TMC: Pay-as-you-Go Distributed Communication

Henri Maxime Demoulin Nikos Vasilakis John Sonchack Isaac Pedisich
Vincent Liu Boon Thau Loo Linh Thi Xuan Phan Jonathan M. Smith
Irene Zhang *
University of Pennsylvania *Microsoft Research

ABSTRACT

We revisit the gap between what distributed systems need from the transport layer and what protocols in wide deployment provide. Such a gap complicates the implementation of distributed systems and impacts their performance. We introduce Tunable Multicast Communication (TMC), an abstraction that allows developers to easily specialize communication channels in distributed systems. TMC is presented as a deployable and extensible user-space library that exposes high-level tunable guarantees. TMC has the potential of improving the performance of distributed applications with minimal-to-zero development and deployment effort.

CCS CONCEPTS

• **Networks → Network design principles**.

KEYWORDS

Networking, Composability, Configurability

1 INTRODUCTION

For most datacenter distributed systems, the network is a black box. Expected to occasionally deliver messages from one end of the datacenter to another, these systems attempt to make few assumptions about the structure, features, or properties of the underlying network. The typical abstraction assumed by most systems is the one provided by TCP: (1) point-to-point connections, (2) ordered, reliable delivery of a connection's bytes implemented over an unreliable network, and (3) no guarantees about the relative behavior of different connections. When the Internet was designed, these properties were assumed to be the union of what applications would reasonably require—any application that did not require these guarantees should use UDP, which provides none whatsoever.

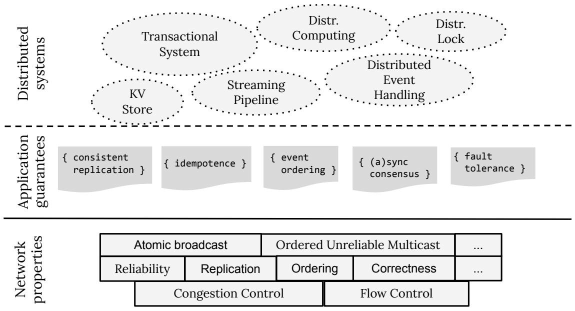

Fig. 1: Exemplary space of communication needs. Distributed systems provide a set of guarantees which depend on a select subset of network properties. Most systems today (*Cf.* Tab. 2) use the network as an over- or under-featured black box.

Recent work has begun to challenge the wisdom of the TCP interface, particularly in datacenter networks where the network is typically easier to control and more reliable than the wider Internet. Mostly-ordered multicast primitives (MOMs) [30] have demonstrated the utility of assuming a more reliable network; ordered, unreliable multicast primitives (OUMs) [24] have demonstrated the power of adding custom network features; and CoFlows [5] have argued for an additional communication abstraction for certain types of application traffic patterns.

In this exploratory work, we argue (perhaps obviously) that applications can benefit from customizing their transport protocols. Less obviously, we argue that more applications *should* use customized transport protocols as TCP effects are harming their performance in subtle and unexpected ways. Our findings show that, for some applications, *adding* network guarantees using programmable switches and NICs can improve performance by orders of magnitude, while, for other applications that can tolerate weaker guarantees, *subtracting* unnecessary features can provide a similar benefit—there is no one-size-fits-all network stack for datacenter applications and networks. Thus, datacenter communication mechanisms must be configurable, message-oriented, and multi-node. In addition, this choice should be easily expressible—in a few lines—and automatically configure the underlining communication infrastructure. We term this capability *Tunable Multicast Communication* (TMC).

APNet '19, August 17–18, 2019, Beijing, China

ACM ISBN 978-1-4503-7635-8/19/08…$15.00
https://doi.org/10.1145/3343180.3343194

Domain	Example Systems
Key-Value Stores	Dynamo, Cassandra, Corfu/Replex, Unispace
Data Processing	MapReduce, Spark, Naiad, Stratosphere, TensorFlow
Lambda/Actor	Orleans, Xenon, Akka.NET, OpenLambda
Other	Acute, J-Orchestra, BreakApp

Tab. 1: TCP-first distributed systems. The use of TCP is the default "go-to" in virtually all distributed systems, coming from both academia and industry. It comes as no surprise that the vast majority of "cloud" datacenter traffic (more than 99.9% [2]) is TCP (*Cf.*§1).

Towards this goal we introduce `libTMC`, a prototype of TMC as a user-space library. The `libTMC` library lets developers define channels with custom features using simple, high-level configurations. Under the hood, `libTMC` realizes a configuration by composing channel primitives that implement the features individually. These primitives can be implemented in user-space (as done in the current prototype) to support deployment without any changes to application logic, language runtimes, operating systems, or the underlying network. In future work, specialized primitives could be implemented to leverage emerging technologies, such as programmable network fabrics [24], for improved performance.

