations in the *contribution* chapters of this thesis (Chapters 5 to 7) have been executed in the *OpenNetworking Testbed* at Saarland University[20]. Apart from evaluations with respect to networking for CPS, the testbed is also used to investigate approaches in network softwarization, i.e. with software-defined networking (SDN) [KRV+14] and network function virtualization (NFV) [MSG+15]. The testbed's topology can been seen in Figure 2.2 and is composed of the *core* at Saarland University and *remote* sites operated by scientific partners. Regarding the systems inside the network, we distinguish between nodes, hosts, and relays (relays will be covered in Chapter 7), as well as an

[20]https://www.on.uni-saarland.de (accessed May 10, 2019)

SDN controller, which is not depicted as it is part of the control plane.

All systems in the network run Ubuntu 18.04 and employ the Linux kernel 4.15. The nodes have Intel Xeon CPUs with 4 cores and 8 GB of RAM, which is also used to run virtual hosts, e.g. using Docker. The controller, which is located at Saarland University, has 6 cores and 32 GB of RAM.

In order to facilitate the experiments and profit from the SDN approach, the network is divided into three separate planes: *control*, *data*, and *time*.

Control-Plane In SDN, the *control plane* carries traffic between the SDN controller and the individual nodes in the network, which is, for instance, used to control the flow tables in OpenFlow [MAB+08]. The testbed can host different SDN controllers and uses dedicated nodes that come with *OpenvSwitch* as a software-switch implementation. For most non-SDN related scenarios, the switches are operated in *stand-alone mode* that runs STP to avoid loops and create a spanning tree. The control plane connection to remote locations is established via IPsec over the public Internet or the DFN[21] for links between German universities.

Data-Plane The *data plane* is, in our case, separate from the control plane to ensure that there is no interference. The links are 1 Gbit/s twisted pair cables and the NICs are Intel I350T server adapters that support 1GbE. The data link to remote locations is established using L2TP tunnelling and covers the same Internet paths as the control plane, i.e. experiments that run across remote links contend with many third parties.

Time-Plane A recent addition to the testbed is the *time plane* that ensures that the nodes at Saarland Informatics Campus are sufficiently synchronized to a world clock. In particular, a GPS integrated grandmaster clock is used that is able to broadcast IEEE1588/PTPv2[22] messages and synchronize the receiving systems. The switching of the time signal is done with a *PTP transparent clock* and the receiving systems use the hardware timestamping capabilities of their I350T NICs. This enables precise measurements within the core of the testbed, but also opens up opportunities to execute highly distributed experiments where timestamps are taken in relation to the GPS world time.

While the composition of the ON-Testbed does not look like being specifically tailored towards low-latency networks of cyber-physical systems (except for the time-plane), this is indeed intended. The goal of this thesis is not to show that application-specific or special-purpose hard- and software systems are capable of supporting reliable real-time communication in CPS—we are sure they do—but to show what we can achieve today when using off-the-shelf, commercially available hardware and pairing it with unconventional, evolutionary approaches at the transport layer. The major gain of this approach is that we *do not* let go of interoperability and end-to-end semantics,

[21]https://www.dfn.de/ (accessed July 30, 2019)
[22]https://www.eecis.udel.edu/~mills/ptp.html (accessed July 3, 2019)

something that highly customized technology stacks do[23]. Certainly, there are evolutions of hardware and lower-layer software systems required to further push the limits of what kind of CPS can be created. But without evolutions at the upper layers, in particular the network layer, transport layer and the operating system, these advances cannot provide gains to the applications—leading to an extremely capable (and probably costly) fundament whose full potential cannot be leveraged.

2.5 Conclusion

In summary, this thesis is concerned with cyber-physical systems that come with strict timing and reliability constraints. A particular focus is put on distributed and therefore networked CPS or networked control systems, where knowledge of the latency and jitter caused by communication is essential to tell whether the system operates reliably or not. These CPS must be open to allow the interoperability that many application domains demand. The design ideas and technologies of the Internet can serve as a basis, but need adaptation to support CPS. A building block for these interoperable and reliable cyber-physical systems is the RNA, which uses components developed in this thesis. Predictable timing in interoperable systems is evaluated using the OpenNetworking testbed, where transport layer performance is measured using commercial, off-the-shelf solutions.

[23]What we see today in industry (and also *smart home*) applications is a vast number of adapters/converters/bridges which allow isolated networks of devices to interact. Even though these interfaces are necessary to connect various link-layer technologies, these interfaces do not include appropriate notions of time and hence are impediments to predictable end-to-end timing. True support for widely distributed CPS can only be achieved when predictable performance goes hand-in-hand with a sufficient degree of interoperability.

The only reason for time is so everything does not happen at once.

Albert Einstein

Chapter 3

Latency-Awareness and Predictability for Computer Networks

When designing and implementing the systems introduced in Chapter 2, their latency and timing constraints must be considered to achieve reliable operations. This involves a careful look into the theoretical limits of system performance as well as the composition of the end-to-end latency and its relation to transport layer network functions.

3.1 The Networking Triad

On an abstract level, a (networked) system can be characterised by the timeliness, throughput, and resilience it provides:

Timeliness considers the time it takes a system to react or communicate and is quantified in relation to the application that uses the system and has some constraints to be fulfilled. In consequence, a system that has an average latency of 50 ms is not per se more *timely* than a system that has 1 s latency, especially when an application requires 1 ms delay—which both cannot fulfil.

Throughput measures the information exchange and can be quantified in absolute terms, e.g. in bit/s. Depending on the use case, the *utilization* can be used instead, which is unit-less and gives the data rate in relative terms with respect to the maximum available capacity of a given network path.

Resilience measures how robust a communication system is, i.e. how it can cope with faults on the network or system layer, temporary outages, or packet losses. A resilient communication system ensures that its own behaviour does not harm others or itself.

Based on these informal definitions, we claim that a system cannot maximize all three dimensions at the same time (cf. the triad in Figure 3.1). The reasoning behind is that for all systems that attempt to maximize two parameters at the same time, the third is bound to suffer:

- Maximizing *timeliness* and *resilience* requires the usage of forward error correction. To achieve the highest timeliness, this error correction must use proactively sent redundancy data. To achieve a high resilience, there must be a high amount of redundancy, which reduces the *throughput* of the actual application.
- Maximizing *timeliness* and *throughput* prohibits the use of error control mechanisms as they need time (retransmissions, coding) and increase the overall throughput, which is constrained by the channel capacity. Hence, *resilience* is sacrificed.
- Maximizing *resilience* and *throughput* leads to the usage of efficient error control mechanisms that require time for (a) aggregation of data and (b) generation of redundant information. From Shannon [Sha48], we know that perfect resilience can be achieved, given infinite time and hence no timeliness.

This *impossibility of joint maximization* can already be seen when a maximization of two parameters is aimed for. For instance, maximizing resilience involves error control which inherently increased the latency—rendering timeliness maximization impossible. It is therefore a careful engineering effort to balance these demands and design networked CPS that provide performance that is optimal for the application, given these constraints on the simultaneous fulfilment in the three dimensions.

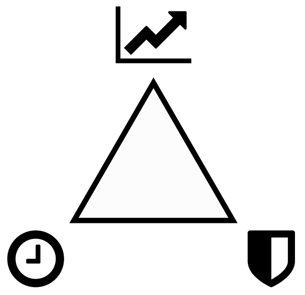


Figure 3.1: The networking triad implies that one cannot achieve optimal throughput, resilience, and timeliness at the same time.

In consequence, we see that some of the advances in high-throughput data networks achieved over the last decades must be sacrificed for implementing CPS that are latency- and resilience-aware. As a side-note, this throughput sacrifice is even done within datacenters when robustness is important [AKEP12]. The following sections separate components of a communication system's latency, allowing to treat them individually and aim for a better *latency-awareness* that can be provided to the application—together with an increased *latency-predictability.*

3.2 The Contributors to End-to-End Latency

In [Che96], a relation is clarified that is often wrongly worded and hence badly understood outside the networking research world—namely that "being fast" is primarily

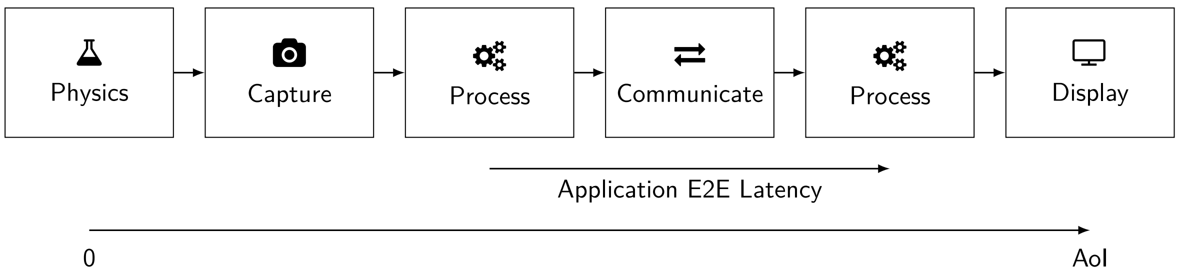


Figure 3.2: Age-of-Information (AoI) describes the time that has elapsed since a physical quantity has been in the state the piece of information indicates.

related to latency and not to bandwidth (or more correctly data rate)[1]. A major insight is that an insufficient data rate can be compensated by adding more capacities (e.g. aggregating two links). This is not possible for inappropriate latency, where fundamentally different approaches are required to optimize it. In its very nature, latency is an additive size as time that has been spent on a specific activity can never be regained. Therefore, the following investigates where these latencies are *caused*, how they can be *avoided*, or how they can be *hidden*[2].

3.2.1 End-to-End Latency

Given the demands that have been discussed in Chapter 2, it is evident that CPS applications require communication that ensures that the received data has a low *age-of-information* (AoI)—a measure that describes the time that has elapsed since a piece of information was *current*. Figure 3.2 depicts how the AoI increases during the process of forwarding and processing in a system. It should be noted that the first step, namely *capturing*, is not instantaneous and there is a lower bound on the AoI that the cyber parts of the system will see. But in contrast to the steps that happen afterwards, the latency caused by capturing is often highly predictable and comparably small.

CPS demand low AoI because outdated data has no more value and is in many cases simply ignored. In addition to the AoI caused by delays in the capturing facilities (e.g. the time between a physical quantity being in a certain state and a sensor taking a sample), the communication system adds the application layer end-to-end (E2E) latency to the AoI.

When it comes to measurement, any application is, in principle, able to quantify how long it takes a message to be delivered and to be acknowledged by taking timestamps. Measuring only the time to deliver a message is also possible, but requires that the sending and receiving systems are sufficiently time-synchronized (or the clocks do not drift and have a systematic synchronization error) to allow correlation of the

[1]More than 20 years later, ISPs still advertise “high-speed broadband” access suggesting short latencies and broad spectrum, but actually delivering high data rates.

[2]Latencies can be hidden to a (human or non-human) user of a system, for instance by executing preparatory work before it is necessary or running independent sub-tasks in parallel and not in sequence.

timestamps. Without this, it is not possible to separate the latency incurred on the forward-trip (delivery) and the backward-trip (feedback).

The E2E latency is the sum of latencies that are involved in the communication process. This latency is composed of the propagation (D_{prop}), transmission (D_{trans}), queuing (D_{queue}), and processing (D_{proc}) latencies, which are all subject to certain dynamics:

$$D_{E2E} = D_{prop} + D_{trans} + D_{queue} + D_{proc} \tag{3.1}$$

The following considers the E2E latency and its summands as random variables with distinct distributions. For each component, investigations around its origin and distribution are made, together with approaches on how to make an application aware of this latency and how to increase this latency's predictability.

3.2.2 Propagation Latency D_{prop}

Fundamentally, transmitting a unit of data involves a physical distance to be covered between the sender and receiver. This travelling—much as any other form of travelling—happens across a route (that is in most cases far from optimal [SCGM14]) and with a certain speed. Therefore, the distribution of the propagation latency D_{prop} depends on the used routes and the speeds along segments on this route. If the segment's medium is shared, access latencies can be involved, e.g. waiting for a well-known timeslot in time-division multiple access (TDMA) or waiting for random backoff in carrier-sense multiple access (CSMA)[3].

Making an application aware of the propagation latency is hard to accomplish, as it must be measured between the physical endpoints of any segment of its route, i.e. it requires correlating and summing up the hardware timestamps taken at the network interface cards. In cellular systems, it is possible to use cross-layer design to bring this piece of information about the physical wireless (last or first) hop to higher layers in the OSI stack. This is not feasible on multi-hop paths as this would require additional network support. The same holds for systems without direct access to information about the physical layer or systems without network cards that have timestamping facilities.

Making the propagation latency more predictable is a task of network operations and of physical communication system design, achieved by keeping routes and link latencies stable, respectively. These tasks are out of scope for this thesis and we instead focus on being *aware* of the propagation latency—independent of it being predictable or not. In consequence, it is going to be assumed that the propagation latency is a stable component with a narrow distribution and a small magnitude. For the *Tactile Internet* [FA14], it would be smaller than 1 ms and hence space-limited to a radius of 300 km).

[3]In this thesis, this latency is treated as a component of D_{prop}, even though it is dependent on the packet length and closely related to D_{trans}. If one creates a cross-layer solution with a stronger focus on lower layers than in this thesis, it is imperative to separately model a medium access latency D_{mac}. For our scenario, D_{mac} either changes D_{prop} by a constant shift (TDMA) or additive jitter (CSMA).

3.2.3 Transmission Latency D_{trans}

The transmission delay D_{trans} is determined by the size of packets and the data rate of the sender's first link. In domains that prioritize low-latency over throughput the packet size is often static, as control applications, for instance, have fixed-length payloads. In contrast, the achievable data rates are not deterministic, e.g. due to wireless interference, mobility, or contention. The distribution of the transmission latency is hence a function of the data rate distribution.

Awareness about the transmission latency can be achieved using cross-layer information, while a prediction relies on appropriate physical layer models. For the sake of this thesis, *delivery rate estimation* [CCYJ17] is providing measurements of the bottleneck data rate, which can be used as a proxy for the data rate and in consequence a conservative upper bound for the transmission latency.

3.2.4 Queueing Latency D_{queue}

The queueing latency D_{queue} quantifies the additional delay a packet faces when it arrives at a busy link. For *store-and-forward* switches, we do not consider the latency that is required to store a packet in a queue, retrieve it and transmit it again on the next link. Instead, the queueing is the latency that is either *self-induced* (i.e. an application with a fast first link transmitting bursts of data that cannot be handled by the bottleneck link) or due to *cross-traffic* (i.e. multiple applications sending more data simultaneously than the bottleneck link can handle). This detrimental effect is due to inappropriate rate and congestion control in combination with large buffers at the system and network layers—commonly referred to as *bufferbloat* [GN12]. While buffers are required to compensate for bursts and fluctuations, their increased number and sizes are detrimental when aiming for low latency. Most importantly, large buffers delay the detection of loss events caused by congestion. The queueing delay distribution depends on the used congestion control algorithm (CCA) and the sizes of the buffers along the routes.

While the first CCAs used loss as a signal for congestion, delay-based congestion control (early examples are e.g. TCP Vegas [BOP94]) evolved, which use RTT-inflation as a signal of congestion and forming queues at the bottleneck. The idea is to build a model of the round-trip latency that is not affected by queueing latency when less than one bandwidth-delay product (BDP) is in flight. *Latency-avoiding* CCAs (cf. [ZMSS19]) respect this bound and aim to not cause significant RTT-inflation over the baseline propagation latency. What we can also deduce from Zarchy et al. (ibid.), is that a latency-avoiding algorithm has certain drawbacks: (a) It is bound to suffer in the presence of CCAs that are loss-based, i.e. they back off only when the bottleneck buffer is filled and congestion has been present for longer. (b) Any algorithm that avoids loss (something that is also the case when aiming to avoid latency) is not able to achieve a high utilization after short flow startup time. In consequence, we must sacrifice highest throughput and utilization to go for low and predictable latency; and we shall not compete with loss-based congestion control algorithms[4].

The rule-of-thumb of having only one BDP in-flight is based on Kleinrock [Kle78, Kle18], who showed that this is indeed the optimal steady-state operation point for deterministic systems, where throughput and latency are optimal at the same time. Stochastic systems, in contrast, cannot achieve the same operating point as deterministic systems due to unpredictability of arrival times (i.e. variation of inter-packet-times) and service times (i.e. processing latency of the transport stack). The optimal operating point for these systems is neither where loss-based TCP is (throughput maximized, but also very high delay) nor where BBR [CCG+16] is. Actually, BBR's claimed operating point is not attainable in stochastic systems according to [Kle18]. For stochastic systems and dynamic parameters, enough headroom to one BDP must be kept in order to achieve low latency with high confidence—but at the cost of throughput, as mentioned in Section 3.1. Kleinrock suggests that an optimal algorithm should track the profile of latency in dependence of data in-flight ($RTprop(N)$) and adapt the operating point ($N = RTprop \cdot DataRate$) to the amount of data in-flight in which the tangent of this curve crosses the origin. Alizadeh et al. formulate a similar idea that "less throughput is more for achieving low latency" [AKEP12] in data center networks and Vulimiri et al. [VGM+13] suggest the use of redundant operations (and hence reduction of throughput) can help to achieve low latency. In Silo [JSBM15], this is achieved using pacing mechanisms to achieve guaranteed latencies for multi-tenant VM setups in data centers. At the network layer, *active queue management* (AQM) solutions, such as *CoDel* [NJ12] provide means to counter the bufferbloat problem.

In Chapter 6, this issue is going to be investigated further and an approach is presented that tackles this issue and leads to predictably low queuing latencies. In conjunction with other measures to make the transport layer more deterministic (cf. Chapter 4), it is also evident that the achievable power curves (as by Kleinrock) get closer to a curve of a deterministic system and better operating points get attainable.

3.2.5 Processing Latency D_{proc}

The processing latency D_{proc} occurs in end-systems and is still considered a major contributor to high E2E latencies as shown by Rumble et al. [ROS+11]. The processing latency in pure forwarding devices is significant (ibid.) and must be reduced by means that are out of scope for this thesis—which again focusses on *being aware* of whatever latency is caused by them. This thesis considers more complex middleboxes as end-systems as they break the E2E paradigm.

Considering end-systems, the layers where processing latencies can occur are the application layer (i.e. user-controlled code), the transport layer (i.e. protocol-related code) and the kernel layer (i.e. operating system code).

[4]Most of the recently developed CCAs (e.g. Copa [AB18]) sense if they are competing with loss-based congestion control and change their strategy to avoid getting a smaller-than-fair share.

Application Layer processing latencies are only a minor focus of this thesis as these are controlled by the user and should also be determined by the user—though the approach in Chapter 6 estimates them and uses this to support just-in-time processing.

Transport Layer processing occurs in conjunction with functions of the transport protocol, e.g. error, congestion, or rate control. Nearly any transport protocol operates in a concurrent manner in which several events can happen at the same time (e.g. feedback is received or a new piece of data is to be sent). This leads to the usage of non-blocking approaches or leveraging multiple processes or threads—in-between which communication has to take place.

Kernel Layer latencies stem from the interaction with and the execution of the operating system, which involves additional processing overhead, e.g. for scheduling or copying as well as moving data.

System noise [TEFK05], as is caused by caching effects, interrupts, or other means, is a major source of processing latencies and in particular low predictability for this portion of the E2E latency. Rumble et al. [ROS+11] also argue that it requires research from the operating systems community to push latencies in the datacenter down to the microsecond regime [BMPR17]. One approach they propose is to move the NIC to the CPU so that compute and communication get closer—achieving datacenter round-trips of about 1 µs.

Awareness of the Processing Latency

Knowing about these sources of processing latencies, the question arises how systems can become aware of these. Tsafrir et al. [TEFK05] propose means to measure and in the next step eliminate OS noise that is caused by the need for synchronization between multiple concurrent processes. The approach X-Lap, which we present in Chapter 5, is a cross-layer solution to measure the processing latencies that occur for each piece of data that travels from the sending to the receiving application. Furthermore, the *cross-layer pacing* approach presented in Chapter 6 uses the processing latencies to tell system-layer bottlenecks apart from network-layer bottlenecks and react appropriately.

Predictable Processing Latency

Several approaches are possible to make the processing latency more predictable. The IX system [BPK+14], a *dataplane operating system*, aims to provide low processing latency using hardware virtualization. IX comes with a native, zero-copy API that processes a bounded patch of packets to completion to amortize context-switch costs and exploit data locality. Central to this system is the division of the OS into two layers, namely the control plane (which takes care of management tasks) and the data plane (which executes the packet processing).

The line of work on *Arrakis* [PA13, PLZ$^+$14, PLZ$^+$15] determined that a major portion of processing latency for networking in Linux comes from "software demultiplexing, security checks, and overhead due to indirection at various layers". A solution to this is implementing *kernel-bypassing*, which means giving direct access to I/O devices through proper virtualization instead of kernel-based multiplexing—providing increased predictability of the latency. Furthermore, copy and other unnecessary operations are eliminated, which helps to achieve an improvement of the absolute latency.

Another approach is qJump [GSG$^+$15], bringing the idea of Internet-QoS to the operating system in the form of a Linux Traffic Control module. At its core, this approach allows traffic with higher latency sensitivity to *jump* over queues filled with lower sensitivity traffic to meet latency bounds.

In the realm of kernel-bypassing solutions, there are frameworks such as DPDK[5] or fd.io[6] that aim to provide low latency. Their performance gains has been validated by solutions using the frameworks, e.g. FastPass [POB$^+$15], mTCP [JWJ$^+$14], or MoonGen [EGR$^+$15] as well as general studies (e.g. [PFLL18]). In Section 7.4.7, these solutions are employed to improve the processing latencies involved into transparent transmission segmentation approaches.

Further, it is possible to use *symbolic execution* methods to derive the performance of network processing units, in particular with respect to the incurred latency. This is leveraged by methodologies, such as SymPerf [RKR$^+$17], that automatically reason about the code and its behaviour under different inputs, i.e. packets.

In consequence, being aware of and predicting a processing latency can help to take it into account for scheduling and synchronizing tasks or when deciding on appropriate error control mechanisms (cf. Chapter 4).

3.3 Network Functions

The unquestioned success of the Internet as a global and universal communication network was only possible with the OSI model[7], whose role in this process is often underrated and the model itself considered as purely academic. History has shown that only by separating communication systems into layers with well-defined—but to some extent also exchangeable—functions has enabled wide interoperability using a planet-scale communication system that is easy to access. Most network functions (e.g. reliable communication or multiplexing in a shared medium) implemented in some of these layers are not new to the Internet, but inherent to all communication systems that have been developed in human history.

As different applications have different requirements and want to rely on certain network functions without needing to implement them themselves, traditions and best practises for placing these functions on certain layers have evolved. A fundamental work

[5]https://dpdk.org/ (accessed May 15, 2019)
[6]https://fd.io/ (accessed June 24, 2019)
[7]https://www.itu.int/ITU-T/recommendations/rec.aspx?rec=2820 (accessed April 17, 2019)

in this area is *End-to-End Arguments in System Design* by Saltzer et al. [SRC84], arguing that a function should be placed on the highest layer that needs it (making lower layers as simple as possible) and on the lowest layer that is used by other layers with this requirement (applying the *don't repeat yourself* (DRY) principle[8]). This work has been questioned since its original release (e.g. by [Moo02]) and practice shows that placement of these functions on different layers is dependent on the performance requirements of the applications using these lower layers (cf. Chapter 7). Generally speaking, the higher the performance requirements are, the likelier a system designer will chose to move a function into the network, migrating from E2E transport layer mode to network layer mode [BKG$^+$01]), and also to smear the abstract interfaces imposed by the OSI model. The latter is referred to as *cross-layer design*, where optimizations are made by exchanging knowledge between and making assumptions about the layers that are adjacent to the layer under consideration (cf. Chapter 6).

This thesis aims to bring the abstract function of *latency-awareness* to the transport layer—providing a service to the application layer that existing transport layer protocols do not. It does so by extensive measuring of lower layers (e.g. estimating round-trip time and delivery rates) and by interaction with the end-host platform (Chapter 5). The protocol that is described in Chapter 4 provides exactly this service and runs on top of any layer that provides end-host addressing and process multiplexing (if it is required to have multiple processes per host). In the following, we revisit the network functions that are provided by existing Internet transport protocols (e.g. TCP [Pos81], SCTP [SRX$^+$04], RTP [SCFJ03], QUIC [Goo18]) and what role latency and the predictability of latency plays with respect to these network functions.

The design of network functions is oriented towards the challenges they should tackle, which includes missing data (e.g. due to unreliable channels or buffer overflows), late data (e.g. due to increased latencies), or insufficient data rate (e.g. due to cross-traffic). The former are control functions that create feedback loops to compensate for these challenges of the network (Sections 3.3.1 to 3.3.3), the latter are management functions that are required to exchange state, handle multiple applications, or consider different link sizes (Sections 3.3.4 to 3.3.7).

3.3.1 Error Control

Error control aims to ensure a certain resilience (at the cost of latency) so that communication can be robust. Central to this function are results from information theory, most notably the Shannon channel theorem [Sha48] and recent results by Polyanski et al. [PPV09]. Conceptually, error control relies on knowledge about the channel's resilience (either through feedback or a priori information) and employs proactive as well as reactive measures.

A joint approach that uses these two mechanisms is *hybrid automated repeat request* (HARQ) [CC84]. The parametrization of this mechanism, i.e. code block dimen-

[8]Often attributed to Hunt and Thomas, authors of *The Pragmatic Programmer* (1999).

sions and retransmission schedule, has been thoroughly studied by Tan [Tan08]. A concrete coding configuration determines the added latency (due to coding, aggregation, and retransmissions) and throughput (for the redundancy information) to achieve a resilience gain (in terms of residual loss rate).

In order to find such a configuration, the application demands (latency, resilience, and throughput) must be taken into account as well as the current channel conditions [Gor12]. An implementation of error control for a latency-aware and -predictable transport is given in Section 4.5. Finally, the process of finding such a configuration takes time and, depending on the system design, might interfere with the actual transmission process.

3.3.2 Flow and Congestion Control

Both flow and congestion control define the maximum amount of data in flight—in TCP via the `cwnd` sender-side variable and the `rwnd` header field. If this value is not appropriate for the current path (e.g. due to a small receiver buffer, connection startup, cross-traffic, ...) the system cannot achieve the highest possible data rate. This in turn leads to increased idle latencies due to waiting for ACKs to advance the sender's window and increases the overall time to complete a flow.

For congestion control, this could be due to an algorithm not reacting appropriately to physical losses (i.e. treating them as congestion signals and hence reducing the cwnd) or not fully and fairly utilizing the available data rate by having the steady-state-optimum of one BDP in flight [Kle18]. For flow control, an insufficient receiver buffer is an effective limit for the achievable throughput (c.f. Section 7.3.2). The reason behind that is that it takes at least one RTT for the receiver to signal empty buffer space and for the sender to provide new data. In consequence, a receiver can at maximum signal to have a full receiver buffer available, i.e. the maximum achievable throughput is the buffer size divided by the RTT.

State of the Art in Congestion Control Algorithms

Solving the bufferbloat issue [GN12] is necessary to allow low-latency applications, and it is clear that congestion control algorithms that solve this must become latency-avoiding. The latest standardized state of TCP congestion control on the Internet can be found in RFC5681 [APB09], which defines a control algorithm that is bound to fill buffers until they are completely filled and congestion losses occur. In the research community, several novel approaches have evolved in the last years and aim to provide low-latency.

BBR [CCG+16] is an approach that builds a model of the bottleneck-bandwidth and round-trip time (hence the abbreviation) and uses this to follow the rule by Kleinrock [Kle18] to have at most one BDP of data in-flight. The original design of BBR included several weaknesses that have been pointed out by several authors (including [HBZ17, ALH+18, ZMRP19]), including proposals for fixing them. Also, the Google team developing BBR has proposed several changes[9] that are gradually integrated into the kernel implementation. BBR changes only the sender-side of the TCP connection by adding delivery rate estimation as well as packet pacing.

Performance-oriented congestion control (PCC) [DLZ+15], which is also a purely sender-based modification, controls its congestion window by carrying out experiments. This means that the algorithm regularly tests certain actions (i.e. increase or reduce rate) and pairs them with the resulting performance in terms of a user-defined *utility* to serve as a base-line for future decisions—performing a form of online learning.

TCP LoLa [HNZB17] aims to keep buffers at a low, non-zero value. Thereto, it implements feedback control where a certain target queuing delay (and hence buffer fill-level) is to be met.

Copa [AB18] uses an objective function on utility that can be tweaked to favour delay over throughput and vice versa using the parameter δ. This function incorporates the estimated queueing delay and estimated rate. Using samples of the current queueing delay d_q, a target rate λ_t is computed as $1/(\delta \cdot d_q)$ and actions are taken in order to move the current estimated rate λ (and `cwnd`) towards the target rate.

With this large number of congestion control algorithms in existence and still in development, the *Pantheon* [YMH+18] has been developed to act as a competition ground where different implementations are executed in a reproducible manner.

In [PWSB11], several challenges for congestion control are pointed out that are going to be considered in the remainder of this thesis. In particular, link heterogeneity and resulting dynamics lead to a situation where none of the current CCAs is performing best across all of them. The solution that is presented in Chapter 7 also allows to choose a CCA on a link granularity across a path—leading to what the authors of the RFC call *multi-domain congestion control*.

Lastly, for effective congestion control, it is evident that network support must also be considered, e.g. through AQM, explicit congestion notification (ECN), or other advanced techniques as are developed in the *L4S* project[10].

3.3.3 Rate Control

While congestion control limits the amount of data in-flight to never occupy more than the path can transmit, it is not sufficient to achieve lowest latency, especially when the bottleneck data rate is not on the first hop from sender to receiver. A first hop that allows a high data rate can create bursts of data that are bound to form a queue at the bottleneck link, as the packets cannot be forwarded fast enough. These enqueued packets experience increased E2E latencies, which is why recent practical efforts [CCG+16, JSBM15, AB18] as well as theoretical considerations [ZMSS19] consider *latency-avoiding* congestion control as well as *pacing* mechanisms for rate control. The

[9]https://datatracker.ietf.org/doc/slides-105-iccrg-bbr-v2-a-model-based-congestion-control/ (accessed August 15, 2019)

[10]https://riteproject.eu/dctth/ (accessed June 22, 2019)

protocol that is presented in Chapter 4 implements such an approach by also incorporating the system layer, leading to rate control via *cross-layer pacing* (Chapter 6).

3.3.4 Fragmentation

Each physical link comes with a maximum transmission unit (MTU) that is defined by its medium as well as the employed medium access technique. The distribution of these MTUs across a path can lead to a situation where large packets must be first fragmented and later reassembled again at the end of the path. The splitting of a packet induces additional processing latency at middle-boxes in the network, while reassembly induces both processing latency as well as aggregation latency at the receiver. Therefore, avoiding fragmentation can avoid latencies and in particular reduce jitter, because inter-packet-times can get skewed when a packet gets fragmented.

3.3.5 Multiplexing

Generally speaking, the multiplexing of multiple users or processes can be done in two ways: using *fixed* or *flexible* schemes. With fixed multiplexing schemes, e.g. TDMA, a process must wait for its slot to transmit something. In this situation, a worst-case bound for the latency can be given—which is the latency between two slots. For flexible (or best-effort) schemes, congestion avoidance and control come into play and it is hard to give bounds on the incurred latency. In summary, there is always going to be additional latency induced when multiple parties share a communication system.

3.3.6 Addressing and Naming

Apart from the functions mentioned before, it is also necessary to discover and address other participants—two processes that incur additional latency. If we consider protocols such as the *address-resolution protocol* (ARP) [Plu82] and the *domain name system* (DNS) [Moc87], it is obvious that (in addition to the connection setup mentioned later) retrieval of meta-data is necessary before a transmission can start. For this reason, many system designs attempt to operate in a multicast or broadcast mode that avoids direct addressing. In combination with *publish and subscribe mechanisms*, a communicating device does not need to know to whom to talk and can start sending and receiving immediately as soon as it is connected to its *message broker*.

3.3.7 Connection Setup and Startup

In particular in highly flexible communication systems where state must be kept or communication parameters must be agreed upon, it is often necessary to setup a connection in prior to the actual communication—a process that induces additional latency. Considering systems that require security, this step is also necessary to admit the communication or ensure the proper identity of all participants. If the communication channel has an elastic data rate, it might also be necessary to probe for the available data rate (i.e. via congestion control), so the setup is followed by a startup phase before we can continuously exchange data at high rates.

In consequence, many system designs attempt to operate in a connection-less mode using e.g. multicast or broadcast mechanisms to avoid the need for direct addressing. At this level, security must be ensured by a priori sharing of cryptographic information—something that is not always practical. Depending on the communication patterns, it could be possible that an association or subscription replaces the individual connection setup, so that everything is in place when actual information packets must be transmitted. Also, the available portion of data rate must be known at this point (cf. open challenges in RFC6077 [PWSB11]).

The problem of increased latency due to a long handshake (in particular for short-lived flows) has been studied by the Internet community for quite some time (e.g. [Mog95]). Solutions have to consider security and deployability issues, as the handshake fulfils a purpose that cannot simply be removed. There are different solutions for TCP, e.g. TCP Fast Open [RCC+11], where a security cookie is issued some time before the actual flow starts and then presented together with the initial data upon the first packet of the flow. Zero-RTT handshakes are one of the major features of QUIC [Goo18, CDCM15] and there are additional mechanisms to shorten the startup times until sufficient data rate is available to the flow [RWS+19].

In summary, systems that aim for lowest latency should make sure that either: (a) connections are not needed because all information that would require a handshake is exchanged a priori to the actual (time-critical) transmission or (b) the connection setup has a low and predictable latency that is adequate for the use case.

3.4 Becoming Latency-aware and -predictable

In consequence, it is essential for distributed cyber-physical systems to be aware of latencies that occur at runtime. Being aware of what causes these latencies and how they are distributed enables to avoid them in the first place or accurately predict them if they cannot be avoided. How this can be achieved within the various network control functions that are part of transport protocols is going to be the focus of the next chapters and is the major contribution of this thesis.

It is difficult to make predictions, especially about the future.

Danish Proverb

Chapter 4

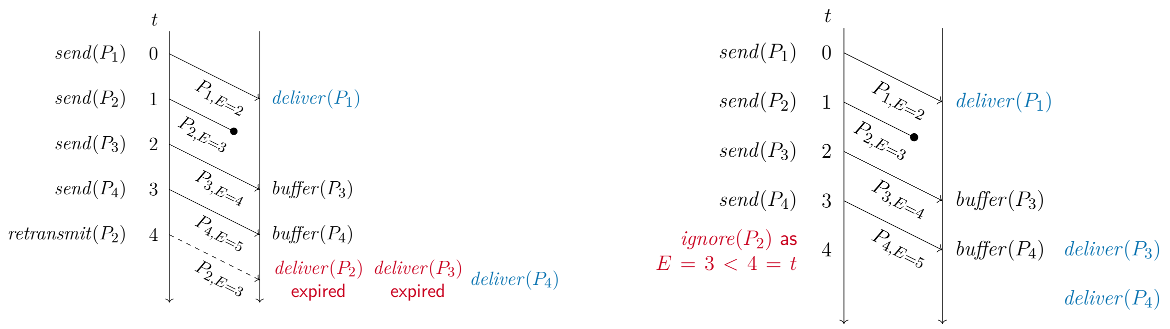
Predictably Reliable Real-time Transport

Considering existing transport layer protocols, there is a lack of latency-awareness and latency-predictability even though many sources [Lee08, DB13] argue that this is required for CPS and large-scale cloud systems. For instance, TCP [Pos81], as a fully reliable, ordered byte-stream protocol, has no capabilities to express timing requirements, e.g. an expiry date for information. As soon as latency-awareness is present, various transport layer functions can be modified in order to take latency into account. This chapter introduces an alternative transport protocol, namely PRRT, to support these specific needs of applications in distributed CPS that were described in Chapter 2. An intensive analysis of PRRT's timing characteristics is given in Chapter 5 and one of its advanced features is described in Chapter 6.

4.1 The Origins of the PRRT Protocol

The original motivation behind the design of an alternative transport protocol to TCP has been that *multimedia* applications have constraints (i.e. maximum tolerable loss, maximum tolerable latency) that are not as in traditional file-transfer or web applications (i.e. 0 % loss and indefinite latency).

Following Shannon's channel coding theorem [Sha48, Fei54], one can deduce that any protocol with full reliability (i.e. an residual error of 0 %) demands an infinite blocklength that corresponds to infinite time. Therefore, it was envisioned to use an application's *tolerable loss* as a loss budget that can be used for making statements about latency—both theoretically and practically. Certain applications also have a *tolerable latency* that has two effects: (a) This delay budget can be used for increasing reliability through error control, if the current achievable end-to-end (E2E) latency is below this. (b) If the current E2E latency is above this tolerable latency, messages lose their value (i.e. they *expire*) and all components of the transmission system can stop forwarding this piece of information, thereby allowing succeeding messages to reach in time.



(a) Full Reliability (b) Partial Reliability

Figure 4.1: Exemplary transmission of time-critical data (expiry time indicated by $P_{E=?}$) experiencing loss, leading to different behaviour depending on the implemented mode of reliability. With partial reliability, data that is buffered out of order does not expire and packets that cannot make it in time are not resent.

Figure 4.1 depicts the difference between fully reliable protocols (e.g. TCP) and partially reliable protocols (such as PRRT) when dealing with *time-critical* data with limited validity time. The first transmission of packet P_2 is lost in both situations and would be retransmitted after three time units, i.e. too late to make it in time. In Figure 4.1a, the succeeding packet P_3 is buffered, but cannot be delivered as full-reliability enforces order so that P_2 must be received before. As the retransmission of P_2 takes longer than the validity period of P_3, the latter expires before getting to the application. Note that an implementation might still forward the packet to the application, but it has no value anymore. In contrast, in Figure 4.1b, P_3 is delivered when it is due, so that the application can work on it while it is still valid. When P_2 is to be retransmitted, it is no longer relevant and is ignored.

Based on this idea of *partial reliability and bounded latency*, Tan [Tan08] developed a mathematical framework around *Hybrid Automated Repeat reQuest* (HARQ) (cf. [LC04]), a generalization of the two traditional approaches of proactive (FEC) and reactive error coding (ARQ). The framework is used to analyse HARQ parameters and optimize them with respect to transmission of minimal redundancy information while fulfilling the application constraints. The next step was to extend these theoretical findings about the steady-state of a channel in two ways: (a) an implementation of an HARQ parameter search and (b) doing this in an adaptive way. The latter is achieved by making the search efficient enough to be carried out in real-time, which in our case means that a search yields a result within less than the coherence time of the channel. This means that the search inputs are still valid and the resulting configuration is still feasible as well as efficient. These advancements have been done by Gorius [Gor12] through the implementation of the initial *Predictably Reliable Real-time Transport* (PRRT) protocol. This prototype implementation of PRRT is running in kernel-space and is tightly integrated into the Linux kernel and its concurrency and scheduling features.

In the meantime, research and practice in the area of transport layer network protocols have shifted towards the usage of (a) user-space implementations that overcome deployment issues[1] and (b) kernel-bypass technologies to circumvent the performance penalties induced by user-space implementations (e.g. in FastPass [POB$^+$15]). As a side effect, kernel-bypassing and the associated features of network interface controllers enable the exploitation of cross-layer information [PBA17]. In particular the need for user-space implementations led to a shift of the development focus from the PRRT kernel module to the start of an independent user-space re-implementation.

