NetClone: Fast, Scalable, and Dynamic Request Cloning for Microsecond-Scale RPCs

Gyuyeong Kim
Sungshin Women's University
South Korea
gykim@sungshin.ac.kr

ABSTRACT

Spawning duplicate requests, called cloning, is a powerful technique to reduce tail latency by masking service-time variability. However, traditional client-based cloning is static and harmful to performance under high load, while a recent coordinator-based approach is slow and not scalable. Both approaches are insufficient to serve modern microsecond-scale Remote Procedure Calls (RPCs). To this end, we present NetClone, a request cloning system that performs cloning decisions dynamically within nanoseconds at scale. Rather than the client or the coordinator, NetClone performs request cloning in the network switch by leveraging the capability of programmable switch ASICs. Specifically, NetClone replicates requests based on server states and blocks redundant responses using request fingerprints in the switch data plane. To realize the idea while satisfying the strict hardware constraints, we address several technical challenges when designing a custom switch data plane. NetClone can be integrated with emerging in-network request schedulers like RackSched. We implement a NetClone prototype with an Intel Tofino switch and a cluster of commodity servers. Our experimental results show that NetClone can improve the tail latency of microsecond-scale RPCs for synthetic and real-world application workloads and is robust to various system conditions.

CCS CONCEPTS

• **Networks** → **Programmable networks**; **In-network processing**; • **Hardware** → **Networking hardware**.

KEYWORDS

Programmable switches, in-network computing, microsecond-scale RPCs, tail latency

ACM Reference Format:
Gyuyeong Kim. 2023. NetClone: Fast, Scalable, and Dynamic Request Cloning for Microsecond-Scale RPCs. In *ACM SIGCOMM 2023 Conference (ACM SIGCOMM '23), September 10–14, 2023, New York, NY, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3603269.3604820

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ACM SIGCOMM '23, September 10–14, 2023, New York, NY, USA
© 2023 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 979-8-4007-0236-5/23/09…$15.00
https://doi.org/10.1145/3603269.3604820

1 INTRODUCTION

Today's online services are made up of multiple microservices that communicate with each other using Remote Procedure Calls (RPCs), allowing access to functions and data as if they were local [11, 42]. These services often have strict Service Level Objectives (SLOs) that require underlying data center systems to provide high throughput with *microsecond-scale* tail latency [6, 11, 14, 24, 35]. This is because RPCs are getting smaller, and their runtime is generally an order of microseconds [24, 26, 42]. Unfortunately, RPC requests often experience excessive tail latency even if the request is the same [14, 38]. One of the causes is unexpected variability in service times, which stems from various factors (e.g., load fluctuation, background tasks, interference among applications, and garbage collection [5, 13, 17, 38]).

Request cloning is a powerful technique to mask service-time variability. The traditional client-based cloning always sends redundant requests (typically 2 [17, 38]) to multiple servers and only accepts the faster response. Owing to its simplicity and efficiency, the cloning technique has been employed in various domains [5, 17, 18, 38–40]. One limitation is that it does not always result in improved performance. The latency is improved only within a sweet spot, and the system performance is rather degraded beyond a certain threshold load [38, 39]. This is not surprising because redundant requests add extra load to servers. Redundancy also doubles the packet processing overhead for clients, reducing the performance gain [39].

A recent solution [38] addresses the limitation by using a centralized coordinator, which dynamically clones requests only if at least two servers are idle. Thanks to dynamic cloning, the performance is not degraded under high load. However, it is not enough to serve microsecond-scale workloads. This is because the coordinator incurs microseconds of additional latency overhead. It is also hard to scale out as throughput grows because of the limited capability of the coordinator CPU. Therefore, its target workload is millisecond-scale workloads with limited throughput. In this context, we ask the following question: *how can we perform dynamic request cloning quickly at scale for microsecond-scale RPCs?*

As the answer to the question, we present NetClone, a new request cloning system for microsecond-scale RPCs. To serve these workloads with high throughput and low tail latency at the cluster-level, cloning decisions should be made in a nanosecond-scale with scalability. However, achieving this in software is difficult because this is beyond the capability of modern CPUs even with advanced networking like RDMA. For this reason, NetClone performs request cloning in hardware. Specifically, we dynamically clone requests and filter redundant slower responses in the Top-of-Rack (ToR) switch by leveraging the capability of programmable switch

ASICs like Intel Tofino [3]. The switch can process a few billion packets per second, and it takes only hundreds of nanoseconds to process a single packet. Therefore, with dynamic in-network request cloning, we can avoid latency overhead and a potential performance bottleneck, which are caused by the cloning coordinator.

However, transforming the high-level idea into a working system is not straightforward because of various technical challenges as follows. First, we need to know server states (i.e., busy or idle) for cloning decisions. While the existing solution [38] can guarantee the idleness of servers by queueing requests in the coordinator, we cannot directly implement it in the programmable switch because of limited memory space. To address this, we make response packets piggyback the state of the server by lookup the vacancy of the request queue, and the switch stores the state in the switch memory. The switch replicates requests only if two candidate servers are idle. Unfortunately, the actual server state may be different due to the time gap. Therefore, we design a server-side mechanism that drops the cloned request if the actual state is busy.

Second, we need to access the server state table twice to get the state of the candidate servers. However, this is not possible with the current programmable switch ASIC that makes packets go through processing stages sequentially. In particular, it requires two stages to access the state table twice, but the table is statically allocated in the first stage. To overcome this limitation, we put a shadow table in the second stage, a copy of the state table. Similarly, we cannot assign the destination IP to the cloned request at the time of cloning. To address this, we recirculate the cloned request by forwarding the clone to a port in loopback mode.

Lastly, we need to block the slower response because it reduces the performance gain by causing unnecessary packet processing in the client. The challenge here is that memory footprint and hash collisions should be minimized. To address this, we make a filter table using the hash index, which can be reused by multiple requests. For the faster response of a request, the switch puts its request ID in the filter table as a fingerprint. In contrast, the switch drops the slower response of the request if the table slot contains the same request ID, since the switch knows that the faster response is already processed. To handle hash collisions, we use multiple filter tables with randomized table indices for requests.

NetClone is in line with emerging in-network computing solutions [22, 30, 41, 43, 44]. We believe that a tier of coordinators like load balancers between clients and servers should be integrated into the network switch because we can eliminate performance overhead and save costs of hardware and software required to build and maintain coordinator nodes as well. In this context, NetClone is a further advance to realize the vision of in-network computing, not just another case to show the benefit of in-network acceleration. To demonstrate this, we show that NetClone can be integrated with RackSched [44], a recent in-network request scheduler.

