Toward Scalable Replication Systems with Predictable Tails Using Programmable Data Planes

Sean Choi  
Stanford University

Seo Jin Park  
Stanford University

Muhammad Shahbaz  
Stanford University

Balaji Prabhakar  
Stanford University

Mendel Rosenblum  
Stanford University

ABSTRACT

Conventional distributed data storage services, like databases and file systems, rely on replication for fault tolerance; as a consequence, the performance of these services depends heavily on the performance of the underlying replication system in use. Existing replication systems, built using a replication protocol (e.g., CURP), are implemented as user-level processes capable of performing replication with relatively low latencies (~10+ µs). However, such user-level processes are susceptible to performance degradation at scale, due to software overheads (e.g., operating system and networking stack), and contention for server resources (e.g., CPU, disk, and memory) between multiple processes; thus, leading to higher latencies with longer tails.

In this paper, we posit that replication systems can achieve lower latency and scale without compromising (tail) latencies by exploiting the characteristics of emerging programmable data planes. To support our thesis, we build a system, called ARGUS, that takes advantage of data plane’s close proximity to the wire, minimal software overhead, and line-rate throughput to accelerate replication. ARGUS maps key components of a replication protocol (e.g., replication, storage, and recovery) to various processing and memory regions of SmartNICs equipped with a programmable match-action data plane. Our preliminary evaluation shows that, in comparison to CURP, ARGUS reduces mean and 99.9th-percentile latencies by 2x and 2.2x respectively, with 6.7x higher throughput. In addition, it lowers the gap between the 99.9th-percentile and median latencies by about 3.3x. Finally, increasing the replication factor in ARGUS has a negligible effect on the tail latency of the system, i.e., an increase of 0.12 µs per witness compared to 12.86 µs in CURP.

CCS CONCEPTS

• Networks → Programmable networks; In-network processing; • Computer systems organization → Reliability;

KEYWORDS

Replication; Programmable network; P4; SmartNIC; NPU;

ACM Reference Format:


1 INTRODUCTION

Many distributed systems today use replication to ensure high availability and to minimize the effects of failures [27]. For example, in a master-backup replication system [3], a master first replicates a client request to a collection of backup servers and acknowledges only when all servers commit their copy of the request. This ensures that the requests are durable and can survive individual server failures.

Unfortunately, replicating requests is costly and introduces overheads in two significant ways: increased resource utilization and reduced application performance. First, each backup server in a replication system consumes additional resources, utilizing CPU cycles and memory to process and store replicated requests [24, 28]. For example, replicating requests of relatively simple applications, like key-value stores, can consume up to 75% of the CPU of a server [14]. Second, client operations must wait for all backup servers (or a majority in a quorum-based system [21]) to finish processing the replicated requests before proceeding forward. This results in a blocking call needing two round-trip times (RTT) to complete: between the client and master, and master and the backup servers. Recent proposals for new replication protocols [23, 24] allow clients to send requests to both the
master and back servers in parallel, such that the replication completes in a single RTT.

However, even with these new proposals, the latencies of clients’ operations are still susceptible to the tail-at-scale effect [18], as clients must wait for replication to complete at both the master and backup servers. All of the master and the backup servers exhibit long tails due to various software overheads (e.g., networking stack and context switching overhead); and with the addition of each new backup server, the clients experience higher latencies with greater probability (see Figure 6). This effect is known to be especially worse in public clouds, where collocated applications and background jobs interfere with the master and backup processes further inflating tail latencies [18, 26].

In this paper, we argue that we can overcome these issues in existing replication systems by taking advantage of the recent advances in SmartNICs with programmable data planes [6, 8, 19] and their unique position in the infrastructure. First, SmartNICs are becoming more flexible and easier to program, enabling developers to program various aspects of their data plane (i.e., parsing packets, matching on header fields, and executing custom actions). They are also getting increasingly powerful and are being built on a variety of hardware architectures (e.g., CPU [8], ASIC [6], and FGPA [13]) that contain hundreds of microprocessors running at multi-GHz speeds and 10+ GBs of on-board memory.

