This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TNSM.2019.2948467, IEEE Transactions on Network and Service Management

XTRA: Towards Portable Transport Layer Functions

Giuseppe Bianchi[1,2], Michael Welzl[3], Angelo Tulumello[1,2], Francesco Gringoli[1,4], Giacomo Belocchi[1,2], Marco Faltelli[1,2], Salvatore Pontarelli[1]
[1]Consorzio Nazionale Interuniversitario per le Telecomunicazioni (CNIT), Italy.
[2]University of Rome Tor Vergata, Italy.
[3]Department of Informatics, University of Oslo, Norway.
[4]University of Brescia, Italy.

***Abstract*—XTRA (XFSM for Transport) aims at providing a first attempt towards a "*code-once-port-everywhere*" platform-agnostic programming abstraction tailored to the deployment of transport layer functions. XTRA's programming abstraction not only fits SW platforms, but is specifically designed to harness, with no re-coding effort, the offloading opportunities offered by CPU-less HW boards or smart NICs. We demonstrate the viability of XTRA with three completely different implementations of the underlying execution engine (HW proof-of-concept on a NetFPGA board, User-space SW over Linux' Open Data Plane, and NS3 emulator). Flexibility is shown via a number of example applications, ranging from a variety of congestion control algorithms, to a middlebox-type TCP proxy functionality, up to a customized "Timer-Based" (TB) TCP which leverages the native reliance of XTRA on timers, so as to produce a loss recovery operation which, despite being formalized only via a handful of code lines, performs almost comparable with the highly optimized Linux and FreeBSD implementations.**

***Index Terms*—SDN, NFV, FPGA, DSL, APIs, state machines.**

I. INTRODUCTION

The ability to program, configure and efficiently deploy virtualized network functions irrespective of the underlying devices and platforms, be they HW or SW, is key for a simple and effective network management. Modern fully programmable data planes (e.g., P4 [1], [2]) have permitted to attain wire-speed operation by software-implemented network functions. Still, they do support only *stateless lower-layer* functions driven by packet arrivals, and hardly fit with the inherently *state-oriented* functions of the transport layer which are not only driven by packet forwarding events, but also by time-related events. For example, a TCP-like protocol needs to carry out functions when a timer (based on stored information) fires, not just when a packet arrives, and need to use and update stored information well beyond packet header fields.

With this paper, we posit that network management and innovation in network protocols will greatly benefit from the availability of programming abstractions that allow to implement stateful higher-layer network functions. We specifically focus on transport-layer tasks which involve network functions beyond packet-level "on-the-fly" operations — operations that fully depend on protocol states, timers, opportunistic packet buffering, signalling messages, etc. These functions are used in very different parts of modern networks, and range from relatively simple connection tracking to mechanisms that require a full transport protocol implementation, e.g. TCP link splitters or other types of proxies. The need to support flexibility at higher layers is highlighted by recent proposals such as Microboxes [3]; yet Microboxes are constructed around TCP and are not directly focused on code portability.

"Code-once-port-everywhere" — Our main design objective was to identify programming abstractions which retain a flexibility comparable to a "standard" software implementation, but at the same time guarantee the possibility to operate at wire speed when "ported" on tailored hardware. This permits to challenge the portability from SW end-systems to *bump-in-the-wire* HW devices — e.g. smart NICs [4]–[7]. NICs capable of relieving the cloud from CPU-costly protocol tasks would cheaply enable a number of performance optimizations (e.g. accurate pacing [8], [9] at no CPU cost), as well as offloading scenarios such as connection management and handshake protection (see the SYN-proxy use-case in Section VII-C). A sufficiently expressive programming abstraction could ultimately permit offloading of the entire transport protocol stack in hardware, or even implementing more sophisticated cloud-like functions [10].

Flexibility/performance/openness trade-offs — One way to address the trade-off consists in providing fine-grained control of the (hardware) platform via a programming language, meanwhile "*hiding*" the platform's internals via a compiler. This approach was successfully applied to P4 programmable switches [1], [2], and was later used by ClickNP [11] to provide a modular programming abstraction tailored to FPGA platforms, but resembling the Click Modular Router with elements programmed using a C-like language. A similar strategy was more recently envisioned for higher layer network functions by Emu [12], which relies on a compiler to cast C# functions into a NetFPGA SUME Verilog hardware description. Still, run-time HW production from higher level descriptions does not come free of concerns. Especially when targeting actual product development, practical programming needs further design phases dedicated to HW verification and to back-end implementation (i.e. synthesis, placement and routing tasks, which require a deep understanding of digital design techniques), and necessarily restricts deployment to programmable HW — e.g., FPGA boards, as opposed to more efficient ASICs.

Programming abstractions based on "*atomic*" primitives — A way to gain in viability (and further meet the vendors' need for closed platforms [13]) is to slightly compromise on flexibility and identify programming abstractions which

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TNSM.2019.2948467, IEEE Transactions on Network and Service Management

offer i) a set of elementary, "*atomic*", primitives, efficiently pre-implemented *once-and-for-all* in the platform, along with ii) HW-amenable ways to combine these primitives so as to deploy a desired protocol logic or network function behavior. While the majority of data plane programming abstractions proposed so far agree on the above principle, they significantly differ in i) the choice of the "*atomic*" primitives, and in ii) the way in which these primitives are combined.

For what concerns the first aspect, even if the choice of elementary building blocks is crucial, at least it does not have to be taken once and forever. A set of instructions can easily be extended as long as we can afford to deploy a new standard or product — see for instance how the actions supported by OpenFlow evolved from the original proposal [13] to the latest publicly available version 1.5.1 [14]. In this paper we are thus more concerned with the second aspect mentioned above: *how to combine elementary primitives so as to formally describe a meaningful transport layer function — or even a whole TCP-like protocol.* Here, the challenge is to devise a solution which, unlike other programmable approaches for transport layer tasks such as [15], is not bound to a SW implementation, but can be ported to CPU-less HW boards.

We argue that extended Finite State Machines (XFSMs) are a compelling answer. First, they are a *natural* way to describe a stateful process such as a transport layer protocol or task. Second, they are suited for HW offloading. The stateful process can be realized using a match/action table lookup, as will be discussed in Section III-A. This approach is able to apply the required transition in a fixed number of clock cycles, regardless of the number of states and the number of relevant transitions. Finally, and most importantly, managing timers or asynchronous events is not possible with programming languages such as P4 [1] or click-like [16] data-flow-based [17] compositions that are extensively used in companion SW platforms [18], [19] — their operation is driven by packet arrivals. Conversely, (X)FSM "events" are natively asynchronous and can be triggered by either timers or signals associated to an internal buffer or other components, including packet arrivals.

"XTRA" (XFSM for TRAnsport) contributions. — Goal of this paper is to show that state-machine-based abstractions do not restrict to packet-level on-the-fly HW-based flow processing, as in our previous work [20]–[22], but can be cast to the broader and more challenging context of transport layer functions. We here specifically contribute as follows:

- We promote XFSMs as a viable programming abstraction for transport layer functions, and we design XTRA, an operating environment able to execute a state-machine-based formalization of a desired protocol operation;
- We show how an XFSM-based design of (almost) the *entire* TCP protocol logic—not only its congestion control algorithm—can be summarized in just about 20 table lookup entries;
- We show[1] that XTRA enables portability of the same XFSM code across three extremely different platforms (OpenDataPlane SW, FPGA HW, NS3 emulator);
- Via selected use-case examples we show performance limits in the management of timers over different platforms (exponential pacing example), and we show the adaptability of our approach to also support middlebox-type transport functions (SYN-proxy example).

With respect to previous work such as HotCocoa's [24] HW-programmable congestion control, XTRA has a wider scope: we challenge programmability of a broad spectrum of transport-level functions (pacing, connection management, proxy operation, etc). XTRA can be considered as a first step in such a direction, and as such is certainly not exempt from limitations, the major one being completeness. For instance, our implementation still lacks a programmable segmentation module and a programmable "scoreboard" data structure analogous to that employed in current high-end TCP implementations managing SACKs. We also cannot claim, at this stage, support for SCTP-like multi-streaming or QUIC-like encryption, although we believe that these protocols will not break the XFSM-based programming model fostered in this paper. They will "simply" yield more complex XFSM specifications and the need to extend the XTRA API with additional actions (e.g. encryption modules).

II. RELATED WORK

Programming abstractions for network functions: Most recent work on data plane programmability has built upon P4 [1], an influential programming language introduced in 2014 for HW-based programmable switching functions. Despite the community's interest in P4, we considered it too far from our session-based programming needs and technical requirements (reliance on timers, calendar, need for asynchronous packet buffers, etc), whose support would require huge extensions to the current P4 specification [2]. Recent NFV programming models such as Netbricks [18], ClickNF [19], CLIMB [25], or tools such as eBPF [26] are closer to our transport layer needs, but their focus is on software platforms, while we also wanted to cover HW implementations with the same portable programming abstraction.

State-machine-based programming models: So far, in the field of network programmability, state machines have been proposed either as a way to model high level SDN-type control policies [27], [28], or—closer to our goals—as low-level abstractions for describing packet-level processing tasks agnostic to the underlying (HW or SW) platform [20], [29]. HW implementation of XFSMs has been carried out in our previous work FlowBlaze [22], a stateful packet processing abstraction based on XFSMs. XTRA partially leverages this previous HW implementation work, but casts it into a completely different layer—protocol implementation versus flow processing—and significantly extends the set of events and technical constructs so as to manage time-related events and relevant calendar. XTRA's API has been also radically modified to support the development of transport layer functions by adding asynchronous timers and a user level interface for buffer management, in addition to the more "usual" on-the-fly packet parsing/processing. FSMs were also considered by

---

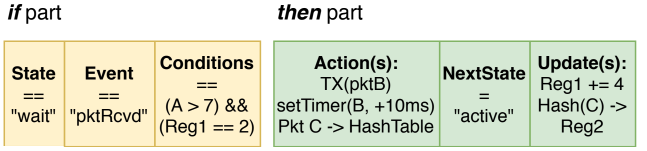
[1] XTRA portability was actually demonstrated live at ACM SIGCOMM 2018 [23], using a preliminary XTRA TCP implementation.

NetASM [30] to act as an intermediate and portable abstract representation between high level programming languages such as P4 [1] and the underlying target-specific machine code.

More flexible transport layer: The quest for flexible transport functions appears to have a twofold root. First, it is hard to dismiss the fact that, after more than 30 years since their inception, the debate on transport protocols and on the very first principles of congestion control is still open — see e.g. [31]. Second, heterogeneous networking contexts (wide area, wireless, data center, etc) and service scenarios (user-generated traffic, map-reduce-type patterns, machine-type communication, etc) hardly permit a *one-size-fits-all* approach, but rather call for *specialized* congestion control protocols tailored to specific environments [32]–[35]. Perhaps, the most significant work in terms of flexibility is the very recent proposal of a Congestion Control Plane (CCP) [15], which employs an user space program able to modify parameters such as congestion window and sending rate to program the congestion control of the underlying TCP implementation. XTRA distinguishes from CCP in two aspects: we primarily target HW offloading, opposed to CCP's focus on SW, and our programming abstraction aims to be more general, i.e. it is not specifically designed for congestion control. Transport flexibility has recently also been addressed in standard bodies: most of the work carried out in the IETF TAPS Working Group, as well as in proposals such as ModNet [36], Socket Intents [37] and NEAT [38], tackles transport layer flexibility by letting applications choose the most suitable transport service. None of them, however, render the transport protocols themselves flexible.

