In Search of a Scalable Raft-based Replication Architecture

Christian Deyerl
Friedrich-Alexander University
Erlangen-Nürnberg (FAU)

Tobias Distler
Friedrich-Alexander University
Erlangen-Nürnberg (FAU)

ABSTRACT

Providing a consistent replicated log across different servers, the Raft consensus protocol greatly facilitates the design of fault-tolerant services. However, due to the protocol following the principle of a single strong leader, such architectures in general do not scale with the number of cores and/or network cards available on each server. To address this problem, we present the NIAGARA replication architecture, which makes it possible to build scalable systems while still relying on Raft for consensus. In particular, we show how NIAGARA parallelizes the process of appending new log entries across multiple Raft instances and discuss NIAGARA's support for read operations with different consistency requirements.

CCS CONCEPTS

• **Computer systems organization → Reliability**.

ACM Reference Format:
Christian Deyerl and Tobias Distler. 2019. In Search of a Scalable Raft-based Replication Architecture. In *6th Workshop on Principles and Practice of Consistency for Distributed Data (PaPoC '19), March 25, 2019, Dresden, Germany.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3301419.3323968

1 INTRODUCTION

The Raft replication protocol [18] represents the foundation of numerous dependable services currently running in production, including key-value stores [6], distributed databases [7], coordination services [9], and message brokers [19]. In such systems, Raft is typically responsible for maintaining a log that is replicated across the participating servers and defines in which order to execute the commands that modify the application state. To ensure the consistency of this log, before appending new entries Raft executes a fault-tolerant consensus algorithm that guarantees that all non-faulty servers add the commands in the same order. Relying on Raft to keep the log consistent significantly simplifies application design and implementation, however, it also introduces problems for use cases in which a high application workload causes Raft to become a performance-limiting bottleneck. Unfortunately, solving such problems by providing Raft with additional computing or network resources in general is not an option due to the protocol's internal structure limiting scalability. That is, in scenarios where the protocol is computation bound, for example due to the application frequently appending comparably small commands to the log, allowing Raft to use more cores on each server usually does not increase performance. In a similar way, adding network cards and/or connections typically does not lead to improvements in cases where Raft is network bound, for example as the result of large entries being added to the log.

In this paper, we address these problems with NIAGARA, a state-machine replication architecture that enables system designers to achieve scalability while still using Raft as underlying consensus algorithm. Like Raft, NIAGARA also maintains a consistent log of state-modifying commands, however, in contrast to Raft, NIAGARA distributes the responsibility for appending new entries across multiple Raft instances. More specifically, a configurable number of loosely coupled Raft instances first create individual instance-local logs, and NIAGARA then deterministically merges these logs into a global log, which is finally accessed by the application. With the involved Raft instances being largely independent from each other, it is straightforward for different instances to effectively and efficiently use different cores or network cards.

In particular, this paper makes the following contributions: (1) It presents the NIAGARA replication architecture that is able to utilize the computing and network resources available on multi-core servers. Unlike existing Raft-based approaches [7], NIAGARA does not require application-state partitioning. (2) It discusses NIAGARA's support for applications that demand read operations with different consistency guarantees. (3) It evaluates the NIAGARA prototype, which is based on one of the most widely used Raft implementations: the protocol implementation of the etcd coordination service [9].

PaPoC '19, March 25, 2019, Dresden, Germany

2 PROBLEM STATEMENT

In this section, we provide details on the Raft protocol and explain how it is typically used to build replicated services. Furthermore, we analyze the scalability limitations generally associated with such architectures and identify the challenges that need to be addressed to solve these problems.

2.1 Background

Figure 1 illustrates a common use-case scenario of Raft in which a stateful application relies on the protocol to keep multiple replicas consistent across different servers. To this end, application replicas only process state-modifying commands in the order in which they appear in a replicated log managed by Raft. In particular, when the application wants to perform a state update (e.g., in the course of executing a client request), the application first submits the corresponding command to Raft. In the next step, Raft then runs a distributed consensus algorithm to replicate the command across servers and agree on the position at which the command will be appended to the log. Once this position is committed by the protocol, Raft instances on all servers add the command to their respective logs. At this point, it is safe for an application replica to apply the command to its local state, provided that the replica has already processed all previous commands from the log. With all replicas applying all commands in the same order, their states remain consistent.