Hence, many data center providers are already deploying and running complicated logic (e.g., hypervisor networking) on SmartNICs with little or no loss in performance [19]. Second, SmartNIC is a gateway between the server and the network, and can process and monitor all data entering or leaving the server. Thus, the SmartNIC can react quickly to incoming data with less processing overhead (i.e., without invoking kernel networking or hypervisor stack) than any other compute unit (e.g., CPUs or GPUs) in the server. Finally, the SmartNIC can mitigate the tail-at-scale effect. With a large number of cores and pipelined design, the SmartNIC can quickly process a request without contending for NIC resources. Furthermore, these cores typically execute in a run-to-completion model [2], ensuring predictable processing times.

To support our argument, we present a replication system called ARGUS that exploits these characteristics of SmartNICs and implements a general-purpose store for caching replicas of clients’ requests that are sent to the master. ARGUS adapts a replication protocol called Consistent Unordered Replication Protocol (CURP), a recent 1-RTT consistent replication protocol that ensures strong consistency, and uniquely maps the temporary cache on the SmartNIC as data plane programs (e.g., P4 [15] and Micro-C programs [10]). The cache is designed and implemented to maximize performance and ensure isolation by taking advantage of the SmartNIC’s specific architecture. Similar to CURP, clients replicate their requests to both the master and caches in ARGUS, allowing the master to asynchronously and safely (i.e., recovering from the caches under failures) replicate requests to the backup servers, in batches for higher throughput. Unlike existing solutions, ARGUS effectively removes the cache from being a user-level process—significantly improving the completion time of these operations—, all the while handling other network traffic and not sacrificing server resources.

In the following sections, we begin by discussing the background on replication protocols and programmable SmartNICs, and the motivations for this paper (§2). We then provide an overview of ARGUS (§3) with preliminary end-to-end evaluations of ARGUS (§4), and conclude by outlining the plans for future work (§5).

\section*{2 BACKGROUND & MOTIVATION}

\textbf{Consistent Replication and CURP.} Traditional replication protocols typically take 2 RTTs to complete, which doubles the response time of an operation compared to when it is not replicated. To ensure consistency across backup servers participating in the replication process, operations from multiple clients must first be ordered and then replicated in the specified order. Otherwise, each backup server may receive and commit operations in a different order leading to inconsistencies in data stored across these servers. To achieve ordering, traditional replication protocols require a primary server that receives all operations from clients, serializes them, and finally replicates them to the backup servers in order. Thus, this need for a primary server causes the clients to wait for 2 RTTs to complete an operation—requests originating from clients to primary server and then primary server to replicas (Figure 1).

CURP [24] takes advantage of the fact that commutative operations do not need to be ordered, thus these operations can be committed in any order without compromising the consistency of the stored data. To allow 1-RTT replication for commutative operations, CURP introduces the notion of witnesses.
of a witness. Clients replicate their requests to one or more of these witnesses in parallel, as well as send it to a master server. Instead of waiting on the backup servers to finish committing the replicated requests, the master in CURP replies back asynchronously. Likewise, witnesses cache the requests and reply immediately, enabling CURP to achieve 1 RTT for commutative operations.

Each witness stores a new operation only if it commutes with all of the cached requests that are not yet committed to the backup servers. This ensures that witnesses can replay requests to the master in any order without affecting the consistency of the system. So, under failure, the master in CURP recovers from the backup servers as normal, and then requests the witnesses to replay the set of requests that have not yet been committed.

Although the replication completes in 1 RTT, each added witness has a direct impact on the overall performance, especially on the tail latency. The witnesses reside on the critical path of the client operations, thus each witness collectively increases the tail latency of the system. In fact, it is shown that each added witness increases the tail latency by about 10 µs [24]. In contrast, a ARGUS witness has lower and more predictable latency and, therefore, does not cause such increase (§4).

Programmable Network Devices and SmartNICs. Traditional networks comprise of data-plane devices having a fixed set of features to accommodate all customers using a single monolithic and vertically-integrated device. This fixed-function nature of these devices makes it hard for vendors to add new features and for customers to manage these devices, forcing them to account for features that they do not require.

Emerging programmable data-plane devices [16] help alleviate these issues. Network operators can specify their features of interest as programs that are then used to configure these programmable devices. Adding or removing a feature is just a matter of updating and compiling a new program. P4 [15] has emerged as a representative language for writing these programs. It provides constructs to specify packet-processing logic for parsing headers and running custom functions using primitive actions defined in the language specification [1]. One example of these programmable network devices is SmartNICs: a network interface card (or NIC) that can run tasks that a CPU normally handles (e.g., hypervisor switches [19], TCP offload engines, and more).

