Abstract

Operating systems have rapidly evolved from simple resource multiplexer into massive software products that provide a wide range of services, many of which are crucial for the increasingly high scalability and reliability demands of modern applications. Unfortunately, commodity OS architectures lack adequate design abstractions to address these concerns and instead resort to scalability and reliability of implementation. This approach typically results in complex error recovery paths and hard-to-maintain synchronization code to deal with the scalability challenges imposed by modern multicore architectures and the underlying cache coherence protocols.

In this paper, we demonstrate that a simple and a principled OS design driven by two key abstractions—*isolation* and *partitioning*—is effective in building scalable and reliable systems while solely relying on scalability and reliability of design. To substantiate our claims, we apply our ideas to the network stack—a key operating system service—present a “clean slate” implementation called NEAT, and discuss the architectural support necessary to deploy our solution in real-world settings. We show that our principled design can (i) intelligently partition the network stack state to minimize the impact of failures and (ii) scale comparably to Linux, but without exposing the implementation to common pitfalls such as synchronization errors, poor data locality, and false sharing.
6.1 Introduction

Building scalable and reliable OS services is a daunting task. Even when opportunities for scalability [42] and reliability [92] exist at the interface level, producing a scalable and reliable implementation is notoriously challenging [31; 37; 65]. This problem is exacerbated by the dominant “build-and-fix” model adopted in the development of commodity operating systems, which generally attempts to retrofit scalability and reliability in legacy implementations. This strategy—scalability and reliability of implementation—misses important opportunities to address key scalability and reliability problems at design time and has troubles scaling with the fast-paced evolution and complexity of modern hardware and software [31].

This paper presents a new coordinated approach to scalability and reliability of design of OS services based on two key abstractions: isolation and partitioning. We show that both abstractions (i) allow scalability and reliability to coexist and symbiotically improve with the number of available cores and (ii) greatly simplify and improve the longevity of the final implementation. To support our claims, we present NEAT, a scalable and reliable componentized network stack implemented using a “clean slate” strategy on top of a microkernel-based architecture. NEAT embraces the proposed abstractions to isolate individual threads of execution in separate components and partition the system state across multiple and independent replicas. Other than the resulting scalability and reliability benefits, NEAT offers support for the standard BSD socket API while preserving application-level sharing and cooperation capabilities. Thanks to our design, NEAT (i) scales comparably to the Linux’ network stack, (ii) survives run-time failures with minimal impact, and (iii) minimizes code explicitly dealing with scalability and reliability concerns—and thus harder to maintain in face of constant hardware and software changes.

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

The Linux kernel is a case in point. With its code base rapidly growing from the original few thousand lines of code to the millions of lines of code and dozens of architectures today [85], maintaining high scalability and reliability standards has become increasingly prohibitive despite the 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 since then updated several times, after repeatedly exhausting their scalability opportunities on contemporary hardware [39]. Further, as the size and complexity of the kernel increase, so does the rate of faults introduced into the code, with an average fault lifespan of around 1 year even for high-impact faults that can bring
the system to a halt [117]. To mitigate such reliability impact, extensive community review and testing [89]—which has troubles scaling even for mission-critical code, as the recent Heartbleed outbreak demonstrated [6]—and complex error handling code—which is hard to test and may lead to new bugs [127]—are necessary.

While NEAT is an incomplete prototype with an explicit focus on networking, and it may thus not be fair to compare it to a full-featured operating system such as Linux, its design based on isolation and partitioning does structurally solve many of the recurring implementation issues described above (§6.2). Finally, while researchers have considered abstractions such as isolation and partitioning before—in different forms—for either scalability [31; 37; 150; 59; 26] or reliability [65; 51; 77; 147; 92; 75], our design rigorously combines these abstractions together to address both the scalability and reliability of operating system services and demonstrates their effectiveness in building a BSD-compliant network stack.

**Contributions** To summarize, our contributions are:

- We present a new coordinated approach to the scalability and reliability of operating system services based on two key design abstractions—*isolation* and *partitioning*—and analyze the opportunities offered to solve recurring and challenging implementation problems in existing systems.