Raft's distributed consensus protocol elects one of the participating servers to act as *leader*, resulting in the other servers to assume the roles of *followers*. Only the current leader starts the consensus algorithm for new commands; if the application submits a command to a follower, the follower forwards it to the leader. To ensure that a new leader is elected in case the old leader crashes or gets disconnected from the system, followers in Raft monitor the leader by periodically exchanging heart-beat messages. If the current leader fails to send such messages over a certain period of time, the followers trigger a new leader election.

As each state modification becomes part of the log, it grows with every state update. To ensure that servers are able to retain the log, Raft truncates it when the log reaches a configurable number of entries. For this purpose, the protocol instructs the local application replica to create a snapshot of the application state and then discards all commands from the log that have been executed prior to the snapshot.

Unlike state modifications, read-only operations accessing the application state do not have to be replicated across servers and therefore do not appear in the log. In general, Raft makes it straightforward for applications to provide read operations with different consistency guarantees [17]: With the application state at all times representing the result of a consistent prefix of the committed sequence of commands, a read that is performed by the application without further coordination returns a *weakly consistent* result. For such reads, it is guaranteed that they see a valid state, however the data they return might be stale, for example, due to the local server not yet having received the latest log entries. To perform a *strongly consistent* read, in a nutshell, an application replica contacts the current leader and requests the log position of the latest committed entry. As soon as the replica has applied all commands up to this position to its local state, it is able to perform the strongly consistent read.

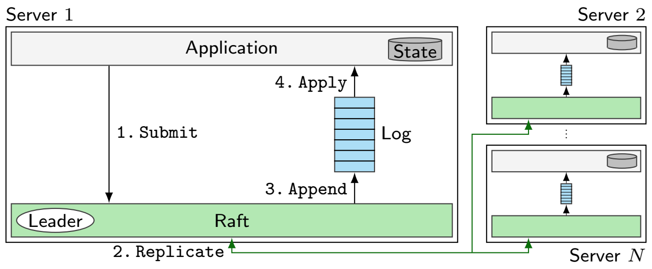


Figure 1: State-machine replication with Raft.

2.2 Limited Scalability

One of Raft's key design principles was the idea of having a strong form of leadership, that is, implementing as much functionality as possible in the leader [18]. While, on the one hand, this approach for many researchers and programmers makes it easier to reason about the protocol, on the other hand, it also limits the scalability of traditional Raft-based replication architectures. In particular, this is due to the fact that in Raft all relevant information (e.g., new log entries) flows through the leader to enable it to fully coordinate the consensus process. Given this internal protocol structure, the single leader may become a performance bottleneck in the face of high application workloads. As confirmed by our experiments in Section 4, providing additional computing and network resources in such case usually does not mitigate this problem, because of Raft being unable to effectively use them.

2.3 Challenges

Considering the drawbacks of the traditional approach, our goal is to develop a scalable Raft-based replication architecture that meets the following requirements: (1) The architecture must ensure consistency while benefiting from multiple cores and/or networks cards. (2) The architecture should still offer the convenience of a single log, without requiring the application state to be partitioned. State partitioning has the potential to further improve scalability (see Section 5), but is outside the scope of this paper. (3) The architecture should not make it necessary to modify intrinsic Raft functionality such as the algorithms for consensus and leader election.

3 NIAGARA

In this section, we present details of our Raft-based replication architecture NIAGARA, which parallelizes the addition of new commands to the log and therefore is able to effectively use the computing and network resources available on multi-core servers. Specifically, we focus on discussing how NIAGARA ensures a consistent log, how it handles log compaction, and how it supports strongly consistent reads.

