A NEaT Design for Reliable and Scalable Network Stacks

Tomas Hruby Cristiano Giuffrida Lionel Sambuc Herbert Bos Andrew S. Tanenbaum
Vrije Universiteit Amsterdam
thruby@few.vu.nl,{giuffrida,lionel.sambuc,herbertb,ast}@cs.vu.nl

ABSTRACT

Operating systems provide a wide range of services, which are crucial for the increasingly high reliability and scalability demands of modern applications. Providing both reliability and scalability at the same time is hard. Commodity OS architectures simply lack the design abstractions to do so for demanding core OS services such as the network stack. For reliability and scalability guarantees, they rely almost exclusively on ensuring a high-quality *implementation*, rather than a reliable and scalable *design*. This results in complex error recovery paths and hard-to-maintain synchronization code.

We demonstrate that a simple and structured design that strictly adheres to two principles, *isolation* and *partitioning*, can yield reliable and scalable network stacks. We present NEaT, a system which partitions the stack across isolated process replicas handling independent requests. Our design principles intelligently partition the state to minimize the impact of failures (offering strong recovery guarantees) and to scale comparably to Linux without exposing the implementation to common pitfalls such as synchronization errors, poor locality, and false sharing.

1. INTRODUCTION

Building reliable and scalable OS services such as the network stack is a daunting task. Even when opportunities for reliability [43] and scalability [19] exist at the interface level, producing a reliable and scalable implementation is notoriously challenging [11, 14, 29]. This problem is exacerbated by the dominant *"build-and-fix"* model adopted in commodity OSes, which generally attempts to retrofit reliability and scalability in legacy implementations. This strategy—*reliability and scalability of implementation*—misses important opportunities to address key problems at design time and has trouble scaling with the fast-paced evolution of modern hardware and software [11].

This paper presents a new coordinated approach to the reliability and scalability of the network stack—one of the most demanding and dependability-sensitive subsystems. Our approach relies on two well-known key principles—*isolation* and *partitioning*. That means, we break large entities into smaller ones and isolate them from each other in such a way that they cannot unexpectedly interfere with execution of each other. What is new is that we apply them to an extreme degree in our design, demonstrating that such a strategy can (i) allow reliability and scalability to coexist and symbiotically improve with the number of available cores and (ii) greatly simplify and improve the longevity of the final implementation. To substantiate our claims, we present NEaT, a reliable and scalable componentized network stack implemented using a "clean slate" approach on top of the NewtOS microkernel-based architecture [37]. NEaT embraces our principles to isolate individual threads of execution in separate processes and transparently partition the network state (and user requests) across N independent network stack *replicas*, while preserving application-level sharing and standard BSD socket interfaces. Thanks to its replicated design, NEaT: (i) survives run-time failures with minimal service disruption, (ii) scales comparably to the Linux' network stack, and (iii) minimizes code explicitly dealing with reliability and scalability concerns—making it easier to maintain in the face of constant hardware and software changes.

We do not claim that our design (and its implementation) is the only one possible for a reliable and scalable network stack, let alone for generic operating system services. Rather, our goal is to demonstrate that reliability and scalability are not conflicting requirements and can be both effectively addressed at design time using well-defined principles. Experience shows that the alternative—reliability and scalability of implementation—is increasingly unsustainable, threatening the growing reliability and scalability demands of modern applications.

CoNEXT '16, December 12 - 15, 2016, Irvine, CA, USA

 ISBN 978-1-4503-4292-6/16/12...$15.00

DOI: http://dx.doi.org/10.1145/2999572.2999579

The Linux kernel is a case in point. With its code base growing from the original few thousand lines of code (LoC) to the millions of LoC and dozens of architectures today [42], maintaining high reliability and scalability standards has become increasingly prohibitive despite huge community efforts. For instance, replacing the big kernel lock with fine-grained locking and lockless data structures took more than 8 years [3]. In addition, the new synchronization mechanisms have been updated several times since then, after repeatedly exhausting their scalability opportunities on contemporary hardware [16]. Further, as the size and complexity of the kernel dramatically increase, so does the rate of faults in the code, with an average fault lifespan of around 1 year even for high-impact faults that can bring the system to a halt [51].

While NEaT is a research prototype and it may not be fair to compare it to full-featured and highly optimized systems such as Linux, its replicated network stack design based on isolation and partitioning does structurally solve many of the recurring implementation issues described above (§2). Similarly, while prior work has considered principles such as isolation and partitioning before—in different forms—for either reliability [29, 23, 38, 63, 43, 37, 21] or scalability [11, 14, 66, 26, 10], our design rigorously combines these principles together to address both reliability and scalability to demonstrate their effectiveness in a BSD-compliant network stack.

Contributions. To summarize, our contributions are:

- We present a new approach to design reliable and scalable network stacks based on the rigorous application of two principles— *isolation* and *partitioning*—and analyze the opportunities to solve challenging implementation problems in existing systems.
- We present the principle-based design and implementation of NEaT, a BSD-compliant network stack that rigorously isolates and partitions its individual components across N independent network stack replicas running on top of a microkernel-based architecture. We show that NEaT only marginally relies on the implementation to obtain the desired reliability and scalability properties. Furthermore, the full decoupling between replicas allows NEaT to assign independent user requests to a random replica, ensuring load balancing and, as a by-product, location unpredictability, resulting in improved security against emerging memory error attacks [13, 57]. NEaT demonstrates that supporting reliability and scalability (and enhanced security) by design is possible and, to our knowledge, is the first network stack of its kind.
- We evaluate NEaT using a representative web server application (lighttpd). Our results show that NEaT can handle up to 13-35% more requests than Linux, despite the underlying microkernel design being traditionally perceived as a major limiting factor to compete with commodity OS architectures.

Outline. The remainder of the paper is laid out as follows. §2 provides background information, highlighting problems in existing OS and network stack designs and comparing our approach to reliability and scalability with prior work. §3 and §4 present the design and implementation of the NEaT network stack. §5 compares to other userspace network stacks and §6 presents experimental results, assessing the reliability and scalability properties of our NEaT prototype and §7 concludes the paper.

2. BACKGROUND

Shaping the design of reliable and scalable systems using *isolation* and *partitioning* makes intuitive sense: (i) they enforce fault containment and conflict-free failure recovery for reliability purposes; (ii) they enforce data locality and conflict-free parallelism for scalability purposes. The vast majority of commodity operating systems—whose original design dates back to the single core computing era—however, opt for a monolithic architecture, where multiple threads typically coexist in a single address space. This design naturally induces a "*shared everything*" model, with no implicit isolation nor partitioning and, as a result, no reliability and scalability of design. In the next subsections, we develop this intuition further and highlight the fundamental limitations of this model and reliability and scalability of implementation in general.

2.1 Reliability

Reliability of Implementation. The lack of *isolation* in monolithic architectures complicates the implementation of effective fault containment mechanisms: a single fault can arbitrarily propagate throughout the kernel, corrupt arbitrary data structures, and lead the entire operating system to fail. To mitigate this problem, commodity operating systems such as Linux adopt a pragmatic approach to reliability, relying on dedicated error-handling logic or killing the offending process when a fault is detected (kernel *oops* [68]). Unfortunately, the former approach may also result in the introduction of a large amount of complex but trusted recovery code, which is often untested [8]—thus creating a vicious circle [29]—while the latter approach provides weak reliability with the inability to detect (or recover from) global error propagation—recently estimated to occur in more than 25% of the cases [68]. To improve fault containment, researchers have devised a number of techniques to retrofit isolation guarantees in existing kernel extensions and, in particular, device drivers. Some approaches rely on hardware-based isolation [63, 27, 17], others on

language- [69] or compiler-based strategies [40, 18], yet others on virtualization techniques [61]. While such approaches are generally effective in containing faults in untrusted components, they are typically limited to relatively small OS subsystems—recovery techniques for subsystems comparable to the network stack do exist, but at the cost of more complex recovery code [62]—and may fail to guarantee full isolation when the subsystem interacts with the rest of the kernel using nonstandard interfaces [40].

Furthermore, when a fault is detected, recovery actions are necessary to ensure that the system is in a globally consistent state. The lack of explicit state *partitioning* in monolithic architectures, however, induces "*hidden*" cross-thread dependencies that significantly complicate this process. For this reason, commodity operating systems generally have to resort to a best-effort failure recovery model [68]. To mitigate this problem, researchers have proposed manual state reconstruction [23]—which, however, introduces pervasive and hard-to-maintain recovery code—or software transactional memory-like schemes to selectively rollback all the threads that yield conflicting state changes with the faulting thread [43]—which, however, greatly limits the performance and scalability of the system [29]. Techniques that retrofit partitioning into monolithic kernels using virtualized domains have also been recently attempted [49], but at the cost of greatly limiting application-level sharing and cooperation.

