IEICE TRANS. INF. & SYST., VOL.E106–D, NO.11 NOVEMBER 2023

LETTER

Switch-Based Quorum Coordination for Low Tail Latency in Replicated Storage

Gyuyeong KIM[†a)], ***Nonmember***

SUMMARY Modern distributed storage requires microsecond-scale tail latency, but the current coordinator-based quorum coordination causes a burdensome latency overhead. This paper presents Archon, a new quorum coordination architecture that supports low tail latency for microsecond-scale replicated storage. The key idea of Archon is to perform the quorum coordination in the network switch by leveraging the flexibility and capability of emerging programmable switch ASICs. Our in-network quorum coordination is based on the observation that the modern programmable switch provides nanosecond-scale processing delay and high flexibility simultaneously. To realize the idea, we design a custom switch data plane. We implement a Archon prototype on an Intel Tofino switch and conduct a series of testbed experiments. Our experimental results show that Archon can provide lower tail latency than the coordinator-based solution.

key words: *in-network computing, programmable switches, latency*

1. Introduction

Microservice applications like distributed data stores supported by NoSQL key-value stores [1]–[3] are the fundamental building block of modern online services. They interact via Remote Procedure Calls (RPCs). The runtime of RPCs is getting smaller, and most of the RPCs last only tens to hundreds of microseconds [4], [5]. Therefore, the underlying infrastructure is expected to provide microsecond-scale tail latency to meet strict Service Level Objectives (SLOs). Meanwhile, object data is typically replicated with multiple servers for high availability [6]. Replicated storage often uses quorum consistency (e.g., Cassandra [7], Dynamo [8]) to achieve the balance between availability and performance. In quorum consistency, data reads and writes require agreement from a quorum of replicas.

Requests in quorum-based replicated storage are handled by a coordinator node, which lies between the client and storage replicas. The coordinator propagates requests to replicas and aggregates the replies to commit the request. Specifically, a client first sends a request to the coordinator. For read requests, the coordinator propagates the request to a subset of replicas. The size of this subset called the quorum (R) is determined by the replication factor (F). In a typical quorum-based replicated storage system, R is set to $F/2 + 1$, meaning that a majority of replicas must be queried to satisfy the quorum. For write requests, the coordinator sends the request to all replicas. The coordinator commits the read/write request when the number of aggregated replies is equal to R.

Unfortunately, with the current coordinator-based quorum coordination, it is hard to achieve microsecond-scale tail latency. This is because the coordinator is a commodity server with a CPU, incurring tens of microseconds of latency overhead to coordinate requests even with optimized networking techniques like kernel bypass. Since the `get`/`put` operation only takes tens of microseconds, the latency overhead caused by the coordinator is crucial to overall request latency. In this context, we ask the following question: *how can we achieve microsecond-scale tail latency in quorum-based replicated storage?*

We answer the question by presenting Archon, a new quorum coordination architecture, which provides fast quorum coordination for replicated storage. Our key idea is to leverage the network switch as an in-network quorum coordinator. Specifically, we directly request propagation and reply aggregation to the network switch. This idea is based on the following observations. First, switches are highly optimized for I/O operations. The switch can process a single packet within hundreds of nanoseconds, and its processing throughput is a few billion packets per second. Second, unlike traditional fixed-function switch ASICs, emerging programmable switch ASICs like Intel Tofino [9] allow us to define a custom packet processing pipeline. This means that we have the opportunity to coordinate quorum requests at a nanosecond-scale. While there exist in-network solutions for replicated storage [10], [11], they do not consider quorum consistency.

Unfortunately, realizing the idea is challenging because the programmable switch has strict hardware constraints. Therefore, we address technical challenges when designing a custom switch data plane. Our switch data plane consists of two modules. The first is the request propagation module, which forwards requests to replicas. The second one is the reply aggregation module. This module aggregates replies and commits the request only if a majority of replicas acknowledged the request. We implement a Archon prototype on an Intel Tofino switch and conduct a series of experiments. Our experimental results demonstrate that Archon can achieve higher throughput and lower latency than the coordinator-based quorum coordination.