3.1 Overview

As shown in Figure 2, in contrast to traditional Raft-based replication architectures, a server in NIAGARA does not execute only a single but multiple instances of Raft, typically each one on a separate core. To reduce the number of synchronization points, the Raft instances on a server are loosely coupled and do not interact with each other. In particular, each instance communicates with its peer instances on other servers via dedicated network connections, possibly even using dedicated network cards. Furthermore, each Raft instance follows its regular workflow and appends commands to its own, instance-local log. Based on these local logs, a NIAGARA-specific architecture component, the *sequencer*, creates a consistent global log by merging the local logs in a deterministic fashion. Once a command appears in the global log, application replicas apply it to their respective states.

As a key benefit, relying on loosely coupled Raft instances allows NIAGARA to minimize development overhead by reusing an existing implementation of Raft. Above all, the architecture does not make it necessary to modify the most complex and crucial protocol parts such as the consensus algorithm or the mechanisms for compacting the local log and for electing a leader. With each Raft instance performing its own independent leader election, as illustrated in Figure 2, the leader role (Ⓛ) of different Raft groups might be assigned to different servers. In such case, in contrast to the traditional approach, NIAGARA's multiple application replicas are able to efficiently submit new commands by handing them over to a Raft leader running on their respective servers. If a server hosts multiple leader instances, the application is allowed to choose which instance to use for this purpose; ideally, the application in such situations balances the submission of new commands across the leader instances in question.

3.2 Sequencer

By executing Raft's distributed consensus protocol, each Raft instance ensures that its local log is kept consistent. Guaranteeing that the same also applies to the global log is the main responsibility of NIAGARA's sequencer. To achieve this, a sequencer combines the local logs following a deterministic pattern: in a round-robin order of increasing instance ids $r \in \{1, 2, ..., R\}$, the sequencer selects the next entry from

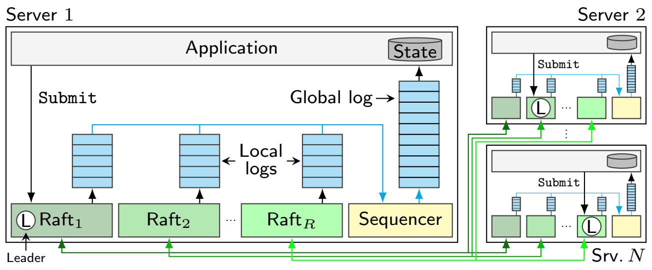


Figure 2: Overview of the NIAGARA architecture.

each local log and appends it to the global log. If there is no new committed command in the local log that is next in line, the sequencer waits for such an entry to become available, independent of whether there are entries remaining in other logs. With all sequencers behaving this way, it is guaranteed that they all produce identical copies of the global log.

To ensure that NIAGARA makes progress when there is at least one local log with commands waiting to be processed, each sequencer periodically executes the following steps: First, for each Raft instance r the sequencer determines the number of committed commands c_r that have yet to be appended to the global log. Next, the sequencer computes the maximum c_{max} of these values. Finally, to each leader instance l that is hosted on the local server, the sequencer submits $c_{max} - c_l$ *no-op* commands. These special commands pass through Raft in the same way as regular commands, but later result in a no-op when the application executes them. Although not having an effect on the application state, no-op commands play an important role in guaranteeing system progress. Once a no-op command is committed and appears in the local log for which a sequencer is waiting, it provides the sequencer with an entry to append to the global log, thereby enabling the sequencer to move on to other logs with newly committed regular application commands.

3.3 Log Compaction

In order to prevent the local and global logs from growing indefinitely, NIAGARA triggers a log compaction as soon as the global log reaches a configurable size. To do so, a sequencer first instructs the application to snapshot the local service state. After this procedure is complete, the sequencer truncates the global log at the position represented by the snapshot, keeping only newer entries whose effects are not yet included in the snapshot. This way, the combination of snapshot and remaining global-log entries still provides all the information necessary to reproduce the current application state, for example, in cases where a server needs to recover from a failure. Apart from garbage-collecting the global log, a sequencer also informs all Raft instances on the same server to truncate the corresponding parts of their local logs.