We implement a prototype of NetClone on an Intel Tofino switch. NetClone consumes 4.77% of the switch memory because we store small soft states in switch memory, which are generally server state information and request IDs in the filter table. To evaluate NetClone, we build a testbed consisting of 8 commodity servers and a 6.5Tbps Intel Tofino programmable switch. We conduct a series of extensive experiments with a combination of synthetic Redis [4] and Memcached [15] workloads. Our key findings include: 1) NetClone can provide lower tail latency compared to the baseline, and has higher throughput than the client-based cloning and LÆDGE [38], the state-of-the-art coordinator-based cloning solution; 2) NetClone can make synergy with RackSched [44] for various workload conditions; 3) NetClone is robust to system conditions.

In summary, this work makes the following contributions.

- We propose NetClone, a request cloning system that provides dynamic, scalable, and fast request cloning and redundant response filtering to reduce the tail latency for modern microsecond-scale RPC workloads. NetClone shows that programmable switches are a vantage point that can be used to accelerate applications with microsecond-scale latencies.
- We address various technical challenges to design a custom switch data plane that clones requests, filters redundant responses, and tracks server states within the strict hardware constraints of switch ASICs.
- We implement a NetClone prototype with a commodity programmable switch and conduct a series of extensive testbed experiments to demonstrate the efficiency and robustness of NetClone.

The remainder of the paper is organized as follows. In Section 2, we describe the motivation of this work. Section 3 provides the design of NetClone. We present implementation and evaluation results in Section 4 and Section 5, respectively. We discuss related work in Section 6. Lastly, we conclude our work in Section 7.

2 BACKGROUND AND MOTIVATION

In this section, we provide background on request cloning and motivate the necessity of in-network request cloning for microsecond-scale RPCs.

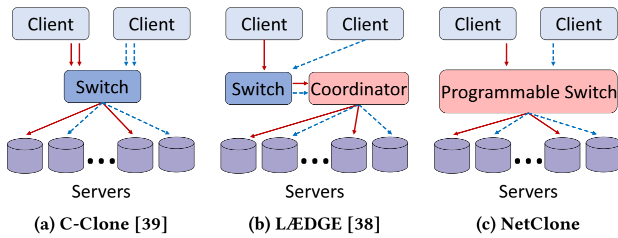
2.1 Latency Variability in RPCs

Modern online services consist of a set of microservices, and the interaction between the microservice applications is often done by RPCs [11, 25]. To guarantee good user experience, online services have strict SLOs, which are generally expressed as tail latency. The runtime of RPCs is typically short as a few to tens of microseconds [24, 42]. Therefore, data center systems that host the services are expected to provide low tail latency with high throughput.

Unfortunately, variability in service times makes it challenging to ensure low tail latency. The processing latency of requests in a server is stochastic and sometimes can be 15 times larger than the median latency [38]. Various factors contribute to the service-time variability, which include load fluctuation, interrupts, garbage collection, background tasks, OS scheduling, power management, and so on [5, 13, 14, 27, 38–40]. Therefore, the service time of RPCs typically follows a heavy-tailed distribution [11, 13, 14], which may violate the SLO of the services.

2.2 Cloning for Microsecond-Scale RPCs

One efficient technique to mask service-time variability is request cloning. The idea is simple as follows. The client sends multiple copies of a request to different servers and takes the fastest response. Optionally, the client may cancel unfinished slower requests. Recent results show that two clones are enough, and canceling



(a) C-Clone [39] (b) LÆDGE [38] (c) NetClone

Figure 1: Different approaches for requeset cloning.

slower requests does not bring meaningful benefits [38]. Owing to its simplicity and efficiency, cloning has been adopted in various works over the past decade [5, 17, 18, 38–40]. There are two approaches for cloning. One is traditional client-based cloning [39], which we call C-Clone in short, and the other is coordinator-based cloning [38]. Unfortunately, these approaches are not enough to serve microsecond-scale RPC workloads.

Client-based cloning (C-Clone). With this, clients perform request cloning in a distributed manner [39] as illustrated in Figure 1 (a). The client typically sends two duplicate requests to servers. One limitation is that cloning is only beneficial within a specific load range. The latency is degraded significantly after a tipping point, which typically lies between 25% and 50% of the load. This is due to the static and load-agnostic cloning of the client, as it always sends duplicate requests regardless of system load. This static cloning also degrades maximum throughput by half as server loads become double.

Coordinator-based cloning. This approach uses a coordinator node to perform request cloning in a centralized manner as shown in Figure 1 (b). LÆDGE [38] is the state-of-the-art coordinator-based solution. Unlike C-Clone, LÆDGE is dynamic and load-aware. The coordinator only replicates requests if at least two servers are idle. If only one server is available, the request is forwarded without replication. In the case where all servers are busy, the coordinator enqueues the request in a request queue and waits for an idle server. The buffered request is dispatched to a server upon receiving a response.

Unfortunately, this is still far from a solution for microsecond-scale RPCs. The cloning decision needs to be as fast as possible since RPCs want to be processed as if they are local functions. However, it takes an order of microseconds to perform request cloning in the coordinator. Because of this, LÆDGE targets millisecond-scale workloads, which can tolerate the latency overhead.

The other limitation is that the coordinator is not scalable because it relies on the CPU to handle requests. Unfortunately, the CPU has inherently limited performance even with kernel-bypass networking like RDMA, which can reduce CPU usage for packet processing. Therefore, the coordinator can be a performance bottleneck easily and provide limited throughput for only a few servers. Furthermore, the LÆDGE coordinator should process redundant slower responses to dispatch another request, making throughput worse. It is possible to use multiple coordinators to scale out. However, this causes burdensome costs to build and maintain a tier of coordinators.

Table 1: Comparison to existing works.

	C-Clone [39]	LÆDGE [38]	NetClone
Cloning point	Client	Coordinator	Switch
Dynamic cloning	×	√	√
Scalability	√	×	√
High throughput	×	×	√
Low latency overhead	√	×	√

2.3 The Case for In-Network Cloning

Design goal and key idea. Our goal is to perform request cloning dynamically and quickly at scale for microsecond-scale RPCs. The key idea to achieve the goal is to perform cloning decisions in the switch by leveraging the capability of programmable switch ASICs like Intel Tofino [3] and Cavium Xpliant [1]. We can achieve high performance using the switch since it is optimized for packet processing. In particular, a switch can process a few billion packets per second, whereas a commodity server can handle a few million packets per second. Furthermore, the per-packet processing delay is guaranteed in hundreds of nanoseconds. Therefore, we propose NetClone, an in-network dynamic request cloning system as shown Figure 1 (c).

Comparison to existing works. Table 1 summarizes the difference between NetClone and the existing solutions. C-Clone can scale out to multiple servers and does not incur excessive latency overhead for cloning decisions. However, as it statically replicates requests regardless of system load, throughput is limited. Despite dynamic cloning, LÆDGE does not provide scalability, high throughput, and low latency overhead as it uses a server-based cloning coordinator. Unlike the existing works, NetClone can clone requests dynamically at scale with high throughput and a nanosecond-scale latency overhead as cloning is performed in the network switch.

Challenges. Designing an in-network cloning system does not mean merely implementing the existing dynamic cloning mechanism on the network switch. This is because the switch has strict resource constraints and timing requirements. When designing a custom switch data plane, we should address several technical challenges as follows.