A further reason for this greenfield development, which is presented in this thesis, has been the idea to broaden the scope of PRRT's applicability from pure multimedia-oriented application towards other applications with the same characteristics (i.e. maximum tolerable loss, maximum tolerable latency), in particular applications of CPS. This involved a reconsideration of assumptions made during the original design of PRRT that manifested in the resulting implementation.

4.2 Design Principles for a CPS-enabled PRRT

In consequence, it is the goal of this thesis to provide a latency-aware and -predictable communication system for process-to-process communication with the following constraints and dependencies on other layers:

- No timing, reliability, or throughput information or guarantees are needed by PRRT per se. Nevertheless, PRRT can only use "what it gets", so if the lower layer cannot fulfil (e.g. the latency) constraints of the application, the application MUST adapt and can be notified about this circumstance by PRRT.

- PRRT only requires addressing and multiplexing to be done by the lower layer (e.g., UDP/IP, Ethernet, or industrial field buses).

The reasoning behind is that any network has its way to identify communicating processes (e.g. industrial buses, Ethernet, Wifi, LTE) and hence it should neither be part of PRRT nor should PRRT have specific needs. These channels established between the processes have varying parameters that the transport layer should be aware of and cope with to ensure reliability and latency constraints of the application.

4.2.1 Applications

As PRRT had been designed for video transport, it has been necessary to rethink many of the design decisions [SH16a] that were made during its first implementation. Thereto, two major application domains have been identified: *live-multimedia applications* and *control applications.*

[1]For instance QUIC [Goo18] which is in the progress of being standardized by the IETF [IET18].

Live-Multimedia Applications There are various domains in which the real-time streaming of multimedia content is required, e.g. for entertainment or education purposes, as well as for ensuring safety in industrial monitoring or traffic surveillance scenarios. Recently, advances in the field of computer vision have led to scenarios where closed-loop feedback control incorporates visual data as raw sensor input and extracts several features from it. Considering the traditional inverted pendulum scenario from control, one would replace the position, speed and angle sensors with a camera that provides data from which these samples are derived. In general, multimedia content comes with a strict temporal order of messages, i.e. if a video frame is to be played, earlier (skipped) frames are no longer of interest. Furthermore, the user has an upper bound on the latency that can be tolerated until she considers the service as useless, e.g. because she is the last to hear about a scored goal or because she cannot react quick enough to stabilize a pendulum. Apart from that, there is also a tolerable unreliability, which is due to how coding schemes and in particular decoder implementations can cope with missing frames, e.g. using error concealment.

Control Applications In addition to the previously mentioned video-based control applications, any networked closed-loop feedback system exposes strict time constraints. Typically, controllers are designed in a way that a tolerable worst-case round-trip time is considered [GSC11]. This class of systems includes, for instance, process control in various industries as well as coordinated and collaborating vehicles (e.g. truck platooning [AAGJ10]). In various scenarios, control systems are based on *Markov decision processes* (MDP)[2], so the same applies as in live-multimedia: a new message is making any older message completely obsolete as the process model is memoryless. In case of learning a system's behaviour through model extraction or when a system has memory that cannot be modelled in a MDP, it can be crucial to reliably (re-)transmit or reconstruct messages that have been generated earlier than the most current received one. While MDP-based control systems cannot leverage all of PRRT's features (e.g. retransmissions), functions such as congestion control are still relevant for reliable operation. The suitability of PRRT for wireless gain scheduled control applications have been shown through evaluations in [GGG$^+$19].

4.2.2 Required Transport Layer Features

Both broad application categories demand that the transport layer provides additional features than pure process-to-process message exchange. First, *error control* is required to compensate for losses, hence increasing the reliability at the expense of increased latency. Second, *congestion control* avoids congestion losses and increased latencies that are either queueing- or loss-induced. Congestion control also probes for the available bottleneck data rate. Finally, *rate and flow control* avoid buffer overflows at (a) the

[2]Markov decision processes [Bel57] are stochastic, memoryless processes using discrete time.

network systems (which are caused by bursts) and (b) the receiving system (which are caused by insufficient buffer draining).

4.2.3 Availability and Deployability

Following the ideas of kernel-bypass networks and QUIC [Goo18], we have been developing a new protocol version as a library that can be used directly by any user-space application and is available as an open source project [Tel19a]. There are only dependencies on glibc and POSIX threads which allows porting the protocol to different platforms. The protocol core is implemented in *plain C*, compiled with the *gnu11* compiler, and can be installed on a system as a static or shared library.

The API follows the POSIX socket terminology as close as possible (allowing a straightforward usage) and adds additional configuration mechanisms (in comparison to a UDP socket) that are required to leverage PRRT's additional features. During the development of PRRT-based applications for demonstration and evaluation, an additional API was created for Python using the Cython framework [Cyt18]. The Python-API aims to expose the full set of PRRT features using a *pythonic* [Mar05] interface, e.g. a Python `PrrtSocket` object is created as opposed to C-style file descriptor handling.

The project code comes with a set of tests that allow to check some of the functional, memory, and concurrency related aspects of PRRT. There is also a proof-of-concept Gstreamer[Gst18] plugin, called `gst-prrt` [Tel18], that allows to use and evaluate PRRT with different types of multimedia processing pipelines.

Currently, PRRT works on the transport layer (on top of UDP) to use it within IP networks, but the requirements towards the lower layer are minimal, i.e. PRRT could also be used on top of Ethernet or other MAC protocols directly. The `send_to_socket()` functional unit is encapsulated, i.e. minimal effort is required to replace this UDP implementation with, for instance, a SocketCAN [Lin18] implementation.

4.3 Protocol Specification

PRRT packets are carried within their underlying data unit and have no direct interaction with it. In IP-based networks, it is advisable to use UDP as a minimal-cost process-multiplexing service underneath PRRT. PRRT does not segment the incoming data and keeps the packet boundaries intact. Therefore, it is strongly recommended that the PRRT packet (i.e. application payload plus PRRT header) fits into one physical-layer MTU so that the coding blocks (cf. Section 4.5) have a 1:1 relationship to packets on the medium and hence to individual losses.

Timestamps in PRRT are measured with respect to a PRRT clock that works at 1 µs precision and has a 32 bit resolution, i.e. a wraparound happens every 71.58 min. For the applications we address, the absolute current world time is not essential, but the relative time differences should have a high resolution. Sequence numbers use a 16 bit field. Given that the applications that we target are unlikely to have an inter-

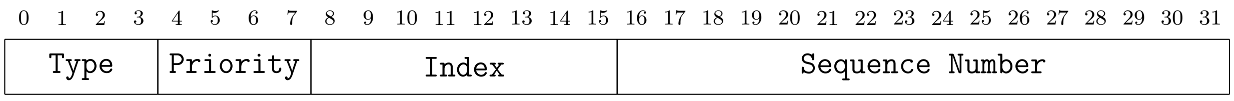


Figure 4.2: PRRT Header Format

packet-interval of less that 1 µs, the sequence numbers are unique in a window of half an hour. For the real-time applications we target, both the timestamp and sequence number spaces are adequate.

The general PRRT packet header[3] is depicted in Figure 4.2. The `Type` field indicates whether a *data*, *redundancy*, or *feedback* packet is sent. The `Priority` is the second half of the first byte and is currently unused. The `Index` field specifies the position of the (data or redundancy) packet within the error coding block. Lastly, the sequence number allows to tell packets apart. It should be noted that a PRRT packet is identified by its sequence number and the type in order to, e.g., make statements about gaps in reception. This avoids the well-known problem in TCP where retransmissions are identified by the same sequence number [KP87]. In general, PRRT can be used for bidirectional communication, but it treats both directions independently. That means that when talking about senders and receivers in the following, we refer to a single communication direction. Sequence numbers, channel measurements, and other values are assigned or measured with respect to this single direction.

After the general header there are additional, packet-type-specific headers:

Data Packets encapsulate the application messages and incorporate metadata that is relevant for the receiving system. The header, as depicted in Figure 4.3, includes the PRRT `Timestamp` at which the packet was generated as well as the `Packet Timeout` at which it will expire. This timeout has been computed by the sender by adding the tolerable latency to the timestamp at which the packet was accepted from the application. In addition, the sender side bottleneck pace is communicated, together with the estimated bottleneck data rate and propagation delay on the link towards the receiver. The data payload starts immediately after the last header field.

Redundancy Packets are sent in case hybrid error control is used. They can be sent proactively (FEC) as well as reactively (ARQ), which is not indicated by the header but defined by the moment in time they are sent. These packets also carry current samples of the bottleneck pace to achieve a higher feedback rate for the pacing. As PRRT uses a block code (cf. Section 4.5) and the sizes can change between blocks, the redundancy packets of a block include the `Base Sequence Number` (from the data packet sequence number space) that started the block. `n` and `k` give the overall dimensions of the block

[3]As PRRT is a protocol under development, this listing reflects the most recent state (v0.4.0 as of September 6, 2019), which can be found here: https://git.nt.uni-saarland.de/LARN/PRRT/tree/f463a13086665594e6c103b03c9d98fce1e4424d

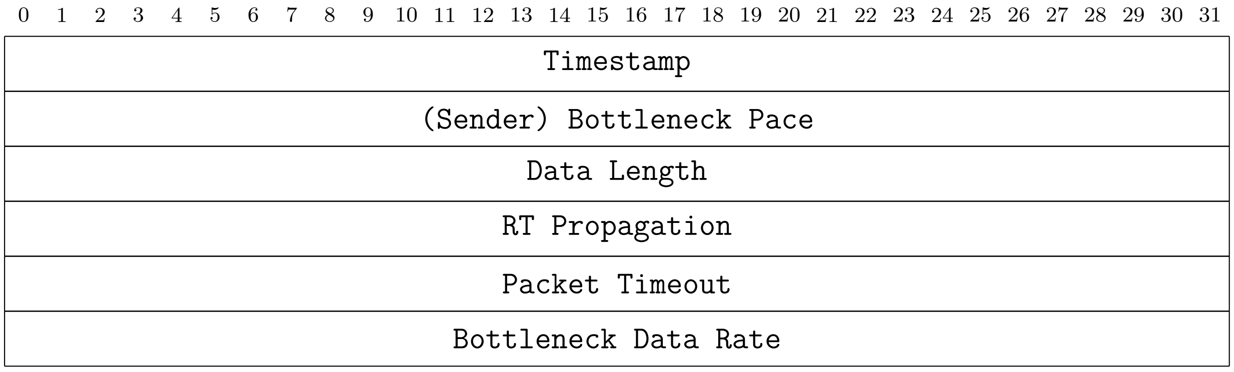


Figure 4.3: PRRT Data Header Format

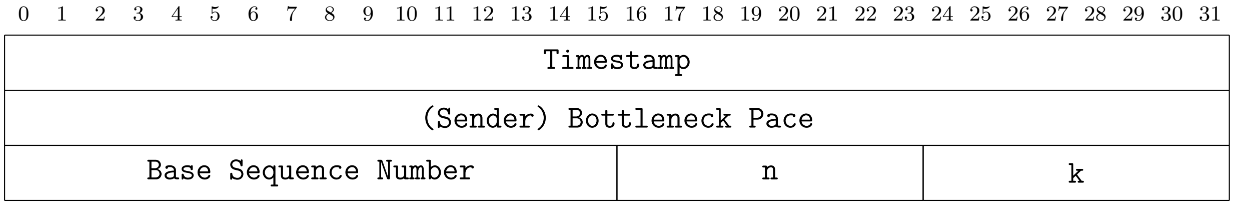


Figure 4.4: PRRT Redundancy Header Format

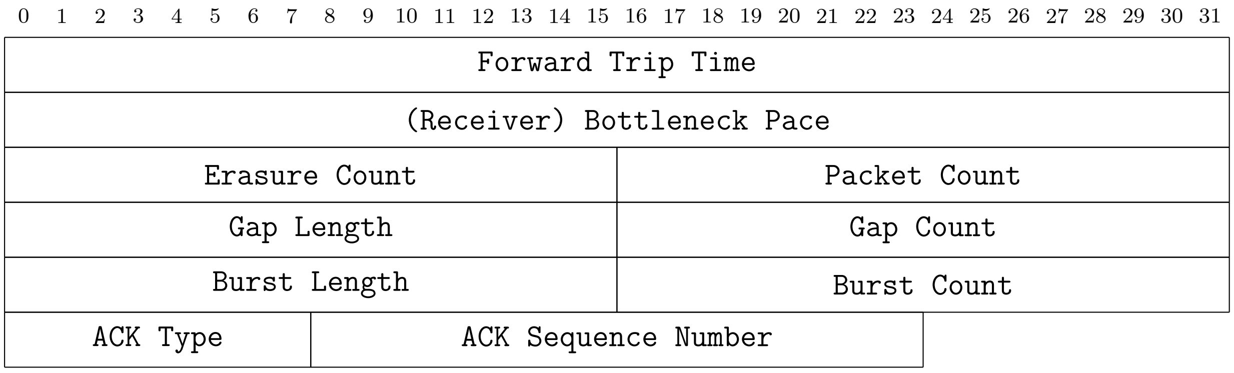


Figure 4.5: PRRT Feedback Header Format

and the portion of data and redundancy blocks (the latter is $n - k$). The redundancy payload also starts immediately after the last header field.

Feedback Packets allow the receiver to transmit channel state measurements to the sender to optimize the sender's operation. These are the propagation delay estimation, the receiver's bottleneck pace, and several fields to compute the loss rate and correlation. As a feedback packet can be sent for data and redundancy packets, `ACK Type` indicates which type was ACKed and `ACK Sequence Number` gives the respective number.

The project code also comes with a *Wireshark dissector* [Wir18], allowing to debug the network behaviour of the currently used PRRT protocol version.

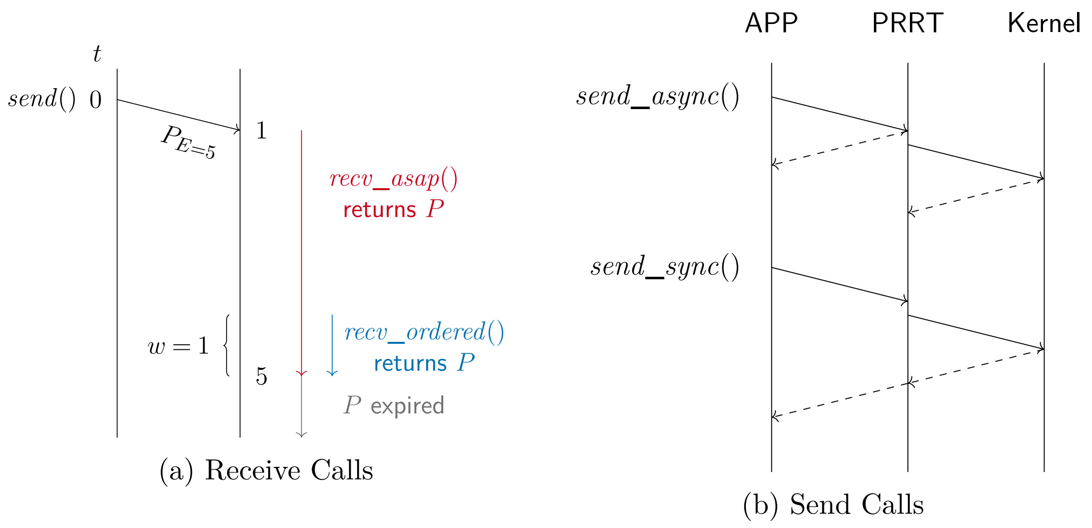


Figure 4.6: The PRRT API supports multiple `send` and `recv` calls that allow to fulfil the demands of different applications with respect to timing behaviour.

4.4 Application Programmable Interface (API)

The API aims to be in line with the way that most transport layer APIs work. Due to the distinct features of PRRT, this is not always possible or desired[4]. This section describes ways a PRRT socket can be used or queried for information that is not found in other protocols.

Receive Modes and Calls As mentioned before, flows of data can have a strict temporal relation between packets, e.g. in a live video stream, where a packet with a frame older that the one currently displayed to the user is irrelevant. Additionally, they have a point in time when they are considered relevant, so they only become useful to the application in a short window before they expire (e.g. the decoding time stamp that indicates the latest point a time a frame can still "make it to the display"). This is not the case for all applications and some applications might have more complex conditions and strategies for judging the relevance of a message. To cater both types of applications, PRRT supports two distinct receive calls types: *ASAP* and *ordered*.

The ASAP calls ensure that data is delivered as fast as possible to the application after it was received from the network interface controller (see red line in Figure 4.6a). After delivery, PRRT has no more means to enforce expiry and reordering.

In contrast, the ordered call type delivers data destined for the current moment in time, independent of when it was received by the network interface controller. This is achieved by initially specifying a *tolerable delay* that determines the maximum latency a packet should face between the `send()` call at the sender's and `recv()` call at the

[4]Future PRRT versions should consider the TAPS architecture that is in the process of standardization at the IETF (https://datatracker.ietf.org/doc/draft-ietf-taps-arch/ (accessed October 2, 2019)).

receiver's application layer. After this time has passed, a packet is considered expired and is no longer forwarded by any layer of the stack. The ordered receive calls also take a *time window* (w in Figure 4.6a) parameter that specifies how close to the deadline a packet should be considered to be due "now". The choice of this parameter influences the amount of residual reordering[5], in a way that the chance of reordering increases with the size of the window. In general, the minimal time between sending of packets at the sender application and the minimal reception time at the receiver application must be taken into account when specifying this parameter.

Send Calls Applications have different requirements towards the behaviour of a send call in terms of their blocking behaviour and the completed transmission steps before they return (cf. Figure 4.6b).

The `send_async()` call returns as soon as PRRT has accepted the packet to be transmitted. This *accepted but not yet transmitted* queue can be limited and if it is full, this call blocks. `try_send_async()` returns an error message when `send_async()` would have blocked due to a full queue.

The `send_sync()` call returns as soon as PRRT has encapsulated the payload in a PRRT data packet and forwarded it to the kernel's UDP socket. In case *pacing* is used, continuously calling `send_sync()` results in periodic behaviour with a frequency that is proportional to the bottleneck data rate and the packet size ($f = \frac{\textit{bottleneck data rate}}{\textit{packet size}}$). This is further discussed in Chapter 6 where the cross-layer pacing approach is explained in more detail and it is shown why `send_sync()` is essential for incorporating the application into the pacing process.

Additional Features PRRT exposes channel measurements, which is in line with the design of *Copa* [AB18], where it is highlighted that exposing throughput measurements from the transport layer to the application is essential for applications that require this information (e.g. in dynamic adaptive streaming). Running in user-space, timestamps related to PRRT protocol events (e.g. packet reception) include a portion of processing delay imposed by Linux. PRRT sockets can operate in *hardware timestamping* mode, where the packet timestamps are taken by the network interface controller and forwarded together with the packet to the higher layers. Latency evaluations (cf. Chapter 5) require the use of *thread-pinning*[6] to allow correlation of processor-cycle stamps taken during a processing step of one thread. As this is also a means to reduce unpredictable overhead imposed by the kernel's scheduler, applications can choose to use this feature for reduced runtime [KOWT11] and increased timing predictability.

[5]Residual reordering refers to the following phenomenon: A packet with sequence number n could be delivered as it is close to the deadline. After this was done, the packet with sequence number $n-1$ arrives, which is still valid for some portion of the time window, and gets delivered—out of order.

[6]Thread-pinning is binding a specific process to a specific core, which reduces the (potentially harmful) impact of the operating system's scheduler.

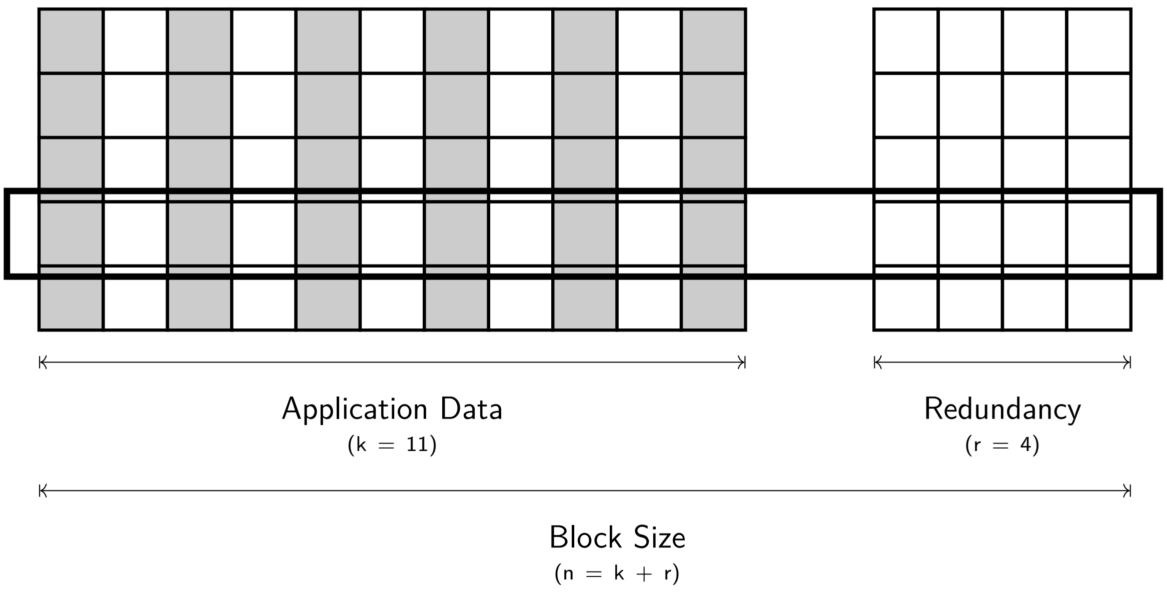


Figure 4.7: PRRT uses block coding, where packets are treated as columns, parity bytes are computed along a row, and afterwards recomposed into redundancy packets (reproduced from [Gor12]).

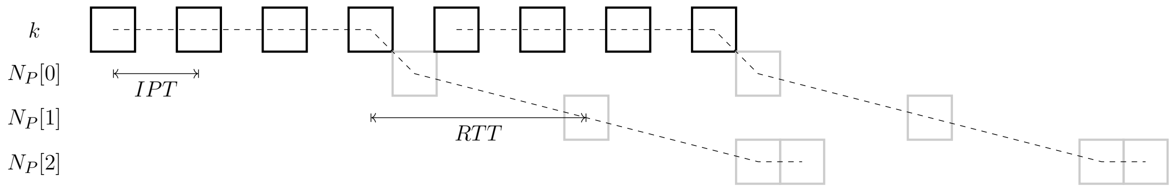


Figure 4.8: Transmissions are grouped into blocks of k (4 in this case) proactive and N_P ($[1,1,2]$ in this case) reactive transmissions. Transmissions belonging to a block are marked by a line. Application packets arrive with an inter-packet-time IPT and it takes one RTT between retransmission rounds (reproduced from [Gor12]).

4.5 Implementation Details

PRRT's implementation has a set of unique features that allow it to implement latency-awareness and -predictability while at the same time ensuring reliability and making it a ready-to-use transport layer implementation.

Error Control

At the core of PRRT's HARQ scheme [Tan08] is a block code as depicted in Figure 4.7. Each column represents a packet and rows are used to compute redundancy bytes based on data bytes from the application data packets. The code works in a way that any combination of k unique packets (data or redundancy) can be used to compute all remaining $n-k$ packets. For latency reasons, it is beneficial if *data* packets arrive, as they are forwarded immediately, but if they are missing, an equal number of redundancy packets allows to recompute them as soon as a total of k packets have arrived.

The coding of single rows is done using Reed-Solomon coding with a Vandermonde matrix. In principle, packets can have different lengths, so the redundancy packets always have the length of the longest packet in a block. Shorter packets are padded with zeros to allow the coding to work on a full row. Nevertheless, it is uncommon in both control and multimedia communication that packets have variable length. Control applications typically have a distinct packet composition that encapsulate the action parameters (e.g. speeds or temperatures) with well-known precision, i.e. the sizes are fixed and known. For video transport, many coding schemes have fixed size units, e.g. MPEG transport stream, so these packets also have a fixed size.

During operation, PRRT always keeps a current coding configuration (k,n,N_P) where k is the number of application data packets, n the overall block size and N_P the *retransmission schedule*, with $k,n \in \mathbb{N}$, and $N_P \in \mathbb{N}^{N_{c,max}+1}$. $N_{c,max}$ is the maximum number of possible retransmission rounds which is per definition arbitrary but fixed in concrete implementations. The first element of N_P is special in that it specifies the amount of proactive redundancy, i.e. the redundancy that is sent out immediately after the block was filled and the redundancy was coded. The subsequent elements specify how many redundancy packets $N_P[i]$ should be sent for retransmission round i. It should be noted that the protocol aims to send out redundancy packets it has not sent out before, in order to maximize the probability of the receiver completing its block.

When PRRT starts to fill a block, the current coding configuration becomes the configuration of the block and cannot be changed afterwards for that block. It can however be changed in-between blocks in order to provide adaptive HARQ [Gor12]. Figure 4.8 shows how transmissions with coding configuration $k = 4$, $n = 7$, and $N_P = [1, 1, 2]$ look on a time axis. The pro- and reactive transmissions of redundancy are interleaved with the sending of application data and several coding blocks can overlap in time. The latency impact of this configuration can be derived from the temporal distance between the first data packet of the block and the last redundancy packet.

Suitable coding configurations for PRRT ensure that by the time the last redundancy packet is received, there is still sufficient time to—in the worst case—reconstruct the first packet while it is still valid. These configurations ideally fulfil the following criteria:

- The reliability constraint of the application is met, i.e. the coding configuration incorporates enough redundancy and can schedule it over multiple retransmissions rounds in a way that the residual loss is smaller than the tolerable loss.

- The latency constraints of the application are met, i.e. the transmissions happen in a way that packets can be retrieved or reconstructed before they expire (and if they cannot, this happens so rarely that this residual loss is tolerable).

- The coding configuration minimizes the used redundancy information (RI) and attains the optimal RI given residual error as close as possible.

Channel Measurements

Recently, transport layers are increasingly built in a model-based manner, i.e. they measure the channel state information. Most implementations estimate the round-trip time and some are starting to consider the delivery rate. In addition to these, PRRT also measures the reliability by looking at erasure patterns. In consequence, PRRT estimates the following parameters:

- *Round-Trip Propagation Time* is measured similarly to the way NTP measures the RTT [MMBK10] by communicating timestamps in every packet and extracting the channel propagation time.

- *Delivery Rate* is estimated based on an IETF draft [CCYJ17] that shows a general scheme for this task in ACK-based transport protocols. This has been adapted to PRRT as it considers packets instead of bytes as the units that get ACKed.

- *Loss* information is captured by success-failure sequences. Computing the current loss statistics for a channel is done over a time-limited window and results in a average success rate as well as a burstiness—or correlation—of errors. The time-window should be proportional to the assumed correlation time of the channel, i.e. how long the gathered evidence is related to the current state of the channel.

Congestion Control

As congestion control is also a crucial feature to achieve predictability of latency and throughput on the network layer, we added an implementation of BBR [CCG+16] to PRRT. This requires careful tuning as PRRT provides a different service compared to TCP or QUIC, for which BBR implementations already exist.

During operation, BBR controls a *pacing_gain* variable to regularly increase the pacing rate and to send faster than the current R_{btl} to probe for more available data rate. The R_{btl} rate is stored in a windowed maximum filter with a size of $10 \cdot RT_{prop}$[7] to compensate for fluctuations. If a permanent drop in the measured R_{btl} is recognized, which happens as soon as the windowed maximum filter is updated, it also reduces the amount of data in flight and increases the pauses between data to keep the pace.

Rate and Flow Control

As has been motivated in Sections 3.3.2 and 3.3.3, these two functions are essential for ensuring predictable low-latency communication. A joint treatment of these is given in Chapter 6 where the cross-layer pacing scheme X-Pace is described in more detail.

[7] $RT_{prop} = 2 \cdot D_{prop}$ in this case

	PRRT	**TCP**	**UDP**	**RTP**	**QUIC** [Goo18]	**SRT** [Hai18]
Reliability	$< 100\,\%$ (configurable)	$100\,\%$	best effort	configurable via profile	$100\,\%$	$< 100\,\%$
Error Control	HARQ	ARQ	n/a	configurable via profile	ARQ & XOR-FEC	ARQ
Acknowledgment Scheme	SACK	CACK & SACK	n/a	configurable via profile	SACK & NACK	SACK & NACK
Congestion Control	BBR-based [CCG$^+$16]	various	n/a	see [PS17]	various	custom (live & file mode)
Segmentation	no	yes	no	yes	yes	yes
Latency-Awareness	yes	no	no	no	no	no
Latency-Predictability	high	low	low	low	low	low

Table 4.1: Comparison of PRRT to other transport layer protocols.

Comparison to other Transport Layer Protocols

The design of PRRT is unique and provides many features that other protocols do not provide or only provide partially. An abridged comparison of established and emerging transport protocols is given in Table 4.1[8]. PRRT stands out in the categories *reliability*, *latency-awareness* and *-predictability*, which is what was intended by the design. Only allowing less than full reliability makes dependable statements about the latency possible. In this sense, even though the application scenarios for protocols such as PRRT and SRT look similar, PRRT provides a unique set of features that make it well suited for CPS and other real-time applications.

4.6 Conclusion

In consequence, PRRT builds upon the theoretical foundation of Shannon: The only way to bound time is to accept partial reliability. As many[9] applications have this bound on communication latency, it is essential that this information is conveyed to the

[8]The version of QUIC that was considered is Google QUIC, because the IETF version of QUIC is still in the process of standardization (https://datatracker.ietf.org/doc/draft-ietf-quic-transport (accessed August 22, 2019)).

[9]To be precise, even protocols that use the fully reliable TCP, e.g. HTTP, have a time limit that takes the form of a human being's (im-)patience.

transport layer, together with a statement about the required, non-zero loss-tolerance. The latency constraint the application specifies is further used to parametrize PRRT's control functions, namely error, congestion, and rate control. PRRT provides several send and receive modes of operation to cater to different applications' needs. Thereby, the new version of PRRT presented here provides a unique set of features and its latency-awareness makes it particularly suited for control and live-multimedia applications.

We have to remember that what we observe is not nature herself, but nature exposed to our method of questioning.

Werner Heisenberg

Chapter 5

Cross-layer Latency and Timing Analysis

When providing communication facilities for cyber-physical systems, it is essential to know about their timing characteristics and be aware of the distribution of the latency that is added (cf. Chapter 3). This chapter in particular considers the *processing latency* incurred by the end-system's communication stack—including software and hardware components. As soon as one is aware of these latencies and their distributions, it is possible to execute a root-cause analysis and plan optimizations that reduce the latency itself or narrow its distribution. As these latencies are directly and indirectly related to the network protocol and the operating system platform, it is essential to holistically analyse the system's processing latencies together with the network conditions.

5.1 Challenges in Reproducible Low-latency Cross-layer Profiling

Though the vision of such a holistic analysis toolchain is clear, its implementation is faced with several challenges: First, networked systems often come with layers that allow interoperability but impede a holistic analysis. Second, CPS operate in latency regimes that have not yet been sufficiently investigated and where adequate optimization approaches are still to be found. Third, these latency regimes demand for testing and profiling utilities that are minimally invasive to not distort the analysis. Lastly, the investigation results must be produced and documented in a form that allows reproducibility in order to comply with the strict reliability constraints of distributed CPS.

5.1.1 Cross-layer Profiling

The rapid growth and overwhelming success of the Internet is often attributed to the central design principle of interoperability, most notably defined by the ISO/OSI model. Though these abstractions foster interoperability by clearly defined APIs, they impede cross-layer analysis as a layer's functionality is often opaque at runtime. Cross-layer profiling must thereby dissolve these layers at *analysis time* and holistically analyse the whole communication system.

Measuring latency on a *system level* is challenging[1] as the measurement process itself adds additional latencies whose magnitude depends on the approach used. Measuring externally is not an option as that would require adding code that creates external signals via IO, which involves syscalls and adds severe latencies. Such an approach would, in many cases, also loose the direct relationship of a piece of code to the execution latency. Measuring internally, in contrast, allows to use a finer granularity (i.e. smaller blocks of code), but poses the challenge of correlating system time to real-world time—the measure we are eventually interested in. Even though these measurements are challenging, solutions exist that profile programs on a source code level to investigate which functions take how much time (e.g. `gprof`[2]). What is missing is the relation to the executing hardware—a crucial aspect in CPS—as well as application level parameters and demands, which provide context for the latency analysis. Furthermore, most profiling tools add excessive latency due to the instrumentation process—an issue that prohibits the analysis of real-time communication stacks.

Analysing latency on a *network packet level* is straightforward and well-supported by tools (e.g. `wireshark`[3]). Nevertheless, the granularity of single packets and inter-packet latencies is not sufficient to analyse the impact of certain protocol functions to the incurred latency. Automated protocol testing is provided by Google's PacketDrill[4], but its major focus is on functional correctness (i.e. protocol unit tests) and not on non-functional properties, such as incurred latency and energy demand. Symbolic execution frameworks, such as SymPerf [RKR+17], allow a characterization of network function processing delays and functional correctness, but focus more on middleboxes and not on end-systems with their transport layer protocols. Having a protocol view on the interactions between systems does not allow for a straightforward correlation with the code and functions that implement these interactions—and whose execution is the source of latency.

For distributed CPS, it is essential to profile and analyse the network and system domains at the same time [BIYC06, DB13]. There are distinct branches of research dealing with *worst-case execution time* (WCET) [WEE+08] and *worst-case travel time* (WCTT) [FFF09, LBBN16], providing empirical or theoretical bounds for practical or modelled systems. Schimmel et al. [SZ10] compute WCET figures for communication in closed industrial systems under the assumption that the channel is static and there is no competing traffic. [LEL+16] uses these findings to give bounds on the end-to-end response times in distributed IEC 61499 applications that are leveraging commercial-off-the-shelf Ethernet hardware and coexisting with best effort traffic. For distributed

[1]http://btorpey.github.io/blog/2014/02/18/clock-sources-in-linux/ provides an excellent tutorial on how to measure latencies in Linux (accessed August 27, 2019).

[2]https://sourceware.org/binutils/docs/gprof/ (accessed April 17, 2019)

[3]https://www.wireshark.org/ (accessed April 17, 2019)

[4]https://github.com/google/packetdrill (accessed April 17, 2019)

CPS, this coexistence must be considered and isolated solutions as in [SZ10] can—in many cases—no longer be assumed. As of today, it is not possible to provide *worst-case end-to-end times* that consider both the network- and system-incurred latencies in the context of distributed CPS.

Therefore, the tools developed in the aforementioned domains cannot be used for cross-layer timing analysis and profiling, and new approaches are required.

5.1.2 The Microsecond Regime

The technical advances that happened over the last decade lead to the situation where the orders of magnitude of certain latency distributions have drastically changed. As Barroso et al. [BMPR17] point out, system designers used to deal with and optimize for hardware operations (e.g. cache access) in the *nanosecond* domain and software operations in the *millisecond* domain (e.g. context switch). In order to prove that this has changed, they give an example from the domain of datacenter network infrastructure, where 40 Gbit/s links are no longer uncommon. In this scenario, a packet with a default MTU of 1500 B takes about 300 ns to be serialized, and a distance of 200 m is covered in 1 µs assuming fibre optics with nearly speed of light propagation speed. The same is the case in the *Tactile Internet* [FA14] where the physical layer is expected to have end-to-end latencies that are below 1 ms, in order to meet human thresholds for haptic feedback which range from 1 to 10 ms. Existing solutions cannot easily scale to the microsecond regime that is increasingly important for new devices and application scenarios. While this research challenge is tackled from different directions, the focus of our work is on the measurement of latencies caused by certain hard- and software operations—enabling latency-awareness of network transport stacks. There are no off-the-shelf approaches for executing empirical analysis in distributed CPS that operate accurately at sub-millisecond latencies.

5.1.3 Reproducibility

There have always been concerns about reproducibility in various fields of science (cf. the set of references mentioned by Munafo et al. [MNB$^+$17]), making reproducibility a core trait of science that has a sustainable impact. The computer science community in general—and networking community in particular [BBF$^+$19]—is facing a high complexity in terms of systems (aka. collection of environment variables) where the interplay of numerous factors leads to high variability. Therefore, several conferences and organizations (e.g. ACM[5], Usenix, or Nature[6]) are pushing to improve the reproducibility of the science that is published through the peer-review process. Finally, as many disciplines are increasingly using computational methods [SNTH13], the purely textual scientific paper as the primary artefact of advancement in science is questioned[7].

[5]https://www.acm.org/publications/policies/artifact-review-badging (accessed April 17, 2019)
[6]https://www.nature.com/collections/prbfkwmwvz/ (accessed April 26, 2019)
[7]https://github.com/andreas-zeller/papers-as-modules (accessed April 16, 2019)

When it comes to reproducibility, the ACM defines three levels:

Repeatability assures that the same team of researchers is—using the exact same experimental setup—able to produce the same results.

Replicability is achieved when the same experimental setup can be used by a different (team of) researcher(s) to produce the same results.

Reproducibility is when the same results can be achieved by another team of researchers, using different conditions or custom solutions that follow the spirit of the solution to be reproduced.

The area of CPS is expected to cover a large range of devices and systems at different scales. For supporting all of these and providing the reliability and real-time characteristics that CPS are demanding, it is essential that evaluation results are reproducible. The last part of this chapter describes some of our work that helps adhering to the following rules: (1) Store and keep track of software artefacts and their version. Many of the artefacts are stored in Docker[8] containers and related to a specific git commit in the GitLab installation of the Telecommunications Chair at Saarland University[9]. (2) The evaluation process is automated as much as possible, using Python scripts as well as orchestrations solutions, such as Salt[10]—effectively having experiment documentation as code. (3) Evaluations and analyses are done using version-controlled scripts and Jupyter notebooks[11] operating on the raw captured data.

5.1.4 Solution Approaches

In summary, these challenges demand for a solution that is able to jointly capture network as well as system latencies with minimal overhead. Such a solution allows the validation of latency- and timing behaviour of distributed CPS. For this reason, the open source tool X-Lap [RSH+17, RSH+18, Tel19b] is developed to profile PRRT (cf. Chapter 4). Finally, the *Network Experiment Automation Tool* (NEAT) [SH17a] that aims to increase the reproducibility of evaluations is presented and details on additional means to make the research reproducible are given.

5.2 Time in Systems

As pointed out by [Lee08], in many systems "timing is merely an accident of the implementation". While we are still lacking system architectures and programming languages that allow arguing about their precise timing under real-world conditions, this section

[8]https://docker.io (accessed April 17, 2019)
[9]https://git.nt.uni-saarland.de (accessed April 17, 2019)
[10]https://www.saltstack.com/ (accessed April 17, 2019)
[11]https://jupyter.org/ (accessed April 17, 2019)

describes the means that are currently available to systems to measure and argue about time. As long as predictable timing *down to the silicon* is not available, these means have to suffice. For our cross-layer profiling, we demand timing measurements that (a) are accurate enough to be able to profile short code sections and (b) induce minimal run-time overhead.

As of today, most computer systems are equipped with some form of crystal oscillator that acts as a low-overhead source of human time, which is often called a real-time clock. Depending on their quality, drift speeds might differ and lead to different levels of precision and reliability. Apart from that, computer systems have system clocks provided by the operating system kernel. They are typically initialized based on measurements from the real-time clock. In order to measure time of operations for certain code blocks, the real-time clocks are the most crucial in a CPS context.