Manuscript received June 30, 2023.
Manuscript publicized August 22, 2023.
[†]The author is with the Department of Computer Engineering, Sungshin Women's University, Seoul, Republic of Korea.
a) E-mail: gykim@sungshin.ac.kr (Corresponding author)
DOI: 10.1587/transinf.2023EDL8038

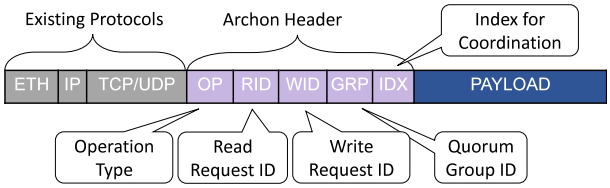


Fig. 1 Archon packet format.

Algorithm 1 Request Processing in Switch Data Plane

```
      – pkt: Packet to be processed
      – ReqR: Register array for pending reads
      – ReqW: Register array for pending writes
      – rid: Register for read request ID
      – wid: Register for write request ID
 1: if pkt.type == READ then
 2:     rid ← rid + 1                          ▷ Increase request ID
 3:     pkt.rid ← rid                          ▷ Assign request ID
 4:     if ReqR[pkt.rid] == EMPTY then
 5:         ReqR[pkt.rid] ← OCCUPIED           ▷ Lock the slot
 6:         Propagate(pkt.grp, pkt.idx)        ▷ Forward to a quorum
 7:     else
 8:         Drop(pkt)                          ▷ Drop if occupied
 9:     end if
10: else if pkt.type == WRITE then
11:     wid ← wid + 1                          ▷ Increase request ID
12:     pkt.wid ← wid                          ▷ Assign request ID
13:     if ReqW[pkt.wid] == EMPTY then
14:         ReqW[pkt.wid] ← OCCUPIED           ▷ Occupy the slot
15:         PropagateAll(pkt.grp, pkt.idx)     ▷ Broadcast request
16:     else
17:         Drop(pkt)
18:     end if
19: end if
```

2. Archon Design

2.1 Packet Format

Figure 1 shows the packet format of Archon. Archon uses a custom L7 protocol message. We reserve a L4 port number for Archon so that the switch can apply different parser logic for Archon packets and normal packets. The Archon message has the header consisting of the following fields.

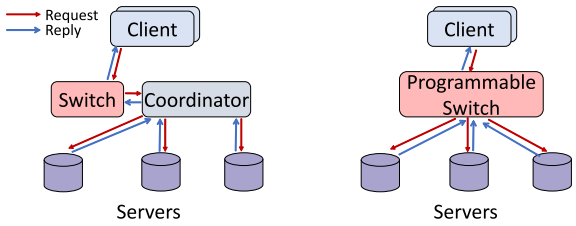
- `OP`: the operation type, which can be `READ`, `WRITE`, `R-REPLY`, and `W-REPLY`.
- `RID`: monotonically-increasing sequence numbers for read request IDs, which are assigned by the switch.
- `WID`: monotonically-increasing sequence numbers for write request IDs, which are assigned by the switch. The value of this field is the version of data.
- `GRP`: the quorum group ID, which identifies a group of replicas that meets the quorum.
- `IDX`: the index for request propagation.

2.2 Request Processing

The Archon data plane consists of the request propagation module and the reply aggregation module. Algorithm 1 is the pseudocode of request processing in the switch data plane.

Read requests. Upon receiving a read request, the switch increases the request ID and assigns the ID to the request (lines 1-3). After that, the switch inserts the request into the list of pending reads by trying to lock the register slot. If the slot is empty, the switch locks the slot and propagates the request to a quorum of replicas by referring to the `GRP` and `IDX` fields (lines 4-6). Locking the slot is to prevent overwriting by possible duplicate IDs. Otherwise, the switch simply drops the request (lines 7-8).