In summary, this paper motivates communication (re-)configurability as a first-class concern for distributed systems in today's networks. It (i) identifies that the need for specialized communication channels in distributed systems is real and pressing (Sec. 2); (ii) sketches a deployable and extensible solution (Sec. 3); and (iii) discusses its potential benefits and limitations (Sec. 3.4).

2 MOTIVATION

This section motivates customized network stacks for datacenter applications. We focus on three features that TMC supports: multicast, message ordering, and replication. We highlight the mismatch between distributed system needs and what TCP or UDP provide (summarized in Tab. 2).

State machine synchronization: Many distributed systems need to maintain consistent replicated state across many nodes, *e.g.*, for configuration [15] or routing databases [20]. Assuming their application code is deterministic, nodes can maintain consistency using algorithms such as Paxos [21]. Those algorithms' performance is largely dependent on the underlying network, and the amount of round trips they have to make to reach consensus. Lower overheads can be obtained when messages use a *replicated* communication channel with *total ordering* (either guaranteed [24] or likely [22]).

Tab. 2: Abstraction mismatch Channel features needed by distributed systems, compared with those provided by TCP and UDP (*Cf.*§2).

	Distributed system task			Transport	
	State synchronization	Distributed messaging	Replicated computation	TCP	UDP
Multicast	✓	✓	✓	✗	✓
Ordering	(partial)	✗	✓	✓	✗
Reliability	✗	✓	✓	✓	✗

Neither TCP nor UDP provide ordering to a *group* of destinations. As a result, several recent works have proposed to make group ordering a network primitive [24, 30], even porting stronger versions—namely, atomic broadcast— in the network as well [18]. Additionally, TCP adds overhead to ensure point-to-point reliability. For state synchronization, this can be a waste because point-to-point reliability is *not* a required channel feature.

To demonstrate those points, we study the case of TAPIR [44], a distributed transaction system that guarantees strong consistency across transactions. TAPIR builds upon an inconsistent replication protocol (IR) to provide its guarantees. To reach consensus, TAPIR requires at least $f + 1$ matching answers from the storage replicas. A "reliable transaction", as understood by TAPIR, is one in which sending a message results in a new entry in the replica's log. Point-to-point reliability in itself does not provide such guarantee. Figure 2 presents the transaction throughput of TAPIR-KV, a key-value store built on top of TAPIR, when the network is experiencing packet losses. We configure the system with a single shard made of three replicas, and run a workload made of 50% GET operations, and 50% PUT.

This experiment is telling: for the baseline, *removing* non required features such as reliability yields about 50% better throughput. *Adding* the right feature, replication (through IP multicast), adds another 13% to this gain.

We might assume that point-to-point reliability, as provided by TCP, would benefit us when the network experiences packet loss. However, when we induce packet loss into the network, TCP's performance is still worse than its less featured counterparts. This is due to the tight coupling between reliability and congestion control in TCP: as packets get dropped, congestion windows decrease and allow for less packets to be sent over the wire.

This small experiment supports our motivation to design a new abstraction for distributed systems, where the right network properties can be combined in a way that benefits the application.

Distributed messaging: Publish–subscribe (pub/sub) messaging systems are a core component of distributed systems, including stream/batch processors, load balancers, and more [9]. Nodes subscribe and publish messages to *topics*. Each published message is transmitted to every subscribed node. In many systems that use pub/sub, nodes are loosely

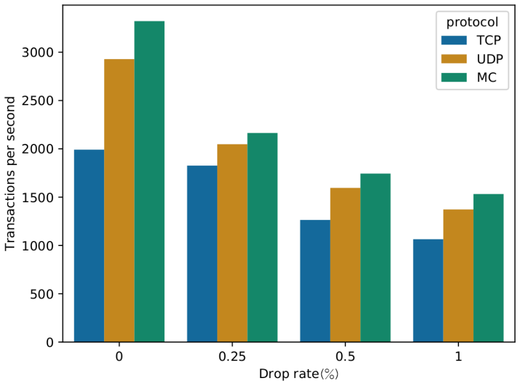

Fig. 2: Throughput comparison. TAPIR-KV transaction throughput for three replicas. Packets are dropped in the network at .25%, .5% and 1% rate. All hardware offload features were disabled during this experiment.

coupled. Subscriber nodes, such as analytics engines aggregating statistics from a set of endpoints [4, 27], need to receive all the messages published to a channel, but are indifferent to their ordering. These scenarios require a channel with *multicast* and *reliability*, but not ordering, would it be point-to-point or at the group level.