- We apply our design abstractions to the network stack and present the principled design and implementation of NEAT, a BSD-compliant network stack that rigorously isolates and partitions its individual components to support scalability and reliability of design. We show that our proof-of-concept prototype—that runs $N$ network stack replicas on top of a microkernel-based architecture—marginally relies on the implementation to obtain the desired scalability and reliability properties. Further, the full decoupling between replicas allows NEAT to assign each user request to a random replica, ensuring load balancing and, as a by-product, replica assignment unpredictability, resulting in improved security against emerging memory error attacks [34; 137].

- We discuss the architectural support necessary to deploy our solution in real-world settings, suggesting logical and straightforward extensions to commodity hardware.

- We evaluate our NEAT prototype using the popular lighttpd web server. Our results demonstrate that NEAT can handle up to 13% more requests than Linux.

**Outline** The remainder of the paper is laid out as follows. §6.2 provides background information, highlighting problems in existing OS designs and comparing our principled approach to scalability and reliability with prior work. §6.3 presents the design and implementation of NEAT. §6.5 presents experimental results, assessing the scalability and reliability properties of our NEAT prototype. Finally, §6.6 concludes the paper.
6.2 Background and Related Work

Shaping the design of scalable and reliable systems using isolation and partitioning makes intuitive sense: (i) they enforce data locality and conflict-free parallelism for scalability purposes; (ii) they enforce fault containment and conflict-free failure recovery for reliability 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 isolation and no partitioning possible and, as a result, no scalability and reliability of design. In the next subsections, we develop this intuition further and highlight the fundamental limitations of this model and scalability and reliability of implementation in general.

6.2.1 Scalability of Implementation

Synchronization To scale to the increasing number of available cores, monolithic kernels have naturally grown to become massively threaded in their modern implementations. 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 the scalability of the system to the implementation of complex synchronization mechanisms that grant safe access to shared data structures. Implementing provably correct synchronization primitives is alone challenging due to the complexity of modern hardware and compilers [53]. Implementing such primitives in a scalable way is even more challenging and also heavily dependent on the particular hardware [48].

Even widely deployed fine-grained locking primitives such as Linux’ ticket spinlocks have been recently found plagued with scalability problems, with researchers demonstrating that more scalable implementations such as MCS locks [106] are necessary to avoid dramatic performance drops on many-core architectures [39]. Other generally more scalable alternatives to locking include lockless data structures or lightweight synchronization mechanism such as read-copy-update (RCU) [104]. While increasingly popular in the Linux kernel, RCU indirectly demonstrates the difficulties of implementing scalable and general-purpose synchronization primitives: its characteristics can only satisfy less than 8% of the entire kernel (9,000 uses) [14] and only provide scalable read-side semantics. Write-side latency grows with the number of processors.

Sharing Monolithic kernels structured around a single address space allow all their threads to implicitly share arbitrary data structures. While commonly perceived as a convenient programming abstraction, the lack of explicit state partitioning comes at the cost of subordinating the scalability of the system to the ability of the implementation to limit sharing and preserve optimal data locality. In modern cache-coherent
multicore architectures, this is particularly crucial to prevent shared data from frequently (and unnecessarily) traveling between caches of individual cores and thus hindering scalability. This is, for example, a well-known scalability bottleneck in many common synchronization primitive implementations [39].

To address this problem, monolithic kernels strive to maintain data structures local to the core where they are most frequently used. Unfortunately, implementation-driven strategies are still insufficient to completely eliminate this problem, with data structures still following processes that migrate between cores—due to load balancing—and threads still sharing 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 [98]. gcc’s controversial \_\_read\_mostly attribute exemplifies the difficulties of addressing this problem entirely at the implementation level and in a scalable way: its adoption in the Linux kernel required 1554 annotations (v3.15.8) to isolate and group all the “read-mostly” variables together—structurally preventing conflicts with more frequently modified cachelines—but only to raise legitimate concerns that many remaining “write-often” variables may then maximize cache line sharing—ultimately degrading write-side scalability [15].