Overall, NIAGARA's approach to performing log compaction has two main benefits: (1) The mechanism requires only a single application snapshot to garbage-collect the local logs of all Raft instances on a server. (2) While the application is creating the snapshot, Raft instances can continue to execute the consensus process for new commands, thereby minimizing the effects of log compaction on system performance.

3.4 Consistent Reads

As in traditional Raft-based replication architectures, an application replica in NIAGARA is always able to perform weakly consistent reads by directly accessing its state (cf. Section 2.1). Consequently, in the following we focus on presenting NIAGARA's support for strongly consistent reads. For such reads, it is essential to guarantee prior to the read that the state of the application is sufficiently up to date. In particular, all commands that previously have been executed on at least one of the servers in the system must also have been applied by the application replica that wants to perform the read. NIAGARA ensures this by relying on the sequencer as a coordinator for strongly consistent read operations.

Specifically, the task of the sequencer in this context is to identify a position in the global log at which the sequencer's application replica will have seen and processed the commands of all append operations that could have previously been committed on one or more servers. To determine this position, the sequencer learns the current local commit progress of one of the Raft instances by querying the instance's leader and then uses this local-log position to compute the position at which the next command ordered by this instance (in the future) will be appended to the global log. If the sequencer's application replica has processed the global log up to this point, it is able to perform the strongly consistent read operation. Below, we elaborate on this process in detail.

When the application contacts the sequencer to request permission for a strongly consistent read, the sequencer forwards the request to an arbitrary (potentially remote) Raft leader instance r. Upon receiving such a request, the leader of r first determines the position[1] $p_{local,r}$ of its latest committed local-log entry, then ensures that it still assumes the leader role by exchanging heart-beat messages with a majority of its followers [17], and finally returns $p_{local,r}$ to the sequencer. Next, the sequencer waits until the global log grows up to (and excluding) position $p_{global} = (p_{local,r} * R) + r$, with R representing the number of Raft instances per server and $r \in \{1, 2, ..., R\}$. As soon as this is the case, the sequencer allows the application to perform the strongly consistent read under the condition that the application first processes all remaining commands in the global log up to position p_{global}.

[1]If the position of an entry is p, it is the p-th entry to have been appended to the log since system start (i.e., the position is not affected by log compaction).

Querying the latest committed local-log entry from a single Raft leader to decide the earliest point in time at which the application may perform a read in NIAGARA is sufficient to provide strong consistency. This is due to the fact that, as discussed in Section 3.2, the global log is created by merging local logs in a deterministic round-robin fashion. Consequently, if the local log of an instance r only contains $p_{local,r}$ committed entries, then at this point at most $p_{local,r} + 1$ entries from local logs with instance ids lower than r can have already been appended to the global log on any server in the system. Therefore, $(p_{local,r} * R) + r$ marks an upper threshold for the commands that may have been executed before the application requested the strongly consistent read.

3.5 Fault Handling

Relying on a collection of loosely coupled Raft groups, most fault scenarios in NIAGARA are handled by the specific Raft instance affected, without requiring interaction with the sequencer or other Raft instances. In cases where, for example, network connections to remote instances fail, Raft's built-in fault-handling mechanism is responsible for reestablishing the connections. If network problems cause a leader to be separated from the majority of its peers over a longer period of time, the fault-handling procedures of a Raft instance may also involve the election of a new leader. In scenarios in which an entire server crashes, fault handling is not limited to individual instances but conducted by all Raft groups in parallel. Specifically, once the failed server recovers, all of its Raft instances rejoin their respective groups as followers and receive from their leaders the newly appended log entries they have missed while the server was disconnected from the system. If due to log compaction a leader has already discarded some of these log entries, it enables the recovering server to catch up by transmitting its latest snapshot.

3.6 Implementation

Our NIAGARA prototype is based on version 3.3.3 of one of the most widely used Raft libraries: the protocol implementation of the etcd coordination service [9], which is written in Go. All main components of NIAGARA are implemented as separate goroutines, that is, lightweight threads that are dynamically mapped to operating-system threads by the Go runtime. Within each server, components communicate via FIFO channels through which they are able to efficiently exchange messages. The channel connecting the sequencer and the application represents the global log, which to minimize NIAGARA's memory footprint, only stores the committed commands until the application collects them. The interaction between servers is handled by HTTP. To improve throughput, Raft batches multiple commands submitted by the application and replicates them within the same consensus instance.

