Live in the Express Lane

Patrick Jahnke[*†] Vincent Riesop[†] Pierre-Louis Roman[‡] Pavel Chuprikov[‡] Patrick Eugster[‡*§]
[*]*TU Darmstadt* [†]*SAP* [‡]*Università della Svizzera italiana* [§]*Purdue University*

Abstract

We introduce Express-Lane (X-Lane), a novel system for mitigating interference in data center infrastructure to improve the liveness of coordination services. X-Lane follows a novel design from the ground up to achieve interactions with ultra-low latency in the single-digit microsecond range and jitter in the nanosecond range, while the remaining interaction is treated as usual. To show X-Lane's applicability and genericity we implemented and evaluated two services atop it on commodity hardware in a production environment of SAP SE: a failure detector (X-FD) with detection time under 10 µs and a Raft implementation (X-Raft) with latencies under 20 µs. We further show the smooth integrability of X-Lane services by replacing the replication protocol of Redis with X-Raft, making it strongly consistent while improving latency 18× and write throughput 1.5×.

1 Introduction

In the last decade, a tremendous increase in Internet connectivity and the need for more computational performance changed the way we conceive applications. Today, most new applications are conceived as distributed, and in particular cloud-based, applications. The design of data centers and middleware layers then has to take into account all requirements for distributed coordination, including performance, fault-tolerance, and consistency [16] — a hard task.

Interference in distributed systems. Most distributed system designs treat the underlying infrastructure as a generic communication system. One of the main issues with this abstraction is the longstanding problem of interference of concurrent interactions and thus unpredictable latency of commodity networks and hosts [19]. Many distributed systems suffer from jitter induced by interference, manifesting through packets that may be arbitrarily delayed in the network (as well as reordered or dropped), and unbounded processing times.

Many applications and components have been designed to cope with the unpredictability of the infrastructure by making weak synchrony assumptions to guarantee a safe execution of their protocol. Yet, they rely on upper bounds for the latency of their interactions to ensure liveness, by way of timeouts, and as thus benefit strongly from interactions with low latency and bounded jitter. This is especially the case for coordination tasks [52] whose use is widespread in practice. Types of systems using the ZooKeeper [28] coordination service based on the popular Paxos [37] protocol by default or as option for coordination/fault tolerance include resource management (e.g., Mesos [25], YARN [54]), key-value and wide-column stores (e.g., Accumulo [20], HBase [1], etcd [4], TiKV [10]), data analytics (e.g., Hadoop [12], Spark [58]), or distributed filesystems (e.g., HDFS [13]) to only name few.

X-Lane. The research question underlying this work is whether interference in data center commodity systems can be mitigated to greatly accelerate coordination tasks lying at the core of distributed systems.

Prior works on low latency communication (e.g., [14,24,41, 44,48,50]) focus on reducing 99$^{\text{th}}$ percentile latency where packets may be sacrificed (dropped) to maintain a good performance in most cases (e.g., fitting a given service-level objective (SLO) for 99% of the packets). Our goal is to address not only fast but also *timely sensitive interactions* for tasks that exhibit severe performance degradation upon delayed message delivery (e.g., when timeouts trigger). To this end, we aim at reducing maximum jitter to a point where it becomes so small relative to an already very low latency, that, in practice, it can be assumed to be bounded. Moreover, we include *endhost response times*, and only provide bounded jitter to applications that rely on it (e.g., for coordination). Thus we introduce with Express-Lane — X-Lane for short — an interference-free environment for select interactions with ultra low latency in the *single-digit microsecond range* and bounded jitter in *nanosecond range*. The remaining interactions follow common design principles. While being more generic in design compared to prior work on minimizing av-

erage latency, and also considering endhosts, X-Lane delivers significantly tighter bounds for latency and jitter for commodity hardware (HW) and software (SW).

In short, X-Lane isolates and prioritizes packets traversing it by using traffic engineering techniques to provision and monitor resources dedicated for X-Lane, and by neutralizing sources of interference inherent to data center infrastructures, i.e., interference present in endhosts/servers, switches, and links. X-Lane strives first and foremost to minimize jitter, and in the process also achieves unprecedented low latency.

Contributions and roadmap. This paper contributes:

- §2 Design of X-Lane atop commodity HW and SW, and for intelligent network devices (smartNICs) when available;
- §3 Traffic engineering approach incorporating residual jitter and queuing delay to perform packet-level latency analysis in X-Lane;
- §4 Implementation of X-Lane overcoming interference causing jitter on top of commodity HW and SW, as well as improvements and simplifications taking advantage of Netronome's NFP-4000-based smartNICs [8];
- §5 Definition and implementation of two example asynchronous services using X-Lane: a failure detector dubbed X-FD, and a state machine replication protocol dubbed X-Raft adapted from Raft [45];
- §6 Evaluation of X-Lane in a production data center of SAP SE through the deployment of the two services. We measure median latency and maximum jitter of X-Lane on commodity HW and SW (Linux) (5.130 µs latency and 655 ns jitter) and smartNICs (4.133 µs latency, 152 ns jitter) *with heavy concomitant traffic over the course of 21 days*. Further comparisons display vast improvements over DPDK [21] (1.735× lower latency, 81,816× lower jitter), and QJump [24] (1.501× lower latency, 72,758× lower jitter), which greatly affect the coordination of distributed systems. We also show the applicability of X-Lane by integrating X-Raft into the Redis key-value store [9], making it strongly consistent while decreasing latency 18× and increasing write throughput 1.5×.

We compare X-Lane to related work in §7 before we draw the conclusions and discuss future work in §8. Additional material can be found on the project website [30].

2 X-Lane Design Overview

With X-Lane we propose an explicit express lane for timely sensitive interactions, following our original design outlined in Fig. 1. X-Lane is isolated from the "regular system" which follows common design principles. This architecture is reminiscent of earlier models of separate systems [55, 56], yet realizes them concretely, in a single infrastructure, with commodity HW and SW.

2.1 Communication Model

X-Lane's novelty is characterized by an explicit upper-bound on the latency of all the messages sent by a given process p to another process q, i.e., X-Lane keeps the latency of every such message within $[\lambda_{\min}^{p,q}, \lambda_{\min}^{p,q} + \delta_{\max}^{p,q}]$, where $\lambda_{\min}^{p,q}$ is the best-case latency, and $\delta_{\max}^{p,q}$ is its concomitant maximum jitter. In the following, we denote jitter δ as a deviation from the best-case latency $\lambda_{\min}$.

We achieve bounded communication latency in X-Lane by implementing a *periodic unicast protocol* where a process p can send a message to a given process q with latency upper bound $\lambda_{\min}^{p,q} + \delta_{\max}^{p,q}$, but under two constraints: p can send only once during every period $\pi^{p,q}$, and the packet size may not exceed $\sigma^{p,q}$. In addition, we specifically address one-to-many communication patterns by a *periodic multicast protocol* that allows a process p to send a message to a *set of processes* Q with a common latency range $[\lambda_{\min}^{p,Q}, \lambda_{\min}^{p,Q} + \delta_{\max}^{p,Q}]$. A crucial requirement for both our protocols is that *all their parameters become known by the sending process at the protocol setup time,* i.e., before the first use, in order to allow services to adjust their internal timeouts for the best possible performance.

Note, purely bandwidth-oriented communication abstractions are *not* suitable for X-Lane, for they leave message size unspecified, while, clearly, no latency bound would hold uniformly for *every* message size, and queuing behind an arbitrarily large message leads to unbounded maximum jitter.

X-Lane is able to provide timely sensitive interaction that exhibits stable behavior as long as interconnecting devices function properly. Hence X-Lane is best used to improve the liveness of coordination tasks that assume an asynchronous communication model to guarantee safety properties.

Timely unicast and multicast serve as *backbone for all communication* between processes in X-Lane. In the following, "periodic protocol" refers to "unicast protocol or multicast protocol". Bounding latency in the sending process is addressed in §2.4 and detailed in §4.

2.2 Components Overview

To achieve the properties provided by the two periodic protocols, X-Lane introduces a software-defined networking (SDN) controller that takes on two main orchestration responsibilities: 1) *resource allocation,* i.e., answering requests from services with the most suitable protocol parameters, subject to network capacity constraints; and 2) *resource tuning,* i.e., keeping overall usage of X-Lane low. Traffic engineering (TE) techniques that underpin this operation are presented in §3.