- The switch contains only 10-20MB of limited memory. This implies that we cannot queue requests in the switch memory as LÆDGE does, and we need a new mechanism to check whether servers are idle or busy.
- It is impossible to access data stored in the memory twice for a single pass because each data is statically allocated to a specific stage at compile time. This means that it is challenging to check the state of two candidate servers for cloning decisions.
- In a similar vein, we should carefully design a mechanism to filter redundant slower responses while minimizing memory footprints.

3 NETCLONE DESIGN

3.1 NetClone Architecture

As illustrated in Figure 2, the NetClone architecture consists of the switch data plane, clients, and servers.

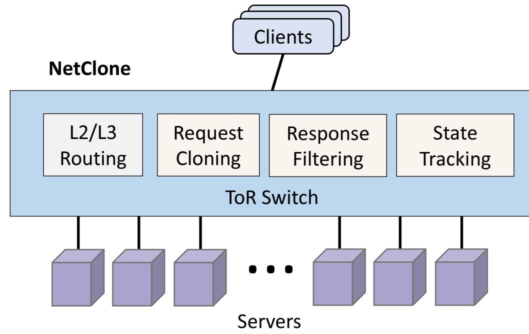


Figure 2: NetClone system architecture.

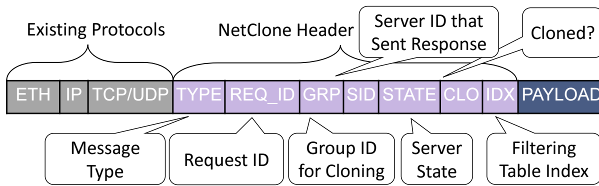


Figure 3: NetClone packet format.

Switch data plane. The core of the NetClone architecture is the switch data plane. We design three custom modules, which are triggered only for NetClone packets. The request cloning module decides whether to replicate requests based on server states. The response filtering module blocks redundant slower responses using request fingerprints. The state tracking module updates the server states upon receiving responses to track the latest state information. Meanwhile, our switch data plane can perform packet forwarding with the traditional L2/L3 routing module.

Clients and servers. To support NetClone, we need modifications on clients and servers to insert metadata (e.g., server state) into the NetClone header, which resides between the L4 header and the application payload. Note that integrating NetClone with existing RPC frameworks needs careful investigation because it may cause interference between request cloning and existing functionality in the framework.

3.2 Packet Format

Figure 3 shows the packet format of NetClone. The NetClone header is encapsulated as a L4 payload. We reserve an L4 port number for NetClone so that the switch can apply different packet processing logic for NetClone packets and normal packets. Since both NetClone packets and normal packets are forwarded using traditional L3 routing, NetClone is compatible with existing network functions. The NetClone header consists of 7 fields as follows.

- `TYPE`: the message type, which can be `REQ` (a request) and `RESP` (a response).
- `REQ_ID`: the request ID, which is a unique sequence number assigned by the switch.

Algorithm 1 Packet Processing in Data Plane

- pkt: Packet to be processed
- SEQ: Global sequence number for request IDs.
- $GrpT$: Match-action table to get a server pair
- $AddrT$: Match-action table to get IP address
- $StateT$: Register array to track server states
- $ShadowT$: The copy of the state table
- $FilterT$: Register arrays to filter redundant responses

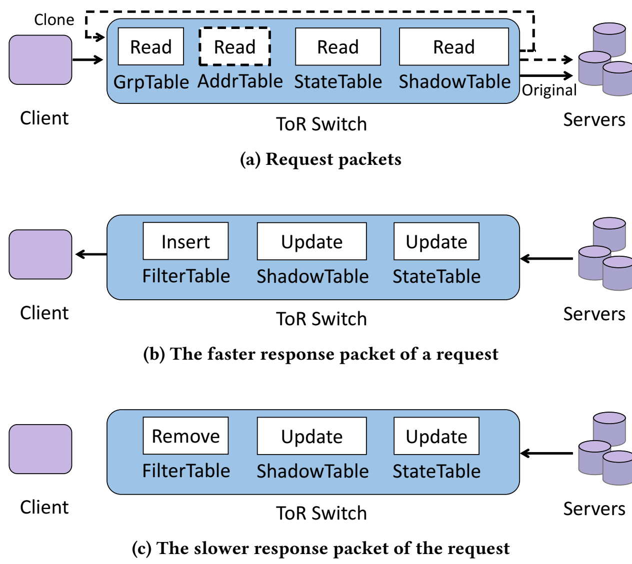
```
1: if pkt.type == REQ and NotCloned then
2:     SEQ ← SEQ + 1
3:     pkt.req_id ← SEQ
4:     Srv1, Srv2 ← GrpT.read(pkt.grp)            ▷ Get server IDs
5:     pkt.dst ← AddrT[Srv1]                       ▷ Get IP addr.
6:     if StateT[Srv1] == IDLE and ShadowT[Srv2] == IDLE then
7:         pkt.clo ← 1                  ▷ Mark as cloned original packet
8:         pkt.sid ← Srv2               ▷ Will be used for forwarding clone
9:         Clone(pkt)                   ▷ Forward pkt and recirculate clone
10:    end if
11: else if pkt.type == REQ and Cloned then
12:    pkt.clo ← 2                                 ▷ Mark as cloned packet
13:    pkt.dst ← AddrT[pkt.sid]                    ▷ Get IP addr.
14: else if pkt.type == REP then
15:    StateT[pkt.sid] ← pkt.state
16:    ShadowT[pkt.sid] ← pkt.state
17:    if pkt.clo > 0 then
18:        Hidx ← Hash(pkt.req_id)                 ▷ Get hash index
19:        if FilterT[pkt.idx][HIdx] == pkt.req_id then
20:            FilterT[pkt.idx][HIdx] ← 0
21:            Drop(pkt)
22:        else
23:            FilterT[pkt.idx][HIdx] ← pkt.req_id
24:        end if
25:    end if
26: end if
27: Forward(pkt)
```

- `GRP`: the group ID that specifies a pair of candidate servers.
- `SID`: the server ID that sent a response. This field is used as the index for the server state table.
- `STATE`: the server state, which can be busy or idle.
- `CLO`: the field to clarify whether the request is cloned or not. 0 means the non-cloned request; 1 means the cloned original request; 2 means cloned request.
- `IDX`: the index for hash tables to filter redundant responses. Note that this is the table index, not the slot index of a table.

3.3 Request Packet Processing

In this subsection, we describe how the switch processes request and response packets. Algorithm 1 describes the high-level pseudocode of request processing in the switch data plane. Figure 4 shows how NetClone handles requests.

Request packets. Clients do not have to know server information since the switch determines the destination server. Clients use a group ID to determine a pair of candidate servers. The group ID is randomly chosen by the client. Each group ID matches two candidate servers, which are predefined by the operator. The number



(a) Request packets

(b) The faster response packet of a request

(c) The slower response packet of the request

Figure 4: Request processing in NetClone.