Reliability of Design. A vast body of research has been devoted to principles supporting reliability of design. Isolation, in particular, is a well-established design principle in reliable OS architectures. Microkernel-based operating systems such as QNX [34], MINIX 3 [4], Sawmill [28], and Singularity [38], in particular, are structured around a number of hardware- or software-isolated processes communicating via message passing to support fault containment by design. NEaT is based on the partitioned network stack of NewtOS [37], a variant of MINIX 3, and supports different configurations. Unlike these systems, NEaT generally provides much stronger isolation guarantees—no multithreading—and relies on isolation to also support scalability by design, not only reliability. In addition, unlike NEaT, these systems do not generally partition or replicate state across components, with important scalability but also reliability drawbacks. For example, prior work demonstrated that most of these systems fail to recover from any failure in their stateful components [21]. Techniques devised to address this problem rely on state replication [21] or checkpointing [29]. Thanks to its state partitioning strategy, in contrast, NEaT can gracefully recover from failures by restarting the faulty replica, with no impact on the other replicas and thus minimal network state loss. Our recovery strategy is inspired by replication-based fault tolerance, a common design pattern in reliable distributed systems [30].

2.2 Scalability

Scalability of Implementation. To scale to the increasing number of available cores, network stacks in monolithic kernel architectures have grown to become massively threaded in modern implementations [67]. Although multithreading is a widely accepted strategy to achieve parallelism—and potentially scalability—the lack of *isolation* between threads comes at the cost of subordinating scalability to the implementation of complex synchronization mechanisms that grant safe access to shared data structures. Implementing provably correct synchronization primitives is challenging due to the complexity of modern hardware and compilers [24]. Implementing such primitives in a scalable way is even more challenging and also heavily dependent on the particular hardware [22].

Even widely deployed fine-grained locking primitives such as Linux' ticket spinlocks have been recently found plagued with scalability problems, with researchers showing that more scalable implementations such as MCS locks [46] are necessary to avoid dramatic performance drops on many-core architectures [16]. More scalable alternatives to locking include lockless data structures or lightweight synchronization mechanisms such as RCU [45]. While increasingly popular in the Linux kernel and especially in its network stack, RCU indirectly exemplifies the difficulties of implementing scalable and general-purpose synchronization primitives: it can only satisfy less than 8% of the entire kernel (9,000 uses) [6] and only provide scalable read-side semantics.

Further, monolithic architectures structured around a single address space allow all their threads to implicitly share arbitrary data structures. While a common programming abstraction, this approach enforces no explicit state *partitioning* and comes at the cost of subordinating scalability to the implementation's ability to limit sharing and preserve optimal data locality. In modern cache-coherent multicore architectures, this is crucial to prevent shared data from frequently (and unnecessarily) traveling between caches and hindering scalability. This is, for example, a well-known scalability bottleneck in many common synchronization primitive implementations [16].

To mitigate this problem, monolithic architectures strive to maintain data structures local to the core where they are most frequently used. Such implementation-driven strategies are, however, insufficient, since data structures still follow processes that migrate across cores—due to load balancing—and threads still share common data structures within the same process. A particularly insidious threat is *false sharing*, where different data structures that are not logically shared happen to reside on the same cache line, unnecessarily causing frequent—but silent—cache line bouncing [44]. gcc's controversial `__read_mostly` attribute exemplifies the difficulties of addressing this scalability problem at the implementa-

tion level: its adoption in the Linux kernel required 1554 annotations to group all the "*read-mostly*" variables together—preventing conflicts with frequently modified cachelines—but only to raise concerns that many remaining "*write-often*" variables may then increase cache line sharing—ultimately degrading write-side scalability [7].

Scalability of Design. While scalability of implementation is still a realistic—but already cumbersome—option today [15], many researchers have recognized the need for principles supporting scalability of design before, without, however, applying them in a radical way or also addressing reliability concerns. Tornado [26] and K42 [10] first argued for "*partitioning*, *distributing*, and *replicating*" data across independent (but not isolated) objects to improve locality on shared-memory multiprocessors. In a similar direction, Corey [14] proposes granting applications the ability to limit sharing of OS data structures and improve scalability. In all these systems, however, multithreading and sharing are still the norm, greatly limiting the opportunities offered by true isolation and partitioning. Corey [14] partitions the network stack state across multiple per-core replicas. Unlike NEaT, however, Corey's library OS-based design imposes strict partitioning at the application level as well, greatly limiting application-level sharing. More recent solutions such as IX [12] and Arrakis [53]— building on top of Dune and Barrelfish [11] multikernel architectures (respectively)—share similar issues imposed by the underlying library OS model.

Unlike all these systems, we will demonstrate that, in the context of a network stack in particular, a more radical design based on even stricter isolation and partitioning—no implicit or explicit sharing—is realistic and effective, with important scalability, but also reliability benefits. More recently, fos [66] also assessed the potential utility of replicating core OS services across cores, but without considering its reliability benefits or evaluating scalability at high bit and request rates. Finally, recent work advocates reconsidering scalability as a property of the interfaces rather than of the implementation, proposing a *commutativity rule* to assess the scalability of high-level operations [19]. Our work is similar in that we advocate for reconsidering the drive for scalability of implementation, but also complementary in that we investigate scalability at the design rather than at the interface level.

Finally, scalable network stacks such as IsoStack [58], MegaPipe [31], and mTCP [39] focus solely on scalability with no emphasis on reliability. They also take a less radical approach to scalability with limited isolation and partitioning as their main goal is to circumvent the performance limitations imposed by monolithic architectures. However, the userspace library stacks such as mTCP can also run in a multiserver OS on top of a microkernel with similar benefits and limitations as in a monolithic system.

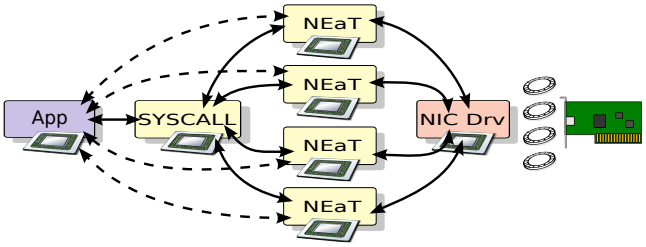


Figure 1: NEaT: a 4-replica example.

3. A RELIABLE AND SCALABLE NETWORK STACK

We present NEaT, the first network stack designed from the ground up for reliability and scalability. To enforce isolation of its components, NEaT relies on a microkernel-based architecture with all the core OS components running as event-driven and hardware-isolated processes. Thanks to such isolation guarantees, each component of the network stack can be provided with fault containment capabilities and also assigned its own core. While beneficial for reliability, this design is highly unsatisfactory for scalability purposes.

The key scalability problem of a system executing as a set of isolated processes is that any of its components may easily get overloaded, even when assigned an entire CPU core for itself. We rule out having multithreaded components spanning multiple cores as this strategy would impose the same scalability limitations evidenced in monolithic architectures. For this reason, NEaT opts for a radically different design, that is partitioning the network state across a number of isolated processes replicated from the original components of the network stack (replicas).

NEaT's design eliminates *implicit* synchronization and sharing both within the individual processes (by disallowing threading) and across generic OS processes (by allowing communication only via message passing). The latter guarantees that each process always modifies only its own data structures—except the messaging queues. At the same time, NEaT's design also eliminates *explicit* synchronization and sharing across stack replicas (by disallowing communication) so that the network stack can both scale and simplify failure recovery. A failing replica does not prevent other replicas from continuing running undisturbed, causing minimal service disruption and state loss.

3.1 Overview

Figure 1 presents the high-level overview of NEaT, which internally partitions the network state across several replicas (4 in the example). Each replica communicates with each NIC driver and, by default, applications communicate with all the replicas of the network stack as though only one logical instance were present—although it is possible to configure NEaT differently. This is

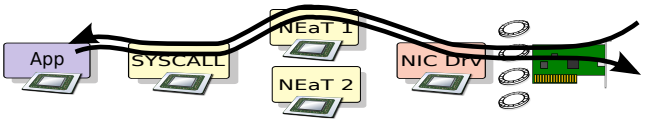


Figure 2: A dedicated path for packets of each connection.

transparent to the application programmer, since all the blocking system calls are routed through a dedicated system call server (SYSCALL). Our socket implementation (§3.2), however, allows applications to largely bypass the SYSCALL server and communicate directly with the assigned replica of the network stack (dashed lines in Figure 1). This complexity is hidden by our user-space POSIX library. The individual OS processes are assigned dedicated cores, allowing fast communication between OS components without intervention of the microkernel, an idea previously explored in the original NewtOS' network stack [37].

With no data sharing or synchronization across replicas, NEaT allows each network socket to live only in a single instance of the network stack. This is especially important for TCP, as each TCP connection requires the stack to maintain a large amount of state and NEaT must ensure that every packet of each connection uses the same path through the network stack (Figure 2). The applications (libraries), the SYSCALL server, and the network devices are responsible for selecting the network stack replica to handle each socket. fos proposes using a fleet coordinator for similar replica selection problems [66]. NEaT, in contrast, allows no process with a special role, since the intention is to avoid explicit communication between the processes of the network stack. Our solution is to delegate part of the data plane functionality to the hardware, similar, in spirit, to Arrakis [53] and IX [12]. In contrast to both projects, NEaT does not aim at separating independent instances of library systems or userspace stacks used by different applications. NEaT aims at separating collections of replicas that form a single network stack accessed by multiple applications.

