Nano-consensus: ultra-fast, quorum-less coordination on the wire

(To appear at ACM Symposium on Cloud Computing 2025)

Davide Rovelli
Università della Svizzera Italiana
SAP
Switzerland

Christian Faerber
Graham McKenzie
Altera
Germany, United Kindgdom

Ali Pahlevan
SAP
Germany

Sina Darabi
Università della Svizzera Italiana
Switzerland

Patrick Jahnke
turbalance
Germany

Patrick Eugster
Università della Svizzera Italiana
Switzerland

ABSTRACT

Consensus, widely regarded as the most fundamental primitive in distributed systems, lies at the core of countless services that require coordination among remote processes. Datacenter services typically achieve consensus through long-established, quorum-based algorithms such as Paxos and Raft, including recent re-adaptations for kernel bypass datapaths (e.g. smartNIC/RDMA-based consensus). While these optimizations can reduce latency to the µs-scale, they remain constrained by inherent message complexity, namely the need for acknowledgments from majority quorums to tolerate faults and arbitrary message delays. Our approach takes a step further from bare acceleration of classical primitives, focusing instead on leveraging FPGA-smartNIC and priority-queue reservation to achieve *synchronous remote interactions* in practice. We use synchrony to devise a novel, efficient quorum-less consensus protocol which we use to build *Nano-consensus*: a novel hardware consensus engine. Nano-consensus operates at network line rate and can reach consensus in 1.03µs for single-packet instances, delivering 3.82× latency and 4.8× improvements over the state of the art. We demonstrate how Nano-consensus can be integrated into distributed applications to boost both performance and consistency.

1 INTRODUCTION

Coordination in datacenters. Modern high-performance, user-facing applications including µs-scale key-values stores and high-frequency trading frameworks are deployed as interactive online services running 24×7 in datacenters with stringent availability and reliability requirements only be met by replication and orchestration on multiple resources. Such distributed coordination is provided by algorithms which solve the well-known consensus problem. As network bandwidth approaches the Tbps limit [3] and demands increase accordingly, a primary concern in datacenter design is making consensus efficient to avoid it being a bottleneck without sacrificing fault tolerance or consistency.

The limits of software coordination. This very challenging task has recently received significant research attention, with focus on accelerating traditional quorum-based consensus algorithms via kernel bypass technologies in software. Recent solutions include adaptations of popular Paxos [50] and Raft [64] algorithms to custom network stacks [41, 46], remote direct memory access (RDMA) [7, 31, 40], data plane development kit (DPDK) [48], and extended Berkeley packet filter (eBPF) express data path (XDP) [72, 86]. By overcoming the limitations posed by the commodity network stack, these approaches manage to reach consensus in tens of µs while withstanding high request throughput in the *common-case*. Alas, software coordination algorithms often have to compromise good common-case performance benefits with high *tail-latency* beyond some $99.x^{\text{th}}$ percentile. This limitation arises from the inherent multi-tasking nature of the underlying software stack and operating system (OS), which must inevitably sacrifice, i.e., delay/preempt, some processing tasks upon contention. In addition, on top of relying on specific kernel bypass technologies (often a given version), these solutions employ disruptive optimizations (e.g., custom priority scheduling [28, 58], heavily customized OS and power configuration settings [31, 41, 72]), and also require a lot of server CPU cores and network bandwidth. As a result, software coordination algorithms are often difficult to deploy and co-locate on general-purpose servers – contrary to the common assumption that software inherently offers such flexibility.

Shortcomings of bare acceleration. The rising availability of smart network interface controllers (smartNICs) equipped with field programmable gate arrays (FPGAs) in major clouds, e.g., Alibaba [9], Amazon [12] and several others [15], offer a more efficient, more stable and self-contained alternative to software-based packet processing routines. A small number of hardware-supported consensus algorithms have emerged in this context, including services which partially or fully offload algorithms to programmable network switches [14, 22, 23, 43] or to programmable smartNICs [41]. FPGA-based smartNIC (FPGA-smartNIC) solutions [10, 36, 39], spearheaded by Consensus in a Box [39], have proven to be able to handle a near-to-capacity rate of client requests at a low latency [10], outperforming software solutions. However, these approaches have a fundamental limitation in that they only focus on *accelerating* classic quorum-based consensus algorithms which were devised for slower, unreliable networks and unpredictable software processing latency, thus overlooking the interaction (communication + processing) *stability* of modern network hardware. As shown in our extensive tests under heavy network and processing stress (see Tab. 1), hardware worst-case remote interaction latency is only *few nanoseconds* more than the average, unlike traditional software stacks which incur heavy latency degradation towards the tail. Our work builds on this key observation: the combination of uninterrupted access to incoming packets on dedicated hardware modules and reliable datacenter network protocols providing bounded latency [30, 41, 72–74] make it practical to build systems relying on *synchronous interactions* for critical tasks. FiDe [72], for instance, builds synchrony support through traffic engineering and process isolation techniques to implement reliable process failure detection. FiDe however targets commodity systems without smartNICs and relies on complex OS fine-tuning which limits its reliability.

Unleashing hardware potential with nano-consensus. We propose Nano-consensus, a novel system based on software/FPGA-based smartNICs co-design which exploits the programmability and predictability of datacenter networks. Our prototype shows that placing logic close to the wire leads to extremely robust processing times which, together with traffic prioritization in network switches, limits the worst-case latency to a mere few clock cycles more than the average (30ns in our experiments). We use the achieved interaction stability to devise a novel, leader-based consensus primitive, dubbed looped one-way imposition (LOWI), which requires only one message delay to reach consensus when multiple instances are executed in series, e.g., in state machine replication (SMR). To be clear, the efficiency and correctness of LOWI depend on assuming synchrony. While all-encompassing synchrony is unfeasible, we find the probability of synchrony violations in a controlled datacenter environment is negligible for practical purposes, as also evidenced by recent software services [41, 72]. Nano-consensus further supports synchrony from the ground up by delegating its execution entirely to network hardware, in contrast to works that simply take synchrony as a given including replicated services [1, 2, 70] and blockchains [4, 33, 60, 61]. As network failures occur less frequently than protocol-level errors like CRC checks, operating under the assumption that the network failures that occur can be masked by redundancy does not substantially affect our system's availability.

Improvements. LOWI introduces important algorithmic improvememts with respect to classical synchronous consensus algorithms, e.g., by Lynch [57] in which processes disseminate n^2 messages per round and different correct processes might decide in different rounds. Nano-consensus handles failures in hardware and can tolerate up to $n-1$ failures of the n endhosts, showcasing how re-architecting services considering modern network capabilities can achieve optimality in both message complexity and resilience. Our prototype can process packets at network line-rate, achieving 1-microsecond latency for a consensus decision – unprecedented to our knowledge. It also handles leader failures in 2µs and can achieve a server response time of 1.03µs when used to implement a simple SMR application, respectively 30× and 3.82× lower latency and 4.8× higher throughput than Waverunner [10] – the fastest FPGA-smartNIC SMR engine. Nano-consensus also outperforms software implementations and is deployed on modern SoC + FPGA off-path smartNICs that can be connected to a server out-of-the-box, without affecting the host.

Contributions. After presenting background information in § 2 this paper makes the following contributions:

- § 3 We propose the design of Nano-consensus, a novel coordination engine operating as fast as the underlying network.
- § 4 We introduce a new quorum-less consensus algorithm using practical synchronous interaction supported by our system.

Table 1: Different percentiles of worst-case latencies measured over 40 days (50th corresponds to the median) in µs.

	50th	99th	99.99th	100th
Nano-consensus	1.41	1.45	1.45	1.48
RDMA	3.15	3.93	8.18	144.78
AF_XDP	13.54	29.03	51.77	170.57
UDP	13.48	72.07	165.04	996.65

- § 5 We outline the implementation of Nano-consensus on Altera's F2000X-PL [37] infrastructure processing unit (IPU).
- § 6 We empirically evaluate our prototype in terms of interaction stability, performance, and failure recovery in a SAP datacenter. In short, Nano-consensus outperforms state-of-the-art hardware and software solutions, e.g., respectively reducing latency by 3.82× and 5.3× and increasing goodput by 4.8× and 12×. We use Nano-consensus to implement SMR which we integrate into a key-value store, providing strong consistency without compromising native performance.

§ 7 puts Nano-consensus into perspective. § 8 contrasts it with related work. § 9 concludes with final remarks.

2 BACKGROUND AND MOTIVATION

2.1 The rise of programmable network devices

From the end of Dennard scaling [44] over a decade ago, processor design has shifted from increasing clock speed to increasing parallelization. Building upon this, another more recent trend shows drastic change of datacenter hardware towards more specialized architectures, creating end-to-end heterogeneous systems optimized for specific workloads [79]. In this ecosystem, programmable network devices have arisen to enable offloading of custom logic to both the core of the network (e.g., programmable switches [47]) and endhosts (smartNICs [84]). Among the latter devices, FPGA-smartNICs offering full-fledged hardware customization have become widely available in the market and in the cloud, with major providers such as Alibaba [9], Amazon [12] and several others [15] exposing them to application developers. Nano-consensus exploits the design of Altera's infrastructure processing unit (F2000X-PL) [37], using the on-board FPGA to implement a high-performance consensus protocol, as in related works [10, 39].

2.2 Accelerating consensus in datacenters

Consensus is the problem of reaching an agreement among multiple processes – often regarded as the most important in distributed systems [83]. Its long research history has produced countless algorithms which are at the heart of critical datacenter services, namely state machine replication (SMR), which ensures that a replicated system is available and consistent even if some servers fail. This is typically achieved by establishing a majority quorum of size $f+1$ where f is the maximum number of tolerated failures and $n = 2f + 1$ is the minimum number of replicas needed. Established quorum-based consensus algorithms such as Paxos [50] and

Raft [64] ensure that a system is safe and live in the partially-synchronous model where messages have to be eventually delivered (i.e., can be arbitrarily delayed until a global stabilization time). These guarantees come at the cost of increased communication complexity. In widely adopted variations of Paxos and Raft, a decision can be taken only as quickly as a quorum round-trip allows. This performance overhead is often so high that practical systems loosen consistency guarantees, (e.g., Redis default replication uses best-effort broadcast [70]). To address the problem, state-of-the-art solutions exploit kernel bypass techniques to reduce packet processing overhead and accelerate consensus. These techniques include RDMA [7, 31, 40, 42, 67], DPDK [48], eBPF XDP [86] and custom network stacks [41, 46]. In this context, few emerging consensus implementations with software/FPGA-smartNIC co-design [10, 14, 36, 39] offer significant performance improvements compared to software-only counterparts. Besides increased performance, we show how FPGA-smartNICs offer extremely stable and predictable processing times [73] and have a negligible impact on tail-latency compared to sources of interference at the host (cf. § 6.2).