Write requests. When receiving a write, the switch assigns the request ID after increasing it by one (lines 10-12).



(a) Coordinator-based solution (b) Archon

Fig. 2 Read and write processing in Archon. Unlike the existing coordinator-based solution, request propagation and reply aggregation are done by the network switch directly, improving tail latency.

Algorithm 2 Reply Processing in Switch Data Plane

```
      – AggR: Register array for number of aggregated read replies
      – AggW: Register array for number of aggregated write replies
      – Ver: Register array for the version of returned values
      – Candi: Register array for candidate values
      – QUORUM: Configured quorum number
 1: if pkt.type == R-REPLY then
 2:     if pkt.wid > Ver[pkt.rid] then          ▷ More recent data?
 3:         Candi[pkt.rid] ← pkt.value
 4:         Ver[pkt.rid] ← pkt.wid
 5:     end if
 6:     AggR[pkt.rid] ← AggR[pkt.rid] + 1
 7:     if AggR[pkt.rid] == QUORUM then
 8:         pkt.value ← Candi[pkt.rid]
 9:         AggR[pkt.rid] ← 0
10:         ReqR[pkt.rid] ← EMPTY               ▷ Free slot
11:         Forward(pkt)                        ▷ Commit read
12:     else
13:         Drop(pkt)
14:     end if
15: else if pkt.type == W-REPLY then
16:     if ReqW[pkt.wid] == OCCUPIED then
17:         AggW[pkt.wid] ← AggW[pkt.wid] + 1
18:         if AggW[pkt.wid] == QUORUM then
19:             AggW[pkt.wid] ← 0
20:             ReqW[pkt.wid] ← EMPTY           ▷ Free slot
21:             Forward(pkt)                    ▷ Commit write
22:         else
23:             Drop(pkt)                       ▷ Need more replies
24:         end if
25:     else
26:         Drop(pkt)          ▷ Drop reply since already commited
27:     end if
28: end if
```

The request ID acts as the version of the object as well. The switch then inserts the request ID into the pending request list for reply aggregation if the matched slot is empty (lines 13-14). After that, the switch propagates writes to replicas (line 15). Otherwise, the request is dropped (lines 16-17).

2.3 Reply Processing

Algorithm 2 is the pseudocode of reply processing in the switch data plane.

Read replies. When the switch receives read replies, it first compares the version of the data in the `WID` field of the packet and the that of stored one in the switch. If the arriving packet has more recent data, the switch updates the stored object data and the version for the object (lines 1-5). With this, the switch can maintain the data, which is considered the most recent data. After that, the switch increases the number of aggregated replies for the requested object by one (line 6). If the number of aggregated replies is equal to the quorum of replicas, the switch commits the read request by clearing related metadata and forwarding the packet to the client (lines 7-11). Otherwise, the packet is simply dropped since it needs more replies to be committed (lines 12-13).

Write replies. Upon receiving a write reply, the switch first checks whether the matched slot is occupied (line 16). This is because, unlike reads, the write is committed if the quorum is satisfied regardless of the remaining replies. If occupied, the switch increases the number of aggregated replies by one (line 17). If the condition to commit the write is met, the switch forwards the reply to the client after clearing related data slots (lines 18-21). Otherwise, the switch drops the packet (lines 22-23). If the slot is not occupied, the switch also drops the packet (lines 25-26).

3. Data Plane Implementation

We now describe our data plane implementation. Programmable switch ASICs like Intel Tofino [9] provide flexibility but have limited on-chip resources, which include match-action stages, stateful memory, and operation types. This is because it is not designed to perform compute/memory-intensive jobs. Therefore, implementing quorum coordination in the switch data plane imposes several technical challenges, and we design a custom switch data plane using various techniques like recirculation and hashing as follows.