We select ASIC-based SmartNICs from Netronome [6] to implement ARGUS; however, ARGUS can run on other SmartNICs too [19], with varying performance and cost. We chose these SmartNICs as they support P4 and a C-like programming language called Micro-C [10], allowing us to easily implement generic programs to run on the NIC. The code implemented on the NIC is directly connected to the server's Linux operating system via updates to the kernel network core that runs on the server. In addition, Netronome SmartNICs contain hundreds of cores, called Network Processing Units (NPUs), capable of running asynchronously at GHz speeds with 10+ GB of on-board DRAM. In ARGUS, we capitalize on these features and run the witnesses, consisting of simple and small units of execution, directly on SmartNICs’ data plane.

3 ARGUS ARCHITECTURE

We now describe the architecture of ARGUS in detail. ARGUS consists of three distinct entities: a master, clients, and a pool of data-plane witnesses.

Master. We refer to master as the main application that requires replication and consistency assistance via ARGUS. Examples of such applications are consistent Redis key-value store [12], Zookeeper coordination system [21], Cassandra distributed key-value store [22] and more. In ARGUS, we modify the master to recover from data-plane witnesses under failures. Furthermore, the master notifies the witnesses to garbage-collect stale entries once an operation is committed to the backup servers.

Clients. ARGUS requires clients to send operations to both the master and the pool of data-plane witnesses. A client therefore opens up and maintains connections between the master and the witnesses, sending requests to each of them in parallel. We can implement clients’ replication logic in the data plane, similar to witnesses, thereby reducing the packet processing overhead on server CPUs. We briefly discuss these optimizations in §5.

Data plane witness. A data-plane witness implements the following three protocols: store for caching duplicated
requests, remove for garbage collection of stale requests, and recover for handling failures by sending the cached requests back to the master. In ARGUS, we run these protocols entirely in the data plane of network devices to ensure low and predictable latency. Doing so, therefore, enforces a set of requirements that an ARGUS data plane must satisfy:

- **Custom parsing and processing (R1):** Able to parse custom headers (e.g., an ARGUS header shown in Figure 2) and run programs that implement the store, remove, and recover protocols.
- **Checking commutativity (R2):** Able to generate hashes from incoming requests and check commutativity between the requests by comparing the hash values [24]. This is needed for both the store and remove protocols.
- **Custom packet generation (R3):** Able to generate custom packets in response to the recovery requests from the master.
- **Low processing overhead (R4):** Able to run alongside traditional NIC logic (i.e., switching).

ARGUS data plane, built using P4 programmable ASIC-based SmartNICs, satisfies these requirements. As shown in Figure 3, a data plane witness parses the ingress packet and checks if it contains an ARGUS protocol by looking at the TCP header and its destination port (R1). If it does, the witness continues reading the ARGUS header to identify the type of protocol. The non-ARGUS packets are forwarded to the host as-is (R4).

Based on the protocol type, the data-plane witness then executes the corresponding program, written as a Micro-C code (R1). For store and remove protocols, the data-plane witness creates a set of lookup tables and computes a hash of the ARGUS header using the data plane’s hardware primitives (e.g., CRC32, and L3 & L4 checksums) (R2). Using this hash as an index, the data-plane witness then checks for commutativity by querying the lookup tables to see if a value exists at the indicated index. If a value does exist, then it indicates that a write to the same variable is being performed. Since overwriting the same location causes inconsistency and breaks commutativity [24], the data plane aborts such write requests (R2). ARGUS can configure the size and storage location of these lookup tables based on the system’s requirements, such as performance, memory size, and isolation needs. If a system requires lowest latency or strict isolation, ARGUS uses the local registers or internal memory [10] for the lookup tables; thus, giving the best performance with strict isolation but at a cost of limited memory space (10+ MBs). For cases that need larger witness caches, ARGUS uses the larger external memory (2+ GBs) for the tables. ARGUS currently uses a simple heuristic to decide which configuration to pick for a given environment. We leave it as future work to identify optimizations that automatically store new and hot requests on the faster local memory and registers, and move older requests to the slower external memory.

During recovery, the data plane witness sends a set of packets in response to a master’s request, with the contents of the lookup table. It then clears the lookup tables (acting as a cache) once the master acknowledges the completion of the recovery phase.