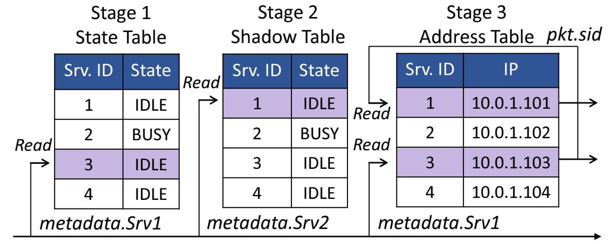


Figure 5: Cloning decisions based on server states.

of groups is $2\binom{n}{2}$ as we choose two servers between n servers. Multiplying by two is to sustain the randomness of server selection because the switch forwards the request to the first candidate server if cloning conditions are not satisfied. For example, assume that we have only two servers. In this case, we have two groups, and each group specifies {Srv1,Srv2} and {Srv2,Srv1}. If we specify only one group (e.g., {Srv1,Srv2}) for this case, all non-cloned requests are forwarded to Srv1.

We use several tables to process requests as follows. The group table *GrpT* is a match-action table that maps from the group ID to the IDs of candidate servers. Since clients do not specify the destination server initially, we also use the address table *AddrT*, a match-action table that assigns a destination IP address to the packet. To track server states, we use two tables, which are the state table *StateT* and the shadow state table *ShadowT*, a copy of *StateT*. The tables contain the state of servers, and the switch performs cloning decisions based on the information.

The switch has different processing logic for original requests and cloned requests. Upon receiving a normal request, the switch assigns a request ID to the request after increasing the sequence number by one (lines 1-3). Next, the switch gets the ID of candidate servers (i.e., *Srv*1 and *Srv*2) by accessing *GrpT* (line 4). After that, the destination IP address is updated using the ID of server 1 as the index for *AddrT* (line 5). The switch now checks whether the tracked server states are both idle or not. This is done by accessing *StateT* and *ShadowT* for server 1 and server 2, respectively (line 6). If positive, the request is marked as cloned but original (line 7). In addition, since we should forward the clone to server 2 as well, we put the ID of server 2 into the `SID` field (line 8). The switch finally clones the request by forwarding the original request to server 1 and recirculating the cloned request into the ingress pipeline (line 9). For recirculated cloned requests, the switch marks the request as cloned by updating the `CLO` field to 2 (lines 11-12). After that, the IP address of server 2 is assigned to the request packet (line 13). Figure 4 (a) outlines the process.

Response packets. When a server sends back a response, the server updates the `SID` and `STATE` fields with its server ID and the current server state. To handle responses, we use three tables: *StateT*, *ShadowT*, and *FilterT*. Between them, *FilterT* is the filter table to block slower responses, which is implemented as a register array. The switch has slightly different logic for the faster response of a request and the slower response of the request, as shown in Figure 4 (b) and (c). Upon receiving a response, the switch first updates the state information of the server in *StateT* and *ShadowT* (lines 14-16). After that, the switch checks whether the response is of a cloned request by lookup the `CLO` field. If positive, the switch data plane gets the hash slot index using the `REQ_ID` field (lines 17-18). If the hash slot of the matched filter table contains the same request ID (i.e., the slower response), the switch clears the slot and drops the packet (lines 19-21). This is because the faster response is already forwarded to the client. Otherwise, for the faster response, the switch puts the value of `REQ_ID` field into the hash slot as a fingerprint to block the slower response (lines 22-23).

3.4 Dynamic Request Cloning

State tracking. LÆDGE [38] dynamically replicates requests only if the candidate servers are idle, which is guaranteed by queueing requests in the coordinator and dispatching only one request at once. However, this cannot be directly implemented in the current generation of programmable switches, as they have limited memory to buffer millions of requests and cannot store complex data structures.

Instead, we track server states and clone requests only if the servers are *considered idle*. We observe that, in general, the request queue in a server is empty if the number of incoming requests is not enough to make the server overloaded. Therefore, if the queue is empty, we can consider the server as idle and is affordable to a cloned request as well. We avoid non-empty queues because the existence of queued requests indicates that the server is too busy to handle incoming requests instantly, leading to performance degradation due to the reverse effect. To deliver the server state to the switch, we make servers piggyback their state in response packets. Upon receiving responses, the switch always updates the state table so that the latest state information can be maintained.

One issue is that the actual server state is not idle when a cloned request visits the server, because there is a time gap between the tracked state and the actual state. Therefore, to avoid this, we make

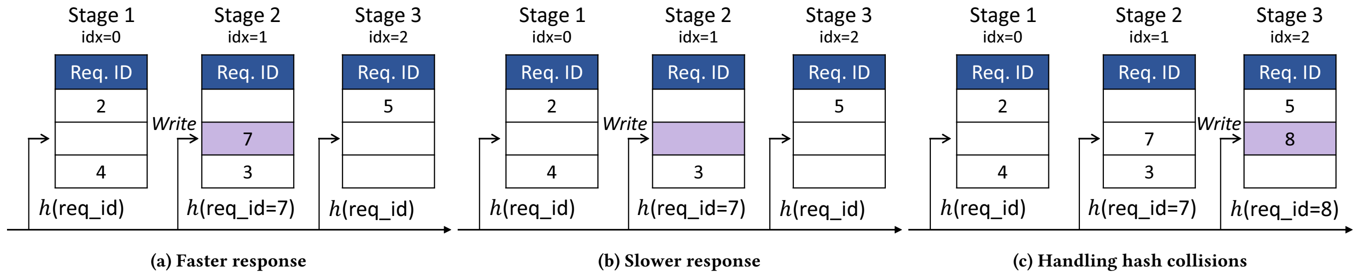


Figure 6: Examples of the filter table operations.

the server drop the packet request if the queue is not empty when receiving a cloned request. It is important to note that only cloned requests (CLO=2) are dropped, while the original request (CLO=1) is processed normally. Another possible solution is to track the server-side throughput and only clone requests when it is below a certain threshold. However, this requires complex performance profiling to determine the threshold.

Shadow state table To decide whether to clone, we must check the state of both candidate servers. This requires the read of the state table twice, but it is not possible with the current switch ASIC using the PISA architecture [9]. This is because, in the architecture, the switch data plane consists of multiple match-action stages and the memory space of the table is statically allocated to a single stage at compile time. A packet passes through the match-action stages in the data plane to preserve a line-rate, which means that a packet can visit the table only once. To overcome this limitation, we put a shadow table, a copy of the state table. This allows the switch to check server states twice indirectly. The consistency between the state table and the shadow table can be preserved since the switch always updates the tables at the same time upon receiving a response.