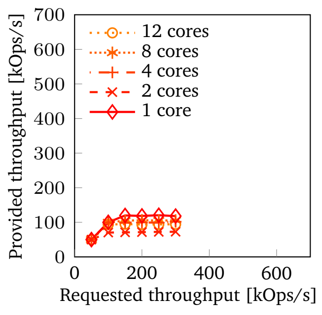
4 EVALUATION

In this section, we evaluate whether NIAGARA is able to benefit from additional computing and network resources. As baseline, we use the traditional replication architecture presented in Section 2.1 (in the following simply referred to as "Raft"), which only comprises a single Raft instance on each server. To get meaningful and comparable results, the two evaluated system implementations originate from the same code base and share as many components as possible. In particular, both systems improve consensus efficiency by replicating new commands in batches (see Section 3.6). All reported numbers represent the average result of three runs.

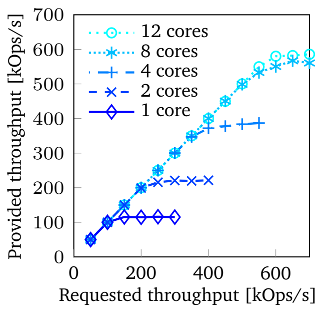
4.1 Throughput

In our first experiment, we determine the throughput with which the two systems are able to append new commands to their respective (global) logs. Furthermore, we investigate whether it is possible to improve performance by allowing the systems to utilize an increased number of cores on each server. For this microbenchmark, we distribute the evaluated system architectures across three 12-core servers (Intel Xeon E5645, 2.4 GHz, Ubuntu 16.04, Linux 4.4.0) that are connected with Gigabit Ethernet. As we are interested in stress testing the consensus layer in this experiment, we keep the size of commands small (i.e., 16 bytes of payload per command) and directly generate the workload in the application, for which we reserve an additional core. That is, for this microbenchmark there is no interaction with clients to avoid interference. In order to evaluate different workload levels, we vary the frequency with which the application submits new commands.

Figure 3 presents the measurement results for this microbenchmark experiment showing the relationship between the throughput requested by the application (horizontal axis) and the throughput with which Raft and NIAGARA actually append new commands to the log (vertical axis). Ideally, both



(a) Raft



(b) NIAGARA

Figure 3: Comparison between requested and provided throughput in Raft and NIAGARA for a microbenchmark continuously appending new entries to the log.

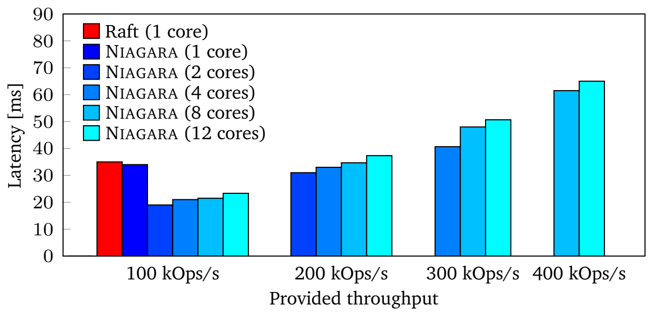


Figure 4: 90th percentile latencies of Raft and NIAGARA at different throughput levels for a microbenchmark continuously appending new entries to the log.

throughput performance numbers match, resulting in data points on the graph's diagonal. When we use the Linux utility `taskset` [14] to allocate a single core for Raft, this is only the case for workloads of up to about 120,000 append operations per second. If the requested throughput increases beyond this level, Raft already reaches saturation and therefore is no longer able to handle all append requests, forcing our workload generator in the microbenchmark application to drop some of them. As our results in Figure 3a illustrate, allowing Raft to run on multiple cores does not solve the problem. Instead, due to increased inter-core synchronization overhead, the throughput achieved by Raft in the evaluated multi-core settings is lower than its single-core throughput.