6.2.2 Reliability of Implementation

Fault containment The lack of isolation in monolithic kernels complicates the implementation of effective fault containment mechanisms: a single fault can arbitrarily propagate throughout 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 [151]). Unfortunately, the former approach may also result in the introduction of a large amount of complex but trusted recovery code—thus creating a vicious circle [65]—while the latter approach provides weak reliability guarantees with the inability to detect (nor recover from) global error propagation—recently estimated to occur in more than 25% of the cases [151]. 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 [147; 60; 36], others on language- [154] or compiler-based strategies [82; 40], yet others on virtualization techniques [144]. While such approaches are generally effective in containing faults in untrusted components, they are typically limited to relatively small kernel subsystems (i.e., device drivers)—recovery techniques for larger subsystems do exist, but at the cost of more complex recovery code [145]—and may fail to guarantee full isolation when the driver interacts with the rest of the kernel using nonstandard interfaces [82].
Failure recovery  Fault containment is alone insufficient to implement effective failure recovery strategies. When a fault is detected, recovery actions are generally necessary to ensure that the system is in a globally consistent state. The lack of explicit state partitioning in monolithic kernels, 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 [151]. To mitigate this problem, researchers have proposed manual state reconstruction [51]—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 [92]—which, however, greatly limit the performance and scalability of the system [65]. Techniques that retrofit partitioning into monolithic kernels using virtualized domains have also been recently attempted [112], but at the cost of greatly limiting application-level sharing and cooperation.

6.2.3 Scalability and Reliability of Design

Scalability  While scalability of implementation is still a realistic—but already cumbersome—option today [38], many researchers have recognized the need for abstractions supporting scalability of design before. Tornado [59] and K42 [26] first argued for “partitioning, distributing, and replicating” data across independent objects to improve locality on shared-memory multiprocessors, presenting microbenchmarks to support their claims. In a similar direction, Corey [37] 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 base case [31], greatly limiting the opportunities offered by true isolation and partitioning. Nevertheless, similar to NEAT, Corey [37] has successfully demonstrated partitioning the network stack state across multiple per-core replicas. Unlike NEAT, however, their library OS-based design advocates for strict partitioning at the user level as well, greatly limiting application-level sharing and cooperation capabilities. Barrelfish [31], in turn, embraces a more radical design, with a “shared nothing” model based on isolation and replication. Unlike NEAT, however, such principles are mainly targeted to low-level kernel components—rather than enforced for OS services—and explicit cross-replica sharing (i.e., message passing) is still the norm to ensure global synchronization and replica consistency. Unlike all these systems, NEAT demonstrates that an even more radical design based on strict isolation and partitioning—no implicit or explicit sharing—is realistic and effective, with important scalability, but also reliability benefits. On the other hand, while we speculate that our design can be effective for several OS services, we limit our attention to networking and do not necessarily draw any absolute conclusions. More recently, fos [150] has also assessed the potential utility of replicating OS components across cores, but without considering its reliability benefits nor evaluating scalability in a real implementation. 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 [42]. 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.

Scalable network stacks like MegaPipe [68] and mTCP [81] focus solely on scalability of performance without any emphasis on reliability. They take a less radical approach to scalability with limited isolation and partitioning as their main goal is to circumvent the limitation of the monolithic systems implementation.

**Reliability** A vast body of research has been devoted to abstractions supporting reliability of design. Isolation, in particular, is a well-established design principle in reliable OS architectures. Microkernel-based operating systems such as QNX [130], MINIX 3 [12], Sawmill [61], and Singularity [77], in particular, are structured around a number of hardware- or software-isolated processes communicating via message passing to support fault containment by design. NEAT adopts a similar organization for its internal structure, but, unlike these systems, generally provides much stronger isolation guarantees—no multithreading—and relies on isolation to also support scalability of 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 [47] that most of these systems fail to recover from any failure in their stateful components. Techniques devised to address this problem rely on state replication [47] or checkpointing [65]. Thanks to its state partitioning strategy, in contrast, NEAT can gracefully recover from failures by simply 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 [67]. Finally, NewtOS [75] relies on a microkernel-based design to implement a fast and reliable network stack. Similar to NEAT, its design takes advantage of modern multicore hardware to improve performance. The lack of state partitioning, however, significantly limits its scalability and reliability guarantees, shortcomings that NEAT fully addresses as part of its design.