Contemporary network devices already have the ability to match incoming packets by a set of rules, split the traffic, and steer packets to the intended network stack replica using multiple internal NIC queues. The NIC driver can thus dispatch the packets to the right replica based on the receive queue of the NIC. Although these modern features were originally motivated by the design of monolithic systems and virtualization, NEaT successfully exploits them to implement replica-aware connection management.

3.2 Sockets

While adhering to the POSIX API, NewtOS refrains from using a traditional implementation of network sockets for scalability reasons. Our socket implementation, which we described in detail in [35], requires the network stack and the applications to run on different hardware threads. While applications occasionally communicate with the SYSCALL server, NEaT resolves the vast majority of the system calls within the application itself, exposing socket buffers to the application level similar to user-space message queues.

The socket design makes the replication transparent to the applications as once a socket is open and the system establishes the corresponding shared-memory channel, an application can automatically communicate with the right network stack replica without exact knowledge of the particular replica on the other end. This makes the fast path completely agnostic to the number of network stack replicas, ensuring scalability to a large number of network stacks (and cores).

In contrast to monolithic systems, multiserver systems have a key and a little counterintuitive CPU load distribution difference. Our experience when running a loaded server on Linux—also confirmed by others [39]—indicates that it is realistic to expect approximately 70-80% of the execution to happen within the operating system (the kernel). If an application in Linux uses 30% of overall cycles, this is exactly equivalent to the same application using all cycles of 30% of all cores in NewtOS. The conclusion is that while NEaT reduces the number of cores normally available to the applications, the load distribution has generally no negative impact on the end-to-end networked application performance.

3.3 TCP Connections

We focus our analysis on TCP as UDP and other protocols build on top of it are fairly simple to handle as they are stateless, however, most Internet traffic uses the more challenging stateful TCP protocol. TCP raises interesting challenges by maintaining per-connection state. When initiating a client connection, the socket library selects the network stack replica to handle the connection, while, when accepting a TCP connection from a remote client, the NIC decides based on hashes or filtering rules, as it is the first component to process each packet. Once the decision on the replica responsible to handle the connection is made, both the NIC and the libraries must honor the choice. This process is transparent to the application programmer and hidden by the provided libraries (§4).

To be able to accept a connection through different network stack replicas, the listening TCP sockets are the only types of sockets that are replicated across all the possible stacks. Binding listening sockets to a single replica would otherwise force NEaT to assign all the incoming connections to the same replica, resulting in load imbalance (and less unpredictability, i.e., security). Note that listening sockets are replicated across all the network stacks only at `listen()`-time, given that there is no general way of knowing whether a socket will be a listening socket at creation time. `listen()` call tells the

library to create a socket per each replica of the stack, they all listen at the same address and are not visible to the application. Each "*subsocket*" remains then fully isolated in a single replica, with the user library hiding the underlying replication from the application itself. Applications can simply use the original file descriptor obtained at socket creation time, allowing the library to perform the necessary socket-subsocket mapping operations behind the scenes if needed. The `accept` implementation, for instance, checks for any subsocket with an available incoming connection and simply "*accepts*" the connection from it. This also allows the loopback devices to be implemented by each of the replicas. After a connection is accepted with a new socket number, the new socket is independent of all other sockets with its own communication set up. This is fully transparent to the application and the library. Neither needs to be aware of which replica owns a particular socket. The library only translates between socket numbers and the internal communication channels. Therefore, there is no need for synchronization. Details of the communication, notifications and buffer mappings are described in [35].

Since connections are naturally spread across several different isolated replicas, NEaT can easily allow for accepting connections in parallel and in a conflict-free way. In a system with a single instance of the listening socket, in contrast, there is contention when multiple threads attempt to access the socket simultaneously. Recent work has sought to directly address this particular problem on Linux [31, 52]. Unlike these systems, NEaT eliminates the need for synchronizing access to the subsockets residing in different network stack replicas altogether, allowing different application threads to accept from a single subsocket while "stealing" connections from others for load balancing purposes with synchronization limited to the application.

3.4 Scaling Up and Down

By default, NEaT runs with a static number of replicas to enforce stable and predetermined reliability guarantees. In settings where a limited number of cores is available and reducing reliability guarantees is an option, we describe how NEaT can fulfill more dynamic scalability requirements by scaling the number of replicas in use up and down depending on the current load and the performance required by the applications. For example, when certain parts of the system are not needed, it is possible to place multiple components on a single core, for example idle NIC drivers.

The system boots with at least one replica, depending on the redundancy requirements. When NEaT becomes overloaded, it automatically spawns a new network stack replica. Since the NIC has rules for the existing connections, their distribution remains intact as long as each connection exists. The new connections are assigned across all the replicas. This may lead to initial imbalance in load distribution after reconfiguration. We expect the system to rebalance itself as soon as existing connections terminate and new connections appear.

When the load drops again, NEaT can also scale down and terminate some network stack replicas. Since the connections are in a general case likely distributed across all the replicas, simply shutting down a single replica would result in abruptly terminating all the TCP connections handled by it—similar to the effect caused by an unexpected failure. Migrating connections from a replica in termination state to another replica in nontermination state, however, would require handing off TCP-related state and in-flight data of currently non-idle connections. NEaT would also need to change the filtering rules in the NIC. This strategy is overly complicated and also violates NEaT's isolation principles, potentially introducing subtle reliability and scalability issues. For these reasons, NEaT adopts a different strategy, which (i) marks the necessary replicas as in termination state and makes the corresponding cores available to the applications; (ii) instructs the NIC to distribute new connections only to replicas in nontermination state but continue to serve packets on existing connections across all the replicas; (iii) garbage collects replicas in termination state as soon as their connection count drops to zero. This approach implements an effective lazy termination strategy without breaking any of the existing connections. The trade off is a slower scaling down phase, which, however, only results in short-lived resource overcommitment periods and can actually better handle load fluctuations—that is quickly scaling up again whenever necessary.

In general, creating and terminating network stack replicas incurs latency and management overhead. In addition, the number of replicas the applications can indirectly use is limited by the ratio of cores dedicated to the system compared to those dedicated to the applications. Many modern CPUs, however, support several hardware threads on each core, for example the SPARC T7 processor features 8 threads per core. This allows NEaT to use significantly more replicas than cores while preserving fast user-space communication, given that each replica maintains its own hardware context. Interestingly, as shown in §6, this strategy allows NEaT to use available cores more efficiently, given that a single process can hardly use up all the core's cycles due to the latency of main memory [50]. We conclude that scaling down the number of replicas is not always necessary and, when hyper-threading is available, a better scale-down strategy is to relocate the replicas.

3.5 Scaling NIC Drivers

The only performance-sensitive component that NEaT does not actively scale up is the NIC driver, since none of our tests demonstrated the driver to be a potential performance bottleneck and also other researchers [54] have reported 10G line rate processing on a single core. From a reliability point of view, previous research has demonstrated the ability to seamlessly recover NIC drivers without replication [33].

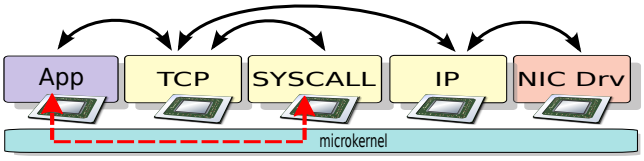


Figure 3: A replica of the multi-component network stack

3.6 Reliability

Isolation and partitioning of the network stack are key to the reliability of NEaT. The individual replicas are isolated and do not interact with each other, preventing a failure in one of the replicas from having any direct or indirect reliability impact on all the other replicas. Currently, NEaT opts for a completely stateless failure recovery strategy: when a replica crashes, a new replica is created and all the TCP-related state associated to the failed replica is lost. While supporting more complex stateful recovery policies is possible, this simple approach fully embraces our state partitioning strategy and ensures minimal state loss with all the other replicas continuing to serve existing and new connections with no interference or global service disruption. During the (short) recovery phase, the driver does not pass any packets to the recovering replica until it announces itself again. This strategy eliminates the need to reconfigure the device.

3.7 Multi-component Network Stack

Although each network stack replica can run as a single process, NEaT can also be configured (at compile time) to vertically split each replica into multiple isolated processes for increased reliability. Although this strategy requires more cores and internal communication, it also yields improved reliability since it can fully isolate faults in smaller network components. Excluding TCP, the other components are essentially stateless (or pseudostateless) and thus more easily amenable to application-transparent recovery even with a stateless recovery strategy. When adopting more heavyweight stateful recovery strategies such as the one described in [29], our state partitioning strategy effectively decreases the state surface to recover and reduces the likelihood of state corruption. Figure 3 presents a simplified version of a multi-component network stack replica, showing only the IP and TCP components used for TCP processing, but additional UDP and packet filter components are also present and isolated from the rest of the stack.