Neither TCP nor UDP provide reliable multicast. Today, many systems implement it at the application layer, on top of TCP, with point-to-point connections between the source and every destination. This adds complexity and reduces performance, from opening and maintaining stateful connections, duplicating messages, and traversing TCP stacks [35].

Replicated computation: Many distributed systems replicate state machines across multiple servers, *e.g.*, for fault-tolerant services [36] or distributed simulations [29]. In these systems, participants broadcast events containing enough information for all recipients to integrate the sender's action to their state machine. For correctness, events must be replayed in order, and cannot be lost. Thus, this class of distributed systems needs all three channel features: *multicast*, *reliability*, and *group ordering*.

As described in the prior example, TCP channels do not provide all of these features, forcing complex application layer protocols and reducing performance. There is an additional TCP-related performance issue for these systems: latency, which is greatly increased at the tail by TCP's packet retransmission feature. Many services that replicate computation are not only fault-tolerant, but also latency sensitive [8]. in practice, systems mitigate the latency of packet retransmissions with custom network protocols, for example that transmit every datagram multiple times to reduce the probability of loss [12].

Conclusion: The systems described above motivate the need for communication channels that selectively support multicast, ordering, and reliability. But, as Tab. 3 illustrates, they only scratch the surface of possible channel features a group of applications might need. In fact, diverse needs may arise even *within* a single application: recent proposals [14, 41, 42] advocate for mixed consistency, replication, and indexing guarantees at object-granularity.

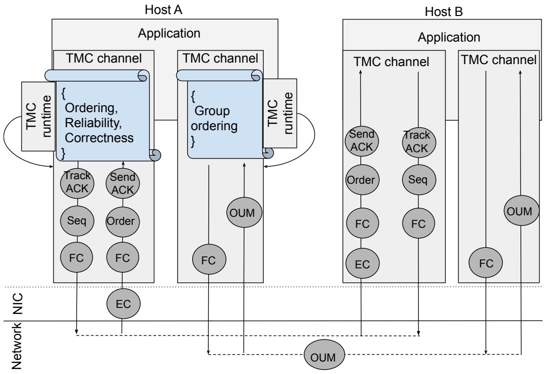

Fig. 3: TMC design with channel examples. Applications declare their requirements using a manifest of properties. `libTMC` composes and optimizes them, exploiting both local hardware and network capabilities. In this example, host B's network interface is not able to compute checksums (EC), so `libTMC` implements it in software.

3 TMC DESIGN

This section introduces the design and semantics of `libTMC`, our TMC prototype. `libTMC` is a flexible library for custom transport *channels* in distributed systems. It meets diverse application needs by mapping channel features, *e.g.*, ordering and reliability, to a library of *channel primitives*. Nodes in a distributed system select, and pay for, only the features that they need. A TMC runtime, implemented at the application layer for simple deployment, composes and optimizes the appropriate channel primitives to realize node-level objectives.

Figure 3 depicts TMC's design through the example of an application using two communication channels. The first guarantees reliable delivery of messages, their (point-to-point) ordering, and payload error detection. The second guarantees group ordering of messages, without reliability.

`libTMC`'s runtime acts as a compiler that composes a set of actions that should be done given the input properties. It decides whether to have actions performed in software or in hardware based on properties available locally and in the network. These properties can either be declared in a configuration file by the network operator, as done in the current prototype, or automatically learned by `libTMC`, as we plan to investigate in future work. For example, as depicted in figure 3, if host A's NIC can verify packet's checksums in hardware, the runtime will make use of this capability by configuring channels to offload error checking to the NIC. If

```
1.  from libTMC import Channel
2.
3.  c = Channel({
4.    'id': '2PC',
5.    'group': '239.0.0.2',
6.    'group ordering': True,
7.  })
8.
9.  c.send(handshake_msg, {'reliability' : True})
10. c.update({'error checking': True})
11. c.send(beacon_msg)
```

Listing 1: `libTMC` Example. The developer imports (1), configures (3–7), and uses (9–11) `libTMC` to send messages of varying requirements (*Cf.*§3.1).

not, as is the case for host B, the runtime configures the channel to perform this stage in software. Likewise, `libTMC` also integrates the semantic knobs from more complex primitives offered by the network, such as OUM [24].

TMC allows the integration of several specialized congestion control mechanisms [2, 19]. Congestion and flow control are components of all the channels, but their *modus operandi* is dependent on the other features with which they are collocated. In our example (fig 3), the application is set to run on a private cluster whose competing traffic is known and under control. For this reason, the runtime sets up a lightweight flow control scheduling policy that operates before messages are dispatched to the NIC.