Portable TCP implementations: A large amount of recent work discusses how to efficiently implement transport protocols in user space [39]–[42]. Going beyond any form of hard-wiring, NetKernel realizes an idea of "network stack as a service", where the entire stack is implemented in a VM, unknown by the guest OS [43]. This could be seen as the next logical step after mechanisms such as vCC [44] or AC/DC TCP [45], which leave the guest OS unchanged yet harmonize the congestion control in data centers by manipulating the receiver window. Moving only the congestion control logic out of its usual position in the implementation is also proposed by [15], [46] in a way that is reminiscent of the Congestion Manager [47], but designed to get out of the way of the data path. While most of the existing references focus on software platforms, to the best of our knowledge, HotCocoa [24] is the first mechanism that addresses the design of hardware-aware abstractions that make it possible to implement a congestion control algorithm in programmable NICs. While this work is probably the closest to ours, it shows significant differences, both in scope (restricted to congestion control) and technical implementation choices—it uses a state machine only for credit control, whereas our proposed XFSM approach, embedding update functions and explicit computing primitives, appears more flexible and general.

Timers and pacing in TCP: Using timers to improve the performance of TCP is not a new idea: the "TCP Loss Probe" (TLP) timer was introduced in [48] to cope with tail loss, preventing an unnecessarily long wait for an RTO; several recent papers describe efficient methods for a software/hardware-based implementations of pacing [8], [49], [50], and pacing is an essential ingredient of congestion control with mechanisms such as BBR [51] (which uses FQ/pacing in Linux) or HULL [52], or to quickly start a new flow [53].



if part			***then*** part		

State == "wait"	**Event** == "pktRcvd"	**Conditions** == (A > 7) && (Reg1 == 2)	**Action(s):** TX(pktB)
			setTimer(B, +10ms)
			Pkt C -> HashTable
				NextState = "active"	**Update(s):** Reg1 += 4
					Hash(C) -> Reg2

Figure 1. Example XFSM rule entry

III. APPROACH

This paper fosters a programming model where extended Finite State Machines do govern a set of elementary building blocks specifically designed for transport layer tasks. Finite State Machines permit to *abstract* the *behavioral description* of a desired application/protocol logic, namely how a state attributed to an entity (e.g. a transport session) shall update and evolve in time, from the set of specific *events* ("input" symbols) which *cause* such an evolution, and the specific *actions* ("output" symbols) that are *triggered* by such state transitions. The neat and upfront separation between "stateful logic" and "stateless bricks" makes them an extremely flexible modeling tool, adaptable to the specific needs of a desired application's domain—it generally suffices to identify and specify the domain-specific building blocks.

More specifically, eXtended Finite State Machines (XFSM), as formally specified in [54], appear to well fit our needs. In a nutshell, XFSMs permit to transform the problem of abstracting a desired protocol operation into a list of structured "if-then" statements, , i.e. a table of rules whose hardware "execution" (perform the "then" part if the "if" part matches) can be made extremely efficient (very few clock cycles per each state transition using TCAMs and ALUs [22]).

A. The case for finite state machines

XFSMs' "if" part. As shown in the toy example of Figure 1, the "if" part of a rule evaluates three different types of information: the programmer-specified **state label** (a string), the **event** which triggers a state transition from such state, and the further **conditions** that must be verified in order to trigger a state transition. With respect to their baseline counterpart, XFSMs permit to further verify conditions on custom quantities — internal memory "registries" — before triggering a state transition. For instance, the arrival of a packet while in state "wait" is not sufficient: the programmer may further condition the transition to the verification that $A > 7$ and that the content of the programmer-defined register #1 is equal to 2.

XFSMs' "then" part. While a standard FSM just triggers a state transition–e.g. to the **next state** "active" in the example—and a Mealy machine further permits to invoke output actions, the most powerful feature of XFSMs is the ability to further initialize or update the internal memory registers via a set of "update" functions. For instance, in the example of Figure 1, the transition causes an update for register #1, and permits

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TNSM.2019.2948467, IEEE Transactions on Network and Service Management

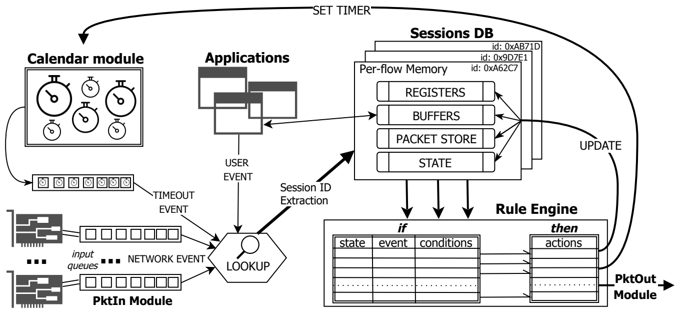


Figure 2. XTRA architecture. The lookup block receives the events from different sources (i.e. the input queues, the timers and the user applications) and extract the session ID. This Id is used to retrieve the session state stored in the per-flow memory. This information is used by the rule engine to execute the relevant actions (packet forwarding, memory update and timer scheduling).

the programmer to cache an hash fingerprint of packet C just stored into register #2. We remark that an XFSM does *not* permit to program actions or update functions: continuing the example of Figure 1, transmission of a packet, scheduling of a timer, and local storage of a packet are pre-implemented actions/functions which the programmer can “only” use according to her desired logic.

B. XTRA: XFSM execution engine

XTRA is an XFSM execution engine specialized to support the programming needs emerging in the design of transport protocols with their congestion control schemes. XTRA’s architecture, high-level sketched in Figure 2, is a computing system comprising i) a processing unit (Rule Engine) tailored to execute extended state machines, ii) the memory necessary to store registers and states for the (generally multiple) sessions supported, and iii) dedicated SW or HW modules which (pre-)implement and/or manage the domain-specific actions and events handled by the XFSM.

As shown in Figure 2, whenever a network-level or user-level event occurs, the module devised to parse such event and transform it into an internal system event identifies the session which this event belongs to (e.g. in the case of packet/ACK, the flow identifier inside the packet header). The session ID permits lookups in a session database which stores all the state information, i.e. state label and associated registries (e.g. standard or customized RTT estimate, transport session parameters, and custom registries) for the given transport session. Such information is then used by the rule engine to i) identify which rule (XFSM entry) is matched, and ii) trigger the associated actions. Finally, the session DB is updated (for the given session) with i) the next state returned by the matched rule, and with ii) the update of the registries as specified by the update field in the matched rule. Table I summarizes actions and events supported; a more extensive presentation of the XTRA API and of the actions/events parameters is provided in the repository [55]. In what follows we briefly discuss its most relevant aspects.

Events and Actions. As discussed before, an XFSM takes as input a finite set of events, and triggers a finite set of actions. Events and actions are as such exogenous, and need to be provided (i.e. pre-implemented) by the XTRA platform via dedicated stand-alone modules. XTRA events may be classified in three different categories: i) network-level events, i.e. triggered by packet/ACK arrivals, ii) user-level events triggered by the application layer, and iii) firing of timers. Similarly, actions include packet/ACK generation, transmission, scheduling of timers, and management of memory stores, as discussed in more details later on. It is worth to remark that the programmer does *not* need to know how a specific event or action is internally implemented; the list of events and actions supported by a given platform and (when applicable) the relevant parameters, is all the programmer needs.

Timers. With respect to works focusing on packet/flow processing (from OpenFlow [13] and OpenState [20] to P4 [1] and FlowBlaze [22]) where the only events handled are related to explicit packet arrivals, transport protocol mandate the explicit management of timers. Indeed, time management emerges in several tasks and at different time scales, from the loose time scale of (re)transmission timeouts (order of tens or hundreds ms) down to the tight granularity required by packet pacing over high speed links. To accommodate timers, the XTRA architecture internally implements a tailored calendar data structure providing support for a timer scheduling action as well as generation of an event towards the XTRA central processing unit when a previously scheduled timer fires. Effectiveness of timer handling of course largely differs between SW and HW (see Section V’s assessment).

Local Packet Memory (key-value store). While the session database, accessed via a session ID, permits to store and systematically manage all the state information associated to a given session, there are scenarios (see e.g. the SYN-proxy example presented in Section VII-C) in which memory is exploited for temporary storage purposes. For maximum flexibility, we provided XTRA with a local key-value store capable of accommodating incoming packets, as well as specialized actions in the API for conveniently managing access to such memory. The API actions manage the Local Packet Memory as a per-connection ID queue in which the packets can be stored and retrieved.

Extensibility. The power of the XFSM model suggests that extensions to support more complex transport protocols needs may be possible *just* via a suitable extension (and relevant implementations) of the register update capabilities, and of the set of events and actions supported, i.e. without any drastic revision of the foundational aspects of XTRA and its proposed XFSM abstraction. Indeed, it is worth to remark that, if no restrictions are imposed on the data types stored in the registries and on the update functions used for updating such registries, the XFSM model is Turing-equivalent[56]—i.e., with same data types and instruction set, anything that can be coded using a general purpose programming language can also be specified via an XFSM.

IV. IMPLEMENTATION

XTRA has been implemented on three widely different platforms: a 10G HW netFPGA SUME board, the NS3 simulator, and in User-Space over OpenDataPlane [57]. The FPGA, NS3 and ODP source code is available on the repository [55].

Action	**Explanation**
sendPacket(pktRef, iface)	send a packet *pktRef* on the specified port (*iface*)
update(op1, op2, dst, opCode)	update a *(dst)* register with an arithmetic operation *(opcode)* on two registers *(op1, op2)* or constant
store/deletePacket(pktRef)	save/clear a packet (with *pktRef* pointer) in its buffer
setField(field, pktRef, val)	set an header field (*field* enumeration) of a *pktRef* packet with a value (*val*: register or constant)
deleteInstance()	delete the session state stored in the per-flow memory of the current flow
random()	returns a random unsigned integer
closeSocket()	deallocate socket after teardown
notifyRcvd()	notify applications the reception of new packets
setTimer(data0, data1, t)	schedule a timer at a relative time *t* which carries data *(data0, data1)*
removeTimer(t)	remove the scheduled timer at relative time *t*

Event type	**Event**	**Explanation**
Timeout event	timeout(data0, data1)	timer expired carrying data *(data0, data1)*
Network event	pktRcvd(iface)	data packet arrived on the *iface*
User event	connect()	app binds the socket
	close()	app unbinds the socket
	appData(data)	new *data* from applications

Table I
XTRA API

A. HW implementation

To demonstrate hardware feasibility and identify issues and possible bottlenecks, we designed and implemented a HW proof-of-concept implementation of XTRA using the latest version of the NetFPGA board as target device.
The main FPGA implementation blocks are:

Calendar: stores events that are sent to the XFSM. A detailed description of this block is available at [9].