Cloning in the switch. The commodity programmable switch provides two options to clone packets. One is port mirroring and the other is multicasting. Both of them generate a copy of the original packet and send the clone to a specific output port. NetClone utilizes multicasting since it is simpler in terms of switch configuration. A challenge here is that we cannot assign the destination IP to the clone because the switch currently processes the original request at the time that replicates the request. Therefore, we make the cloned request visit the ingress pipeline again using recirculation. The recirculation is implemented by forwarding the clone to the port in loopback mode. When the clone is recirculated, the switch assigns the destination IP address and forwards it to the corresponding output port.

Example. Figure 5 shows an example of request cloning. In this example, the switch confirms that the servers *Srv*1 and *Srv*2 are idle by accessing the state table and the shadow table. The switch then assigns 10.0.1.103 to the request for the destination IP and forwards the request. At the same time, the switch generates the cloned request for *Srv*2 and recirculates it. When recirculated, the address table assigns 10.0.1.101 to the clone. After that, the switch finishes the process by forwarding the clone.

3.5 Response Filtering with Fingerprinting

For the client, it is redundant to process the slower response of a request, since the client already received the faster response. This overhead degrades the performance gain [39]. Our switch data plane filters redundant responses using request fingerprinting. The idea is simple as follows. We assign a monotonically-increasing sequence number for the request ID to each request, which is shared by the original and the clone. The faster response puts the request ID in the filter table as a fingerprint to let the slower response know that the faster one is already processed. The switch drops the slower response if the table contains the same request ID. Note that the filter table is a register array, not a match-action table.

Minimizing memory usage. A challenge here is how to minimize the memory footprint for response filtering because switch memory is a scarce resource. However, the current switch ASIC does not allow dynamic memory allocation, and the memory space must be allocated at compile time [9, 12]. Reserving memory space as many as possible is not feasible because the sequence number for request IDs (i.e., the number of requests) can be over billions.

To reserve space for filtering while minimizing the memory footprint, we make the filter table use the hash index. Our insight behind the idea is that the request ID only exists until the slower response arrives, which is a few microseconds in common. Therefore, each hash slot can be reused for multiple request IDs. We also allow responses to overwrite the existing request ID in a slot. This is to handle hash collisions and packet drops. If we prohibit the overwrite, responses can be dropped even if the response is the fastest one. In a similar vein, if the slower response of a request is dropped or missed before visiting the switch, the hash slot becomes unavailable permanently. Meanwhile, the overwrite may cause the failure to block the slower response, but it is not often since hash collisions and packet drops are rare because of microsecond-scale latency.

To further minimize hash collisions, we arrange multiple filter tables in the switch data plane. We randomly assign a table index in the IDX field at the client side. Since the IDX field remains consistent for a request and its responses, all related packets access the same table. It is important to note that the index refers to the table index, not the hash slot index. As a result, responses of two different requests with the same hash index can be processed concurrently without collisions, unless they have the same assigned table index.

Example. Figure 6 shows examples of our filter table when we have three tables. Let us consider a request with the request ID $req_id = 7$ and the table index $idx = 1$. As shown in Figure 6 (a), the faster response of the request inserts 7 to the empty hash slot in the second filter table. When the slower response arrives at the switch, the switch resets the hash slot to empty and drops the response as illustrated in Figure 6 (b). We now consider when a hash collision occurs in Figure 6 (c). Although the hash index of the request with $req_id = 8$ collides with the request with $req_id = 7$, we can avoid the overwrite thanks to the different table index.

3.6 Failure Handling

In this section, we describe how NetClone handles failures.

Dropped messages. The loss of requests does not cause a problem as NetClone simply concerns request cloning. Meanwhile, the drop of responses may cause issues. As mentioned in a previous subsection, if the slower response of a request is dropped, the filter table slot will be permanently occupied and unavailable. However, our design allows responses to overwrite the hash slot with a different request ID, thus avoiding this problem.

Server failures. In the event of a server failure, the overall performance will be degraded until the server is either recovered or removed. The switch control plane can quickly remove the failed server from the list of potential destination servers by updating relevant tables (e.g., the group table and the address table) in the switch data plane and the number of groups on the client side.

Switch failures. NetClone does not cause any permanent misbehavior during switch failures as it stores only soft states, such as server states, the global sequence number for request IDs, and the filter table entries. Once the switch is recovered, the server states can be updated through the following responses. The loss of table entries does not lead to any serious consequence, although there may exist temporary overhead on the client side as slower responses can be forwarded. Additionally, while the sequence number restarts from 0, this does not result in any fatal outcome, as most requests with earlier sequence numbers have already been completed.

3.7 Handling Practical Requirements

We now describe how NetClone can support a variety of practical requirements.

Integration with RackSched. RackSched [44] is an in-network request scheduler for microsecond-scale workloads. It performs the Join-the-Shortest-Queue (JSQ) load balancing [10] by utilizing the power of two choices [36] in the switch data plane. NetClone can integrate RackSched into its design to make synergy as follows. First, we change the state table to the load table and store the queue length of request queues in servers instead of binary states. NetClone still can make a cloning decision since we consider the server with the empty queue as idle. If all candidate servers have empty queues, we replicate requests as usual. Otherwise, we fall back to RackSched. The switch compares the queue lengths of the servers and chooses the one with the shortest queue as the destination server. When integrating the two solutions, we address several challenges caused by the computational limits of the switch ASIC, but we omit the detail due to space constraints.

Multi-rack deployment. NetClone generally targets a single-rack model like rack-scale computers, but it is possible to deploy for a multi-rack model like cloud-scale data centers. This is because NetClone leverages the existing forwarding function to route both normal and cloned requests. In multi-rack deployment, aggregation switches do not have to be aware of request cloning and only ToR switches need to use NetClone logic. However, the ToR switch of servers may apply the NetClone logic to packets even if the NetClone processing should be done only in the ToR switch of the client. Therefore, we add a switch ID field to the NetClone header, with an initial value of zero that is set to the pre-defined switch ID when the packet passes through the ToR switch of the client. ToR switches then apply the NetClone logic only to packets with a switch ID field of zero or matching their own ID.

Multiple pipelines and scalability. Modern programmable switches consist of multiple pipelines and each pipeline is connected to a number of ports. For example, for a 64-port switch with 4 pipelines, 16 ports are assigned to each pipeline. Each pipeline basically does not share its table entries, metadata, and registers. Therefore, the solution has limited scalability with a limited number of ports (i.e., the number of servers) if it supports a single pipeline only. NetClone can work with multiple pipelines. For example, NetClone can work between the client with pipeline 0 and the server with pipeline 1. This is because the NetClone mechanism only requires soft states like server states and request IDs to be maintained in a pipeline connected to the client. The entry of match-action tables of all pipelines can be updated by the switch control plane at the same time. In a similar vein, NetClone does not limit the number of supported servers compared to the vanilla switch even with multi-rack environments. Our soft states are updated in the switch data plane at line-rate and do not rely on the switch control plane, which has a limited update throughput.