3.8 Security

Replication has been previously proposed for security purposes, using synchronized multivariant execution to detect arbitrary memory error attacks [20, 55, 56]. Unlike these approaches—which incur high run-time overhead—NEaT pursues the less ambitious goal of enforcing *address space re-randomization* [59] across user connections. NEaT naturally enforces this security defensive mechanism as part of its design at no extra cost. This is simply done by binding each connection to a random replica, while creating each replica independently and with ASLR [1] enabled. The latter strategy yields completely different memory layouts across (semantically equivalent) replicas, resulting in consecutive user connections being handled by processes with *unpredictably* different memory layouts. Albeit not our primary focus here, our design can support such address space re-randomization strategy in a natural and inexpensive way, effectively countering recent memory error attacks that rely on a stable memory layout across user connections [13, 57].

4. HARDWARE SUPPORT

NEaT requires certain hardware features for an efficient implementation. For example, the current implementation of our fast user-space communication channels relies on the `MWAIT` x86 instruction that allows NEaT to halt a core and enables a process running on another core to wake up the first core using a simple memory write. This eliminates kernel assistance for expensive interprocessor interrupts and process halting. Note that NEaT actually switches to such slower communication channels as needed automatically, in particular when the load is low and allowing the scheduler to colocate processes on shared cores—thus freeing up the dedicated ones—becomes an appealing option. To share the cores more efficiently, NEaT also exploits hardware multithreading, which results in increased parallelism, coalescing replicas on fewer cores, and the ability to leverage `MWAIT`-based communication channels. We evaluate the benefits of hardware multithreading in §6.

Efficiently scaling the network stack replicas requires dedicated NIC support to split the packet flow. Commonly available NICs feature many pairs of transmit and receive queues. NEaT uses one pair of queues to direct packets to each network stack replica. The controller can place the packets on different queues based on different criteria, using protocol fields to uniquely identify each flow. In contrast to the number of cores, the number of queues is not a limiting factor. In particular, modern network cards can already classify and steer packets to different endpoints (i.e., different processes and/or virtual machines) based on a hash of a 5-element tuple including destination, source addresses, ports, and protocol number, or also use precise filters over the same fields. For example, Intel 10G cards can hold up to 8 thousand filters. Software is, however, responsible for configuring the filters, which makes issuing frequent updates impractical with hundreds of thousands of connections per second. The NIC programming interface, in particular, requires many read-write transactions across the PCI bus to configure the filters, which can negatively impact network processing [25]. For instance, the Intel ixgbe driver on Linux can optionally use TCP packet

sampling, which sets up the NIC's filters to track locality of connections [15]—hence offloading the work done by receive flow steering (RFS [32]). Due to performance concerns, the driver only samples every 20^{th} packet.

Since the NIC already inspects packets in its local memory, we believe a much more practical solution—which, however, requires extensions not yet available in contemporary commodity hardware—is adding extra logic into the NIC and creating "tracking" filters based on the packets the NIC handles. Such filters would simply instruct the NIC to ensure all the corresponding packets of each flow follow the same route.

On contemporary hardware, it is theoretically possible to implement the missing connection tracking features in the NIC driver itself (mimicking a smart NIC). We, however, felt this was a step in the wrong direction, when compared to the much more realistic option of offloading such support to the hardware, similar to TCP segmentation (TSO), large receive (LRO), and other similar features which eventually made their way into modern hardware. For this reason—but also not to introduce artificial overhead moving forward—we opted for a different solution in our experiments, compensating for the missing hardware features by limiting our evaluation to server applications, while still relying on contemporary NIC's hash functions to randomly distribute the inbound connections.

Summarizing, similar to Arrakis [53] and IX [12], our scalability design strongly advocates for the devices taking over part of the system data plane while the operating system acts only as a control plane managing its settings, for instance the TCP `TIME_WAIT` timeout. We can also look at the NIC as an additional processing core that runs certain parts of the stack very efficiently, while lacking the huge flexibility of a generic core. Therefore, dedicating a core of a truly many-core system might be an option. We believe that, as modern NICs have already evolved to match the requirements of monolithic systems, reliability and scalability of design can effectively initiate similar developments or may lead to a renaissance of programmable network cards. If the programmable NIC were to offer the same interface as the network driver (as we have done previously [36]), there would be no need for the drivers and we could free their cores.

5. COMPARISON WITH OTHER USERSPACE NETWORK STACKS

As discussed in §2, the literature documents many existing efforts to build user-mode network stacks. The primary motivation to implement a network stack in user space is to bypass and eliminate the overhead imposed by the operating system, typically implemented as a monolithic kernel that runs its own network stack in privileged mode and exposes it to user applications. The most common examples of such an operating system architecture are Linux, BSD, and Windows.

Unlike traditional user-mode network stacks, NEaT's goal is not to bypass the operating system itself, but to implement a generic network stack that is presented by the operating system to the applications through a standard and widely used POSIX interface. NEaT is a user-mode stack by definition, as all parts of the NewtOS operating system itself are implemented as unprivileged code running on top of a microkernel. We believe that any microkernel such as L4 or a multikernel such as Barrelfish could offer similar IPC mechanisms the NewtOS kernel provides. We also believe that even a monolithic kernel such as Linux could provide such mechanisms, and even though the NEaT design does not naturally map to a monolithic system design, it can represent a scalable and reliable alternative to the native network stack of monolithic operating systems.

Userspace stacks such as mTCP [39], MegaPipe [31], and OpenOnload [5] target low latencies by avoiding system calls and interacting directly with the devices. These stacks are linked directly against a given application to bypass the operating system. Thereby, they are not generally available to other applications on the system to facilitate sharing and cooperation. This strategy minimizes various overheads that a generic network stack cannot afford ignoring and hence applications that use such a network stack outperform applications that use the good-for-all stack provided by the operating system. NEaT and NewtOS draw inspiration from these projects and their sockets (§3.2 and [35]) are implemented in a nearly system-call-less fashion. Similarly, the networks stacks of Arrakis [53] and IX [12] are tuned for low latency and low overhead, closer to a library OS design rather than a general-purpose stack like NEaT.

On the reliability side, userspace network stacks increase the overall reliability of the system, as a fault in the network stack does not affect applications that do not use it. However, there is no isolation between the stack and its application. Nevertheless, such network stacks are useful in certain domains and it is, in fact, plausible to allow particular applications in NewtOS to use such a design next to other applications that use NEaT within the same system.

6. EVALUATION

We evaluated NEaT using two different multicore machines: (i) a 12-core AMD Opteron 6168 (1.9 GHz) and (ii) a dual-socket quad-core Intel Xeon E5520 (2.26 GHz). The AMD machine has more physical cores, but the Xeon machine features 2 hardware threads per core (hyperthreading). For our experiments, we used a 10G Intel i82599 network card in each machine. This network card implements some features required in order to steer incoming network traffic to different replicas, as well as TSO (TCP Segment Offloading) which greatly improves performance and allows smaller configurations to reach a full 10Gb/s throughput utilization. The two machines are connected through a 10GbE DAC—Direct Attach

Option Tuned	kreq/s
defaults	184.118
sched+eth+irqAff+rxAff	186.667
sched+eth+irqAff+rxAff+serv	**223.987**

Table 1: Request rate breakdown per option tuned, with 12 concurrent httperf instances, each opening 1000 connections, with 1000 requests for a 20 byte file per connection. *sched* refers to the scheduler policy set to deadline, *eth* turns off auto-negotiation and uses TSO, *irqAff* sets IRQ affinities, *rxAff* sets receive queues affinity, *serv* denotes lighttpd being pinned to cores.

Copper—SFP+ cable. Depending on the experiment they alternate roles between system under test and load generator.

6.1 Linux

For comparison purposes, we selected Ubuntu Server 14.04 LTS as our Linux reference system. We configured our installation with default settings except for the fine tuning presented in Table 1 to yield the highest possible performance. Likewise, we used a default lighttpd configuration, except for the two following changes: (i) we allowed up to a thousand (1000) requests per TCP connection, and (ii) we ran each lighttpd server on a different TCP port. This section provides details on tuning Linux on the AMD machine.

Table 1 presents a breakdown of options we tuned in order to improve as much as possible the performance of our Linux baseline to ensure a fair comparison. We first evaluated the out-of-the-box performance (defaults) on our testbed, then we changed the scheduler policy to earliest deadline. Next, we used `ethtool` to turn auto-negotiation off and TSO on. As shown in the table, we obtained the first substantial gains when setting the IRQs affinities (irqAff). We observed, in turn, that the request rate drops when we manually set the receive queues affinity (reqAff), most likely explained by the lighttpd processes being scheduled on different cores than where the related receive queues are pinned. The best performing configuration is when we also pin the lighttpd processes to a specific core (serv). Using RFS did not result in observable benefits and as such was not considered in the final configuration.