### 6.3 A Scalable and Reliable Network Stack

We present NEAT, a network stack designed from ground up for scalability and reliability. As a proof of concept for our design, we opted for a network stack because networking is a crucial part of every operating system in today’s interconnected world. In addition, the network stack is one of the most reliability and scalability sensitive subsystems and thus an ideal candidate to demonstrate the effectiveness of our design.

To our knowledge, NEAT is the first system of its kind that relies on a principled
design for both scalability and reliability at the same time. To enforce isolation of system components and applications, 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 somewhat beneficial for reliability, this design is still highly unsatisfactory for scalability.

The key scalability problem of every system executing as a set of isolated processes is that any of its components may easily get overloaded, even if it has been assigned an entire CPU core for itself. To address this problem, an option would be to run multithreaded components on multiple cores, but this strategy would impose the same scalability limitations evidenced in monolithic kernels. 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, which the scheduler can scale up and down.

NEAT’s design eliminates implicit synchronization and sharing both within the individual processes (no threading) and across generic OS processes (communication only possible 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 process replicas (no communication possible) allowing the network to scale and minimize the impact of failures—a failing replica does not prevent other replicas from continuing running undisturbed, causing minimal service disruption and state loss.

### 6.3.1 Overview

Figure 6.1 shows the basics of our architecture which uses several (in this case 4) replicas of the network stack. Each replica communicates with each NIC driver.
and, by default, each application communicates with all the replicas of the network stack as if there was only one, although it is possible to configure NEAT differently. This is transparent to the application programmer, since blocking system calls are routed through a system call server (SYSCALL). The implementation of our sockets (§6.3.2), however, allows the application to largely bypass the SYSCALL server and communicate directly with the right replica of the network stack. This complexity is hidden by the user space POSIX library. The individual system processes are assigned dedicated cores, allowing fast communication across user space OS components without intervention of the microkernel, an idea also used in prior work [75]. Only the applications use real microkernel-based message-passing for blocking system calls to the SYSCALL server, while the user space library handles the majority of the socket API within the context of the application itself (dashed lines in Figure 6.1).

With no data sharing nor 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 fairly large amount of state and NEAT must ensure that every packet of each connection uses the same path (Figure 6.2). The applications, the SYSCALL server, and the network devices are responsible for selecting the network stack replica to handle each socket. fos proposes using fleet coordinator for similar replica selection problems [150]. NEAT, in contrast, allows no process with 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 [120]. Contemporary network devices already have the ability to match incoming packets by a set of rules, split the traffic and steer the packets to the right replica of the stack using multiple internal NIC queues. The NIC driver can thus dispatch the packets to the network stack replica based on the receive queue of the NIC. Although these modern features are primarily motivated by the design of monolithic systems and virtualization, hence they were not originally conceived to comply with our requirements, NEAT successfully relies on them to implement replica-aware connection management. Without loss of generality, in the following sections, we assume that the NIC can effectively track connections—similarly to NAT support in routers—and we discuss architectural support required to deploy NEAT in realistic settings along with current limitations in §6.4.
6.3.2 Sockets

While adhering to the POSIX API to retain backward compatibility with legacy software, NEAT refrains from using a traditional implementation of network sockets for scalability reasons. In a microkernel-based system, legacy communication provided by the kernel incurs high costs, switches between kernel and user space execution, and involves multiple processes on different cores. In addition, the SYSCALL server can become a scalability bottleneck. For these reasons, NEAT opts for fast user space channels, which Barrelfish [31] uses between kernels and NewtOS [75] uses between system processes. Unlike these systems, NEAT uses such channels more pervasively, including for fast application-to-OS communication. While applications occasionally communicate with the SYSCALL server, NEAT resolves the vast majority of the system calls within the application itself, exposing the socket buffers to the application level similar to message queues. For optimal performance, NEAT also locates the network stack and the other system processes on different cores or hardware threads (hyper-threads) than those in use by the applications.