3 DESIGN

Nano-consensus provides a novel uniform consensus and replication engine designed for a cluster of FPGA-smartNICs in a datacenter.

3.1 System model

For a long time, all distributed systems have been communally considered to be *asynchronous*, i.e., devoid of upper bounds on communication and processing delays, confounding all kinds of setups including local-area vs wide-area deployments, wired vs wireless communication, stationary vs mobile hosts, etc. There have been significant improvements on all fronts since, and more differentiation between setups. Several systems have thus started to go against that common wisdom including

- distributed datacenter services, e.g., reliable failure detectors [72], low-latency messaging systems [41] which achieve bounded interaction *in software* through complex OS instrumentation, priority networking, and traffic engineering;
- disaggregated memory (DM) replication frameworks assuming the process fail-stop model [76], which implies perfect failure detection and thus synchrony, as well as reliable networks [32, 51, 53, 87];
- recent works [4, 33, 60, 61] on distributed process coordination in the context of Byzantine failures [49] which (similarly to DM works) take upper time bounds for commodity hard- and software as a given without introducing any system support to enforce these assumptions, and disregard network security concerns like (distributed) denial of service attacks that one could expect with such security-sensitive deployments;
- replicated variants of widely-used distributed applications such as Kafka [1], Redis[70], and PostgreSQL [2].

Nano-consensus goes a step further by fully exploiting network hardware support to *actively* clamp down on nondeterminism that

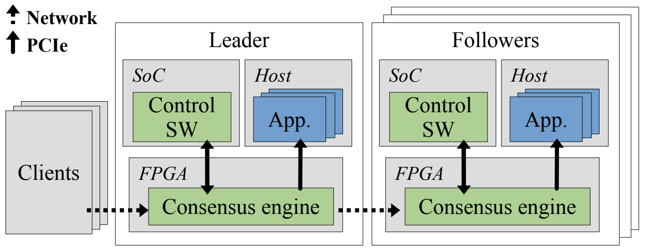

Figure 1: Nano-consensus architecture across multiple nodes.

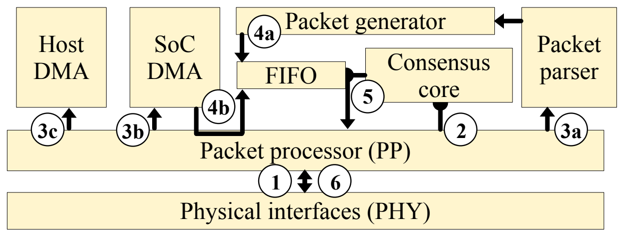

Figure 2: Data flow through the hardware modules of the consensus engine.

can hamper upper time bounds, in the context of benign process crash failures. Unlike current synchronous software frameworks [41, 72], which need disruptive, complex and error-prone OS configurations risking to compromise the integrity of claimed synchrony, Nano-consensus achieves better performance, determinism and usability by running in a self-contained module on widely-available FPGA-smartNICs. Tab. 1 shows that hardware reaches *2 orders of magnitude* better stability in our long-running benchmarks (cf. § 6 for setup). In short, Nano-consensus 1. runs as a core coordination service on highly precise FPGAs of smartNICs to achieve timely guaranteed process response times, 2. leverages traffic engineering and priority scheduling and queuing for entirely avoiding packets drops due to congestion thus ensuring timely predictable packet delivery, and 3. uses redundant communication for shielding remote process interaction from transient network link or switch failures. As a consequence, Nano-consensus can rely on low, timely bounded response times for its own processes and on timely bounded reliable (multicast) communication between them.

Note that other processes execute and communicate as normal, and priorities ensure that network resources are not lost in absence of actual Nano-consensus traffic. Nano-consensus interaction is set up through a controller which is not a limitation for the targeted service and respective applications, as these are typically long-running services such as key-value stores.

3.2 Architecture

System overview. Fig. 1 shows Nano-consensus's architecture in a typical multi-node deployment. A node can be either a *leader* or a *follower*, with only one leader being present at any time. During normal operation, requests from clients are forwarded to the leader which starts a consensus instance and communicates with the followers through the datacenter network. Our system is fault-tolerant and ensures that a new leader is elected in case the old one is faulty. Each node executes an exact replica of Nano-consensus and is divided into three sub-systems with separate computational units: a host server, an system on a chip (SoC) and an FPGA. The latter two are connected via peripheral component interconnect express (PCIe) and physically placed inside an off-path smartNIC, which is itself attached to the host server via PCIe. Nano-consensus core logic is the consensus engine, which is entirely offloaded to the FPGA hardware and directly attached to the network. The SoC contains a control software layer which allows for dynamic reconfiguration and monitoring of the consensus engine. Applications using Nano-consensus are running on the host in isolation, only receiving the outcome of a consensus instance. This architecture is ideal to ensure separation of concerns, leaving the host CPU free from costly packet processing. However, it could be adapted to on-path smartNICs (e.g. FPGA only, no SoC) and other card designs with minor modifications.

Consensus hardware engine. Fig. 2 shows a more detailed view of the consensus engine, outlining its components and a sample data flow. Incoming packets are sent from the physical interfaces to the programmable packet processor (PP) ①, which provides layer 2 and layer 3 switching functionality. When the PP matches Nano-consensus packets immediately trigger a notification to the consensus core module ②, needed for the correct functioning of the consensus algorithm. The PP then routes packets to one or more modules based on the Nano-consensus control settings and the payload content. One possible path is through the packet parser (3a); the relevant information is stripped and sent to the packet generator, which assembles a new packet with software-configurable headers + payload and queues it in the FIFO egress (4a). This data path is executed exclusively in hardware, enabling packet processing and consensus logic at line rate. The second possible path is through the SoC: the PP forwards packets to the control software (3b) which processes and queues them in the egress FIFO queue (4b). This mode enables operation when the hardware path is switched off, and is particularly useful for batching and intermittent operations. The third data path directly forwards packets to the host either to propagate the result of a consensus instance to the application or as a passthrough for generic packets (3c). Finally, ⑤ the consensus core triggers send notifications at regular intervals, dequeueing packets from the FIFO queue, passing them to the PP and sending them to the wire ⑥. This rate limiting functionality is core to our consensus algorithms in § 4.

Communication. Nano-consensus uses UDP with IP multicast for communication, combined with several other techniques to increase reliability. Packets are either created in software and piped to the egress FIFO queue or a base packet is pre-filled and written to the FPGA memory to be modified by the packet generator. Each process participating in consensus can belong to groups identified by unique multicast addresses, where processes can dynamically join/leave. UDP greatly simplifies hardware complexity compared to a TCP stack and avoids re-transmission overhead which would increase jitter, hampering interaction stability. Nano-consensus rate-limits traffic inside multicast groups to prevent packet drops due to congestion which, alongside redundant links to cater for several network failures, make the probability of packet loss so low that it can be considered negligible for realistic uptimes, as discussed shortly.

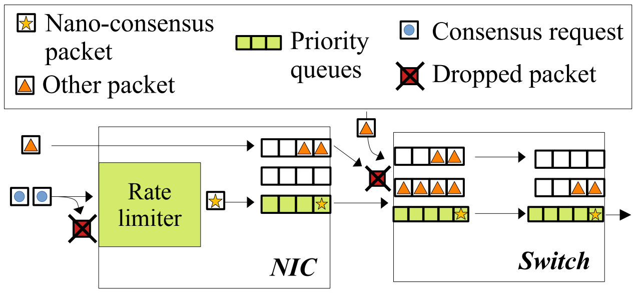

Figure 3: Nano-consensus sample communication in a section of the network. Consensus requests from clients and other packets can be dropped while Nano-consensus makes its packets never exceed network capacity.

3.3 Stable interactions on network hardware

We substantiate how stable interactions (intended as communication + processing latency) can be achieved inside datacenters by exploiting the predictability of programmable network hardware, leading to reliable, low upper time bounds on message delivery (>100× smaller than standard software configurations, cf. Tab. 1).

Deterministic processing in network hardware. The traditional network stack is notoriously a bottleneck for high-throughput applications, leading to poor, unpredictable packet processing latency. A large number of software approaches propose optimized data paths [41, 46, 85] and scheduling policies [28, 58] to mitigate this issue, achieving very low latency below some $99.x^{\text{th}}$ percentile or claiming upper time bounds on communication for a specific system setup [41]. Even if to a smaller extent, such systems still suffer from consequences of the many sources of interference at the end-host [41], causing costly context switches and related latency spikes. Nano-consensus overcomes software unpredictability by offloading all latency-critical logic to custom hardware on FPGA-smartNICs which allows direct access to packets from the wire, bypassing interference otherwise induced by resource contention in the PCIe bus and OS. The resulting custom circuit is dedicated exclusively to processing Nano-consensus packets as soon as they arrive, leading to ultra-low, predictable, stable latency. Low-jitter processing is also exhibited by similar hardware algorithms [10, 39] which, however, focus only on low latency and use it to accelerate Zab [45] and Raft respectively. A major limiting factor in FPGA-smartNICs algorithms is the frequency at which the hardware can deterministically process packets. In Nano-consensus this is given by the consensus core module synthesised on Altera F2000X-PL which can handle up to 122 million packets per second (Mpps), bandwidth which is largely sufficient to saturate 100Gbps networks and beyond.

Controlled priority traffic for zero-congestion. The other cause of unpredictable communication latency peaks and packet drops is congestion due to traffic bursts exceeding network capacity. Fig. 3 shows how Nano-consensus prevents congestion by using traffic engineering (TE) techniques to configure the network, while other packets sent through the usual best-effort path where they can be dropped or delayed. In short, we reserve highest-priority queues in the hops connecting Nano-consensus nodes and apply rate limiting at the NICs through the consensus core module (see Fig. 2). If the input frequency of consensus requests coming either from clients or the SoC is higher than the rate limiter allows, packets are dropped before starting a new consensus instance. We rely on client retransmission for such cases. When requests go through the leader node, they are multicast to the followers via highest-priority queues at a rate which is pre-fixed below network capacity. Our TE approach also limits aggregation of multiple flows that might overfill queues, in case non-fully overlapping Nano-consensus groups are deployed. This is in practice achieved by decreasing the frequency of the rate limiter of every node, as in previous work [30, 41, 72].