Repeating the same experiment with NIAGARA, we configure the number of Raft instances to always match the number of allocated cores. As shown in Figure 3b, when using a single Raft instance NIAGARA provides a similar throughput as Raft. However, in contrast to Raft, due to distributing the responsibilities for appending new log entries across different instances, NIAGARA's performance improves for an increasing number of instances. In the setting with 12 Raft instances, when the number of instances matches the total number of cores available on each server, the application inevitably needs to share computing resources with NIAGARA, therefore enabling only small gains compared with the 8-core setting. At maximum, NIAGARA appends almost 600,000 new commands per second to its global log, a factor of 5 improvement over the traditional Raft-based replication architecture.

4.2 Latency

During the microbenchmark experiment presented in Section 4.1, we not only evaluated the throughputs achieved by Raft and NIAGARA, but also recorded how long it took each system to perform the requested append operations. Figure 4 presents the 90th percentiles of the measured response times for different throughput levels and system configurations.

At a throughput of 100,000 operations per second, appending new log entries requires about 35 milliseconds when using the best-performing Raft configuration, that is, the setting in which the protocol runs on a single core; for the other Raft settings that are able to sustain this throughput level we observed 90th percentile response times of up to 100 milliseconds. With the servers in our experimental environment being connected via local-area links, large parts of the response time are not the result of network latency, but instead can be contributed to queueing delays as well as the serialization and deserialization of HTTP messages. As a consequence, NIAGARA is able to provide significantly lower latency than Raft when being configured to rely on multiple Raft instances due to parallelizing protocol execution across different cores. For the same throughput level, NIAGARA settings with fewer Raft instances offer lower latency than those with more instances, because of the sequencer needing to merge fewer local logs to construct the global log.

4.3 Network Resources

In our second experiment, we evaluate the scalability of Raft and NIAGARA with respect to the network resources available. For this purpose, we deploy the systems on 3 servers (4 cores, Intel Xeon E3-1245 v3, 3.4 GHz, Ubuntu 16.04, Linux 4.4.0) and use Linux Traffic Control [15] to split their respective Gigabit Ethernet links into four 250-Mbit subnets. This allows us to vary the amount of network resources each system has at its disposal. For NIAGARA, we create a dedicated Raft instance for each provided subnet. In this experiment, Raft and NIAGARA are permitted to utilize all available cores on a server. To investigate a typical use case of Raft in practice, we implement a multi-threaded key-value store that can be remotely accessed by clients via HTTP and offers read and write operations. The workload in this experiment consists of data chunks with 8-byte keys and 2-kilobyte values. With both evaluated systems processing weakly consistent reads entirely within the application (see Sections 2.1 and 3.4), our analysis in the following focuses on strongly consistent reads.

Figure 5a shows the maximum throughputs achieved by Raft and NIAGARA for reads that guarantee strong consistency. Such reads are executed by the application, but coordinated by the replication protocol (see Sections 2.1 and 3.4). Due to the coordination messages of Raft and NIAGARA being small, most of the data in this case is exchanged between clients and the application, not between Raft instances on different servers. Consequently, the replication protocol for small and medium workloads does not become a performance-limiting factor, enabling both systems to increase performance when provided with additional network resources. For high workloads, the fact that NIAGARA parallelizes the coordination of strongly consistent reads across

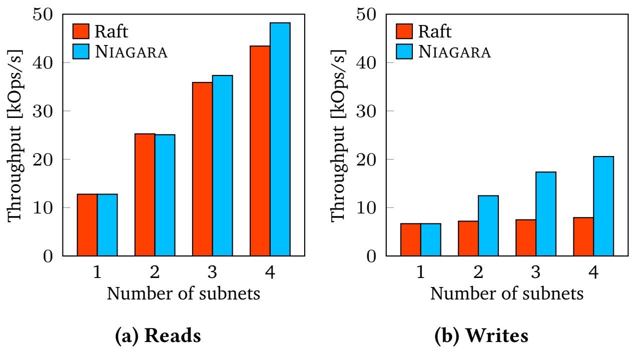


Figure 5: Throughput provided by Raft and NIAGARA for a key-value store offering read and write access.