There are many extensions to monolithic systems which avoid system calls and setup message queues between applications and the OS kernel. FlexSC [138] seeks to avoid mode switching costs in the general case, while IsoStack [134], Megapipe [68], and netmap [122] specifically target networking to avoid contention on shared data structures within the kernel or the devices. NEAT draws inspiration from such techniques to allow applications to peek into socket buffers and determine whether an application-issued operation would block or not. In the latter case, the operation can be completed directly at the application level (in our library) without issuing a system call. In other words, applications go through the SYSCALL server only when no event is pending and blocking is needed. This ultimately translates to a lower number of interactions with the SYSCALL server as the application becomes more loaded. Since applications like nginx, memcached, lighttpd, and many others opt for an event-driven model which extensively relies on nonblocking operations and socket monitoring using select, poll, epoll, kqueue and similar mechanisms, we have observed this approach to typically reduce the number of system calls for heavily loaded applications—and thus with a constant stream of events to process—by more than 99%.

Allowing the network stack to expose socket buffers to the applications might appear as a violation to our isolation—and no sharing—guarantees. Nevertheless, we only share unidirectional (and lockless) single-producer and single-consumer queues. Once data are placed on the queue, they are never accessed again by the producer, thus there is no competition and no contention for the shared data structures. In addition, once a socket is open and the system establishes the shared-memory channel, an application automatically communicates with the right network stack replica without exact knowledge of which replica is on the other end.

Using shared-memory channels without knowledge of which and how many replicas and heavily loaded applications communicate, allows our implementation
to be completely agnostic to the number of stack replicas, and thus effectively scale to a large number of network stacks (and cores). The only time when the application uses the exact information on which replica handles each socket (exported to the library level) is when the load is low and the application needs to block using a `select`-like mechanism. In such a case, the application indeed needs to split the set of requests and request notifications from each network stack replica available. This is, however, not a scalability concern, given that lightly loaded applications can spare cycles for management activities.

Avoiding system calls when the load is high allows applications to run undisturbed on their own cores, without being interrupted by blocking operations or having to interleave their execution with the system on the same cores. Figure 6.3a shows a hypothetical example of a highly loaded memcached server, a popular in-memory key-value store, running on a monolithic system. Each core hosts a memcached worker thread processing a fraction of the network traffic. Our experience—also confirmed by other researchers [81]—indicates that it is realistic to expect approximately 70-80% of the execution to happen in the kernel, mostly in steering packets to the right cores, processing network protocols, and passing data through the sockets. On the other hand, the load of NEAT (or FlexSC and IsoStack) is not balanced, as different cores have different roles. Figure 6.3b shows an example of a
similarly loaded memcached server hosted by NEAT. Since the applications do not share their cores with the system, they can take full advantage of private CPU structures such as caches, TLB, and branch predictors. Table 6.1 depicts the differences in terms of instruction cache miss rate. The table compares lighttpd—a popular web server—running on Linux and on two configurations of NEAT: (i) default configuration, which avoids system calls when possible; (ii) syscall-only configuration, which always uses system calls. As expected, the results demonstrate that the default configuration of NEAT experiences a significant reduction in instruction miss rate.

In contrast to the monolithic setup, NEAT uses only a single core to handle the network card interrupts, interact with the device, and access the driver’s data structures. While this core is likely not 100% used, it cannot be used to host another process. The figure shows that the other two cores handle the networking load and in combination with the application cores deliver similar throughput as the ones hosted by Linux. The remaining core (OS) hosts the rest of the system and cannot be meaningfully used for networking or running the application unless the core implements multiple hyper-threads. Nevertheless, it is used in our current prototype to run all the remaining services of the operating system—which are not the focus of this paper.

### 6.3.3 TCP Connections

In this work, we primarily focus on TCP since it interests most of the internet traffic and, in contrast to connectionless protocols, raises more interesting state management challenges to maintain per-connection state. With TCP, each endpoint plays a different role, either initiating connections (client) or accepting connections (server) from remote clients. Similarly, when establishing a client connection, the application and the system select the network stack replica to handle the connection, while, when accepting a connection, the NIC is the first entity to process and observe each packet. Once the decision on the replica responsible to handle the connection is made, both the NIC and the applications must honor the choice.