Scalability. Our system can also scale out without downtime providing very strong availability (see § 6.4) since initialization (TE setup and state transfer) can take place in the background (see § 5.2). Multiple applications can use a single Nano-consensus module given they share the same set or a subset of processes, with the only limit being rate limiter. For different overlapping sets of processes, applications are required to use separate Nano-consensus modules (one per set), which can be co-located on the same FPGA up to ~100 modules with additional multiplexing logic.

Redundancy and safe exit for network failures. If unhandled, network link or switch failures could compromise interaction stability and make a remote process falsely suspect a failure of the sender. This in turn could affect the consensus algorithm's safety (cf. § 4) if network failures partially affect Nano-consensus's multicast (e.g., a link fails and only a subset of remote processes receives a packet while others suspect the sender to have failed). Nano-consensus exploits physical redundancy, commonly available in datacenter network topologies, e.g., fat-trees, to tolerate a number of network failures. Since Nano-consensus nodes send exact copies of a packet over every redundant multicast tree, they can detect a network failure when they receive a smaller number of packets than the number of trees, e.g., receiving only one packet with a twofold redundancy. Once a network failure detection occurs, all Nano-consensus nodes employ a *safe exit*, i.e., they simply stop processing consensus messages, to prevent additional network failures from causing false leader suspicions. Nano-consensus can be additionally set to use network recovery mechanisms introduced in F10 [56], Hydra [18], and FiDe [72] to quickly establish alternative redundant paths.

Reliability of stable interactions in perspective. Predictability of modern network devices alongside established queuing engineering techniques make the probability of a message being lost or delayed beyond a conservative upper time bound so low in practice that it can be considered negligible. In our extensively tested setup (§ 6.2), this probability is 8.7×10^{-12} in the worst case, i.e., *once every 40 days* with an average throughput of 400Mbps. As discussed, multiple network failures affecting all redundant paths at the same time can break synchronous interactions. However, this is extremely unlikely in practice: recent estimates [72] based on popular real-world network failure statistics [29] (section C) give a worst-case probability of any two link or switch failures to occur *once every 6 years* in a 3-tiered fat-tree topology. Also, inconsistencies to the consensus algorithm caused by multiple simultaneous failures or delays occur only if these manifest in specific combinations (further reducing probability of a safety violation) and can be prevented by deployment, as we discuss in § 4.2. To put things into perspective, TCP – widely regarded as reliable and used by several systems for critical coordination as such (e.g. Zookeeper [35]) — has a higher probability of a packet corruption going undetected. A popular study [77] on a smaller sample base than in our experiments reports that "the Ethernet CRC + TCP checksum will fail to detect errors for roughly 1 in 16 million to 10 billion packets" (10^{-10} probability in the best case, e.g., *once every 2.9 days* with an average throughput of 400Mbps), with recent real-world cases being reported [80, 81].

4 QUORUM-LESS OPTIMAL CONSENSUS

This section outlines the algorithm behind the consensus core module (Fig. 2). Unlike the vast majority of classical and modern consensus algorithms adopted in real-world deployments, Nano-consensus does not require a majority quorum of processes to operate, leading to a *simple* solution with *optimal* message complexity. This novel approach is made possible by exploiting upper time bounds of Nano-consensus's system-supported stable interactions inside the datacenter (cf. § 3.3).

4.1 Reliable sychronous broadcast (RSBCAST)

Properties. We define the primitive RSBCAST (which we use in Alg. 3) to denote Nano-consensus's communication mechanism using timely interactions, IP multicast, redundancy and safety backstop. The primitive has an homonymous RSBCAST downcall, DELIVER and NET-FAULT upcalls. The interaction latency between SEND and RECV events has upper time bound $\Delta_I + \Delta_D$, i.e., the sum of the maximum processing and communication latency (Δ_I) and the maximum bounded clock drift (Δ_D) between any two nodes. RSBCAST has the usual properties of uniform reliable broadcast (cf. [16]) plus an additional property that messages which are RSBCAST are DELIVERed within $\Delta_I + \Delta_D$.

RSBCAST (Alg. 1). The sender sends a packet to every redundant IP multicast group $G_i \in \mathcal{G}$; these groups correspond to disjoint trees connecting the same set of Nano-consensus nodes. A receiver starts a timer once it receives the first packet: if less than $|\mathcal{G}|$ packets arrive within $\Delta_I + \Delta_D$, it means that either a network failure or packet drop has compromised the reliability of the channels. In that case (line 12-13), the upper layers are notified through the NET-FAULT upcall. Otherwise, when a receiver RECVs a message from all redundant multicast trees, RSBCAST safely DELIVERs the message. The correctness of RSBCAST's uniform delivery is probabilistic as a delay or $|\mathcal{G}|$ network failures could violate synchrony and potentially affect safety. We show how this is extremely unlikely in our system (cf. § 1,§ 3.3,§ 6.2) and discuss how only certain combinations of failures impact the correctness of consensus in the next section.

Algorithm 1: Reliable sychronous broadcast (RSBCAST)

```
1  $\mathcal{G} \leftarrow \{G_1, G_2 ... G_n\}$            // redundant multicast groups
2  $received \leftarrow \emptyset$
3  to RSBCAST$(m)$:
4      foreach $G_i \in \mathcal{G}$ do
5          SEND$(m)$ to $G_i$
6  upon RECV$(m)$ from $G_i$:
7      if $received = \emptyset$ then
8          START$(timer, \Delta_I + \Delta_D)$
9      if $received = \mathcal{G}$ then
10         DELIVER$(m)$
11     $received \leftarrow received \cup \{G_i\}$
12 upon TIMEOUT$(timer)$:
13     NET-FAULT
```

4.2 Consensus core: LOWI

Nano-consensus's core engine is optimized to execute multiple consensus instances in series which can be easily used to implement state machine replication (SMR), i.e., the most common use case for consensus. We first specify our consensus algorithm dubbed one-way imposition (OWI) in Alg. 2, then introduce its looped variant for SMR in Alg. 3, dubbed looped one-way imposition (LOWI).

Properties. We define the uniform consensus specification below. The primitive has a PROPOSE downcall and DECIDE upcall. OWI satisfies the following properties:

Validity: If a process calls DECIDE with value v, then v was PROPOSED by some process.

Integrity: No process does DECIDE twice.

Termination: Every correct process eventually does DECIDE some value.

Uniform agreement: No two processes DECIDE different values.

We assume that PROPOSEd values are distinguishable, e.g., representing requests or messages with unique identifiers, hence preventing duplicate delivery. Like in § 4.1, the upper time bound Δ_I denotes the maximum interaction latency between any two nodes. Furthermore, all processes start within a fixed time Δ_W which is the initial synchronization time. We discuss how synchronization is implemented in practice in § 5.2. All handlers are assumed to execute uninterrupted; if multiple handlers are enabled at the same time, they are triggered in a fair manner.

OWI (Alg. 2). At initialization, every process is assigned an ID from 1 to n to establish a hierarchy, with the leader being the process with the smallest ID (line 1) and other processes being followers. We use the variable $self$ to refer to the process ID of the executing process. When the leader is correct, it uses RSBCAST to reliably disseminate a value chosen deterministically (with DET()) from the *proposals* set (lines 4-7). All processes DECIDE on the value imposed by the leader (lines 19-20). Note that a leader may or may not have received proposals sent from other processes (line 8) depending whether it RECVs them before starting to PROPOSE. This does not affect consensus properties since the *proposals* set contains at least the leader's value (preserving *Validity*) and once a value is proposed, RSBCAST guarantees that it is DELIVERed by all correct processes within Δ_I. The timeout of $\Delta_W + \Delta_I + \Delta_D$ (line 9) ensures that, if the leader is correct, every correct process receives its imposed message through RSBCAST before triggering leader election (lines 10-12). If the leader fails before triggering RSBCAST, all processes will deterministically elect the next leader in the new round (i.e. at TIMEOUT, lines 10-12). The algorithm proceeds until at least $n-1$ timeouts (i.e. rounds) in the case $n-1$ failures occur and the process with the largest ID is the only correct process. Processes safely quit if RSBCAST signals a network fault (NET-FAULT) since they cannot know whether all processes have received the leader's decision.

Algorithm 2: One-way imposition (OWI). Uses RSBCAST. All processes call PROPOSE within a fixed time window Δ_w.

```
1  $leader \leftarrow$ MIN$(P)$
2  $proposals \leftarrow \emptyset$
3  $decision \leftarrow \bot$
4  to PROPOSE$(v)$:
5      $proposals \leftarrow proposals \cup \{v\}$
6      if $self = leader$ then
7          RSBCAST(DET$(proposals)$) // pick DETerministically
       else
8          SEND$(v)$ to $leader$
9          START$(timer, \Delta_W + \Delta_I + \Delta_D)$
10 upon TIMEOUT$(timer)$ and $decision = \bot$:
11     $P \leftarrow P \setminus leader$
12     $leader \leftarrow$ MIN$(P)$
13     if $self = leader$ then
14         RSBCAST(DET$(proposals)$)
       else
15         SEND$(v)$ to $leader$
16         START$(timer, \Delta_W + \Delta_I + \Delta_D)$
17 upon RECV$(v)$:
18     $proposals \leftarrow proposals \cup \{v\}$
19 upon DELIVER$(v)$:
20     if $decision = \emptyset$ then
21         DECIDE$(v)$
22         $decision \leftarrow v$
23 upon NET-FAULT$(timer)$:
24     QUIT
```

LOWI (Alg. 3). The full algorithm runs a series of synchronized (within Δ_W) consensus instances, takes REQUESTs from clients, PROPOSEs them through OWI, and stores them into the *decisions* log. Processes take REQUEST values asynchronously and store them into the *pending* variable. In every synchronous round of the LOOP, processes check whether the current consensus instance has ended with a decision through the *cins* and *lins* variables, which respectively indicate the current consensus instance and the last consensus

Algorithm 3: Looped one-way imposition (LOWI). Uses OWI. All processes start within a fixed time window Δ_w

```
1  $pending \leftarrow \emptyset$
2  $decisions \leftarrow [\ ]$
3  $cins \leftarrow 0$                    // current consensus instance
4  $lins \leftarrow 0$       // last consensus instance with a decision
5  to REQUEST($v$):
6      $pending \leftarrow pending \cup \{v\}$
7  upon LOOP and $cins = lins$:
8      $cins \leftarrow cins + 1$
9      if $pending \neq \emptyset$ then
10         PROPOSE($v$) | $v \in pending$
11         $pending \leftarrow pending \setminus \{v\}$
       else
12         PROPOSE($\perp$)                 // heartbeat value
13 upon DECIDE($v$):
14     if $v = \perp$ then
15         $cins \leftarrow lins$    // no decision, rerun this instance
       else
16         $decisions[cins] \leftarrow v$
17         $lins \leftarrow cins$              // go to next instance
18 upon QUIT:
19     exit LOOP
```

instance in which processes DECIDED (line 7). Note that OWI guarantees that all DECIDE in the same round even in the occurrence of failures, preserving synchronization across multiple consensus instances. In a "proposal" round, processes either PROPOSE a value from the *pending* queue, or PROPOSE a heartbeat value (lines 7-12) in case there are no pending *proposals*. This is a key mechanism preventing non-leader nodes to time out on correct leaders. As a consequence, processes might DECIDE heartbeat values, in which case LOWI makes sure to repeat the same consensus instance (lines 13-15). Decisions of values coming from client requests are instead added to the *decisions* log, and the algorithm moves on with the next instance (lines 16-17). Note that Processes exit the LOOP when OWI calls QUIT in consequence of a NET-FAULT.