For latency analysis, there are several types of (time-)stamps that can be taken:

- Timestamps that are related to human time, i.e. measured in seconds, can be taken from the real-time or system clocks through the OS (e.g using `clock_gettime()`, querying `CLOCK_MONOTONIC` with nanosecond granularity). The drawback of this approach is that taking a timestamp through a system call takes additional latency—a process that makes the actual measurement less reliable. Later sections show that in our evaluations, taking a timestamp took around 70 ns.
- Cyclestamps, in contrast, use the processor's cycle counter as the cycle count increases while time passes (e.g. using the `rdtsc` instruction on x86). Due to changing processor speeds and code that runs on multiple cores, this measure must be treated carefully—though it being obtainable in short time is highly beneficial (about 10 ns according to our evaluations).

When considering code sections that execute within a single thread on a single core, it is straightforward to map a number of cycles to a duration in fractions of seconds, using linear interpolation, as can be seen in Figure 5.1 [RSH+17]. Thereto, the first and last cycle stamp for a thread are accompanied with a timestamp sample, allowing a count of seconds (e.g. T_{Start}) to be directly mapped to a count of cycles (e.g. C_{Start}). This allows to take only a minimum number of expensive timestamps and reconstruct the remaining timestamps from cyclestamps that can be captured faster.

Finally, modern systems provide multiple layers at which timestamps can be taken, e.g. hardware timestamps, where the network interface controller queries the real-time-clock immediately after packet reception, or software timestamps, which are taken within kernel or userspace applications.

5.3 Cross-layer Timing Analysis with X-Lap

In order to tackle the challenges around predictably low latency in CPS, we have developed a custom toolchain, X-LAP, which allows to profile communication stacks using minimally invasive time- and cyclestamping.

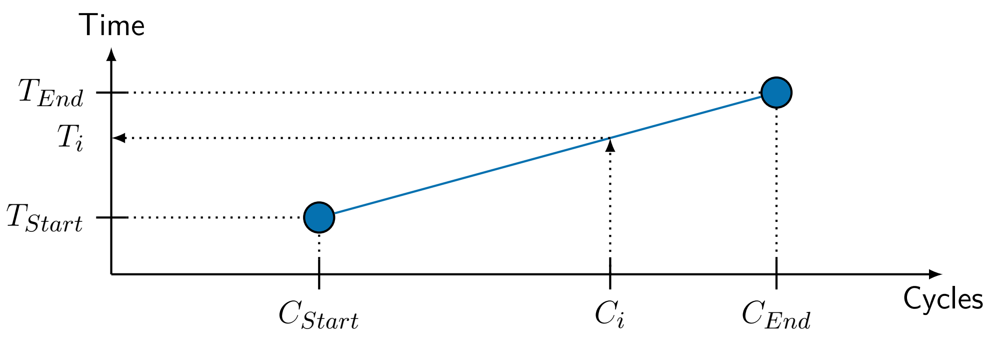


Figure 5.1: Mapping per-thread cyclestamps to respective timestamps using linear interpolation to achieve sufficient precision with reduced overhead.

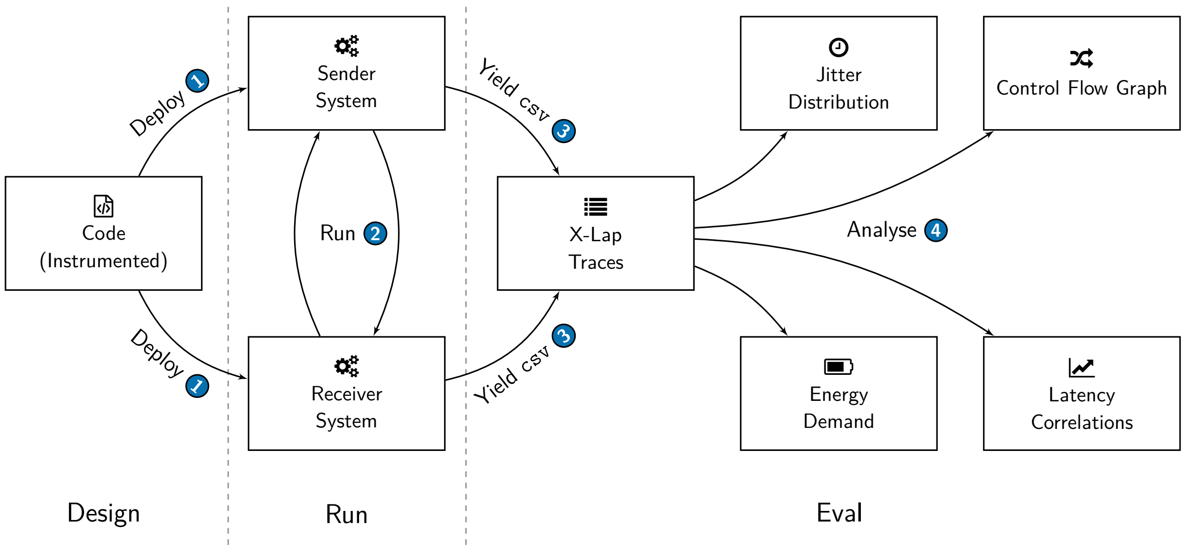


Figure 5.2: X-Lap is used to instrument code at *design time* and profile the code at *run time* to allow *offline evaluations*.

5.3.1 Approach

Using X-Lap involves three phases that are depicted in Figure 5.2. Initially, the developer has to set up the `xlap.yml` file that describes the capturing details. This file is used to generate a header file for usage in the code to be profiled. The user specifies all stamps that should be taken and whether these are taken as time- or cyclestamps. She also specifies the threads (e.g. transmission or coding) that are involved and the stamps that act as cycle references for the interpolation. Additionally, the sides (i.e. sender and receiver) are specified including their respective reference timestamps (i.e. at which timestamp the processing of a side starts and ends). Lastly, durations can be specified as pairs of stamps, which are computed at analysis time automatically.

```
void data_transmitter_loop() {
  PrrtPacket* pkt;
  do {
    pkt = pull(sendDataQueue);
  }
  while (!pkt);
  transmit(pkt);
}

void transmit(PrrtPacket* pkt) {
  XlapStamp_Time(PrrtTransmitStart, pkt->seqno);
  XlapStamp_Cycle(PrrtTransmitStart, pkt->seqno);
  create_coding_block_if_not_present();
  add_feedback_information_to_packet(pkt);

  int result = send_packet(pkt);
  if (result) {
    XlapStamp_Time(PrrtTransmitEnd, pkt->seqno);
    XlapStamp_Cycle(PrrtTransmitEnd, pkt->seqno);

    add_packet_to_current_coding_block(pkt);

    if (coding_block_ready()) {
      code_and_send_redundancy();
    }
  }
}

void send_packet(PrrtPacket* pkt) {
  pace(); // delays sending of data using cross-layer pacing
  wait_for_free_congestion_window_space();
  compute_next_send_time(); // for cross-layer pacing
  char[] bytes = serialize_packet_to_bytes(pkt);

  XlapStamp_Cycle(LinkTransmitStart, pkt->seqno);
  struct timespec timestamp;
  uint64_t cyclestamp;
  send(sock_fd, bytes, &timestamp, &cyclestamp); // UDP send() + hardware stamps
  XlapStamp_TimeValue(ChannelTransmit, timestamp, pkt->seqno);
  XlapStamp_CycleValue(ChannelTransmit, cyclestamp, pkt->seqno);
  track_outstanding_packet(pkt);

  XlapStamp_Cycle(LinkTransmitEnd, pkt->seqno);
}
```

Listing 5.1: Simplified code that deals with the actual transmission of a data packet that was given to the PRRT stack by the application thread (cf. https://git.nt.uni-saarland.de/LARN/PRRT/blob/prrt-0.4.1/prrt/proto/processes/dataTransmitter.c).

At *design time* the networking code is extended by calls to capturing facilities. For the sake of usability, these calls are C macros that also take care of preparing and post-processing of the measurements. Furthermore, there are several distinct calls to capture time and processor cycles and deal with sequence numbers. An example of this can be seen in Listing 5.1. The `transmit()` function starts by taking both time and cycle stamps to serve as a starting point for the thread's processing of one packet. Similarly, both stamps are taken after the packet was successfully sent to the network—serving as the ending point for the thread's processing of one packet. The `ChannelTransmit` stamps are provided by hardware timestamping facilities, if they are available to the evaluation system. All the other stamps are only taken as cyclestamps in order to achieve a minimal profiling footprint.

At *run time*, the capturing is initially set up, which is a costly operation but does not affect the measurements. During the processing of each packet, low-overhead capture calls are used to store the current time or cycle. Cycle calls (`XlapStamp_Cycle()`) are preferred wherever possible to reduce the instrumentation overhead (cf. Section 5.2). In order to ensure that time and cycles are related for a given thread, the evaluations use thread-pinning (cf. Section 4.4). After the experiment has ended and all samples are taken, the in-memory traces are serialized to `CSV` files, enabling further processing.

In the *evaluation*, the analyses are carried out offline. The analyses are provided as Python scripts and methods, accompanied with a Jupyter Notebook making use of several scientific computation libraries for Python (numpy[12], pandas[13], and matplotlib[14]).

5.3.2 Analysis

X-Lap's evaluation starts with generic preprocessing steps and continues with specific steps for the desired analysis. First, the data set is cleaned by removing those trace numbers that do not have the full set of stamps associated with them. In the context of lossy communication links or steps with high delay, this cleaning step must filter out packets that got lost or have been expired before reaching the receiver application. In any case, the number of dropped traces is reported to the analyst so that she is aware and can confirm if this dropping is intended and does not distort the analysis results.

The two data sets (i.e. sender and receiver) are joined, based on the sequence numbers of the packets. Based on this full data set, the low-overhead cyclestamps are used to reconstruct timestamps based on thread-based linear interpolation. Additionally, the durations (e.g. pairs of timestamps defined in `xlap.yml`) are computed, which are often associated with a certain processing step.

[12]http://www.numpy.org/ (accessed April 23, 2019)
[13]https://pandas.pydata.org/ (accessed April 23, 2019)
[14]https://matplotlib.org/ (accessed April 23, 2019)

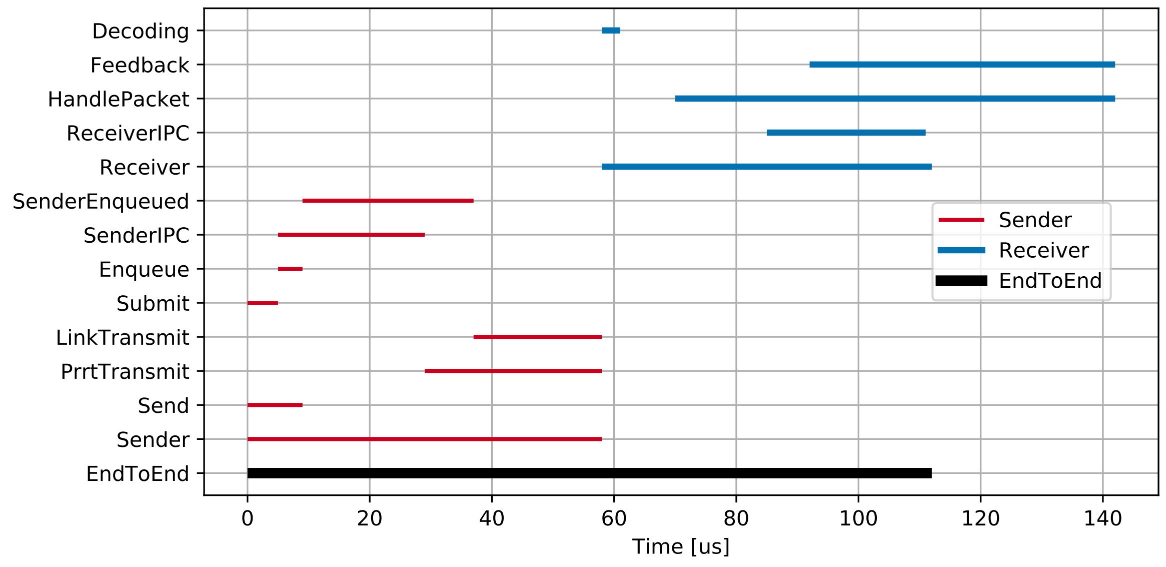


Figure 5.3: Timeline of a single packet trace, showing when and for how long a packet has been in a certain processing or communication step.

After preprocessing the raw data set, the following analyses can be carried out:

Packet Trace represents a detail view on each individual packet (cf. Figure 5.3), providing a timeline of when the packet has been in which processing step. The sides are marked by separate colours and the communication delay is omitted. Thereby, the history of a certain packet can be analysed a posteriori—making exceptional cases visible.

Trace Jitter Based on the durations in the traces, box-plots are created to show the latency distribution for each step. The Figure 5.4 [RSH^{+}17] shows that *Inter-Process Communication* (IPC) used to be one of the main causes of jitter—a circumstance we have tackled in recent PRRT versions using wait-free synchronisation mechanisms.

Outlier Detection Using the outliers identified through the trace jitter analysis, Figure 5.5 provides an overview on the processing steps that are frequently causing this type of anomaly. When making systems more reliable and predictable, these are the first to be addressed. We consider outliers (w.r.t. a specific duration) as being beyond a margin of 50 % of the inter-quartile range (IQR) around the IQR. For the root causes of increased end-to-end (E2E) delay, we consider traces where the E2E latency is an outlier and count the partial latencies of this trace that are also outliers. Eventually, we arrive at a frequency measure of how many times a certain partial latency was exceptionally high and correlated with exceptionally high E2E delay.

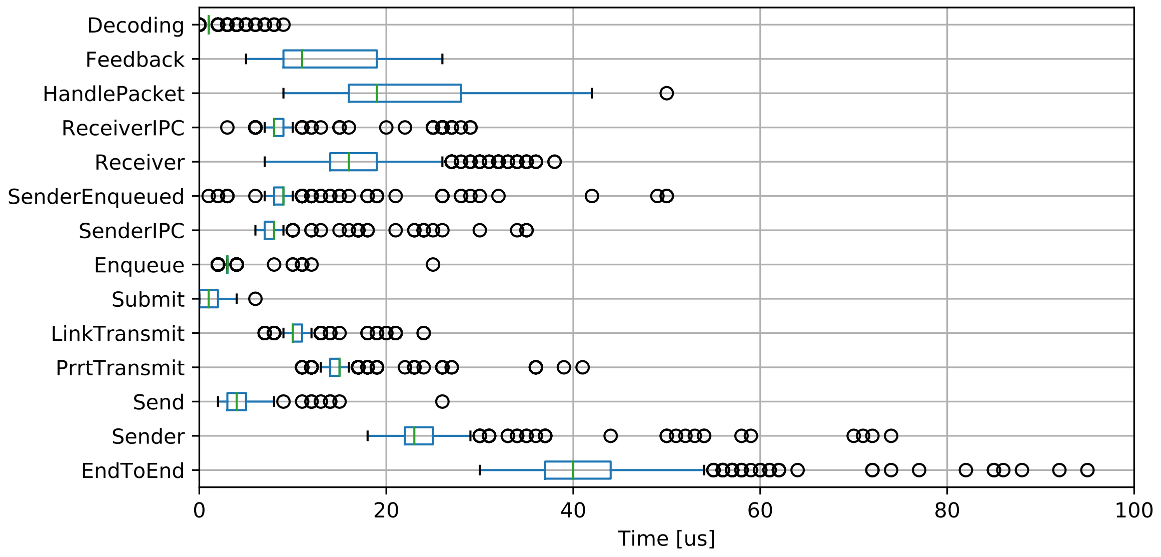


Figure 5.4: Trace jitter for different steps within PRRT, including transmission and inter-process-communication, which cannot be jointly measured by other tools.

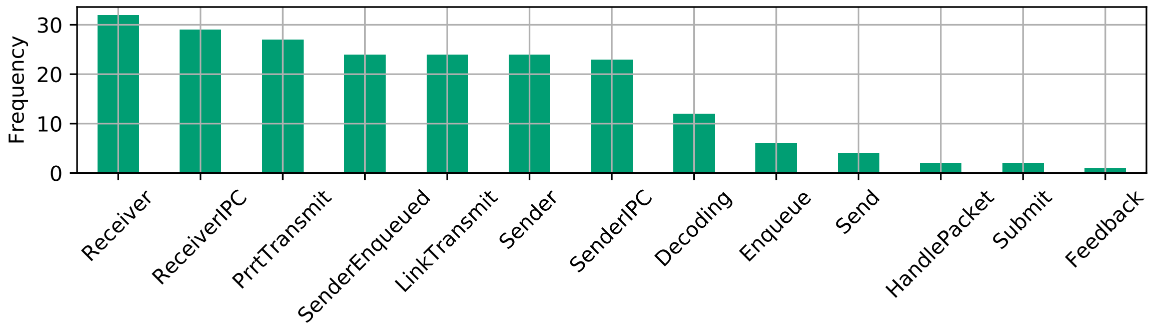


Figure 5.5: A histogram of processing latencies shows cases where the partial latency and the end-to-end latency are outliers simultaneously.

Correlations between partial latencies and the E2E latency are visualised in diagrams such as Figure 5.6. While we cannot directly infer causation from correlation, this analysis reveals which partial latencies are promising candidates for a further code-based root cause analysis.

Generalization and Usability

As of today and to the best of our knowledge, X-Lap has been used only with PRRT. The X-Lap facilities are nevertheless generic and allow for the analysis of any datagram-oriented transport (e.g. UDP). We are confident that a segment-level analysis of TCP

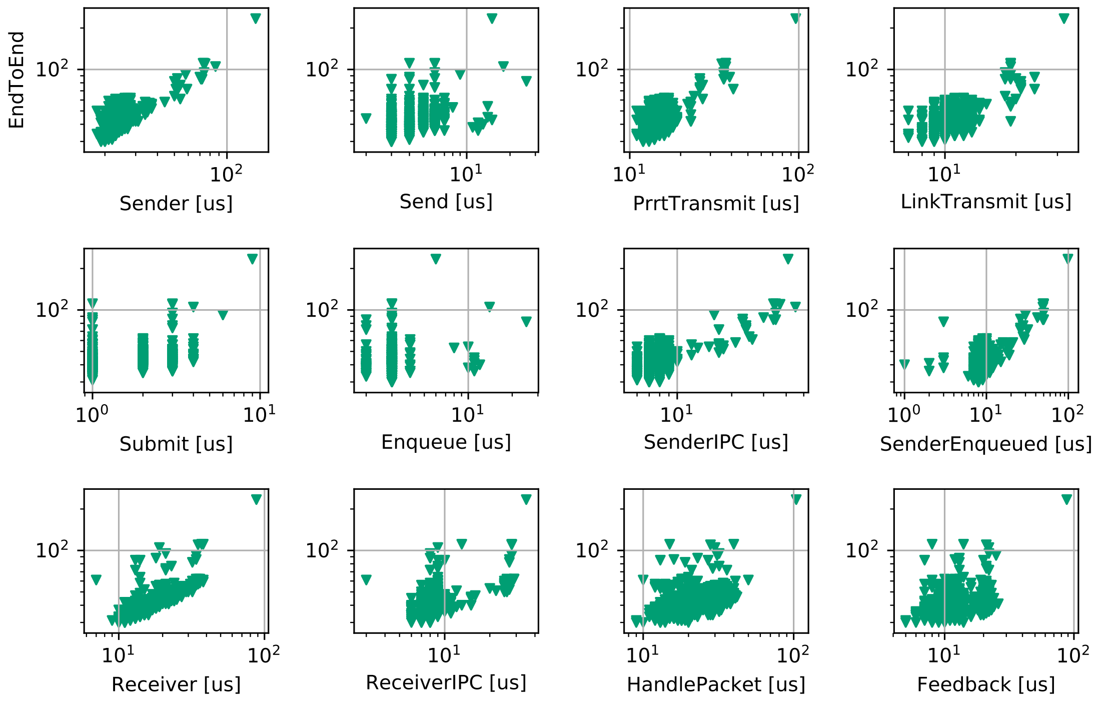


Figure 5.6: Correlations between partial latencies and the end-to-end latency reveal whether a processing step is likely to influence the end-to-end delay.

is also possible, though extra care must be taken to comply with TCP's semantics and its implementation, e.g. in the Linux kernel.

5.3.3 Δelta Extension

Δelta [RSH+18] extends X-Lap by additional analysis methods and capturing facilities. The former includes means to evaluate multiple experiments at the same time to investigate code or environment changes. The latter is concerned with capturing the energy demand during the evaluation. Thereto, Δelta extends X-Lap by the following additional evaluations:

Multi-series Analyses allow to compare multiple experiments (e.g. full repetition, same-code with varying processor speeds, or several code versions) and check how changes to the environment or code have an impact on timing (e.g. Figure 5.7).

Energy Efficiency Evaluations investigate how the processor speed affects a certain processing step's latency (cf. Figure 5.7). While it is evident that faster, more energy-intensive execution generally leads to shorter latencies, the relationship is non-trivial as the curves show (see also [MLVH+02, LSH11]). The average ET^2 metric [MNP02] is a

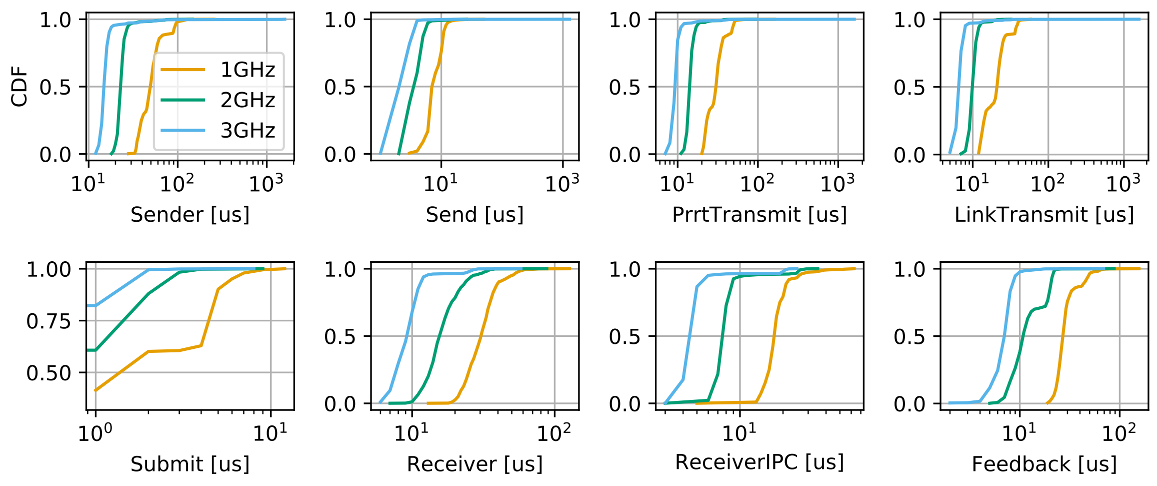


Figure 5.7: CDFs for latency of specific steps—depending on processor speed.

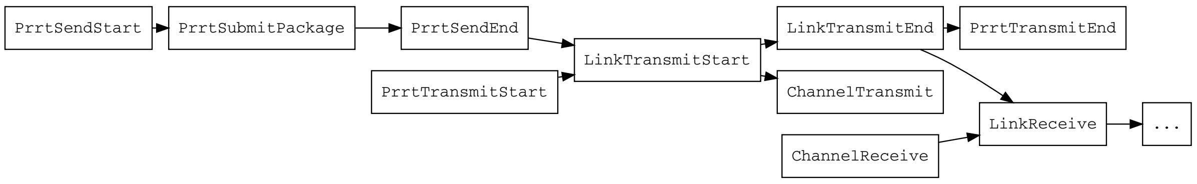


Figure 5.8: Control-flow graphs can be extracted from the data set, shedding light on concurrency aspects of the transport stack. The last few segments are omitted.

fair comparison for *dynamic voltage and frequency scaling* (DVFS) settings. Latencies are measured using X-LAP and the energy demand using RAPL [Int18]. For the data that is shown in Figure 5.7, we have achieved different average energy efficiencies (1 GHz: $0.27\,\mathrm{nJs}^2$, 2 GHz: $0.13\,\mathrm{nJs}^2$, 3 GHz: $0.16\,\mathrm{nJs}^2$), showing that the most energy-efficient configuration is neither achieved by the fastest nor the slowest processor speed.

Control-flow Extraction is an intermediate analysis step and allows X-LAP to argue about the sequence of events by evaluating the *happens-directly-before* relation [Lam78]—a subgraph is displayed in Figure 5.8. While we expect whitebox testing for X-LAP, i.e. this information is known, this algorithm saves the experimenter from manually specifying and updating this relation—saving time, avoiding mistakes and making the analysis more robust to further cater to reproducibility. Furthermore, it captures this relation between threads by analysing the experienced interleavings.

Latency-criticality Using the control-flow graph as a basis, we consider all code segments (i.e. edges between nodes in the graph). Afterwards, we compute the non-parametric Spearman correlation coefficient to identify code segments that have a high positive correlation with the end-to-end latency, i.e. high partial latencies correlate with

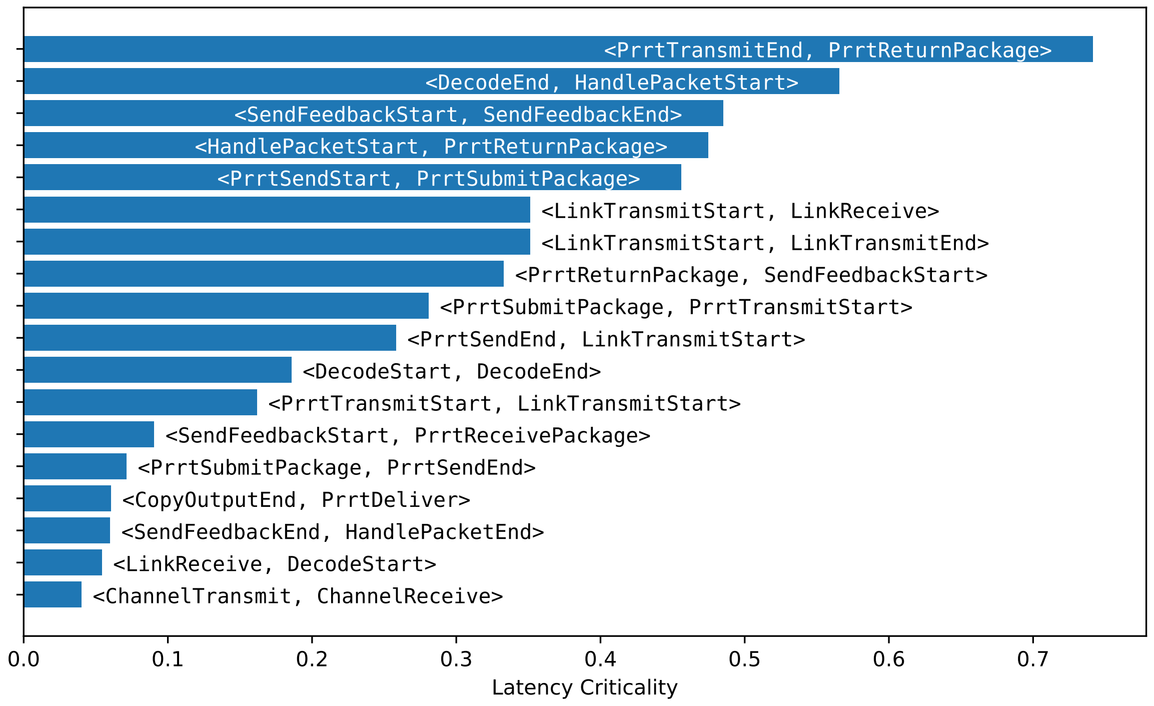


Figure 5.9: Based on the extracted control-flow graph, the Spearman correlation between partial latencies and the end-to-end latencies are computed.

high E2E latencies. The results can be seen in Figure 5.9. When looking at the associated source code, we find that some of these segments with high correlation contain code that deals with process synchronisation—while others contain sequential data processing. The former gives incentives to further decouple processes. The latter indicates that optimization of these process steps is likely to improve the E2E latency.

5.4 Network Experiment Automation Tool (NEAT)

Empirical studies of networking research are often concerned with physical or virtual testbeds that provide higher fidelity than pure simulations. The drawback of testbeds, in comparison to simulations, is that the environment is not automatically documented and harder to reproduce. Using state-of-the-art tools from the domain of network and systems operations, these drawbacks can be compensated. Thereto, a number of information systems are used together with specific technologies to make the experimental setup as transparent as possible. These include:

Configuration Management provides means to have *infrastructure-as-code*, i.e. to use a software tool and a well-known, human- and machine-readable configuration language to enforce a specific state of the infrastructure. For network experiments, this could be the configuration of kernel parameters or the presence of specific configuration files.

Version Control (VC) Systems such as Git or SVN allow to keep track of the changes that are made to specific files—independent of the fact whether they are used for building software artefacts or act as descriptions for configuration management. As each *commit* has a unique identifier, it is possible to relate results to this specific code version.

Continuous Integration (CI) is often built into collaboration solutions that also include version control systems (e.g. GitHub, GitLab, or the Atlassian tool suite). CI allows to automate the generation of software artefacts used in the evaluation—ideally in a reproducible manner[15]. Especially when the artefacts itself cannot be archived, having a way to reproduce them after some time has passed is valuable.

Software-defined Networking (SDN) makes routing and forwarding policies transparent, which means that system as well as networking aspects are documented. SDN controllers can help, and even be drivers for network experiments.

Network Function Virtualization (NFV) enables to orchestrate and deploy experiment components in a straightforward and automated way. Having these components as virtual network function (VNF) allows to (a) integrate them into version control, (b) capture their placement in configuration management, and (c) automate their configuration (using the fact that a VNF typically has a more common interface, e.g. REST, than most physical network functions).

As this degree of automation is not common in networking research yet, we developed the *Network Experiment Automation Tool* (NEAT) [SH17a]—a first prototype that uses all of these features to automate experiments.

Figure 5.10 shows NEAT in action: First, a researcher commits and pushes her code to a version control system with CI ①. After that, she updates the experiment description in case anything needs to be updated for the next experiment ②. When the description is ready, the experimentation process is started ③. Thereto, the experiment description is parsed by NEAT ④ and related software artefacts and infrastructure descriptions are retrieved from VC ⑤. As soon as everything is in place and the testbed is configured, the actual experiment runs ⑥. Afterwards, NEAT makes sure that all evaluation artefacts (e.g., logs, measurements) are retrieved and stored together with a copy of the experiment description ⑦.

Using this approach, a vast majority of otherwise opaque experiment parameters are gathered and archived together with the results—which in the aftermath will be used to generate analyses or visualisations. By following the rules described in [SNTH13,

[15]Mainly (but not only) for security reasons, there is recently an increasing interest in making software build processes reproducible; e.g. in the Debian project (https://wiki.debian.org/ReproducibleBuilds, accessed August 14, 2019).

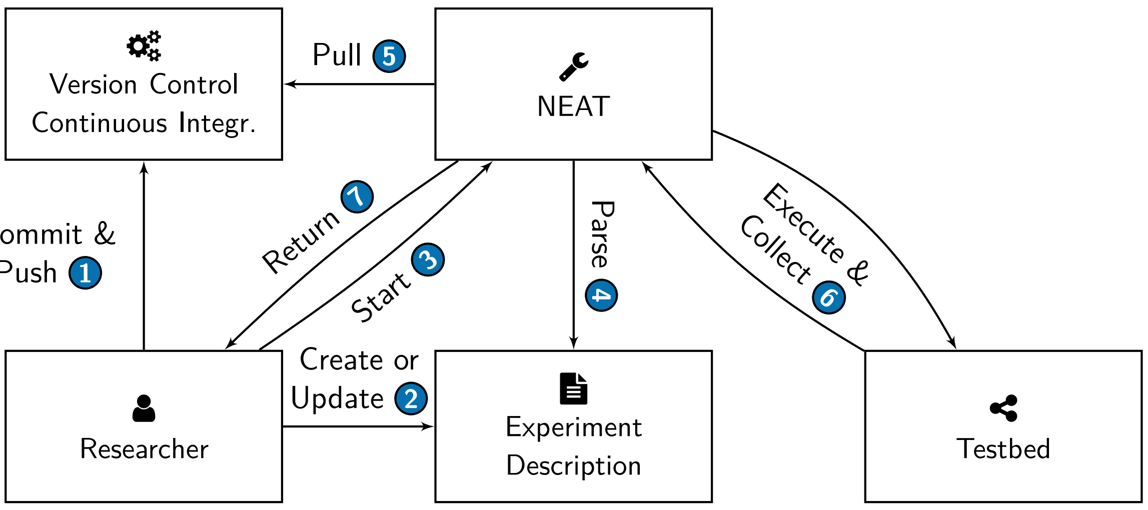


Figure 5.10: NEAT decouples the experiment description process from the (fully automated) execution of the experiments and archiving of gathered data.

BBF⁺19], these analyses are also under version control—leading to a situation where we can, e.g., backtrace a diagram in a publication to all the relevant parameters and code artefacts that were used during execution. The tool is available as open source[16].

5.5 Conclusion

Distributed cyber-physical systems rely on reliable timing of communication stacks. Using X-Lap, these stacks can be measured by considering *protocol interactions* (i.e. packets) and *system interactions* (i.e. code blocks) at the same time—and in a minimally invasive manner. Having access to this data, various evaluations are possible, in order to analyse timing as well as energy aspects and further guide the engineering process by uncovering correlations and interdependencies. This holistic measurement approach is also going to be relevant for tuning a system to achieve cross-layer pacing that makes latencies more predictable and is going to be covered in the next chapter.

[16]http://neat.larn.systems (accessed April 17, 2019)

The right word may be effective, but no word was ever as effective as a rightly timed pause.

Mark Twain

Chapter 6

Cross-layer Pacing

Queuing is a major source of high and unpredictable latencies in packet-switched networks of cyber-physical systems, as discussed previously in Chapter 3. In general, there are two approaches to avoid queuing: (a) making the network deterministic and getting rid of extensive buffers or (b) adapting the sending behaviour of end-systems, attempting to keep the queues empty.

Deterministic Buffer-free Networks

Uncontrolled queuing delays can be avoided by making the whole network as predictable and deterministic as possible [KAGS05]. This involves scheduling traffic upfront, usually in offline-analysis, and making all participants adhere to the schedule using admission control. One such solution is *Time-sensitive networking* (TSN)[1], a collection of standards to bring determinism and real-time behaviour to, e.g., industrial networks. This determinism is achieved using three mechanisms: (a) time synchronization, (b) scheduling and traffic shaping, as well as (c) routing, reservation, and fault tolerance. These approaches impose reduced flexibility in workloads as re-computation and roll-out of transmission schedules can be expensive [PRGS18]. These schedules can only be enforced when we have full control over the network topology and when this topology as well as the network workloads are mostly stable.

This is unlikely to be the case for all future low-latency applications, especially those that interface with infrastructure on the Internet (where a fully deterministic mode of operation is infeasible—and would contradict the design-principle of the Internet in itself [BM02]) For instance, a manufacturing site might be under the control of an enterprise, but connecting multiple factories involves links with minimal control—most likely through application-agnostic service-level agreements). So while there is ongoing work at the IETF to develop *Deterministic Networks* (DetNets), this is primarily focussing on networks that are under the control of a single authority[2] and interoperable solutions will need different approaches.

[1]http://www.ieee802.org/1/pages/tsn.html (accessed August 14, 2019)

[2]https://datatracker.ietf.org/wg/detnet/about/ (accessed August 20, 2019)

Coordinatively Rate-controlled Senders

As the end-to-end path in a distributed cyber-physical system is likely not to be under full control by the operator, the only measures available are those in the end systems. Systems have—to a variable extent—control over their sending behaviour, in particular the data in-flight and the sending rate.

The data in flight can be given in two ways, either in packets or bits/bytes. Apart from the unit the data in flight is measured in, the process of measurement depends on the type of the protocol. For reliable protocols, the data that is not ACKed is considered to be in flight. After a certain time has passed, e.g. when a retransmission timer has a timeout, this data is considered as lost and no longer in flight (cf. TCP[Pos81]). For unreliable protocols, the data in flight is typically the data that has been sent within a recent time window, e.g. dimensioned by the RTT.

The sending rate is subject to the packet sizes and the packet intervals. The packet sizes can be controlled if applications have variable payloads or allow for aggregation without risking a violation of application constraints (e.g. message deadlines). The packet intervals can be controlled if payload sizes are fixed, using a strategy commonly known as *pacing*.

In this regime, it should be noted what RFC6077 [PWSB11] highlights as the hazardous assumptions made about the relationship between packets and bits during the design of networking hardware and congestion control algorithms during the last decades. With increasing data rates, the per-packet overhead is a dominant performance factor as packet sizes are not increasing in today's Internet. In particular, a shift to packet- (and not byte-) oriented congestion control is considered by the authors and it is going to be a major concern in the next years.

The current lack of pacing in latency-sensitive—as well as other—networks is expensive, as buffers need to be big (hardware cost), en- and dequeuing packets is done regularly (energy cost), and buffers are filled most of the time (latency cost). This chapter presents X-Pace, an approach to reducing queuing delays by implementing cross-layer pacing to rate control the end-hosts. Thereby, system as well as network buffers are kept mostly empty, resulting in close to the optimum end-to-end transmission times and reduced costs. In the end, an evaluation across the public Internet will show that X-Pace is able to reduce tail latencies in comparison with optimized low-latency TCP by up to 54 %.

6.1 A Brief History of Pacing

Before we look at the design and implementation of X-Pace, we describe what pacing is and why it is not (yet) present in transport layer protocols and adjacent operating system components.

6.1.1 Bufferbloat & The Need for Pacing

The queuing delay (D_{queue}) is a major threat to latency predictability in best-effort packet-switched networks, as empirical evaluations show that *bufferbloat* [GN12, DB13, BMPR17] can easily cause a round-trip time of a residential Internet access to increase from 10 ms to 1 second or more. This is due to the large capacity of queues or buffers to compensate bursty payloads without causing loss—an architectural consequence of congestion incidents on the early Internet [JK88]. These queues can be network queues found in routers on the Internet, but also within system software and hardware, where increasing complexity causes an increase in layers with—often unmanaged—buffers [GN12].

The root cause of bufferbloat is that end-host applications are not aware of two aspects: (a) the size of existing queues at system and network level as well as (b) the throughput provided by the different steps of the processing chain and hops on the network. The former requires coordinated network management strategies, e.g. using *active queue management* [BCC+98, BF15]; the latter requires pacing as defined in Def. 3 and introduced for TCP by Zhang et al. [ZSC91].

6.1.2 The Pacing Concept

The is no general, commonly used definition of pacing in a data networking context, so this thesis refers to the following definition of the term:

Definition 3 (Pacing) *Pacing is the* intentional delaying *of a transmission in order to create* temporal gaps *to pre- or succeeding transmissions.*

The general concept of *TCP pacing* that can be found in the literature covers both: delaying the sending of ACKs after data reception and sending of data after ACK reception. Pacing attempts to evenly spread the transmission of a set of packets across, for instance, the entire round trip time so that no bursts occur.

Figure 6.1 shows the processing of work units 1 to 4 through steps S_1 to S_3, where step S_2 is the bottleneck. Without pacing, queues form at the bottleneck while the throughput stays the same (cf. Figure 6.1a). In contrast, pacing the first step to the pace of the bottleneck helps to keep the buffer empty (cf. Figure 6.1b). This also means that both steps S_1 and S_3 can decide to process work units slower as long as they do it faster than the bottleneck. The bottleneck of a system can be at various locations, including a fully utilized CPU, a fully utilized link, or inadequate timing behaviour at the network or system layer.