Lastly, the ARGUS’s data-plane witness, using ASIC-based SmartNICs, can generate packets and support running programs on separate NPUs with their own memory space. This enables ARGUS to run witness logic alongside traditional NIC features with low or no overhead (R4).

4 EVALUATION

In this section, we provide our experimental methodology and preliminary performance results for ARGUS. The results demonstrate that ARGUS provides significant performance improvements for average and tail latency, as well as, throughput (§4.2).

4.1 Methodology

**Baseline system.** In our baseline system, we implement the master as an in-memory key-value store. We modify Redis v3.2.8 [12]—a popular key-value store—to interact with witnesses, as discussed in §3, to improve clients’ response times while ensuring durability and consistency. The modified Redis instance supports protocols for recovering from witnesses and issuing garbage-collection operations to the witnesses for persisted data. Similarly, baseline witnesses are implemented as a custom Redis instance that supports the add, remove, garbage collection, and recovery protocols. Finally, we implement a custom C++ Redis client [20] that issues requests to both the master and witnesses.

In a typical setup, a Redis store is not durable. The data is stored in memory and is lost when it fails or shuts down. Thus, to ensure durability, Redis commits requests to an append-only log file before acknowledging the clients. However, doing so incurs 10x to 100x performance penalty for both throughput and latency. With ARGUS, a Redis store writes to log files asynchronously and acknowledges clients...
Witness through-put of 100,000 requests with a random key of 30 bytes and random value of 100 bytes in size. To emulate tail behavior, we artificially induce load on the master and witness servers pinned to a core using stress-ng [11], which runs a collection of representative CPU stress methods in the background. Both workloads generate 100,000 requests with a random key of 30 bytes and random value of 100 bytes in size.

4.2 Preliminary Evaluation

Our evaluation demonstrates that ARGUS significantly improves the system’s performance with much lower mean and tail latencies and higher through-put compared to CURP.

ARGUS witnesses show 2x faster response times (Table 1) and 6.7x higher through-put (Table 2) compared to Redis witnesses—Redis witnesses operate around 113 Kops/s, which is consistent with published benchmarks [9]. The master in the end-to-end ARGUS system adds an extra 27 µs and 23 µs to witnesses’ mean and tail (i.e., 99.9th percentile) latencies, respectively, in ARGUS (Table 1), but still shows an improvement of 1.3x and 1.8x over using Redis witnesses.

One of the distinguishing characteristics of ARGUS is its ability to enforce strict and, hence, predictable tail latency. The evaluation shows that using ARGUS, the difference between median and 99.9th percentile latencies of a witness is just 5.84 µs—3.3x less than that of the Redis witness (Figure 5). Furthermore, ARGUS mitigates the tail-at-scale effect; adding a new witness in CURP increases the tail latency by 12.86 µs, whereas adding a witness in ARGUS incurs negligible increase (0.12 µs) in tail latency (Figure 6).

As an added benefit, we observed that ARGUS completely eliminates the added CPU cycles incurred when running the witnesses on the server. (It is reported in [24] that witnesses incur around 7% additional CPU cycles.) As for the memory consumption, ARGUS frees the the host memory, but instead consumes the equivalent amount of SmartNIC memory. Finally, we were able reduce the effect of ARGUS on other network traffic by allocating subset of the NPUs to only run ARGUS. Given that NPUs in SmartNICs are often over-provisioned, using only a small amount of NPUs dedicated for ARGUS has limited effect on the sideway traffic, and will most likely not degrade the performance.

Final thoughts. The performance improvements we observe in ARGUS for both latency and throughput is primarily the result of reduced processing overhead that each request experiences. In ARGUS, both packet processing and witness computation happens in the data plane. However, in CURP each request needs to traverse the NIC, the OS network processing stack, and the witness process. Furthermore, user processes, OS kernel, and background jobs running on servers constantly contend for CPU and memory resources—stalling whenever the required resources are not available. As CURP runs entirely in software, these characteristics of a server therefore lead to more operations and delays, degrading the performance of the overall system. With data-plane witnesses in ARGUS, we mitigate many of these overheads. ARGUS dedicates some cores for witness operations, while using the rest of the cores for regular NIC operations, thus, further mitigating performance degradation that can arise.
due to contention for NIC resources. Higher throughput of these witnesses also implies that a replication system can serve more masters simultaneously, requiring application developers to deploy less witness instances to support the same number of applications; thus, saving CPU cycles and operational cost of running a replication system.