Correctness. For *Validity*, OWI ensures that processes DECIDE values from the leader's *proposals* set (Alg. 2, line 5). In LOWI, these can be either heartbeat values ($\perp$) or values previously PROPOSED. Only the latter are stored in the *decisions* log (Alg. 3, lines 14-15). *Integrity* is guaranteed by line 20 in Alg. 2. Regarding *Termination*, the TIMEOUT mechanism in OWI ensures that a correct leader will propose within $n - 1$ rounds at most. To show *Uniform agreement* we argue that only one process at any time can RSBCAST, i.e., there can only be a leader for every consensus instance. This is ensured by the timing guarantees of our system, namely upper-bounded interaction latency Δ_I among Nano-consensus nodes, and the initial synchronization window Δ_W. The TIMEOUT of $\Delta_I + \Delta_W$ (Alg. 2, line 9), and the reliability of RSBCAST ensure that either all processes DELIVER within a given time before the timeout, or the sender has crashed hence the message will never be received.

Complexity and optimizations. OWI solves consensus in 1 message delay in failure-free executions, but requires initial synchronization to make sure all replicas start within Δ_W, which in practice consists in at least another message delay. LOWI mitigates this by synchronizing nodes only once at the beginning of the consensus series, then maintaining synchronization by running back-to-back consensus instances. This is the key mechanism to bridge asynchronous client requests with efficient synchronous algorithm execution, using heartbeats to maintain synchronization when requests are delayed. OWI is more efficient than classical synchronous consensus by Lynch [57] in which processes disseminate $O(n^2)$ messages per round and different correct processes might decide in different rounds, not allowing for our "looped" optimization. To mitigate excessive heartbeat traffic, *pending* proposals can be accumulated by setting a loop period higher than the incoming proposal rate and vice versa when a large number of proposals need to be "consumed". While likely only the leader ends up proposing its values in OWI, this is a common feature of most consensus/SMR algorithms [7, 10, 31, 86] and is mitigated by directing requests only to the leader. DET in Alg. 2 chooses non-heartbeat values for efficiency.

RSBCAST and multiple network failures. RSBCAST mitigates network failures with redundancy, leading to an extremely low probability (at worst once every 6 years, cf. § 3.3) of *any* multiple network failures to affect all redundant trees. In reality, the actual probability of such failures affecting consensus safety is even lower: only network failures which create network partitions (whether transient or not) are critical. Furthermore, this risk is null when redirecting client request to a leader and not to replicas which is a common deployment adopted in Paxos/Raft -based algorithms [7, 10, 48, 86]. As long as the leader continues its operation, no requests are forwarded to other replicas, hence preventing *uniform agreement* breach by deployment.

5 IMPLEMENTATION

5.1 Development

We designed and implemented Nano-consensus on the F2000X-PL Altera smartNIC [37], leveraging the on-board SoC and Intel Agilex 7 FPGA [6] for software/hardware co-design. The built-in PP (refer to Fig. 2) was programmed to provide ingress packet classification and routing among the physical ports, the SoC, the host and our custom hardware module. The latter was compiler onto the board's application stack acceleration framework (ASAF) module and amounts to a total of 803 lines of code in System Verilog and VHDL. Iterative testing for the RTL module was done with ModelSim [63]. We developed the following software components to complement the hardware module: a Rust CLI app for controlling and monitoring (383 lines of code), PP Bash configuration scripts (142 lines of code) and a high-performance thin layer in eBPF's XDP to provide application specific logs (412 lines of code).

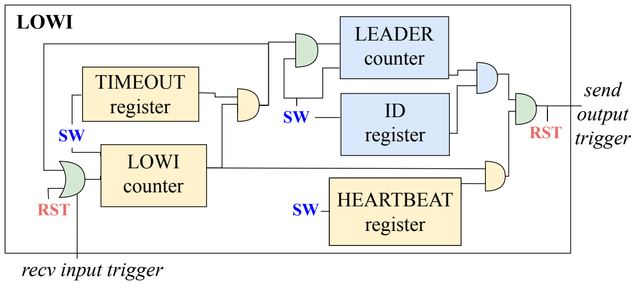

Figure 4: Simplified circuit schematics of LOWI. "SW" labels define components configurable from software.

5.2 LOWI system integration

Fig. 4 shows how LOWI is synthesized on the consensus core module and interacts with the other modules via triggers. In the following we describe how LOWI is integrated in our design on Altera F2000X-PL FPGA-smartNICs. Please see § 3 for architecture details.

Initial synchronization. The start of the sequence simply consists in a "start" message which the designated leader send to the followers in order to activate their receive input trigger, which resets and start the LOWI counter. The timeout register is previously set to at least double the maximum latency observed in the system plus the maximum clock drift, i.e., the initial synchronization window $\Delta_W = \Delta_I$, hence TIMEOUT duration $> 2\Delta_I + drift$. We rely on modern FPGAs' very robust clocks with negligible bounded drift (as assumed by recent systems, e.g., [31, 41, 72]).

Main operation. After setting FPGA memory and registers from software, clients send consensus requests to the leader node directly on the FPGA-smartNIC as typically done in other consensus setups, e.g., Waverunner [10]. The on-board PP routes requests towards the packet parser (cf. Fig. 2), which extracts the payload and passes to the packet generator which prepares a PROPOSE packet and enqueues it into the FIFO. Once the LOWI counter of the leader triggers the next send signal (AND logic comparison among LOWI counter, blue registers and heartbeat register in the circuit), the PP is set to RSBCAST the decision to the followers, the host DMA and as well as replying directly to the clients with an "OK" response. This ensures minimal server response latency since the packet has to only do a loopback inside the FPGA. Packet reception at the followers (RECV) triggers LOWI, resetting its counter and preventing it from reaching the value of the timeout register previously set from software. The packet is forwarded up the host DMA (i.e., via DECIDE) without further processing. Note that the reception of requests at the leader does not trigger a LOWI receive as it would compromise the timeliness of the loop period. The LOWI counter is reset after every successful send, creating the loop. Note that the loop period (i.e., heartbeat register in Fig. 4) might be set as small as the FPGA clock allows (~8ns) order to keep multiple consensus instances in flight for very high throughput.

Heartbeats and leader election. The LOWI circuit triggers a send signal even if there are no incoming requests and the FIFO is empty. In such case, a pre-filled, static heartbeat packet (⊥ in Alg. 3) is sent instead from memory, ensuring that followers never timeout on a leader that is still alive. Upon leader failure, the LOWI counter in each follower hits the timeout value, outputting a signal which increases the leader counter. This constitutes our deterministic leader election, since we assign monotonically increasing IDs to all processes during the initial configuration. The follower with ID = leader ID + 1 is elected as next leader and starts LOOP.

Flawless joining in state machine replication. New processes can join an existing cluster as followers without disrupting an on-going execution. At the start of a join procedure, a follower sets itself in a "joining" state, enables the consensus engine, and starts logging the decision values of the running consensus instances. In parallel, the control software sends a request to any other node asking for the current state and the latest consensus instance it has recorded. After the state transfer, the new node will apply previously recorded changes starting from $c + 1$ and set itself as "active". Before this last step it might be necessary to increase the timeout value in case the follower is "further away" than any other followers in the group. This can be achieved by the software layer through a new consensus instance. Reducing the timeout value would require halting the system, but it is unlikely required since high timeout values do not impact failure-free performance.

6 EVALUATION

We evaluate Nano-consensus by comparison with state-of-the-art services and applications, addressing four research questions:

RQ1: How stable and reliable are Nano-consensus remote process interactions?
RQ2: How well does Nano-consensus perform?
RQ3: What is the impact of failures and joins on Nano-consensus availability?
RQ4: How and by how much does Nano-consensus improve real-world applications?

6.1 Methodology

Evaluation cluster. We implement and evaluate Nano-consensus in a production datacenter of a major cloud service provider. The evaluation cluster consists of 3 Altera F2000X-PL [37] attached to 2 Dell R740 servers, each equipped with 2 Intel Xeon Gold 6138 at 2.00GHz (40cores, 80 threads) and running CentOS 8 [17]. F2000X-PL's SoCs are equipped with Intel Xeon D-1736 CPUs at 2.30GHz (8 cores, 16 threads) running Rocky Linux 8.9[71]. For most tests, we connect both 100Gbps QSFP28 ports of every F2000X-PL to a 100Gbps TOR switch. To benchmark other applications, we use Mellanox ConnectX-4 100 GbE [59] (making sure that bandwidth is not limiting maximum achievable throughput), connected to the same switch. For multi-switch evaluation we use an additional cluster of 2 Cloudlab [26] xl170 servers, connecting their ConnectX-5 NICs to a number of Dell s4048 type switches, as outlined later.

Comparison. We compare Nano-consensus with three different OS-bypass datapaths, namely FPGA-smartNICs, RDMA, and eBPF XDP, chosen for their relevance and widespread adoption. For each technology, we compare Nano-consensus with the following state-of-the-art consensus systems:

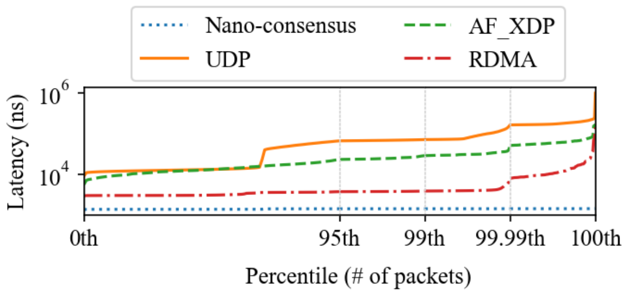

Figure 5: Maximum interaction (communication + processing) latency over 40 days (115 billion packets).