Our data plane is written in P4 [12] and is compiled to P4 SDE 9.7.0. Archon consumes only 5 match-action stages, 7.19% match input bar, 7.98% hash bits, 11.25% gateway, 15.50% SRAM, and 20.00% ALU. Our resource usage does not exceed the available resource budget of the programmable switch ASIC, hence the switch can preserve line-rate throughput.

3.1 Request Propagation

To propagate requests, we need to replicate requests inside the switch. To implement this, we leverage the multicast function that the switch clones packets to multiple designated output ports. However, since the multicast function is performed at the link-layer level, the switch does not update the destination IP address of each packet. This results in the packet drop by the network stack of storage replicas because the destination IP address is not equal to that of the replica.

To address this, we leverage packet recirculation, which forwards the packet to the loopback input port for further packet processing. We explain this in detail. The client sets the `GRP` field to a random number, which identifies the preconfigured quorum of replicas. For read requests, the `IDX` field is set to the quorum number. In the case of write requests, we use the number of replicas for the `IDX` field. The switch decreases the `IDX` field by one every packet. If the `IDX` field is zero, the propagation is finished. When propagating a request, the switch forwards the original packet to the matched output port and recirculates the replicated packet for further processing. The switch performs the same processing for replicated packets until the `IDX` field reaches to 0.

3.2 Reply Aggregation

We need to maintain the list of pending requests and the list of the number of aggregated replies for each request. These lists can be implemented using register arrays. However, the current switch architecture allocates a small portion of on-chip memory to each match-action stage statically. Therefore, we can use only a limited number of register slots while the request ID can be larger than the number of register slots.

To address this, we use hashing for the slot index. Specifically, we use a hash of the `RID`/`WID` field as an index key, which guarantees that the index does not exceed the number of register slots. Each register array for the lists has 128K slots, which are enough to serve millions of requests per second.

4. Performance Evaluation

4.1 Methodology

Testbed setup. To evaluate Archon, we use a cluster consisting of 8 commodity servers, which are connected by an APS Networks BF6064X-T switch. The switch data plane is based on a 6.5 Tbps Intel Tofino switch ASIC [9]. The servers are equipped with a Intel i5-12600K @ 3.7 Ghz, 32 GB of DDR5 memory, and a single-port 100GbE RDMA-capable NIC. The servers run Ubuntu 20.04 LTS with Linux kernel 5.15.0. Unless specified, 2 servers act as clients to generate requests, 5 servers are used as worker servers, and 1 server acts as the quorum coordinator node. Therefore, our replication factor is 5 and the quorum number is 3.

Applications and compared scheme For evaluation, we implement an open-loop application in C with NVIDIA Messaging Accelerator library (VMA) [13], which provides kernel-bypass networking, minimizing the host-side packet

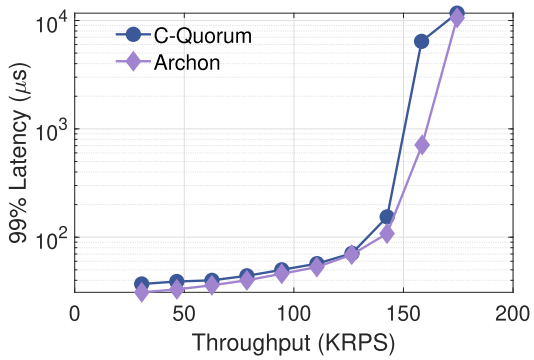


Fig. 3 Main result.