Packet parser: receives packets from the network, parse them and provide data to the XFSM. In particular it provides the events related to the arrival of an ACK packet, together with the relevant parameters for the XFSM such as the TCP ACK number and the Timestamp Echo Reply value used to compute the RTT.

Session database: holds the state of each TCP connection in a hash table as a $\{key,value\}$ pair. The key is the 5-tuple connection ID while the $value$ is composed by the XFSM state together with the set of registers associated to the connection.

XFSM executor: a small TCAM in which each row corresponds to a possible transition of the XFSM to execute.

Packet sender: read the data stored in the Local Packet Memory and generates the packets to transmit when the XFSM launches the *sendPacket* action. The Local Packet memory is filled by the host with the data to transmit. The host signals the presence of new data to transmit using the appData event.

The proof-of-concept exploits the standard IP blocks provided by the NetFPGA framework [58] to provide the 10GbE input/output interfaces. In particular, the packet parser is connected to an input interface and activated by any packet that arrive to the interfaces, while the packet generator is connected to the corresponding output interface and sends the TCP packets outside the NetFPGA. The packet generator is a very simple hardware block that does not present specific engineering and design issues.

The hash table for session management contains 12×32 bits registers for each flow and 128 bits to store the flow ID key, therefore each table row is composed by 64 bytes. Our proof of concept utilizes a hash table of 4K rows realized using 64 block RAMs. The hash table is realized using a 4x1 cuckoo hash table [59] similar to the one presented in [22].

The XFSM executor contains several arithmetic operations that are used to update variables like RTT and *cwnd*. While some are simple additions or multiplications, we also need divisions, i.e., to compute RTT and pacing delay. In order to limit hardware complexity and keep clock frequency at 156.25 MHz (corresponding to 10 Gbps when the data bus is 64 bits) we compute divisions A/B as products $A \times 1/B$ and we retrieve values $1/B$ from a look-up table that we pre-compute. The table is a 1024x16 bits (2KB) memory block addressed with the 10 least significant bits of the divisor B. We use similar approaches for other functions, like $\log_2{(1+1/n)}$, that we use for exponentially pacing the TCP slow start described in Section V: for this table we reserved 16KB of memory. We discuss details about tables in Section V.

The requirements of the calendar block in terms of number of timers, time resolution and complexity to insert a new timer are critical for the proof-of-concept implementation and require specific design choices. The readers interested in the details of the hardware implementation of the calendar block and on its use to provide programmable pacing can refer to [9]. In the following we simply provide a brief discussion of the main design choices. In order to achieve a good time resolution we clocked the calendar at 200 MHz, corresponding to a 5ns clock tick. The calendar uses a d-left hash table [60] to store *expiration times*: we allocate its tables in the dual port Block RAMs of the FPGA. As several MB of such memory are available, the calendar can store tens of thousands of timers. Dual port RAMs allow using one port to insert new timers and the other one to check if a timer has expired. The insertion in the calendar is done as follows: (1) the *expiration time* is computed by adding the *present time* (with 5ns resolution) to the interval time; (2) the calendar checks if a slot is available in the d-left hash table, and in that case insertion is successful; (3) otherwise, the calendar keeps increasing the *expiration time* with clock-tick steps until insertion is successful.

We highlight that adding a clock tick to solve a collision occurring in a hash table is a novel approach that can be applied in our case since the shift of a few nanoseconds when a collision occurs gives a negligible impact on the accuracy of the calendar, as it will be shown in Section V-C. Alternative (and more complex) approaches such as cuckoo tables [59] or de-amortized cuckoo hashing [61] provide an insertion time that is at best comparable to the one achievable by adding a clock tick. Moreover, standard collision resolution techniques are based on the hypothesis that multiple occurrences of keys with the same value cannot exist. Instead, for the calendar it is possible to have different timers with the same *expiration time*. Adding a clock tick also solves this "hard collision" event that cannot be managed with standard techniques. The operation of checking if a timer expires is trivial, since it only requires to check if a query to the hash table with *present time* as key provides a positive answer.

The prototype implementation uses 42895 LUTs, i.e., less than 10% of the FPGA logic resources, and 288 Block RAMs, i.e. around 18% of the total available capacity. The calendar takes 32 Block RAMs (2% of available Block RAMs), corresponding to 128 KB and can store more than 16 000 timers; the lookup tables used to offload complex operations require another 18 Block RAMs.

B. SW implementations: NS3 and ODP

We chose ns-3.27 as reference software platform for evaluating XTRA performance. We ported XTRA in the simulator by reusing only those parts of the original TCP code related to packet/header generation, packet parsing and buffer management. We extensively patched `tcp-socket-base.cc` source: there we added hooks for triggering XFSM activation as soon as transport layer events arise. We implemented the corresponding handlers in `tcp-xfsm.cc` that encodes the complete logic behind the XFSM abstraction layer.

Tests in ns-3 were conducted on a dumbbell topology with configurable parameters, they are available in [55].

Besides NS3, we implemented XTRA on top of the high performing OpenDataPlane [57] network stack, that makes coding and debugging faster than with a kernel based approach. For our proof-of-concept we did not exploit all the functionalities available in ODP, in particular those that target system optimization. For this reason our demonstrator runs in a single thread that transforms received packets and expired timers into events: on their occurrence, the same thread parses the XFSM and evaluates the conditions and executes the actions associated with determined state transitions. We use an `odp_pktio` object attached to a configured network interface for receiving and transmitting TCP segments that are respectively fed to and obtained from the XFSM description, and an `odp_pool` of timeouts for arming timers that we schedule inside an `odp_queue`. Timers are kept ordered in a circular array whose tail advances as the XFSM arms new ones. To support multiple flows, we used a hash table to store the states associated to an instance. The key used is configurable as an array of header fields (e.g. the 5-tuple IP src, IP dst, IP proto, src port and dst port). When an event is triggered, it contains the flow key that is used to retrieve the context. If there isn't any matching instance, a new one is created and inserted to the hash table.

C. The XFSM-Lang

In order to describe the state machines, we developed a Domain Specific Language (DSL) named *XFSM-Lang* that provides high level abstraction with a clear notion of state. We also developed a *compiler* for translating a program written in *XFSM-Lang* into platform independent JSON code. The same JSON code, that includes all conditions, states, table entries and registers describing the XFSM, can be uploaded to the different platforms and directly executed without any need to change it.

We opted for developing a new language rather than using an existing one for two main reasons. First, even though several programming languages easily handle events, none of them can be used for controlling network related events: i.e., they are usually focused on the development of user interfaces, or tailored to both describe hardware blocks (VHDL/Verilog are event driven languages) and check their responsiveness [62]. Second, languages that can be used for describing FSM are either too domain specific, like in the case of Esterel [63], or not completely developed, so that they lack many of the features that are required in our case.

In the end, we develop *XFSM-Lang* from scratch with the following features:

- it provides the primitives needed for network processing, such as packet header definition, packet manipulation;
- it is able to describe an XFSM;
- it is able to support per-flow state management;
- it is able to describe how the system reacts to different events (network events, timers, application requests etc).

A program in *XFSM-Lang* begins with register declarations and macro action definitions. A macro action is a set of primitives and other macro actions, and can be used to reduce redundancy in the code. For instance in the proxy code (Figure 13), the macro `reply` is used in the syn-proxy in order to reply to a packet, switching ports and addresses. After registers and macro definition the flow ID definition is provided. This ID is used to associate the incoming packets to different flow, thus allowing the per-flow state management. For example, in the case of a TCP connection the flow ID corresponds to the 5-tuple and a different XFSM is instantiated for each different flow ID. After the program lists all the states of the XFSM, with events and conditions considered for every state.

For each state we can define some actions to be executed every time the XFSM comes into that state, and we can define events. For every event, we can have actions executed when the event happens. Inside an event we can define some conditions which when are satisfied, cause the actions inside the condition block to be executed. As an example, Figure 3 shows the `slowStart` state definition in the TB-TCP implementation. Inside the state block, we have two `on` blocks that are used to define the XFSM behavior when a `timeout` (line 2) or a `pcktRcvd` (line 15) event happens in the `slowStart` state. Inside the two `on` blocks, `if` blocks define a set of conditions that have to be verified (line 3 and 9 for the `timeout` event and line 16 for the condition associated to the `pktRcvd` event). Inside the `if` blocks we have some actions, like a setTimer (to schedule a timer, line 6 and 19), some updates (e.g. line 5, 11, 17 or 20) and macro action calls (e.g. line 4, 7, 12 or 21).

Associated with an event, the programmer can access to the informations carried by the event itself, like for instance `timeout.data0`, which returns the data-field `0` carried by the timer, or in the case of a packet received event, `pktRcvd.port` returns the port where the packet has been received. The programmer can also access to the field of the packet, using the dotted notation protocol dot field name, e.g. `tcp.ackNo`, `ip.src` o `udp.sport`.

D. Debug interfaces

The *XFSM-Lang* provides a set of features that facilitate functional debugging. The main target of these features is to expose the internal state of the XFSM to the programmer. We developed three types of debugging interfaces.

1) *Event based debug interface* - we introduce a specific "debug event" that generates a snapshot of the flow registers associated to the flow ID under execution plus the current timestamp. The NetFGPA prototype transmits the snapshot to the host via the PCIe interface, embedded in a special packet. The debug event, that is similar to a state transition (i.e., it is activated from a specific state when a certain condition occurs) is defined by the *XFSM-Lang* using the `#debug` directive. If placed before a state declaration (e.g. before line 1 in Figure 3), it fires every time the XFSM goes into that specific state independently of events and conditions. If placed before an event (e.g. before line 2), it fires only if that event occurs in the state that defines the event block itself (in this case `slowStart`). Finally, if placed before a condition statement (e.g. before line 3), it fires only on these specific conditions and when the event on which these conditions are associated occurs.
2) *Sampling based debug interface* - the XFSM execution engine can automatically generate a snapshot of specific flow registers with configurable frequency. In *XFSM-Lang* this is obtained by placing debug directives at the beginning of an XL program, with syntax `#debug reg interval`, where `reg` is the register to sample and `interval` is the time period in microsecond. The NetFPGA prototype implements this debug mechanism by using an internal timer: when it expires, the FPGA generates a special packet with the requested information similarly to the previous debug interface.
3) *Flow context table inspection/manipulation* - the flow table can be accessed from the external to monitor the state of the XFSM engine, to count how many flows are active, to query the state of specific flows, etc. Using this mechanism, we can also modify the flow table.