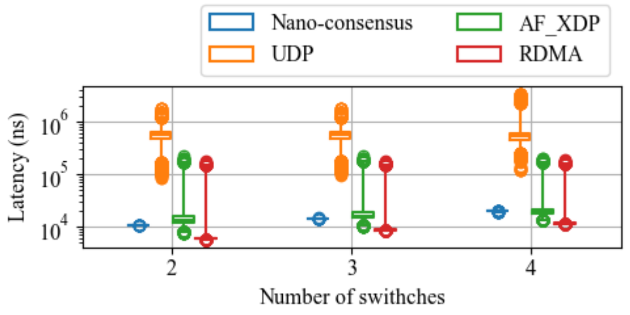

Figure 6: Interaction latency across multiple switches. Horizontal dashes represent median values, whiskers 0.01th and 99.99th percentile latency, circles are outliers.

Waverunner [10], hardware-accelerated Raft on Alveo U280[11] FPGA-smartNICs, the fastest SMR module to our knowledge;

Electrode [86], XDP-based consensus service used to implement viewstamped replication [54];

Mu [7], a state-of-the-art RDMA-based consensus algorithm.

We also compare the integration of Nano-consensus into two widely used applications: Redis [68], a distributed key-value store and Zookeeper [35] against their native performance. For Redis, we also compare against RedisRaft [69], an official module developed by RedisLabs using Raft for SMR.

6.2 RQ1: interaction stability

Our first experiments evaluate stability and reliability of time bounds on interaction latency, a core aspect of Nano-consensus's design.

Long-running latency. This benchmark consists in running a simple ping-pong protocol between the leader and the two followers and computing the worst leader-to-follower, one-way latency for every round. We run our benchmark for *40 consecutive days* collecting measurements for a total of 115 billion packets, more than 650× the amount of similar stability evaluations [41] and 56× the amount used in a widely-cited TCP reliability study [77]. Nano-consensus packets are sent at a constant throughput of ∼400Mbps through Nano-consensus's PP. For each packet, we log both Nano-consensus latency and UDP software latency at the SoC, in order to show the cost of traversing the network stack. We also evaluate latency of RDMA Unreliable Connection (UC, UDP equivalent) and XDP XSK sockets through the same ping-pong test, but using a much smaller sample base of 10 million packets (giving them an advantage over Nano-consensus). We use `stress-ng` [78] and `iPerf` [38] to generate periodic spikes of maximum CPU and network utilization. Fig. 5 shows the results. Nano-consensus exhibits the lowest and the most stable latency for 100% of the measured packets with an average latency of 1.41µs and a maximum of 1.48µs and *no packet loss.* All other approaches show a sharp increase in maximum latency at the tail of more than 100×, indicating the limits of software approaches beyond some 99.xth percentile, especially if we consider that they were run for a fraction of the time only. This benchmark showcases the interaction stability of modern datacenter network hardware, which makes the probability of failure (breaking an upper time bound) smaller than services that are widely considered reliable, e.g., TCP's detection of corrupted packets (TCP + Ethernet CRC checksum, see § 3.3).

Multi-switch and packet loss. We also run two additional microbenchmarks in the CloudLab [26] setup to evaluate the stability of hardware processing and Nano-consensus communication layer in the core of the network. We manually reserve the highest-priority queues in the switches and use Mellanox ConnectX-5 NIC hardware timestamp [62] to evaluate the FPGA-smartNIC datapath, which is equivalent to Nano-consensus timestamps in the consensus engine. We use the previously mentioned ping-pong applications with varying network load generated with `iperf3` [38]. For this experiment, the switches are connected in series with two nodes connected at opposite ends, so a packet must traverse all hops. Fig. 6 shows the effects on latency of scaling to 2, 3 and 4 switches under minimal network traffic. The figure confirms Nano-consensus's stability: each switch adds around 4µs latency, respectively 10.5 ± 0.15µs, 14.4 ± 0.2µs and 19.6 ± 0.25µs average with jitter (intended as max - min) within the error value. Other technologies show much higher jitter in the order of hundreds of µs. Interestingly, the average of RDMA and XDP is comparable or even smaller than Nano-consensus, which can be explained by the batching and interrupt coalescing adopted by the former two to save the CPU from costly frequent interrupts, resulting in peaks of latency followed by a long series of packet with latency that is shorter than the wire speed. However, given their high jitter, these approaches are unsuitable for practical synchrony. The second microbenchmark (Fig. 7) compares packet loss with varying network load between Nano-consensus path and the normal network path depicted in Fig. 3, in the 4-switch topology. Here we can observe the effect of highest-priority queue reservation and rate limiting: normal traffic is dropped by the switches and endhosts while Nano-consensus packets are never dropped.

6.3 RQ2: consensus performance

Our second set of benchmarks aims to evaluate latency of consensus instances at increasing throughput with a traffic generator to test the limit of our system. Since all compared approaches use a leader-based consensus algorithm, we measure the latency from when the leader accepts a PROPOSE request to when it DECIDE as

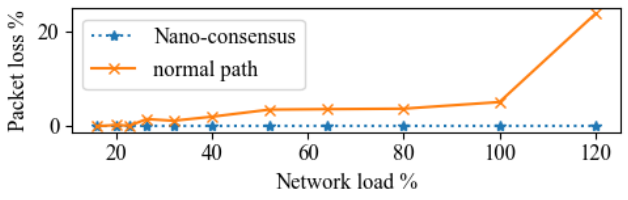

Figure 7: Comparison of packet loss between Nano-consensus packets and normal packets at increasing network load. Both flows go through the same NICs and switch. Thanks to rate limiting and traffic priority, Nano-consensus packets are never lost, even when the incoming traffic exceeds network capacity (120%).

well as the end-to-end latency at the client side for fair comparison (compared approaches are evaluated in the same manner). Clients send requests directly to the leader using small random packets – 50B for Nano-consensus and Waverunner and 64B for Mu and Electrode, disadvantaging the former two approaches for total goodput (i.e. payload vs packet headers). The packets include 44B of Ethernet, IP and UDP headers, chosen for fair comparison to Waverunner's evaluation [10]. Unlike Mu and Electrode, we do not deploy Waverunner due to the source code being unavailable. Instead, we report Waverunner's results from their paper [10], using an identical network setup, and only substitute the RTT of our system, leaving the throughput as it is. We include one consensus request per packet with minimal payload size, intentionally avoiding client-side batching (which would trivially increase throughput for all appreaches) to show the base performance of every approach.

Throughput and network utilization. Fig. 8 shows median throughput results in terms of consensus instances per s. Nano-consensus achieves 62.4 Mpps using the DPDK `testpmd` tool [25] with receive-side scaling over 4 SoC cores enabled in F2000X-PL for high- performance software reception. The value, limited to avoid packet loss in software, is lower than Nano-consensus Nano-consensus's core processing rate (121Mpps), i.e. the theoretical bottleneck of the system; fine-grained performance tuning of the DPDK `testpmd` is likely to further increase software throughput [24] up to network bandwidth saturation (100Gbps). Nano-consensus improves over Waverunner, the fastest consensus system in literature, by ~2.4×. With both systems using FPGA-smartNICs and only being limited by the processing speeds of the hardware modules and by the underlying network bandwidth, Nano-consensus's performance improvements come from its algorithmic advantage over quorum-based approaches. Waverunner uses Raft which adds $n - 1$ (vs 0 for Nano-consensus) acknowledgment messages for the leader, dramatically impacting its bandwidth usage in the network which Waverunner saturates at 1.25% goodput/single link utilization ratio, as shown in Tab. 2. Nano-consensus achieves a strongly increased goodput of 12% (~4.8×) with only twofold network redundancy, leaving additional room for 75.1 Gbps at the same rate, which could be achieved with client-side or server-side batching by increasing the payload by 150B (minus the additional Ethernet field – 7B preamble, 1B packet start delimiter and 12B inter-packet gap). Software-only approaches exhibit similar results as Waverunner

Table 2: Maximum goodput achievable with relative network utilization for small requests. We assume 2-level redundancy for both approaches.

	Packet size (44B header)	Maximum goodput	Leader bandwidth (200Gbps)
Nano-consensus	50B	2.99 Gbps	24.96%
Waverunner [10]	50B	1.25 Gbps	50%

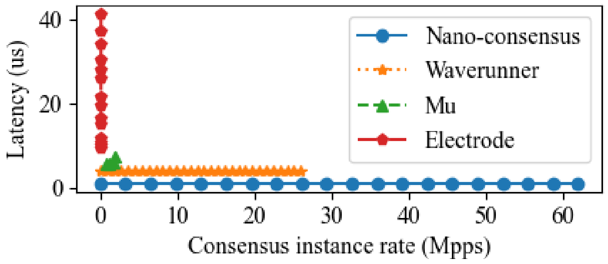

Figure 8: Consensus performance measured at the leader. Measurements at clients add an additional ~40µs for every approach due to client-server RTT.

since they are also using acknowledgements, and furthermore cannot saturate the network bandwidth with small requests (Fig. 8) since they limit software packet processing overhead with batching in order to keep CPU utilization low.

Latency. Nano-consensus exhibits a constant median latency of 1.03µs (Fig. 8), limited by the loopback speed of F2000X-PL NIC through our custom hardware module. It improves over by over ~3.82× w.r.t. Waverunner's constant latency, demonstrating the stability of FPGA-smartNIC-based hardware processing. Additional testing shows that packet sizes up to 1500B marginally increase the processing latency by 10%, while the 99th percentile latency is within 3% for hardware approaches. Mu and Electrode show 5.38× and 7.39× higher latency respectively and sharp increases when approaching their throughput limit, following a "hockey stick" pattern. For completeness, we performed the same measurements from the client side to obtain the end-to-end latency from a client perspective. Clients add an additional ~40µs on average for all approaches, corresponding to the RTT between the clients and the leader in our setup (both connected to 1 TOR switch).

6.4 RQ3: fault tolerance and availability

We evaluate Nano-consensus's fault tolerance by injecting failures at the leader of a cluster, which results in a leader election round involving *no explicit communication*; followers simply time out when they stop receiving messages from the leader (data or heartbeats). While follower failures do not cause downtime in any of the compared approaches, a substantial gain of Nano-consensus is that it tolerates $f = n - 1$ processes failures while compared quorum-based approaches require $2f + 1$ processes, as shown in Tab. 3. We evaluate the downtime impact of a new follower joining an existing cluster.