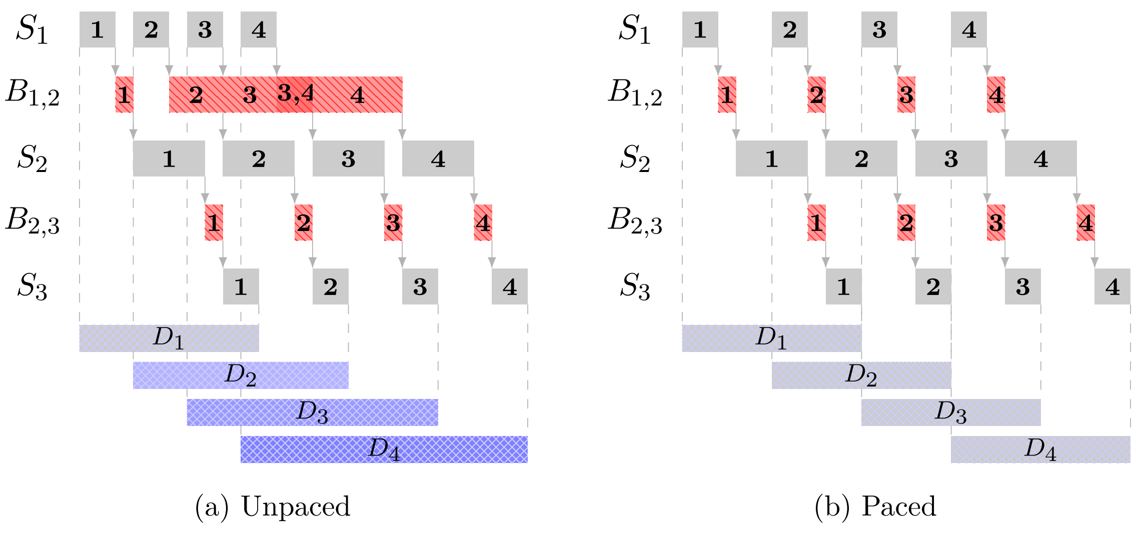


Figure 6.1: Pacing keeps the bottleneck throughput, but avoids excessive buffering latencies (marked with red stripes) and increased E2E latencies (marked with blue hatches).

6.1.3 Controversy Around TCP Pacing

There have been numerous apparently contradicting research studies on TCP pacing over the last decades, leading to different conclusions with respect to its positive effects and viability. Shenker, Zhang, and Clark [SZC90, ZSC91] were the first to propose a pacing approach for TCP congestion control (Tahoe [JK88] in this case) in order to avoid harmful "clustering" of TCP traffic. At that time, the research community was discussing about whether window-based ACK-clocking or explicit rate control should be used for controlling the sending behaviour of TCP. Pacing represents a hybrid approach [ASA00], as it uses windows, ACK-clocking, and controls the rate by delaying packets.

Aggarwal et al. [ASA00]

Aggarwal et al. started with the intuition that pacing should be unconditionally adopted by Internet protocols, but found that pacing is "too good", as it delays the congestion events too far and causes severe synchronized drops. From today's perspective, this result must be revisited for two reasons: First, model-based CCAs (e.g. [CCG$^+$16, AB18]) are able to detect congestion earlier, mostly by a rise in E2E latencies that are attributed to increasing queuing delays[3]. Second, explicit-congestion notification (ECN) [RFB01] is increasingly deployed on the Internet [KNT13] and interactions with the transport layer protocols are proposed [FWK16] to allow an earlier handling or avoidance of congestion events [FW17].

[3]Other effects that increase E2E latency are re-routing events in which a back-off is also advisable as the path changed and is likely to have a different bottleneck data rate too.

From the body of work on TCP pacing, it is not evident whether synchronization should generally considered as good or bad. Aggarwal et al. highlight that pacing has a de-synchronizing effect as tail-drops affect random flows leading to higher aggregate in-flight windows. Cardwell et al. [CCG+16] argue that a self-synchronizing CCA can approach the desired state of an empty queue better than other CCAs. In contrast, a perfect synchronization can lead to traffic oscillations [FJ94] where systems first overwhelm the network by jointly increasing their sending rate, back off, and overwhelm the network again after a short time.

Aggarwal et al. also highlight that bursty connections make the paced connections suffer, leading to fairness problems. This coexistence of paced and non-paced flows is described as a general problem that leads to unfairness with respect to the traditional fairness metrics, as non-paced flows are likely to have larger in-flight windows.

In summary, pacing can provide better fairness, higher throughput and lower loss rates, but not for all scenarios, which is also due to the implementations present at that time—in particular drop-only congestion detection.

Wei et al. [WL06]

Wei et al. review the TCP pacing concept and results from Aggarwal et al. in the light of new network topologies with higher link speeds and new TCP implementations. The authors introduce a *worst-flow latency* as an additional non-throughput-oriented metric into the evaluation. Wei et al. also strengthen the positive aspects of pacing, with respect to reduced burstiness and an increased flow synchronization. They conclude that there are enough incentives for applications to migrate from non-paced TCP to paced TCP—a clear contradiction to Aggarwal et al. Even though non-paced flows—acting somewhat egotistical—gain higher benefits than paced ones, flows of both types gain after a critical mass of flows employ pacing. Hence, they advise to gradually add pacing to network flows, eventually migrating to a scenario where all flows are paced.

Ghobadi et al. [GG13]

Ghobadi et al. show that with TCP and a specific buffer size, there is an upper bound or *point of inflection* (PoI) on the number of concurrent flows that can be paced and lead to significant benefits for all participants. It is important to note that their performance measures, which are used to quantify the benefits, are all about throughput or consider a throughput-maximizing network design. The *flow completion time* (FCT) [DM06] measure depends mainly on the throughput, transmission time, and in-flight window size; not on the end-to-end latency inflation caused by queueing.

In distributed cyber-physical systems, where predictably low latency is preferred at the expense of a smaller throughput (cf. Alizadeh et al. [AKEP12]), this PoI must be questioned and needs to undergo further examination. This PoI is in absolute terms (number of flows), while the critical mass by Wei et al. [WL06] is in relative terms (portion of flows), so in an ideal *controlled* scenario one would limit the number of concurrent flows to the maximum and ensure that they are all paced. In case the network is expected to support more flows, a best-effort approach must be considered together with pacing (if latency is of highest priority) or without pacing (if utilization and throughput are of highest priority). The authors also proposed leveraging SDN technology to communicate the number of flows in data centers and thereby adaptively decide to pace or not, a form of network-assisted pacing.

Summary

In summary, most of the work has focused on the throughput and fairness aspects of pacing, coming to a conclusion that controlled pacing can lead to increased throughput and fairness. Nevertheless, a number of common scenarios could be identified where there is evidence that higher throughput can be achieved without pacing. Most of these studies have not considered the end-to-end latency improvements of pacing and how this can lead to predictably low latencies that are essential to real-time applications—an open question that is going to be investigated in this chapter.

6.1.4 Recent Pacing Endeavours

Apart from the controversial studies of TCP pacing, there is a set of recent publications that present solutions that involve general forms of pacing or rate control; primarily in a data center context.

Alizadeh et al. [AKEP12] The authors leave link capacity unused in order to compensate for the rise in congestion window size during the time between congestion detection and reaction of end-points. They also argue that pacing should be done between the last "component" that aggregates data and might cause bursts of data (in their concrete example *Large Send Offloading*). Their approach can significantly reduce application end-to-end times (at mean and tail) by combining this capacity headroom and pacing.

Jang et al. [JSBM15] (Silo) The authors present a multi-tenant data-center solution to achieve guaranteed network bandwidth, guaranteed packet delay, and guaranteed burst allowance. They identify network guarantees required for predictable message latency and present admission control and virtual-machine placement algorithms to guarantee these properties.

Saeed et al. [SDV$^+$17] (Carousel) The "Carousel" is another traffic shaping solution that combines rate limiting with packet pacing—highlighting that the latter is a key technique to achieve low latency. The authors argue that shaping must be done at end-host as the network does not have enough knowledge about the flows and end-systems states in order to do the pacing right. They highlight that the performance overhead caused by end-host shaping in general limits overall scalability of the approach. This is for instance due to the increased CPU and memory consumption, packet drop due to overflows, head-of-line blocking, and a lack of backpressure. With their approach, they use a single shaper for various processes involving multi-cores. While their approach provides backpressure to the application, they do not implement cross-layer pacing that takes the paces of all layers into account and makes it an explicit measure that is communicated using feedback.

6.1.5 Pacing outside of Networking

Pacing itself is not an invention of the computer networking community, but has its roots in the design of production lines [BSW73, BS06]. Pacing is a tool to implement *lean manufacturing*, a method to minimize the waste caused by a production process. The underlying *just-in-time* (JIT) approach has its origins in the Japanese *Toyota Production System* (TPS) [Ohn88], but is nowadays used in many production environments [Ahm91] and also in other fields of computer science [Ayc03].

While waste avoidance is not a central design goal of research around Internet communication yet, there are several studies that investigate this aspect [GS03, CSB$^+$08, HSM$^+$10, BBDC11, OAL14]. The avoidance of wasted energy is going to be of increasing importance for future energy-aware and energy-sufficient systems. But also with respect to latency—time between order and delivery for a manufacturing system—there are enormous reductions possible by implementing pacing.

6.2 The Cross-layer Pacing Concept

Cross-layer pacing aims at reducing the end-to-end (E2E) latency (cf. Section 3.2.1) at the application level, both in terms of absolute value as well as variance. The pacing approach is increasingly relevant for distributed CPS to achieve predictably low latency by not filling buffers excessively [GN12] and avoiding harmful load spikes when completing operating system tasks [ROS$^+$11]. For the remainder of this thesis, we define a pace as follows:

Definition 4 (Pace) *A pace ($P, [P] = \frac{time}{unit\ of\ work}$) is the time spent to apply a certain step to a certain unit of data.*

Paces are often stated in terms of time only, implicitly assuming a "per work unit"[4]. Most of the following considerations assume the unit of work to be one application layer datagram or message. For paces, it intuitively holds that "lower is better", and one often talks about "fast pace" (low value) and "slow pace" (high value). A general system

[4]An amateur runner can achieve a pace of "5:00 minutes" (per kilometre).

processes and communicates data in a sequence of n steps, where step $i \in [0 : n-1]$ happens with pace $P^{(i)}$.

In any system of steps ($[0 : n-1]$), there exists at least one bottleneck step $S^{(btl)}$ with a pace of $P^{(btl)}$ so that $\forall j \in [0 : n-1] : P^{(btl)} \geq P^{(j)}$. In theory, there could be multiple bottlenecks with the same pace, in which case the last one (the one with the highest index) is the bottleneck that must be used as a starting point for back- and forward propagation of paces (cf. Section 6.4.2). In practice, measuring dynamic paces is unlikely to lead to this case, not to mention that this state is probably going to last for so short that special treatment is futile. A bottleneck is sometimes characterised as a step or transition between steps that has a non-negligible effect on the system's performance as the succeeding step is slower than the preceding (also referred to as choke points). This thesis does not use this local perspective, but a global perspective where there is only a single bottleneck on a path that must not cause measurable performance degradation, but is simply the slowest step. This characterisation is also different from the *bottleneck-link* used to achieve max-min fairness [LB05] where the bottleneck is related to a source and the rates of other sources sharing this link.

Definition 5 (Cross-layer Pacing and Paced Systems) *A (sub-)system S implements* cross-layer pacing *if it ensures that each step* $i \in [0 : n-1]$ *is executed at a pace* $P^{(i)}$ *that considers the bottleneck pace* $P^{(btl)}$ *in the overall system's 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, or more formally:*

$$S \textit{ is paced iff } \forall i, j \in [0 : n-1] : i < j \Rightarrow P^{(i)} \geq P^{(j)}$$

6.3 Benefits of Cross-layer Pacing

Systems that implement cross-layer pacing—and are thereby *paced*—gain multiple benefits. Most of them are reductions of the *muda* or waste as identified by the Toyota Production System [Ohn88], in particular the types of *inventory* (a message is only useful when it is delivered to the receiver), *waiting* (a message loses value over time, i.e. the age-of-information is important[5]), and *over-processing* (sending a message as-fast-as-possible with significant effort might not be required by the end-user):

[5]There is an increasing interest in *age-of-information* and *information freshness* for recent interactive communication systems: https://infocom2019.ieee-infocom.org/age-information-workshop (accessed December 3, 2019)

Near Zero Queuing Delay The pacing of work units to the processing rate of a step ensures that these units would not arrive faster than the bottleneck rate. Thereby, units of work are not put into buffers, effectively avoiding queuing delays.

Just-in-time Processing When the pace at which units of work are handled is known, the respective succeeding layer can run preparation tasks, e.g. allocate memory or wake up threads, so that everything is ready as soon as the next unit of work is passed along.

Reduced Waste of Work When the rate at which the transport stack processes data and can transmit it over a link is known, the operating system and the physical computation platform can run at exactly this speed. This reduces clock-cycles, as polling can be avoided and a slow-down of the processor can reduce the energy demand and prolong the runtime of battery-driven devices. This is also the case for wireless communication links that could pick a throughput and scheduling scheme that can reduce energy demand or increase reliability.

Lower Resource Footprint As soon as buffers—at system and network level—are not filled and required to compensate for insufficiencies of the processing chain (e.g. TCP sending in bursts), the resources allocated for buffers can be reduced. This leads to a lower memory footprint for pure software systems and reduced cost for memory in hardware systems.

In dynamic systems with changing paces—and, in consequence, switching bottlenecks—the system cannot continuously be in the paced state. Due to imperfections of the measurement and latencies in the control itself, cross-layer pacing cannot remove the need for intermediate buffers completely or continuously achieve zero queueing delay. Nevertheless, a prototypical cross-layer implementation is able to approach these goals successfully (cf. Section 6.6).

6.4 A General Cross-layer Pacing Architecture

As many of the studies around pacing (cf. Section 6.1) focus on the networking aspects and in particular TCP, this section describes an abstract pacing architecture that is going to be implemented in the concrete X-Pace system presented in Section 6.5.

6.4.1 Measuring Paces

For synchronizing the paces across the different layers, it is essential to precisely measure the pace of each layer. Their quantification can be done in three ways:

A Priori Paces can be known at design-time, e.g. time per packet transmission (the transmission pace) on an Ethernet link or time per computation step in a real-time operating system running on known hardware.

A Posteriori The system is quantified once using dedicated profiling measurement runs, and the resulting values are considered during operation but without measuring again and validating, i.e. the pace is static across runs of the identically configured system.

In Vivo Finally, it can be required to measure paces because they can depend on a platform and the current load, i.e. the pace is dynamic.

These approaches represent a scale between purely static (a priori) to fully dynamic (in vivo) characterisation of paces. In the following, only in vivo measurements are used as CPS are expected to be dynamic and interface with highly flexible systems such as the Internet. However, on systems where more accurate models are available, for instance in data centers or network-on-chips, the overall performance can benefit from a priori knowledge or a posteriori profiles. The design of PRRT also allows to quantize the achievable paces during evaluation runs and use these static measures in production environments. The feasibility of both of these approaches is out of scope for this thesis, which solely focusses on *in vivo*.

Pace Measurement API

The source code of the communication transport stacks is instrumented using four methods for pace quantification: `start(`$P^{(i)}$`)`, `end(`$P^{(i)}$`)`, `pause(`$P^{(i)}$`)` and `resume(`$P^{(i)}$`)`. Paces of each step have variation by nature, e.g. system noise and fluctuations of network performance, making it necessary to filter them appropriately. Figure 6.2 shows the general process to measure the different components of a step's pace. The code section associated with a step is surrounded by calls to `start(`$P^{(i)}$`)` and `end(`$P^{(i)}$`)`. All steps depend on their pre- or succeeding step, i.e. their pace can contain a portion of *dependent* time that must be treated separately, e.g. when interacting with a different step as could happen when a packet is awaited in `recv()`. These code sections are surrounded with a pair of `pause(`$P^{(i)}$`)` and `resume(`$P^{(i)}$`)`.

Components of a Pace

Thereby, we are able to quantify three different components of the pace $P^{(i)}$ directly (1-3) and compute two additional paces (4+5):

1. *Internal* ($P^{(i)}_{int}$) is the time taken between `start(`$P^{(i)}$`)` and `end(`$P^{(i)}$`)`
2. *Dependent* ($P^{(i)}_{dep}$) is the sum of times taken between `pause(`$P^{(i)}$`)` and `resume(`$P^{(i)}$`)` blocks since the last call to `start(`$P^{(i)}$`)`

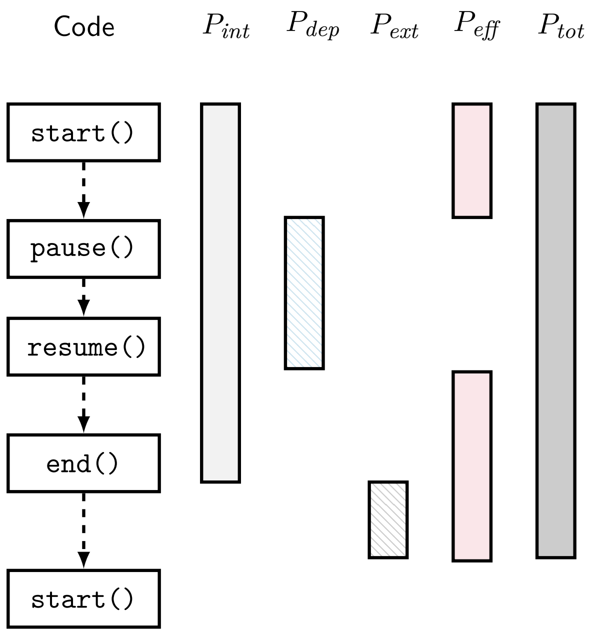


Figure 6.2: General process of measuring the different components of a step's pace.

3. *External* ($P^{(i)}_{ext}$) is the time taken between `end(`$P^{(i)}$`)` and the next `start(`$P^{(i)}$`)`
4. *Total* $P^{(i)}_{tot} = P^{(i)}_{int} + P^{(i)}_{ext}$
5. *Effective* $P^{(i)}_{eff} = P^{(i)}_{tot} - P^{(i)}_{dep}$

The external pace is used to distinguish between the call periodicity $(P^i_{int} + P^i_{ext})$ and the actual work carried out by the step (P^i_{int}). For instance, the total pace for application sending can be high despite the step running fast if the application does computation in-between sending. The subtraction of the dependent pace from the total pace ensures that fluctuations in the effective pace only capture fluctuations in the processing step itself. Otherwise, the system would not converge to a stable bottleneck pace.

6.4.2 Communicating Paces

Using the measured paces, it is imperative to synchronize them throughout the system, starting from the bottleneck $S^{(btl)}$ (cf. Figure 6.3).

Backward propagation Communicate to $S^{(btl-1)}$ that data *cannot be processed* as fast as it could be provided. $S^{(btl-1)}$ MUST run at a slower pace to complete tasks and forward this information further backwards.

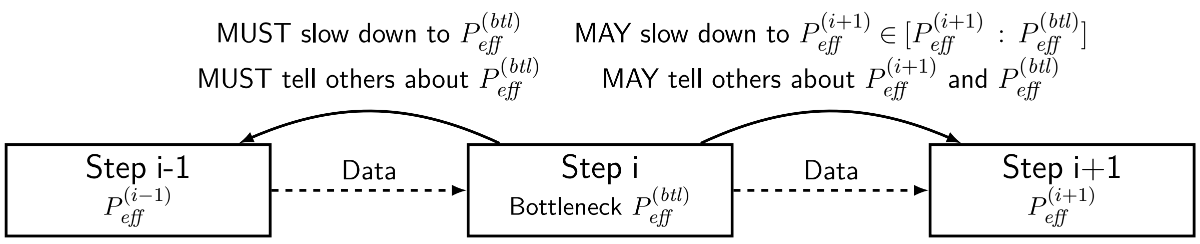

MUST slow down to $P_{\mathit{eff}}^{(btl)}$
MUST tell others about $P_{\mathit{eff}}^{(btl)}$
MAY slow down to $P_{\mathit{eff}}^{(i+1)} \in [P_{\mathit{eff}}^{(i+1)} : P_{\mathit{eff}}^{(btl)}]$
MAY tell others about $P_{\mathit{eff}}^{(i+1)}$ and $P_{\mathit{eff}}^{(btl)}$
Step i-1
$P_{\mathit{eff}}^{(i-1)}$
Data
Step i
Bottleneck $P_{\mathit{eff}}^{(btl)}$
Data
Step i+1
$P_{\mathit{eff}}^{(i+1)}$

Figure 6.3: Pacing across a chain of processing steps by propagation a pace backward and forward.

Forward propagation Communicate to $S^{(btl+1)}$ that data *cannot be provided* as fast as it could be processed. $S^{(btl+1)}$ MAY run at a slower pace. Tasks may decide to keep their pace, because finishing "fast" has benefits, e.g. low resources footprint or minimal E2E delivery time. This is line with [LSH11], where it is stated that the relation between energy-efficient processing and processing speed is non-trivial. In any case, the new effective pace $P_{\mathit{eff}}^{(btl+1)} \in [0 : P_{\mathit{eff}}^{(btl)}]$ may be forwarded to the succeeding steps to be acted upon. This allows these steps to optimize their behaviour with respect to, for instance, energy demand.

6.4.3 Adapting to Paces

The measured and communicated paces can be acted upon—with a strategy that depends on whether the pace is *tractable* or *intractable*. An intractable pace (e.g. medium access) depends on the environment and is not under control of the communication and computation stack. If a step with an intractable pace must be slowed down, the only way is to find the next preceding step with a tractable pace and reduce this one so that the total pace of the intractable pace changes—it keeps executing fast but is executing less often. Depending on the actual step under consideration, a tractable pace can be changed in several ways:

- A step can keep on executing as fast as possible, but uses a buffer towards the next step with one unit of work and only proceeds if the buffer is empty. Thereby, the step is mostly idle but the succeeding step always has a unit of work to process. This can be compared to a *pull-based* processing model, which is also propagated by lean manufacturing.
- A step can implement just-in-time task completion, which means that the step waits until only P_{eff} is left before the completion of the succeeding step, at which the step starts to process its current unit of work, i.e. the unit of work is processed and immediately passed on. Thereby, a piece of work is never stored.
- Any other effective pace between as-fast-as-possible and the bottleneck pace is possible, i.e. the processing step can pick whatever is best—optimizing for energy usage, buffer allocation time or other metrics.

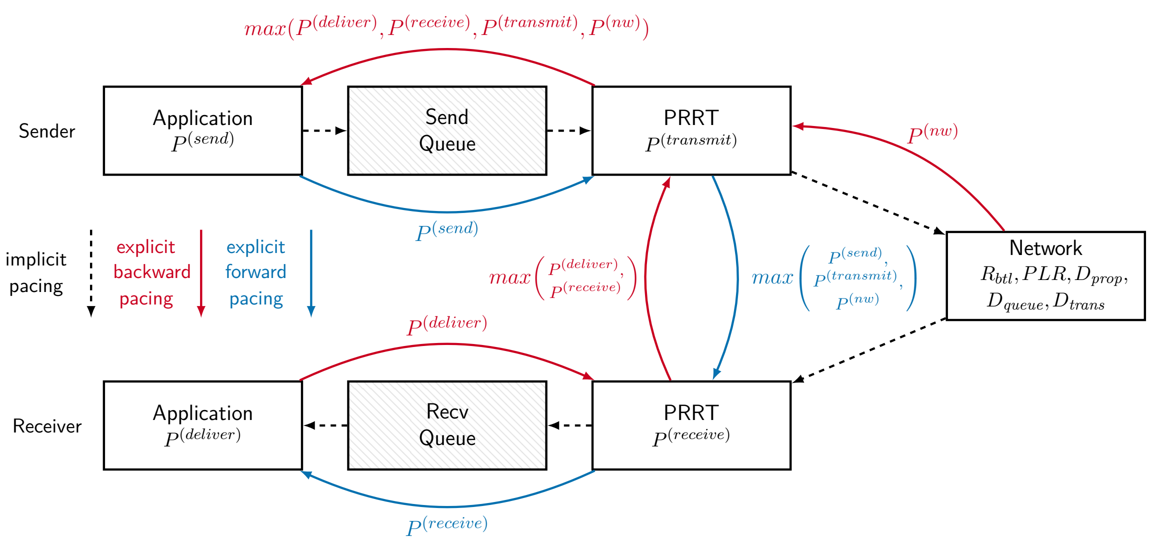

$max(P^{(deliver)}, P^{(receive)}, P^{(transmit)}, P^{(nw)})$
Sender
Application $P^{(send)}$
Send Queue
PRRT $P^{(transmit)}$
$P^{(nw)}$
$P^{(send)}$
Network $R_{btl}, PLR, D_{prop}, D_{queue}, D_{trans}$
implicit pacing
explicit backward pacing
explicit forward pacing
$max\begin{pmatrix}P^{(deliver)},\\P^{(receive)}\end{pmatrix}$
$max\begin{pmatrix}P^{(send)},\\P^{(transmit)},\\P^{(nw)}\end{pmatrix}$
$P^{(deliver)}$
Receiver
Application $P^{(deliver)}$
Recv Queue
PRRT $P^{(receive)}$
$P^{(receive)}$

Figure 6.4: Cross-layer pacing in PRRT communicates and adapts to paces across the complete chain.

More concretely, a pace can be changed by the following means: adding artificial processing delays, setting different configuration parameters in the transport stack (e.g. processor speed) or changing a network configuration (e.g. static WLAN data rates).

6.5 Cross-layer Pacing in PRRT

With the general architecture of cross-layer pacing in mind, this section presents the design of X-Pace, a concrete implementation inside PRRT (cf. Chapter 4).

6.5.1 Measuring Paces in PRRT

PRRT tracks the paces of different layers (cf. Figure 6.4) of the communication stack. For a given application, the packet size L is assumed to be fixed, while the pace P in which a step is executed is variable. As most most control application use payloads of fixed size and lower message rates are preferred over significantly larger end-to-end delays caused by queuing, it is appropriate to vary only P.

Pace Filter

The measurements for paces are put into statistical filters and three instances of a pace filter are kept for the internal, external and dependent portions of the pace.

The current implementation of the filter represents a windowed maximum filter (cf. Listing 6.1) and follows the design of the filters used in [CCG+16]. For performance reasons, the filter does not use an array to store samples, but instead a last-update timestamp and a current maximum value. When the configurable window size (in seconds)

```c
typedef struct prrtPaceFilter {
    prrtTimedelta_t window_us;
    bool valid;
    prrtTimestamp_t updated;
    prrtTimedelta_t value;
} PrrtPaceFilter;

void invalidate(PrrtPaceFilter* filter) {
    filter->valid = false;
    filter->updated = PrrtClock_get_current_time_us();
}

PrrtPaceFilter* PrrtPaceFilter_create(prrtTimedelta_t window_us) {
    PrrtPaceFilter* filter = (PrrtPaceFilter*) calloc(1, sizeof(PrrtPaceFilter));

    filter->updated = PrrtClock_get_current_time_us();
    filter->valid = false;
    filter->value = 0;
    filter->window_us = window_us;

    return filter;
}
// ----8<--------------
prrtTimedelta_t PrrtPaceFilter_get(PrrtPaceFilter* filter) {
    prrtTimestamp_t now = PrrtClock_get_current_time_us();
    if (filter->updated + filter->window_us < now) {
        invalidate(filter);
    }
    return filter->value;
}

void PrrtPaceFilter_update(PrrtPaceFilter* filter, prrtTimedelta_t value) {
    prrtTimestamp_t now = PrrtClock_get_current_time_us();
    if (filter->valid == false) {
        filter->value = value;
        filter->valid = true;
        filter->updated = now;
    } else {
        if (value > filter->value) {
            filter->value = value;
            filter->updated = now;
        } else {
            // ignore sample
        }
    }

    if (filter->updated + filter->window_us < now) {
        invalidate(filter);
    }
}
```

Listing 6.1: Code excerpt of the *Pace Filter* (cf. https://git.nt.uni-saarland.de/LARN/PRRT/blob/prrt-0.4.1/prrt/proto/stores/paceFilter.c).

has passed since the last update, the filter is invalidated. When invalid, it continues to return the most recent maximum. In this state, the filter accepts any new sample as the new maximum, updates the last-update timestamp, and validates the filter again.

While this represents a valid first implementation of the pace filter for all involved steps, future implementations should consider more sophisticated designs that are adjusted to the dynamics of the step-under-measurement and allow to compute a *representative current pace.*

Sender Application Pace

Abstractly, applications are processes that send messages with an average size of L ($[L] = \mathsf{B}$) and a frequency of f_{send} ($[f_{send}] = \mathsf{Hz}$), i.e. a pace of $P^{(send)} = 1/f_{send}$. These measures can be taken by tracking the `send()` calls on the PRRT socket (cf. [CCAP10]). The current time is stored on each call to later compute the application sending frequency. The application data rate is therefore $R_{send} = L \cdot f_{send}$ ($[R_{send}] = \mathsf{bit/s}$).

Receiver Application Pace

A receiving application can be abstracted as a process that consumes new messages with a frequency of $f_{deliver}$. The time between calls to `recv()` in the PRRT socket is tracked, but as this call is blocking, it is dependent on the paces before. The receiver application pace would always be considered as the bottleneck pace of the overall system as it cannot receive faster than data is delivered by the preceding chain. Therefore, the effective time between deliveries of the packet (the moment shortly before an application layer `recv()` returns) is tracked and the time spent waiting for the previous layer, the dependent pace, is subtracted.

Transport Protocol Paces

Both sending and receiving transport stacks 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. These code sections are used to execute all the encoding, error correction and additional functions. This happens in a sequential manner that allows to track the time these functions take to be executed on the current system. This yields the two paces $P^{(transmit)}$ and $P^{(receive)}$.

Network Pace

The network pace depends on the bottleneck data rate R_{btl} ($[R_{btl}] = \mathsf{bit/s}$) 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 (cf. Section 4.4). The network pace is $P^{(nw)} = \frac{L}{R_{btl}}$, the transmission delay per packet.

6.5.2 Controlling Paces in PRRT

For each step, the maximum of the step's pace and the pace received from the succeeding step is computed and propagated backwards. The same happens for the pace received from the preceding step: The maximum with the step's pace is computed and propagated forward. The pace is controlled at two locations: The link-layer transmission step within the protocol and through the sending application's API.

Automatic Pacing

Currently, the PRRT stack has a single location where the pace is enforced, namely in the sending of packets to the UDP socket. If the network or the receiver are the bottleneck, this slows down the transmission by inserting additional delay between packets.

Cross-layer Interface

The application can participate in the cross-layer pacing using different approaches, either directly or by means of the operating system:

- *Pacing-aware Applications:* The described system provides an interface for the application to query the current bottleneck data rate (`btl_dr_fwd`). Using this information, a pacing-aware application can fine-tune its parameters (e.g. sampling rate, sensor resolution) to adjust to the bottleneck. Such an application can use the `send_async()` function, where the control flow returns as soon as the packet is submitted to PRRT, but is potentially not sent on the network yet. In this scenario, it is the responsibility of the application to regularly query the current bottleneck pace and adapt to it appropriately.

- *Reactive Measures:* The system *detects* when the 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 applications need no modifications.

- *Proactive Measures:* The network *monitors* the application behaviour 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 behaviour. Such an application behaviour is generally detectable by monitoring the timing of function calls [CCAP10]. Thus, the `send_sync()` ensures that the application is able to immediately send the next packet when the call returns. As some applications require work to be done before the next call to `send_sync()`, this time is measured and the call returns earlier. Consequently, one complete cycle of application processing and send takes exactly as long as the bottleneck pace of the system.

6.6 Evaluation

In the evaluation (cf. [SRGP+19]), cross-layer pacing in PRRT is analysed in comparison to other transport layer protocols. The goal is to show that the bottleneck-awareness of PRRT and its ability to pace are beneficial for applications that demand predictably low latency. The following scenarios compare PRRT with several variants of TCP and evaluate their performance with different bottleneck locations. In the figures, time series of measured latencies during an experiment are shown, together with *cumulative density functions* (CDFs) to shed light on the distributions of these latencies. 99th percentiles are presented to show that tail latencies can be reduced and inter-percentile ranges between 1st and 99th percentile are used to show that the latency predictability is also improved by reducing the variation or *jitter*.

6.6.1 Methodology

Pacing is evaluated by comparing the inter-packet time (IPT) and the E2E application layer delivery time (DT) of individual packets sent using different transport protocols. The different scenarios put the bottleneck into different locations of the system. Based on this, theoretical optimal values can be derived for DT, IPT and experiment duration (EXP).

$$DT_{opt} = D_{prop} + P^{(nw)} \tag{6.1}$$

$$IPT_{opt} = max\{P^{(send)}, P^{(nw)}, P^{(deliver)}\} \tag{6.2}$$

$$EXP_{opt} = IPT_{opt} \cdot rounds \tag{6.3}$$

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

Testbed Setup

The evaluation uses the *OpenNetworking Testbed* as described in Section 2.4. The experiment topology is comprised of two physical hosts at Saarland University that run OpenvSwitch and execute the test application as a Docker container. Hosts are connected via a direct 1 Gbit/s Ethernet link. In most experiments, `netem` traffic shapers are used on the interfaces at both ends of the link to control its delay characteristics and make it the bottleneck, if intended. The *Precision Time Protocol* (PTPv2)[6] is used to make sure the system clocks are synchronized and the samples of time are reliable. Interference between the congestion-induced queues and the time synchronization is avoided by running the time synchronization out-of-band. This control path is also used to trigger the evaluations using SSH. Hosts run Ubuntu 16.04 and Linux kernel 4.15, which incorporates a recent version of the BBR congestion control algorithm for TCP by default.

[6]https://www.eecis.udel.edu/~mills/ptp.html (accessed April 25, 2019)

Measurement Application

The evaluation employs a minimalistic time-measuring application to capture the inter-packet time (IPT) in the sender application as well as the E2E delivery time (DT) a packet takes from the sender to the receiver application. For all transport layer implementations, the application layer is identical and sends fixed size messages containing the timestamp and the round number. This resembles a sensor application in distributed CPS that reports its value to the controller. The sending application captures the current time, composes a message, and calls `send()/send_sync()`, which can block for both TCP and PRRT. Thereby, the cross-layer interface mentioned in Section 6.5.2 is implemented. At the receiving application, the packet is taken out of the socket and the current time is used to calculate the DT between a message being put into the socket and it being received on the other side. On the sender side, the time differences between calls to `send()` yield the IPT.

6.6.2 Parameter & Protocol Tuning

Several TCP socket parameters and kernel options are fine-tuned to make a fair comparison between TCP and PRRT.

The sending queue size of PRRT is exactly one packet, while the TCP socket option `SO_SNDBUF` is set to different values depending on the scenario. `SNDBUF` in TCP limits the total amount of unacked data in-flight. The PRRT queue works differently and is only used to decouple application `send()` and link layer `transmit()`.

PRRT uses a receive buffer between the receiving part of the stack and the application that is not limited in terms of bytes. Instead, PRRT drops packets that are not delivered to the application in time. For all experiments, this limit is set to 100 ms. The receiver buffer `SO_RCVBUF` for TCP is varied for the different scenarios to enable or disable flow-control—a method to implement pacing through backpressure.

The TCP sender has `TCP_NODELAY` and `TCP_QUICKACK` activated to avoid any end-host aggregation of data or ACKs. Additionally, TCP timestamps and SACK are disabled and the `low_latency` option is enabled. Data is passed to the TCP socket using `write()` to set the `PSH` flag, telling the end-host to deliver it immediately.

The pace filter in PRRT (cf. Section 6.5.1) is configured to have a window size of 2 s in our evaluations, as observed system load and network dynamics in our testbed were relatively stable during periods of this length.

Evaluating Optimized TCP

Figure 6.5a shows the effect the different options have on the ability of TCP to pace packets. Sending and receiving application run as fast as possible, and the network is configured to be the bottleneck. The ideal IPT is 5 ms, the network pace in this scenario where 1000 B packets are sent over a 1.60 Mbit/s link. The one-way propagation delay is 15 ms, relating to a BDP of 6000 B. For the optimized TCP variants, `SNDBUF` and

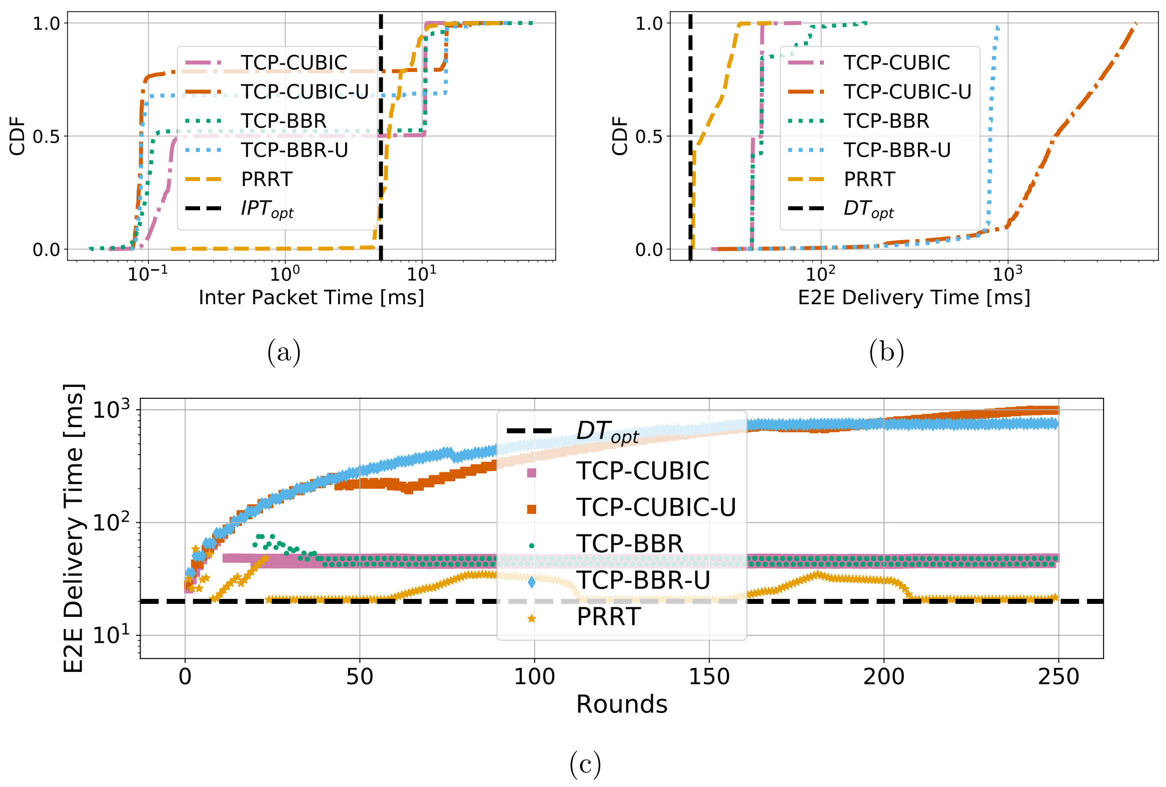
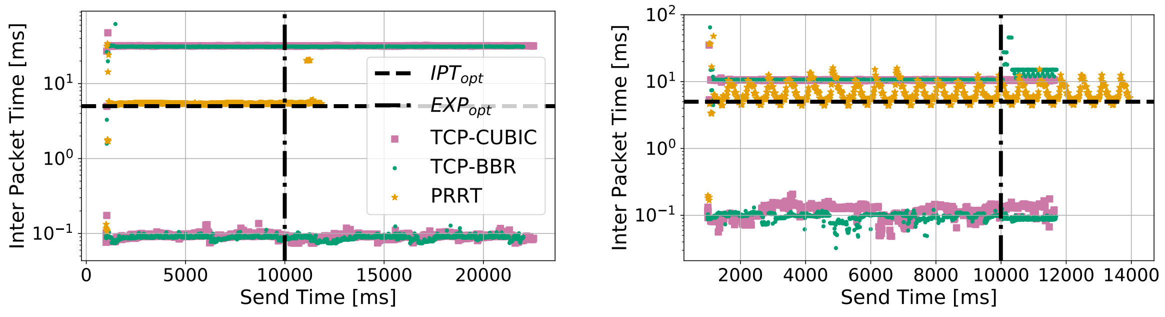


Figure 6.5: Measurements of inter-packet and delivery times for PRRT, as well as optimized and unoptimized (-U) variants of TCP.