3.1 `libTMC` API

Channels allow sending and receiving messages, with a simple, POSIX-influenced interface—essentially, `send` and `receive` methods. They are configured programmatically via *manifests*, declarative runtime configuration objects expressed using a domain-specific language embedded in the source language. Manifests can apply to a single message or the entire communication. By controlling automated generation of channel parameters, manifests allow developers to tune several communication trade-offs without requiring manual development.

Internally, the `libTMC` runtime maintains state associated with each channel. When a node joins a channel, `libTMC` associates the node with that channel. Other information includes acknowledgment and ordering metadata, as well as various statistics related to its usage—such as latency information and loss ratios.

We illustrate `libTMC`'s API with the example code snippet in listing 1. The snippet is written using Python bindings. The developer first imports the `libTMC` library (line 1) which exposes a `Channel` constructor in the current scope. It then calls the constructor by passing a manifest that declares the channel properties (lines 3–7) which returns the object used to send and receive messages. The channel is required to provide group ordering to messages (line 6). For the first message (line 9), the semantics of which is application-specific, the developer configures reliable delivery. For the rest of the messages, the desired semantics is ensuring payload checksum verification, expressed by permanently updating the channel manifest (line 10).

3.2 Channel Configuration

Table 3 lists preliminary network features supported by `libTMC`. These (and other) features are available as pluggable modules (Sec. 3.3). We provide more details for a select subset below.

Multicast: At a minimum, a TMC channel can target multiple nodes. Every node in the group effectively targets all other nodes in the group when sending messages. TMC implements naming and filtering mechanisms for flexible group management at the application-layer. Specifically, message properties can target a named subset or a fraction of endpoints in a group, such that an application can treat individual endpoints as they would with point-to-point connections, while still reasoning about a group of nodes in general.

Point-to-point ordering: TMC channels are ordered if all messages sent by one node are read by receivers in the same order as they were sent. When a node joins a channel, a local base sequence number (epoch) is randomly generated, and is then used to identify messages sent by the node through the channel. The process is enforced by the `libTMC` runtime, which stamps sent messages and drops those received with a sequence number smaller than the latest received.

Reliable Delivery: `libTMC`'s reliable channels allow applications to ensure delivery of messages through acknowledgments to either all or a subset of the channel's members. The channel can be configured such that acknowledgments are expected from a subset of hosts designated by name, or a fraction of the channel's members. The base reliability mechanism is composable with flow and congestion control to pace messages retransmission. The library supports sending windows for channels, similarly to TCP, so that multiple non-acknowledged packets can be in-flight at the same time. In addition, it drops all duplicate packets and handles lost ACK messages by re-acknowledging duplicate packets retransmitted by the sender.

3.3 Ongoing Work

Currently, there are several features of `libTMC` that are works in progress. `libTMC` needs to facilitate extensibility beyond the "built-in" presets, acceleration based on features provided by the underlying network, integration of application contextual knowledge, and easy retrofit into existing codebases.

Extensibility: `libTMC` should have the ability to be extended with capabilities that were not anticipated by its designers. To

Tab. 3: `libTMC` channel features. Network features each come with a fundamental trade-off between guarantees and speed (*Cf.*§3.2).

Feature	Fundamental trade-off	Examples use cases
Multicast	Trade fine grained control over one-to-one traffic for scalable one-to-many properties	Consensus, Gossip
Point-to-point ordering	Trade latency for providing a guarantee to the application	Zookeeper Atomic broadcast
Group ordering	Trade latency for providing a guarantee to the application	Distr. transactions
Reliability	Trade bandwidth usage and inter-message delay for tolerance to network losses	TLS handshake, Distr. locks
Pacing	Trade latency for throughput, bandwidth, and reduced network and CPU usage	Playout delay buffers
Flow control	Trade latency and availability for less packet drops	Message scheduling
Congestion control	Trade latency for intermediate devices availability	Message scheduling
Synchronicity	Trade simplicity in application logic for liveliness	Event driven APIs
Duplexity	Simplex, half and full duplex offer varying degrees of channel complexity	Publish/subscribe systems

solve this, `libTMC` provides a module manager, `libTMC_MM`, that enables the integration of new primitive implementations. Such implementations are provided through a verified repository and, at times, expose a handful of high-level parameters with their default values. Examples include different congestion control algorithms and ordering guarantees.