different Raft instances leads to performance improvements, for example allowing NIAGARA to achieve an 11% higher throughput than Raft when using four Raft instances. In contrast to reads, writes must be appended to the log and therefore are replicated by Raft and NIAGARA. With all writes in Raft flowing through the leader, high write workloads may cause the leader's network link to be filled to capacity. As Figure 5b confirms, adding links under such circumstances in Raft allows the application to distribute the transmission of the (comparably small) replies to clients, but otherwise has no significant effect on throughput. Raft instances in NIAGARA, on the other hand, are able to benefit from their own network links, enabling NIAGARA with four instances to handle about 2.6 times more writes per second than Raft.

4.4 Discussion

Our evaluation has shown that the etcd implementation of Raft we have used for the experiments does not scale with the computing and network resources available. This is not a problem of this particular implementation, but a direct consequence of Raft's strong-leader design principle. Technically, it would probably be possible to modify existing Raft implementations to mitigate at least some of the effects observed in our experiments, for example by maintaining multiple network connections between the leader and each of its followers. However, our experiences with such efforts in the context of the BFT-SMaRt replication library [5], which implements a different consensus algorithm, have shown that this approach (1) requires non-trivial changes to the replication protocol, (2) introduces additional synchronization points, and (3) may lead to throughput increases in some scenarios, but does not offer general scalability [3]. In contrast, the NIAGARA replication architecture represents a way to achieve scalability in Raft-based systems without the need to adapt essential parts of the protocol.

5 RELATED WORK

Several works aimed at improving the performance of Raft-based systems. Howard et al. [11] proposed optimizations to speed up leader election. Arora et al. [1] used quorum reads involving multiple followers to offload work from the leader while still ensuring strong consistency. Both approaches are applicable to NIAGARA. For scalability, CockroachDB [7] divides its state into partitions that each are coordinated by a dedicated Raft instance. In CockroachDB, the Raft instances on a server are tightly coupled and, amongst other things, share the same execution context and heart-beat mechanism. This approach is efficient if a server hosts thousands of partitions, but unlike NIAGARA it requires significant modifications to the original protocol implementation. In NIAGARA, the number of Raft instances is small (e.g., one instance per core) and thus the overhead of heart-beat messages negligible.

Application-state partitioning is also used to achieve scalability in replication protocols besides Raft. In Multi-Ring Paxos [16], a server only participates in the multicast groups that distribute the requests accessing the server's partitions. The Agora coordination service [20] relies on parallel instances of the Zab protocol [12] and synchronizes them via vector clocks that are attached to application requests and replies. In contrast, NIAGARA provides scalability without requiring a partitioned state or extended application messages. That said, for use cases for which state partitioning at application level is beneficial, it is still possible to implement such a scheme on top of NIAGARA, for example, using the global log as input for a dispatcher that assigns commands to partitions.

The Atum middleware [10] offers scalable and resilient group communication based on replica groups whose composition may vary at runtime, for example, due to a new server joining the system. With NIAGARA focusing on effectively utilizing the local resources available on a server, Raft groups in NIAGARA are static and co-located on the same machines.

Various Byzantine fault-tolerant agreement protocols parallelize the consensus process for incoming requests [2–4, 8, 13]. Apart from the fault model—Raft tolerates crashes—a major difference to NIAGARA is the fact that these protocols achieve strong consistency for reads by ordering them in the same way as writes. NIAGARA, on the other hand, relies on the replication protocol for coordinating strongly consistent reads, but the read itself is only performed by a single server.

6 CONCLUSION

The NIAGARA replication architecture makes it possible to build scalable systems that rely on Raft for consensus. In contrast to traditional Raft-based systems, NIAGARA parallelizes the responsibilities for replicating writes and coordinating strongly consistent reads across different Raft instances. As a result, it is able to effectively use both multiple cores as well as multiple network cards on each participating server.

ACKNOWLEDGMENTS

This work was partially supported by the German Research Council (DFG) under grant no. DI 2097/1-2 ("REFIT").