The X-Lane controller interacts with each endhost via a client integrated in the X-Lane (Linux) kernel module (X-KM) loaded on each endhost. The client exposes the controller API (cf. List. 1) to services forwarding requests and responses in both directions. It is important to note that only the bounded communication over X-Lane is managed by the X-Lane con-

```
// Service request parameters for X-Lane resources
struct request {
  int loadsize;             // max packet size (B)
  int period;               // packet period (µs)
  struct {
    uint32_t ip;            // MCast or UCast IPv4
    uint16_t port;          // service port
  } receiver;
};
// Resources approved by the X-Lane controller
static const int UNBOUNDED = -1;
struct resources {
  int loadsize;             // max packet size (B)
  int period;               // approved period (µs)
  int minLatency;           // minimum latency (ns)
  int maxJitter;            // maximum jitter (ns)
};
// Reason for resource modification
enum Reason { TE, BW_EXCEEDED, BW_UNUSED };
// Downcalls from services to controller
↓ resources requestBandwidth(request req);
↓ void releaseBandwidth();
↓ void changeBandwidth(request req);
// Upcalls from controller to applications
↑ void bandwidthChanged(resources res,
    Reason reason = TE);
↑ void bandwidthTerminated();
```

List. 1: Extract of the X-Lane controller C API used for resource allocation and tuning. Structure `resources` defines a timely periodic protocol. The first three methods are called by services the next two are upcalls/callbacks.

troller. The rest of the communication proceeds as usual and uses the remaining resources in the usual best-effort manner. If no requests are ever made to the X-Lane controller, no network resources are spared or lost.

2.3 Overview of Jitter Sources

To implement an express lane usable in practice for time-sensitive tasks, we need to mitigate the inherent interferences in data center computing. We expose and address numerous jitter sources in §4. In short, we identify the following causes:

- **Packet loss:** Packets can be lost, leading to retransmissions and thus uncertain latency. Besides intentional drops (e.g., for security), packet loss has two well-known causes:
  - **Bit flip errors:** Bits can get flipped in links, leading to packets being marked as corrupted and discarded (§4.1);
  - **Buffer overflows:** Packets are dropped when the finite resources on processing units are overloaded (§4.2).
- **Intrinsic jitter:** While commodity switching devices forward packets with little jitter (§4.2), endhosts and their commodity components have been becoming more complex, leading to many sources of jitter (§4.3) and motivating the need for moving the intelligence closer to network devices (§4.4). The lack of bounds on jitter further makes packet delay hard to distinguish from packet loss.

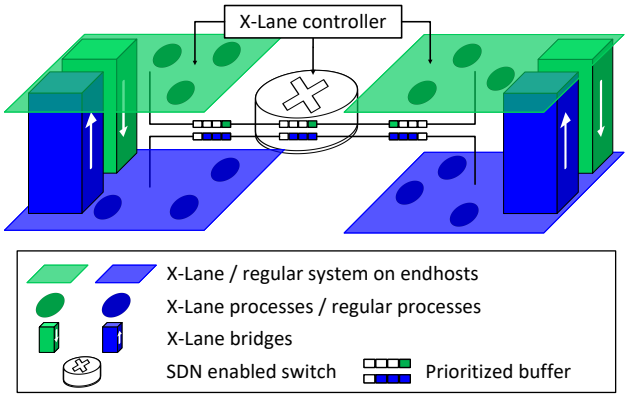

Figure 1: Separating the traffic of Express-Lane (X-Lane) and regular communication on switches to prioritize packets and prevent losses in the former. An SDN controller sets switches' rules to adapt buffer allocation and processing priority. X-Lane is interfaced to the regular system via its bridges.

2.4 X-Lane (Based) Services

X-Lane enables processes executing on the regular system to interact with the X-Lane services that may offer *timely* responses thanks to the unique timing properties of communications of X-Lane. There are a few intricacies to X-Lane that developers must take into account when interacting with and/or developing these services. First, applications and services, being in separate lanes, must use a specific interface to exchange data with each other. Second, X-Lane handles communications differently than in the regular system.

Building bridges between lanes. On endhosts, services must communicate with applications which have to deal with shared processor time. This resource sharing introduces unpredictable jitter for those processes while critical interactions need an upper bound for certain tasks. Hence X-Lane provides two sets of queues, called *bridges*, to establish the interface between processes in X-Lane and on the regular system.

Fig. 1 depicts the bridges. The express-to-regular (X-R) bridge (green cuboid) grants write access to X-Lane (green parallelogram) and read access to the regular system (blue parallelogram); inversely for the R-X bridge (blue cuboid).

Bridges are addressable using direct memory access (DMA) over PCIe (to minimize jitter, cf. §4.3) but are placed at different locations depending on the endhost HW configuration.

Using X-Lane services. Services are implemented as components of the X-KM (cf. §5 for already available services), and as thus have direct access to the client of the controller and to the network interface card (NIC) bridge, another X-KM component responsible for communication with the NIC. Each service has a dedicated queue in the R-X bridge where it

can receive (1) queries from applications wishing to start/stop using that service, and (2) queries and payloads specific to that service API. When an application starts using a service, the service requests network resources from the X-Lane controller and spawns a new queue in the X-R bridge dedicated to messages from this service to that application. The NIC bridge bundles up all the payloads from a service into packets and sends them over the wire at the allowed periodicity (cf. `period` in List. 1), and unpacks payloads on the receiver side. Like drivers, the bridge implementation varies between HWs.

Express communication on commodity HW. While commodity NICs rapidly process and copy packets to the main memory, they are not programmable. Procedures to send and receive packets must thus be executed by the CPU.

When handling packets that belong to X-Lane, guaranteeing minimal response time and tight timing bounds for these procedures is especially challenging on commodity HW. There is an abundance of sources of jitter within the CPU itself and in the communication path between the CPU and the NIC that prevents a jitter-free streamline flow of packets. As a response, we implemented a series of countermeasures to enable X-Lane on commodity HW, greatly improving the time bounds over the regular system, as detailed in § 4.3. On commodity HW, all bridges are in the main memory.

Express communication on smartNICs. Unlike commodity NICs, new generation NICs — so-called smartNICs — are highly programmable. Tasks can be offloaded from the CPU to the processing engine of a smartNIC, ranging from packet pre-processing to complex programs. The (relative) simplicity of the HW and SW stacks of smartNICs, over those of an endhost operated by a Linux kernel, and their proximity to the physical interface enable for packets to be processed on smartNICs with far lower latency and jitter (cf. § 6.2). This makes smartNICs ideal to handle X-Lane services.

Processing for sending and receiving packets over X-Lane is confined within the smartNIC. This processing is mostly as with commodity HW, but with direct access to the packet processing pipeline and the ingress and egress buffers on the NIC (cf. § 4.4). The X-R bridge is stored in the smartNIC's memory while the R-X bridge is in the endhost main memory.

3 Traffic Engineering for Tunnel Trees

The key underlying mechanism of the controller are *latency-bounded fixed-bandwidth* tunnels, more precisely — tunnel trees (due to multicast), from sender to receiver processes.

3.1 Tunnel Allocation Model

The X-Lane controller relies on SDN for tunnel setup. In particular, by acting as an SDN controller, it gets access to network-wide view in a form of a *network topology graph* G and the means to manage switches. For every link $(u,v) \in G$, the following information is used: bandwidth $\mathrm{bw}(u,v)$, size of an egress queue $\mathrm{qlen}(u,v)$, minimum delay $\lambda_{\min}(u,v)$, and maximum jitter $\delta_{\max}(u,v)$. Importantly, $\lambda_{\min}(u,v)$ and $\delta_{\max}(u,v)$ need only include processing and propagation delays, which are stable for switches and are made stable at endhosts by X-Lane's endhost implementation (see § 4).

A resource *allocation* is represented by a set $\mathcal{T}$ of *tunnels*, where every $T \in \mathcal{T}$ is a *directed subtree* of the topology graph G with a sender source $\mathrm{snd}(T)$ and a set of receiver sinks $\mathrm{rcvs}(T)$. Tunnels are in one-to-one correspondence with allocated `resources` shown in List. 1; hence, for every $T \in \mathcal{T}$, we have packet size $\sigma(T)$, period $\pi(T)$, minimum latency $\lambda_{\min}(T)$, and maximum jitter $\delta_{\max}(T)$. X-Lane further employs TE techniques [26, 27, 31] to guarantee channel availability. The particular TE algorithm used for X-Lane is close to B4's state-of-the-art approach [27] (with worst-case estimation of available throughput) but is built upon a finer-grained network model to allow for packet-level latency bounds.

The X-Lane controller does not make any explicit resource reservations in the network but instead relies on *rate limiting* at the endhosts, forcing services to adhere to periodic protocol parameters. Thus, the traffic for a given tunnel T consists of packets of size $\sigma(T)$ entering node $\mathrm{snd}(T)$ precisely every $\pi(T)$ with starting time chosen arbitrarily for each T. Once a packet p from T arrives at a node u, p is either delivered, if $u \in \mathrm{rcvs}(T)$, or p is placed into u's egress queue(s) corresponding to next hop(s) in T, provided there is sufficient buffer space, if not — p is dropped. Switching and/or processing delays at u are incorporated into latency and jitter of incoming links. At the egress queue, p waits for its turn to be transmitted according to FIFO order, and after $\mathrm{size}(p)/\mathrm{bw}(u,v)$ seconds more p leaves the queue. It takes anywhere between $\lambda_{\min}(u,v)$ and $\lambda_{\min}(u,v) + \delta_{\max}(u,v)$ before p enters the next hop v accounting for the minimum residual jitter remaining after applying techniques described in § 4.