In each case, we used 12 httperf processes —one per client machine's core— each targeted at one specific, different port. For each port we started a different lighttpd server.

Httperf dismisses from the request rate and throughput any connection which has an error. Given that errors might happen at any time during the connection, the actual bandwidth is higher than reported when connection errors occur. Figure 4 shows a clear correlation between the number of requests issued and the evolution of latency. As soon as we switch to moderately large files (between 100K - 1M), the latency dramatically increases,

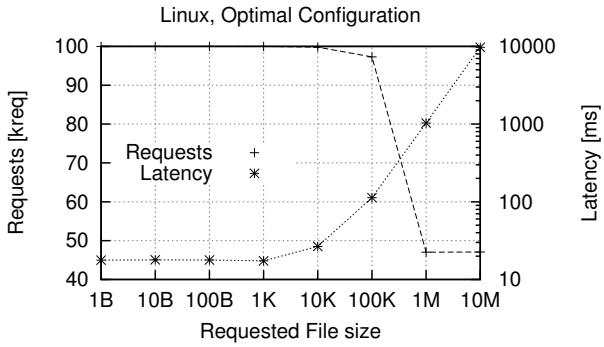


Figure 4: Latency and total number of requests vs. file size

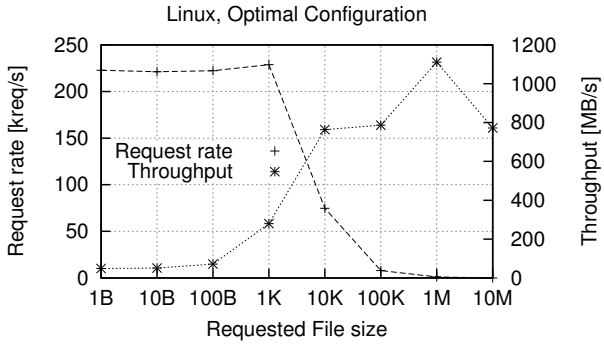
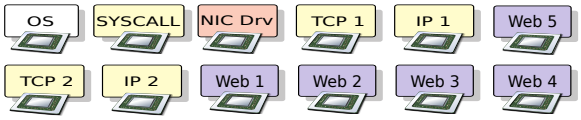


Figure 5: Throughput and request rate vs. file size

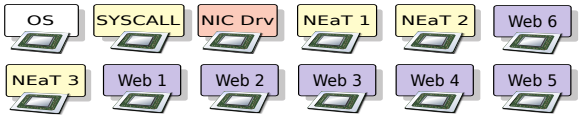
the number of requests drops, as the link bandwidth becomes the limiting factor. Figure 5 shows the relationship between the *successful* request rate and bandwidth. As soon as the link bandwidth reaches saturation, we see a high variance in the number of successful transfers. For file sizes between 100KB, 1MB and 10MB, the bandwidth fluctuates heavily as connection errors start appearing —timeouts and connection resets mainly— forcing httperf to discard results in its reports, lowering the scores. As figure 5 shows, as soon as the file size exceeds 7KB, the 10Gb/s bandwidth becomes the bottleneck. This is why, in all the following experiments, we used a very small size of 20 bytes for the requests as this is what stresses the most the network stack, the least the web server and other components of the OS, and also guarantees that the hardware doesn't introduces limitation other than CPU-bound issues.

6.2 Scalability

Evaluating the scalability of NEaT required us to select a representative server application, known to efficiently scale to a large number of CPUs and thus able to expose scalability deficiencies in the underlying network stack. As we wanted to evaluate the stack by itself, the actual application is not really important, as long as it "does as little as possible", lest we (partially) measure the application or other subsystems rather than the net-



(a) Multi-component stacks



(b) Single-component stacks

Figure 6: AMD - The best configurations - all 12 cores

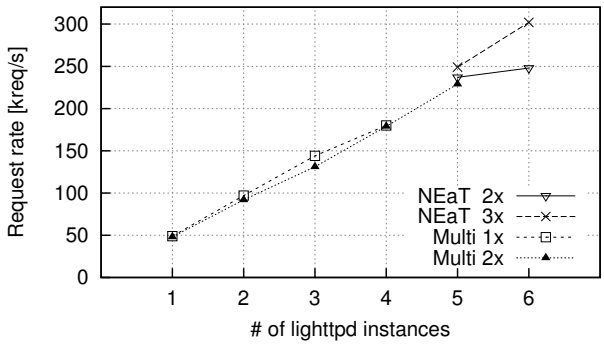


Figure 7: AMD - Scaling lighttpd and the network stack

work stack. For our purposes, we selected lighttpd, an efficient web server application, and made it serve only static files cached in memory to minimize interference with other operating system subsystems.

We evaluated NEaT both in single- and multi-component configurations. In the figures, we denote such configurations as NEaT Nx and Multi Nx respectively, where N refers to the number of replicas used. We also refer to a particular replica R as NEaT R and TCP R (or IP R).

To evaluate the scalability, we used the *httperf* benchmarking utility to repeatedly open persistent connections and request a small 20-byte file 100× per connection. This workload stresses the network stack without taking advantage of NIC offloading features, while allowing us to scale up the number of servers freely. A lighttpd instance serving just 8 or more simultaneous connections can drive the application cores close to 100% utilization. At the same time, repeatedly requesting small files stresses the network stack as it must handle many requests from the network as well as the applications.

6.3 Scalability on a 12-core AMD

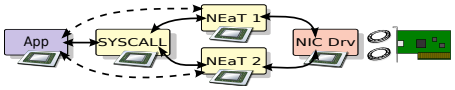
We first present results on the 12-core AMD. The number of available cores allows us to compare both single- and multi-component configurations. Figure 6 illustrates both configurations in their best-performing setups and Figure 7 presents scalability results for the different configurations of NEaT. In our experiments, we dedicated one of the cores to all the remaining operating system processes (OS) and one of the cores to the SYSCALL server, which generally needs its own core for low-latency messaging when the load is relatively low, but its role is crucial to ramp up the load for testing purposes. As the load grows, the core becomes increasingly idle, since the applications can bypass it with our mostly system-call-less socket design.This leaves our test machine with 10 cores available for the network stack and lighttpd. As Figure 7 shows, a configuration consisting of a single replica of the multi-component stack can scale fairly linearly up to 4 lighttpd instances. At that point, the stack becomes overloaded, but 3 cores remain completely unused. Adding one more replica can use up 2 (TCP, IP) of the 3 remaining cores, allowing the throughput to scale further, up to 5 lighttpd instances. Although the CPU usage suggests that NEaT could effectively scale further, no more cores are available to scale up our multi-component stack with more replicas or applications. While we do not have more cores at our disposal, we believe NEaT would in fact scale to larger numbers with no restrictions. The single-component version of NEaT, on the other hand, scales further. In particular, *NEaT 2x* performs comparably to its multi-component counterpart when serving up to 5 lighttpd instances. With an additional replica (*NEaT 3x*), NEaT perfectly scales up to 6 lighttpd instances. Similar to *Multi 2x*, NEaT is not overloaded yet.

We have conducted similar experiments for lighttpd running on Linux on the same hardware. When using all 12 cores, in the best performing configuration, lighttpd handled 224 kilo-requests per second (krps). With 3 single-component replicas, NEaT reached 302 krps (i.e., 34.8% more requests per second).

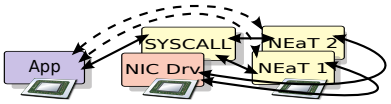
6.4 Scalability on a 8-core Xeon

Current trends suggest that the number of cores will keep growing (for example, Intel announced a new 72-core version of Knight's Landing [2] and SPARC T7 has 256 threads on a die) and, above all, as other researchers have suggested [41, 48, 60, 65], cores will become more heterogeneous and specialized. Nevertheless, our experience demonstrates that NEaT can replace cores by hardware threads—which are much cheaper—allowing NEaT's processes to use the available cores more efficiently. The general intuition is that hyper-threading allows NEaT to efficiently colocate relatively idle processes—which need their own context—like the SYSCALL server. Although a hardware thread is not the same as a fully-fledged core, the threaded setup can handle more load and provide redundancy. To confirm this intuition, we present our scalability results on a Xeon with hyper-threading.