V. PLATFORMS' EVALUATION

Our current HW NetFPGA prototype, mainly meant to prove the feasibility of our concept, is still driver- and memory-limited and cannot yet support a large number of flows. Nevertheless, it is fully functional at wire speed (once a content file is preloaded in the board), and can be used to gather a number of interesting performance insights in relation to latency and timing accuracy improvements with respect to its ODP SW counterpart.

```
1  State slowStart {
2    on(timeout) {
3      if (timeout.data1 <= 0, availWin > 0) {
4        sendAndUpdateWithTimeout();
5        highTxMark = nextTxSeq + 1448;
6        setTimer(nextTxSeq, 0, 1);
7        updateAvailWin();
8      }
9      if (timeout.data1 > 0, timeout.data0 >= lastAckedSeq,
10         currentRetxRound < timeout.data1) {
11       currentRetxRound = currentRetxRound + 1;
12       initPacing();
13     }
14   }
15   on(pktRcvd) {
16     if (tcp.flags > 2, lastAckedSeq != tcp.ackNo) {
17       cwnd = cwnd + 1448;
18       lastAckedSeq = tcp.ackNo;
19       setTimer(nextTxSeq, 0, 1);
20       rtt = max(rtt, tcp.timestampEchoReply);
21       updateAvailWin();
22     }
23   }
24 }
```

Figure 3. An example of state definition in *XFSM-Lang*

A. Latency assessment: HW vs SW

A major reason to move from SW to HW is to more efficiently support "thin stream" [64] applications with very stringent temporal requirements. Indeed, applications such as Voice-over-IP, multiplayer games, remote control systems, or online trading usually are not bandwidth-hungry (though in some cases they also might be, e.g. in some datacenter applications or 5G scenarios). Rather, they usually communicate via short sized packets, which have critical latency requirements not only in absolute terms, but also in terms of variability, worst-case (tail-latency) requirements, and sensitivity of latency to the processing load [65].

In the experimental setup we forced XTRA to execute a simple stop-and-wait ARQ in which after the connection establishment the sender waits for an ACK before sending another packet. The latency is measured as the time difference between the arrival of one ACK and the sending of the next packet. The measurement of the latency was done with a NetFPGA configured to monitor the traffic between the sender (implementing the XTRA machine) and a receiver and to copy the monitored packet to a mirror port adding a $6.4ns$ resolution timestamp. The use of the NetFPGA allows a very efficient and reliable tool for latency measurement since avoids the typical error sources of a software measurements (NIC latency, uncontrolled offloading features, unexpected interrupts etc) that can affect the measure.

The measures have been performed with the SW (ODP) and HW (NetFPGA) prototypes executing the same stop-and-wait ARQ application and changing the number of active connections from 1 up to 100 connections. In particular for each experiment we logged 10000 latency samples and we computed the average latency and the 99th and the 99.9th percentile. The collected results are shown in Figure 4. Since the results with the FPGA are insensitive to the number of active flows, for this implementation we only report the case of 100 connections. In fact, for the software implementation the time needed to retrieve the flow context can vary depending if the data are already in the cache or must be retrieved from the main memory. Instead, the FPGA uses a plain memory structure and the access time to the hash table storing the flow context is fixed.

From the Figure it is possible to see that the SW not only has a higher latency than the FPGA implementation, but its variability is extremely high. In particular, the software implementation has an average latency between 13-15μs and increase slightly with the number of active flow, while the FPGA is constantly at 4μs of average latency. The variability of the latency for the software is significant, since the 99.9th percentile go up to 190μs with 100 active connection. On the other hand, the FPGA implementation delivers the packet with a latency that is 5μs in the worst case.

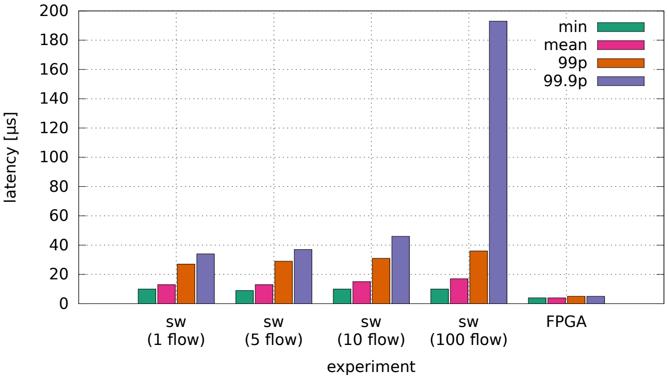


Figure 4. Latency comparison with ODP and FPGA, changing the number of active flows.

B. HW flow scalability

The current implementation of the XTRA prototype uses the internal block RAMs of the NetFPGA to store the flow context for each active connection. The current prototype is limited to 4K rows and use around 5% of the internal FPGA memory resources. Since the internal RAMs are used also for other purposes (e.g to implement the internal tx/rx queues) it is not possible to reserve more than 50% of Block RAMs to allocate the flow context. This limits the scalability of the number of flows using the internal BRAM to a maximum of 40K concurrent flows. However, the NetFPGA, as almost all the available FPGA based SmartNICs, are equipped with several external memories that can be used to scale the number of flows that the system is able to manage. In particular, the NetFPGA is equipped with two kind of external memories: i) the QDR-II memories able to provide an additional 9 MB of storage and ii) 8GB of DRAM memories. Since DRAM memories have access times in the order of tens of clock cycles, the use of these memories for extending the number of active flows is quite challenging and the actual performances are deeply dependent on the traffic distribution. In fact, using the DRAM as the main memory for storing the flow context and the internal FPGA memory as a fast access cache memory is effective only if most of the traffic is due to a limited number of flows, which can be stored in the BRAM, limiting the number of access to the external memories. Instead, the use of QDR is much more efficient, since this memory can be accessed in just one clock cycle (when the clock frequency is less than 167 MHz) and it is able to scale up to 150K flows.

C. Timing flexibility and accuracy

Another reason which calls for HW offloading of transport layer functions is accurate management of timing. Besides the ability to accurately control transmission schedules (e.g. in terms of pacing requirements mandated by many recent congestion control mechanisms [35], [51]–[53]), a fine-grained SW implementation (e.g., based on spin locks) may incur in severe CPU resource consumption. Improvements such as relying on the Linux kernel's FQ scheduler (considered for BBR [51]), or careful designs such as Carousel [8] may come along with limited flexibility, and cannot provide support beyond constant (or, at best, piecewise-constant) schedules.

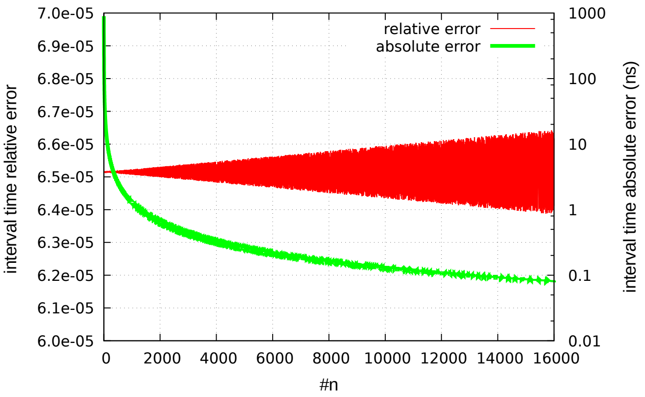


Figure 5. HW Timer accuracy (exponential pacing): absolute (right y-axis, nanosecond) and relative (left y-axis) errors versus packet #.

Conversely, the ability of a state-machine-based specification to natively handle arbitrarily timed events directly in the HW permits to trivially incorporate management of transmission times inside the protocol implementation itself. To prove this point, we show how simple the deployment of an uneven pacing strategy would become with XTRA.

VI. RESULTS

We report in this Section results we collected to better evaluate the timer-scheduling complexity in both the NetFPGA HW prototype and the ODP software-based implementation. **Insertion time and accuracy of hardware calendar.** To test the effects of the insertion algorithm of the hardware calendar we compared both the average insertion time and the worst case insertion time of a standard cuckoo hash with those achievable with our algorithm. For this experiment we simulated a hash structure composed of 4 tables of 4096 rows which first is loaded to a specific load value, and then a dynamic insertion/remove procedure is applied. The procedure substitutes each expired timer with a new one maintaining the same load factor for each insertion. This substitution has been applied 1000 times for each test. The comparison with the insertion time of the standard cuckoo algorithm is reported in Figure 6. For all the graphs the x-axis reports the load factor at which the hash table is loaded before starting the insertion and removal procedure. The plot in Figure 6.a) reports the average insertion time for the standard and for the calendar insertion algorithms, while the plot in Figure 6.b) reports the worst case insertion time for the two algorithms. As expected, the calendar insertion is always faster than the standard one, both on average and in the worst case. For this last parameter, it is worth to notice that the overall worst case, which occurs when the table is fully loaded (97%), is only of 71 clock cycles, corresponding to 355 ns. As expected, the time resolution of the calendar is much better than the target resolution of 1 μs.

Exponential Pacing: Hardware Assessment. For our assessment we borrowed the "exponential pacing" proposed in [9]. A "paced Slow Start" can be implemented by ensuring that the interval of time between transmission of segment n and segment $n+1$ is $\log_2\left(1+\frac{1}{n}\right) RTT$. We have verified the pacing accuracy of the HW prototype, by uploading and executing a single state / single event (one line) XFSM

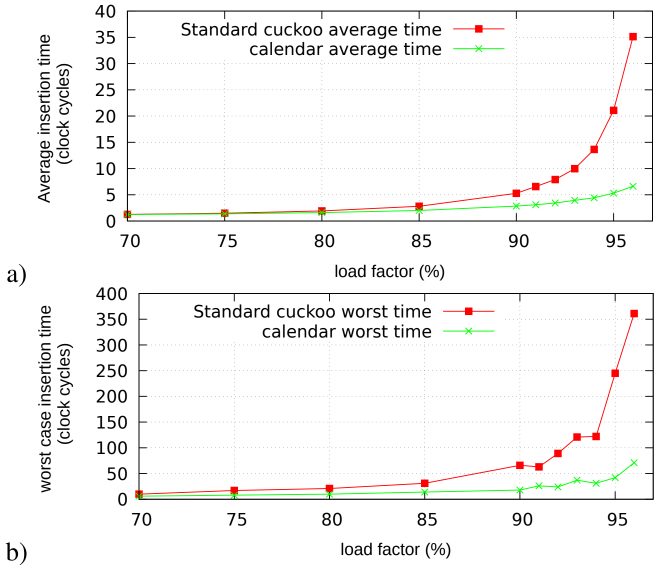


Figure 6. Comparison between average insertion time of standard cuckoo and calendar. a) the plot reports the average insertion time. b) the plot the worst case insertion time.

implementing the above rule with a constant RTT. As already mentioned in Section IV-A we extended the update functions with a precomputed 16K entries' lookup table which takes n as input and provides a 32-bit fixed point representation of $\log_2\left(1+\frac{1}{n}\right)$, so as to support time scales ranging from about 1 μs—roughly the time that saturates a 10Gbps link with 1500-byte packets— to several milliseconds. Figure 5 shows that such time scales are not nearly challenging for a HW implementation, with an absolute error with respect to the theoretical computation even dropping below the nanosecond, and a relative error always in the order of $6 \cdot 10^{-5}$ (it grows just because the scheduled time intervals get smaller as the packet index increases).