TE of X-Lane accounts for both the intrinsic uncertainties of the system and uncertainties arising from multiple services sharing network resources. Ultimately, TE ensures that allocation $\mathcal{T}$ is *valid w.r.t.* topology G, meaning that no actual system behavior violates $\lambda_{\min}(T)$ and $\delta_{\max}(T)$ for $T \in \mathcal{T}$.

3.2 Two-Phase Allocation Approach

Resources in X-Lane are allocated reactively, upon concrete requests by services.

To bootstrap a periodic protocol, a service calls the `requestBandwidth` method of the controller API passing the desired packet size and periodicity in a `request` structure r. The controller handles r as follows: 1) a new tunnel T is allocated between the sender and receiver(s); 2) switches' meter tables are updated for resource monitoring; 3) parameter adjustments for other affected tunnels in $\mathcal{T}$ are communi-

cated to corresponding services using the `bandwidthChanged` callback; 4) the approved resources with periodicity adjusted according to the allocation are returned to the service. Naturally, the new tunnel T must *match* the request r, i.e., packet size $\sigma(T)$ is equal to r.`loadsize`, snd(T) is the process that originated r, rcvs(T) correspond to r.`receiver.ip`, and $\pi(T) \geq r$.`period` mind the adjustment). The returned structure reflects all the T's parameters of a periodic protocol (cf. § 2.1): latency range $[\lambda_{\min}(T), \lambda_{\min}(T) + \delta_{\max}(T)]$, periodicity $\pi(T)$, and load size $\sigma(T)$. The service frees the resources by using `releaseBandwidth`. For the X-Lane properties to be reliable, every `bandwidthChanged` callback invoked by the controller comes with a grace period, during which the service can send messages under the old periodic protocol guarantees.

A distinguishing feature of our setting is the inevitable interference between already established tunnels and the new tunnel. Trying to minimize such interference, we arrive to an optimization problem underlying steps 1) and 3) above.

Problem (X-TE)**.** *Given a network G, an allocation $\mathcal{T}$, and a sequence of service requests $r_1, \ldots, r_k$, find a sequence of new tunnels $\mathcal{T}' = T'_1, \ldots, T'_k$ and adjust parameters of $\mathcal{T}$, s.t., T'_i matches r_i for $1 \leq i \leq k$, $\mathcal{T} \cup \mathcal{T}'$ is valid w.r.t. G, and $\sum_{T \in \mathcal{T} \cup \mathcal{T}'} (\lambda_{\min}(T) + \delta_{\max}(T))$ is minimized.*

Solving X-TE directly is challenging as deriving parameters (or even checking validity) for a general $\mathcal{T}$ is highly non-trivial due to interdependency between arrival times for packets queuing behind each other. Hence to simplify the problem, we split the allocation into two phases: *optimization* and *adjustment*. The *optimization* phase takes as input a request sequence and decides on the matching sequence of tunnels. In the current implementation, we allocate trees one-by-one; each tree is allocated incrementally by greedily attaching receiver sinks while minimizing the current value of the X-TE's objective function. The *adjustment* phase alters the parameters of all tunnels so they become valid w.r.t. the network G. Each tree is adjusted independently using depth-first search traversal calculating worst-case parameters.

3.3 Resource Monitoring and Tuning

In addition to its resource allocation task, the X-Lane controller improves resource utilization by monitoring and refining the set of already allocated tunnels.

Controller oversight. For instance, if a service wants to adjust its `loadsize` and/or `period` without disrupting other services, the `requestBandwidth` and `releaseBandwidth` methods force it to establish a new periodic protocol first, migrate all clients there, and only then release the old resources. This two-phase approach incurs artificial delay, adds complexity, and wastes X-Lane's resources. The `changeBandwidth` method of the controller's API shortcuts the process by leveraging the `bandwidthChange` mechanism discussed earlier.

When a service attempts to use more resources than assigned, some of its packets get dropped at a rate limiter. X-Lane can do nothing to maintain timeliness for those packets, and neither should it as the service has violated the protocol. To ensure an already broken interaction does not waste resources, the controller decreases priority of that service's packets right after the drop, voiding their timing guarantees. Then, the jitter reduction is communicated to services sharing queues with the misbehaving one, and the latter is notified by `bandwidthChanged` with `resources.priority` set to `UNBOUNDED` and `reason` to `BW_EXCEEDED`. This service may recover later with `changeBandwidth`. Further, switches' meter tables are used to identify services that behave well but *underutilize* resources. The controller reclaims a portion of their bandwidth through the `bandwidthChanged` callback with higher period and `reason` set to `BW_UNUSED`.

In the extreme scenario when a service keeps violating the protocol and/or drives its bandwidth allocation to zero by not utilizing resources, the controller terminates the protocol unilaterally with `bandwidthTerminated`.

Fine-grained jitter control with sub-lanes. Earlier, we saw newly set up tunnels adding jitter to existing ones and vice versa, whose effect we incorporated in periodic protocol parameters. Certain combinations of services require a different approach. A low-traffic jitter-sensitive service (e.g., failure detection, cf. § 5.1) and a throughput-oriented one needing a "small enough" latency bound (e.g., replication, cf. § 5.2), affect each other in very unequal ways leading to suboptimal overall performance. The controller addresses this issue through virtual sub-lanes — virtual controller instances that use different priority levels for timely communication, isolating services in a higher-priority sub-lane from lower-priority sub-lanes. This separation needs only be reflected at the tunnel setup, where lower-priority tunnels must include jitter from higher-priority ones but not the other way around.

4 Overcoming Jitter in Data Centers

Comprehensive mitigation of jitter sources due to interference with the rest of the data center (outlined in § 2.3) is key to achieving latency with tight bounds. In what follows we describe our technique and discuss implementation details.

4.1 Bit Flips Errors in Links

Most of the messages transmitted via X-Lane are expected to be much smaller than the MTU size. To reduce the data transmission overhead, X-Lane tries to pack multiple data chunks into a single physical packet. The increased chance of packet loss due to bit flip errors is mitigated by using two custom error correction schemata that provide the same mean time to fault packet acceptance as layer 2 headers (i.e., 10^6 years with

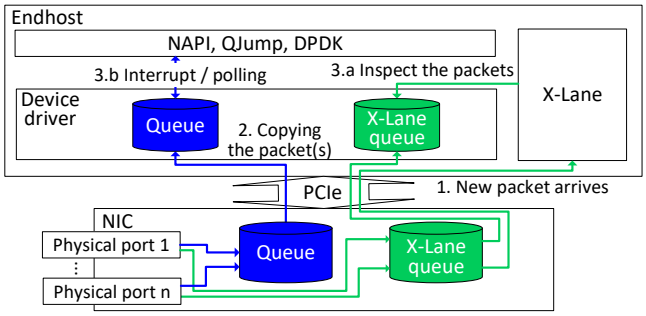

Figure 2: Overview of packet reception on commodity HW.

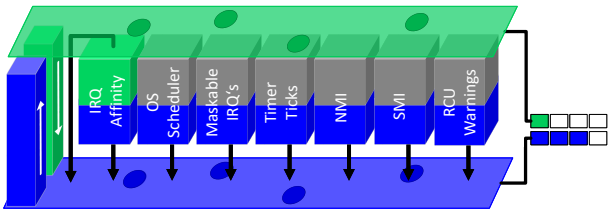

Figure 3: X-Lane is pinned to a dedicated core on which the sources of preemption (cuboids) are entirely (gray) or partly (green) disabled. The regular system is running on all other cores with all the side effects.

a bit error rate of 10^{-12}), while supporting either up to 55 chunks of 26 bytes per MTU or up to 40 chunks of 36 bytes (depending on the schema). Both schemata use a specific choice of cyclic redundancy code (CRC) polynomials.

4.2 Buffer Overflows and Jitter in Switches

Endhost NICs have a large amount of buffer memory available, allowing them to enqueue large numbers of packets before they are constrained to drop some. In contrast, commodity switching HW has a much smaller amount of (shared) buffer memory, that is commonly exceeded in the case of congestion, leading to packet losses ultimately hampering latency and jitter bounded communication. Commodity switches using an ASIC as forwarding processor can have their shared buffer split in multiple queues that are populated with packets from incoming traffic and are processed following a given scheduling strategy.

X-Lane uses a strict priority scheduler to realize the TE approach introduced in §3, to serve queues in order of priority, i.e., a non-empty queue is chosen over any other queue with lower priority. For each switch handling X-Lane's flows, the X-Lane controller (cf. § 2.2) dedicates the switch's highest priority queues to X-Lane, and adapts the queues' size to the expected load. X-Lane packets are therefore processed as fast as possible, reducing both jitter and the risk of packet drops since packets are processed before the queue is full. Furthermore, commodity switches are tailor-fitted to forward packets, they thus do so deterministically in the ns range [2].