This process is transparent for the application programmer, since the library automatically selects the stack for the outbound connections. In the case of inbound connections, in turn, the network stack requests the library to provide the mappings for the socket buffers. The NIC is responsible to steer all the packets of the same connection to the same replica, using dedicated connection tracking support.

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Linux</td>
<td>8.5%</td>
</tr>
<tr>
<td>NEAT- forcing system calls</td>
<td>4.5%</td>
</tr>
<tr>
<td>NEAT- w/o system calls</td>
<td>1.5%</td>
</tr>
</tbody>
</table>

Table 6.1: lighttpd I-cache miss rate on Linux and on a dedicated core in NEAT, with or without using system calls.
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). For example, a machine hosting a web server listening only on port 80 would handle all the connections in a single network stack replica, leaving all the other stacks completely unused. 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. 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. The accept implementation, for instance, checks for any subsocket with an available incoming connection and simply “accepts” the connection from it.

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 [68; 119]. 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.

6.3.4 Scaling Up and Down

To obtain peak performance, NEAT uses dedicated cores for some of its components. Replicating such components across different replicas (and thus cores) faces the natural challenge of exhausting the limited number of available cores. To address this problem, NEAT can scale the number of replicas 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, e.g., idle NIC drivers.

The system boots with a predefined number of replicas depending on the expected load and available cores. When the load exceeds the expected maximum, scaling up is necessary, NEAT automatically creates a new replica. It announces itself to the NIC drivers, which, in turn, request the network card to use a new queue. Since the connection tracker has rules for the existing connections, their distribution remains intact as long as each connection exists. The NIC assigns each new connection to a particular replica with the same probability across all the replicas. While this may initially lead to imbalance in load distribution (depending on exist-
ing connections), we expect the system to quickly rebalance itself as soon as existing connections terminate and new connections arrive.

When the load drops again, NEAT can also scale down and terminate some network stack replicas. Since the connections are distributed across all the replicas fairly uniformly, simply shutting down a single replica would result in abruptly terminating all the TCP connections handled by the replica—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 degrade performance due to the cost of copying TCP-related data structures and data not yet consumed by the application or still waiting for delivery acknowledgment. The system would also need to reset the connection tracker in the NIC. This strategy is overly complicated and also violates NEAT’s isolation principles, potentially introducing subtle scalability and reliability issues. For these reasons, NEAT adopts a radically different strategy, which (i) marks the necessary replicas as in termination state; (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 closing strategy without breaking any of the existing connections. The only 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—i.e., 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. This allows NEAT to colocate some of the replicas on the same core while preserving fast user-space communication, given that each replica maintains its own hardware context. As a matter of fact, as shown in §6.5, 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 [114]. The conclusion is that scaling down the number of replicas is actually not always necessary and, when hyper-threading is available, it is better to scale down by scheduling decisions without terminating the replicas.

### 6.3.5 Scaling NIC Drivers

The only performance-sensitive component that NEAT does not actively scale up is the network card driver, since none of our tests demonstrated the driver to be a potential performance bottleneck and also other researchers [123] have reported 10G line rate processing on a single core. From the reliability point of view, it is possible to recover NIC drivers seamlessly [72].
6.3.6 Reliability

Isolation and replication 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. In its current prototype, NEAT opts for a completely stateless 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 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.

6.3.7 Security

Replication has been previously proposed for security purposes, using synchronized multivariant execution to detect arbitrary memory error attacks [45; 128; 129]. Unlike these approaches—which incur high run-time overhead—NEAT pursues the less ambitious goal of enforcing address space re-randomization [137] 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, we found remarkable that 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 [34; 137].
6.3.8 Multi-component Network Stack

For increased reliability, NEAT can be configured at compile time to vertically split each network stack replica into multiple isolated components, resulting in a finer-grain multi-component network stack, much like NewtOS [75] or HelenOS [105]—which pioneered splitting the network stack along the protocol processing boundaries. Figure 6.4 presents a simplified version of our multi-component network stack, showing only the IP and TCP components which NEAT uses for TCP processing, but additional UDP and packet filter components are separated out as well in our current prototype.