Software-based Timers. Albeit quite obvious, HW timing accuracy is best appreciated when compared with its ODP software counterpart. Here, we assumed the most challenging scenario where a node powered by an Intel Core-i7 CPU clocked at 4.2GHz is pacing 1500B long TCP segments over an Intel X520 10Gb/s NIC. We report in Figure 7 the timers' accuracy evaluated with the help of an Anritsu MD1230B Data Quality Analyzer for exponentially decreasing scheduling delays. The central mark is the median, the box includes data between the 25th and 75th percentiles, and whiskers exclude those points considered as outliers that are plotted individually. As ODP could not schedule reliably segments with the minimum pacing delay permitted by the channel, i.e., $1.22\mu s$, we excluded it in the top diagram that reports the relative error achievable by ODP. In the bottom diagram, instead, we report the absolute error that we achieved after replacing the ODP timers with spinlocks that check the realtime clock of the system. While not limiting the portability that we envision (in the end we used the POSIX function `clock_gettime`), this modification brings in two positive effects: it enables minimum pacing delays and decreases CPU load from 160% when using ODP to 110%. We can appreciate a median error very close to zero for all explored delays.

VII. APPLICATION EXAMPLES

To show the breadth of XTRA's applicability, we first implemented a version of TCP that performs comparable to up-to-date Linux and FreeBSD implementations and is a basic and simplified yet "complete" implementation of the protocol (complete enough to correctly communicate as a sender with ncat and iperf receivers). Then, to show that we are not limited to TCP, we implemented a version of LEDBAT congestion control [66] over UDP. To understand whether our code is expressive enough to cover an even broader range of congestion controls, we ported one of the mechanisms from the most closely related work—TIMELY from HotCocoa [24]—to XTRA, and found that we can do this without problems and with a similar number of code lines. Finally, we implemented a simple SYN Proxy, which implements a stateful middlebox operation on a switch. We discuss all of these implementations in the following.

A. TB-TCP: TCP with Timer-Based Loss Recovery

TCP's connection (LISTEN, SYN-SENT, etc.) and congestion control (Slow Start, Congestion Avoidance, Fast Retransmit / Fast Recovery) state machines can be readily transformed into an XFSM. The Fast Recovery phase itself, however, is exceedingly complex: over the years, the recommended standard behavior in this phase [67] has been extended and updated in many ways, addressing issues such as wrongly reacting to packet reordering [68], creating unnecessary bursts ("Proportional Rate Reduction" (PRR) [69]), or timeouts after losses at the end of a burst ("tail loss") [70]. Generally, these algorithms are sets of rules operating on a data structure called the "scoreboard". While it is in principle no problem to transfer them to an XFSM, we found this added complexity to be unnecessary thanks to one of the most recent additions to the Linux code, RACK ("Recent ACKnowledgment") [71].

RACK significantly changes the underlying logic of TCP's loss recovery: it detects losses and decides to retransmit segments using *time* instead of counting segments or byte sequences. When an ACK arrives that acknowledges some but not all transmitted segments, any segments that were sent earlier (by a minimum time called "reordering window") than the acknowledged segments are assumed to be lost. This way RACK can detect more losses than other algorithms, and it can effectively replace all of them with its straightforward logic, rendering the entire recovery phase much simpler.

RACK is now deployed and enabled by default in Linux [71]. While it relies on (SACK) information from DupACKs, it seems obvious that the idea of considering the amount of time that has expired since a segment was sent could also be implemented more directly, by using a timer for every transmitted segment. Indeed, the authors of RACK state that it "conceptually arms a (virtual) timer on every packet sent" [72]. Because per-packet timers are affordable in our implementation, we implemented a strictly timer-based variant of RACK that we call *Timer-Based TCP (TB-TCP)*. Our goal

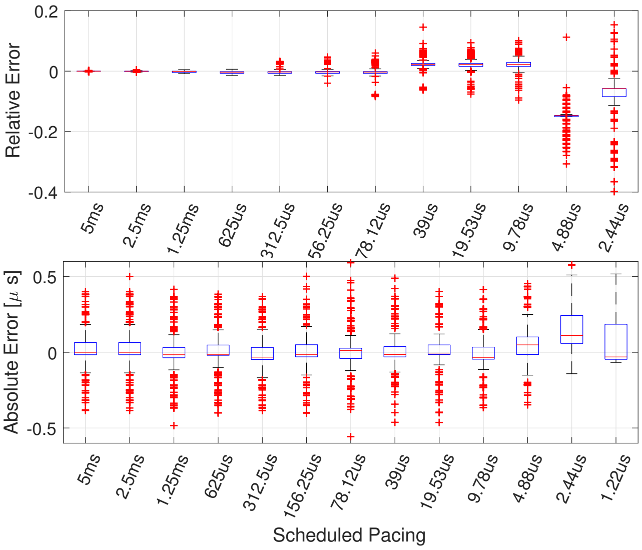


Figure 7. Scheduling error achievable with the software implementation: top, relative error with ODP; bottom, absolute error with spin lock.

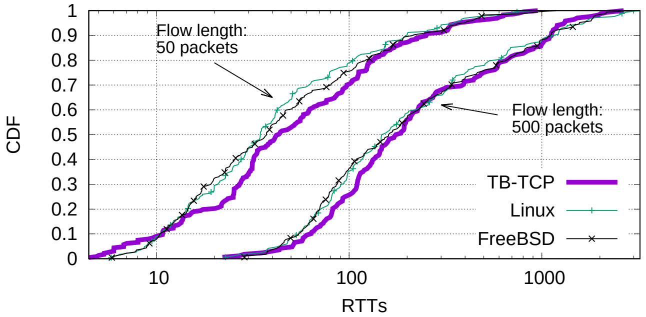


Figure 8. Flow Completion Time (FCT) in RTTs of the slowest of six flows

for TB-TCP was to be as simple as possible while operating roughly as good as a standard TCP implementation.

Standard loss recovery methods seek to avoid large bursts and ensure that the correct number of segments are kept in flight by maintaining an "ACK clock". The need for an "ACK clock" is obviated by the ability to pace every transmission in a strict timer-based implementation, causing some subtle changes in the RACK logic. For example, because pacing is not assumed in the normal TCP standard, RACK needs a different way to limit the number of segments that are immediately transmitted (as a burst) when it learns that many packets were dropped. Hence, it is recommended in [72] to combine RACK with PRR, which uses Slow Start as a way to quickly clock out packets while limiting burstiness to some degree. For TB-TCP, we can do this differently.

Our algorithm works as follows:[2]

1) Whenever a new segment is transmitted, a timer is armed with 2×RTT (from the most recent RTT sample). This is similar to RACK's "Tail Loss Probe" (TLP) timer.
2) When a timer fires, TCP enters the recovery period. We backup $ssthresh$ and $cwnd$ (as $ssthresh\_prev$ and $cwnd\_prev$, respectively) and then reduce their values following the congestion control algorithm. We also remember the state (Congestion Avoidance or Slow Start). Then, we calculate the minimum pacing delay as $RTT/ssthresh$, where we use the most recent RTT sample. Similarly to PRR, this limits the sending rate to $ssthresh/ssthresh\_prev$ times the rate before recovery.
3) For each firing timer, a segment is retransmitted. In doing so, it is ensured that the minimum pacing delay is kept, by delaying transmissions if necessary.
4) Generally, whenever an ACK arrives, any outstanding timers of segments that are acknowledged (either cumulatively or via SACK) are cancelled. If we are in recovery, a "full ACK" (i.e., covering all the data that was outstanding when loss was detected) ends the recovery period.

A full ACK informs us that all segments have been successfully retransmitted, and there are no more segments in flight; yet, we have a $cwnd$ worth of segments to send at the end of recovery, and there will be no more incoming ACKs that allow us to send them. Therefore, to gracefully "hand over" to normal ACK-clocked TCP behavior, we need to pace these segments too, in a phase that we call "Post Recovery". We calculate the pacing delay as $pktsToPace/RTT$, where $pktsToPace$ is initialized with $cwnd\_prev$ and RTT is the most recent RTT sample.[3] From now on, segments are paced with this rate until the available window becomes zero; at this point, TB-TCP "hands over" to Congestion Avoidance. To ensure that the available window becomes zero at some point, we gradually increase our rate in this phase whenever we notice that the available window has increased, by adding 1 to $pktsToPace$. This is more conservative than standard implementations, which typically use Slow Start to send out all the allowed segments before Fast Recovery terminates.

As with normal RACK, reacting to lost retransmits is easy with TB-TCP: with every scheduled timer, we also store a "transmission round", which initially is 1. Then, in step 2 above, when the first timer fires, we compare this transmission round with a global transmission round variable (initially 0), and reduce *ssthresh* and *cwnd* when the timer's round is greater than the global one. In doing so, we also increase the global round. Because this exponentially decreases the sending rate every 2 RTTs when there is congestion (resembling standard TCP's reaction to ECN-Echo), the effect quickly becomes similar to a Retransmission Timeout (RTO). We therefore see no need to additionally calculate an RTO and return to Slow Start in an ongoing connection. Surely, situations can be constructed where an RTO with Slow Start would either be better or worse than our approach; given our goal to be simple, introducing Slow Start's bursty and aggressive increase after congestion does not seem to be better justified than using Congestion Avoidance after an exponential back-off, and so we omit it from our design.

To better understand the practical impact of our changes to RACK, we compared TB-TCP against the Linux and FreeBSD

---

[2] To simplify this description, $ssthresh$ and $cwnd$ are assumed to be in segments, not bytes. In reality, our code divides them by the segment size for all pacing calculations.

[3] We use $cwnd\_prev$ because it gives us the ACK-clocked sending rate, yet the total number of segments that we will send is less than $cwnd\_prev$. The *rate* before congestion may have been fine, but the total number of segments that were sent before waiting for an ACK was too large.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TNSM.2019.2948467, IEEE Transactions on Network and Service Management

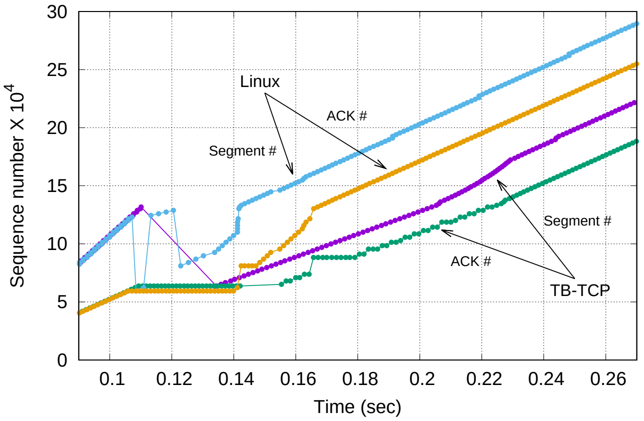


Figure 9. Time-sequence diagram of TB-TCP and Linux loss recovery at the end of Slow Start. The bottleneck capacity was 10 Mbit/s, the configured RTT was 20 ms and the bottleneck queue had a buffer of 10 packets

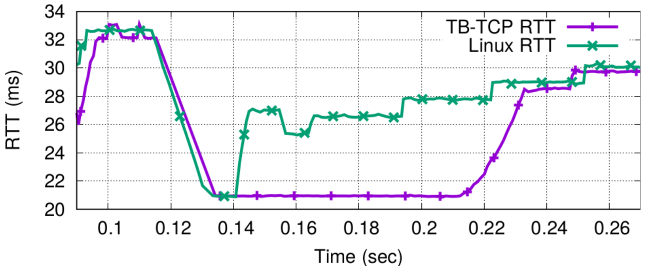


Figure 10. RTT in the same scenario as in Figure 9

kernel TCP implementations (v4.10 and v11.0, respectively) in a local single-bottleneck testbed, using ns-3's emulation facility to send real traffic to an *ncat* receiver. We configured "newreno" congestion control everywhere. Also, because ns-3 does not implement Appropriate Byte Counting (ABC) [73], it would increase *cwnd* slower than the other implementations when Delayed ACKs are used, which we therefore disabled.