Hardware Acceleration: `libTMC`'s prototype comes with portable user-space implementations of its features, but should leverage hardware acceleration in the network when available. We envision a solution where programmable hardware elements, *e.g.*, smartNICs [33] or switches [37], run line-rate implementations of most channel primitives [11, 24]. Authorized servers will pre-reserve capacity on these elements via an extended control-plane interface along the lines of participatory networking [10]. At runtime, `libTMC` will tag packets that need to be processed by the line-rate primitives and the network will route them through the appropriate elements.

Application-guided decisions: an important design goal for TMC is to let applications hint the network stack about how certain decisions should be done. For instance, the application knows how important the delivery of certain messages really is: informing the network stack that a set of non acknowledged messages can be forgotten allows for optimized resource management. Similarly, an application can share load related information with the network stack, such as the occupancy of a local event queue, to influence flow control decisions (a principle whose benefits have been exploited in recent work [28]). As this paper and several of the works we build upon have shown, the data center is an ideal environment for such co-design, which can yield thousand-fold performance improvements [18, 24].

Retrofit: To simplify deployment, `libTMC` should require minimal-to-zero changes to the code of legacy applications. Using a combination of automated source rewriting, name rebinding, and runtime reflection, a transformation subsystem can rewire connections to their TMC equivalents—requiring only the manifest that specifies desired channel properties. [1]

3.4 Discussion

This paper introduces TMC early in the project life-cycle, and is intended to sparkle discussion with the community.

From our experience, the most controversial aspect of this work is the decision to supply developers with more knobs. There are two possible criticisms here: (i) developers do not need more knobs, as they increase the risk of getting things wrong; (ii) whoever needs true specialization can build it from scratch. The former misses the point: distributed-system trade-offs dwarf the ones of centralized systems, and thus developers are forced to implement specialization from scratch, a process that is more error-prone than expressing high-level annotations. But this concern is precisely the reason why we went for understandable high-level properties (*e.g.*, "ordering") rather than bare protocol building blocks (*e.g.*, "ACKS"). As for the latter, the few companies with unlimited engineering resources will still benefit from TMC, but long-term they may be better off handcrafting specialized protocol stacks from scratch. This work targets everyone else, from the vast majority of developers not working for these select few companies, to researchers in the field of distributed systems (like us), to designers of novel network protocols (who can expose their work as a TMC module).

4 RELATED WORK

Prior work on communication specialization can be grouped into network stack (re)configurability, protocol specialization, and kernel bypassing.

Configurability: Our work can be viewed as revisiting the need for modular, configurable stacks [6, 16, 38]. For example, x-Kernel [16] exposed network services as coarsely composable protocol objects. Horus [38] extended the idea into the distributed setting, and P2 [6] introduced reconfiguration patterns (*e.g.*, network function reordering and replacement).

---

[1] Here we leverage prior work on runtime transformations [39, 40].

These works modularize network stacks into fine-grained building blocks and expose them for synthesis from within the application. With TMC, developers specify intuitive, high-level properties which the system translates into end-to-end guarantees. Moreover, TMC properties can be specified at the level of individual messages within applications, rather than entire network stacks.

Protocol Specialization: Several proposals change the semantics or implementation of transport protocols according to application needs [1, 3, 13, 17, 19, 23]. Examples include group reliability [3, 32], adaptive changes to TCP's send buffer size [13], and congestion window sharing [17]. These recognize the mismatch between a couple of transport configurations and the space of possible application needs, but offer more "point" solutions. TMC's goal is a fundamentally different framing of the problem—the need for an application-*tunable* abstraction that eases the testing, integration and adoption of novel "point" solutions as pluggable components.

Kernel Bypassing: Operating system kernel bypassing and user-space network processing shares our goal of improving application control [7, 31, 34, 43]—in the limit, the entire network stack can be specialized for the application [25, 26]. Those techniques are fundamentally orthogonal (and complementary) to TMC and can be used to further reduce the performance costs for distributed applications (Sec. 3.3).

5 CONCLUSION

Distributed systems are inherently communicating systems. Developers pay too much by not being able to specialize communication in distributed applications—most notably, in terms of development and performance costs. This paper proposes a new abstraction, Tunable Multicast Communication (TMC), that allows developers to easily specify the channel features that best match their needs. Using TMC, they can compose features at the granularity of messages by providing high-level, semantic guidelines. The design allows extensibility beyond the "built-in" presets and further acceleration based on network capabilities. Our prototype implementation, `libTMC`, is in progress.

ACKNOWLEDGMENTS

We would like to thank André DeHon, Ben Karel, and the anonymous reviewers for their helpful feedback. This research was funded in part by NSF grants CNS-1703936, CNS-1750158, CNS-1513687, CNS-1845749 and CNS-1513679; DARPA contracts HR0011-16-C-0056 and HR0011-17-C-0047; and ONR N00014-18-1-2557. Any opinions, findings, conclusions, or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of NSF, DARPA or ONR.