4.3 Jitter in Endhost Commodity Hardware

While the standard network stack built upon endhost commodity HW can be used for throughput-oriented communication, the many sources of jitter it contains preclude X-Lane from using it for communication with bounded latency. Fig. 2 depicts how packets are handled when received on X-Lane (green) compared to the regular system (blue); X-Lane focuses on timestamping packets as early as possible to minimize stamping jitter, doing so even before their payload is inspected, therefore performing *optimistic timestamping*. In the following, we give an overview of the measures implemented in X-Lane to drastically reduce jitter and latency of transmitting packets atop endhost commodity HW and SW.

First, at least a CPU core must remain available at all times for X-Lane services to promptly send and receive packets to/from other services running on other endhosts. To do so, X-Lane runs on a dedicated core (§4.3.1) and shunts preemption on it to minimize completion time of X-Lane services (§ 4.3.2). Second, packets must be copied between the CPU, for processing, and the NIC, for remote exchange, while avoiding jitter-prone kernel memory management (§4.3.3). Fig. 3 gives an overview of preemption sources X-Lane disables compared to a regular system.

4.3.1 Highly Responsive X-Lane Dedicated CPU Core

Execution slots on CPU cores are managed by the OS kernel scheduler which, typically, distributes these slots in a fair manner across all applications to avoid resource starvation. Timing-sensitive tasks are therefore regularly preempted to leave room for other tasks, increasing both latency and jitter for the former. Even earliest deadline first (EDF) schedulers [43] are affected by their jitter-prone environments and cannot guarantee the highest degree of responsiveness for such tasks. Furthermore, CPUs can switch between power consumption modes (i.e., C-states defined by the ACPI standard) to save energy when idle but need to wake up from an idle mode to execute a task, hampering response time [18].

X-Lane thus is *pinned* to a core, and isolates it from the scheduler to avoid task preemptions for a better response time. We call this core *X-Lane's core* as it is (almost) exclusively managed by X-Lane. X-Lane's core is isolated by including it in the `isolcpus` kernel boot parameter. To avoid costly wake-ups, X-Lane's core remains in the highest active state by setting the following kernel boot parameters: `cpuidle.off=1`, `powersave=off`, `processor.max_cstate=0`.

4.3.2 X-Lane's Uninterrupted Execution

Interrupt request (IRQ) signals are generated by HW devices, e.g., I/O devices or CPU, to notify a core of an event to handle. The CPU preempts the task it is running to treat the received IRQ, which in effect increases the task's completion time and completion jitter due to the unpredictability of these IRQs.

We mitigate these delays by shielding X-Lane's core from as many IRQs as possible, as overviewed in Fig. 3. Those that cannot be ignored see their impact reduced (e.g., timer ticks).

IRQ affinity. On multi-core systems, IRQs can be distributed among cores statically — IRQs are always routed to the same core, or dynamically — IRQ affinity is set such that IRQs are handled by the core running the lowest priority task.

Most IRQs are routed away from X-Lane's core via a static distribution while other cores use a dynamic distribution, achieved by changing each IRQ's `smp_affinity` file in `/proc`.

IRQ masking. Some IRQs cannot be re-routed by setting their IRQ affinity, such as inter-processor interrupts (IPIs) that target a specific core. These IRQs can however be masked to prevent them from preempting the targeted core.

X-Lane uses the `local_irq_save(int state)` kernel function to masks IRQs before it executes an X-Lane service, and it restores the IRQ `state` afterwards using `local_irq_restore(int state)` with the same parameter. The masked IRQs are routed to other cores, by adapting their affinity, to preserve the correct operation of the system.

NMI watchdog. The Linux kernel integrates a watchdog timer that regularly sends non-maskable interrupts (NMIs) to each core to test for HW failures; it halts the system if the HW does not handle the NMI. There exists no standard kernel mechanisms to ignore the watchdog's NMIs.

X-Lane prevents these jitter-inducing NMIs by disabling the watchdog using the `nowatchdog` kernel boot parameter.

Timer ticks. Timer ticks are a special type of IRQs originating from CPU-local timers or external timers. They are used to run routines at a set frequency, typically between 100 and 1000 Hz, as configured in the kernel [46]. In our experiments, we have observed a substantial processing time for each of these interrupts, ranging from 1.5 µs to 50 µs.

X-Lane mitigates timer interrupts by configuring the kernel with the `CONFIG_NO_HZ_IDLE=y` option and adding X-Lane's core to the `nohz_full` kernel boot parameter, which sets the given core to adaptive-tick mode. While this mode does not completely oust interrupts, it greatly reduces their frequency to 1 Hz, offering significant timing improvements. For even greater improvements, X-Lane masks timer interrupts during the execution of its services. Masking these IRQs however does trigger warnings from the read, copy, update (RCU) stall detector that preempts the tasks on the masked cores.

RCU warnings. The RCU stall detector issues a warning if a core is looping (1) in an RCU read-side critical section or (2) with interrupts and preemptions disabled. The stall detector triggers these warnings, i.e., time-wise unpredictable offloadable callbacks, once its grace period is over.

The RCU stall detector issues warnings to X-Lane's core as a side-effect of masking timer (and other) interrupts on them. X-Lane thus offloads RCU callbacks to other cores by configuring the kernel with the `CONFIG_RCU_NOCB_CPU=y` option and adding X-Lane's core to the `rcu_nocbs` kernel boot parameter. Further, less callbacks are triggered and offloaded by increasing the grace period of the RCU stall detector set in the `rcu_cpu_stall_timeout` kernel boot parameter.

Unmaskable SMIs. System management interrupts (SMIs) are x86-specific unmaskable interrupts that force *all* cores to switch to system management mode to run safety-related tasks. These thus monopolize all cores for up to milliseconds, creating jitter. Some SMIs are critical to the safety of the system/HW such as the ones forcing cores throttling to prevent overheating and HW damage. These SMIs however are rare and typically do not happen in nominal scenarios.

To prevent SMIs and still protect system health, core throttling is disabled in the BIOS and X-Lane manages fans itself.

4.3.3 Packet Transfer Between X-Lane's Core and NIC

Sending and, in particular, receiving packets on an endhost is not a task as straightforward as on a switch. The complexity of this task lies within the memory management and device management modules of the Linux kernel that contain design decisions typically favoring fairness, i.e., reducing overall latencies, over prioritizing accesses for select applications.

To reduce latency and jitter, X-Lane optimizes (1) how packets are copied between X-Lane's core, that packs outgoing and unpacks incoming packets, and a NIC, that encodes/decodes packets to/from the wire, and how (2) these two devices notify one another that a packet is ready to be handled by the other.

Packet copy. When booting, the NIC's driver initializes a queue on the NIC for outgoing packets waiting to be sent (i.e., TX ring buffer), and two queues for received packets waiting to be processed by a CPU core (i.e., RX ring buffers): one on the NIC and one in the main memory. Queues hosted on the NIC are accessible by every CPU via DMA over PCIe. However, different cores experience different access timings since computer architectures nowadays have non-uniform memory accesses (NUMA). As such, both CPU and the main memory are split into several NUMA nodes; memory accesses and device accesses via PCIe within the same NUMA node are faster than across nodes as the latter are forced to use the slower QuickPath interconnect (QPI) link.

X-Lane operates its *dedicated RX ring buffers*, one on the NIC and one in the main memory (X-Lane queues in Fig. 2), for packets received on the lane to prevent jitter from the regular system packets' head-of-line blocking. The TX ring buffer remains unaffected as there is no risk of head-of-line blocking when the NIC transmits packets. In addition, X-Lane selects its dedicated core such that it runs on the NUMA node that the NIC's PCIe lanes are connected to, thus avoiding the QPI link when performing a DMA to the NIC to send or receive packets.

Packet notification. While the NIC constantly polls its local TX ring buffer, populated by cores, and thus does not need any extra step to send packets, the NIC driver running on a core must be informed by the NIC that a packet is waiting to be processed in an RX ring buffer. The driver can be notified by: (1) receiving an IRQ sent by the NIC for each received packet, which is fast but inefficient for bursty traffic that creates a lot of IRQ masks, or (2) regularly polling the NIC's RX ring buffer (as with DPDK [21]), that fetches packets in batches but incurs a latency penalty for older packets (at the front of the queue) and for low polling frequencies.

X-Lane uses the IRQ-based approach to optimize delivery timing. X-Lane's core is not subject to bursty IRQs as the bandwidth is carefully managed and smoothened by the X-Lane controller (cf. §2.2). As shown in Fig. 2, a NIC receiving a packet sends an IRQ to X-Lane's core, set with a fitting IRQ mask, using receive flow steering [47] (step 1). In response, X-Lane's core timestamps the packet, doing it as early as possible to minimize pre-stamping jitter, and copies the packet via DMA from X-Lane's queue in the NIC to X-Lane's queue in the main memory to prepare it for inspection (step 2). X-Lane then shares the packet timestamp with the application via the X-R bridge and only delivers the unpacked payload once it has been inspected (step 3.a), also via the X-R bridge. X-Lane does not change how packets are handled on the regular system, e.g., with NAPI, DPDK (step 3.b).