Multi-packet messages. A microsecond-scale RPC message is generally small and consists of a single packet. For example, DeathStarBench [16] states that 75% of RPC requests are less than 512 bytes in size, while over 90% of RPC responses are smaller than 64 bytes. Therefore, the current NetClone design does not consider multi-packet requests and responses by default. For multi-packet requests, since we assign the group ID at the client, the request affinity is naturally preserved. However, we need a cloned request table that stores the ID of cloned but unfinished requests since every packet of a cloned request should be cloned regardless of system load. To filter multi-packet responses, we can use multiple ordered filter tables and make the server assign a unique table index to each packet of a multi-packet response. The switch then uses the corresponding ordered filter table index to perform the filtering logic. For example, for a 4-packet response, the server could assign table indices from 0 to 3, and the switch would filter each packet using the matching filter table index.

Protocol support. Our design basically considers UDP for the L4 protocol since our focus is on microsecond-scale RPCs, which usually consist of a single packet [16, 42, 44]. However, some RPC applications may choose to use TCP. To support TCP without any unexpected behavior, the request ID assignment logic should be revised. This is because the switch will assign a different request ID for retransmitted requests, which can lead to misbehavior for multi-packet requests. To address this, we use a tuple of the client

ID and a local sequence number generated by the client for request IDs like Lamport clocks [28, 31]. Additionally, we also append the NetClone header to the TCP handshake packet for applying the NetClone logic though they do not contain a payload.

Integration with RPC frameworks. We clarify that integrating NetClone with existing RPC frameworks like eRPC [26] may require a considerable amount of engineering effort because the framework provides various functions and diverse packet transport logic like RDMA, which can cause a functional collision with NetClone. For example, some RPC frameworks can generate multiple packets for a single request even if the request size is small. In this case, NetClone may clone only partial packets of a multi-packet request when the server state information is updated. This does not provide as much performance improvement as full cloning. However, this still provides a degree of improvement since cloned packets of the request see the performance gain. The retransmission of a request is a similar case since the tracked server state is continuously changed. However, it is intentional that original packets and retransmitted packets may differ in copying or not because we should clone packets by considering the state of servers. Furthermore, RPC frameworks should have a redundancy filtering mechanism or redundancy-aware response handling mechanism because a redundant response may not be filtered by the switch. Without such a mechanism, RPC frameworks may not work correctly. To support RDMA-based RPC frameworks, we need to revise the switch data plane partially for parsing the RDMA header and should address potential issues. Note that it is possible to parse and craft RDMA packets in the switch data plane [29].

3.8 Generality

A high-level takeaway from our work is that switches are an attractive high-performance vantage point to perform various functions in the era of microseconds. Therefore, we believe that the design choice of NetClone can inspire the emergence of other in-network computing systems for microsecond-scale applications. Furthermore, the proposed techniques can be applied to various in-network computing systems. For example, we leverage recirculation to assign the IP address to the cloned packet. This can be applied to other systems that replicate packets in the programmable switch, which include data replication, consensus, and multi-path routing. Request filtering using request fingerprints in the data plane can also be applied to other systems. For example, an admission control mechanism that requires frequent and quick rule updates can utilize this. This is because the rule update by the switch control plane is slow and has limited update throughput whereas the register update in the switch data plane is fast and offers line-rate throughput.

4 IMPLEMENTATION

4.1 Switch Data Plane

Our switch data plane is written in $P4_{16}$ [8] and is compiled with Intel P4 Studio SDE 9.7.0 for Intel Tofino [3]. We implement our data plane modules in the ingress pipeline because the switch should finish cloning decisions before packet forwarding. NetClone consumes 7 match-action stages when using two filter tables. We use 18.04% SRAM, 12.28% Match Input Crossbar, 26.79% Hash Unit, and 21.43% ALUs of the switch ASIC. Most memory space is used to keep track of request IDs in the filter tables where each table has 2^{17} hash slots. We can calculate how much throughput can be supported by the tables using a back-of-the-envelope calculation [44]. When the average request latency is 50μs, each slot can handle 20 KRPS. As we have a total of 2^{18} slots, the current NetClone prototype can support roughly 5.24 BRPS throughput. Since we use a 32-bit slot, our hash tables use roughly 1.05 MB, which is 4.77% of the switch memory.

4.2 Client-Server Application

We implement an open-loop multi-threaded application in C like prior work [11, 24, 44]. We use the NVIDIA Messaging Accelerator library (VMA) [2] for high-performance packet processing. The VMA allows applications to process packets in userspace with RDMA-like kernel-bypass networking, minimizing the packet processing delay in hosts. The client measures the throughput and latency by generating requests at a given target sending rate. It consists of two threads, one is the sender thread and the other is the receiver thread. The inter-arrival time between two consecutive requests is exponentially distributed. The server consists of a single dispatcher thread and multiple worker threads. The dispatcher enqueues received requests into a global request queue with FCFS policy. Worker threads dequeue requests and process them in parallel.

5 EVALUATION

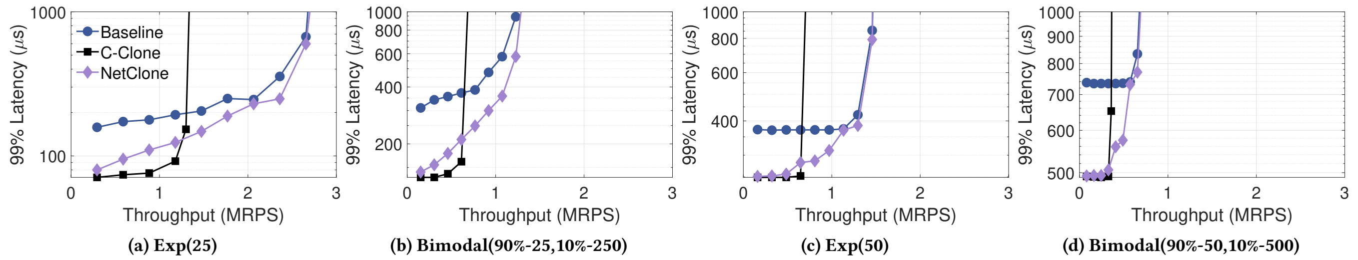
In this section, we evaluate NetClone. We first describe our experiment methodology. Next, we present experimental results with various workloads and system conditions.

5.1 Methodology

5.1.1 Testbed setup. To evaluate NetClone, 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 [3]. The servers are equipped with a 10-core CPU (Intel i5-12600K @ 3.7 Ghz, 12 hyperthreads and 4 non-hyperthreads), 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 and the remaining 6 servers are used as worker servers. The performance bottleneck is at worker servers.

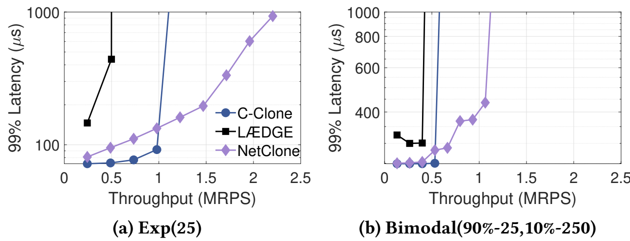
5.1.2 Workloads. We use a variety of synthetic and real-world application workloads similar to recent works [24, 37, 44]. The workloads use one-packet requests and responses with UDP like RackSched [44].