RCVBUF both are set to $3 \times BDP$, allowing congestion control to work but also limiting the amount of data in the queues for TCP.

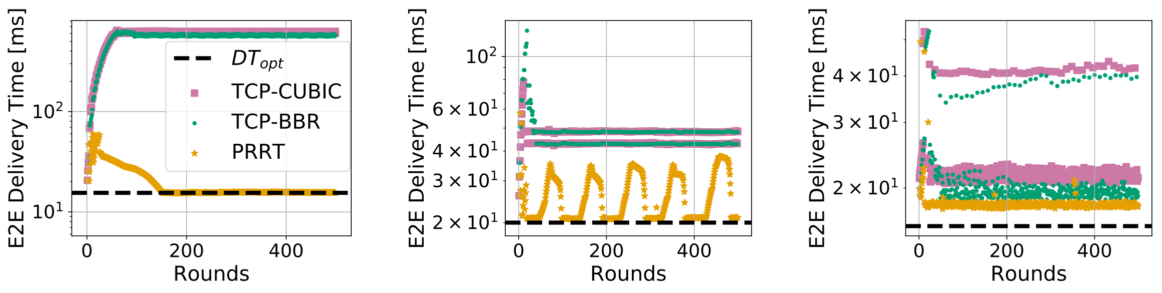
Buffer-size and kernel-level optimizations lead to nearly bimodal distributions for all TCP variants, with CUBIC [HRX08] and BBR performing only slightly differently. PRRT narrows the distribution significantly with 4.40 ms and 12.79 ms as 1st and 99th percentiles. The distribution for PRRT is skewed towards slower paces caused by a conservative approach to avoid filled buffers as much as possible.

Figure 6.5b shows the E2E delivery time, where the ideal line is at 20 ms, consisting of 15 ms one-trip propagation delay and 5 ms network pace (aka transmission delay). PRRT can approach this bound, but periodically deviates from it (cf. Figure 6.5c). The reason for this are probing periods for a higher share of the data rate by increasing the pacing rate and thereby filling buffers. This oscillation can be avoided in scenarios with static links and no contention, but if the data-rate probing function of congestion control should be part of the protocol, this cannot be avoided. The TCP series also display the differences in the congestion control algorithms—depending on their parameters. Unoptimized variants fill the buffers significantly, leading to seconds instead of milliseconds of E2E delay. The optimized versions, in contrast, constantly perform at around 45 ms.



(a) Receiver (b) Network

Figure 6.6: Inter-packet times for different bottlenecks.



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

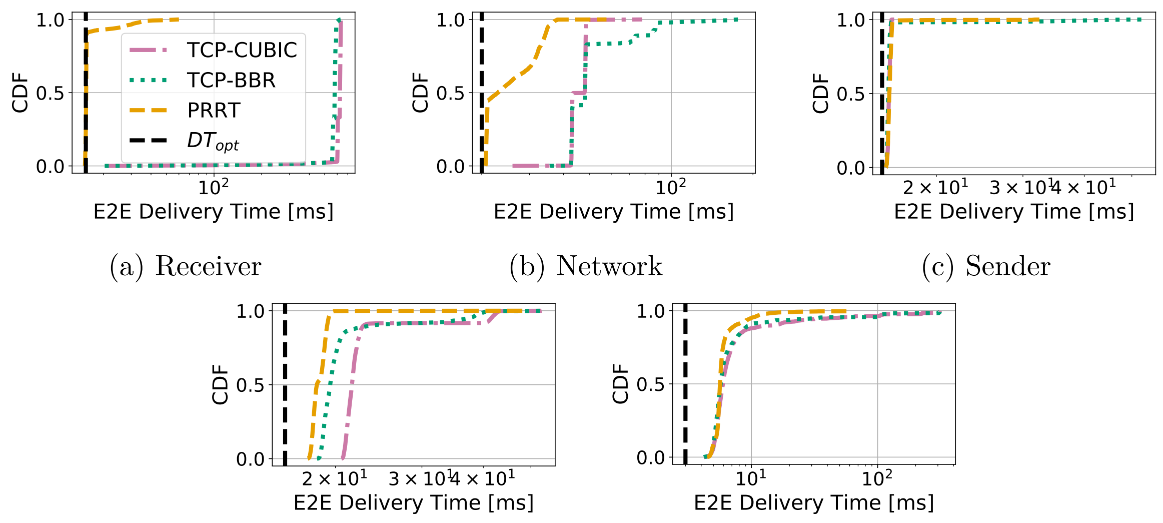
Figure 6.7: E2E Delivery Times for different bottlenecks.

6.6.3 Receiving Application as Bottleneck

This scenario is an end-to-end test as the bottleneck pace must be propagated through the complete communication pipeline. Therefore, the receiving application is turned into the bottleneck by adding an artificial delay of 5 ms (with 20 % jitter) between calls to `recv()`, resulting in $P^{(btl)} = P^{(receive)} \approx 5\,\mathrm{ms}$. Packets are sent with a size of 1000 B at a network rate of 16 Mbit/s (i.e. $P^{(nw)} = 0.50\,\mathrm{ms}$). The configured one-way propagation delay is 15 ms, which results in *BDP* of 60 kB. The `SNDBUF` of TCP is set to this value so that this buffer is not a constraint. The `RCVBUF` is instead set to the “*BDP*” of the receiving application (6 kB), based on the receiver pace of 5 ms and RTT of 30 ms.

Using this approach, TCP’s flow control can cause backpressure, implementing a variant of pacing compatible with standard TCP. The pace of the sending application can be arbitrarily fast, as it is only bounded by the processing speed of the sender. Nevertheless, it is going to be throttled down by cross-layer pacing during the experiment.

Figure 6.6a shows the IPTs of the different protocols during the evaluation—displaying the first 2000 packets of an experiment run. PRRT meets the optimal pace of 5 ms after a short startup of about 200 ms. For PRRT, the overall time until experiment completion (11.85 s) is close to the optimum ($EXP_{opt} = 10.00\,\mathrm{s}$), but far away for TCP CUBIC (22.61 s) and TCP BBR (22.53 s). Both TCP variants have IPTs that are either signif-



(a) Receiver (b) Network (c) Sender

(d) Internet (e) WLAN

Figure 6.8: CDFs for E2E Delivery Times for different bottlenecks.

icantly higher (30 ms) or lower (0.10 ms) than the optimum of 5 ms, which causes these inflated completion times. PRRT needs a startup phase until the optimal DT of 15 ms is achieved and sustained for the rest of the experiment (cf. Figure 6.7a).

Hence, PRRT is avoiding queues as much as possible, in contrast to TCP, which fills buffers that lead to a constant delay of around 60 ms. The cumulative density function in Figure 6.8a further shows this near perfect pacing of PRRT except for a short period caused by the startup.

6.6.4 Network as Bottleneck

This scenario puts the bottleneck at the network level, hence the sender must be instructed to slow down the sending rate. The one-way propagation delay is 15 ms and there is no artificial delay at application layer, but a limit in the data rate of the network of 1.60 Mbit/s. This yields a *BDP* of $1.60\,\text{Mbit/s} \cdot 30\,\text{ms} = 48\,000\,\text{bit} = 6000\,\text{B}$ and a network pace of 5 ms for packets of 1000 B size. Both `SNDBUF` and `RCVBUF` are set to $3 \times BDP$ so that congestion control can work and potentially cause buffers to fill if more than one *BDP* is in-flight. The first 2000 packets of a communication are shown, while the remaining evaluation follows the same pattern.

Figure 6.6b shows that PRRT is able to perform close to the optimum. The IPTs in this scenario oscillate, in contrast to the previous evaluation with the receiver side bottleneck. This oscillation is caused by PRRT's BBR implementation probing for a large share of the bottleneck data rate using a faster pace. A similar oscillation, this time with respect to the DTs, can be seen in Figure 6.7b. The faster probing pace causes queuing at the bottleneck so that the delivery time for these packets is increased. However, PRRT is still able to achieve significantly lower DTs because it only causes

```
traceroute to 79.199.28.123 (79.199.28.123), 30 hops max, 60 byte packets
 1  vlan-herfet-neu.nt.uni-saarland.de (134.96.86.1)  0.400 ms  0.358 ms  0.420 ms
 2  c65eb36-win.net.uni-saarland.de (134.96.6.54)  0.328 ms  0.314 ms  0.384 ms
 3  cr-dui1-te0-5-0-2-4.x-win.dfn.de (188.1.241.185)  8.687 ms  8.688 ms  8.680 ms
 4  cr-fra2-be16.x-win.dfn.de (188.1.144.178)  9.618 ms  9.732 ms  9.727 ms
 5  ffm-b12-link.telia.net (213.248.97.40)  9.226 ms  9.563 ms  9.191 ms
 6  ffm-bb3-link.telia.net (62.115.142.46)  10.069 ms  9.761 ms  9.704 ms
 7  ffm-b4-link.telia.net (62.115.120.6)  9.802 ms  9.792 ms  10.024 ms
 8  dtag-ic-319284-ffm-b4.c.telia.net (213.248.93.187)  10.374 ms  10.175 ms  10.171
 ↪  ms
 9  91.23.246.213 (91.23.246.213)  13.230 ms  13.252 ms  13.234 ms
10  * * *
...
30  * * *
```

Listing 6.2: Traceroute from Saarbrücken to Homburg (a 20km straight-line distance), with intermediate stops in Dui(sburg) and Fra(nkfurt).

probing queues and no queues due to inadequate buffer management as is the case for TCP. This is also visible in Figure 6.8b, where TCP operates between 40 and 50 ms, while PRRT is way closer to the optimum.

6.6.5 Sending Application as Bottleneck

This evaluation aims to investigate if both PRRT and TCP perform optimally in case that cross-layer pacing is not required as the system is inherently paced. The setup is again as in the receiver bottleneck scenario (Section 6.6.3), except that an artificial delay of 5 ms (again with 20 % jitter) is added to the sender application instead of the receiver. Sending and receiving buffers are configured to be the *BDP* relating to the 16 Mbit/s data rate and 15 ms one-way propagation delay. Figure 6.8c shows that both TCP variants as well as PRRT are now able to achieve a near-optimum DTs. Consequently, the TCP optimization by setting specific kernel options leads to “as-fast-as-possible” forwarding, without buffering or increased latencies.

6.6.6 Internet as a Network Bottleneck

To show the performance of cross-layer pacing on the Internet, this evaluations transmit across a straight-line distance of 20 km, using a residential access as one side and the testbed as the other. A traceroute (cf. Listing 6.2) reveals that the actual covered distance is much higher and includes multiple hops at Internet Exchange Points. Regarding the channel parameters, the one-way ping delay is 15 ms and iperf3 yielded a sustainable data rate of 10 Mbit/s. Hence $P^{(nw)} = 0.80\,\mathrm{ms}$ and $BDP = 10\,\mathrm{Mbit/s} \cdot 30\,\mathrm{ms} = 37.50\,\mathrm{kB}$ and again $3 \times BDP$ for the TCP buffers is used on both sides.

Figure 6.7c and Figure 6.8d show TCP BBR achieving lower latencies than TCP CUBIC, being inline with the intentions of the BBR authors [CCG+16]. PRRT does not approach the optimum as close as in previous experiments. The reason for this gap could be the precision of the NTP-based time-synchronization between the end-systems. This unavoidable imprecision affects all evaluated protocol variants and could cause a constant systematic error on the absolute timescale. However, the shape of the CDF for PRRT suggests that latency is more predictable and stable as for the TCP variants that have a heavy tail. In consequence, 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.

6.6.7 Wireless LAN as a Network Bottleneck

The final evaluation tests the approach in an 802.11ac wireless LAN, composed of an access point and two nodes. Figure 6.8e shows the results of this evaluation in terms of E2E delivery times. Curves are closer to each other, with 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).

6.7 Conclusion

Despite the controversy around pacing, it is evident that the pace at which packets are sent to the network is essential for low-latency applications. Using pacing allows to gain major benefits, e.g. near-zero queueing delay as well as just-in-time processing. With cross-layer pacing, this approach is extended to the system layer, allowing all processing steps between two applications to know about their paces and to synchronize them. X-Pace provides an implementation of cross-layer pacing in PPRT that is shown to significantly reduce latencies and increase latency-predictability over TCP variants optimized for low latency—even over the public Internet.

"My motives, as ever, are entirely transparent." Hughnon reflected that 'entirely transparent' meant either that you could see right through them or that you couldn't see them at all.

Terry Pratchett, The Truth

Chapter 7

Transparent Transmission Segmentation

While Chapter 5 and Chapter 6 have mainly focused on solutions for latency-awareness and predictability using changes at the end-hosts, this chapter investigates a complementary solution that shows how these properties can be improved by altering components *inside* the network. Having such solutions in the network can support both CPS and other applications that require reliable timing and are facing heterogeneous multi-hop communication paths.

7.1 The Vices and Virtues of the End-to-End Principle

In the 1980s, when the Internet was established together with the nowadays predominant *Transmission Control Protocol* (TCP) [Pos81], many components in packet-switched networks were unreliable. In order to compensate for this, TCP implements reliable transport in adherence to the *end-to-end* (E2E) principle as stated by Saltzer et al. [SRC84]. The authors of this classic work claim that E2E is the *only* way to provide reliability along a (network and system) transmission path that is composed of unreliable components. Another central design principle of the Internet is to have a *dumb core* and a *smart edge*, allowing the core to stay the same while applications evolve.

Today, many of the motivations behind the work of Saltzer et al. are no longer true. Memory is increasingly reliable and wired communication channels have extremely low error rates—thanks to improved physical facilities and the change of typical network topologies to more reliable ones, e.g. in the context of data centers [GHJ$^+$09]. End-to-end paths are nowadays composed of a vast range of link types with different latencies, data rates and loss characteristics. In addition, a significant number of middleboxes have been developed over the years, effectively making the network core not as *dumb* as initially planned (consider e.g. WAN accelerators or various NAT incarnations). In

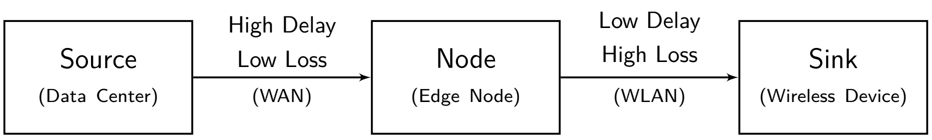


Figure 7.1: A typical streaming scenario where a lossy last WLAN link causes increased delays through end-to-end retransmission—though retransmission would only be necessary for the last hop.

consequence, adhering to the E2E paradigm has performance drawbacks and deviating from it can reduce, for instance, transmission latencies (cf. [SRC84, Moo02]).

End-hosts do not “see” differences in link parameters as only one virtual link with accumulated parameters is observable. An example of a situation where this is problematic can be seen in Figure 7.1, where a typical streaming scenario is shown[1]. End-users operating a wireless device (e.g. a smartphone) are attached to an edge node (e.g. an access point) via a link that has a relatively short delay but a comparably low reliability. In contrast, the link from the edge node to the cloud providing the content is longer in terms of delay but more reliable as it uses wired, guided media. Operating error control in the end-to-end mode is bound to cause retransmissions over the complete path, though the loss is more likely to have happened on the last (and short) link. If we use protocols with proactive error control using forward error coding, an E2E scheme has to pick a code rate that is suitable for the composed link; consequently over- or under-estimating the redundancy information required for both links individually.

These heterogeneities in link parameters cannot be exploited using an E2E mode of operation, so it is necessary to split the connection—an approach that is going to be intensively discussed in the remainder of this chapter. While this approach is well-known through traditional works, e.g. on Split-TCP [KKFT02], it is only by the advent of network softwarization and an increased interest in edge-computing and latency-critical CPS applications that this scheme can achieve wide deployment and serve a great number of users in different network scenarios.

7.2 Transparent Transmission Segmentation

Knowing about the deficiency of the end-to-end principle with respect to run-time performance and the impact on different transport layer functions, this section proposes the *transparent transmission segmentation* (TTS) approach that aims to overcome these inefficiencies without disrupting network architectures that have been build with the E2E principle in mind.

[1]Control applications face similar timing problems, as they share the demand for low communication latency and low age-of-information.

7.2.1 Requirements

For the TTS approach to be deployed widely and to sustainably improve network performance, a set of requirements must be fulfilled, also briefly covered in [SH16b]. These include transparency towards different network components, ease of deployability, as well as providing sufficient utility improvements.

Transparency In order to allow straightforward deployability on the Internet, two things are mandatory when segmenting a transmission: (a) end-hosts should not be required to change, e.g. by modifications of userspace or kernel code, and (b) the wire format that end-hosts use should still be in line with the protocol specifications, i.e. the RFCs. Complying with this increases the acceptance of the approach and enables gradual upgrades of network core systems to use this approach. Thereby, it is *transparent* to the end-host whether the connection is segmented or not, apart from a change in the observed *quality-of-service* (QoS), i.e. achievable data rate, reliability, and latency. Such an approach is a transport-layer *Performance-Enhancing Proxy* (PEP), as described in RFC3135 [BKG$^+$01]. While not strictly being a transport protocol, many applications run on-top-of HTTP, which is effectively terminated at HTTP proxies—a widely used application-layer PEP component on the Internet [LL15].

Feasibility While the requirement towards transparency implies that no implementation effort at the end-host should be necessary, it is also advisable that a TTS implementation poses *minimal effort* in the network core. This effort comes in various forms, e.g. additional software running on switches, increased number or complexity of protocols, or additional devices to be deployed. In consequence, a feasible solution keeps the network mostly untouched and incorporates only minimal changes.

Utility By design, TTS adds overhead as the segmentation is more complex than the pure forwarding of data. Furthermore, not all network functions profit from TTS in a certain scenario and a certain TTS configuration (i.e. position of segmentation points and used intermediate buffer sizes). If there is one function that does not profit from TTS, it does not in general mean that applying TTS is detrimental to the overall performance. This is because other functions might perform better or that negative effects can be avoided or compensated for. Thereto, it should be possible to decide on a per-link basis to use TTS or not. This drives the need for heuristics to predict the benefits and drawbacks induced by TTS and to guide the process of placing segmentation points and parametrizing them (e.g. buffer sizes or transmission parameters).

Fairness Considering that the TTS approach is targeted at networks that are (a) used by multiple parties or (b) might be transit networks of ISPs, it is desirable that the approach retains suitable forms of fairness. In the congestion control community, new algorithms are evaluated by comparing their behaviour to generic models of TCP transports. For TTS, these aspects have to be considered and the goals of the operators must

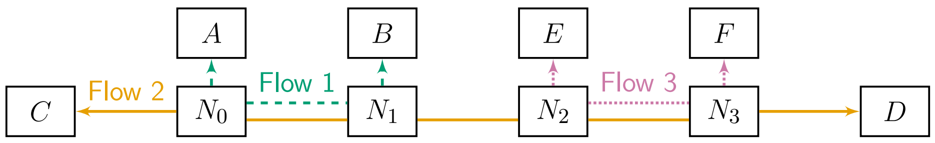


Figure 7.2: Contention is a transitive relation, i.e. even though Flow 1 and Flow 3 have no shared link, they contend through the intermediate Flow 2.

be taken into account—especially because we know, not just from [ZMSS19], that not all properties can be maximized with a single congestion control algorithm.

Security TTS is similar to a *man-in-the-middle*, as it reroutes traffic through an additional component. There are three reasons why—despite of this similarity—TTS should not be considered harmful to security: First, the manipulation happens on transport layer information units (TCP segments or UDP datagrams) and it is not much different from what load balancers, NATs, or other devices that change routing and forwarding do. Second, the fact that TTS works on the transport layer means that services such as transport layer security (TLS) work *on-top* of the TTS approach. Third, the parties that apply TTS are assumed to be in charge of operating the current (sub-)path of the communication so they are privileged to do this kind of manipulation. In consequence, it needs to be ensured that the software implementation required for TTS is secure and has no vulnerabilities, but the approach itself does not impede security.

7.2.2 Dimensions

Single virtual segments—as visible to the transport layer—do not expose the heterogeneities of link parameters but resemble a non-decomposable aggregation. TTS allows to split these segments in a way that one or more homogeneous links can form a segment to which the transport layer functions are tuned. Whether two links are homogeneous or not is decided based on a number of parameters, now called *dimensions*.

Latency There are multiple components that contribute to the overall end-to-end latency (cf. Chapter 3). On a link level, we have differences in terms of propagation latency as well as the transmission latency. Being aware of the distributions of the different latencies on a link allows splitting the end-to-end path into segments. This split enables RTT-dependent network functions, in particular congestion and error control, to operate on the segment-local RTT.

Contention In circuit-switched or deterministic packet-switched networks, the problem of multiple access and contention is solved at design or admission time. In the networks we target, i.e. predictable statistically multiplexed packet-switched networks, connections and their flows compete for medium access time and data rate. As we focus on the transport layer, we are primarily concerned with the congestion window and in turn the sending rate that is constantly adapted due to congestion control. These distributed algorithms aim to converge to a fair share of the available channel—a process that takes some time and might take even longer than the actual flow needs to complete.

Contention itself happens on any link on the end-to-end path, making flows that share at least one link *directly contending*. Apart from link congestion, there is also flow-state congestion at in-network nodes, which is further discussed in [TS12]. Contention is a transitive relation, as can e.g. be seen in Figure 7.2, and even with segmentation, this transitivity cannot be completely removed. Nevertheless, the segmentation reduces the strength of the influence between flows that are not sharing a common link.

Resilience The fundamental resilience of a link is determined by the physical properties and medium access technologies used to implement it. A further source of unreliability is loss caused by congestion events. Treating resilience on an end-to-end level for heterogeneous links leads to either over- or underestimations of the required redundancy information to achieve reliable transport. With segmentation, it is possible to adjust to individual links with stronger or weaker error coding. In consequence, both congestion and error control are affected by the (visible) resilience of an individual link.

Buffers Considering the end-to-end path, many links and intermediate processing steps have buffers to compensate for bursty workloads or traffic. By segmenting, we (a) enable the targeted treatment of individual buffers and (b) can introduce additional buffers in case that performance improvements that require larger buffers are desired. TTS is concerned with the fill-level of buffers as well as their total capacity, which have influence on congestion control and flow control.

Data Rates Maximum data rates differ at different links due to their technology and demand patterns. Regarding the end-to-end path, there is exactly one link with the bottleneck data rate at any time—defining the maximum achievable long-term throughput when using this link. Naturally, as data rates fluctuate, the bottleneck link can change, too. Buffers, as mentioned before, cannot increase this capacity, but allow to compensate for bursts and get closer to the bottleneck rate—but at the expense of increased latency or age of information (cf. Chapter 6). The effective data rate also depends on the previously mentioned resilience of a channel, in combination with the used FEC blocklength [PPV09].

7.2.3 Impact on Transport Layer Functions

Considering the aforementioned dimensions, we take a look at individual transport layer network functions (introduced in Section 3.3) and how they are affected by segmentation on a theoretical level. Later, we are going to evaluate these effects in Section 7.5.

Error Control The choice of error control parameters, e.g. retransmission timeout for ARQ and code rate as well as blocklength for FEC, heavily depends on the link parameters (cf. Section 4.5). As TTS *localizes* the error control, one can choose parameters that are suitable for the segment at hand, allowing to, e.g., fine-tune the redundancy information by choosing an appropriate FEC code rate. End-to-end error control based on ARQ incurs retransmission over the complete path again, even though segments have been successfully traversed before. In consequence, it is advised to let error control work on local segments instead of one E2E segment (cf. [KH14]) as evaluated in Section 7.5.2.

Congestion Control Many traditional congestion control algorithms increase the congestion window upon reception of a new ACK (e.g. TCP Tahoe, Reno), a process that happens in a frequency that is relative to the RTT between both end-systems. With TTS, this RTT is reduced and the ACKs arrive faster, allowing more frequent and timely advancements of the congestion window.

Algorithms such as CUBIC [HRX08] derive the current congestion window from the current time and not the frequency of ACKs, making it *less* RTT-dependent. The latter is because the feedback frequency and retransmission timeouts still rely on the RTT. While the RTT-independence of a CCA does not allow TTS to achieve the same gains as with other CCAs, the CCA can still benefit from TTS.

Furthermore, loss events happen on the local segment, which means that the congestion window is only reduced on this segment and not over the whole path. Preceding and succeeding segments can therefore keep their window and continue sending at the same rate. Nevertheless, a reduction of throughput in a intermediate segment might lead to a succeeding segment starving due to a lack of data; or a preceding segment ceasing to send due to flow control, when the buffer prior to the congested segment runs full. But as the preceding segment is only limited by the receive window value, it can quickly resume sending at the last congestion window value as soon as the congested segment has recovered and can read from its buffer.

Flow Control While congestion control tries to avoid dropped packets *inside the network* due to full queues, flow control avoids that packets *at the receiver* must be dropped due to a full buffer. In TCP, this means that the size of the receiver buffer is limiting the bytes of data that can be in transit, which effectively limits the throughput of the application. How to go beyond this limit using TTS is to be shown in Section 7.3.2. The reduced RTT also shortens the interval in which the fill level of the receiving buffer is communicated, which means that the sending side can react quicker. This results in more bytes in flight on the first as well as the second link.

Rate Control Apart from controlling the amount of data in-flight through flow and congestion control, a separate control parameter is the short-term sending rate at the sender side. When e.g. the first link has a higher maximum throughput as other links or processing steps, it is possible that the sender causes bursts of data that can be harmful

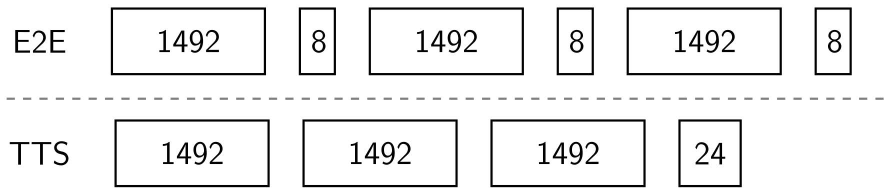


Figure 7.3: A switch that connects links with MTU 1500 B and 1492 B leads to IP layer fragmentation (in this example, three 1500 B packets are sent). For E2E, this creates small packets (two outgoing packets per incoming one), while TTS can accumulate data and attempt to fill a full MTU.

for the rest of the processing and communication pipeline. With TTS, it is possible to do rate control via pacing (see also Chapter 6) within the network, as pointed out by Aggarwal et al. [ASA00]. While this can help to spread out bursts of data to avoid losses due to buffer overflow, it adds latency for aggregation and pacing—something that might be deterrent if striving for low latency.

Ordered Delivery TCP, for instance, ensures that bytes leave the receiver socket in the exact same order they entered the sender socket. Thereto, the receiving end-system buffers out-of-order packets and only forwards them when the gap is filled. With TTS, this form of *reordering* can already be achieved inside the network. For certain scenarios, this can be beneficial but especially for low-latency applications this can be detrimental, as additional time is spent on aggregation for reordering.

Fragmentation Due to flow and congestion control, as well as implementation details on when to flush a buffer, TCP fragments the messages of an application to fit into multiple packets. TTS changes these boundaries as the stream of data is reassembled at the segmentation point and newly fragmented afterwards. Furthermore, TTS also has an impact on the IP fragmentation as new packets are created. On the IP layer, it is possible that one switch creates lots of fragments (cf. Figure 7.3), e.g. because it is connecting a link with an MTU of 1500 with one with MTU 1492[2]. Each incoming packet results in exactly two outgoing packets with a highly skewed packet length and a high header overhead for the second packet. Using TTS, the fragmentation can create fewer packets and is able to reduce the reassembly effort at the receiver.

7.3 A Formal Approach to TTS

Before we have a close look at the implementation of TTS in softwarized networks, we formalize the segmentation process and shed light on theoretical derivations that can be made from this.

[2]Typical values for Ethernet and DSL in Europe.

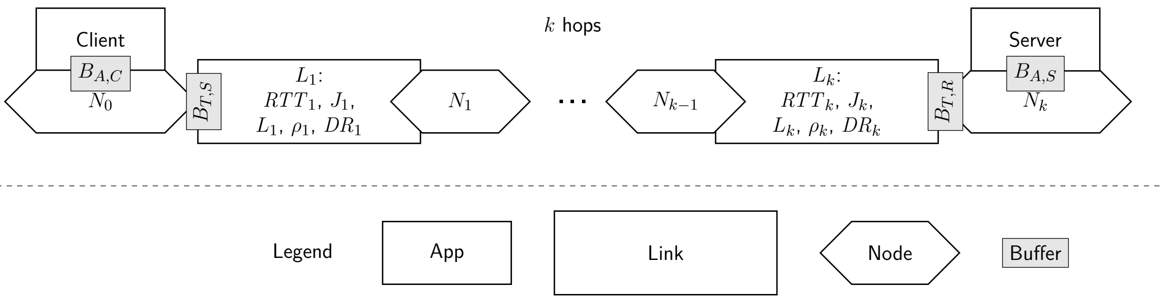


Figure 7.4: On an abstract level, the E2E path can be decomposed into k hops connecting $k+1$ systems with various link and system parameters.

7.3.1 Formalization

In order to quantify the impact of TTS on a theoretical basis, we first model the links, nodes, and paths in the network (cf. Figure 7.4). While there are more parameters that affect transmissions, only those that are observable for a network layer are considered.

Paths and Transmissions When arguing about transmissions, paths composed of multiple observable links are considered. The transmission traverses k hops or links, which in turn connect $k+1$ nodes by a path. Nodes are denoted N_i with $i \in [0,k]$ and links are denoted L_i with $i \in [1,k]$. N_0 is the source of a transmission, while N_k is the sink. Such a path is split at segmentations $S \subset \{N_i : i \in [1,k-1]\}$. A segmentation s is implemented using a PEP.

Link Measures Each observable link—either purely physical or a logical composition of many—has a number of properties that can be measured. Regarding timing, the *round-trip time* RTT ($[RTT] = \text{s}$) and its variation as *jitter* J ($[J] = \text{s}$) are captured. Furthermore, the link's resilience in terms of *packet loss rate* L ($[L] = \%$) and *loss correlation* ρ ($[\rho] \in [-1,1]$) is given. In our cases, a simplified Gilbert-Elliot [Gil60, Ell63] model $SGE(\alpha,\beta)$ is used to model the correlation $\rho(SGE(\alpha,\beta))$, which has been fitted to erasure sequences [Gor12]. More details and a derivation of multi-hop SGE are in Appendix A. Finally, transmission capabilities in terms of *data rate* R ($[R] = \text{B/s}$) and its *utilization* U ($U = \frac{R_{actual}}{R_{max}}, [U] = 1$) can be measured.

For the virtual single link that is visible to the application, the following holds:

$$RTT_{E2E} = \sum_{i=1}^{k} RTT_i \tag{7.1}$$

$$L_{E2E} = 1 - \prod_{i=1}^{k} (1 - L_i) \tag{7.2}$$

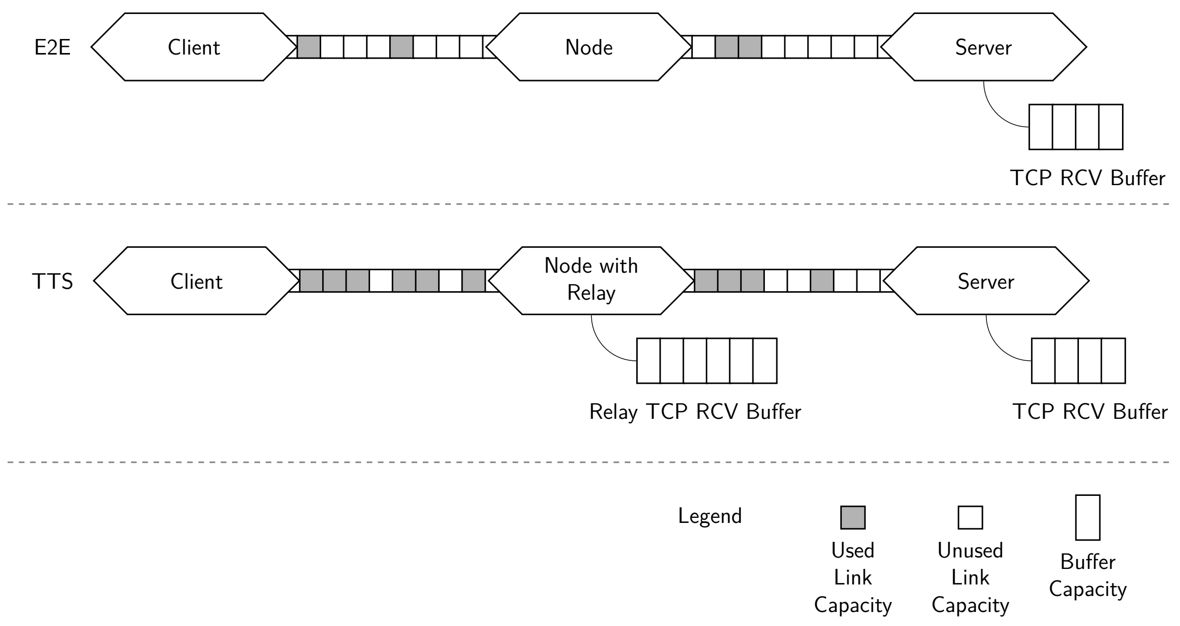


Figure 7.5: Small TCP receiver buffers limit the amount of data in-flight per RTT ($B_{recv} = 4$ in this case). TTS can increase utilization and hence throughput by (a) terminating the connection at the PEP leading to a shorter RTT and (b) providing buffer at the intermediate node that is larger than the one at the destination ($B_{relay} = 6$). For this example, this increases the overall utilization from 4/20 to 10/20.

$$R_{E2E} = \min_{\forall i \in [1,2,\ldots,k]} R_i \tag{7.3}$$

$$\rho_{E2E} = \rho(SGE(\alpha_{E2E}, \beta_{E2E})) \tag{7.4}$$

Node- & Connection-based Measures When communicating, data travels across many nodes and is put into many different buffers, whose size is denoted by B in Byte. While there are buffers on all layers of the network stack, the application layer buffers (B_A) as well as transport layer buffers, such as send ($B_{T,S}$) and receive ($B_{T,R}$) buffers for TCP, are of particular interest for TTS. These buffers reside on a node, but they are connection-based, i.e. a system would allocate individual send and receive buffers per TCP connection.

7.3.2 Theoretical Results on Flow Control

Flow control (cf. Section 3.3) is an essential network function in TCP, in particular when dealing with constrained devices [BEK14] that have small buffers, e.g. embedded IoT devices. Having little memory, the size of the receive buffer B_{recv} can be so small that it has an impact on the achievable throughput—in case the application consumes faster than $\frac{B_{recv}}{RTT}$ and regularly idles due to an empty buffer. We consider a two-hop scenario[3] as depicted in Figure 7.5, where the number of packets in flight is limited by the size of the TCP receive buffer at the receiver (B_{recv}). The following section derives the effective data rate that can be used considering this buffer.

Preliminary note on Practical Relevance In sense-and-act scenarios, which are common in IoT, is in unlikely that the constrained IoT device is the *sink* of large-scale flows. An exception to this are *software updates*, which might be large in size and need to be downloaded to the device in a reasonable time. So while most of the in-field operation time might not need TTS, system upgrades can still benefit from it.

Effective Data Rate in E2E Flow Control Scenarios

A TCP connection's throughput is limited by the maximum amount of in-flight data on the links, hence the maximum rate R_{max} is the one of the bottleneck link:

$$R_{max} = min(R_1, R_2) \tag{7.5}$$

The actual rate at which TCP is operating depends on two of its functions, namely congestion and flow control. Congestion control aims to avoid overflow at in-network buffers, by measuring and approaching the channel's data rate ($R_{channel}$) as close as possible, so that we can define the congestion control rate R_{CC} as:

$$R_{CC} \coloneqq R_{channel} - \epsilon \quad , \quad R_{CC} \gg \epsilon > 0 \tag{7.6}$$

The flow control rate R_{FC} depends on the size of the receiver buffer and derived from the BDP (R = data rate, RTT = round-trip time, N = number of bytes in flight):

$$\underbrace{R_{FC} \cdot RTT}_{BDP} = N \Leftrightarrow R_{FC} \coloneqq \frac{N}{RTT} \tag{7.7}$$

As flow control ensures that there is no buffer overflow ($N \leq B_{recv}$), we know:

$$R_{FC} \leq \frac{B_{recv}}{RTT} \tag{7.8}$$

In order to control congestion and flow at the same time, TCP chooses the window size to be the minimum of the two, which in turn limits the data rate:

$$R_{eff} = min(R_{CC}, R_{FC}). \tag{7.9}$$

For now, we assume that the links have sufficient available data rate so that congestion control is not the limiting factor. This means that the flow control throughput is always smaller ($R_{CC} > R_{FC}$), hence $R_{eff} = R_{FC}$. The round-trip time RTT is composed of the individual RTTs on the two links, so we get:

$$R_{eff,E2E} = \frac{B_{recv}}{RTT_1 + RTT_2} \tag{7.10}$$

[3]In general, any segmentation can be considered as a two-hop scenario, where the hops before and after the segmentation are treated as an aggregated single link.

Effective Data Rate with TTS

For TTS, we effectively get two rates depending on the receiver buffer and relay buffer B_{relay}. Again, the bottleneck data rate is determining the overall throughput:

$$R_{eff,TTS} = min(\frac{B_{relay}}{RTT_1}, \frac{B_{recv}}{RTT_2}) \overset{(*)}{=} \frac{B_{recv}}{RTT_2} \tag{7.11}$$

The last equality $(*)$ can be ensured by design: The receiver buffer is fixed and small, e.g. because it is an embedded device, and the RTTs are fixed. Consequently, the design parameter is B_{relay}, which should be dimensioned such that the bottleneck link is the second and not the first link:

$$\frac{B_{relay}}{RTT_1} \geq \frac{B_{recv}}{RTT_2} \Leftrightarrow B_{relay} \geq \frac{RTT_1}{RTT_2} \cdot B_{recv} \tag{7.12}$$

When we consider the case of equality, we have found an ideal size for the relay buffer, namely:

$$B_{relay} = \frac{RTT_1}{RTT_2} \cdot B_{recv} \tag{7.13}$$

Comparing the rates, we get the following improvement factor Λ[4]:

$$\begin{aligned} \Lambda = \frac{R_{eff,TTS}}{R_{eff,E2E}} &= \frac{\frac{B_{recv}}{RTT_2}}{\frac{B_{recv}}{RTT_1+RTT_2}} \\ &= \frac{RTT_1 + RTT_2}{RTT_2} = \frac{RTT_1}{RTT_2} + 1 \end{aligned} \tag{7.14}$$

Considering limits, we can deduce that for $RTT_1 \gg RTT_2$, the improvement is significant, while for $RTT_1 \ll RTT_2$ there is no difference from the E2E case. If both RTTs are equal, the utilization of TTS would be doubled compared to E2E. We can deduce that in situations where flow control is the limiting factor—when congestion control alone could approach maximum data rate—a segmentation is always beneficial. In order to achieve this, the relay buffer must be sufficiently large, but given the receiver buffer size and delays, its size can be chosen appropriately, according to Equation (7.13). TTS mitigates the performance bottleneck caused by insufficient buffers and makes the network data rate the bottleneck again.