Table 3: Leader election and downtime resulting from a new follower joining an existing cluster. *Slanted* numbers are taken from respective publications and blank values were not addressed therein. Time values are in µs.

	Tolerated failures	Failover latency (50th/99th)	Downtime on node join
Nano-consensus	$n-1$	2/2	0
Consensus in a Box [39]	$\lfloor \frac{n-1}{2} \rfloor$	*60/60*	*2E5*
Waverunner [10]	$\lfloor \frac{n-1}{2} \rfloor$	*1E6/1E6*	*2E5*
uKharon [31]	$\lfloor \frac{n-1}{2} \rfloor$	*50/139*	

Our LOWI algorithm (Alg. 3) is designed for high-availability and can flawlessly react to changes in the cluster, as discussed below.

Leader election. Unlike common solutions, our timeouts are optimal as they can rely on the stable latency ($\Delta_I = 1.41$µs) from the communication layer and precision of the hardware sending rate ($T_{LOOP} = 8$ns), and therefore can be safely set at $\Delta_I + T_{LOOP}$ We choose 2µs in our experiments (including a safety margin), achieving a 30× improvement against Consensus in a Box [39] and 25× over uKharon [31], respectively the fastest hardware and software solutions to our knowledge, as revealed by Tab. 3. Nano-consensus also improves over Waverunner's conservative timeout by 500000×. However, we believe that Waverunner's performance is very likely to sustain more aggressive timeouts.

Follower joins. Another benefit of LOWI is that it enables followers to join an existing cluster without disrupting the ongoing operation. For this evaluation we use Nano-consensus to implement a basic SMR service which uses a leader to deterministically order incoming requests, run a consensus instance with the request number and writes it to a log. Tab. 3 shows that Nano-consensus achieves 0 downtime thanks to the parallel state transfer strategy described in § 5.2. Once again, this results in a substantial advantage over quorum-based approaches which need to halt operations and wait for a follower to synchronize its state with the leader (200ms in Waverunner and Consensus in a Box evaluations).

6.5 RQ4: real-world applications

Our final sets of benchmarks analyzes Nano-consensus's performance as part of Redis and Zookeeper. We evaluate latency and throughput from external clients connected to our 3-node cluster via a TOR switch. As for previous benchmarks, all requests are forwarded directly to the leader. For both applications, we build custom sequential (i.e. blocking on server response) clients which send Nano-consensus requests to the cluster, which our system safely replicates. The leader then acknowledges the request to the client and forwards to the respective hosts at the same time, allowing the application to process requests in the background. We use a thin XDP layer and receive-side scaling on the host to efficiently process incoming requests and log them into in-memory maps. We modify Redis and Zookeeper to read from these logs to process requests and refer to these versions as Redis-NC and Zookeeper-NC respectively. We evaluate only SET requests but not GET requests,

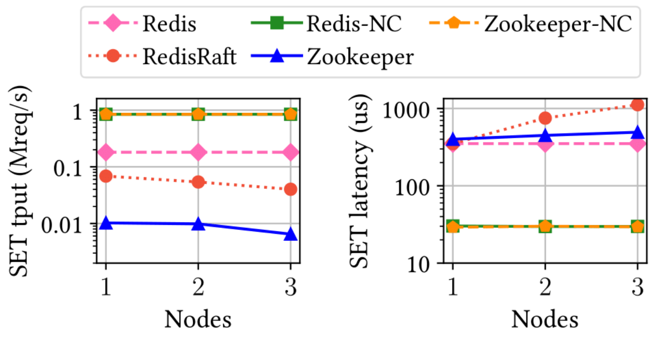

Figure 9: SET request latency and throughput of Redis and Zookeeper with different replication algorithms and numbers of nodes. y-axis is logarithmic.

as all approaches would simply return the requested value, adding no overhead to native performance.

Results. Fig. 9 depicts throughput (left) and latency (right) at small scale $n = 1, 2, 3$. (At larger scales the performance of compared approaches degrades quickly.) Nano-consensus applications outperform the native SMR approaches RedisRaft and Zookeeper by at least 10× throughput and at least >11.6× lower latency. Performance gains come from the fast hardware response directly from the FPGA-smartNIC which avoids traversing the server network stack, achieving improvements even on original Redis unreplicated by 4.7× throughput and 11.6× latency. Both Redis-NC and Zookeeper-NC show very close performance since they use the same custom UDP clients. These have suboptimal packet processing capabilities which limits maximum throughput to 846Kpps, leaving a lot of performance on the table as shown by the results achieved in § 6.3 with the DPDK test suite. Further engineering effort (e.g. using DPDK, RDMA) could easily bring the performance up to tens of Mpps as shown by several works [24, 28, 46]. The obtained results are a good representation of Nano-consensus benefits with an average programming effort. In addition, note that the fault tolerance of Nano-consensus is increased, so for instance to be able to tolerate $f = 2$ failure(s) the performance of Nano-consensus-based Zookeeper on $n = 3$ nodes would have to be compared to that of original ZooKeeper on $n = 5$ (cf. Tab. 3).

7 NANO-CONSENSUS IN THE BIG PICTURE

We position Nano-consensus in a broader context delineating where and how it should be used, its benefit-cost tradeoff and future work.

Application notes. Nano-consensus is designed for datacenters and relies on network predictability, programmability, and speed. It primarily targets highly available core services with high throughput and low latency requirements, but can be also used as accelerator to efficiently "consume" a large volume of accumulated requests when latency is not a primary concern. The latter scenario allows for high-level traffic scheduling based on network utilization, e.g., intermittent enabling of Nano-consensus engine to compensate external, bursty traffic, as well as mitigating the overhead caused by heartbeat messages. Nano-consensus is best used in combination with a high-performance packet processing software layer to avoid throughput limitation (e.g., see UDP clients in § 6.5) or cause unwanted packet loss. Several modern systems such as eRPC [46], DPDK [24], or eBPF (see server-side software layers in § 6.3 and § 6.5) can easily reach 10Mpps per core, especially considering that Nano-consensus takes away the send overhead of proposal messages. Additional engineering effort in implementing a batching module aggregating multiple requests together as done by Waverunner [10] would also greatly mitigate the pressure on the endhost.

Resource footprint. One of the main benefits of using smartNICs is alleviating the burden on the host CPU. Nano-consensus takes on packet processing and all consensus logic, leaving only asynchronous reception to the host (i.e. which can be deferred at later times hence relieved from latency and throughput requirements). In terms of FPGA footprint, LOWI uses less than 1% of the available resources in the F2000X-PL board, leaving room for 100 equivalent modules available for scaling up. Nano-consensus's highest-priority queue reservation can be shared with when using other concurrent high-priority services by decreasing Nano-consensus's sending rate. Moreover, explicit resources reservation is often intrinsic to the deployment of highly available core services[27, 88] to avoid co-locating too many other communication-intensive processes. The same applies for network redundancy, which is also commonly available by default in datacenter network topology (e.g., fat-tree). Benefits of redundancy and resource reservation can easily outweigh the cost of resource reservation, as shown in § 6.3.

8 RELATED WORK

Coordination with stable interactions. Traditionally, distributed systems are designed assuming that messages can be arbitrarily delayed by the network and the packet processing stack. This common belief is challenged by the rise of more programmable, precise, and high-performance networks and endhosts [73, 74]. A number of systems assume stable communication and stable processing as a given for, e.g., optimal weak failure detectors [8], leader election [75]. Several recent works assume synchrony in wide-area scenarios with Byzantine failures in the context of blockchains. BoundBFT [60] investigates to what extent synchrony can be violated in practice without hampering consensus correctness. AlterBFT [61] is a novel BFT consensus protocol which assumes synchrony for short messages only, significantly improving latency compared to fully synchronous protocols while retaining throughput and fault tolerance. Unlike nano-consensus, these and other BFT algorithms [5, 33, 55] assume a classical software stack without providing any concrete underlying system to enforce such assumptions. Seminal work on deterministic distributed processes was introduced by DDOS [34], however with significant overhead in its remote process interaction. X-Lane [41] introduces a communication layer relying on programmable networks and process isolation, resulting in upper time bounds in interaction which are used to accelerate a Raft-based consensus service. However, this approach – inherited by FiDe [72] for reliable failure detection – requires fine-tuning of the OS which can be easily misconfigured affecting interaction stability. Nano-consensus takes a step further by pushing logic to network hardware, dramatically reducing interaction jitter and using it to devise a custom consensus algorithms with optimal complexity.

Distributed algorithms on network hardware. Consensus in a Box [39] is a seminal work which pushes Zookeeper Atomic Broadcast (ZAB) off the critical path by fully porting it to FPGA. Waverunner [10], the most recent and fastest work to our knowledge, takes a different approach and moves only the failure-free operations of Raft onto FPGA-based smartNICs, leaving failover routines to the software. Paxos in the NIC [14] proposes a high-level abstraction to offload Paxos-like consensus algorithm on the NIC with quantitative analysis to prove its benefits and an early-stage prototype with limited evaluation. NanoPU [36] proposes a more generic datapath for low-latency communication by a custom communication channel which writes directly to the CPU registers, bypassing PCIe bus and OS jitter sources. While the authors simulate their design with Raft, NanoPU does not provide algorithmic novelty and does not provide a physical implementation. Thanks to its novel design on stable interaction and optimal algorithm (e.g., with respect to classical synchronous consensus [57]), Nano-consensus fully exploits hardware properties and uses an optimal consensus algortihm improving over Waverunner throughput, latency and availability.

Fast software packet processing. High tail-latency of a service can impact client retention and cause loss of revenue [19, 20, 82]. This led to the development of a plethora of systems which optimize tail-latency for datacenter remote procedure calls through specialized networking stacks at the endhosts [13, 21, 28, 46, 46, 52, 65, 66, 85]. These works achieve µs tail-latency through optimal endhost packet processing but fail to prevent outliers beyond the 99.xth percentile. QJump [30] leads the way to achieve minimal, stable tail-latency in networks but does not consider jitter at the endhost, leading to the same issue. Nano-consensus exploits a custom hardware design on FPGA-smartNICand reservation of priority queues in the network to achieve ultra-stable interactions and server-response with 100th percentile tail-latency of as low as 1.03µs, outperforming software state of the art in latency, throughput and failover time.

9 CONCLUSIONS

We propose Nano-consensus, a hardware-supported consensus engine which runs on FPGA-smartNICs. Unlike common approaches accelerating existing algorithms, Nano-consensus exploits and supports the stability of network hardware to introduce a novel consensus primitive which is efficient for series of consensus instances. Nano-consensus provides optimal message complexity and can run as fast as the underlying network allows with ns-scale latency, outperforming state-of-the-art hardware and software consensus implementation by 3.82× and goodput by 4.8×, substantially improving availability upon failures.