In the investigated scenario, six flows start at the same time. We consider the Flow Completion Time (FCT)—which we define as the time between transmission of the first data packet and the time of arrival of the last ACK—of the slowest flow. Figure 8 shows the distribution of FCTs as the number of RTTs, obtained by varying the bottleneck capacity (1-5, 10 and 15 Mbit/s; this covers 72% of worldwide access links in Q1 2017 [74]), the RTT (10, 20, 40, 80, 100, 150, 200 ms; this covers more than 80% of Internet base RTTs [75]) and the length of the bottleneck queue (5, 10, 50, 100, 200 packets). After filtering out some cases where flows did not reach the considered length during the measurement period, we were left with 168 and 227 experiments for the flow length=50 and 500 packets cases, respectively.

We compare TB-TCP with Linux TCP in Figure 9. We collected the time-sequence data by connecting TB-TCP to a ncat receiver, while we used iperf for Linux TCP case.

We can see how TB-TCP reacts slightly later than Linux TCP because it ignores the first DupACKs and waits for a $2 \times RTT$ timer to fire before entering loss recovery. Then it starts retransmitting lost segments until the end of recovery is reached at around 0.2 seconds. From then on, TB-TCP is in Post Recovery; notice how the rate grows (the angle of the TB-TCP line slightly increases) until about 0.23 seconds, which is when we hand back to Congestion Avoidance. In case of Linux, after sending two more segments when the ACK line becomes flat (Limited Transmit [8]), the RACK algorithm decides that two segments were lost and should be retransmitted. Next, cwnd and the "pipe" variable allow the sender to transmit four new segments; it is not yet clear to the sender whether more packets were lost. Around 0.12 seconds, a new SACK block arrives—RACK uses this information to decide that all segments that were sent at least a "reordering window" earlier were lost and retransmits them. The benefit of sending out packets early and keeping the recovery phase short is visible in Figure 10: the Linux line is generally above the TB-TCP line, which means that it has come further ahead in the data stream and can finish the transfer earlier. However, as Linux finishes the recovery phase after 0.14 seconds, it transmits a burst (note the steep angle of the sequence line here). Figure 10 shows the impact of this burst: the delay grows, and it has a higher starting point for growing further every RTT as Congestion Avoidance increases the cwnd. TB-TCP reaches a similar delay later, when it terminates its Post Recovery at around 0.25 seconds.

The time-sequence diagram in Figure 11 visualizes just how close the behavior of the three implementations is. This test was carried out using a single sender and receiver, interconnected with a physical link and using linux Traffic Control (TC) *netem* module. The parameters were: capacity 6 Mbps, queue length 20 packets and one-way delay 10 ms. At around t=0.25, the diagram shows the characteristic "Post Recovery" part of TB-TCP in which we gradually increase the pacing rate until we can finally hand over to Congestion Avoidance.

We can also see minor deviations between the three implementations, which are caused by small timing differences in how packets are clocked out. For instance, ns-3 is slightly slower at transmitting packets using emulation. The hardware implementation appears to be a little faster, which is due to the lower precision for divisions.

We were surprised to find that our simplified timer-based implementation was able to compete quite well with the state-of-the-art Linux and FreeBSD TCP implementations and in some cases even surpass them. This is evident in the 500-packet case towards the tail of the distribution, and with quickly terminating 50-packet flows (these were large Bandwidth×Delay Product (BDP) cases, where packet loss was more severe than when the BDP was smaller).

The complete TB-TCP XFSM table is available in our public repository [55]: lines 9+ implement congestion control. Obviously, at this length, this implementation lacks some features (e.g., the sending application is assumed to be greedy, all segments have the same size, there is no Path MTU Discovery, no ECN, ...). However, we stress that, with only packet parsing and header generation (with checksum offloading) being hardcoded inside XTRA, this implementation is already a quite "complete" TCP, as it was able to communicate as a sender with real ncat and iperf receivers in our tests.

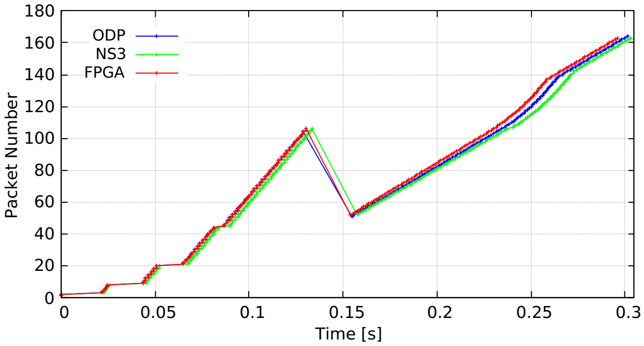


Figure 11. Time-sequence diagram of a TB-TCP flow beginning in Slow Start, then losing a packet, and concluding with our timer-based Fast Recovery phase.

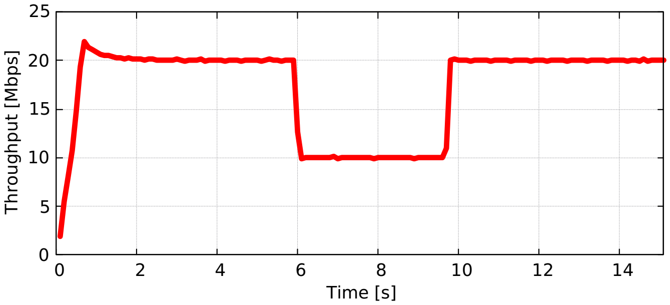


Figure 12. LEDBAT throughput measurement

B. LEDBAT over UDP

Our LEDBAT implementation follows the basic "overview" algorithm in Section 2.4.1 of RFC 6817 [66], without considering Slow Start. Similar to BitTorrent, we used an unreliable implementation over UDP, adding sequence numbers and a timestamp in the first 8 bytes of the payload. Instead of the One-Way Delay (OWD) recommended in [66], we used the Round-Trip Time calculated from a timestamp echo-reply similar to TCP to determine the target queuing delay. Our XL implementation of LEDBAT had 54 lines of code and only two states (including the initial state).

To evaluate our LEDBAT implementation on XTRA we used the OpenDataPlane platform in a Linux machine. We connected two interfaces in loopback and used the Linux Traffic Control (TC) netem module with the following parameters: one-way delay of 25 milliseconds and 20Mbps of capacity. The receiver was a simple server implemented in python which solely responded to incoming udp packets by echoing an UCP "acknowledgment" with a 32 bit sequence number and a 32 bit timestamp echo-reply. To emulate newly arriving foreground traffic that LEDBAT should react to, the link capacity was reduced to 10Mbps at 6 seconds and reset to 20Mbps around 10 seconds. Figure 12 shows how LEDBAT correctly reacts; after the capacity drop, at around 10 seconds it quickly increases its rate up to the capacity limit.

C. A SYN proxy

We believe that a compelling application of XTRA is the ability to offload, with no coding effort whatsoever, transport functions from software servers to HW NICs. Although simplistic, the following SYN-proxy application is sufficiently elaborate to challenge the usability of XTRA also inside intermediary agents. A well known way to defend against SYN-flooding attacks consists in deploying a dedicated agent which replies with a SYNACK on behalf of the backend server. Only when this handshake is completed (i.e. a remote client is not spoofed but exists), the agent starts a new connection setup towards the main server for the client, but (unlike an ordinary proxy) replaying the real client's initial SYN (duly stored for the purpose) and, later on, the real client's ACK as well. From now on, communication between the client and server is "direct" (no need for the agent to act as a real proxy). However there is a caveat: since the client expects sequence numbers starting from the one that the proxy originally wrote into the SYN/ACK message, the data packets from the server would not be acceptable to the client. Thus, the agent must continuously perform a NAT-like translation of sequence and acknowledge numbers until the connection is closed—a fast-path processing task which would clearly benefit from HW offloading. Figure 13 shows that an *XFSM-Lang* description of the above function reduces to few rules, which could be *entirely* offloaded to a HW NIC supporting XTRA!

The proxy code is also available from the repository, along with a pcap trace file showing its operation (a TCP client connecting to a server via the proxy).

D. Flexibility assessment: HotCocoa

The most closely related work, HotCocoa [24], offers programming abstractions that allow to implement a congestion control fully in hardware. The authors of [24] show that their language is very expressive by porting a number of congestion controls to their abstraction. Since we claim that our XFSM approach is broader, it can be regarded as a superset of HotCocoa. To support this claim, we converted the "TIMELY" congestion control from HotCocoa code to XL.

In Table 1 of [24], TIMELY is described to have 60 lines of code in total. The full code available from the github repository referenced in the paper contains 83 lines of code, probably from counting static blocks like function signatures and such. After converting TIMELY to an XFSM, we ended up with 68 lines of code, putting us in the same rough ballpark. The reason why XTRA uses slightly more lines of code than HotCocoa primarily stems in the fact that XTRA, at least at this stage, does not yet provide any dedicated data structure for congestion control. We remark that our reduced specialization is in line with our goal of covering the programmability of the whole set of transport layer tasks, opposed to HotCocoa's focus on "just" its congestion control part. Indeed, we demonstrated the feasibility of applications such as LEDBAT or a SYN Proxy, which do not seem easily implementable on HotCocoa, owing to its focus on congestion control only.