7.4 Relaying

Knowing about the requirements for successful TTS as well as having theoretical incentives to implement it, this section focusses on the practical aspects and design decisions of deploying TTS to actual networks. Regarding terminology, *segmentation* is the effect we want to produce and we do so by *relaying* traffic at *relay* components.

[4]($\Lambda > 1$: TTS better, $\Lambda < 1$: E2E better)

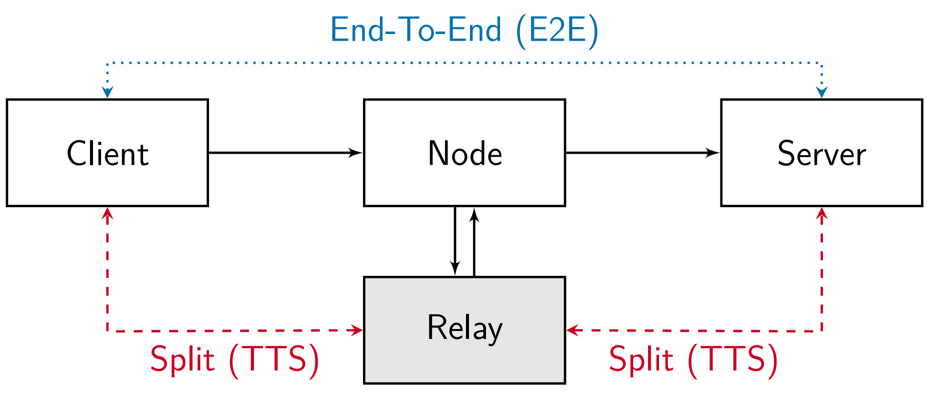


Figure 7.6: Applying TTS turns a E2E connection into two separate (but to the end-hosts invisible) segments.

7.4.1 Performance-enhancing Proxies (PEP)

RFC3135 [BKG+01] describes the concept of a PEP (our relay) and sketches possible implementations on various layers. In comparison to the transport layer PEPs, *application layer PEPs* have better opportunities to take the payloads and communication patterns into account, but the interoperability and generality of the resulting approach is limited. *Network or link layer PEPs*, in contrast, implement only basic functions (forwarding, routing, and medium access) and segmentation requires many changes as, for instance, the medium access must operate differently (cf. [Dom17]). In summary, we found that with respect to improving latency-awareness and predictability, the transport layer provides the broadest applicability and an appropriate set of network functions to be manipulated for improving performance.

7.4.2 General Relaying

In general, an end-to-end connection between two sockets can be split at an intermediate node—the *relay*. This process can (and should) be applied repeatedly wherever appropriate, but for the rest of this section, we consider a single split leading to two segments, as shown in Figure 7.6. Conceptually, segmenting a connection means transparently splitting it at an intermediate node by connecting its “ends” to the respective ends of the intermediate node. Thereby, both sides are segments that are to some extend independent of each other, and network functions work locally (cf. Section 7.2.3).

The “ends” are system-level sockets, which come with associated data buffers and connection metadata. The relay is incorporated into the communication via software-defined networking and can be deployed as a physical or virtual network function.

The aforementioned process describes the *split mode*, where the connection is fully terminated at intermediate sockets. An alternative to this is the *non-split mode* where the relay directly interacts with the packets in a manner that must not follow the rules

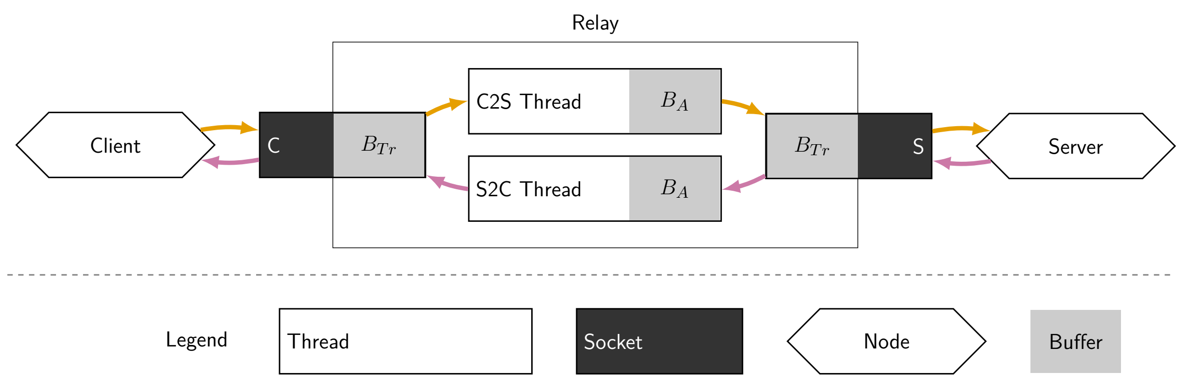


Figure 7.7: The relay forwards data between sockets facing the two original "ends" of the communication.

governed by the protocol in question (e.g. delaying or aggregating TCP packets without sending ACKs).

7.4.3 TCP Relaying

For TCP relaying, the socket file descriptors in the Linux kernel are used, which come with dedicated send and receive buffers—later we show how crucial their dimensions are for performance. Figure 7.7 depicts the architecture of such a relay. Both sides of a relayed communication (i.e. client and server) are implemented using a socket that connects to the original ends of the connection. The relaying itself works as follows: Per direction (i.e. client-server, server-client), a thread reads as much data as possible from the incoming socket, sending it to the outgoing socket as fast as possible. In-between, we have an *application-level* buffer B_A used to store incoming data before it can be sent to the outgoing socket.

This architecture leads to ACKs being sent upon reception at an intermediate node, "violating" the E2E reception guarantee that only data received at the other transport end is ACKed. This is in contrast to Split-TCP [KKFT02] where this is handled via distinct Local-ACKs. We must commit this violation to allow transport layer functions to work at the segment level, and to be transparent to the end-systems at the same time.

At first, this violation looks harmful, but a closer look at the practical relevance reveals that it is not: The semantics of TCP ACKs should not be mistaken for application layer acknowledgements that all data has been received and (will be) acted upon. As the relay makes crashes transparent (if either side is detected to be abruptly closed, it closes the opposite side abruptly), a host cannot distinguish a relay crash from a server crash when receiving a RST packet. At the same time, if a host is not sending data, it can also not detect if the peer is gone, independently of whether the peer is the original end-host or the relay. In consequence, we do not consider this to be a hard violation of the E2E principle.

7.4.4 RTP Relaying

Due to certain shortcomings of TCP for multimedia streaming and in particular real-time transmission, the RTP protocol has been designed [SCFJ03]. Though there are profiles that define ARQ and FEC for error control, our studies have yielded that they are rarely used in well-established open source projects (e.g. VLC media player[5] or Mozilla's WebRTC[6]).

Thereto, we have designed *application-independent* relays that add error control to a connection without it. These relays are placed after the first hop (at the first node) and before the last hop (at the last node before the receiver) [Bir17]. Thereby, the intermediate links can make use of RTP's error control profiles and we can add both FEC and ARQ—while being transparent to the end-systems. Additional relays can be added along the path, i.e. a specific error control configuration can be chosen for every pair of relays. Such a configuration defines whether retransmissions are enabled (ARQ) and how many data and redundancy packets are used per block (FEC).

7.4.5 PRRT Relaying

The PRRT protocol (cf. Chapter 4) is another candidate for applying relaying. The process itself can work identical to the TCP relaying, i.e. the relay uses sockets to terminate and re-open the connection in both directions. What is different from TCP is the number of parameters that are available to be set at the relay. For TCP, the major design parameters are the send and receive buffers. For PRRT, it is crucial to set parameters such as the *tolerable latency* and *error coding*. The tolerable latency must be *split* between segments as it still needs to resemble the actual application constraints. This split also has an impact on the choice of the error coding as it determines, for instance, the number of possible retransmissions. A detailed treatment of this dimensioning issue and the splitting of the latency budgets has been given by [Kar15].

7.4.6 SDN and NFV as Enablers for Relaying

In traditional packet-switched networks, core devices are simple and the complexity is in the end-hosts. With the increasing number of middleboxes over the last decades, this is no longer true and this shift in paradigm is also supported by the trends towards softwarized networking. This is also following the concepts of *fog computing* [BMZA12], where intelligence is moved inside the network. While circuit-switching led to significant resource reservation and therefore inefficiencies (i.e. small utilization), packet-switching faces inefficiencies due to the fact that each packet can potentially be treated differently. Therefore, we recently see a trend towards flow-switched networks that treat a single flow in a coherent manner. The concept of a flow is abstract by design, to allow for

[5]https://www.videolan.org/vlc/ (accessed August 29, 2019)

[6]https://developer.mozilla.org/en-US/docs/Web/API/WebRTC_API (accessed August 29, 2019)

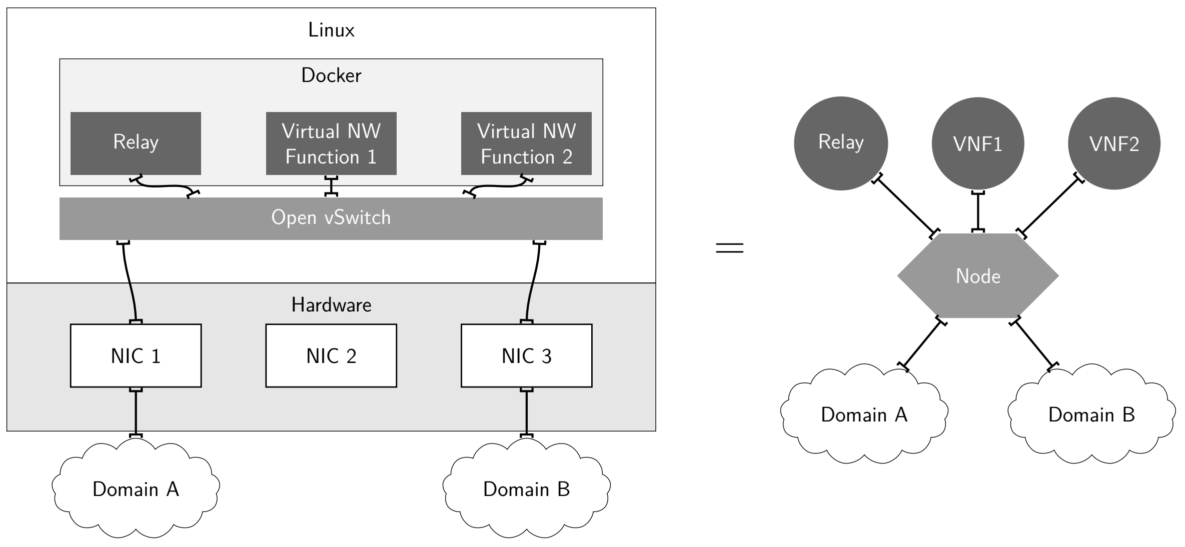


Figure 7.8: The softswitch architecture resembles a straightforward logical network view despite connecting several physical and virtual systems together.

different interpretations and mapping to other network parameters, e.g. TCP header information and IPs for a TCP connection.

In the context of TTS, softwarized networking (i.e. software-defined networking (SDN) and network function virtualization (NFV)) is suitable for several reasons: First, SDN provides flow-based routing, an abstraction that is particularly useful for relaying individual connections. Second, SDN implementations provide a broad view on the network topology and its link parameters—information that is essential when deciding on deployment of relays. Finally, relaying represents a network function that can be physically deployed, but will most likely be deployed as a *virtual network function* (VNF) as it is of small footprint and beneficial for many services.

Softswitches and Containers

In SDNs, it is common to use so-called softswitches, where a single physical device is turned into a network node with virtual end-systems directly attached to it (cf. Figure 7.8). By using general purpose operating systems, such as Linux, a large range of hardware platforms are supported and different functions can be implemented. This makes it possible to build compact solutions that only contain the software that is necessary for switching and to implement VNFs. Our evaluations are using Ubuntu 18.04[7], which ships with *Open vSwitch*[8] (OVS), offering support for extended switching capabilities and OpenFlow [MAB+08].

In addition, Docker[9] containers in Linux are used in a way that they look like phys-

[7]http://releases.ubuntu.com/18.04/ (accessed May 16, 2019)
[8]https://www.openvswitch.org/ (accessed May 16, 2019)
[9]https://www.docker.com/ (accessed May 16, 2019)

ical devices topology-wise. Docker gained traction in data center and cloud operations, as it is a lightweight alternative to *virtual machines* (VM). NFV was initially done using virtual machines, but emulating full systems made it inefficient. Furthermore, some functions require fast startup that could not be achieved with VMs, as they boot a complete OS. In contrast, Docker only virtualizes what is required by the specific applications, e.g. application frameworks or libraries, and reuses what is common between the applications, namely the kernel. This makes Docker a lightweight virtualization solution to keep the footprint, startup times as well as the communication latencies of VNFs small. Therefore, the naive relay we present in this section can be deployed as a Docker container on the node to enable a straightforward and lightweight usage of TTS.

Relaying Process using SDN

Incorporating relay components into network operations can be achieved using different approaches, e.g. using an SDN controller. First, a controller has to be aware of *relaying rules*, i.e. where and how to insert relays and reroute traffic so that TTS can be implemented. This includes match rules on host and port names to specify which services should be relayed. Our evaluations use a modified version of the SDN controller Floodlight[10], extended by RESTful interfaces that accept relaying rules. In the following, we assume that relays are placed *in prior* to a client establishing a connection, though other—more reactive—solutions are also possible. The choice of this proactive approach is to limit the latency penalty of TTS, which is already non-negligible due to the relay establishment process and the traffic redirection.

Connection Establishment When everything is set up, clients are able to establish a (segmented) connection to servers using a process as depicted in Figure 7.9. This process incurs additional latency during connection setup, but as we assume reactive SDN routing, this is a common approach. When going for low latency, it must be ensured that the flow rules are present *before* the client starts to connect to the server.

The first step in this process happens when the client sends a SYN-packet to the first switch. As the switch has no flow rule for it, a `PACKET_IN` event is raised at the controller, which looks up the relaying rule for this flow. If a rule is found, the controller programs the flow tables of the nodes along the path (installing R2 and R3 of Table 7.1) and configures the relay to act upon this new relayed connection.

In this process, the transport layer ports and buffer sizes are configured and the relay is instructed to connect to the server. While this connection is being opened, packets are already rewritten by the relay and appear to originate from the client. When the connection between the relay and the server is established, additional flows are programmed (R1 and R4) and the initial SYN packet is forwarded to the relay (and rewritten on the way). As soon as the relay answers, the client can complete the handshake.

[10]http://www.projectfloodlight.org/floodlight/ (accessed April 26, 2019)

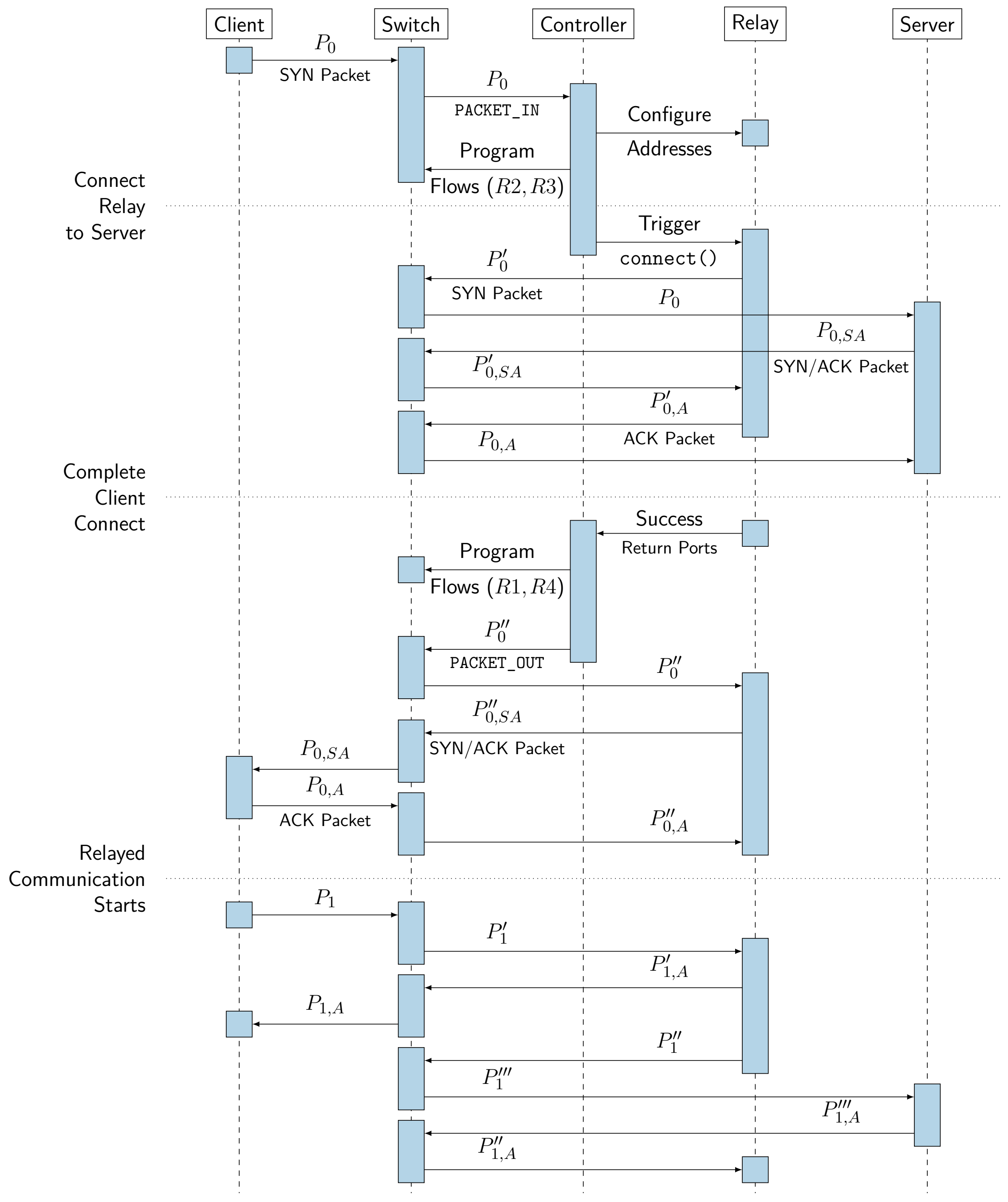


Figure 7.9: Upon a client connecting to a server, the controller establishes flow rules and configures the relay so that the connection gets relayed.

R	Src. IP	Src. TCP Port	Dst. IP	Dst. TCP Port	Action
1	IP_C	$TPort_C$	IP_S	$TPort_S$	$SetDstIP(IP_R)$ $SetDstTPort(TPort_{R,I})$ $Output(PhyPort_R)$
2	IP_R	$TPort_{R,O}$	IP_S	$TPort_S$	$SetSrcIP(IP_C)$ $SetSrcTPort(TPort_C)$ $Output(PhyPort_S)$
3	IP_S	$TPort_S$	IP_C	$TPort_C$	$SetDstIP(IP_R)$ $SetDstTPort(TPort_{R,O})$ $Output(PhyPort_R)$
4	IP_R	$TPort_{R,I}$	IP_C	$TPort_C$	$SetSrcIP(IP_S)$ $SetSrcTPort(Port_S)$ $Output(PhyPort_C)$

Table 7.1: Flow entries that are installed at the node in Figure 7.10 to rewrite and redirect the packets to allow TTS.

Eventually, both "ends" of the original connection have transparently established connections to the ends of the relay—but believe that they are connected to each other directly (the last part of the diagram shows no ACKs for clarity reasons). Finally, when either side closes the connection, this is propagated through the relay, allowing a proper tear-down of the connection (not shown in Figure 7.9).

Rewriting After the flow rules are established, the packets are transmitted with normal rate, except the additional latency that occurs for the round-trip from the node to the relay. As this happens using direct connections from OVS to the Docker container, this is a relatively small overhead. In order to keep up the *transparency* and integrate the relay into the connection, the network node must rewrite packets[11]. In Figure 7.10, we see the rewriting of IP and transport layer addresses a single packet faces when traversing the TTS path. The node redirects the packet to the relay and redirects the answer towards the server. It should be noted that the packets on the connection from relay to node look as if the relay would be communicating directly with the other ends of the connection. Table 7.1 shows the flow entries that are programmed into the node using OpenFlow.

7.4.7 TCP Relay Implementations

While investigating the effects of TTS, two distinct implementations of TCP relaying have evolved that differ in performance, but also in requirements towards the system they are deployed to.

[11]This is another reason why SDN is a straightforward choice for TTS—even though rewriting is also possible without it.

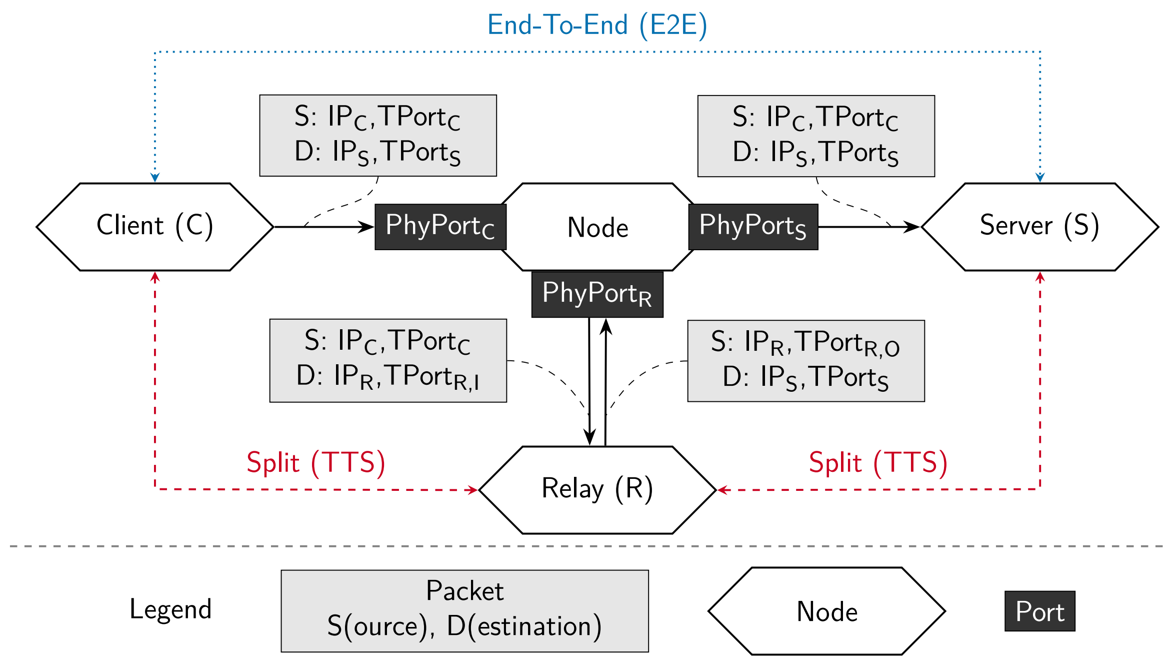

End-To-End (E2E)
S: IP$_C$,TPort$_C$ D: IP$_S$,TPort$_S$
S: IP$_C$,TPort$_C$ D: IP$_S$,TPort$_S$
Client (C)
PhyPort$_C$
Node
PhyPort$_S$
Server (S)
PhyPort$_R$
S: IP$_C$,TPort$_C$ D: IP$_R$,TPort$_{R,I}$
S: IP$_R$,TPort$_{R,O}$ D: IP$_S$,TPort$_S$
Split (TTS)
Relay (R)
Split (TTS)
Legend
Packet S(ource), D(estination)
Node
Port

Figure 7.10: Packets from client to server (and vice versa) are rewritten at the node to traverse the relay and thereby allow the application of TTS.

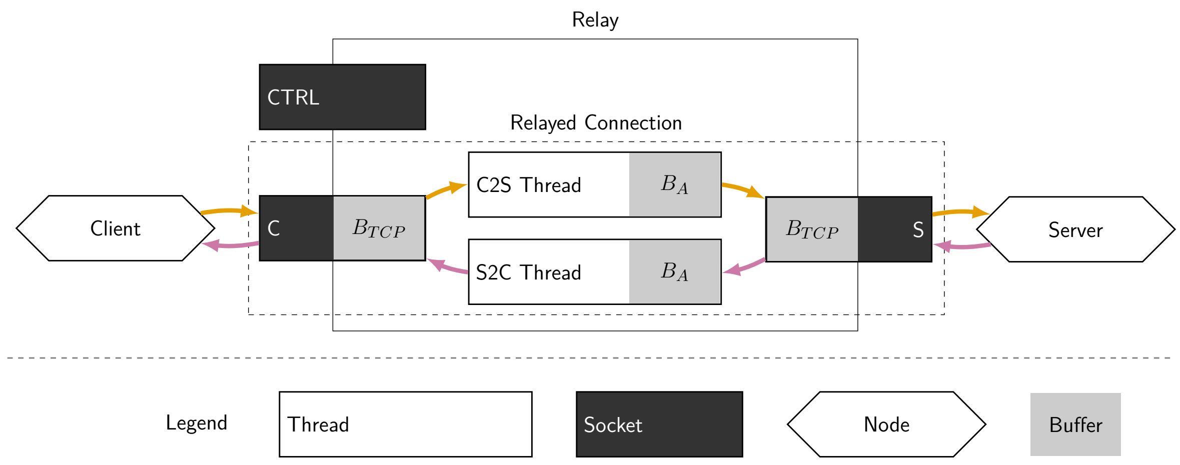


Figure 7.11: The relay manages multiple relayed connections, which are configured using the CTRL socket that is connected to the SDN controller.

Naive TCP Relay

The naive TCP relay implementation follows the design in Figure 7.11 and is a pure software solution. The implementation uses the *glibc*, interacts with the POSIX TCP sockets, and uses *pthreads* for multi-threading. The latter ensures proper operation as most calls on sockets are blocking. Furthermore, the relay's performance can be enhanced in high throughput scenarios, because both directions can be served simultaneously by two

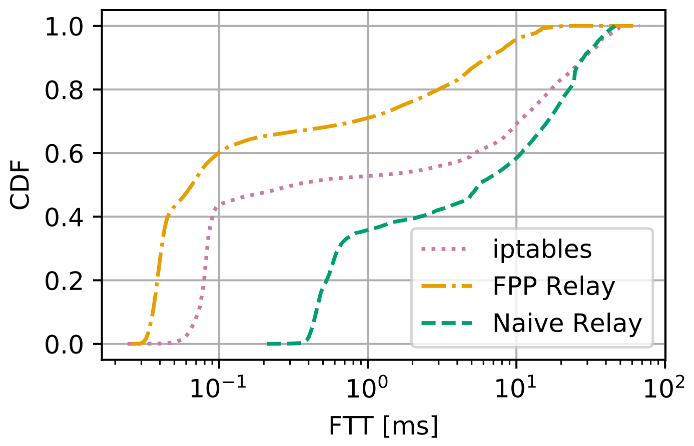


Figure 7.12: Comparing CDFs of forward-trip latency of the FPP-based relay, the naive relay without FPP and lower-layer iptables forwarding [Vog19].

threads running at the same time on separate cores. This makes it flexible in terms of deployment, e.g. as a Docker container within a softswitch as described in Section 7.4.6.

The relay's architecture in Figure 7.11 depicts its interfaces as well as intermediate buffers. The relay has a socket on either side of a relayed connection, which is named after the host it *points to*. An additional control socket serves as an interface to the SDN controller to install relaying rules. During operation, the relay tracks open ports and reuses ports of closed connections.

The relay uses multiple threads (potentially on multiple cores) to separate its functionality. The main thread waits for incoming controller messages to update the internal list of relaying rules. For each relayed connection, there are two threads—one for each direction of the TCP connection. This architecture leads to six additional buffers for a relayed connection, i.e. one TCP send buffer, one TCP receive buffer, and an application-layer buffer per direction. In order to avoid bufferbloat and provide low latency, these buffer sizes are important configuration parameters.

A major drawback of the naive relay is that an intermediate application layer buffer is needed to forward the data between two sockets, which incurs additional copy instructions. Furthermore, plain POSIX sockets are used, which do not provide the most efficient API to use TCP. For this reason, we developed the FPP relay that is covered in the next section. The naive implementation has been used for several publications [SH16b, SH17b] and is available as open source[12].

Kernel-bypass & Fast Packet Processing TCP Relay

After the development on the naive relay, it was investigated what the overhead per relay is (in terms of transmission and processing) and how to reduce it. As part of [Vog19], a

[12]https://git.nt.uni-saarland.de/ON/Applications/Relay (accessed May 15, 2019)

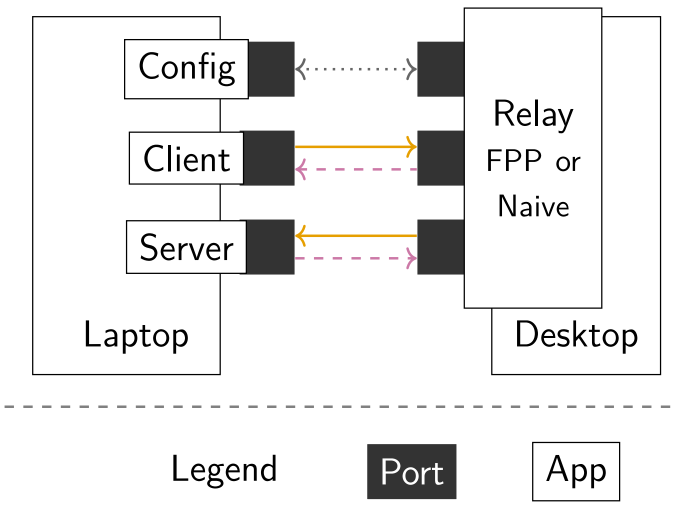


Figure 7.13: Comparing the FPP and the naive relay is done on a two-host setup with appropriate ports to allow measuring the forward-trip time [Vog19]

relay based on kernel-bypass networking and fast-packet-processing (FPP) technologies has been developed[13]. The FPP relay uses the same approach for configuring relayed connections as the naive relay, so it is a direct replacement. Internally, the solution uses mTCP[14] [JWJ$^+$14] which in turn leverages DPDK. As DPDK uses direct access to cores and NICs, the relay cannot be easily set up as a Docker container but is started on the system itself—limiting its range of possible deployments.

We compare the performance of the FPP relay to the naive relay using a setup as in Figure 7.13. In Figure 7.12 we see the CDFs of forward-trip latencies over a series of 1 million short packets sent from the sending application to the receiver. This shows how the distribution of latencies can be significantly improved (i.e. lower and steeper CDF) when using FPP. The iptables series acts as a reference, showing kernel-based forwarding, which is slower than the FPP version that also includes TCP forwarding.

7.5 Evaluation

Now that we know how to build relays and how to place them while maintaining transparency, the question remains whether our intuitive ideas about the benefits of TTS can be proven empirically.

[13]https://git.nt.uni-saarland.de/ON/Applications/fpprelay (accessed May 29, 2019)
[14]https://github.com/mtcp-stack/mtcp (accessed May 15, 2019)

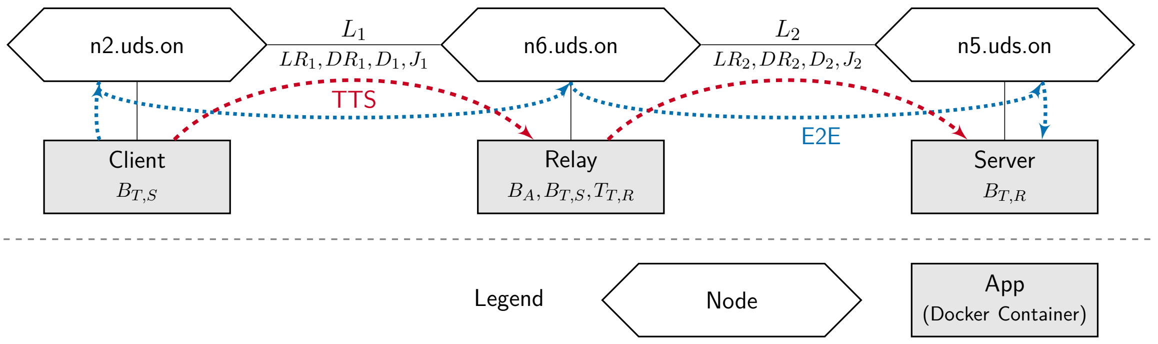


Figure 7.14: The evaluation is executed using three connected OpenvSwitch nodes on which Docker containers running the relay and the measurement app are deployed.

7.5.1 Methodology

Evaluation Metric and Measurement Applications

Even though TTS can have positive effects on many network parameters, our major figure of merit in the following is the flow completion time[15] $FCT, [FCT] = \mathrm{s}$ for a flow of size $S, [S] = \mathrm{B}$. Thereto, we can shed light on how well the TTS paradigm is able to narrow the latency distribution of TCP-based communication and how it can reduce the latency on an absolute scale. Thereby, latency-awareness is increased and the predictability of latency can be improved at the same time. For the measurement, we are using a simple TCP application, written in Go[16]. We specify the size of the flow and the application measures how long it needs to send all the data and finally get an application-layer acknowledgement by the peer. The optimal FCT is based on link rate $R, [R] = \mathrm{B/s}$ and is computed as:

$$FCT_{opt} = RTT + \frac{S}{R} \tag{7.15}$$

For simplicity, we omit the processing latency here, as it is in the order of µs, while the others sizes are in ms. The analysis uses cumulative density functions showing the FCT values over a set of identical experiments with flow size S.

Network Topology and Systems

The experiments use a topology as depicted in Figure 7.14, with nodes that have 8 cores and 8GB of RAM (except for an Internet evaluation, where the residential node has 4 cores and 2GB RAM) and all run Ubuntu 18.04[17] with the Linux 4.15 kernel. The application as well as the relay run as Docker containers on the nodes. For this reason, the naive relay is used throughout the evaluation, as the FPP relay cannot run inside a

[15]Consider [DM06] for further motivation on the usage of the FCT metric.
[16]https://golang.org/ (accessed May 17, 2019)
[17]http://releases.ubuntu.com/18.04/ (accessed June 5, 2019)

Docker container. Network link parameters are set using the netem[18] tool available in Linux. Finally, the default congestion control algorithm used is CUBIC [HRX08] unless stated otherwise.

7.5.2 Investigations

The evaluation is going to investigate the following hypotheses on TTS:

H1: TTS is beneficial for lossy last miles. For video streaming scenarios, where the receiver is e.g. in a home network with a wireless link, TTS is able to compensate for highly biased loss and latency parameters, as motivated by Figure 7.1.

H2: TTS is beneficial in flow-control-limited scenarios. In IoT scenarios, the speed of downloading software updates might be constrained by the small end-host buffers and not by the wireless link they are using to connect. We expect that in such scenarios, TTS is able to reduce the FCT thanks to an intermediate relay with a sufficiently large buffer (cf. Section 7.3.2).

H3: TTS is beneficial in scenarios with high jitter. Thoughts around the impact of TTS on the ordered delivery network function might lead to the conclusion that the aggregation in TTS is harmful for the FCT. In contrast, high jitter leads to a larger number of retransmits—a scenario that is detrimental for achieving low FCTs. We are investigating which aspect dominates and under which circumstances TTS outperforms E2E.

H4: TTS has different impacts on different congestion control algorithms. TCP performance is significantly determined by the congestion control algorithm used, mainly because it controls the amount of data in-flight and defines the loss handling. Therefore, we expect different behaviours and turn-over points where E2E is better than TTS and vice-versa.

Validating H1

In order to validate the suitability of TTS for video streaming scenarios with a lossy last mile, we look at the FCTs for a flow of 5MB. The first link—the residential Internet access—has a one-way delay of 20 ms and the home WLAN has a one-way delay of 1 ms. Both links have a maximum data rate of 16 Mbit/s and the buffers are sized such that the complete flow can fit in. The residential link is error free, but the WLAN link has the following loss rates: 1.50 , 1.00 , 0.50 , 0.10 , 0.01 , and 0.001 %. We run 100 trials per loss rate and give the CDF of the FCTs in Figure 7.15.

As can be seen, TTS is able to compensate high loss rates and can achieve lower and more predictable latencies. For instance in the scenario with 1.00 % loss, the 95th percentile FCT is reduced from 12.84 s to 5.51 s and the inter-quartile range (75th to

[18]http://man7.org/linux/man-pages/man8/tc-netem.8.html (accessed May 17, 2019)

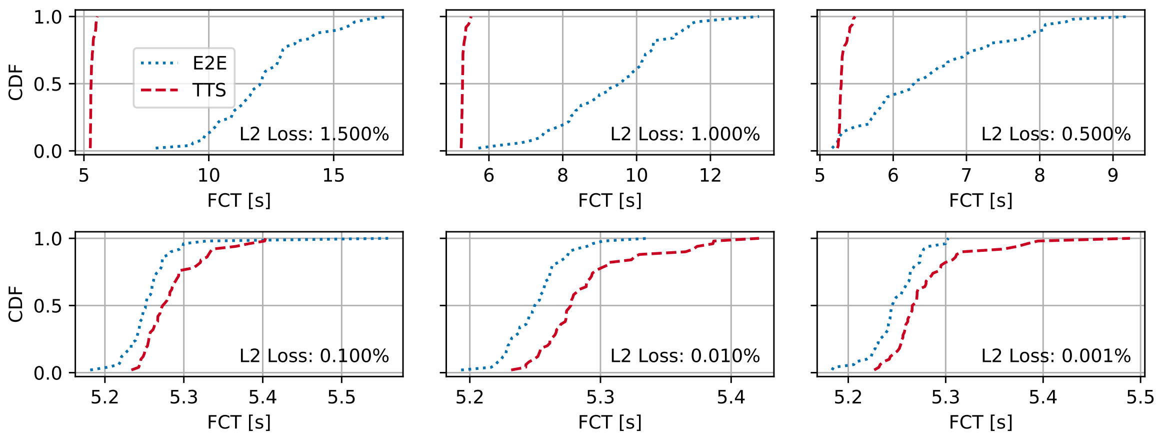


Figure 7.15: TTS can significantly reduce FCTs for scenarios with a lossy last mile and a skewed delay. The higher the loss, the greater the benefit of using TTS (note the varying x-axis).

25th) from 2.17 s to 0.05 s. For the congestion control algorithm used here (CUBIC), there is a turn-over point between 0.50 % and 0.01 %, where the overhead caused by TTS starts getting larger than the gain by local retransmissions. This validates our hypothesis, showing that TTS is beneficial for local losses and we can deduce that the gain through TTS is proportional to the loss rate.