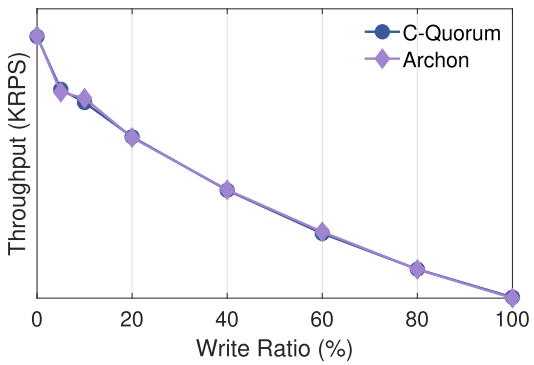


Fig. 4 Impact of write ratio on throughput.

processing delay. We use Redis [1], a popular key-value store. We use a read-heavy workload with 1M 32-bit objects 5% of write ratio, which is a typical production-like workload [10]. We compare Archon with the coordinator-based quorum coordination scheme, which we call C-Quorum.

4.2 Results

Throughput vs. latency. Figure 3 shows the 99th percentile tail latency as throughput grows. We can see that Archon generally provides lower latency than C-Quorum. This is because, in Archon, quorum messages do not visit additional nodes (i.e., quorum coordinator). As throughput grows, the gap between C-Quorum and Archon goes large because the coordinator node becomes the performance bottleneck. These results demonstrate that Archon can reduce the tail latency of quorum-based replicated storage.

Impact of write ratio. Figure 4 is the throughput with different write ratios. C-Quorum and Archon do not have distinct differences. This is because Archon does not increase server loads and requires only a few nanosecond processing delays to add quorum coordination functionality in the switch. Furthermore, though omitted, the tail latency is improved across the given write ratios.

5. Conclusion

This paper presented Archon, an in-network quorum coordination architecture for replicated storage. Archon provides low tail latency by directly performing request propagation and reply aggregation in the emerging programmable switch. We have implemented a Archon prototype and conducted testbed experiments. The experimental results demonstrated that Archon provides better performance than the existing coordinator-based quorum coordination.

Acknowledgements

This work was supported by the Sungshin Women's University Research Grant of 2023.

References

[1] "Redis key-value store," https://redis.io/
[2] B. Fitzpatrick, "Distributed caching with memcached," Linux J., vol.2004, no.124, p.5, Aug. 2004.
[3] "Rocksdb: A persistent key-value store for flash and ram storage," https://rocksdb.org/
[4] J. Zhao, I. Uwizeyimana, K. Ganesan, M.C. Jeffrey, and N.E. Jerger, "Altocumulus: Scalable scheduling for nanosecond-scale remote procedure calls," Proc. IEEE/ACM MICRO, pp.423–440, 2022.
[5] K. Kaffes, T. Chong, J.T. Humphries, A. Belay, D. Mazières, and C. Kozyrakis, "Shinjuku: Preemptive scheduling for μsecond-scale tail latency," Proc. USENIX NSDI, Boston, MA, pp.345–359, Feb. 2019.
[6] P. Hunt, M. Konar, F.P. Junqueira, and B. Reed, "Zookeeper: Wait-free coordination for internet-scale systems," Proc. USENIX ATC, USA, p.11, 2010.
[7] "Apache cassandra," https://cassandra.apache.org/
[8] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, "Dynamo: Amazon's highly available key-value store," Proc. ACM SOSP, New York, NY, USA, vol.41, no.6, pp.205–220, 2007.
[9] "Intel tofino programmable ethernet switch," https://github.com/barefootnetworks/Open-Tofino
[10] G. Kim and W. Lee, "In-network leaderless replication for distributed data stores," Proc. VLDB Endow., vol.15, no.7, pp.1337–1349, March 2022.
[11] G. Kim, "Netclone: Fast, scalable, and dynamic request cloning for microsecond-scale rpcs," Proc. ACM SIGCOMM, Sept. 2023.
[12] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, and D. Walker, "P4: Programming protocol-independent packet processors," SIGCOMM Comput. Commun. Rev., vol.44, no.3, pp.87–95, July 2014.
[13] "Nvidia messaging accelerator (vma)," https://docs.nvidia.com/networking/spaces/viewspace.action?key=VMAv940