REFERENCES

[1] [n.d.]. SPDY: An Experimental Protocol for a Faster Web. http://www.chromium.org/spdy/spdy-whitepaper.

[2] Mohammad Alizadeh, Albert Greenberg, David A. Maltz, Jitendra Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan. 2010. Data Center TCP (DCTCP). In *Proceedings of the ACM SIGCOMM 2010 Conference (SIGCOMM '10)*. ACM, New York, NY, USA, 63–74. https://doi.org/10.1145/1851182.1851192

[3] T. Bova and T. Krivoruchka. [n.d.]. RELIABLE UDP PROTOCOL. https://tools.ietf.org/html/draft-ietf-sigtran-reliable-udp-00/.

[4] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. 2015. Apache Flink: Stream and Batch Processing in a Single Engine. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering* 36, 4 (2015).

[5] Mosharaf Chowdhury and Ion Stoica. 2012. Coflow: a networking abstraction for cluster applications.. In *HotNets*. 31–36.

[6] Tyson Condie, Joseph M Hellerstein, Petros Maniatis, and Sean Rheaand Timothy Roscoe. 2005. Finally, a use for componentized transport protocols. In *HotNets IV*, Vol. 13.

[7] RDMA Consortium. October 2002. An RDMA Protocol Specification. http://rdmaconsortium.org/.

[8] Jeffrey Dean and Luiz André Barroso. 2013. The Tail at Scale. *Commun. ACM* 56, 2 (Feb. 2013), 74–80. https://doi.org/10.1145/2408776.2408794

[9] Patrick Th. Eugster, Pascal A. Felber, Rachid Guerraoui, and Anne-Marie Kermarrec. 2003. The Many Faces of Publish/Subscribe. *ACM Comput. Surv.* 35, 2 (June 2003), 114–131. https://doi.org/10.1145/857076.857078

[10] Andrew D. Ferguson, Arjun Guha, Chen Liang, Rodrigo Fonseca, and Shriram Krishnamurthi. 2013. Participatory Networking: An API for Application Control of SDNs. *SIGCOMM Comput. Commun. Rev.* 43, 4 (Aug. 2013), 327–338. https://doi.org/10.1145/2534169.2486003

[11] Hans Giesen, Lei Shi, John Sonchack, Anirudh Chelluri, Nishanth Prabhu, Nik Sultana, Latha Kant, Anthony J McAuley, Alexander Poylisher, André DeHon, and Boon Thau Loo. 2018. In-network Computing to the Rescue of Faulty Links. In *Proceedings of the 2018 Morning Workshop on In-Network Computing (NetCompute '18)*. ACM, New York, NY, USA, 1–6. https://doi.org/10.1145/3229591.3229595

[12] Glenn Fiedler. 2014. Deterministic Lockstep. https://gafferongames.com/post/deterministic_lockstep/.

[13] Ashvin Goel, Charles Krasic, and Jonathan Walpole. 2008. Low-latency Adaptive Streaming over TCP. *ACM Trans. Multimedia Comput. Commun. Appl.* 4, 3, Article 20 (Sept. 2008), 20 pages. https://doi.org/10.1145/1386109.1386113

[14] Rachid Guerraoui, Matej Pavlovic, and Dragos-Adrian Seredinschi. 2016. Incremental Consistency Guarantees for Replicated Objects. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation (OSDI'16)*. USENIX Association, Berkeley, CA, USA, 169–184. http://dl.acm.org/citation.cfm?id=3026877.3026891

[15] Patrick Hunt, Mahadev Konar, Flavio P. Junqueira, and Benjamin Reed. 2010. ZooKeeper: Wait-free Coordination for Internet-scale Systems. In *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference (USENIXATC'10)*. USENIX Association, Berkeley, CA, USA, 11–11. http://dl.acm.org/citation.cfm?id=1855840.1855851

[16] Norman C Hutchinson and Larry L Peterson. 1991. The x-kernel: An architecture for implementing network protocols. *IEEE Transactions on Software engineering* 17, 1 (1991), 64–76.