4.3.4 Endhost Implementation Discussion

Additional work *in* the kernel would further improve the readiness of the implementation. For instance, X-Lane is currently limited by the granularity of some kernel boot parameters that affect all cores (e.g., disabling the NMI watchdog) and would benefit from per-core feature selection to better isolate its core. Further, most of these features are statically set at boot time, or even compile time. A dynamic configuration would help X-Lane's adaptation at runtime, reducing its endhost footprint when X-Lane is unused. Ideally, we would be able to fully isolate cores at runtime to greatly improving X-Lane's efficiency both in terms of endhost resource utilization and implementation effort.

X-Lane currently uses one core but can scale to multiple without introducing delays as long as they are in the same NUMA node. The implementation currently focuses on Intel Xeon architecture, but AMD's EPYC has fewer NUMA nodes yet more cores, different memory management, and PCIe 4 that could improve X-Lane's performance.

4.4 Jitter in Endhost Specialized Hardware

As an alternative to endhost commodity HW, we propose an implementation of X-Lane on recent intelligent network devices (smartNICs) that completely avoid kernel-induced jitter since they are not managed by it.

Our implementation supports Netronome's smartNICs with NFP-4000 network flow processors. The NFP-4000 natively supports programs in microC, a dialect of C, and P4 [15] via a P4-to-microC transpiler. We chose microC to implement X-Lane's services on the NFP-4000-powered smartNIC as it is more expressive than P4 despite recent developments on the latter, e.g., microC can directly access packet processing, flow processing cores, internal and external memory units.

Following the NFP-4000's architecture [8], X-Lane components are running on a flow processing island that has 12 flow processors and its own memory to buffer packets. The number of flow processors used for X-Lane can be scaled on demand to match the traffic. Unlike the commodity HW implementation, here X-Lane has direct access to the packet processing pipeline and the ingress/egress buffers closest to the physical interface which greatly reduces the jitter associated to sending/receiving packets on endhosts (cf. §4.3.3).

5 Example Services Exploiting X-Lane

We propose two services (cf. §2.4), a failure detector (FD) service and a state machine replication (SMR) service, that exploit X-Lane to accelerate asynchronous protocols.These services are available for applications as part of the X-KM.

5.1 Failure Detector X-FD

We leverage a periodic multicast protocol (cf. §2.1) that resides at the core of X-Lane to propose a heartbeat-based FD, X-FD, with a heartbeat period T. Unlike $\mathcal{HB}$ [11] that outputs a vector of message counters to the application, X-FD tracks the state of remote processes in an alive table stored in the X-R bridge that can be read by any application.

X-FD operates in three successive steps. First, a user space application increments a timer value in the R-X bridge at least once per period T. Due to the jitter-prone nature of the application, the value update period must be much smaller than T (e.g., T/3 in §6.4). Second, X-FD reads the corresponding value once per T from the R-X bridge and uses it for the heartbeat message, which is sent through X-Lane every period. Finally, when the destination endhost receives the packet at the queue dedicated to X-Lane on the NIC, X-FD optimistically timestamps the packet (cf. §4.3.3) and, while the

packet's payload is being analyzed, the alive table is updated with sender IP, port and last alive message timestamp.

5.2 State Machine Replication X-Raft

We offer a second service by adapting Raft [45], a popular SMR protocol [35, 36], to X-Lane in the form of the X-Raft service — a faster version of Raft using the periodic multicast protocol (and Raft's acks).

We adapted the well known etcd Raft [4] without any structural modifications to the algorithm or to its different phases (e.g., leader election, log replication/recovery, membership).

X-Raft uses the R-X bridge to enable an application to interact with the SMR (e.g., to propose a value) and uses the X-R bridge to notify the application. Leader election and consensus rounds are performed in X-Lane without interacting with the application.

X-Raft uses X-FD to detect process failure and initiate leader reelection if needed. Throughput-oriented log replication packets are sent via a lower-priority sub-lane with a very small period while commit statements are piggybacked on X-FD's low-jitter periodic messages. In addition, X-Raft batches parallel consensus instances in one packet akin to other consensus protocols [57]. Timeouts are greatly reduced thanks to X-Lane's low latency.

The log hosted by the leader is a buffer for uncommitted inputs; an input i is removed from the log when all replicas commit to a state that includes i. X-Raft uses a ring buffer for the log that is big enough to store the logs long enough for all replicas to commit a state or fail. The commit state pointer on the ring buffer is updated when replicas commit a new state.

6 Evaluation

In this section we assess the performance of X-Lane by first evaluating the latency and jitter of the underlying switching HW (§ 6.1), followed by extensive evaluation of X-Lane's communication timings (§ 6.2) and their variability (§ 6.3). We then evaluate the X-Lane-enabled services by measuring latency and accuracy of the FD service (§ 6.4), and latency and throughput of the SMR service both in isolation and once integrated in the Redis key-value store (§ 6.5).

Tab. 1 presents an overview of the implementation efforts behind each endhost component of X-Lane.

6.1 Hardware Setup

We ran our evaluation in a production data center of SAP SE hosting Arista 7280CR-48 [3] switches and 17 servers with Intel Xeon E5-2680 v4 at 2.40GHz (26 cores, 52 threads), 1 TB RAM, Mellanox ConnectX-4 4x10 GbE [7] and Intel XL710 4x10 GbE [6] as commodity NICs, and Netronome Agilio CX 2x10 GbE [8] smartNICs.

Table 1: Number of lines of code for each X-KM component.

Core component	#LoC	Service (cf. § 5)	#LoC
Controller client	476	X-FD	223
NIC bridge	515	X-Raft	843
SmartNIC bridge	163		

Switches' timing impact. We evaluated the impact of switches on latency and jitter by running multiple benchmarks with varying numbers of switches between endhosts. We observed a stable latency overhead per switch of 3 µs for unicast and 6 µs for multicast with no measurable jitter beyond this difference, as expected [23]. We also evaluated the accumulated impact of switches in common data center topologies [5], by running benchmarks up to a 4-hop topology; it only impacted latency, not on jitter. For this reason, we evaluated X-Lane and its services on a 1-hop topology. This topology simulates in-rack computing that represents the majority of communication in optimized systems [5].

Note that the Arista 7280CR-48 switches we used are much slower than, for instance, switches from the Arista 7150 series with processing times of 350 ns according to their data sheet [2]. *Theoretically*, such switches could thus reduce the latency of our setup by at least 2.6 µs, without affecting jitter.

6.2 Timing Observations

Most related works focus on reducing overall latency and maximizing network utilization, this work emphasizes jitter as another, crucial, dimension for many applications and in particular coordination tasks. Hence, we compare latency and jitter of three variants of X-Lane to each other, against QJump [24], and DPDK [21]. DPDK was used at a lower level by, and thus frames the performances of, many related works on low latency (e.g., Homa [44], Fastpass [48]), high performance OSs (e.g., IX [14], ZygOS [50]), and high performance SMRs (e.g., HovercRaft [33]) (cf. § 7).

Setup. We compare five configurations — DPDK, QJump, and three variants of X-Lane. The two main variants are specific to the used HW, and the third serves as a baseline:

X-Lane$_{\text{SNIC}}$: X-Lane on intelligent network devices;
X-Lane$_{\text{COM}}$: X-Lane on commodity HW;
X-Lane$_0$: X-Lane$_{\text{COM}}$ with modifications made to CPU scheduling (cf. § 4.3.1) and interrupts (cf. § 4.3.2) disabled.

We report DPDK's values using default settings as its maximum jitter did not vary measurably when varying its settings, only the number of packets with such high jitter.

We measured latency and jitter of the periodic unicast protocol on all configurations. We report latency as the time between a process sending a packet and the receiving process timestamping said packet. Sender and receiver processes

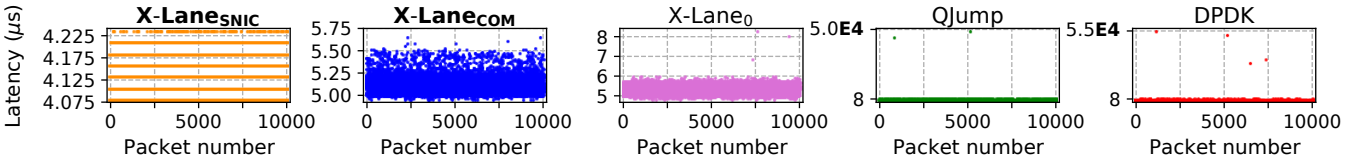

Figure 4: Overview of 10,000 packet latency (in µs) on the three X-Lane variants, QJump and DPDK. Note y-axes greatly vary.

Table 2: Summary of X-Lane's timings showing 0th, 50th, 99th, 100th latency λ percentiles (in µs), maximum jitter δ_{max} (in ns) from λ_{min}, and a metric based on probability bound (i.i.d. assumption) for $10 \times \lambda_{99th}$ violation over next 100,000 packets. Replacing our Arista 7280CR-48 by an Arista 7150 could in theory reduce all latencies by 2.6 µs (cf. §6.1).