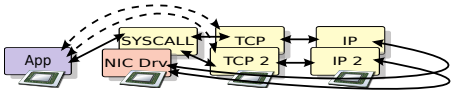
Figure 8 depicts an example of using hyper-threading to reduce the number of cores that 2 NEaT replicas normally require in core-only configurations (a). For example, NEaT can colocate the NIC driver with the SYSCALL server, since as the driver's thread becomes



(a) All components on dedicated cores



(b) Colocated components using hyper-threading



(c) Colocated multi-component configuration of NEaT

Figure 8: Reducing the used number of cores with threads

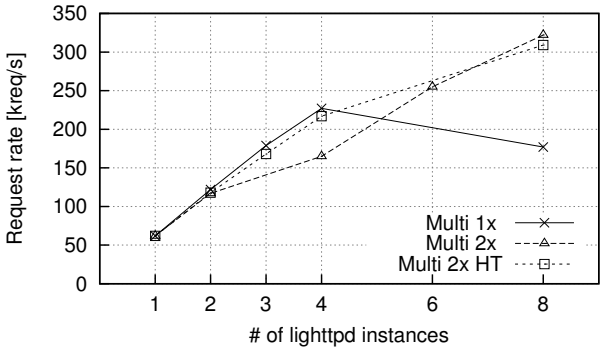


Figure 9: Xeon - Scaling the multi-component stack

more loaded, the SYSCALL's thread gets less loaded, ensuring little interference between the two processes. NEaT can also successfully place multiple NIC drivers on the same core. Similarly, the multi-component configuration of NEaT can use only 3 cores instead of the 6 cores used in the original configuration (Figure 8c). We use HT to denote the NEaT configurations that use hyper-threading.

In contrast to the AMD, deploying the TCP and IP processes leaves only 4 cores available for lighttpd. NEaT can now, however, take advantage of hyper-threads to run 8 instances. Figure 9 shows that similar to the AMD case, the throughput of the network stack peaks when using 4 lighttpd instances.

To scale further, we ran a second replica (*Multi 2x*), which leaves only 2 cores for lighttpd. Using all 4 threads of those cores reported similar performance to the 3-application-instance scenario in the previous test. This is expected, since the network stack is not the bottleneck and the 33% speedup (2 cores instead of 3) of the application is within the bounds of the benefits of hyper-threading. Further scaling up—denoted by points 6 and 8 in Figure 9—uses threads on the cores occupied by the network stack itself, first using both TCP (6 lighttpd instances) and then both IP cores (8 lighttpd

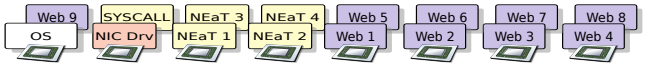


Figure 10: Xeon - The best-performing configuration of the NEaT, fully exploiting hyper-threading.

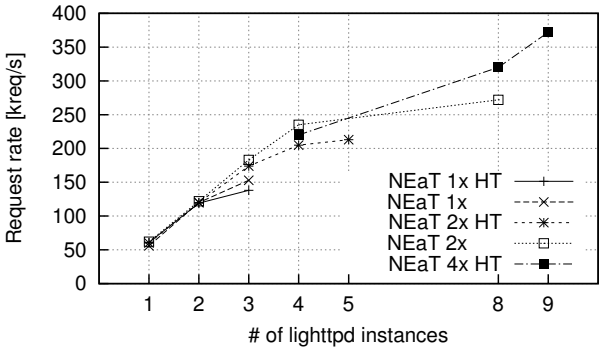


Figure 11: Xeon - Scaling the single-component stack

instances). In the latter configuration, Figure 9 shows the throughput peaking at 322 krps.

Finally, we colocated two replicas on different threads from the same cores (*Multi 2x HT*, Figure 8c)—enforcing this policy for both TCP and IP replicas. As expected, 4 lighttpd instances yielded similar performance to the 1-replica configuration, since the stack is not the bottleneck and—while using the same number of cores as in the 1-replica configuration—can handle 8 lighttpd instances (4 cores, both threads).

When evaluating single-component configurations, we used up to 4 replicas, as shown in Figure 10. The labels (*NEaT*x and *WebX*) illustrate the order in which we scaled up NEaT and lighttpd. We present the results in Figure 11. Note that once NEaT becomes overloaded, it can spawn another replica and keep processing the growing load. As the figure shows, *NEaT 4x* can sustain the load of 372 krps, using all cores up to 100%. This is a 13.4% improvement over 328 krps maximum we measured for the best Linux configuration on the Xeon, running 16 lighttpd instances on each of the 8 cores / 16 threads.

6.5 Impact of Different Configurations

Finally, we evaluated the impact of the different configurations of NEaT and compared their overhead in detail. We used a modified test issuing only a single request per connection, which significantly increases the load on the network stack itself. We evaluated 5 different configurations of the network stack using the 12-core AMD and different workloads. We deployed (i) 1 lighttpd instance processing 8 to 64 simultaneous connections, (ii) 2 instances processing 32 simultaneous connections, and (iii) 4 instances processing 64 simultaneous connections. Figure 12 reports our results, demonstrating that when the load is relatively low, using only a single replica of the multi-component stack to handle 8 connections is better than using 2 such replicas. This is primarily

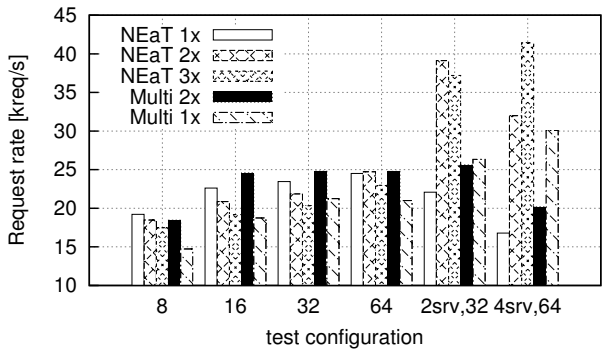


Figure 12: 12-core AMD - Comparing performance of different configurations stressed by the same workload.

CPU load	Active in kernel	Polling	Web krps
6%	33.3%	51.8%	3
60%	14.2%	27.9%	45
88%	5.4%	19.7%	90
97%	0.1%	7.4%	242

Table 2: 10G driver CPU usage breakdown on Xeon.

due to the fact that lightly loaded components often sleep, which introduces latency that is more evident in the multi-component stack. At higher loads, having multiple network stack replicas becomes more obviously beneficial.

Table 2 presents samples collected using statistical profiling of the NIC driver under a range of loads serving 3 replicas. A mostly idle driver spends a significant portion of the active time suspending/resuming in the kernel (as Intel's `MWAIT` is a privileged instruction), polling the 3 stacks and the NIC queues. The "wasted" time shrinks with increasing load. When the core's usage is close to 100%, the driver almost never enters the kernel and uses more than 90% of its time processing. Other components show similar behavior. Note that the CPU usage grows sharply, but levels off as the driver trades more of the "wasted" time for useful processing. While the CPU load is 60% when the lighttpd handles 45 krps, the load is only 88% when the number of requests doubles and still lower than 100% when the number of requests increases almost 5-fold.

6.6 Reliability

To thoroughly assess the reliability properties of our design, we first performed fault injection experiments on the different components of a NEaT replica. For our purposes, we developed a fault injection tool inspired by similar tools adopted in prior work to evaluate fault-tolerant designs [64, 47]. In our experiments, we injected faults into various (randomly selected) parts of the code in the network stack and monitored their impact on the execution. We ignored runs without any visible crash

Fully transparent recovery	53.8%
TCP connections lost	46.2%

Table 3: Fault injection experiment results.

within a minute from the injection (e.g., runs with no fault activation) and collected results from 100 failing runs. We restarted the system after every run to clean up. For our experiments, we used the same test workload as the one used for our scalability evaluation.

Table 3 shows the recovery success rate in our fault injection experiments. In more than 50% cases, we observed fully transparent failure recovery—applications nor the users notice—with no manual intervention needed and the effect on network traffic being no worse than a packet delay or loss. Since packet losses happen regularly on the Internet, we expect the impact of (infrequent) failure recovery to be negligible in practice—even under heavy load conditions. In all the other cases, the faults caused a crash in the TCP component, which resulted in the abrupt termination of all the open TCP connections (and loss of TCP state in general). After recovery, however, the TCP server was always reachable and, as expected, ready to establish new client connections.

Our fault injection results directly derive from the properties of our stack implementation, which, for drivers, IP, UDP, and packet filter component, maintain no or little read-mostly state which can thus be preserved in an independent data store, configuration file, or just recreated at restart time. In contrast, our TCP component maintains significant per-connection read/write state, read/write control state, and in-flight data. As a result, only TCP faults can cause visible state loss (and failures) in a single replica of our stack. In addition, thanks to the state partitioning and isolation strategy used by NEaT, only the TCP connections (and state) of the failing replica are lost while the connections managed by other replicas are completely unaffected. In other words, the larger the number of replicas, the lower the fraction of state lost after failures and thus the higher the reliability guarantees.

We discuss the reliability of a single replica experiment in more detail in our previous work [37]. Due to the complete isolation of the replicas, the fact that we run multiple replicas of the same multi-component network stack side by side has no impact on the fault recovery capabilities of each replica independently. We verified each replica can recover from a crash of its components without any interference with the other replicas. More interestingly, the overall resiliency of a network stack that consists of multiple replicas grows with the number of replicas.