[17] Safiqul Islam and Michael Welzl. 2016. Start Me Up: Determining and Sharing TCP's Initial Congestion Window. In *Proceedings of the 2016 Applied Networking Research Workshop (ANRW '16)*. ACM, New York, NY, USA, 52–54. https://doi.org/10.1145/2959424.2959440

[18] Zsolt István, David Sidler, Gustavo Alonso, and Marko Vukolic. 2016. Consensus in a Box: Inexpensive Coordination in Hardware. In *Proceedings of the 13th Usenix Conference on Networked Systems Design and Implementation (NSDI'16)*. USENIX Association, Berkeley, CA, USA, 425–438. http://dl.acm.org/citation.cfm?id=2930611.2930639

[19] Eddie Kohler, Mark Handley, and Sally Floyd. 2006. Designing DCCP: Congestion control without reliability. In *ACM SIGCOMM Computer Communication Review*, Vol. 36. ACM, 27–38.

[20] Teemu Koponen, Martin Casado, Natasha Gude, Jeremy Stribling, Leon Poutievski, Min Zhu, Rajiv Ramanathan, Yuichiro Iwata, Hiroaki Inoue, Takayuki Hama, and Scott Shenker. 2010. Onix: A Distributed Control Platform for Large-scale Production Networks. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation (OSDI'10)*. USENIX Association, Berkeley, CA, USA, 351–364. http://dl.acm.org/citation.cfm?id=1924943.1924968

[21] Leslie Lamport. 1998. The Part-time Parliament. *ACM Trans. Comput. Syst.* 16, 2 (May 1998), 133–169. https://doi.org/10.1145/279227.279229

[22] Leslie Lamport. 2006. Fast Paxos. *Distrib. Comput.* 19, 2 (Oct. 2006), 79–103. https://doi.org/10.1007/s00446-006-0005-x

[23] Adam Langley, Alistair Riddoch, Alyssa Wilk, Antonio Vicente, Charles Krasic, Dan Zhang, Fan Yang, Fedor Kouranov, Ian Swett, Janardhan Iyengar, Jeff Bailey, Jeremy Dorfman, Jim Roskind, Joanna Kulik, Patrik Westin, Raman Tenneti, Robbie Shade, Ryan Hamilton, Victor Vasiliev, Wan-Teh Chang, and Zhongyi Shi. 2017. The QUIC Transport Protocol: Design and Internet-Scale Deployment. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication (SIGCOMM '17)*. ACM, New York, NY, USA, 183–196. https://doi.org/10.1145/3098822.3098842

[24] Jialin Li, Ellis Michael, Naveen Kr. Sharma, Adriana Szekeres, and Dan R. K. Ports. 2016. Just Say No to Paxos Overhead: Replacing Consensus with Network Ordering. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation (OSDI'16)*. USENIX Association, Berkeley, CA, USA, 467–483. http://dl.acm.org/citation.cfm?id=3026877.3026914

[25] Anil Madhavapeddy, Richard Mortier, Charalampos Rotsos, David Scott, Balraj Singh, Thomas Gazagnaire, Steven Smith, Steven Hand, and Jon Crowcroft. 2007. Unikernels: Library Operating Systems for the Cloud. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '13)*. ACM, New York, NY, USA, 461–472. https://doi.org/10.1145/2451116.2451167

[26] Ilias Marinos, Robert N.M. Watson, and Mark Handley. 2014. Network Stack Specialization for Performance. In *Proceedings of the 2014 ACM Conference on SIGCOMM (SIGCOMM '14)*. ACM, New York, NY, USA, 175–186. https://doi.org/10.1145/2619239.2626311

[27] Srinivas Narayana, Anirudh Sivaraman, Vikram Nathan, Prateesh Goyal, Venkat Arun, Mohammad Alizadeh, Vimalkumar Jeyakumar, and Changhoon Kim. 2017. Language-Directed Hardware Design for Network Performance Monitoring. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication (SIGCOMM '17)*. ACM, New York, NY, USA, 85–98. https://doi.org/10.1145/3098822.3098829

[28] Amy Ousterhout, Joshua Fried, Jonathan Behrens, Adam Belay, and Hari Balakrishnan. 2019. Shenango: Achieving High CPU Efficiency for Latency-sensitive Datacenter Workloads. In *Proceedings of the 16th USENIX Conference on Networked Systems Design and Implementation (NSDI'19)*. USENIX Association, Berkeley, CA, USA, 361–377. http://dl.acm.org/citation.cfm?id=3323234.3323265

[29] Stefan Poledna. 1996. *Fault-Tolerant Real-Time Systems: The Problem of Replica Determinism.* Kluwer Academic Publishers, Norwell, MA, USA.