Approach	λ_{min}	$\lambda_{50^{th}}$	$\lambda_{99^{th}}$	λ_{max}	δ_{max}	$P^{100,000}_{10\cdot\lambda\ 99^{th}}$
X-Lane$_{SNIC}$	4.082	4.133	4.234	4.234	152	0.104
X-Lane$_{COM}$	4.938	5.130	5.446	5.649	655	0.301
X-Lane$_0$	4.789	5.351	5.823	8.247	3.2**E3**	0.823
QJump	4.270	7.702	5.1**E2**	4.8**E4**	4.8**E7**	1.000
DPDK	4.103	8.904	4.0**E2**	5.4**E4**	5.4**E7**	1.000

are co-located on the same server to avoid cross-server clock skew; packets are still sent though the network. Processes sent packets with a 1 s period for QJump and DPDK due to high jitter, and a 10 ms period for X-Lane.

Dataset. The runs resulted in 181,440,000 packets for each approach, sampled over 21 days in a production data center of SAP SE. X-Lane variants ran with substantial cross-traffic and varying endhost utilization (up to an average CPU usage of 90%) while DPDK and QJump ran on an idle network of idle endhosts, setting the bar much higher for X-Lane. All possible point-to-point connections between servers were evaluated.

Latency and jitter results. Overall the results reveal: (1) holistic approaches (X-Lane$_{SNIC}$, X-Lane$_{COM}$) perform better than network-focused ones (X-Lane$_0$, QJump) and endhost-focused ones (DPDK), (2) offloading X-Lane to smartNICs (X-Lane$_{SNIC}$) further improves timings compared to the already efficient commodity HW approach (X-Lane$_{COM}$).

Tab. 2 overviews the timing measurements while Fig. 5 complements the table by exhibiting the main percentiles of the packet jitter distribution of each configuration. Even when running on commodity HW, X-Lane$_{COM}$ shows great performance benefits compared to QJump and DPDK, e.g., 1.501× and 1.735× lower median latency, and 72,758× and 81,816× lower maximum jitter, respectively. Unsurprisingly, the results indicate that offloading X-Lane to an intelligent network device achieves the best results across the board. Compared to X-Lane$_{COM}$, X-Lane$_{SNIC}$ achieves 1.241× lower median latency and 4.377× lower maximum jitter. As jitter is the most

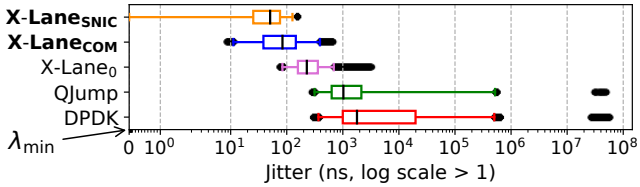

Figure 5: Distribution of X-Lane's packet jitter δ (in ns, log scale for data > 1). A jitter of 0 corresponds to the packet(s) with minimum latency λ_{min} within a dataset. Boxes are 25th/75th percentiles, black bars are medians, whiskers are 1st/99th percentiles, further data points are grayed out.

important factor for coordination tasks in distributed systems, X-Lane shows its drastic reduction of maximum jitter makes it a prime candidate for such tasks (cf. §6.4, §6.5). The difference in timings between X-Lane$_0$ and X-Lane$_{COM}$ shows the importance of tuning on endhost commodity HW (cf. §4.3) to reduce maximum jitter, i.e., tail latencies.

Fig. 4 further shows the individual latency of 10,000 packets among the highest outliers. Some packets for QJump and DPDK, i.e., the "outliers" in Fig. 5, dramatically increase the jitter implying all the bad side-effects for coordination.

6.3 Latency Bound Tightness

We study the variability of the results obtained after 21 days of sampling in §6.2 to determine the tightness of X-Lane's bounds. We first focus on packets whose latencies are beyond the 99th percentile, then propose an extrapolation using a simple probability-based metric.

Beyond the 99th percentile. Fig. 6 depicts percentiles characteristic of tail latency based on the sampled dataset. DPDK, which has the highest λ_{avg}, makes one jump at the 99.98th percentile. At the 99.997th percentile, we see once again that as more of QJump's "outliers" are taken into account, there is a sharp increase in tail latency. All X-Lane variants exhibit a stable behavior with X-Lane$_{SNIC}$ being the most stable followed by X-Lane$_{COM}$ and X-Lane$_0$. Another indication that X-Lane fully bounds the latency is the relative jitter defined as $(\lambda_{max} - \lambda_{min})/\lambda_{avg}$. While the relative jitter is ≈ 0.02 for X-Lane$_{SNIC}$, ≈ 0.13 for X-Lane$_{COM}$, and ≈ 0.36 for X-Lane$_0$, the values for DPDK and QJump are orders of magnitude higher: $\approx 1,807$ and $\approx 1,113$, respectively.

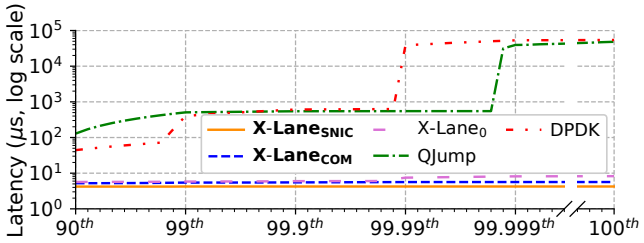

Figure 6: Tail latencies at different percentiles (different numbers of "nines") observed over 21 days.

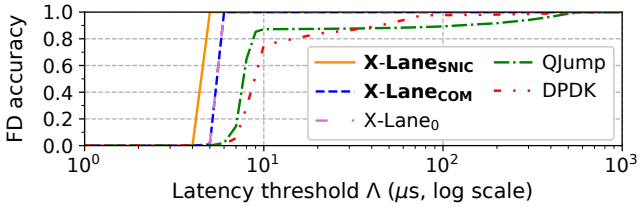

Figure 7: Accuracy of X-FD achieved when varying the latency threshold Λ. An alive process is incorrectly suspected of having failed if its heartbeat latency is greater than Λ.

Probability-based metric. We consider as a metric the probability of having among the next N packets *at least one* with latency exceeding λ, $\lambda > \lambda_{avg}$. We cannot get that probability's true value, so we use instead an upper bound P_λ^N under a simplifying assumption that the law of large numbers applies; i.e., packet latencies are independent and identically distributed, and we have performed enough experiments for sample mean λ_{avg} and variance σ^2 to be close to their true values. We derive the probability bound P_λ^1 for a single violation from the following tail-bound: $P_\lambda^1 \leq \sigma^2/(\lambda-\lambda_{avg})^2$. By using an independence assumption we further get $P_\lambda^n \leq 1-(1-P_\lambda^1)^n$. P_λ^n is a rough bound used only as a metric: the smaller its value is for an approach, the less that approach is prone to outliers.

Tab. 2 shows the probability to violate an SLO of $10 \times \lambda_{99^{th}}$ over 100,000 packets. The results support a greater reliability of X-Lane's measured latency over that of QJump and DPDK.

6.4 Failure Detector X-FD

We implemented the X-FD service (cf. § 5.1) atop all five configurations described in § 6.2 to compare the accuracy and completeness they provide in practice. We ran X-FD with 17 servers and a heartbeat period T of 1 ms whose value is incremented in an application every T/3. We varied the latency threshold Λ after which a process p is suspected of failure by others if no message was received from p in Λ.

Fig. 7 shows the rate of correct detection (i.e., accuracy) of the FDs with various threshold Λ (i.e., timeliness of completeness). We omitted T in the computation of the threshold. In practice, X-FD implemented on X-Lane reached perfect accuracy with practical thresholds well below 8 µs, and even below 5 µs for X-Lane$_{SNIC}$. QJump reaches ≈ 90% accuracy within 10 µs but struggles for *a few milliseconds* for the remaining 10% needed for perfect accuracy. DPDK takes longer.

These results mean for instance that X-Lane can detect leader failures (e.g., in Raft [45]) orders of magnitude faster than its "low-latency" counterparts. Re-elections can promptly start hence greatly improving liveness.

6.5 State Machine Replication X-Raft

We implemented X-Raft (cf. § 5.2) using X-Lane$_{COM}$ and evaluated it against etcd Raft [4] by measuring the latency and throughput of write requests (i.e., operations) in groups of 3 to 9 processes, one per server. The configuration was evaluated by having an application send write requests to the group. Latencies were measured as the time between the user space sender emits a request and the time it is available for all user space applications in the group. Accesses to the log, hosted in a RAM disk, were thus not included in the latencies. The sender emits once 10 M write requests whose size follows a truncated normal distribution: min = 1 B, max = 10 MB and observed mean = 25.6 B, standard deviation = 10 B.