With a synthetic workload, a worker server processes a dummy RPC for a duration that we specify. The synthetic workload allows us to evaluate the performance of NetClone with various applications by emulating any target distribution of services and variability. Unless specified, we consider an exponential distribution with mean = 25 μs by default, which can represent common short-lasting RPCs. We also consider a bimodal distribution where 90% are 25 μs and 10% are 250 μs, which represents a mix of simple and complex RPCs. To inspect the impact of RPC duration, we use 50 μs and 500 μs as well. To emulate the service-time variability, we



(a) Exp(25) (b) Bimodal(90%-25,10%-250) (c) Exp(50) (d) Bimodal(90%-50,10%-500)

Figure 7: Experimental results for synthetic workloads.



(a) Exp(25) (b) Bimodal(90%-25,10%-250)

Figure 8: Comparison with the existing solutions.

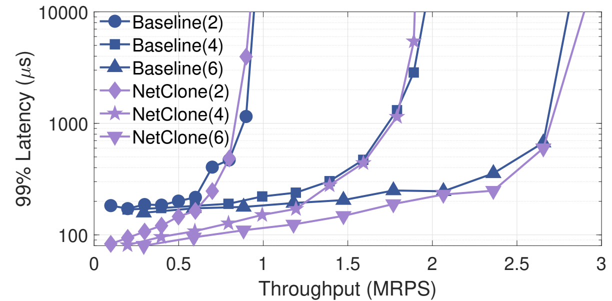


Figure 9: Impact of the number of servers.

follow the observations from LÆDGE [38]. We consider $p = 0.01$ and $p = 0.001$ to represent a high variability and a low variability, where p denotes the jitter probability to experience excessive long latency. We basically consider that workloads have high service-time variability, and the runtime of an RPC experiencing the unexpected jitter can take 15 times more than the normal case. For the real-world application workload, we use Redis [4], a widely deployed in-memory key-value store in many production systems.

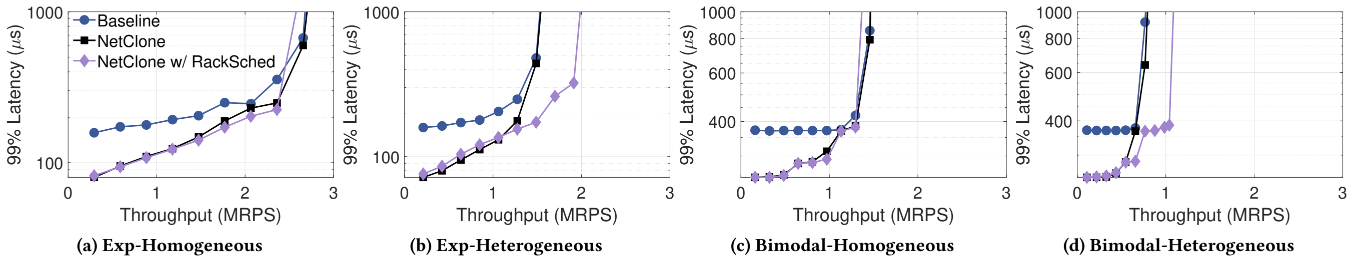
5.1.3 Compared Schemes. We compare our work against the baseline, C-Clone, LÆDGE [38]. The baseline sends requests to workers randomly without cloning. C-Clone is the client-based cloning mechanism that always sends duplicate requests to two random worker servers. LÆDGE performs dynamic cloning using the coordinator. In most experiments, we compare NetClone to the baseline and C-Clone because LÆDGE has significantly lower throughput than NetClone.

5.2 Synthetic Workloads Results

We plot the performance of three schemes, the baseline, C-Clone, and NetClone in Figure 7 for different workloads. Note that Y-axis is in log scale for better visibility. C-Clone shows limited throughput in all figures due to its static request cloning, which overloads worker servers beyond a certain point. Thanks to the dynamic cloning and response filtering, NetClone achieves low tail latency while maintaining similar throughput to the baseline. In Figure 7 (a) and (b), we can find that NetClone achieves better latency than the baseline across almost all loads. The average improvement is 1.48× and 1.27× for Exp(25) and Bimodal(90%-25,10%-250), respectively. Since the work servers become busier as throughput grows, NetClone clones requests less as well. Therefore, the degree of improvement decreases as the system load grows. Meanwhile, at low loads, NetClone experiences worse latency than C-Clone. This occurs because NetClone does not replicate requests when the tracked queue length is not zero. Note that the queue can build up occasionally, even at low loads. In Figure 7 (c) and (d), NetClone provides low tail latency at low loads, similar to Figure 7 (a) and (b). However, the performance improvement at high loads is negligible due to the longer processing time of RPCs that keeps the queue length non-zero at high loads.

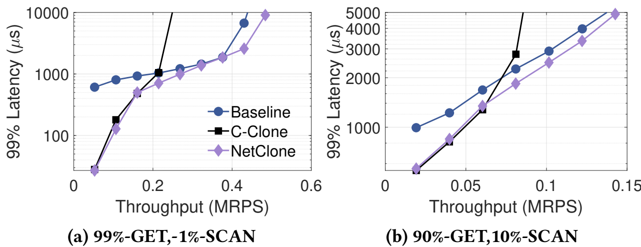
5.3 Scalability

5.3.1 Comparison with the existing solutions. In this experiment, we compare NetClone with C-Clone and LÆDGE to show that NetClone has better throughput and scalability. We use five worker servers because one server should be dedicated to the LÆDGE coordinator. The results, shown in Figure 8, indicate that NetClone provides high throughput, while LÆDGE and C-Clone exhibit low throughput. C-Clone does not incur latency overhead to clone requests but its static cloning limits system throughput. LÆDGE performs even worse than C-Clone since it relies on a CPU-based coordinator to clone requests. The coordinator server easily becomes a performance bottleneck, making it difficult to support high request rates with multiple worker servers. Even with a highly optimized coordinator, LÆDGE would be still behind NetClone, as switches can process billions of packets per second, while optimized servers can handle only a few million packets per second. This result demonstrates that performing request cloning in the switch is a desirable approach to achieve high performance.



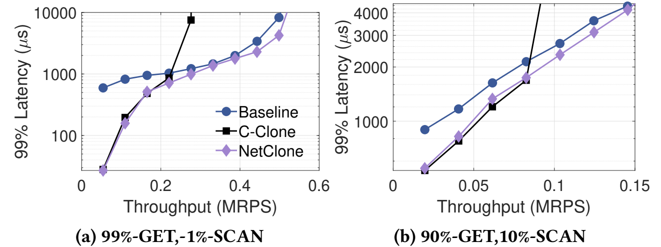
(a) Exp-Homogeneous (b) Exp-Heterogeneous (c) Bimodal-Homogeneous (d) Bimodal-Heterogeneous

Figure 10: Performance with RackSched under homogeneous and heterogeneous workloads.