5 RELATED & FUTURE WORK
ARGUS is still in its early stages. In this section, we list future work necessary to deploy ARGUS in real settings, and also compare ARGUS against existing kernel-bypass techniques.

Running client-side replication on SmartNICs. The current implementation of ARGUS requires clients to be aware of data-plane witnesses. The clients must maintain connections with all of the witnesses, requiring a tighter coupling with the underlying replication protocol and the infrastructure. Furthermore, busy-polling a NIC to wait for responses from master and witnesses is inefficient and wasteful, causing clients to waste precious CPU cycles.

To mitigate these issues, we can implement request management and replication to witnesses in the data plane, itself. Doing so would allow clients to generate requests as they would without replication, requiring no changes in the clients. The data-plane NIC on the clients’ server will interpret these requests, replicating the packets locally or using scalable in-network multicast (e.g., Elmo [25]), and forward the replicas to the participating witnesses after adding the ARGUS header on these packets (Figure 2). As for the response, the SmartNIC sends a single response, aggregated from all witnesses and the master, to the clients.

Reliable data-transfer protocols on SmartNICs. SmartNICs implement a restricted compute model and cannot yet run the entire TCP stack in the data plane. Some NICs exist but they are not reliable and have scalability issues (e.g., number of supported connections). Therefore, current implementation of ARGUS requires other forms of data-transfer protocols to keep connection state and ensure reliable delivery. Fortunately, such protocols exist (e.g., RoCEv2 [4] and Lightweight Transport Layer [17]) that today’s data-center providers deploy in production. Alternatively, we can use emerging SmartNICs containing both programmable data planes and general-purpose CPUs [6] to let ARGUS manage connections using traditional protocols like TCP.

Other domain-specific hardware accelerators. A number of domain-specific architectures and flavors of SmartNICs exist [6–8, 19]. Therefore, it is worth investigating which of these hardware is a best fit for ARGUS. We have specifically chosen Netronome ASIC-based SmartNICs due to its capabilities to run P4 and custom C programs directly in the data plane, allowing ARGUS to benefit from both the ease of programming and low overhead for running custom programs. In addition, Netronome SmartNICs are much cheaper than NetFPGAs, which can cost up to $7,000 [13]. One may argue that we can achieve higher performance or higher-level of offload, if we use other types of SmartNICs, such as Mellanox Bluefield [8]. The design of ARGUS does not limit itself to run only on ASIC-based SmartNICs, and it can run on any NIC that is capable of doing custom packet processing. Thus, a possible future work would be to research and evaluate alternative SmartNICs or hardware accelerators that can run ARGUS more efficiently.

Comparison against kernel-bypass techniques. Our current evaluation compares Redis witnesses running on bare-metal servers with ARGUS data-plane witnesses. We believe that optimizing the server software stack using techniques like kernel bypass (e.g., DPDK [5]) may improve the performance of Redis witnesses. However, we expect the performance would still be lower than ARGUS, since any request that reaches the Redis witness must be processed by the NIC first, resulting in a cost that no software optimization can eliminate. ARGUS mitigates this cost by running witnesses directly in the data plane.

6 CONCLUSION
In this paper, we proposed ARGUS a system that harnesses programmable data plane and SmartNICs to improve the performance of replication protocols. We discussed the design and implementation of building ARGUS and related challenges that we are currently pursuing to overcome. In addition, we demonstrated how ARGUS can sustain low and predictable latency, especially at the tail. Our preliminary evaluation shows that ARGUS improves both latency and throughput by 2x and 6.7x compared to existing approaches. This was achieved while saving the costly host CPU cycles and memory, and instead using the relatively cheaper SmartNIC NPU cycles and NIC memory. ARGUS presents a promising future for scaling replication systems using programmable data planes that achieve both increased performance and reduced host resource usage.
ACKNOWLEDGMENTS

We thank the members of the Stanford Platform Lab and the anonymous APNet reviewers for their valuable feedback that helped improve the quality of this paper. This research was supported by industrial partners of the Stanford Platform Lab. Seo Jin was supported by the Samsung Scholarship and Muhammad Shahbaz was also supported by Air Force Research Laboratory (AFRL) and Defense Advanced Research Projects Agency (DARPA) under agreement number FA8650-18-2-7865. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of AFRL and DARPA or the U.S. Government.

REFERENCES