Fig. 8 shows X-Raft performs much better than etcd both in terms of average latency, 15.7 µs for X-Raft, 26 ms for etcd, and average throughput, 96 MB/s for X-Raft, 1.1 MB/s for etcd. We note that, compared to a unicast protocol, X-Raft experiences 3 µs of added delay due to the switch processing multicast (cf. § 6.1). Unlike etcd, X-Raft batches requests before sending them and relies on multicast that scales well with regard to group size. etcd's bandwidth requirement however is linearly proportional to group size.

Treating write requests as operations, with 25.6 B mean request size, X-Raft achieves 3.7 M ops/s mean throughput. As a comparison, HovercRaft [33] achieves 1 M ops/s with 24 B requests but uses programmable switches, and NOPaxos [40] achieves 250 k ops/s (unknown size) but centralizes traffic.

Redis integration. To evaluate the genericity of X-Lane, we replaced the default inconsistent replication protocol of the Redis key-value store [9] with X-Raft. The result, a strongly consistent replicated key-value store, only took 26 lines of code of integration. Fig. 8 shows latency and write throughput for Redis and Redis+X-Raft with 3-9 servers. X-Raft reduces latency 18× on average and increases throughput 1.5×.

7 Related Work

Distributed coordination and failure detection. Over the years, several authors have explored improvements of coordination for distributed systems but only considering individual components or specific problems. Seminal works like mostly-ordered multicast [49] and unreliable ordered multicast [40] are multicast approaches where the ordering is done at the switches. Both approaches greatly improve the

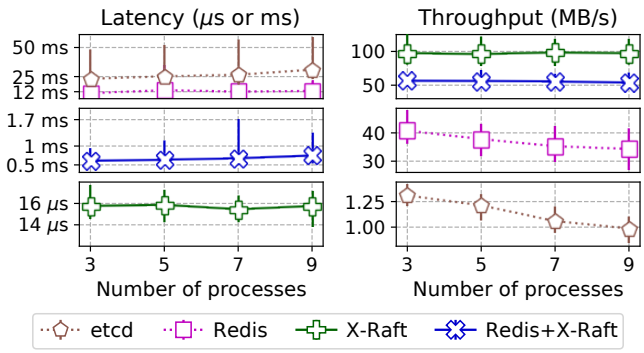

Figure 8: Write latency and throughput of X-Raft, etcd Raft, and Redis stand-alone vs with X-Raft. Mean values are plotted with min-max vertical bars.

Paxos [37] consensus protocol thanks to in-network ordering. R2P2 [34]-based HovercRaft [33], NetPaxos [17], and Consensus in a Box [29] similarly leverage switches for consensus protocols; like the Albatross [38] membership service, they do not give guarantees under an overloaded network. Their main goal is to speed up resolution of individual services via specific switch instrumentation, without considering other instances of the same protocol, other such protocols, or the network as a whole. Additionally, these approaches do not include controlled interaction to the endhosts' user space required for many jitter-bounded applications (e.g., FDs).

Silo [32] shows feasibility of guarantees without constraining network elements; the guarantees provided are however not strong enough for applications like FDs in terms of jitter and packet loss. Falcon [39] focuses on what the network needs to provide to implement a perfect (reliable) FD, rather than how it can do so, and resorts to program-controlled crashes when the FD falsely suspects processes of being crashed due to missed timeouts, contradicting reliability.

Low latency. In recent years there were numerous proposals for achieving low latency network communication. The introduced approaches typically bound latency at the 99th percentile. The reason for the 99th percentile is that it is hard to deal with the sources of jitter in a complex system (cf. §4.3). Tails of the tail [41], a seminal work in this area, identifies major jitter sources on endhosts, but does not consider the network, and focuses on 99th and 99.9th percentile latency, not 100th. Another path leading work is QJump [24] which proposes to achieve bounded latency on commodity hardware, but focuses on queues' priorities for low latency delivery and does not consider sources of jitter on endhosts (cf. §4.3).

The DPDK framework is known for its fast and efficient poll mode drivers and fast packet processing capabilities. It has a wide range of driver implementations for various NICs. The DPDK developers have restructured and implemented a majority of the network device driver code and structure. DPDK operates by polling the network device from the user space application, which allows the programmer to harvest network packets bypassing the kernel network stack completely. As mentioned in §6.2 many works build on DPDK, e.g., Homa [44], Fastpass [48], IX [14], ZygOS [50]. These approaches try to optimize utilization and 99th percentile latency. Thus, they could be applied for the regular system but as shown in §6.3 are insufficient for X-Lane.

Time synchronization. DTP [53], Huygens [22] and Sundial [42] are time synchronization schemes for data centers with precision below 100 ns. However, time synchronization alone does not enable interactions with bounded latency.

Endhost synchrony. Efforts on achieving real-time (RT) guarantees for commodity OSs like Linux are related to X-Lane. RTLinux [51] is a real-time OS microkernel running the entire Linux OS as a fully preemptive process. RTLinux treats every process as having RT requirements, while X-Lane can treat a process in fair scheduled manner, or with even stronger RT guarantees; traditional RT schedulers, e.g. EDF [43], can actually not guarantee that a specific task is performed by a given deadline, as they can not predict the system environment and are influenced by system service executions.

8 Conclusions

X-Lane implements unprecedented low latency and jitter for coordination interaction crucial to the liveness of many applications in data centers. As this is not needed for all types of distributed interaction, X-Lane confines these bounds to an express lane, which is carefully isolated from the regular existing environment for best-effort traffic both in the network and at the endhosts. X-Lane's original design uses commodity SW and HW, and smartNICs when available.

X-Lane opens up many avenues for future research, e.g., which parts of an application best benefit from X-Lane, how to design and optimize coordination protocols accordingly. We are exploring extensions and refinements of our work such as expanding the endhost implementation (cf. §4.3), adding services (e.g., clock synchronization), and enhancing X-Lane's safety towards practical synchronous services.

Acknowledgments

We thank the anonymous reviewers and our shepherd Dan Ports for their valuable feedback. This work was partially funded by ERC Consolidator grant #617805 (LiveSoft), DFG Center #1053 (MAKI), SNSF grant #200021_192121 (FORWARD), SNSF grant #200021_197353 (BASIS), NSF grant #1618923, Hasler Foundation, and a Facebook Distributed Systems Research Award.

References

[1] Apache HBase. http://hbase.apache.org/.

[2] Arista 7150 Series. https://www.arista.com/assets/data/pdf/Datasheets/7150S_Datasheet.pdf.

[3] Arista 7280R Series. https://www.arista.com/assets/data/pdf/Datasheets/7280R-DataSheet.pdf.

[4] etcd. https://github.com/etcd-io/etcd/.

[5] F16 - Facebook's topology. https://engineering.fb.com/data-center-engineering/f16-minipack/.

[6] Intel XL710. https://www.intel.com/content/dam/www/public/us/en/documents/datasheets/xl710-10-40-controller-datasheet.pdf.

[7] Mellanox ConnectX-4. http://www.mellanox.com/related-docs/prod_adapter_cards/PB_ConnectX-4_VPI_Card.pdf.

[8] Netronome NFP-4000 network processor. https://www.netronome.com/static/app/img/products/silicon-solutions/WP_NFP4000_TOO.pdf.

[9] Redis. https://redis.io/.

[10] TiKV. https://github.com/tikv/tikv/.

[11] Marcos K. Aguilera, Wei Chen, and Sam Toueg. Heartbeat: A Timeout-Free Failure Detector for Quiescent Reliable Communication. In *Distributed Algorithms*, pages 126–140, 1997.

[12] Apache. Hadoop. https://hadoop.apache.org.

[13] Apache Software Foundation. Hadoop Distributed File System. http://hadoop.apache.org/.

[14] Adam Belay, George Prekas, Ana Klimovic, Samuel Grossman, Christos Kozyrakis, and Edouard Bugnion. IX: A Protected Dataplane Operating System for High Throughput and Low Latency. In *11th USENIX Symposium on Operating Systems Design and Implementation*, OSDI '14, pages 49–65, 2014.

[15] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, and David Walker. P4: Programming Protocol-Independent Packet Processors. In *ACM SIGCOMM Computer Communication Review*, volume 44, pages 87–95, July 2014.

[16] Manuel Bravo, Nuno Diegues, Jingna Zeng, Paolo Romano, and Luis ET Rodrigues. On the use of Clocks to Enforce Consistency in the Cloud. In *IEEE Data Eng. Bull.*, volume 38, pages 18–31, 2015.

[17] 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 5:1–5:7, 2015.

[18] Shuhaizar Daud, R Badlishah Ahmad, Ong Bi Lynn, Zahereel Ishwar Abd Kareem, Latifah Munirah Kamaruddin, Phaklen Ehkan, Mohd Nazri Mohd Warip, and Rozmie Razif Othman. The Effects of CPU Load & Idle State on Embedded Processor Energy Usage. In *2nd International Conference on Electronic Design*, ICED '14, pages 30–35, 2014.

[19] Jeffrey Dean and Luiz André Barroso. The Tail at Scale. In *Communications of the ACM*, volume 56, pages 74–80, 2013.