Validating H2

In Section 7.3.2, it has been pointed out that small receiver buffers can—through flow control—be the bottleneck of a communication path. Again we assume a two-hop scenario and have effective maximum rates that are determined by $\frac{B_{recv}}{RTT}$ and not limited by the link transmission rate itself. TTS is able to increase the effective rates on the two links by (a) the relay buffer acting as a larger buffer for the first link and (b) reducing the RTT of the second link that is determining the effective rate on the link. To provide a clearly visible effect, we picked the one-way latencies to be 20 ms for the first and 5 ms for the second link—allowing up to a $\frac{20}{5} = 4$-fold increase in transmission rate in theory. The receiver buffer always has 4 kB, we are transmitting a payload of 1 MB, and set the relay buffer size to 4, 8, 16, and 32 kB.

Based on the theory in Section 7.3.2, the 16 kB buffer should be optimal, as it is four times the size of the receiver buffer. In Figure 7.16, we see that the distance of the FCT CDFs increases with the buffer size. Notably, an increase beyond the optimal buffer (e.g. the 32 kB) does not further improve the FCTs[19], which validates our model from Section 7.3.2.

[19]Apart from a tiny margin that is most likely due to the fact that theory and practice are—in practice—not the same.

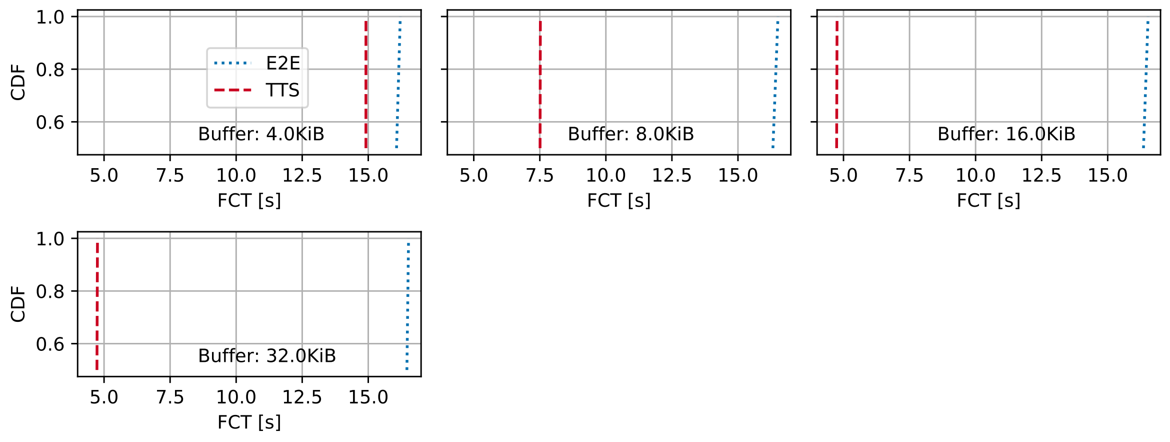


Figure 7.16: When receiver buffers are small and flow control is a limiting factor, TTS can install an additional buffer. The size of this buffer has a direct impact on the potential speed-up and the optimal buffer size can be determined which maximizes utilization (in this scenario it is 16 kB).

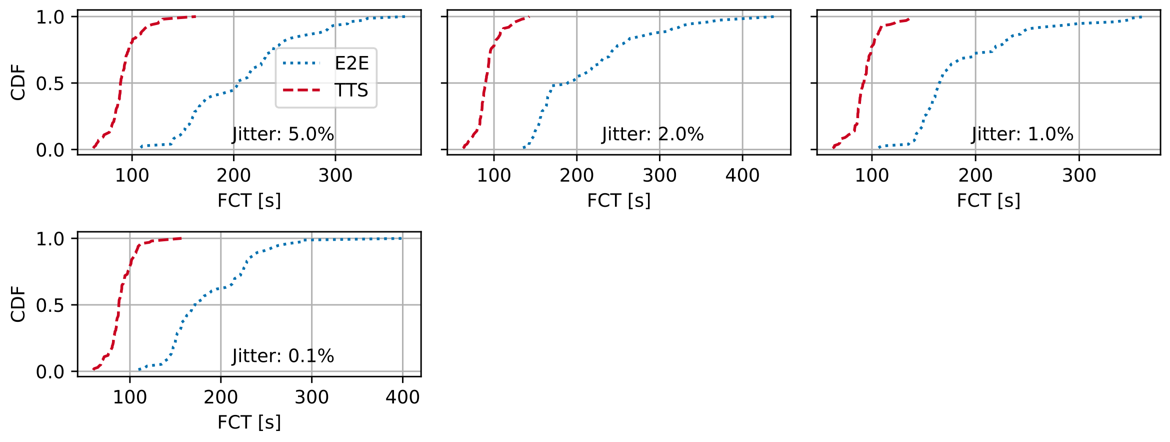


Figure 7.17: In scenarios with even small amounts of jitter, TTS can improve the flow-completion times.

Validating H3

As mentioned in Section 7.2.3, relays reorder packets leading to additional delays due to out-of-order packets being stored and only forwarded later. Therefore, the conclusion might be natural that TTS increases the latencies in scenarios with high jitter. We evaluate in the same two-hop scenario as before using two links with 16 Mbit/s rate and a constant delay of 20 ms for the first and 2 ms for the second link. The flow size in this scenario is 512 kB. In addition to this constant latency, we add jitter to both links at a magnitude that is relative to the constant latency.

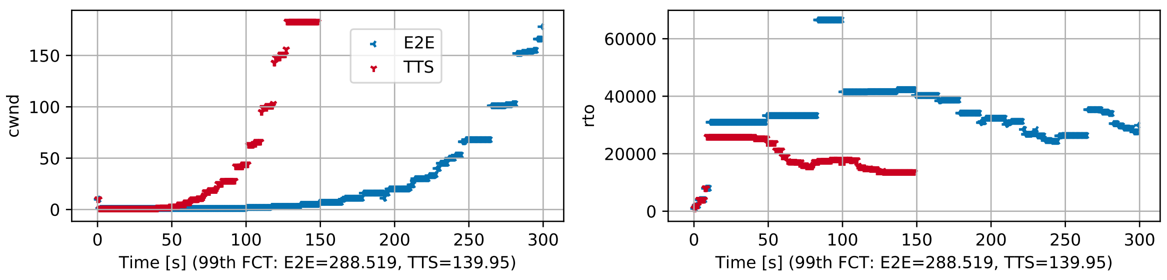


Figure 7.18: The improved FCTs are due to lower retransmission timeouts (`rto`) and higher `cwnd` due to shortened RTTs. Depicted are `ss` samples for two evaluations which caused the 99th percentile FCT when applying 0.10 % jitter for both links.

Figure 7.17 shows the resulting FCT CDFs for jitter between 5 % and 0.10 %. The curves show that jitter is in fact less harmful for TTS than for the E2E scenario—countering the initial assumption that reordering might be a reason for TTS performing worse. Apart from TTS achieving better tail latencies, it is also evident that the range of possible FCTs is smaller; which indicates that TTS increases the latency-predictability.

In addition, we have measured TCP runtime parameters using the Linux `ss` tool and sampled every 10 ms. Figure 7.18 shows `cwnd` and `rto` time series for the two evaluations that caused the 99th percentile FCT when applying 0.10 % jitter on both links. The diagrams shows clearly that with TTS the `cwnd` can achieve higher values and that the `rto` is kept smaller in comparison to the E2E case.

Validating H4

Recently, the area of congestion control has seen an increased interest with several new implementations being published. For the sake of this evaluation, we have focused on the algorithms available in the Linux kernel as this is sufficient to prove our point. We are aware that more recent algorithms exists (and regularly compete in the Pantheon [YMH+18]) and that they provide different characteristics. This evaluation aims to show how not only different CCAs achieve different performance, but also how TTS has a different impact on them. Thereto, BBR [CCG+16], CUBIC [HRX08], NewReno [HFGN12], Veno [FL03], and Westwood [MCG+01] are evaluated using a scenario similar to the first one with the lossy last mile, i.e. 16 Mbit/s data rate on both links, 20 ms and 1 ms one-way delay on the first and second link. To show the effects of loss, we have chosen to do two evaluations per CCA with the second link having 0.10 % and 0.50 % respectively, while the first link is error-free.

The results depicted in Figure 7.19 show differences in the performance of various CCAs depending on the loss rate. Due to TTS, the sender can keep a high `cwnd` when facing loss and retransmissions are reduced—depending on the CCA implementation. We decided not to evaluate cross-traffic scenarios and fairness metrics, in particular because achieving fairness is still work-in-progress for the most recent CCAs (e.g. [ZMRP19]).

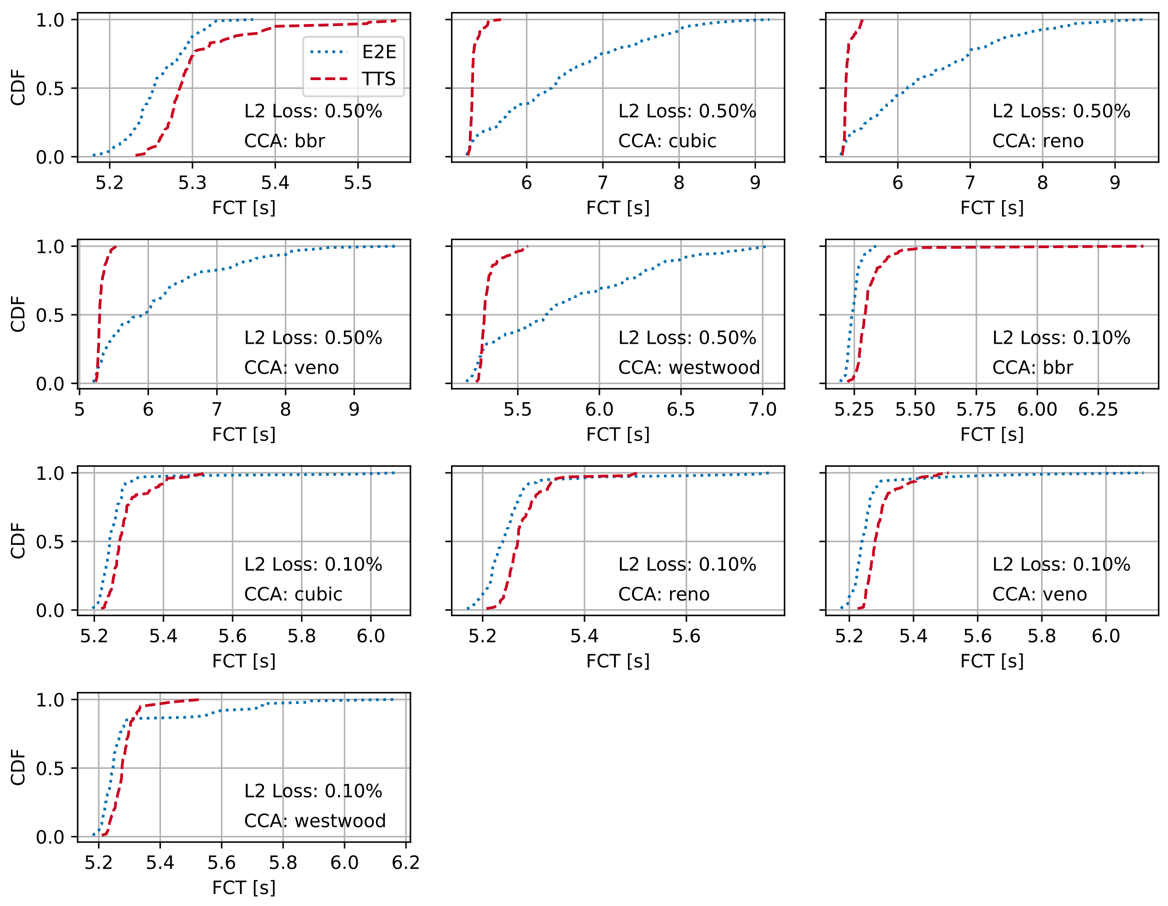


Figure 7.19: The gains achieved by TTS depend on the congestion control algorithm used, which has a direct impact on whether TTS operation is better than E2E and how narrow the distribution of latency can be (note the varying x-axis).

7.6 Related Work

We have presented a summary of theoretical segmentation models, together with practical implementations that use SDN approaches, leave the E2E semantics intact and achieve performance gains—effects that others have also discovered and achieved using different implementations that have limitations our approach does not have.

7.6.1 Origins of Transmission Segmentation and Splitting

As mentioned before, the splitting approach—which we call TTS—is not particularly new and has historically received interest in the context of wireless communications since the 1990s. The IETF defines this approach as *Performance Enhancing Proxies* (PEP), which have been first proposed in RFC 3135 [BKG$^+$01]. The first mentioning of splitting TCP connections is by Balakrishnan et al. [BSAK95], who present the *Snoop* approach.

Snoop detects duplicate ACKs at the proxy, interprets them as loss, and retransmits packets it stored locally. Bakre et al. [BB95] have developed *I-TCP*, an approach that splits TCP to cope with mobility and unreliability, allowing to, for instance, have different congestion control algorithms for the wired than for the wireless links. With *Mobile-TCP* (M-TCP), Brown et al. [BS97] show a solution that also splits the connection but can (in contrast to I-TCP) maintain E2E semantics and handle longer disconnects. One central feature of M-TCP is that if it senses congestion will happen soon, it instructs the sender to stop sending by setting the receive window to 0.

A solution that breaks the E2E semantics is *Split-TCP* by Kopparty et al. [KKFT02], which led to a thorough mathematical analysis by Baccelli et al. [BCF08]. Split-TCP modifies the TCP protocol to allow for an integration of TCP proxies within the network. Several challenges with special network scenarios, e.g. links with large bandwidth-delay product, have been tackled by this. Split-TCP poses deployability issues with commodity systems and lacks transparency towards end-systems as well as transit systems that are unaware of the segmentation. Pathak et al. [PWH10] propose to use Split-TCP for data-center to data-center communication to reduce service time for end-users, with the result that it can effectively reduce the time web searches take.

While the previous approaches are targeted at cellular networks, *Augmented Split-TCP* (AS-TCP) by Jung et al. [JCS$^+$06] deals with optimizations for 802.11 wireless LAN and presents an extension of I-TCP. In particular, AS-TCP leverages MAC-layer ACKs to create fake local ACKs to instruct the transport layer that data has successfully passed the first, wireless hop.

Our work on segmentation is grounded on [KGH10], which identified that segmentation is in principle beneficial, but the choice of segmentation points and their number is non-trivial—in particular when using approaches underlying PRRT. In [KH11], it is identified that clustering links into homogeneous groups can make segmentation more efficient, as relays are only put at the borders between these clusters.

7.6.2 Recent Investigations on Segmentation and Relaying

In the last decade, several approaches have been proposed which consider the general proxy/relay scheme, but have various benefits and drawbacks, or come with different deployment approaches.

Ladiwala et al. [LRW09] propose an approach similar to ours, but change the router implementation, which is against our *minimal effort* requirement defined in the beginning—not to mention that it makes wide deployment extremely unlikely.

Ren et al. [RL10] consider shortcomings of TCP in cellular networks, an issue that is also tackled by the publications mentioned later. In 3G networks, bandwidth oscillations occur frequently and cause TCP to back off due to supposed congestion. The authors model the TCP performance under these circumstances and propose a window-adaptation proxy using a *robust sliding mode variable structure control law* [Itk76] to improve the performance. This approach is used to control the sending window at the proxy, which allows to resist bandwidth oscillations and improve utilization on cellular links. The proxy is located at the wireless base station, hence the measures are applied to the last link, which is also the major contributor to performance degradation if no care is taken. Their proxy is therefore a cross-layer solution and the window-adaptation mechanisms could be added to our approach in case the relay is deployed at a cellular base station.

Liu et al. [LL15] propose a *mobile accelerator*, specifically aiming to improve TCP throughput and utilization over wireless links. This approach, similar to ours, comes without any modifications or reconfigurations to the end-systems and only changes the behaviour of TCP at the accelerator relay. The authors propose an *opportunistic transmission scheme* for the accelerator-to-sink-segment that partly disables flow-control. This scheme is able to cope with the receive window being a limiting factor despite the application being able to read fast enough—which is often the case with mobile phones in recent cellular network technologies. In practice, this means that this scheme risks packets being discarded as out-of-the-window, but often succeeds in sending more packets per round-trip than the receiver window would allow.

With respect to rate control, the scheme ignores loss events to determine the sending rate and does not use the congestion window. The accelerator leverages cross-layer design by using the known available rate at the cellular base station to set the initial rate in the startup phase—achieving high utilization right from the start. Afterwards, the algorithm enters the *adaptive phase*, where the current delivery rate is measured and averaged over a sliding window. Their goal is to occupy the link buffer enough so that it never runs empty (wasting utilization), but also to keep the buffer fill level low (to avoid losses).

By modifying the behaviour of intermediate components, it is necessary to reimplement parts of TCP, something our solution does not require. Finally, their approach does not leverage SDN technology, which makes it harder to deploy and integrate.

Haq et al. [HD15] describe a relay that focusses on reliable wide-area communication. They use the cloud in a very distinct way by deploying their relays at various data-centers and use inter-datacenter links for their communication. Depending on the vendor, these connections do not cross multiple ASes and are different from the public Internet. The reliable transport they build uses FEC (to make a single stream reliable) as well as network coding (to let multiple streams make each other reliable). Our solution is more general, but could in fact benefit from a cloud-based deployment and inter-DC links. Another cloud-based approach is taken by Siracusano et al. [SBK$^+$16], who use a a Xen-based lightweight implementation of a TCP proxy. Their major contribution is achieving short boot-times that allow to enable relaying *on-the-fly*.

Dombrowski [Dom17] proposes a relaying scheme for the physical and link layer in order to achieve predictably low latency and high resilience for wireless communication, e.g. in industrial applications. The relays he proposes can implement different *cooperation methods*, a concept that is similar to the different network functions that are affected by our transport layer TTS approach. These methods are (a) *amplify and forward*, which works purely on the analog level, (b) *decode and forward* that removes noise and does error decoding before transmitting again, or (c) *coded cooperation/compress and forward*, where more complex functions are applied to the signal, e.g. to achieve reliability by path diversity. Though this approach operates on the physical layer, a combination of this approach with TTS can improve matters as transport layers govern—to some extent—how many frames are sent on the link layer.

Polese et al. [PMZ$^+$17] Wireless standards such as mmWave (e.g. in *New Radio* [DPS18]) pose problems with respect to TCP, because TCP cannot e.g. anticipate link changes and has "slow reactiveness", which is addressed by *milliProxy*[20]. The main goal of milliProxy is to use cross-layer information (e.g. buffer occupancy, estimated PHY layer data rate) to control the flow and congestion windows as well as the MSS. This information comes from the next generation node base (gNB). Similar to our approach, it does not require remote-side modifications and achieves full end-host transparency.

Kim et al. [KKK17] highlight the trend towards mmWave communication and describe why the use of TCP is problematic for links that employ this technology. This is was also investigated by Zhang et al. [ZPM$^+$19], who consider different congestion control algorithms, buffer sizes, and AQM methods in mmWave scenarios. In particular, mmWave links face high variance in terms of RTT and throughput due to the significant difference between line-of-sight (LOS) and non-line-of-sight (NLOS) paths. Kim et al. propose the mmPEP to overcome this weakness and harness the benefits of mmWave, namely the vast amount of capacity a LOS link provides.

These PEPs are deployed at base stations and implement two functions: First, the proxy manages ACKs, by sending own ACKs to the origin of the transmission (e.g. the cloud server) to advance its window and allow filling the base station buffer, despite the downlink to the mobile node being temporarily in NLOS state. As soon as the path becomes LOS again, the high capacity allows to quickly increase the transmission rate and consume the buffer at the relay. Second, the proxy is able to do batch retransmissions, i.e. it does not only send the lost packet and waits for recovery, but also retransmits subsequent packets at the same time, assuming that a batch of packets got lost.

While the ACK management is the same as in our approach, the batch retransmissions are not possible with our implementation, as we keep the TCP implementation unchanged and hence rely on the loss recovery strategy of the congestion control used for the cellular link. Nevertheless, TTS could leverage any approach as long as it is implemented as a congestion control algorithm that can be used for the downlink.

[20]Despite the names, there is no ancestor- or sibling-relationship between miniProxy and milliProxy.

Pennekamp et al. [PGH$^+$19] propose that in-network processing is an essential enabler for the *Internet-of-Production*. In this context, our approach could be considered as an additional means to not just have processing for control tasks inside the network elements [RGW$^+$18], but also adapt the transport layer to the specific needs of the application that consumes in-network-processing services.

7.7 Conclusion

The demands of CPS towards latency-awareness and -predictability require more than only changes at the end-hosts, as communication flows in CPS will traverse paths with various heterogeneous link parameters. Typically, these heterogeneities are invisible to the transport layer, which operates in E2E mode. The TTS approach presented in this chapter addresses the needs of CPS in providing predictable latencies by making the transport layer aware of individual links along an E2E path. This is done by segmenting the connection into small groups of links and putting relay components at their ends to terminate and reopen the connection. The formalization of this process provided in this chapter allowed to chose relay configurations, e.g. buffer sizes, that are appropriate for certain scenarios. These positive effects on network performance have been showed using a complete SDN/NFV implementation that makes the TTS approach deployable. In consequence, the requirements (Section 7.2.1) can be fulfilled and enable CPS to be aware of and benefit from predictable latencies.

Thus we may have knowledge of the past but cannot control it; we may control the future but have no knowledge of it.

Claude Elwood Shannon

Chapter 8

Conclusion

As of today, it is foreseeable that merging digital computation and communication with physical processes is going to have significant impact on many application domains—allowing to achieve higher efficiency, but also to enable new ways of operation, e.g. in medical services. Hence, it is imperative that the domains of engineering and computer science investigate cyber-physical systems in more detail. Especially the cooperation of computation, communication, and control has to attract further attention. Central to this is the treatment of time and latencies, as control of the physical world can only be achieved with predictable timing in the cyber world.

8.1 Summary

While there are many ways to achieve predictable timing and certainly many sub-systems to be addressed, the major focus of this thesis is on *latency-awareness and -predictability* at the transport layer of distributed CPS. The design, implementation, and evaluations of bringing this feature to protocols for CPS has led to three essential conclusions:

- In order to achieve latency-awareness and predictably low latency in cyber-physical systems, the latency must be holistically analysed throughout and across the various sub-systems in a cross-layer fashion.
- By knowing about the different latency distributions at runtime, it is possible to identify and react to the bottleneck of the CPS and make all non-bottleneck components adapt to it—saving latency and preventing waste of resources.
- Distributed CPS integrate time-critical systems into large network infrastructures that consist of highly heterogeneous links—so that segmentation and decoupling of transport layer network functions is essential to achieve predictably low latency.

This analysis has yielded several software artefacts:

X-Lap is a toolchain that allows for minimally invasive, cross-layer analysis of transport stacks for cyber-physical systems. By using it, system designers can validate timing, discover correlations of processing steps with the overall end-to-end latency, and compare code versions as well as system configurations with respect to induced latencies. The source code is openly available[1].

X-Pace holistically measures and synchronizes the paces of processing and communication steps in CPS applications. Thereby, the age-of-information can be reduced and made more predictable, which caters to control and live-multimedia applications. The evaluations of X-Pace[2] as well as its implementation in PRRT[3] are openly available.

TTS Relays allow transport layer functions to work on segments instead of the whole end-to-end path. Thereby, latencies can be reduced and made more predictable, e.g. through localizing loss-recovery or congestion control. The evaluation code for TTS[4] and the relay implementation[5] are openly available.

8.2 Outlook

While this thesis provides several contributions in order to bring latency-awareness and latency-predictability to communication in cyber-physical systems, there are further directions that must be investigated to enable reliable, highly distributed CPS.

Energy-aware Transport So far, this thesis has primarily focused on providing transport layer implementations that are latency-aware and latency-predictable (cf. Chapter 3), while also considering the relation to reliability (cf. Chapter 4). When we consider CPS that are mobile (i.e. they use intermittent or highly constrained energy sources) and if we take the goal for decarbonization of digital technology seriously, sustainable CPS should also incorporate and optimize for energy aspects. Again, a first step is to establish the *awareness*, i.e. correlating the different operations and decisions a CPS can make with appropriate estimations of energy demand. In the next step, system designers can leverage this information to make a CPS adhere to certain restrictions on energy demand and allow a CPS to adapt its function to this—while keeping the latency and resilience demands of the application in mind. In this spirit, one can imagine the network triad depicted in Section 3.1 to be extended by energy as an additional domain. The impossibility of joint maximization of the different dimensions persists and

[1]http://xlap.larn.systems (accessed September 2, 2019)
[2]http://xpace.larn.systems (accessed September 2, 2019)
[3]http://prrt.larn.systems (accessed September 2, 2019)
[4]http://tts.larn.systems (accessed September 2, 2019)
[5]https://git.nt.uni-saarland.de/ON/Applications/Relay (accessed September 2, 2019)

while energy adds another dimension and increases the design space, it puts significant constraints on the practically achievable solutions.

Statistically Shapeable Transport Using cross-layer pacing and the X-Pace implementation (Chapter 6), the latency of processing and communication chains can be controlled to match the bottleneck pace. When applying this to several, parallel flows (of several concurrent control applications) that share a common bottleneck, a perfect synchronization can lead to detrimental effects. A good example for this from network control is [FJ94], which talks about the detrimental effects of synchronized bursts of routing traffic and their origins.

Therefore, control engineers are demanding communication services with latencies that can be shaped to fit their needs, i.e. to match a certain distribution with specific parameters, e.g. a normal distribution with a certain mean and standard deviation. Based on the capabilities of X-Pace to control the latency deviation to be close to 0, advanced techniques must be developed that transform this into the distributions that controllers wish for. This statistical shaping is not only relevant for the dimension of latency, but is also essential with respect to reliability: A certain control design can be considered to be stable given a certain loss distribution over sensor and actor messages. For that reason, PRRT and its error control component can be extended to provide these loss distributions over channels that expose less reliable behaviour.

Multicast-enabled Transport and Segmentation Distributed cyber-physical systems often feature multiple controllers and plants that are simultaneously interested in receiving sensor or actor messages. Therefore, multicast-enabled transport protocols can provide a means to achieve higher efficiency in contrast to multiple concurrent unicast connections. With respect to the design of PRRT, it has to be investigated how its advanced features (e.g. latency-measurement and pacing) can be *multicast-enabled*. This is particularly challenging, as multicast scenarios tend to expose feedback implosions that hinder scaling. Typically, this is solved by suppressing feedback [NB99] and using negative acknowledgements [PTK94] to be sent only when data has not been received.

Implementing these features reduces the feedback that the senders uses to infer network and system parameters, so it is likely to degrade its estimation and adaptation capabilities. It is therefore advised to carefully investigate how negative ACK feedback schemes can be implemented to sustain an adequate level of estimation quality. In combination with the transmission segmentation presented in Chapter 7, a plethora of challenges arises that must be investigated.

Heuristics for Transparent Transmission Segmentation The TTS approach presented in Chapter 7 highlights how such an approach can be implemented in a manner that is transparent to the end-hosts and uses in-network components with low overhead. While this chapter clearly states theoretical and practical benefits of using such an approach, the question *whether to relay or not* is highly dependent on the individual scenario and requires a thorough understanding of the interconnections between *channel parameters*, *network functions*, and the configuration of the *segmentation*. Therefore, it is essential that heuristics are developed that allow to dimension a segmentation in a *spatial* and *quantitative* way. This is non-trivial as (a) channel parameters change, (b) concurrent flows compete, and (c) models of network functions usually lack fidelity. Under these circumstances, it is worth investigating whether *data-driven, machine-learning-based approaches* to decide on number and placement of relays can be beneficial for network operations. One could imagine that an agent is trained using reinforcement learning [SB$^+$98] to achieve better network performance by deploying relays in the network and conditionally enabling them for certain connections.

Applied Machine Learning for Real-time Self-adapting Transport Stacks Finally, using *energy-awareness*, *statistical application requirements*, and *multicast-enabled transport* the domains to be considered and the adaptation-decisions to be made become intractable for a human designer. Therefore, it must be investigated how this complex real-time optimization problem can be formulated and how machine learning models can be trained that are able to balance all these aspects to achieve good performance.

Appendix A

Gilbert-Elliot Models for Correlation of Erasures

In order to model loss in packet-based networks, Gilbert-Elliot (GE) models [Gil60, Ell63] are an adequate means [YMKT99, TG04, YMT05, HH08, BLZ09]. In the following, we give a definition of a general GE, simplify them to SGEs, and provide formulas to merge two SGEs into a single SGE that behaves as the cascade of the two SGEs.

A.1 Gilbert-Elliot Models

In general, a 2-state Markov model as depicted in Figure A.1 is considered as a Gilbert-Elliot (GE) model. We follow the notation of a good (G) and a bad (B) state, which generate errors with a rate of δ_G in good and δ_B in bad. The transition matrix is:

$$A = \begin{pmatrix} \beta & 1-\beta \\ 1-\alpha & \alpha \end{pmatrix} \tag{A.1}$$

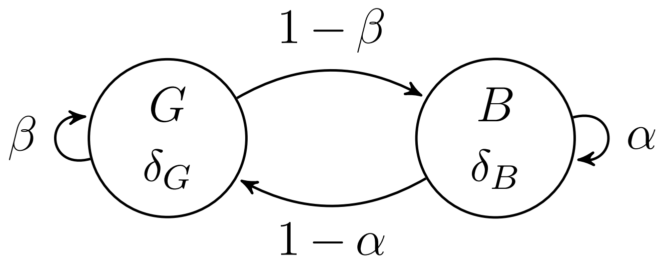


Figure A.1: General Gilbert-Elliot Model as described in [HH08].

[Gor12] analyses sequences to fit a simplified GE (SGE) with the following constraints:

$$\delta_G = 0, \delta_B = 1 \tag{A.2}$$

Thereby, it leaves the parameters α and β free. We arrive at:

$$\mu = p_G \cdot 0 + p_B \cdot 1 = p_B \in [0,1] \tag{A.3}$$

$$\rho = \alpha + \beta - 1 \tag{A.4}$$

The probabilities to be in the states are:

$$p_G = \frac{1-\alpha}{2-\alpha-\beta} \tag{A.5}$$

$$p_B = \frac{1-\beta}{2-\alpha-\beta} \tag{A.6}$$

The error probability becomes:

$$E\{X\} = \mu = p_G \cdot \delta_G + p_B \cdot \delta_B \overset{SGE}{=} p_B \tag{A.7}$$

A.2 Multi-Hop Simplified GEs

In order to calculate the correlation in multi-hop scenarios (cf. Chapter 7), we first look at how a two-link scenario can be mapped to a four-state simplified GE:

As erasures happen whenever a packet is lost on either hop, the states B_1G_2, G_1B_2, and B_1B_2 (all of which have $\delta = 1$) can be merged into a single *bad* state. The resulting SGE_{E2E} has the following parameters:

$$\beta_{E2E} = \beta_1 \cdot \beta_2 \tag{A.8}$$

$$\alpha_{E2E} = 1 - [\beta_1 \cdot (1-\alpha_2) + (1-\alpha_1) \cdot \beta_2 + (1-\alpha_1) \cdot (1-\alpha_2)] \tag{A.9}$$

Note that we derived α_{E2E} by summing the transitions in Figure A.2 which lead from *bad* states to the *good* state, i.e. which contribute to $1-\alpha_{E2E}$. The correlation of the composed SGE is again $\rho = \alpha_{E2E} + \beta_{E2E} - 1$.

If we want to compute α and β across more than two hops (e.g. n), a recursive approach is suitable:

$$SGE(\alpha_{E2E}, \beta_{E2E}) = CombinedSGE(SGE(\alpha_{1,\dots,n-1}, \beta_{1,\dots,n-1}), SGE(\alpha_n, \beta_n)) \tag{A.10}$$

Starting with the first two SGEs that correlate to their links, we combine them into a single one and continue by merging it with the SGE that correlates with the next link.

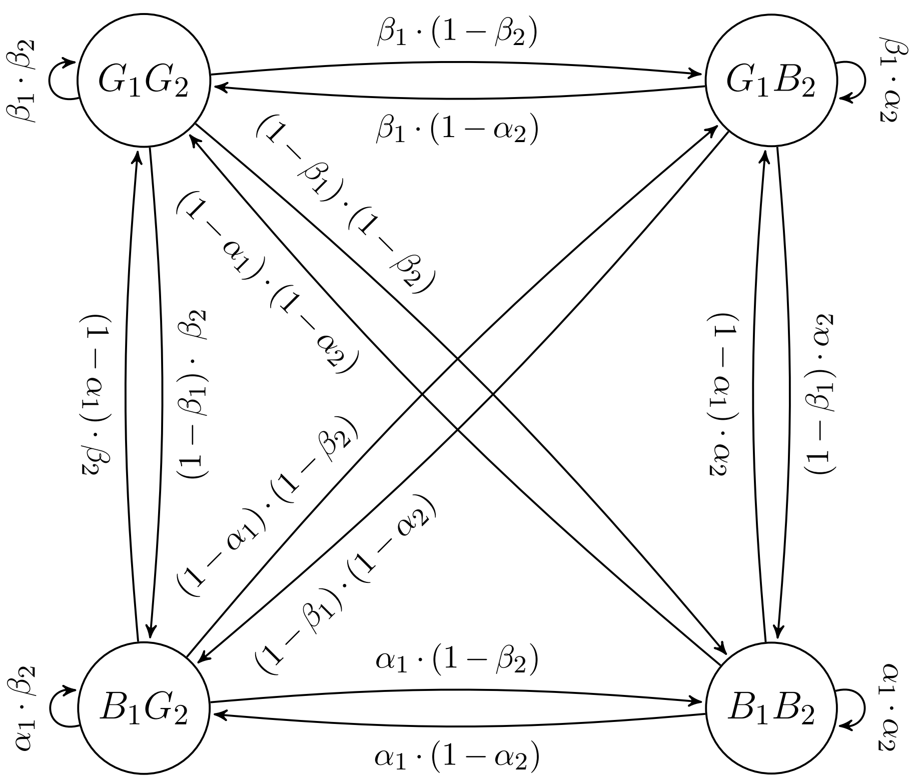

G_1G_2
G_1B_2
B_1G_2
B_1B_2
$\beta_1 \cdot \beta_2$
$\beta_1 \cdot (1-\beta_2)$
$\beta_1 \cdot (1-\alpha_2)$
$\beta_1 \cdot \alpha_2$
$(1-\beta_1) \cdot (1-\beta_2)$
$(1-\alpha_1) \cdot (1-\alpha_2)$
$(1-\alpha_1) \cdot \beta_2$
$(1-\beta_1) \cdot \beta_2$
$(1-\alpha_1) \cdot \alpha_2$
$(1-\beta_1) \cdot \alpha_2$
$(1-\alpha_1) \cdot (1-\beta_2)$
$(1-\beta_1) \cdot (1-\alpha_2)$
$\alpha_1 \cdot \beta_2$
$\alpha_1 \cdot (1-\beta_2)$
$\alpha_1 \cdot (1-\alpha_2)$
$\alpha_1 \cdot \alpha_2$

Figure A.2: Composition of two cascading simplified GEs leads to four different states.

Publications and Open Source Software

[GGG^{+}19] Sebastian Gallenmüller, René Glebke, Stephan Günther, Eric Hauser, Maurice Leclaire, Stefan Reif, Jan Rüth, Andreas Schmidt, Georg Carle, Thorsten Herfet, Wolfgang Schröder-Preikschat, and Klaus Wehrle. Enabling Wireless Network Support for Gain Scheduled Control. In *Proceedings of the 2nd International Workshop on Edge Systems, Analytics and Networking (EdgeSys)*, EdgeSys, Dresden, Germany, 2019. ACM.

[RSH^{+}17] Stefan Reif, Andreas Schmidt, Timo Hönig, Thorsten Herfet, and Wolfgang Schröder-Preikschat. X-Lap: A Systems Approach for Cross-layer Profiling and Latency Analysis for Cyber-Physical Networks. In *Proceedings of the 15th International Workshop on Real-Time Networks*, RTN, Dubrovnik, Croatia, June 2017.

[RSH^{+}18] Stefan Reif, Andreas Schmidt, Timo Hönig, Thorsten Herfet, and Wolfgang Schröder-Preikschat. Δelta: Differential Energy-Efficiency, Latency, and Timing Analysis for Real-Time Networks. In *Proceedings of the 16th International Workshop on Real-Time Networks (ECRTS RTN)*, RTN, Barcelona, Spain, July 2018. ACM SIGBED.

[SH16a] Andreas Schmidt and Thorsten Herfet. Approaches for Resilience- and Latency-Aware Networking. In *International Symposium on Networked Cyber-Physical Systems*, NetCPS, Munich, Germany, September 2016. IEEE.

[SH16b] Andreas Schmidt and Thorsten Herfet. Improving Multimedia Streaming from the Network's Core. In *Proceedings of the 6th International Conference on Consumer Electronics-Berlin*, ICCE-Berlin, Berlin, Germany, September 2016. IEEE.

[SH17a] Andreas Schmidt and Thorsten Herfet. NEAT: Network Experiment Automation Tool. In *Proceedings of the 1. KuVS Fachgespräch "Network Softwarization"*, KuVS-FG-NetSoft, Tübingen, Germany, October 2017.

[SH17b] Andreas Schmidt and Thorsten Herfet. Transparent Transmission Segmentation in Software-Defined Networks. In *Proceedings of the 3rd Conference on Network Softwarization*, NetSoft, Bologna, Italy, July 2017. IEEE.

[SRGP$^+$19] Andreas Schmidt, Stefan Reif, Pablo Gil Pereira, Timo Hönig, Thorsten Herfet, and Wolfgang Schröder-Preikschat. Cross-layer Pacing for Predictably Low Latency. In *Proceedings of the 6th International IEEE Workshop on Ultra-Low Latency in Wireless Networks (Infocom ULLWN)*, ULLWN, Paris, France, 2019. IEEE ComSoc.

[Tel18] Telecommunications Lab - Saarland University. Gstreamer Plugin for PRRT. http://gst-prrt.larn.systems, 2018.

[Tel19a] Telecommunications Lab - Saarland University. Predictably Reliable Real-time Transport (PRRT). http://prrt.larn.systems, 2019.

[Tel19b] Telecommunications Lab - Saarland University. X-Lap. http://xlap.larn.systems, 2019.

Bibliography

[AAGJ10] Assad Al Alam, Ather Gattami, and Karl Henrik Johansson. An experimental study on the fuel reduction potential of heavy duty vehicle platooning. In *13th International IEEE Conference on Intelligent Transportation Systems*, pages 306–311. IEEE, 2010.

[AB18] Venkat Arun and Hari Balakrishnan. Copa: Congestion control combining objective optimization with window adjustments. In *USENIX NSDI*, 2018.

[Ahm91] Shehzad Ahmed. Reduction of bottleneck operations in just-in-time manufacturing. 1991.

[AKEP12] Mohammad Alizadeh, Abdul Kabbani, Tom Edsall, and Balaji Prabhakar. Less is More: Trading a little Bandwidth for Ultra-Low Latency in the Data Center. In *Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*, pages 253–266, 2012.

[ALH$^+$18] Eneko Atxutegi, Fidel Liberal, Habtegebreil Kassaye Haile, Karl-Johan Grinnemo, Anna Brunstrom, and Ake Arvidsson. On the use of TCP BBR in cellular networks. *IEEE Communications Magazine*, 56(3):172–179, 2018.

[APB09] M. Allman, V. Paxson, and E. Blanton. TCP Congestion Control. RFC 5681 (Draft Standard), September 2009.

[ASA00] A. Aggarwal, S. Savage, and T. Anderson. Understanding the performance of TCP pacing. In *Proceedings of the 19th IEEE Conference on Computer Communications (INFOCOM)*, pages 1157–1165, 2000.