Although the multi-component network stack requires more cores and internal communication, it also yields improved reliability since it fully isolates faults in smaller network components. Excluding TCP, the other components are essentially stateless (or pseudostateless) and thus are more easily amenable to application-transparent recovery even with the simple stateless recovery strategy supported by our current prototype. Even when adopting more heavyweight stateful recovery strategies such as the one described in [65], our state splitting strategy effectively reduces the state surface to recover and reduces the likelihood of state corruption.

6.4 Architectural 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. The latter 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 operation. This strategy eliminates the need for expensive inter-processor interrupts and kernel-assisted process halting. Note that NEAT actually switches to such slower communication channels as needed automatically, in particular when the load is low and colocating 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 and the ability to leverage MWAIT-based communication channels. We evaluate the benefits of hardware multithreading in §6.5.

In addition, efficiently scaling the network stack replicas requires dedicated NIC support to split the packet flow. Commonly available NICs feature many pairs of queues for transmitting and receiving. 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 particular, thanks to the widespread adoption of virtualization, modern network cards can already classify and steer packets to different endpoints (i.e., 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 thousands 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 requires many read-write transactions across the PCI Express bus to configure the filters, which can negatively impact network processing [58]. 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 [38]—hence offloading the work done by receive flow steering (RFS [71]). Due to performance concerns, the driver only samples each $20^{th}$ 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.

Another feature already present in modern NICs is packet duplication, commonly used for packet broadcasting purposes across virtual machines. The NICs we experiment with in our current prototype, however, do not have this feature available without virtualization. In particular, NEAT would benefit from a NIC delivering a copy of each ARP reply to each network stack replica, since no knowledge is available on the replica which sent the original query. Moreover, each of the replicas needs a similar set of ARP translations. Maintaining a shared ARP table is not an option due to our isolation requirements. In our experiments, we compensated for the missing hardware by trusting that packets in our testbed have matching ethernet and IP addresses which we use to fill up the ARP tables.

Summarizing, similar to Peter et al. [120], our 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 believe that, as modern NICs already evolved to match the requirements of monolithic systems, scalability and reliability of design can effectively initiate similar developments.
6.5 Evaluation

We evaluated NEAT using two different multicore machines, subject to the hardware availability in our lab: (i) a 12-core AMD Opteron 6168 (1.9 GHz) and (ii) a dual-socket quad-core Intel Xeon 2.26 GHz (E5520). The AMD machine supports more physical cores, but the Xeon machine features 2 hardware threads per core (Hyper-Threading). For our experiments, we used a 10G Intel i82599 network card.

For our scalability evaluation, we selected lighttpd, a popular web server. Lighttpd serves static files, cached in memory to avoid interference with other operating system components.

We evaluated NEAT in both its single- and multi-component configuration. In the figures (for example in Figure 6.6), we denote such configurations as NEAT N×x and Multi N×x (respectively), where N refers to the number of replicas used. We also refer to a particular replica R as NEAT R and TCP (or IP) R in the two configurations considered (for example in Figure 6.5).

Workload

To evaluate the scalability of our solution we relied on the httpperf benchmarking utility to open persistent connections and repeatedly request (100 times) a small 20-byte file over each connection. This setup does not overload the 10G NIC, allowing us to scale up the number of servers freely. A lighttpd instance serving 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. In contrast to workloads sending large TCP segments or receiving back-to-back requests, the hardware is not of any help in offloading.