ACKNOWLEDGEMENTS

This work was supported by Swiss National Science Foundation grants #192121 and #197353, SAP grant DEDIC, Hasler Foundation grant DEEDS, and Meta Research in Distributed Systems program.

REFERENCES

[1] Apache Kafka Synchronous replication. https://cwiki.apache.org/confluence/display/kafka/kafka+replication#:~:text=In%20primary-backup%20replication%2C%20the,write%20to%20the%20remaining%20replicas. Online; accesses 10-Jan-2025.

[2] Synchronous replication in postgresql. https://www.crunchydata.com/blog/synchronous-replication-in-postgresql. Online; accessed 10-Jan-2025.

[3] IEEE 802.3. Ieee draft standard for ethernet amendment: Media access control parameters for 800 gb/s and physical layers and management parameters for 400 gb/s and 800 gb/s operation. *IEEE P802.3df/D3.0, July 2023*, pages 1–286, 2023.

[4] Ittai Abraham, Dahlia Malkhi, Kartik Nayak, Ling Ren, and Maofan Yin. Sync HotStuff: Simple and Practical Synchronous State Machine Replication. In *2020 IEEE Symposium on Security and Privacy (SP '20)*, volume 1, pages 106–118, 2020.

[5] Ittai Abraham, Kartik Nayak, and Nibesh Shrestha. Optimal good-case latency for rotating leader synchronous bft. In *25th International Conference on Principles of Distributed Systems (OPODIS 2021)*, 09 2021.

[6] Agilex™ 7 FPGA and SoC FPGA. https://www.intel.com/content/www/us/en/products/details/fpga/agilex/7.html. Online; accessed 14-Jul-2025.

[7] Marcos K. Aguilera, Naama Ben-David, Rachid Guerraoui, Virendra J. Marathe, Athanasios Xygkis, and Igor Zablotchi. Microsecond consensus for microsecond applications, November 2020.

[8] Marcos K. Aguilera, Carole Delporte-Gallet, Hugues Fauconnier, and Sam Toueg. On implementing omega in systems with weak reliability and synchrony assumptions. *Distributed Computing*, 21(4):285–314, 2008.

[9] Alibaba Cloud ECS. Deep Dive into Alibaba Cloud F3 FPGA as a Service Instances . https://www.alibabacloud.com/blog/deep-dive-into-alibaba-cloud-f3-fpga-as-a-service-instances_594057. Online; accessed 10-Jul-2025.

[10] Mohammadreza Alimadadi, Hieu Mai, Shenghsun Cho, Michael Ferdman, Peter Milder, and Shuai Mu. Waverunner: An elegant approach to hardware acceleration of state machine replication. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 357–374, Boston, MA, April 2023. USENIX Association.

[11] Xilinx alveo 280 product frief. https://www.xilinx.com/content/dam/xilinx/publications/product-briefs/alveo-u280-product-brief.pdf. Online; accessed 14-Jul-2025.

[12] Amazon EC2 F2 Instances. https://aws.amazon.com/ec2/instance-types/f2/. Online; accessed 10-Jul-2025.

[13] Adam Belay, George Prekas, Mia Primorac, Ana Klimovic, Samuel Grossman, Christos Kozyrakis, and Edouard Bugnion. The ix operating system: Combining low latency, high throughput, and efficiency in a protected dataplane. *ACM Trans. Comput. Syst.*, 34(4), dec 2016.

[14] Giacomo Belocchi, Valeria Cardellini, Aniello Cammarano, and Giuseppe Bianchi. Paxos in the NIC: Hardware Acceleration of Distributed Consensus Protocols. In *2020 16th International Conference on the Design of Reliable Communication Networks DRCN 2020*, pages 1–6, March 2020.

[15] Christophe Bobda, Joel Mandebi Mbongue, Paul Chow, Mohammad Ewais, Naif Tarafdar, Juan Camilo Vega, Ken Eguro, Dirk Koch, Suranga Handagala, Miriam Leeser, Martin Herbordt, Hafsah Shahzad, Peter Hofste, Burkhard Ringlein, Jakub Szefer, Ahmed Sanaullah, and Russell Tessier. The future of fpga acceleration in datacenters and the cloud. *ACM Trans. Reconfigurable Technol. Syst.*, 15(3), February 2022.

[16] Christian Cachin, Rachid Guerraoui, and Lus Rodrigues. *Introduction to Reliable and Secure Distributed Programming*. Springer Publishing Company, Incorporated, Heidelberg, Germany, 2nd edition, 2011.

[17] Centos - download. https://www.centos.org/download/. Online; accessed 14-Jul-2025.

[18] Inho Choi, Ellis Michael, Yunfan Li, Dan R. K. Ports, and Jialin Li. Hydra: Serialization-Free network ordering for strongly consistent distributed applications. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI '23)*, pages 293–320, 2023.

[19] The Cost of Latency. https://perspectives.mvdirona.com/2009/10/the-cost-of-latency/. Online; accessed 14-Jul-2025.

[20] OR Forum—The Cost of Latency in High-Frequency Trading. https://www.jstor.org/stable/24540485. Online; accessed 14-Jul-2025.

[21] Alexandros Daglis, Mark Sutherland, and Babak Falsafi. Rpcvalet: Ni-driven tail-aware balancing of µs-scale rpcs. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, page 35–48, New York, NY, USA, 2019. Association for Computing Machinery.

[22] Huynh Tu Dang, Marco Canini, Fernando Pedone, and Robert Soulé. Paxos made switch-y. *SIGCOMM Comput. Commun. Rev.*, 46(2):18–24, may 2016.

[23] Huynh Tu Dang, Daniele Sciascia, Marco Canini, Fernando Pedone, and Robert Soulé. NetPaxos: consensus at network speed. In *Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research*, SOSR '15, pages 1–7, New York, NY, USA, June 2015. Association for Computing Machinery.

[24] Intel Ethernet' s Performance Report with DPDK 23.03. https://fast.dpdk.org/doc/perf/DPDK_23_03_Intel_NIC_performance_report.pdf. Online; accessed 14-Jul-2025.

[25] DPDK testpmd app. https://doc.dpdk.org/guides/testpmd_app_ug/. Online; accessed 14-Jul-2025.

[26] Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, Aditya Akella, Kuangching Wang, Glenn Ricart, Larry Landweber, Chip Elliott, Michael Zink, Emmanuel Cecchet, Snigdhaswin Kar, and Prabodh Mishra. The design and operation of CloudLab. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, pages 1–14, July 2019.

[27] Etcd hardware reccomendations. https://etcd.io/docs/v3.5/op-guide/hardware/#network. Online; Accessed 14-Jul-2025.

[28] Joshua Fried, Zhenyuan Ruan, Amy Ousterhout, and Adam Belay. Caladan: Mitigating interference at microsecond timescales. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 281–297. USENIX Association, November 2020.

[29] Phillipa Gill, Navendu Jain, and Nachiappan Nagappan. Understanding network failures in data centers: measurement, analysis, and implications. *SIGCOMM Comput. Commun. Rev.*, 41(4):350–361, August 2011.

[30] Matthew P. Grosvenor, Malte Schwarzkopf, Ionel Gog, Robert N. M. Watson, Andrew W. Moore, Steven Hand, and Jon Crowcroft. Queues Don't matter when you can JUMP them! In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, pages 1–14, Oakland, CA, May 2015. USENIX Association.

[31] Rachid Guerraoui, Antoine Murat, Javier Picorel, Athanasios Xygkis, Huabing Yan, and Pengfei Zuo. uKharon: A membership service for microsecond applications. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, pages 101–120, July 2022.

[32] Zhisheng Hu, Pengfei Zuo, Yizou Chen, Chao Wang, Junliang Hu, and Ming-Chang Yang. Aceso: Achieving Efficient Fault Tolerance in Memory-Disaggregated Key-Value Stores. In *ACM SIGOPS 30th Symposium on Operating Systems Principles (SOSP '24')*, page 127–143, 2024.

[33] Kaiwen Huang, Ronghui Hou, and Yingming Zeng. Lwsbft: Leaderless weakly synchronous BFT protocol. *Computer Networks*, 219:109419, 2022.

[34] Nicholas Hunt, Tom Bergan, Luis Ceze, and Steven D. Gribble. Ddos: taming nondeterminism in distributed systems. *SIGARCH Comput. Archit. News*, 41(1):499–508, mar 2013.

[35] Patrick Hunt, Mahadev Konar, Flavio P. Junqueira, and Benjamin Reed. ZooKeeper: Wait-free coordination for internet-scale systems. In *2010 USENIX Annual Technical Conference (USENIX ATC 10)*. USENIX Association, June 2010.

[36] Stephen Ibanez, Alex Mallery, Serhat Arslan, Theo Jepsen, Muhammad Shahbaz, Changhoon Kim, and Nick McKeown. The nanopu: A nanosecond network stack for datacenters. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*, pages 239–256. USENIX Association, July 2021.

[37] Intel Corporation. Intel® Infrastructure Processing Unit (Intel® IPU) Platform F2000X-PL . https://cdrdv2-public.intel.com/792306/ipu-f2000-pl-platform-product-brief.pdf. Online; accessed 14-Jul-2025.

[38] iPerf tool. https://iperf.fr/. Online; accessed 14-Jul-2025.

[39] Zsolt István, David Sidler, Gustavo Alonso, and Marko Vukolic. Consensus in a box: inexpensive coordination in hardware. In *Proceedings of the 13th Usenix Conference on Networked Systems Design and Implementation*, NSDI'16, pages 425–438, USA, March 2016. USENIX Association.

[40] Joseph Izraelevitz, Gaukas Wang, Rhett Hanscom, Kayli Silvers, Tamara Silbergleit Lehman, Gregory Chockler, and Alexey Gotsman. Acuerdo: Fast Atomic Broadcast over RDMA. In *Proceedings of the 51st International Conference on Parallel Processing*, ICPP '22, pages 1–11, New York, NY, USA, January 2023. Association for Computing Machinery.

[41] Patrick Jahnke, Vincent Riesop, Pierre-Louis Roman, Pavel Chuprikov, and Patrick Eugster. Live in the express lane. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pages 581–597, July 2021.

[42] Sagar Jha, Jonathan Behrens, Theo Gkountouvas, Mae Milano, Weijia Song, Edward Tremel, Robbert Van Renesse, Sydney Zink, and Kenneth P. Birman. Derecho: Fast State Machine Replication for Cloud Services. *ACM Transactions on Computer Systems*, 36(2):4:1–4:49, April 2019.