REFERENCES

[1] Vaibhav Arora, Tanuj Mittal, Divyakant Agrawal, Amr El Abbadi, Xun Xue, Zhiyanan, and Zhujianfeng. 2017. Leader or Majority: Why Have One When You Can Have Both? Improving Read Scalability in Raft-like Consensus Protocols. In *Proc. of the 9th Conference on Hot Topics in Cloud Computing (HotCloud '17)*.

[2] Johannes Behl, Tobias Distler, and Rüdiger Kapitza. 2014. Scalable BFT for Multi-Cores: Actor-based Decomposition and Consensus-oriented Parallelization. In *Proc. of the 10th Workshop on Hot Topics in System Dependability (HotDep '14)*.

[3] Johannes Behl, Tobias Distler, and Rüdiger Kapitza. 2015. Consensus-Oriented Parallelization: How to Earn Your First Million. In *Proc. of the 16th Middleware Conference (Middleware '15)*.

[4] Johannes Behl, Tobias Distler, and Rüdiger Kapitza. 2017. Hybrids on Steroids: SGX-Based High Performance BFT. In *Proc. of the 12th European Conference on Computer Systems (EuroSys '17)*.

[5] Alysson Bessani, João Sousa, and Eduardo E. P. Alchieri. 2014. State Machine Replication for the Masses with BFT-SMaRt. In *Proc. of the 44th Int'l Conference on Dependable Systems and Networks (DSN '14)*.

[6] Consul. 2019. https://www.consul.io/.

[7] Ben Darnell. 2015. Scaling Raft. https://www.cockroachlabs.com/blog/scaling-raft/.

[8] Michael Eischer and Tobias Distler. 2019. Scalable Byzantine Fault-tolerant State-Machine Replication on Heterogeneous Servers. *Computing* 101, 2 (2019).

[9] etcd. 2019. https://coreos.com/etcd/.

[10] Rachid Guerraoui, Anne-Marie Kermarrec, Matej Pavlovic, and Dragos-Adrian Seredinschi. 2016. Atum: Scalable Group Communication Using Volatile Groups. In *Proc. of the 17th Int'l Middleware Conference (Middleware '16)*.

[11] Heidi Howard, Malte Schwarzkopf, Anil Madhavapeddy, and Jon Crowcroft. 2015. Raft Refloated: Do We Have Consensus? *SIGOPS Operating Systems Review* 49, 1 (2015).

[12] Flavio P. Junqueira, Benjamin C. Reed, and Marco Serafini. 2011. Zab: High-Performance Broadcast for Primary-Backup Systems. In *Proc. of 41st Int'l Conference on Dependable Systems and Networks (DSN '11)*.

[13] Bijun Li, Wenbo Xu, Muhammad Zeeshan Abid, Tobias Distler, and Rüdiger Kapitza. 2016. SAREK: Optimistic Parallel Ordering in Byzantine Fault Tolerance. In *Proc. of the 12th European Dependable Computing Conference (EDCC '16)*.

[14] Linux Manual Page. 2019. taskset(1). http://man7.org/linux/man-pages/man1/taskset.1.html.

[15] Linux Manual Page. 2019. tc(8). http://man7.org/linux/man-pages/man8/tc.8.html.

[16] Parisa Jalili Marandi, Marco Primi, and Fernando Pedone. 2012. Multi-Ring Paxos. In *Proc. of the 42nd Int'l Conference on Dependable Systems and Networks (DSN '12)*.

[17] Diego Ongaro. 2014. *Consensus: Bridging Theory and Practice.* Ph.D. Dissertation. Stanford University.

[18] Diego Ongaro and John Ousterhout. 2014. In Search of an Understandable Consensus Algorithm. In *Proc. of the 2014 USENIX Annual Technical Conference (ATC '14)*.

[19] RabbitMQ. 2019. https://www.rabbitmq.com/.

[20] Rainer Schiekofer, Johannes Behl, and Tobias Distler. 2017. Agora: A Dependable High-Performance Coordination Service for Multi-Cores. In *Proc. of the 47th Int'l Conference on Dependable Systems and Networks (DSN '17)*.