[Ayc03] John Aycock. A brief history of just-in-time. *ACM Computing Surveys (CSUR)*, 35(2):97–113, 2003.

[BA07] John Baillieul and Panos J. Antsaklis. Control and communication challenges in networked real-time systems. *Proceedings of the IEEE*, 95(1):9–28, 2007.

[BB95] Ajay Bakre and BR Badrinath. I-TCP: Indirect TCP for mobile hosts. In *Proceedings of 15th International Conference on Distributed Computing Systems*, pages 136–143. IEEE, 1995.

[BBDC11] Raffaele Bolla, Roberto Bruschi, Franco Davoli, and Flavio Cucchietti. Energy efficiency in the future internet: A survey of existing approaches and trends in energy-aware fixed network infrastructures. *IEEE Communications Surveys & Tutorials*, 13(2):223–244, 2011.

[BBF+19] Vaibhav Bajpai, Anna Brunstrom, Anja Feldmann, Wolfgang Kellerer, Aiko Pras, Henning Schulzrinne, Georgios Smaragdakis, Matthias Wählisch, and Klaus Wehrle. The dagstuhl beginners guide to reproducibility for experimental networking research. *arXiv preprint arXiv:1902.02165*, 2019.

[BCC+98] Bob Braden, David D. Clark, Jon Crowcroft, Bruce Davie, Steve Deering, Deborah Estrin, Sally Floyd, Van Jacobson, Greg Minshall, Craig Partridge, Larry Peterson, K.K. Ramakrishnan, Scott Shenker, John Wroclawski, and Lixia Zhang. Recommendations on queue management and congestion avoidance in the internet. RFC 2309, RFC Editor, April 1998. http://www.rfc-editor.org/rfc/rfc2309.txt.

[BCF08] François Baccelli, Giovanna Carofiglio, and Serguei Foss. Proxy Caching in Split TCP: Dynamics, Stability and Tail Asymptotics. In *IEEE INFOCOM 2008-The 27th Conference on Computer Communications*, pages 131–135. IEEE, 2008.

[BEK14] C. Bormann, M. Ersue, and A. Keranen. Terminology for constrained-node networks. RFC 7228, RFC Editor, May 2014. http://www.rfc-editor.org/rfc/rfc7228.txt.

[Bel57] Richard Bellman. A markovian decision process. *Indiana Univ. Math. J.*, 6:679–684, 1957.

[BF15] F. Baker and G. Fairhurst. IETF Recommendations Regarding Active Queue Management. BCP 197, RFC Editor, July 2015.

[Bir17] Daniel Birtel. Transparent Transmission Segmentation for Multimedia Applications, 2017. Master's Thesis, Saarland University, January 2017.

[BIYC06] Pete Beckman, Kamil Iskra, Kazutomo Yoshii, and Susan Coghlan. The influence of operating systems on the performance of collective operations at extreme scale. In *Proceedings of the 8th IEEE Annual International Conference on Cluster Computing (CLUSTER 2006)*, pages 1–13. IEEE, 2006.

[BKG$^+$01] J. Border, M. Kojo, J. Griner, G. Montenegro, and Z. Shelby. Performance Enhancing Proxies Intended to Mitigate Link-Related Degradations. RFC 3135 (Informational), June 2001.

[BLZ09] Leonardo Badia, Marco Levorato, and Michele Zorzi. A channel representation method for the study of hybrid retransmission-based error control. *IEEE Transactions on Communications*, 57(7):1959–1971, 2009.

[BM02] R. Bush and D. Meyer. Some internet architectural guidelines and philosophy. RFC 3439, RFC Editor, December 2002. http://www.rfc-editor.org/rfc/rfc3439.txt.

[BMPR17] Luiz Barroso, Mike Marty, David Patterson, and Parthasarathy Ranganathan. Attack of the Killer Microseconds. *Communications of the ACM*, 60(4):48–54, March 2017.

[BMZA12] Flavio Bonomi, Rodolfo Milito, Jiang Zhu, and Sateesh Addepalli. Fog computing and its role in the internet of things. In *Proceedings of the first edition of the MCC workshop on Mobile cloud computing*, pages 13–16. ACM, 2012.

[BOP94] Lawrence S Brakmo, Sean W O'Malley, and Larry L Peterson. *TCP Vegas: New techniques for congestion detection and avoidance*, volume 24. ACM, 1994.

[BPK$^+$14] Adam Belay, George Prekas, Ana Klimovic, Samuel Grossman, Christos Kozyrakis, and Edouard Bugnion. IX: A protected dataplane operating system for high throughput and low latency. In *Proc. of the 11th Symp. on Operating Systems Design and Implementation (OSDI'14)*, pages 49–65, 2014.

[BPZ00] Michael S Branicky, Stephen M Phillips, and Wei Zhang. Stability of networked control systems: Explicit analysis of delay. In *Proceedings of the 2000 American Control Conference. ACC (IEEE Cat. No. 00CH36334)*, volume 4, pages 2352–2357. IEEE, 2000.

[BS97] Kevin Brown and Suresh Singh. M-TCP: TCP for mobile cellular networks. *ACM SIGCOMM Computer Communication Review*, 27(5):19–43, 1997.

[BS06] Christian Becker and Armin Scholl. A survey on problems and methods in generalized assembly line balancing. *European journal of operational research*, 168(3):694–715, 2006.

[BSAK95] Hari Balakrishnan, Srinivasan Seshan, Elan Amir, and Randy H Katz. Improving TCP/IP performance over wireless networks. In *Proceedings of the 1st annual international conference on Mobile computing and networking*, pages 2–11. ACM, 1995.

[BSW73] G.M. Buxey, N.D. Slack, and Ray Wild. Production flow line system design–a review. *AIIE Transactions*, 5(1):37–48, 1973.

[CC84] Richard Comroe and Daniel Costello. ARQ schemes for data transmission in mobile radio systems. *IEEE Journal on Selected Areas in Communications*, 2(4):472–481, 1984.

[CCAP10] Tommaso Cucinotta, Fabio Checconi, Luca Abeni, and Luigi Palopoli. Self-tuning schedulers for legacy real-time applications. In *Proceedings of the 5th European Conference on Computer Systems (EuroSys'10)*, pages 55–68. ACM, 2010.

[CCG$^+$16] Neal Cardwell, Yuchung Cheng, C Stephen Gunn, Soheil Hassas Yeganeh, and Van Jacobson. BBR: Congestion-based congestion control. *ACM Queue*, 14(5):50, 2016.

[CCYJ17] Yuchung Cheng, Neal Cardwell, Soheil Hassas Yeganeh, and Van Jacobson. Delivery rate estimation - IETF DRAFT, 2017.

[CDCM15] Gaetano Carlucci, Luca De Cicco, and Saverio Mascolo. HTTP over UDP: an Experimental Investigation of QUIC. In *Proceedings of the 30th Annual ACM Symposium on Applied Computing*, pages 609–614. ACM, 2015.

[Che96] Stuart Cheshire. It's the latency, stupid. http://www.stuartcheshire.org/rants/latency.html, 1996.

[Cla18] David D Clark. *Designing an Internet.* MIT Press, 2018.

[CSB$^+$08] Joseph Chabarek, Joel Sommers, Paul Barford, Cristian Estan, David Tsiang, and Steve Wright. Power awareness in network design and routing. In *INFOCOM 2008. The 27th Conference on Computer Communications. IEEE*, pages 457–465. IEEE, 2008.

[CSL18] Hoon Sung Chwa, Kang G Shin, and Jinkyu Lee. Closing the gap between stability and schedulability: a new task model for cyber-physical systems. In *2018 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 327–337. IEEE, 2018.

[Cyt18] Cython. Cython. https://cython.org/, 2018.

[DB13] Jeffrey Dean and Luiz André Barroso. The tail at scale. *Communications of the ACM*, 56(2):74–80, 2013.

[DLZ$^+$15] Mo Dong, Qingxi Li, Doron Zarchy, P Brighten Godfrey, and Michael Schapira. PCC: Re-architecting congestion control for consistent high performance. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, pages 395–408, 2015.

[DM06] Nandita Dukkipati and Nick McKeown. Why flow-completion time is the right metric for congestion control. *ACM SIGCOMM Computer Communication Review*, 36(1):59–62, 2006.

[Dom17] Christian Dombrowski. *Design and Evaluation of an Ultra-reliable Low-latency Wireless Network Protocol.* PhD thesis, RWTH Aachen University, 2017.

[DPS18] Erik Dahlman, Stefan Parkvall, and Johan Skold. *5G NR: The next generation wireless access technology.* Academic Press, 2018.

[DS16] Werner Damm and Janos Sztipanovits, editors. *CPS Summit Action Plan: Towards a Cross-Cutting Science of Cyber-Physical System for Mastering all-Important Engineering Challenges.* 2016.

[DXHL14] Li Da Xu, Wu He, and Shancang Li. Internet of things in industries: A survey. *IEEE Transactions on industrial informatics*, 10(4):2233–2243, 2014.

[EGR$^+$15] Paul Emmerich, Sebastian Gallenmüller, Daniel Raumer, Florian Wohlfart, and Georg Carle. Moongen: A scriptable high-speed packet generator. In *Proceedings of the 2015 Internet Measurement Conference*, pages 275–287. ACM, 2015.

[Ell63] Edwin O Elliott. Estimates of error rates for codes on burst-noise channels. *The Bell System Technical Journal*, 42(5):1977–1997, 1963.

[FA14] Gerhard Fettweis and Siavash Alamouti. 5G: Personal Mobile Internet beyond What Cellular Did to Telephony. *Communications Magazine, IEEE*, 52(2):140–145, 2014.

[Fei54] Amiel Feinstein. A new basic theorem of information theory. 1954.

[FFF09] Thomas Ferrandiz, Fabrice Francès, and Christian Fraboul. A method of computation for worst-case delay analysis on SpaceWire networks. In *Proceedings of the 11th IEEE International Symposium on Industrial Embedded Systems (SIES 2009)*, pages 19–27. IEEE, 2009.

[FJ94] Sally Floyd and Van Jacobson. The synchronization of periodic routing messages. *IEEE/ACM Transactions on Networking (TON)*, 2(2):122–136, 1994.

[FL03] Cheng Peng Fu and Soung C Liew. TCP Veno: TCP enhancement for transmission over wireless access networks. *IEEE Journal on selected areas in communications*, 21(2):216–228, 2003.

[FMXY11] Xi Fang, Satyajayant Misra, Guoliang Xue, and Dejun Yang. Smart grid—the new and improved power grid: A survey. *IEEE communications surveys & tutorials*, 14(4):944–980, 2011.

[FW17] G. Fairhurst and M. Welzl. The Benefits of Using Explicit Congestion Notification (ECN). RFC 8087, RFC Editor, March 2017.

[FWK16] Marcel Flores, Alexander Wenzel, and Aleksandar Kuzmanovic. Enabling router-assisted congestion control on the internet. In *2016 IEEE 24th International Conference on Network Protocols (ICNP)*, pages 1–10. IEEE, 2016.

[GG13] Monia Ghobadi and Yashar Ganjali. TCP pacing in data center networks. In *Proceedings of the 21st IEEE Annual Symposium on High-Performance Interconnects, (HOTI)*, pages 25–32, 2013.

[GHJ$^+$09] Albert Greenberg, James R Hamilton, Navendu Jain, Srikanth Kandula, Changhoon Kim, Parantap Lahiri, David A Maltz, Parveen Patel, and Sudipta Sengupta. Vl2: a scalable and flexible data center network. In *ACM SIGCOMM computer communication review*, volume 39, pages 51–62. ACM, 2009.

[Gil60] Edgar N Gilbert. Capacity of a burst-noise channel. *Bell system technical journal*, 39(5):1253–1265, 1960.

[GKRW13] Kenneth Gillingham, Matthew J Kotchen, David S Rapson, and Gernot Wagner. Energy policy: The rebound effect is overplayed. *Nature*, 493(7433):475, 2013.

[GN12] Jim Gettys and Kathleen Nichols. Bufferbloat: Dark Buffers in the Internet. *Communications of the ACM*, 55(1):57 – 65, 2012.

[Goo18] Google Chromium Project. QUIC, a multiplexed stream transport over UDP, 2018.

[Gor12] Manuel Gorius. *Adaptive Delay-constrained Internet Media Transport.* PhD thesis, Saarland University, 2012.

[GS03] Maruti Gupta and Suresh Singh. Greening of the Internet. In *Proceedings of the 2003 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 19–26. ACM, 2003.

[GSC11] Dip Goswami, Reinhard Schneider, and Samarjit Chakraborty. Co-design of cyber-physical systems via controllers with flexible delay constraints. In *Proceedings of the 16th Asia and South Pacific Design Automation Conference*, pages 225–230. IEEE Press, 2011.

[GSG$^+$15] Matthew Grosvenor, Malte Schwarzkopf, Ionel Gog, Robert Watson, Andrew Moore, Steven Hand, and Jon Crowcroft. Queues don't matter when you can JUMP them! In *Proc. of the 12th Symposium on Networked Systems Design and Implementation (NSDI'15)*, pages 1–14. USENIX Association, 2015.

[Gst18] Gstreamer. Gstreamer. https://gstreamer.freedesktop.org/, 2018.

[Hai18] Haivision. SRT - Secure Reliable Transport. https://github.com/haivision/srt, 2018.

[HBZ17] Mario Hock, Roland Bless, and Martina Zitterbart. Experimental Evaluation of BBR Congestion Control. In *Proceedings of the 25th IEEE International Conference on Network Protocols*, ICNP. IEEE, 2017.

[HD15] Osama Haq and Fahad R Dogar. Leveraging the power of cloud for reliable wide area communication. In *Proceedings of the 14th ACM Workshop on Hot Topics in Networks*, page 19. ACM, 2015.

[HFGN12] T. Henderson, S. Floyd, A. Gurtov, and Y. Nishida. The NewReno Modification to TCP's Fast Recovery Algorithm. RFC 6582, RFC Editor, April 2012. http://www.rfc-editor.org/rfc/rfc6582.txt.

[HH08] Gerhard Haßlinger and Oliver Hohlfeld. The Gilbert-Elliott model for packet loss in real time services on the Internet. In *14th GI/ITG Conference-Measurement, Modelling and Evalutation of Computer and Communication Systems*, pages 1–15. VDE, 2008.

[HNZB17] Mario Hock, Felix Neumeister, Martina Zitterbart, and Roland Bless. TCP LoLa: Congestion Control for Low Latencies and High Throughput. In *2017 IEEE 42nd Conference on Local Computer Networks (LCN)*, pages 215–218. IEEE, 2017.

[HRX08] Sangtae Ha, Injong Rhee, and Lisong Xu. CUBIC: A New TCP-Friendly High-Speed TCP Variant. *ACM SIGOPS Operating Systems Review - Research and developments in the Linux kernel*, 42(5):64–74, 2008.

[HSM$^+$10] Brandon Heller, Srinivasan Seetharaman, Priya Mahadevan, Yiannis Yiakoumis, Puneet Sharma, Sujata Banerjee, and Nick McKeown. Elastictree: Saving energy in data center networks. In *NSDI*, volume 10, pages 249–264, 2010.

[HVLA$^+$07] D. Hristu-Varsakelis, W.S. Levine, Rajeev Alur, Karl-Erik Årzén, John Baillieul, and Tom Henzinger. *Handbook of networked and embedded control systems.* Springer Science & Business Media, 2007.

[IET18] IETF. QUIC. https://datatracker.ietf.org/wg/quic/charter/, 2018.

[Int18] Intel Corporation, Santa Clara, California, USA. *Intel Architecture Software Developer's Manual*, 2018.

[Itk76] Uri Itkis. *Control systems of variable structure.* Wiley New York, 1976.

[JBM$^+$17] Sabina Jeschke, Christian Brecher, Tobias Meisen, Denis Özdemir, and Tim Eschert. Industrial internet of things and cyber manufacturing systems. In *Industrial Internet of Things*, pages 3–19. Springer, 2017.

[JCS$^+$06] Hakyung Jung, Nakjung Choi, Yongho Seok, Taekyoung Kwon, and Yanghee Choi. Augmented Split-TCP over Wireless LANs. In *2006 IEEE International Conference on Communications*, volume 12, pages 5420–5425. IEEE, 2006.

[Jev65] William Stanley Jevons. *The coal question.* 1865.

[JK88] Van Jacobson and Michael J. Karels. Congestion avoidance and control. In *ACM SIGCOMM computer communication review*, volume 18, pages 314–329. ACM, 1988.

[JSBM15] Keon Jang, Justine Sherry, Hitesh Ballani, and Toby Moncaster. Silo: Predictable Message Latency in the Cloud. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication (SIGCOMM)*, pages 435–448, 2015.

[JWJ$^+$14] EunYoung Jeong, Shinae Wood, Muhammad Jamshed, Haewon Jeong, Sunghwan Ihm, Dongsu Han, and KyoungSoo Park. mtcp: a highly scalable user-level TCP stack for multicore systems. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 489–502, 2014.

[KAGS05] Hermann Kopetz, Astrit Ademaj, Petr Grillinger, and Klaus Steinhammer. The time-triggered ethernet (TTE) design. 2005.

[Kar15] Michael Karl. *Optimized Wide Area Media Transport Strategies.* PhD thesis, Saarland University, 2015.

[KGH10] Michael Karl, Manuel Gorius, and Thorsten Herfet. Routing: Why less intelligence sometimes is more clever. In *2010 IEEE International Symposium on Broadband Multimedia Systems and Broadcasting (BMSB)*, pages 1–6. IEEE, 2010.

[KH11] Michael Karl and Thorsten Herfet. On the efficient segmentation of network links. In *2011 IEEE International Symposium on Broadband Multimedia Systems and Broadcasting (BMSB)*, pages 1–5. IEEE, 2011.

[KH14] Michael Karl and Thorsten Herfet. Transparent multi-hop protocol termination. In *2014 IEEE 28th International Conference on Advanced Information Networking and Applications*, pages 253–259. IEEE, 2014.

[KKFT02] Swastik Kopparty, Srikanth V Krishnamurthy, Michalis Faloutsos, and Satish K Tripathi. Split TCP for mobile ad hoc networks. In *Global Telecommunications Conference. GLOBECOM'02*, volume 1, pages 138–142. IEEE, 2002.

[KKK17] Minho Kim, Seung-Woo Ko, and Seong-Lyun Kim. Enhancing TCP end-to-end performance in millimeter-wave communications. In *2017 IEEE 28th Annual International Symposium on Personal, Indoor, and Mobile Radio Communications (PIMRC)*, pages 1–5. IEEE, 2017.

[Kle78] Leonard Kleinrock. On flow control in computer networks. In *Proceedings of the International Conference on Communications*, volume 2, pages 27–2, 1978.

[Kle18] Leonard Kleinrock. Internet Congestion Control Using the Power Metric: Keep the Pipe Just Full, But No Fuller. *Ad Hoc Networks*, 80:142–157, 2018.

[KNT13] Mirja Kühlewind, Sebastian Neuner, and Brian Trammell. On the state of ECN and TCP options on the Internet. In *International Conference on Passive and Active Network Measurement*, pages 135–144. Springer, 2013.

[KOWT11] Tobias Klug, Michael Ott, Josef Weidendorfer, and Carsten Trinitis. autopin - Automated Optimization of Thread-to-Core Pinning on Multicore Systems. *Transactions on high-performance embedded architectures and compilers*, pages 219–235, 2011.

[KP87] Phil Karn and Craig Partridge. Improving round-trip time estimates in reliable transport protocols. In *ACM SIGCOMM Computer Communication Review*, volume 17, pages 2–7. ACM, 1987.

[KRV$^+$14] Diego Kreutz, Fernando Ramos, Paulo Verissimo, Christian Esteve Rothenberg, Siamak Azodolmolky, and Steve Uhlig. Software-defined networking: A comprehensive survey. *arXiv preprint arXiv:1406.0440*, 2014.

[Lam78] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, July 1978.

[LB05] Jean-Yves Le Boudec. Rate adaptation, congestion control and fairness: A tutorial. *Web page, November*, 2005.

[LBBN16] Meng Liu, Matthias Becker, Moris Behnam, and Thomas Nolte. Using Segmentation to Improve Schedulability of Real-Time Traffic over RRA-based NoCs. *ACM SIGBED Review*, 13(4):20–24, September 2016.

[LC04] Shu Lin and Daniel J. Costello. *Error Control Coding, Second Edition.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 2004.

[Lee08] Edward A. Lee. Cyber Physical Systems: Design Challenges. In *Proceedings of the 11th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC)*, 2008.

[Lee17] Edward Ashford Lee. *Plato and the Nerd: The Creative Partnership of Humans and Technology.* MIT Press, 2017.

[LEL$^+$16] Per Lindgren, Johan Eriksson, Marcus Lindner, Andreas Lindner, David Pereira, and Lus Miguel Pinho. End-to-end response time of IEC 61499 Distributed Applications Over Switched Ethernet. *IEEE Transactions on Industrial Informatics*, 13(1):287–297, 2016.

[LFK$^+$14] Heiner Lasi, Peter Fettke, Hans-Georg Kemper, Thomas Feld, and Michael Hoffmann. Industry 4.0. *Business & information systems engineering*, 6(4):239–242, 2014.

[Lin18] Linux Kernel Developers. SocketCAN. https://www.kernel.org/doc/Documentation/networking/can.txt, 2018.

[LL15] Ke Liu and Jack YB Lee. On improving TCP performance over mobile data networks. *IEEE transactions on mobile computing*, 15(10):2522–2536, 2015.

[LNPV18] Kirsten Liere-Netheler, Sven Packmohr, and Kirsten Vogelsang. Drivers of digital transformation in manufacturing. *The Digital Supply Chain of the Future: Technologies, Applications and Business Models;*, 2018.

[LRW09] Sameer Ladiwala, Ramaswamy Ramaswamy, and Tilman Wolf. Transparent TCP acceleration. *Computer Communications*, 32(4):691–702, 2009.

[LS17] Edward A. Lee and Sanjit A. Seshia. *Introduction to Embedded systems: A Cyber-Physical Systems Approach.* MIT Press, Second edition, 2017.

[LSH11] Etienne Le Sueur and Gernot Heiser. Slow down or sleep, that is the question. In *Proceedings of the USENIX Annual Technical Conference (ATC 2011)*, pages 1–6. USENIX, 2011.

[MAB$^+$08] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. OpenFlow: enabling innovation in campus networks. *ACM SIGCOMM Computer Communication Review*, 38(2):69–74, 2008.

[Mar05] Martijn Faassen. What is Pythonic? https://blog.startifact.com/posts/older/what-is-pythonic.html, accessed 2018-12-04, 2005.

[Mar11] Peter Marwedel. *Embedded System Design*. 2011.

[MCG$^+$01] Saverio Mascolo, Claudio Casetti, Mario Gerla, Medy Y Sanadidi, and Ren Wang. TCP Westwood: Bandwidth estimation for enhanced transport over wireless links. In *Proceedings of the 7th annual international conference on Mobile computing and networking*, pages 287–297. ACM, 2001.

[MLVH$^+$02] Akihiko Miyoshi, Charles Lefurgy, Eric Van Hensbergen, Ram Rajamony, and Raj Rajkumar. Critical power slope: Understanding the runtime effects of frequency scaling. In *Proceedings of the 16th International Conference on Supercomputing*, ICS, pages 35–44. ACM, 2002.

[MMBK10] D. Mills, J. Martin, J. Burbank, and W. Kasch. Network time protocol version 4: Protocol and algorithms specification. RFC 5905, RFC Editor, June 2010. http://www.rfc-editor.org/rfc/rfc5905.txt.

[MNB$^+$17] Marcus R. Munafò, Brian A. Nosek, Dorothy V. M. Bishop, Katherine S. Button, Christopher D. Chambers, Nathalie Percie Du Sert, Uri Simonsohn, Eric-Jan Wagenmakers, Jennifer J. Ware, and John P. A. Ioannidis. A manifesto for reproducible science. *Nature human behaviour*, 1(1):0021, 2017.

[MNP02] Alain J. Martin, Mika Nyström, and Paul I. Pénzes. ET^2: A metric for time and energy efficiency of computation. In Robert Graybill and Rami Melhem, editors, *Power Aware Computing*, pages 293–315. Kluwer Academic Publishers, May 2002.

[Moc87] P. Mockapetris. Domain names - implementation and specification. STD 13, RFC Editor, November 1987. http://www.rfc-editor.org/rfc/rfc1035.txt.

[Mog95] Jeffrey C Mogul. The case for persistent-connection HTTP. 25(4), 1995.

[Moo02] Tim Moors. A critical review of “End-to-end arguments in system design”. In *ICC 2002 - IEEE International Conference on Communications*, volume 2, pages 1214–1219. IEEE, 2002.

[MSG$^+$15] Rashid Mijumbi, Joan Serrat, Juan-Luis Gorricho, Niels Bouten, Filip De Turck, and Raouf Boutaba. Network function virtualization: State-of-the-art and research challenges. *IEEE Communications Surveys & Tutorials*, 18(1):236–262, 2015.

[NB99] Jörg Nonnenmacher and Ernst W Biersack. Scalable feedback for large groups. *IEEE/ACM transactions on Networking*, (3):375–386, 1999.

[NJ12] Kathleen Nichols and Van Jacobson. Controlling Queue Delay. *Communications of the ACM*, 55(7):42–50, 2012.

[OAL14] Anne-Cecile Orgerie, Marcos Dias de Assuncao, and Laurent Lefevre. A survey on techniques for improving the energy efficiency of large-scale distributed systems. *ACM Computing Surveys (CSUR)*, 46(4):47, 2014.

[Ohn88] Taiichi Ohno. *Toyota production system: beyond large-scale production.* crc Press, 1988.

[PA13] Simon Peter and Thomas Anderson. Arrakis: A case for the end of the empire. In *Proc. of the 14th Conference on Hot Topics in Operating Systems (HotOS'13)*, pages 1–7. USENIX Association, 2013.

[PBA17] Mia Primorac, Edouard Bugnion, and Katerina Argyraki. How to measure the killer microsecond. In *Proceedings of the Workshop on Kernel-Bypass Networks*, pages 37–42. ACM, 2017.

[PFLL18] Nikolai Pitaev, Matthias Falkner, Aris Leivadeas, and Ioannis Lambadaris. Characterizing the performance of concurrent virtualized network functions with OVS-DPDK, FD.IO VPP and SR-IOV. In *Proceedings of the 2018 ACM/SPEC International Conference on Performance Engineering*, pages 285–292. ACM, 2018.

[PGH$^+$19] Jan Pennekamp, René Glebke, Martin Henze, Tobias Meisen, Christoph Quix, Rihan Hai, Lars Gleim, Philipp Niemietz, Maximilian Rudack, Simon Knape, et al. Towards an Infrastructure Enabling the Internet of Production. In *2019 IEEE International Conference on Industrial Cyber Physical Systems (ICPS)*, pages 31–37. IEEE, 2019.

[Plu82] David C. Plummer. Ethernet address resolution protocol: Or converting network protocol addresses to 48.bit ethernet address for transmission on ethernet hardware. STD 37, RFC Editor, November 1982. http://www.rfc-editor.org/rfc/rfc826.txt.

[PLZ$^+$14] Simon Peter, Jialin Li, Irene Zhang, Dan R. K. Ports, Doug Woos, Arvind Krishnamurthy, Thomas Anderson, and Timothy Roscoe. Arrakis: The operating system is the control plane. In *Proc. of the 11th Symposium on Operating Systems Design and Implementation (OSDI'14)*, pages 1–16, 2014.

[PLZ$^+$15] Simon Peter, Jialin Li, Irene Zhang, Dan Ports, Doug Woos, Arvind Krishnamurthy, Thomas Anderson, and Timothy Roscoe. Arrakis: The operating system is the control plane. *Transactions on Computer Systems (TOCS)*, 33(4):11:1–11:30, November 2015.

[PMZ$^+$17] Michele Polese, Marco Mezzavilla, Menglei Zhang, Jing Zhu, Sundeep Rangan, Shivendra Panwar, and Michele Zorzi. milliProxy: A TCP proxy architecture for 5G mmWave cellular systems. In *2017 51st Asilomar Conference on Signals, Systems, and Computers*, pages 951–957. IEEE, 2017.

[POB$^+$15] Jonathan Perry, Amy Ousterhout, Hari Balakrishnan, Devavrat Shah, and Hans Fugal. Fastpass: A centralized zero-queue datacenter network. *ACM SIGCOMM Computer Communication Review*, 44(4):307–318, 2015.

[Pos81] Jon Postel. Transmission control protocol. STD 7, RFC Editor, September 1981. http://www.rfc-editor.org/rfc/rfc793.txt.

[PPV09] Yury Polyanskiy, Vincent Poor, and Sergio Verdú. Channel Coding Rate in the Finite Blocklength Regime. *IEEE Transactions on Information Theory*, 55(4):2306–2360, May 2009.

[PRGS18] Paul Pop, Michael Lander Raagaard, Marina Gutierrez, and Wilfried Steiner. Enabling fog computing for industrial automation through time-sensitive networking (TSN). *IEEE Communications Standards Magazine*, 2(2):55–61, 2018.

[PS17] C. Perkins and V. Singh. Multimedia Congestion Control: Circuit Breakers for Unicast RTP Sessions. RFC 8083, RFC Editor, March 2017.

[PTK94] Sridhar Pingali, Don Towsley, and James F Kurose. A comparison of sender-initiated and receiver-initiated reliable multicast protocols. In *ACM SIGMETRICS Performance Evaluation Review*, volume 22, pages 221–230. ACM, 1994.

[PWH10] Abhinav Pathak, Y. Wang, and Cheng Huang. Measuring and evaluating TCP splitting for cloud services. *PAM'10 Proceedings of the 11th international conference on Passive and active measurement*, pages 41–50, 2010.

[PWSB11] D. Papadimitriou, M. Welzl, M. Scharf, and B. Briscoe. Open research issues in internet congestion control. RFC 6077, RFC Editor, February 2011. http://www.rfc-editor.org/rfc/rfc6077.txt.

[RCC$^+$11] Sivasankar Radhakrishnan, Yuchung Cheng, Jerry Chu, Arvind Jain, and Barath Raghavan. Tcp fast open. In *Proceedings of the Seventh COnference on emerging Networking EXperiments and Technologies*, page 21. ACM, 2011.

[RFB01] K. Ramakrishnan, S. Floyd, and D. Black. The Addition of Explicit Congestion Notification (ECN) to IP. RFC 3168, RFC Editor, September 2001. http://www.rfc-editor.org/rfc/rfc3168.txt.

[RGW$^+$18] Jan Rüth, René Glebke, Klaus Wehrle, Vedad Causevic, and Sandra Hirche. Towards in-network industrial feedback control. In *Proceedings of the 2018 Morning Workshop on In-Network Computing*, pages 14–19. ACM, 2018.

[RKR$^+$17] Felix Rath, Johannes Krude, Jan Rüth, Daniel Schemmel, Oliver Hohlfeld, Jó Á Bitsch, and Klaus Wehrle. Symperf: Predicting network function performance. In *Proceedings of the SIGCOMM Posters and Demos*, pages 34–36. ACM, 2017.

[RL10] Fengyuan Ren and Chuang Lin. Modeling and improving TCP performance over cellular link with variable bandwidth. *IEEE Transactions on Mobile Computing*, 10(8):1057–1070, 2010.

[RLSS10] R. Rajkumar, Insup Lee, Lui Sha, and J. Stankovic. Cyber-physical systems: The next computing revolution. *Design Automation Conference (DAC), 2010 47th ACM/IEEE*, pages 0–5, 2010.

[ROS$^+$11] Stephen Rumble, Diego Ongaro, Ryan Stutsman, Mendel Rosenblum, and John Ousterhout. It's time for low latency. In *Proceedings of the 13th Conference on Hot Topics in Operating Systems (HotOS'11)*, pages 1–5, 2011.

[RWS$^+$19] Jan Rüth, Konrad Wolsing, Martin Serror, Klaus Wehrle, and Oliver Hohlfeld. Blitz-starting QUIC Connections. *arXiv preprint arXiv:1905.03144*, 2019.

[SB$^+$98] Richard S. Sutton, Andrew G. Barto, et al. *Introduction to reinforcement learning*, volume 2. MIT press Cambridge, 1998.

[SBK$^+$16] Giuseppe Siracusano, Roberto Bifulco, Simon Kuenzer, Stefano Salsano, Nicola Blefari Melazzi, and Felipe Huici. On-the-Fly TCP Acceleration with Miniproxy. *arXiv preprint arXiv:1605.06285*, pages 44–49, 2016.

[SCFJ03] H. Schulzrinne, S. Casner, R. Frederick, and V. Jacobson. RTP: A Transport Protocol for Real-Time Applications. STD 64, RFC Editor, July 2003. http://www.rfc-editor.org/rfc/rfc3550.txt.

[SCGM14] Ankit Singla, Balakrishnan Chandrasekaran, P. Brighten Godfrey, and Bruce Maggs. The Internet at the Speed of Light. *Proceedings of the 13th ACM Workshop on Hot Topics in Networks - HotNets-XIII*, pages 1–7, 2014.

[SCZ$^+$16] Weisong Shi, Jie Cao, Quan Zhang, Youhuizi Li, and Lanyu Xu. Edge computing: Vision and challenges. *IEEE Internet of Things Journal*, 3(5):637–646, 2016.

[SDV$^+$17] Ahmed Saeed, Nandita Dukkipati, Vytautas Valancius, Vinh The Lam, Carlo Contavalli, and Amin Vahdat. Carousel: Scalable Traffic Shaping at End Hosts. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, pages 404–417, 2017.

[SG10] Andrew Sorensen and Henry Gardner. Programming with time: cyber-physical programming with impromptu. In *ACM Sigplan Notices*, volume 45, pages 822–834. ACM, 2010.

[Sha48] Claude Elwood Shannon. A Mathematical Theory of Communication. 1948.

[Sno03] Deborah Snoonian. Smart buildings. *IEEE spectrum*, 40(8):18–23, 2003.

[SNTH13] Geir Kjetil Sandve, Anton Nekrutenko, James Taylor, and Eivind Hovig. Ten simple rules for reproducible computational research, 2013.

[SRC84] Jerome H. Saltzer, David P. Reed, and David D. Clark. End-To-End Arguments in System Design. *ACM Transactions on Computer Systems*, 2(4):277–288, 1984.

[SRX$^+$04] R. Stewart, M. Ramalho, Q. Xie, M. Tuexen, and P. Conrad. Stream Control Transmission Protocol (SCTP) Partial Reliability Extension. RFC 3758, RFC Editor, May 2004. http://www.rfc-editor.org/rfc/rfc3758.txt.

[Sta17] John A Stankovic. Research Directions for Cyber Physical Systems in Wireless and Mobile Healthcare. *ACM Transactions on Cyber-Physical Systems*, 1(1):1–12, 2017.

[Ste92] Jonathan Steuer. Defining virtual reality: Dimensions determining telepresence. *Journal of communication*, 42(4):73–93, 1992.

[SZ10] Andreas Schimmel and Alois Zoitl. Real-time communication for IEC 61499 in switched ethernet networks. In *Proceedings of the 2nd IEEE International Congress on Ultra Modern Telecommunications and Control Systems and Workshops*, ICUMT, pages 406–411. IEEE, 2010.

[SZC90] Scott Shenker, Lixia Zhang, and David D. Clark. Some observations on the dynamics of a congestion control algorithm. *ACM SIGCOMM Computer Communication Review*, 20(5):30–39, 1990.

[Tan08] Guoping Tan. *Optimum Hybrid Error Correction Scheme under Strict Delay Constraints*. PhD thesis, Saarland University, 2008.

[TEFK05] Dan Tsafrir, Yoav Etsion, Dror G. Feitelson, and Scott Kirkpatrick. System noise, OS clock ticks, and fine-grained parallel applications. In *Proceedings of the 19th annual international conference on Supercomputing*, pages 303–312. ACM, 2005.

[TG04] Shu Tao and Roch Guérin. On-line estimation of internet path performance: an application perspective. In *IEEE INFOCOM 2004*, volume 3, pages 1774–1785. IEEE, 2004.

[TS12] Brian Trammell and Dominik Schatzmann. On flow concurrency in the internet and its implications for capacity sharing. In *Proceedings of the 2012 ACM workshop on Capacity sharing*, pages 15–20. ACM, 2012.

[VGM+13] Ashish Vulimiri, Philip Brighten Godfrey, Radhika Mittal, Justine Sherry, Sylvia Ratnasamy, and Scott Shenker. Low latency via redundancy. In *Proceedings of the ninth ACM conference on Emerging networking experiments and technologies*, pages 283–294. ACM, 2013.

[Vog19] Kai Vogelgesang. Fast Transparent Transmission Segmentation with Kernel-Bypass Networking, 2019. Bachelor's Thesis, Saarland University, April 2019.

[WEE+08] Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, Frank Mueller, Isabelle Puaut, Peter Puschner, Jan Staschulat, and Per Stenström. The worst-case execution-time problem—overview of methods and survey of tools. *Transactions on Embedded Computing Systems (TECS)*, 7(3):36:1–36:53, May 2008.

[Wie48] Norbert Wiener. *Cybernetics or Control and Communication in the Animal and the Machine.* The Technology Press, 1948.

[Wir18] Wireshark. Dissectors. https://wiki.wireshark.org/Lua/Dissectors, 2018.

[WL06] David X Wei and Steven H Low. TCP Pacing Revisited. In *Proceedings of IEEE INFOCOM*, pages 1–11, 2006.

[YMH+18] Francis Y Yan, Jestin Ma, Greg D Hill, Deepti Raghavan, Riad S Wahby, Philip Levis, and Keith Winstein. Pantheon: the training ground for internet congestion-control research. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 731–743, 2018.

[YMKT99] Maya Yajnik, Sue Moon, Jim Kurose, and Don Towsley. Measurement and modelling of the temporal dependence in packet loss. In *IEEE INFOCOM 1999. Proceedings of the 18th Conference on Computer Communications.*, volume 1, pages 345–352. IEEE, 1999.

[YMT05] Xunqi Yu, James W. Modestino, and Xusheng Tian. The accuracy of Gilbert models in predicting packet-loss statistics for a single-multiplexer network model. In *Proceedings IEEE 24th Annual Joint Conference of the IEEE Computer and Communications Societies.*, volume 4, pages 2602–2612. IEEE, 2005.

[ZMRP19] Menglei Zhang, Marco Mezzavilla, Sundeep Rangan, and Shivendra Panwar. Improving Google's BBR for Reduced Latency and Increased Fairness, 2019.

[ZMSS19] Doron Zarchy, Radhika Mittal, Michael Schapira, and Scott Shenker. Axiomatizing congestion control. *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, 3(2):33, 2019.

[ZPM$^+$19] Menglei Zhang, Michele Polese, Marco Mezzavilla, Jing Zhu, Sundeep Rangan, Shivendra Panwar, and Michele Zorzi. Will TCP Work in mmWave 5G Cellular Networks? *IEEE Communications Magazine*, 57(1):65–71, 2019.

[ZSC91] Lixia Zhang, Scott Shenker, and David D. Clark. Observations on the Dynamics of a Congestion Control Algorithm: The Effects of Two-Way Traffic. *ACM Computer Communication Review*, 21(4):133–147, 1991.

This dissertation was typeset using LaTeX, originally developed by Leslie Lamport and based on Donald Knuth's TeX. The body text is set in 12 point with Palatino designed by Hermann Zapf. This document is using the Memoir class created by Peter R. Wilson. The diagrams contain icons from FontAwesome 4.0, licensed under SIL OFL 1.1.