To estimate the concrete reliability benefits of our replicated design, Figure 13 shows the expected fraction of state preserved after a failure against the maximum throughput across all the different configurations we

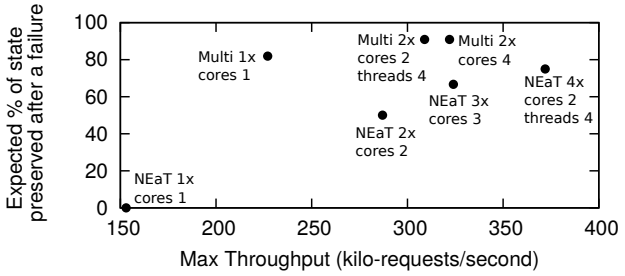


Figure 13: Expected fraction of state preserved after a failure vs. max throughput across network stack setups.

evaluated on the Xeon. We used the code size (proportional to code coverage for our test workload) of each component to estimate the probability that a single component fails when a failure occurs within the network stack—assuming uniform failure probability throughout the code—and the resulting expected fraction of state preserved after a failure. We also assumed the simple stateless TCP recovery strategy used in NEaT, which results in (only) the TCP state being always irrecoverable—after a TCP failure—as evidenced earlier. To further improve reliability, an option is to rely on checkpointing techniques [9] to support a (TCP) stateful recovery strategy allowing existing connections to survive failures. However, such techniques typically incur nontrivial run-time and recovery-time overhead (other than being generally unable to recover from a corrupted state), trading off performance for reliability. In contrast, as shown in the figure, NEaT's performance and reliability both increase with the number of replicas across all our configurations, confirming that our design allows reliability and scalability to effectively coexist with no compromises. In addition, as the figure demonstrates, given any fixed number of available cores (and HTs), NEaT's single- and multi-component configurations yield different performance-reliability tradeoffs, opening up interesting research opportunities on fine-grained isolation and allocation policies.

7. CONCLUSION

Reliability and scalability are commonly perceived as largely independent—and possibly conflicting— requirements concerning the implementation of operating system services. This paper challenged the common belief, demonstrating that two key design principles, isolation and partitioning, can effectively address reliability and scalability of a core OS service such as the network stack at the same time, and both with constant returns with respect to the number of available CPUs on modern multicore architectures. The principles we employ also demonstrated that not only is reliability and scalability of design possible, but represents a realistic and far superior alternative to reliability and scalability of implementation, plagued with problems and increasingly unable to "scale" to the complexity of modern hardware and software systems. Using such principled design, we implemented NEaT, a reliable and scalable network stack that isolates and partitions its state across multiple and truly independent replicas. Thanks to the design principles, NEaT can outperform Linux both in terms of reliability—withstanding failures and reducing service disruption as the number of replicas increases—and scalability—up to 13-35% higher throughput in our benchmark with 4 replicas—while retaining full compatibility with the BSD socket API and supporting full sharing and cooperation at the application level.

8. ACKNOWLEDGEMENT

We would like to thank the anonymous reviewers and our shepherd Olivier Bonaventure for their valuable feedback. This work was supported by the European Commission through project H2020 ICT-32-2014 "SHARCS" under Grant Agreement No. 64457 and the Netherlands Organisation for Scientific Research (NWO) through the grant 639.023.309 VICI "Dowsing".

9. REFERENCES

[1] ASLR: Leopard versus Vista. http://blog.laconicsecurity.com/2008/01/aslr-leopard-versus-vista.html.

[2] Intel's "knights landing" xeon phi coprocessor detailed. http://www.anandtech.com/show/8217/intels-knights-landing-coprocessor-detailed.

[3] Killing the Big Kernel Lock. http://lwn.net/Articles/380174/.

[4] MINIX 3. http://www.minix3.org.

[5] OpenOnload. http://www.openonload.org.

[6] RCU Linux Usage. http://www.rdrop.com/users/paulmck/RCU/linuxusage.html.

[7] RFC: remove __read_mostly. http://lwn.net/Articles/262557.

[8] The "too small to fail" memory-allocation rule. http://lwn.net/Articles/627419/.

[9] CRIU. http://criu.org/, 2015.

[10] J. Appavoo, D. D. Silva, O. Krieger, M. Auslander, M. Ostrowski, B. Rosenburg, A. Waterland, R. W. Wisniewski, J. Xenidis, M. Stumm, and L. Soares. Experience Distributing Objects in an SMMP OS. *ACM Trans. Comput. Syst.*, 2007.

[11] A. Baumann, P. Barham, P.-E. Dagand, T. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schüpbach, and A. Singhania. The Multikernel: A New OS Architecture for Scalable Multicore Systems. In *Proceedings of the ACM Symposium on Operating Systems Principles*, pages 29–44, 2009.

[12] A. Belay, G. Prekas, A. Klimovic, S. Grossman, C. Kozyrakis, and E. Bugnion. IX: A Protected Dataplane Operating System for High Throughput and Low Latency. In *Proceedings of the USENIX Symposium on Oper. Sys. Design and Impl.*, Oct. 2014.

[13] A. Bittau, A. Belay, A. Mashtizadeh, D. Mazieres, and D. Boneh. Hacking Blind. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2014.

[14] S. Boyd-Wickizer, H. Chen, R. Chen, Y. Mao, F. Kaashoek, R. Morris, A. Pesterev, L. Stein, M. Wu, Y. Dai, Y. Zhang, and Z. Zhang. Corey: An Operating System for Many Cores. In *Proceedings of the USENIX*

Symposium on Oper. Sys. Design and Impl., pages 43–57, 2008.

[15] S. Boyd-Wickizer, A. T. Clements, Y. Mao, A. Pesterev, M. F. Kaashoek, R. Morris, and N. Zeldovich. An Analysis of Linux Scalability to Many Cores. In *Proceedings of the USENIX Symposium on Oper. Sys. Design and Impl.*, pages 1–8, 2010.

[16] S. Boyd-Wickizer, M. F. Kaashoek, R. Morris, and N. Zeldovich. Non-scalable locks are dangerous. In *Proceedings of the Linux Symposium*, July 2012.

[17] S. Boyd-Wickizer and N. Zeldovich. Tolerating Malicious Device Drivers in Linux. In *Proceedings of the USENIX Annual Technical Conference*, page 9, 2010.

[18] M. Castro, M. Costa, J.-P. Martin, M. Peinado, P. Akritidis, A. Donnelly, P. Barham, and R. Black. Fast Byte-granularity Software Fault Isolation. In *Proceedings of the ACM Symposium on Operating Systems Principles*, pages 45–58, 2009.

[19] A. T. Clements, M. F. Kaashoek, N. Zeldovich, R. T. Morris, and E. Kohler. The scalable commutativity rule: Designing scalable software for multicore processors. In *Proceedings of the ACM Symposium on Operating Systems Principles*, pages 1–17, 2013.

[20] B. Cox, D. Evans, A. Filipi, J. Rowanhill, W. Hu, J. Davidson, J. Knight, A. Nguyen-Tuong, and J. Hiser. N-variant systems: A secretless framework for security through diversity. In *Proceedings of the USENIX Security Symposium*, pages 105–120, 2006.

[21] F. M. David, E. M. Chan, J. C. Carlyle, and R. H. Campbell. CuriOS: Improving Reliability Through Operating System Structure. In *Proceedings of the USENIX Symposium on Oper. Sys. Design and Impl.*, pages 59–72, 2008.

[22] T. David, R. Guerraoui, and V. Trigonakis. Everything you always wanted to know about synchronization but were afraid to ask. In *Proceedings of the ACM Symposium on Operating Systems Principles*, pages 33–48, 2013.

[23] A. Depoutovitch and M. Stumm. Otherworld: Giving Applications a Chance to Survive OS Kernel Crashes. In *Proceedings of the European Conference on Computer Systems*, pages 181–194, 2010.

[24] M. Desnoyers, P. E. McKenney, and M. R. Dagenais. Multi-core Systems Modeling for Formal Verification of Parallel Algorithms. *SIGOPS Oper. Syst. Rev.*, 47(2):51–65, 2013.

[25] M. Flajslik and M. Rosenblum. Network Interface Design for Low Latency Request-Response Protocols. In *Proceedings of the USENIX Annual Technical Conference*, pages 333–346, 2013.

[26] B. Gamsa, O. Krieger, J. Appavoo, and M. Stumm. Tornado: Maximizing Locality and Concurrency in a Shared Memory Multiprocessor Operating System. In *Proceedings of the USENIX Symposium on Oper. Sys. Design and Impl.*, pages 87–100, 1999.

[27] V. Ganapathy, M. J. Renzelmann, A. Balakrishnan, M. M. Swift, and S. Jha. The Design and Implementation of Microdrivers. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 168–178, 2008.

[28] A. Gefflaut, T. Jaeger, Y. Park, J. Liedtke, K. J. Elphinstone, V. Uhlig, J. E. Tidswell, L. Deller, and L. Reuther. The SawMill Multiserver Approach. In *Proceedings of the ACM SIGOPS European Workshop*, pages 109–114, 2000.

[29] C. Giuffrida, L. Cavallaro, and A. S. Tanenbaum. We Crashed, Now What? In *Proceedings of the Workshop on Hot Topics in System Dependability*, pages 1–8, 2010.