[43] Xin Jin, Xiaozhou Li, Haoyu Zhang, Nate Foster, Jeongkeun Lee, Robert Soulé, Changhoon Kim, and Ion Stoica. NetChain: Scale-Free Sub-RTT coordination. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 35–49, Renton, WA, April 2018. USENIX Association.

[44] L Johnsson and G Netzer. The impact of moore's law and loss of dennard scaling: Are dsp socs an energy efficient alternative to x86 socs? *Journal of Physics: Conference Series*, 762(1):012022, oct 2016.

[45] Flavio P. Junqueira, Benjamin C. Reed, and Marco Serafini. Zab: High-performance broadcast for primary-backup systems. In *2011 IEEE/IFIP 41st International Conference on Dependable Systems and Networks (DSN)*, pages 245–256, 2011.

[46] Anuj Kalia, Michael Kaminsky, and David Andersen. Datacenter RPCs can be general and fast. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 1–16, Boston, MA, February 2019. USENIX Association.

[47] Elie F. Kfoury, Jorge Crichigno, and Elias Bou-Harb. An exhaustive survey on p4 programmable data plane switches: Taxonomy, applications, challenges, and future trends. *IEEE Access*, 9:87094–87155, 2021.

[48] Marios Kogias and Edouard Bugnion. HovercRaft: achieving scalability and fault-tolerance for microsecond-scale datacenter services. In *Proceedings of the Fifteenth European Conference on Computer Systems*, EuroSys '20, pages 1–17, New York, NY, USA, April 2020. Association for Computing Machinery.

[49] L. Lamport, R. E. Shostak, and M. C. Pease. The Byzantine Generals Problem. *Transactions on Programming Languages and Systems*, 4(3):382–401, 1982.

[50] Leslie Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, may 1998.

[51] Youngmoon Lee, Hasan Al Maruf, Mosharaf Chowdhury, Asaf Cidon, and Kang G. Shin. Hydra: Resilient and Highly Available Remote Memory. In *20th USENIX Conference on File and Storage Technologies (FAST '22)*, pages 181–198, February 2022.

[52] Jialin Li, Naveen Kr. Sharma, Dan R. K. Ports, and Steven D. Gribble. Tales of the tail: Hardware, os, and application-level sources of tail latency. In *Proceedings of the ACM Symposium on Cloud Computing*, SOCC '14, page 1–14, New York, NY, USA, 2014. Association for Computing Machinery.

[53] Nanqinqin Li, Anja Kalaba, Michael J. Freedman, Wyatt Lloyd, and Amit Levy. Speculative Recovery: Cheap, Highly Available Fault Tolerance with Disaggregated Storage. In *2022 USENIX Annual Technical Conference (ATC '22)*, pages 271–286, July 2022.

[54] Barbara H. Liskov and James A. Cowling. Viewstamped replication revisited. 2012.

[55] Shengyun Liu, Paolo Viotti, Christian Cachin, Vivien Quéma, and Marko Vukolic. XFT: practical fault tolerance beyond crashes. In *12th USENIX Symposium on Operating Systems Design and Implementation, (OSDI '16')*, pages 485–500, 2016.

[56] Vincent Liu, Daniel Halperin, Arvind Krishnamurthy, and Thomas Anderson. F10: A Fault-Tolerant engineered network. In *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, pages 399–412, Lombard, IL, April 2013. USENIX Association.

[57] Nancy Lynch. *Distributed Algorithms*. 1996.

[58] Sarah McClure, Amy Ousterhout, Scott Shenker, and Sylvia Ratnasamy. Efficient scheduling policies for Microsecond-Scale tasks. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 1–18, Renton, WA, April 2022. USENIX Association.

[59] Mellanox Connectx-4. http://www.mellanox.com/related-docs/prod_adapter_cards/PB_ConnectX-4_VPI_Card.pdf. Online; accessed 14-Jul-2025.

[60] Nenad Milosevic, Daniel Cason, Zarko Milosevic, and Fernando Pedone. How robust are synchronous consensus protocols? In *28th International Conference on Principles of Distributed Systems, OPODIS 2024, December 11-13, 2024, Lucca, Italy*, volume 324 of *LIPIcs*, pages 20:1–20:25, 2024.

[61] Nenad Milosevic, Daniel Cason, Zarko Milosevic, Robert Soulé, and Fernando Pedone. Message size matters: Alterbft's approach to practical synchronous BFT in public clouds. *CoRR*, abs/2503.10292, 2025.

[62] NVIDIA MLNX_OFED Hardware Timestamping documentation. https://docs.nvidia.com/networking/display/mlnxofedv543681lts/time-stamping. Online; accessed 14-Jul-2025.

[63] Intel® Infrastructure Processing Unit (Intel® IPU) Platform F2000X-PL . https://www.intel.com/content/www/us/en/software-kit/750666/modelsim-intel-fpgas-standard-edition-software-version-20-1-1.html. Online; accessed 14-Jul-2025.

[64] Diego Ongaro and John Ousterhout. In Search of an Understandable Consensus Algorithm. In *2014 USENIX Annual Technical Conference*, USENIX ATC '14, pages 305–319, 2014.

[65] Amy Ousterhout, Joshua Fried, Jonathan Behrens, Adam Belay, and Hari Balakrishnan. Shenango: Achieving high CPU efficiency for latency-sensitive datacenter workloads. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 361–378, Boston, MA, February 2019. USENIX Association.

[66] Simon Peter, Jialin Li, Irene Zhang, Dan R. K. Ports, Doug Woos, Arvind Krishnamurthy, Thomas Anderson, and Timothy Roscoe. Arrakis: The operating system is the control plane. *ACM Trans. Comput. Syst.*, 33(4), nov 2015.

[67] Marius Poke and Torsten Hoefler. Dare: High-performance state machine replication on rdma networks. In *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing*, HPDC '15, page 107–118, New York, NY, USA, 2015. Association for Computing Machinery.

[68] Redis. https://redis.io. Online; accessed 14-Jul-2025.

[69] RedisRaft, consistent key-value store. https://github.com/RedisLabs/redisraft. Online; accessed 14-Jul-2025.

[70] Redis Replication Docs. https://redis.io/docs/latest/operate/oss_and_stack/management/replication/. Online; accessed 14-Jul-2025.

[71] Rocky linux - download. https://rockylinux.org/download. Online; accessed 14-Jul-2025.

[72] Davide Rovelli, Pavel Chuprikov, Philipp Berdesinski, Ali Pahlevan, Patrick Jahnke, and Patrick Eugster. FiDe: Reliable and Fast Crash Failure Detection to Boost Datacenter Coordination. In *2025 USENIX Annual Technical Conference (ATC'25)*, pages 765–788, 2025.

[73] Davide Rovelli and Patrick Eugster. Digital cluster circuits for reliable datacenters. In *2025 55th Annual IEEE/IFIP International Conference on Dependable Systems and Networks - Supplemental Volume (DSN-S)*, pages 201–205, 2025.

[74] Davide Rovelli, Michele Dalle Rive, and Patrick Eugster. Toward a practical deterministic datapath for 6g end devices. *IEEE Network*, 39(3):75–82, 2025.

[75] Nicolas Schiper and Sam Toueg. A robust and lightweight stable leader election service for dynamic systems. In *2008 IEEE International Conference on Dependable Systems and Networks With FTCS and DCC (DSN)*, pages 207–216, 2008.

[76] R.D. Schlichting and F.B. Schneider. Fail-Stop Processors: An Approach to Designing Fault-Tolerant Computing Systems. *ACM Transactions on Computer Systems (TOCS)*, 1(3):222–238, 1983.

[77] Jonathan Stone and Craig Partridge. When the crc and tcp checksum disagree. *SIGCOMM Comput. Commun. Rev.*, 30(4):309–319, aug 2000.

[78] Stress-ng tool. http://colinianking.github.io/stress-ng/. Online; accessed 20-May-2025.

[79] Mark Sutherland, Siddharth Gupta, Babak Falsafi, Virendra Marathe, Dionisios Pnevmatikatos, and Alexandres Daglis. The nebula rpc-optimized architecture. In *Proceedings of the ACM/IEEE 47th Annual International Symposium on Computer Architecture*, ISCA '20, page 199–212. IEEE Press, 2020.

[80] Catching Corrupted OSPF Packets! - Blog. https://routingfreak.wordpress.com/2011/03/01/catching-corrupted-ospf-packets/. Online; accessed 14-Jul-2025.

[81] How both TCP and Ethernet checksums fail - Blog. https://www.evanjones.ca/tcp-and-ethernet-checksums-fail.html. Online; accessed 14-Jul-2025.

[82] Parth Thakkar, Rohan Saxena, and Venkata N. Padmanabhan. Autosens: inferring latency sensitivity of user activity through natural experiments. In *Proceedings of the 21st ACM Internet Measurement Conference*, IMC '21, page 15–21, New York, NY, USA, 2021. Association for Computing Machinery.

[83] Maarten Van Steen and Andrew S Tanenbaum. *Distributed systems*. Maarten van Steen Leiden, The Netherlands, 2017.

[84] Xingda Wei, Rongxin Cheng, Yuhan Yang, Rong Chen, and Haibo Chen. Characterizing off-path SmartNIC for accelerating distributed systems. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, pages 987–1004, Boston, MA, July 2023. USENIX Association.

[85] Irene Zhang, Amanda Raybuck, Pratyush Patel, Kirk Olynyk, Jacob Nelson, Omar S. Navarro Leija, Ashlie Martinez, Jing Liu, Anna Kornfeld Simpson, Sujay Jayakar, Pedro Henrique Penna, Max Demoulin, Piali Choudhury, and Anirudh Badam. The demikernel datapath os architecture for microsecond-scale datacenter systems. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, page 195–211, New York, NY, USA, 2021. Association for Computing Machinery.

[86] Yang Zhou, Zezhou Wang, Sowmya Dharanipragada, and Minlan Yu. Electrode: Accelerating Distributed Protocols with {eBPF}. pages 1391–1407, 2023.

[87] Yang Zhou, Hassan M. G. Wassel, Sihang Liu, Jiaqi Gao, James Mickens, Minlan Yu, Chris Kennelly, Paul Turner, David E. Culler, Henry M. Levy, and Amin Vahdat. Carbink: Fault-Tolerant Far Memory. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI '22)*, pages 55–71, July 2022.

[88] Zookeeper administrator's guide. https://zookeeper.apache.org/doc/r3.1.2/zookeeperAdmin.html. Online; Accessed 14-Jul-2025.