6.5.1 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 6.5 illustrates both configurations in their best-performing setups and Figure 6.6 presents scalability results for the different configurations of NEAT. In our experiments, we dedicated one of the cores to the operating system (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 was 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 syscall-less socket design. This leaves our testbed with 10 cores available for the network stack and lighttpd. As Figure 6.6 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 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. While we do not have manycore machines at our disposal, we believe NEAT would in fact scale to those machines with no restrictions. The single-component version of NEAT, on the other hand, can scale further. In particular, the NEAT 2x configuration 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. Using all 12 cores, lighttpd can handle 320 kilo-requests per second (kprs). We configured Linux to assign one of the 12 NIC queues to each core and we enabled RFS. While with 3 single-component replicas NEAT reached only 302 kprs, had the AMD processor support for multiple hardware threads per core, we could have also used the OS and SYSCALL cores and likely outperformed Linux. We substantiate this claim in the next section.

### 6.5.2 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 [9]) and, above all, as other researchers have suggested [83; 111; 143; 149], 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 6.7 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 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 6.8). We use HT to denote the NEAT configurations that use hyper-threading.

In contrast to the AMD machine, deploying the TCP and IP of the network stacks leaves only 4 cores available for the testing application. NEAT can now, however, take advantage of hyper-threads to run 8 web server instances. Results in Figure 6.9 show, 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 to run 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 6.9—uses threads on the cores occupied by the network stack itself, first using both TCP cores (6 lighttpd instances) and then both IP cores as well (8 lighttpd instances). In the latter configuration, Figure 6.9 shows the throughput peaking at 322 krps.

Finally, we colocated two replicas on different threads from the same cores (Multi 2x HT, Figure 6.8)—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 1-replica configuration, it can process up to 8 lighttpd instances (4 cores, both threads).

When evaluating single-component configurations, we used up to 4 replicas, as
6.5. EVALUATION

Figure 6.10 shows. The labels (NEATx and WebX) illustrate the order in which we scaled up NEAT and lighttpd. We present the results in Figure 6.11. 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, which is 13.4% more than the 328 krps maximum we measured for Linux, running lighttpd on each of 16 threads, using all cores to 100%.

6.5.3 Impact of Different Configurations

Finally, we evaluated the impact of the different configurations of NEAT and compare their overhead in detail. We used a modified test issuing only a single request per connection, which significantly increases the load on the 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 6.12 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 two such replicas. The same holds for the single-component stack. This is primarily due to the fact that lightly loaded components have higher communication overhead and often sleep, which introduces latency that is more evident in the multi-components stack. A component may spend as much as half of its execution time polling, while the polling time decreases as the load rises. Table 6.2 presents samples collected using statistical profiling of the NIC driver under a range of loads serving 3 replicas. When the driver is often idle, it spends a significant portion of the active time suspending and resuming in the kernel (unfortunately MWAIT is a privileged instruction on In-
CPU load | Active in kernel | Polling | Web krps
---|---|---|---
97% | 0.1% | 7.4% | 242
88% | 5.4% | 19.7% | 90
60% | 14.2% | 27.9% | 45
6% | 33.3% | 51.8% | 3

Table 6.2: 10G driver - CPU usage breakdown.

![Figure 6.13](image)

Figure 6.13: Expected fraction of state preserved after a failure vs. max throughput across different network stack setups.

tel) and polling the 3 stacks and NIC queues. As the load grows, the fraction of the “wasted” time shrinks and, 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 network packets. Other components show similar behavior. Note that the CPU usage grows sharply, but levels off when the load is high since the driver trades the “wasted” time for packet processing. While the CPU load is 60% when the lighttpd handles 45 krps, it is only 88% when the load doubles, and it is still not 100% when the number of requests increases up to almost 5-fold. When the load grows, the network stack scales and the advantage of having multiple replicas is more obvious.

6.5.4 Reliability

We also evaluated the reliability guaranteed to be provided by NEAT across the different configurations. Figure 6.13 shows the expected fraction of the preserved state after a failure against the maximum throughput across all the different configurations we evaluated on the Xeon. We used the code size 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. For this purpose, we assumed the simple (and stateless) recovery strategy supported by our current prototype, which results in (only) the TCP state being always irrecoverable—after a TCP failure. As the figure shows, both performance and reliability increase with the number of available cores across all our configurations, confirming that our design allows scalability and reliability to effectively coexist with no compromises. In addition, as the figure confirms, 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-grain isolation and allocation policies.

6.6 Conclusion

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