[30] R. Guerraoui and A. Schiper. Software-based Replication for fault Tolerance. *Computer*, 30(4):68–74, 1997.

[31] S. Han, S. Marshall, B.-G. Chun, and S. Ratnasamy. MegaPipe: A New Programming Interface for Scalable Network I/O. In *Proceedings of the USENIX Symposium on Oper. Sys. Design and Impl.*, 2012.

[32] T. Herbert. RFS: Receive Flow Steering. http://lwn.net/Articles/381955/, 2010.

[33] J. N. Herder, H. Bos, B. Gras, P. Homburg, and A. S. Tanenbaum. Failure Resilience for Device Drivers. In *Proceedings of the Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, 2007.

[34] D. Hildebrand. An Architectural Overview of QNX. In *Proceedings of the Workshop on Micro-kernels and Other Kernel Architectures*, pages 113–126, 1992.

[35] T. Hruby, T. Crivat, H. Bos, and A. S. Tanenbaum. On sockets and system calls: Minimizing context switches for the socket api. In *Proceedings of the Conference on Timely Results in Operating Systems*, 2014.

[36] T. Hruby, K. van Reeuwijk, and H. Bos. Ruler: easy packet matching and rewriting on network processors. In *Symposium on Architectures for Networking and Communications Systems (ANCS'07)*, Orlando, FL, USA, December 2007.

[37] T. Hruby, D. Vogt, H. Bos, and A. S. Tanenbaum. Keep Net Working - on a Dependable and Fast Networking Stack. In *Proceedings of the Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 1–12, 2012.

[38] G. C. Hunt, J. R. Larus, D. Tarditi, and T. Wobber. Broad New OS Research: Challenges and Opportunities. In *Proceedings of the Workshop on Hot Topics in Operating Systems*, 2005.

[39] E. Jeong, S. Wood, M. Jamshed, H. Jeong, S. Ihm, D. Han, and K. Park. mTCP: a Highly Scalable User-level TCP Stack for Multicore Systems. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation*, 2014.

[40] A. Kadav, M. J. Renzelmann, and M. M. Swift. Fine-grained Fault Tolerance Using Device Checkpoints. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 473–484, 2013.

[41] Khubaib, M. A. Suleman, M. Hashemi, C. Wilkerson, and Y. N. Patt. MorphCore: An Energy-Efficient Microarchitecture for High Performance ILP and High Throughput TLP. In *Proceedings of the Annual IEEE/ACM International Symposium on Microarchitecture*, pages 305–316, 2012.

[42] O. Koren. A Study of the Linux Kernel Evolution. *SIGOPS Oper. Syst. Rev.*, 40(2):110–112, Apr. 2006.

[43] A. Lenharth, V. S. Adve, and S. T. King. Recovery Domains: An Organizing Principle for Recoverable Operating Systems. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 49–60, 2009.

[44] T. Liu and E. D. Berger. SHERIFF: Precise Detection and Automatic Mitigation of False Sharing. In *Proceedings of the ACM International Conference on*

Object Oriented Programming Systems Languages and Applications, pages 3–18, 2011.

[45] P. E. Mckenney and J. D. Slingwine. Read-Copy Update: Using Execution History to Solve Concurrency Problems. In *Parallel and Distributed Computing and Systems*, pages 509–518, Oct. 1998.

[46] J. M. Mellor-Crummey and M. L. Scott. Algorithms for Scalable Synchronization on Shared-memory Multiprocessors. *ACM Trans. Comput. Syst.*, 9(1):21–65, Feb. 1991.

[47] W. T. Ng and P. M. Chen. The systematic improvement of fault tolerance in the rio file cache. In *Proceedings of the 29th Annual International Symposium on Fault-Tolerant Computing*, pages 76–83, 1999.

[48] E. B. Nightingale, O. Hodson, R. McIlroy, C. Hawblitzel, and G. Hunt. Helios: Heterogeneous Multiprocessing with Satellite Kernels. In *Proceedings of the ACM Symposium on Operating Systems Principles*, pages 221–234, 2009.

[49] R. Nikolaev and G. Back. Virtuos: An operating system with kernel virtualization. In *Proceedings of the ACM Symposium on Operating Systems Principles*, pages 116–132, 2013.

[50] K. Olukotun, L. Hammond, and J. Laudon. *Chip Multiprocessor Architecture: Techniques to Improve Throughput and Latency*. 2007.

[51] N. Palix, G. Thomas, S. Saha, C. Calvès, J. Lawall, and G. Muller. Faults in linux: Ten years later. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 305–318, 2011.

[52] A. Pesterev, J. Strauss, N. Zeldovich, and R. T. Morris. Improving Network Connection Locality on Multicore Systems. In *Proceedings of European Conference on Computer Systems*, 2012.

[53] S. Peter, J. Li, I. Zhang, D. R. K. Ports, D. Woos, A. Krishnamurthy, T. Anderson, and T. Roscoe. Arrakis: The Operating System is the Control Plane. In *Proceedings of the USENIX Symposium on Oper. Sys. Design and Impl.*, Oct. 2014.

[54] L. Rizzo and G. Lettieri. iVALE, a Switched Ethernet for Virtual Machines. In *Proceedings of the International Conference on Emerging Networking Experiments and Technologies*, pages 61–72, 2012.

[55] B. Salamat, A. Gal, T. Jackson, K. Manivannan, G. Wagner, and M. Franz. Multi-variant Program Execution: Using Multi-core Systems to Defuse Buffer-Overflow Vulnerabilities. In *Proceedings of the International Conference on Complex, Intelligent and Software Intensive Systems*, pages 843–848, 2008.

[56] B. Salamat, T. Jackson, A. Gal, and M. Franz. Orchestra: Intrusion detection using parallel execution and monitoring of program variants in user-space. In *Proceedings of the European Conference on Computer Systems*, pages 33–46, 2009.

[57] J. Seibert, H. Okkhravi, and E. Söderström. Information leaks without memory disclosures: Remote side channel attacks on diversified code. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*, pages 54–65, 2014.

[58] L. Shalev, J. Satran, E. Borovik, and M. Ben-Yehuda. IsoStack: Highly Efficient Network Processing on Dedicated Cores. In *Proceedings of the USENIX Annual Technical Conference*, 2010.

[59] K. Z. Snow, F. Monrose, L. Davi, A. Dmitrienko, C. Liebchen, and A.-R. Sadeghi. Just-In-Time Code Reuse: On the Effectiveness of Fine-Grained Address Space Layout Randomization. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 574–588, 2013.

[60] R. Strong, J. Mudigonda, J. C. Mogul, N. Binkert, and D. Tullsen. Fast Switching of Threads Between Cores. *SIGOPS Oper. Syst. Rev.*, 43(2):35–45, Apr. 2009.

[61] Y. Sun and T.-c. Chiueh. SIDE: Isolated and Efficient Execution of Unmodified Device drivers. In *Proceedings of the Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 1–12, 2013.

[62] S. Sundararaman, S. Subramanian, A. Rajimwale, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, and M. M. Swift. Membrane: Operating System Support for Restartable File Systems. In *Proceedings of the USENIX Conference on File and Storage Technologies*, page 21, 2010.

[63] M. M. Swift, M. Annamalai, B. N. Bershad, and H. M. Levy. Recovering Device Drivers. *ACM Trans. Comput. Syst.*, 24(4):333–360, Nov. 2006.

[64] M. M. Swift, B. N. Bershad, and H. M. Levy. Improving the reliability of commodity operating systems. *ACM Transactions on Computer Systems*, 23(1):77–110, 2005.

[65] K. Van Craeynest, A. Jaleel, L. Eeckhout, P. Narvaez, and J. Emer. Scheduling Heterogeneous Multi-cores Through Performance Impact Estimation (PIE). In *Proceedings of the Annual IEEE/ACM International Symposium on Microarchitecture*, pages 213–224, 2012.

[66] D. Wentzlaff, C. Gruenwald, III, N. Beckmann, K. Modzelewski, A. Belay, L. Youseff, J. Miller, and A. Agarwal. An Operating System for Multicore and Clouds: Mechanisms and Implementation. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 3–14, 2010.

[67] P. Willmann, S. Rixner, and A. L. Cox. An Evaluation of Network Stack Parallelization Strategies in Modern Operating Systems. In *Proceedings of the USENIX Annual Technical Conference*, 2006.

[68] T. Yoshimura, H. Yamada, and K. Kono. Is Linux Kernel Oops Useful or Not? In *Proceedings of the Workshop on Hot Topics in System Dependability*, page 2, 2012.

[69] F. Zhou, J. Condit, Z. Anderson, I. Bagrak, R. Ennals, M. Harren, G. Necula, and E. Brewer. SafeDrive: Safe and Recoverable Extensions Using Language-based Techniques. In *Proceedings of the USENIX Symposium on Oper. Sys. Design and Impl.*, pages 45–60, 2006.