VIII. CONCLUSIONS

The main conclusive message of this paper is that seamless *portability* of suitably coded transport level functions across widely heterogeneous SW *and* HW platforms is viable, and

```
Register temp, diff;
Action reply {
 //switch ports
 temp = tcp.sport;
 setField(tcp.sport, CURR_PKT,
          tcp.dport);tc
 setField(tcp.dport, CURR_PKT, temp);
 //switch IPs
 temp = ipv4.src;
 setField(ipv4.src, CURR_PKT,
          ipv4.dst);
 setField(ipv4.dst, CURR_PKT, temp);
 //switch MAC addresses
 temp = eth.src;
 setField(eth.src, CURR_PKT, eth.dst);
 setField(eth.dst, CURR_PKT, temp);
}
Action set_ackno {
 temp = tcp.seqNo + 1;
 setField(tcp.ackNo, CURR_PKT, temp);
}
State initial init {
 on (pktRcvd) {
 //Syn from the external iface
  if (tcp.flags == 2,
      pktRcvd.port == 1) {
   storePacket(CURR_PKT);
   reply();
   set_ackno();
   diff = random();
   setField(tcp.seqNo, CURR_PKT,
            diff);
   //Setting the SynAck flag
   setField(tcp.flags, CURR_PKT, 18);
   sendPacket(CURR_PKT, 1);
   setTimer(1, 500000000, 1);
   setNextState(syn_rcvd);
  }
 }
}
State syn_rcvd {
 on (pktRcvd) {
  //Ack from the external iface
  if (tcp.flags == 16,
      pktRcvd.port == 1) {
   sendPacket(CURR_PKT, 2);
   setNextState(syn_sent_to_server);
  }
 }
 on (timeout) {
  //No answer,delete the XFSM instance
  deleteInstance();
 }
}
State syn_sent_to_server {
 on (pktRcvd) {
  if (tcp.flags == 18,
      pktRcvd.port == 2) {
   //synAck from the internal iface
   reply();
   diff = diff − tcp.seqNo;
   temp = tcp.ackNo;
   set_ackno();
   setField(tcp.seqNo, CURR_PKT,
            temp);
   setField(tcp.flags, CURR_PKT,
            16);
   sendPacket(CURR_PKT, 2);
   setNextState(forward);
  }
 }
}
State forward {
 on (pktRcvd) {
  //ACK from the external iface
  if (tcp.flags == 16,
      pktRcvd.port == 1) {
   temp = tcp.ackNo − diff;
   setField(tcp.ackNo, CURR_PKT,
            temp);
   sendPacket(CURR_PKT, 2);
  }
  //ACK from the internal iface
  if (tcp.flags == 16,
      pktRcvd.port == 2) {
   temp = tcp.seqNo + diff;
   setField(tcp.seqNo, CURR_PKT,
            temp);
   sendPacket(CURR_PKT, 1);
  }
 }
}
```

Figure 13. SYN Proxy Server in *XFSM-Lang*

may pave the road towards novel forms of high-performance HW-native network functions' design and management.

As we have shown in the paper, an XTRA-coded abstract TCP specification can run on three different platforms, both in software and in hardware, without changing a single line of code. As an additional "side" contribution, our TB-TCP shows that a purely timer-based variant of RACK can massively simplify the most complex part of TCP without a significant cost in performance. In fact, given the extreme simplicity of our design, we wonder if a slightly extended TB-TCP might generally outperform other TCP implementations. This is one of various future research avenues that we plan to explore.

Our implementations of XTRA and TB-TCP are far from production-ready, but we believe that our prototypes convincingly show the feasibility of our idea, to implement transport functions (or even entire protocols) *once*, in *one* appropriate representation and run them *everywhere*. If end-device OSs, smart NICs and Middleboxes were offering the XTRA API, operators would become able to manage and deploy optimized/tailored transport functions/protocols for each scenario on the fly, and applications could ship with their own transports which get dynamically replaced as needed.

Of course, there may be dangers to such ease of programmability; today, congestion control is designed based on agreed-upon general rules, to avoid a global congestion collapse. Could the ability for anyone to easily change the behavior endanger the stability of the Internet? Then again, what protects the Internet today from potentially harmful changes in open Operating Systems such as Linux and FreeBSD is only the complexity of the code base. Given that "security by obscurity" has never been a winning principle, we are convinced that offering these tools is the right thing to do.

IX. ACKNOWLEDGEMENTS

This work has been partially funded by the European Commission in the frame of the Horizon 2020 project 5G-PICTURE (grant #762057).

REFERENCES

[1] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, *et al.*, "P4: Programming protocol-independent packet processors", *SIGCOMM Comput. Commun. Rev.*, 2014.

[2] The P4 language Consortium, *P4$_{16}$ Language Specification, version 1.0.0*, available at https://p4.org/p4-spec/docs/P4-16-v1.0.0-spec.pdf.

[3] G. Liu, Y. Ren, M. Yurchenko, K. K. Ramakrishnan, and T. Wood, "Microboxes: High performance nfv with customizable, asynchronous tcp stacks and dynamic subscriptions", in *ACM SIGCOMM '18*, 2018.

[4] P. Shinde, A. Kaufmann, T. Roscoe, and S. Kaestle, "We Need to Talk About NICs", in *Proc. HotOS'13*, 2013.

[5] Y. Le, H. Chang, S. Mukherjee, L. Wang, A. Akella, M. M. Swift, and T. V. Lakshman, "UNO: Uniflying Host and Smart NIC Offload for Flexible Packet Processing", in *SoCC'17*.

[6] A. Kaufmann, S. Peter, N. K. Sharma, T. E. Anderson, and A. Krishnamurthy, "High performance packet processing with flexnic", in *ASPLOS*, 2016.

[7] P. M. Phothilimthana, A. Kaufmann, S. Peter, R. Bodík, and T. E. Anderson, "Floem: A programming system for nic-accelerated network applications", in *OSDI*, 2018.

[8] A. Saeed, N. Dukkipati, V. Valancius, V. The Lam, C. Contavalli, and A. Vahdat, "Carousel: Scalable Traffic Shaping at End Hosts", in *ACM SIGCOMM '17*, 2017.

[9] S. Pontarelli, G. Bianchi, and M. Welzl, "A programmable hardware calendar for high resolution pacing", in *IEEE HPSR 2018*.

[10] A. Caulfield, P. Costa, and M. Ghobadi, "Beyond SmartNICs: Towards a Fully Programmable Cloud", in *IEEE HPSR 2018*.

[11] B. Li, K. Tan, L. L. Luo, Y. Peng, R. Luo, N. Xu, Y. Xiong, P. Cheng, and E. Chen, "Clicknp: Highly flexible and high performance network processing with reconfigurable hardware", in *ACM SIGCOMM'16*.

[12] N. Sultana *et al.*, "Emu: Rapid prototyping of networking services", in *2017 USENIX Annual Tech. Conf. (ATC'17)*, 2017, pp. 459–471.

[13] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "OpenFlow: enabling innovation in campus networks", *ACM SIGCOMM CCR*, 2008.

[14] Open Networking Foundation, "OpenFlow Switch Specification ver 1.5.1 http://www.opennetworking.org/images//openflow-switch-v1.5.1.pdf", Tech. Rep., Mar. 2015.

[15] A. Narayan, F. Cangialosi, D. Raghavan, P. Goyal, S. Narayana, R. Mittal, M. Alizadeh, and H. Balakrishnan, "Restructuring Endpoint Congestion Control", in *ACM SIGCOMM*, 2018.

[16] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek, "The click modular router", *ACM Trans. on Comp. Sys. (TOCS)*, 2000.

[17] R. Bifulco and G. Retvari, "A survey on the programmable data plane: Abstractions, architectures, and open problems", in *IEEE HPSR 2018*.

[18] A. Panda, S. Han, K. Jang, M. Walls, S. Ratnasamy, and S. Shenker, "NetBricks: Taking the V out of NFV", in *OSDI'16*, 2016.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TNSM.2019.2948467, IEEE Transactions on Network and Service Management

[19] M. Gallo and R. Laufer, "Clicknf: A modular stack for custom network functions", in *USENIX ATC'18*, 2018.

[20] G. Bianchi, M. Bonola, A. Capone, and C. Cascone, "Openstate: Programming platform-independent stateful openflow applications inside the switch", *ACM SIGCOMM Comp. Commun. Rev.*, 2014.

[21] S. Pontarelli, M. Bonola, G. Bianchi, A. Capone, and C. Cascone, "Stateful Openflow: Hardware Proof of Concept", in *HPSR'15*.

[22] S. Pontarelli, R. Bifulco, M. Bonola, C. Cascone, M. Spaziani Brunella, V. Bruschi, D. Sanvito, G. Siracusano, A. Capone, M. Honda, and G. Bianchi, "Flowblaze: Stateful packet processing in hardware", in *NSDI '19*, 2019.

[23] G. Bianchi, M. Welzl, A. Tulumello, G. Belocchi, M. Faltelli, and S. Pontarelli, "A fully portable tcp implementation using xfsms", in *Proc. of the ACM SIGCOMM'18 on Posters and Demos*, 2018.

[24] M. T. Arashloo, M. Ghobadi, J. Rexford, and D. Walker, "HotCocoa: Hardware Congestion Control Abstractions", in *Proc. 16th ACM Workshop on Hot Topics in Networks*, ser. HotNets'17, 2017.

[25] R. Laufer, M. Gallo, D. Perino, and A. Nandugudi, "Climb: Enabling network function composition with click middleboxes", *ACM SIGCOMM Comp. Commun. Rev.*, 2016.

[26] S. Jouet and D. P. Pezaros, "BPFabric: Data Plane Programmability for Software Defined Networks", in *Proc. Symp. on Architectures for Networking and Commun. Sys.*, 2017, pp. 38–48.

[27] H. Kim, J. Reich, A. Gupta, M. Shahbaz, N. Feamster, and R. Clark, "Kinetic: Verifiable dynamic network control", in *USENIX NSDI '15*.

[28] Y. Yuan, R. Alur, and B. T. Loo, "NetEgg: programming network policies by examples", in *HotNets '14*, ACM, 2014, p. 20.

[29] M. Moshref, A. Bhargava, A. Gupta, M. Yu, and R. Govindan, "Flow-level state transition as a new switch primitive for SDN", in *3rd workshop on Hot topics in software defined networking*, 2014.

[30] M. Shahbaz and N. Feamster, "The case for an intermediate representation for programmable data planes", in *1st ACM SIGCOMM Symposium on Software Defined Networking Research*, 2015.

[31] M. Schapira and K. Winstein, "Congestion-Control Throwdown", in *HotNets '17*, 2017.

[32] Y. Zaki, T. Pötsch, J. Chen, L. Subramanian, and C. Görg, "Adaptive congestion control for unpredictable cellular networks", in *ACM SIGCOMM '15*, 2015.

[33] M. Alizadeh, A. Greenberg, D. A. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan, "Data Center TCP (DCTCP)", in *ACM SIGCOMM*, 2010.

[34] Y. Zhu, H. Eran, D. Firestone, C. Guo, M. Lipshteyn, Y. Liron, J. Padhye, S. Raindel, M. H. Yahia, and M. Zhang, "Congestion Control for Large-Scale RDMA Deployments", in *ACM SIGCOMM '15*, 2015.

[35] M. Handley, C. Raiciu, A. Agache, A. Voinescu, A. W. Moore, G. Antichi, and M. Wójcik, "Re-architecting datacenter networks and stacks for low latency and high performance", in *SIGCOMM '17*.

[36] S. Pathak and V. S. Pai, "ModNet: A Modular Approach to Network Stack Extension", in *ACM USENIX NSDI*, 2015.

[37] P. S. Schmidt, T. Enghardt, R. Khalili, and A. Feldmann, "Socket Intents: Leveraging Application Awareness for Multi-access Connectivity", in *CoNEXT '13*, 2013.

[38] N. Khademi, D. Ros, M. Welzl, Z. Bozakov, A. Brunstrom, G. Fairhurst, K. J. Grinnemo, *et al.*, "NEAT: A Platform- and Protocol-Independent Internet Transport API", *IEEE Commun. Magaz.*, 2017.

[39] I. Marinos, R. N. Watson, and M. Handley, "Network Stack Specialization for Performance", *SIGCOMM CCR*, 2014.