[30] Dan R. K. Ports, Jialin Li, Vincent Liu, Naveen Kr. Sharma, and Arvind Krishnamurthy. 2015. Designing Distributed Systems Using Approximate Synchrony in Data Center Networks. In *Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation (NSDI'15)*. USENIX Association, Berkeley, CA, USA, 43–57. http://dl.acm.org/citation.cfm?id=2789770.2789774

[31] Ian Pratt and Keir Fraser. 2001. Arsenic: A user-accessible gigabit ethernet interface. In *Proceedings IEEE INFOCOM 2001. Conference on Computer Communications. Twentieth Annual Joint Conference of the IEEE Computer and Communications Society (Cat. No. 01CH37213)*, Vol. 1. IEEE, 67–76.

[32] Dave Presotto and Phil Winterbottom. 1995. The IL protocol. *AT&T ell Laboratories, Murray Hill, NJ* (1995), 277–282.

[33] Andrew Putnam, Adrian M Caulfield, Eric S Chung, Derek Chiou, Kypros Constantinides, John Demme, Hadi Esmaeilzadeh, Jeremy Fowers, Gopi Prashanth Gopal, Jan Gray, et al. 2014. A reconfigurable fabric for accelerating large-scale datacenter services. *ACM SIGARCH Computer Architecture News* 42, 3 (2014), 13–24.

[34] Luigi Rizzo. 2012. Netmap: A Novel Framework for Fast Packet I/O. In *Proceedings of the 2012 USENIX Conference on Annual Technical Conference (USENIX ATC'12)*. USENIX Association, Berkeley, CA, USA, 9–9. http://dl.acm.org/citation.cfm?id=2342821.2342830

[35] Luigi Rizzo. 2012. Revisiting Network I/O APIs: The Netmap Framework. *Queue* 10, 1, Article 30 (Jan. 2012), 10 pages. https://doi.org/10.1145/2090147.2103536

[36] Fred B. Schneider. 1993. Distributed Systems (2Nd Ed.). ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, Chapter Replication Management Using the State-machine Approach, 169–197. http://dl.acm.org/citation.cfm?id=302430.302437

[37] Anirudh Sivaraman, Alvin Cheung, Mihai Budiu, Changhoon Kim, Mohammad Alizadeh, Hari Balakrishnan, George Varghese, Nick McKeown, and Steve Licking. 2016. Packet Transactions: High-Level Programming for Line-Rate Switches. In *Proceedings of the 2016 ACM SIGCOMM Conference (SIGCOMM '16)*. ACM, New York, NY, USA, 15–28. https://doi.org/10.1145/2934872.2934900

[38] Robbert van Renesse, Kenneth P. Birman, and Silvano Maffeis. 1996. Horus: A Flexible Group Communication System. *Commun. ACM* 39, 4 (April 1996), 76–83. https://doi.org/10.1145/227210.227229

[39] Nikos Vasilakis, Ben Karel, Yash Palkhiwala, John Sonchack, André DeHon, and Jonathan M. Smith. 2019. Ignis: Scaling Distribution-Oblivious Systems with Light-Touch Distribution. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2019)*. ACM, New York, NY, USA, 1010–1026. https://doi.org/10.1145/3314221.3314586

[40] Nikos Vasilakis, Ben Karel, Nick Roessler, Nathan Dautenhahn, André DeHon, and Jonathan M. Smith. 2018. BreakApp: Automated, Flexible Application Compartmentalization. In *Networked and Distributed Systems Security (NDSS'18)*. https://doi.org/10.14722/ndss.2018.23131

[41] Nikos Vasilakis, Yash Palkhiwala, and Jonathan M. Smith. 2017. Query-efficient Partitions for Dynamic Data. In *Proceedings of the 8th Asia-Pacific Workshop on Systems (APSys '17)*. ACM, New York, NY, USA, Article 23, 8 pages. https://doi.org/10.1145/3124680.3124744

[42] Dimitrios Vasilas, Marc Shapiro, and Bradley King. 2018. A Modular Design for Geo-distributed Querying: Work in Progress Report. In *Proceedings of the 5th Workshop on the Principles and Practice of Consistency for Distributed Data (PaPoC '18)*. ACM, New York, NY, USA, Article 4, 4 pages. https://doi.org/10.1145/3194261.3194265

[43] T. von Eicken, A. Basu, V. Buch, and W. Vogels. 1995. U-Net: A User-level Network Interface for Parallel and Distributed Computing. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles (SOSP '95)*. ACM, New York, NY, USA, 40–53. https://doi.org/10.1145/224056.224061

[44] Irene Zhang, Naveen Kr. Sharma, Adriana Szekeres, Arvind Krishnamurthy, and Dan R. K. Ports. 2015. Building Consistent Transactions with Inconsistent Replication. In *Proceedings of the 25th Symposium on Operating Systems Principles (SOSP '15)*. ACM, New York, NY, USA, 263–278. https://doi.org/10.1145/2815400.2815404