[20] Apache Foundation. Apache Accumulo. https://accumulo.apache.org.

[21] Linux Foundation. DPDK: Data Plane Development Kit. https://www.dpdk.org.

[22] Yilong Geng, Shiyu Liu, Zi Yin, Ashish Naik, Balaji Prabhakar, Mendel Rosenblum, and Amin Vahdat. Exploiting a Natural Network Effect for Scalable, Fine-grained Clock Synchronization. In *15th USENIX Symposium on Networked Systems Design and Implementation*, NSDI '18, pages 81–94, 2018.

[23] Phillipa Gill, Navendu Jain, and Nachiappan Nagappan. Understanding Network Failures in Data Centers: Measurement, Analysis, and Implications. In *Proceedings of the 2011 Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '11, pages 350–361, 2011.

[24] 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 *Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation*, NSDI '15, pages 1–14, 2015.

[25] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D. Joseph, Randy Katz, Scott Shenker, and Ion Stoica. Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center. In *Proceedings of the 8th USENIX Symposium on Networked Systems Design and Implementation*, NSDI '11, pages 295–308, 2011.

[26] Chi-Yao Hong, Srikanth Kandula, Ratul Mahajan, Ming Zhang, Vijay Gill, Mohan Nanduri, and Roger Wattenhofer. Achieving High Utilization with Software-driven WAN. In *ACM SIGCOMM Computer Communication Review*, volume 43, pages 15–26, 2013.

[27] Chi-Yao Hong, Subhasree Mandal, Mohammad Al-Fares, Min Zhu, Richard Alimi, Kondapa Naidu B., Chandan Bhagat, Sourabh Jain, Jay Kaimal, Shiyu Liang, Kirill Mendelev, Steve Padgett, Faro Rabe, Saikat Ray, Malveeka Tewari, Matt Tierney, Monika Zahn, Jonathan Zolla, Joon Ong, and Amin Vahdat. B4 and After: Managing Hierarchy, Partitioning, and Asymmetry for Availability and Scale in Google's Software-Defined WAN. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '18, pages 74–87, 2018.

[28] Patrick Hunt, Mahadev Konar, Flavio Paiva Junqueira, and Benjamin Reed. ZooKeeper: Wait-free Coordination for Internet-scale Systems. In *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference*, volume 8 of *USENIX ATC '10*, pages 145–158, 2010.

[29] Zsolt István, David Sidler, Gustavo Alonso, and Marko Vukolic. Consensus in a Box: Inexpensive Coordination in Hardware. In *13th USENIX Symposium on Networked Systems Design and Implementation*, NSDI '16, pages 425–438, 2016.

[30] Patrick Jahnke, Vincent Riesop, Pierre-Louis Roman, Pavel Chuprikov, and Patrick Eugster. Live in the Express Lane (project website). https://github.com/patrickjahnke/X-Lane.

[31] Sushant Jain, Alok Kumar, Subhasree Mandal, Joon Ong, Leon Poutievski, Arjun Singh, Subbaiah Venkata, Jim Wanderer, Junlan Zhou, Min Zhu, Jon Zolla, Urs Hölzle, Stephen Stuart, and Amin Vahdat. B4: Experience with a Globally-deployed Software Defined WAN. In *Proceedings of the 2013 Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '13, pages 3–14, 2013.

[32] Keon Jang, Justine Sherry, Hitesh Ballani, and Toby Moncaster. Silo: Predictable Message Latency in the Cloud. In *ACM SIGCOMM Computer Communication Review*, volume 45, pages 435–448, August 2015.

[33] 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 25:1–25:17, 2020.

[34] Marios Kogias, George Prekas, Adrien Ghosn, Jonas Fietz, and Edouard Bugnion. R2P2: Making RPCs first-class datacenter citizens. In *2019 USENIX Annual Technical Conference*, USENIX ATC '19, pages 863–880, 2019.

[35] Leslie Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Communications of the ACM*, 21(7):558–565, July 1978.

[36] Leslie Lamport. Using Time Instead of Timeout for Fault-Tolerant Distributed Systems. In *ACM Transactions on Programming Languages and Systems*, pages 254–280, April 1984.

[37] Leslie Lamport. The Part-Time Parliament. In *ACM Transactions on Computer Systems*, volume 16, pages 133–169, May 1998.

[38] Joshua B. Leners, Trinabh Gupta, Marcos K. Aguilera, and Michael Walfish. Taming Uncertainty in Distributed Systems with Help from the Network. In *Proceedings of the Tenth European Conference on Computer Systems*, EuroSys '15, pages 9:1–9:16, 2015.

[39] Joshua B. Leners, Hao Wu, Wei-Lun Hung, Marcos K. Aguilera, and Michael Walfish. Detecting Failures in Distributed Systems with the Falcon Spy Network. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 279–294, 2011.

[40] Jialin Li, Ellis Michael, Naveen Kr. Sharma, Adriana Szekeres, and Dan R. K. Ports. Just Say NO to Paxos Overhead: Replacing Consensus with Network Ordering. In *12th USENIX Symposium on Operating Systems Design and Implementation*, OSDI '16, pages 467–483, 2016.

[41] 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, pages 1–14, 2014.

[42] Yuliang Li, Gautam Kumar, Hema Hariharan, Hassan Wassel, Peter Hochschild, Dave Platt, Simon Sabato, Minlan Yu, Nandita Dukkipati, Prashant Chandra, and Amin Vahdat. Sundial: Fault-tolerant Clock Synchronization for Datacenters. In *14th USENIX Symposium on Operating Systems Design and Implementation*, OSDI '20, pages 1171–1186, 2020.

[43] Chang L. Liu and James W. Layland. Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment. In *Journal of the ACM*, volume 20, pages 46–61, 1973.

[44] Behnam Montazeri, Yilong Li, Mohammad Alizadeh, and John K. Ousterhout. Homa: A Receiver-Driven Low-Latency Transport Protocol Using Network Priorities. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '18, pages 221–235, 2018.

[45] 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.

[46] Linux Kernel Organization. NO_HZ: Reducing Scheduling-Clock Ticks. https://www.kernel.org/doc/Documentation/timers/NO_HZ.txt.

[47] Linux Kernel Organization. Scaling in the Linux Networking Stack. https://www.kernel.org/doc/Documentation/networking/scaling.txt.

[48] Jonathan Perry, Amy Ousterhout, Hari Balakrishnan, Devavrat Shah, and Hans Fugal. Fastpass: A Centralized "Zero-queue" Datacenter Network. In *Proceedings of the 2014 Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '14, pages 307–318.

[49] Dan R. K. Ports, Jialin Li, Vincent Liu, Naveen Kr. Sharma, and Arvind Krishnamurthy. Designing Distributed Systems Using Approximate Synchrony in Data Center Networks. In *12th USENIX Symposium on Networked Systems Design and Implementation*, NSDI '15, pages 43–57, 2015.

[50] George Prekas, Marios Kogias, and Edouard Bugnion. ZygOS: Achieving Low Tail Latency for Microsecond-Scale Networked Tasks. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, pages 325–341, 2017.

[51] Federico Reghenzani, Giuseppe Massari, and William Fornaciari. The Real-time Linux Kernel: A Survey On Preempt_RT. In *ACM Computing Surveys*, volume 52, pages 1–36, February 2019.

[52] Laura S. Sabel and Keith Marzullo. Election Vs. Consensus in Asynchronous Systems. Technical report, Cornell University, 1995.

[53] Vishal Shrivastav, Ki Suh Lee, Han Wang, and Hakim Weatherspoon. Globally Synchronized Time via Datacenter Networks. In *IEEE/ACM Transactions on Networking*, volume 27, pages 1401–1416, 2019.

[54] Vinod Kumar Vavilapalli, Arun C. Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, Jason Lowe, Hitesh Shah, Siddharth Seth, Bikas Saha, Carlo Curino, Owen O'Malley, Sanjay Radia, Benjamin Reed, and Eric Baldeschwieler. Apache Hadoop YARN: Yet Another Resource Negotiator. In *Proceedings of the 4th Annual Symposium on Cloud Computing*, SoCC '13, pages 5:1–5:16, 2013.

[55] Paulo Verissimo and António Casimiro. The Timely Computing Base Model and Architecture. In *IEEE Transactions on Computers*, pages 916–930, 2002.

[56] Paulo E. Veríssimo. Travelling Through Wormholes: A New Look at Distributed Systems Models. In *SIGACT News*, pages 66–81, 2006.

[57] Maofan Yin, Dahlia Malkhi, Michael K. Reiter, Guy Golan Gueta, and Ittai Abraham. HotStuff: BFT Consensus with Linearity and Responsiveness. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*, PODC '19, pages 347–356, 2019.

[58] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, and Ion Stoica. Resilient Distributed Datasets: a Fault-Tolerant Abstraction for In-memory Cluster Computing. In *9th USENIX Symposium on Networked Systems Design and Implementation*, NSDI '12, pages 15–28, 2012.