[40] E. Y. Jeong, S. Woo, M. Jamshed, H. Jeong, S. Ihm, D. Han, and K. Park, "mTCP: A Highly Scalable User-level TCP Stack for Multicore Systems", in *USENIX NSDI'14*, 2014.

[41] K. Yasukata, M. Honda, D. Santry, and L. Eggert, "StackMap: Low-latency Networking with the OS Stack and Dedicated NICs", in *USENIX ATC'16*, 2016.

[42] A. Langley *et al.*, "The QUIC Transport Protocol: Design and Internet-Scale Deployment", in *ACM SIGCOMM '17*, 2017.

[43] Z. Niu, H. Xu, D. Han, P. Cheng, Y. Xiong, G. Chen, and K. Winstein, "Network Stack As a Service in the Cloud", in *HotNets*, 2017.

[44] B. Cronkite-Ratcliff, A. Bergman, S. Vargaftik, M. Ravi, N. McKeown, I. Abraham, and I. Keslassy, "Virtualized Congestion Control (Extended version of SIGCOMM 2016)", Tech. Rep., 2016.

[45] K. He, E. Rozner, K. Agarwal, Y. (Gu, W. Felter, J. Carter, and A. Akella, "AC/DC TCP: Virtual Congestion Control Enforcement for Datacenter Networks", in *ACM SIGCOMM '16*, 2016, pp. 244–257.

[46] A. Narayan, F. Cangialosi, P. Goyal, S. Narayana, M. Alizadeh, and H. Balakrishnan, "The Case for Moving Congestion Control Out of the Datapath", in *HotNets*, 2017.

[47] H. Balakrishnan, H. S. Rahul, and S. Seshan, "An Integrated Congestion Management Architecture for Internet Hosts", in *ACM SIGCOMM '99*, 1999.

[48] T. Flach, N. Dukkipati, A. Terzis, B. Raghavan, N. Cardwell, Y. Cheng, A. Jain, S. Hao, *et al.*, "Reducing web latency: The virtue of gentle aggression", in *ACM SIGCOMM 2013*.

[49] G. Kumar, S. Kandula, P. Bodik, and I. Menache, "Virtualizing Traffic Shapers for Practical Resource Allocation", in *5th USENIX Workshop on Hot Topics in Cloud Computing*, 2013.

[50] S. Radhakrishnan, Y. Geng, V. Jeyakumar, A. Kabbani, G. Porter, and A. Vahdat, "SENIC: Scalable NIC for End-host Rate Limiting", in *Proc. 11th USENIX NSDI*, 2014.

[51] N. Cardwell, Y. Cheng, C. S. Gunn, S. H. Yeganeh, and V. Jacobson, "BBR: Congestion-Based Congestion Control", *Queue*, 2016.

[52] M. Alizadeh, A. Kabbani, T. Edsall, B. Prabhakar, A. Vahdat, and M. Yasuda, "Less is More: Trading a Little Bandwidth for Ultra-low Latency in the Data Center", in *Proc. 9th USENIX NSDI*, 2012.

[53] Q. Li, M. Dong, and P. B. Godfrey, "Halfback: Running Short Flows Quickly and Safely", in *CoNEXT*, 2015.

[54] K. T. Cheng and A. S. Krishnakumar, "Automatic Functional Test Generation Using The Extended Finite State Machine Model", in *ACM Int. Design Automation Conference (DAC)*, 1993, pp. 86–91.

[55] XTRA, *Source Repository*, available at https://github.com/angelotulumello/xtra.

[56] C.-J. Wang and M. T. Liu, "A Test Suite Generation Method for Extended Finite State Machines Using Axiomatic Semantics Approach", in *PSTV*, 1992.

[57] *OpenDataPlane*, available on line at https://www.opendataplane.org.

[58] The NetFPGA SUME team, *Github NetFPGA-SUME-public repository*, https://github.com/NetFPGA/NetFPGA-SUME-public/wiki.

[59] R. Pagh and F. F. Rodler, "Cuckoo hashing", *Journal of Algorithms*, vol. 51, no. 2, pp. 122–144, 2004.

[60] M. Mitzenmacher, "The power of two choices in randomized load balancing", *IEEE Trans. on Parallel and Distrib. Sys.*, 2001.

[61] A. Kirsch and M. Mitzenmacher, "Using a queue to de-amortize cuckoo hashing in hardware", in *Proc. 45th Annual Allerton Conf. on Communication, Control, and Computing*, vol. 75, 2007.

[62] V. Gupta, E. Jackson, S. Qadeer, and S. Rajamani, "P: Safe asynchronous event-driven programming", Tech. Rep. MSR-TR-2012-116, Nov. 2012.

[63] G. Berry and G. Gonthier, "The esterel synchronous programming language: Design, semantics, implementation", *Science of Computer Programming*, pp. 87–152, 2 Nov. 1992.

[64] K. Evensen, A. Petlund, C. Griwodz, and P. Halvorsen, "Redundant bundling in tcp to reduce perceived latency for time-dependent thin streams", *IEEE Communications Letters*, 2008.

[65] R. Kapoor, G. Porter, M. Tewari, G. M. Voelker, and A. Vahdat, "Chronos: Predictable low latency for data center applications", in *Third ACM Symp. on Cloud Computing*, 2012.

[66] S. Shalunov, G. Hazel, J. Iyengar, and M. Kuehlewind, *Low Extra Delay Background Transport (LEDBAT)*, RFC 6817 (Experimental), Int. Eng. Task Force, Dec. 2012.

[67] E. Blanton, M. Allman, L. Wang, I. Jarvinen, M. Kojo, and Y. Nishida, *A Conservative Loss Recovery Algorithm Based on Selective Acknowledgment (SACK) for TCP*, RFC 6675 (Proposed Standard), Int. Eng. Task Force, Aug. 2012.

[68] S. Bhandarkar, A. L. N. Reddy, M. Allman, and E. Blanton, *Improving the Robustness of TCP to Non-Congestion Events*, RFC 4653 (Experimental), Int. Eng. Task Force, Aug. 2006.

[69] N. Dukkipati, M. Mathis, Y. Cheng, and M. Ghobadi, "Proportional Rate Reduction for TCP", in *ACM SIGCOMM IMC'11*, 2011.

[70] M. Rajiullah, P. Hurtig, A. Brunstrom, A. Petlund, and M. Welzl, "An evaluation of tail loss recovery mechanisms for tcp", *SIGCOMM Comput. Commun. Rev.*, vol. 45, no. 1, pp. 5–11, Jan. 2015.

[71] Y. Cheng, N. Cardwell, and N. Dukkipati, "Rack: A time-based fast loss recovery", in *Proc. 98th IETF Meeting*, 2017.

[72] Y. Cheng, N. Cardwell, N. Dukkipati, and P. Jha, "RACK: a time-based fast loss detection algorithm for TCP", Internet Engineering Task Force, Internet-Draft draft-ietf-tcpm-rack-03, 2018.

[73] M. Allman, *TCP Congestion Control with Appropriate Byte Counting (ABC)*, RFC 3465 (Experimental), Int. Eng. Task Force, Feb. 2003.

[74] *Akamai's "state of the internet" Q1 2017 report*, https://www.akamai.com/.

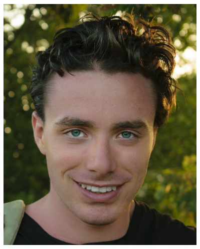
[75] T. Høiland-Jørgensen, B. Ahlgren, P. Hurtig, and A. Brunstrom, "Measuring latency variation in the internet", in *ACM CoNEXT*, 2016.

Giuseppe Bianchi is Full Professor of Networking at the School of Engineering, University of Roma Tor Vergata since January 2007. His research activity includes programmable network systems, wireless networks, privacy and security, traffic control, and is documented in about 230 peer-reviewed international journal and conference papers, having received more than 17.000 citations to date (source scholar.google.com). He has held general or technical coordination roles in six European projects, has served as editor for IEEE/ACM Transactions on Networking, IEEE Transactions on Wireless Communications, IEEE Transactions on Network and Service Management, and Elsevier Computer Communications, and has chaired several conferences and workshops in his field (including IEEE INFOCOM, ACM CoNext, IEEE LANMAN, IEEE WoWMoM, IEEE HPSR, ITC).

Michael Welzl is a full professor in the Department of Informatics of the University of Oslo since 2009. He received his Ph.D. (with distinction) and his habilitation from the University of Darmstadt / Germany in 2002 and 2007, respectively. His book ”Network Congestion Control: Managing Internet Traffic”, is the only introductory book on network congestion control. Michael has been active in the IETF and IRTF for many years, e.g. by chairing the Internet Congestion Control Research Group (ICCRG) leading the effort to form the Transport Services (TAPS) Working Group. He has also participated in several European research projects, including roles such as coordinator and technical manager.

Angelo Tulumello is a researcher at CNIT/University of Rome Tor Vergata since March 2017. He received his Master’s Degree in Internet Engineering in October 2019 and he is currently enrolled in a PhD. in Electronic Engineering at University of Rome Tor Vergata. His research focus is on fast packet processing, SDN/NFV and network programmability. He participated to various EU projects: BEBA, SUPERFLUIDITY and 5G-Picture. In August 2018 he received the 3rd place in the Student Research Competition at the SIGCOMM 2018 conference, presenting a DEMO related to the XTRA project.

Francesco Gringoli received the Laurea degree in telecommunications engineering from the University of Padua, Italy, in 1998 and the PhD degree in information engineering from the University of Brescia, Italy, in 2002. Since 2018 he is Associate Professor of Telecommunications at the Dept. of Information Engineering at the University of Brescia. His research interests include security assessment, performance evaluation and medium access control in Wireless LANs. He is a senior member of the IEEE.



Giacomo Belocchi is a researcher at CNIT/University of Rome Tor Vergata since October 2017. He received his Master’s Degree in Computer Engineering in October 2019 and he is currently enrolled in a PhD. in Computer Science at the University of Tor Vergata. His research interests include SDN/NFV, network programmability and fast packet processing. He is recently focused on studying abstractions for offloading computation to heterogeneous computing systems exploiting Domain Specific Architectures.

Marco Faltelli is a researcher at the CNIT research unit of the University of Rome Tor Vergata since 2017. He received his Master’s Degree in Computer Engineering in October 2019 at the University of Rome Tor Vergata and he’s currently involved there in his PhD in Computer Engineering. He is broadly interested in Computer Networking. His research activities combine computer networking and computer architecture to design programmable state machine based architectures and primitives for both network hardware and software solutions.

Salvatore Pontarelli received a master degree in electronic engineering at University of Bologna and the PhD degree in Microelectronics and Telecommunications from the University of Rome Tor Vergata. Currently, he works as Senior Researcher at CNIT (Italian National Inter-University Consortium for Telecommunications), in the research unit of University of Rome Tor Vergata. In the past he worked with the National Research Council (CNR), the Department of Electronic Engineering of University of Rome Tor Vergata, the Italian Space Agency (ASI), the University of Bristol. He participated in several national and EU funded research program (ESA, FP7 and H2020). His research interests include hash based structures for networking applications, use of FPGA for high speed network monitoring and hardware design of software defined network devices.