(a) 99%-GET,-1%-SCAN (b) 90%-GET,10%-SCAN

Figure 11: Experimental results for Redis.



(a) 99%-GET,-1%-SCAN (b) 90%-GET,10%-SCAN

Figure 12: Experimental results for Memcached.

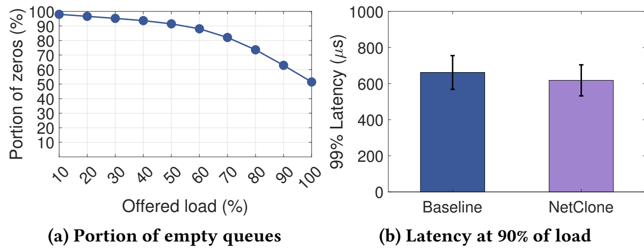
5.3.2 Impact of the number of servers. We now evaluate the scalability of NetClone by varying the number of worker servers. As NetClone performs request cloning in switches, it can scale out to multiple servers while maintaining low tail latency. Figure 9 shows the results for 2, 4, and 6 worker servers. We did not conduct an experiment with one server as NetClone requires a minimum of two servers for redundancy. As the number of worker servers increases, both NetClone and the baseline show improved throughput. NetClone maintains lower tail latency than the baseline regardless of the number of servers. One observation worth mentioning is that when the number of worker servers is two or four, NetClone shows worse latency at very high loads. This can be attributed to two reasons. First, NetClone sends cloned requests only when the server is idle, but the server may be busy in fact. We drop cloned requests if the actual state is busy, but the processing cost can be harmful if the number of redundant requests is large at very high loads. Second, with a small number of servers, there may not be enough idle servers available. Therefore, many cloned requests are forwarded to actually overloaded servers for a short time with herding effects, resulting in high tail latency at very high loads. However, when the number of servers is large, the probability of performance degradation decreases as NetClone has a larger pool of servers to choose.

5.4 Performance with RackSched

We now show how NetClone can make synergy with Racksched [44]. NetClone contributes to reducing latency and RackSched is effective to improve throughput. Figure 10 is experimental results for Exp(25) and Bimodal(90%-25,10%-250) workloads with a different number of workers. The homogeneous workloads assume that each worker server has an equal number of worker threads (15 worker threads and 1 dispatcher thread). In the heterogeneous workloads, three of the worker servers have 15 worker threads, while the other three have 8 worker threads. We see that NetClone with RackSched achieves the best performance, thanks to RackSched's ability to handle possible load imbalances between worker servers. NetClone with RackSched performs better with heterogeneous workloads than with homogeneous workloads because the latter workloads result in more imbalance loads. Meanwhile, in homogeneous workloads, NetClone with RackSched is worse than NetClone at very high loads, and we suspect that this is because the cases when the tracked state and the actual state are unmatched increase as RackSched makes the request queue empty more often.

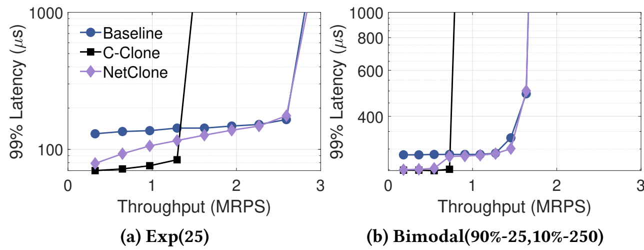
5.5 Applications: Redis and Memcached

We now show that NetClone is effective with real-world applications using Redis [4] and Memcached [15], which are popular in-memory key-value stores, commonly used in production services. We conduct experiments using 1 million objects with 16-byte keys and 64-byte values [33] by considering replicated key-value storage. Unlike previous in-network solutions for key-value stores [22, 23, 30, 43], NetClone does not impose any limitations on the key or value sizes, as it does not store keys or values in the switch data plane. In this experiment, clients generate read requests, and worker servers return values with a skewed key access pattern with Zipf-0.99. Note that NetClone does not clone write requests because the write coordination should be handled by replication protocols. We use 8 worker threads in each worker server. We vary the portion of GET and SCAN requests to 99%-GET,1%-SCAN and 90%-GET,10%-SCAN where GET reads a single object and SCAN reads 100 objects.



(a) Portion of empty queues **(b) Latency at 90% of load**

Figure 13: Confidence of the empty queue for state signaling.



(a) Exp(25) **(b) Bimodal(90%-25,10%-250)**

Figure 14: Experimental results with a low service-time variability (*p*=0.001).

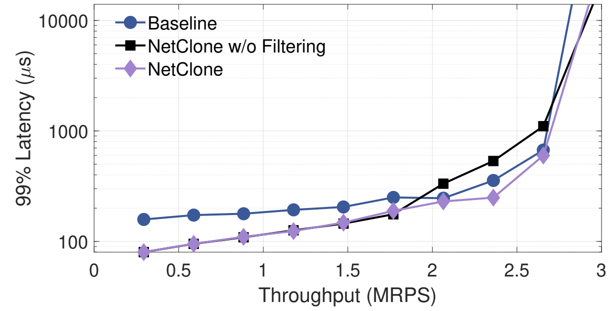


Figure 15: Impact of redundant response filtering.

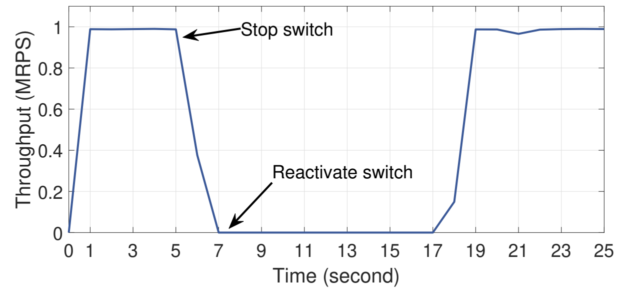


Figure 16: Performance under switch failures.

Figure 11 and Figure 12 show results, which have similar trends. We can see that, like the result with the synthetic workload, NetClone improves tail latency by masking service-time variability. The performance gap is the biggest at low loads, and the gap becomes small as throughput grows. C-Clone shows similar tail latency to NetClone, but its throughput is limited to half of NetClone as expected. In the Redis experiment, NetClone is better than the baseline by up to 22.59× and 1.77× for 99%-GET,1%-SCAN and 90%-GET,10%-SCAN, respectively. In Memcached, the largest improvement degree is 22.00× and the smallest one is 1.06× for 99%-GET,1%-SCAN. For 90%-GET,10%-SCAN in Memcached, NetClone achieves better tail latency than the baseline by 1.24× on average.

5.6 Deep Dive

5.6.1 Confidence of State Signals. NetClone considers the server as idle if the queue length of the server is zero. Therefore, we investigate the portion of empty queues by varying loads. We make a server record its current queue length when sending a response. In Figure 13 (a), we can see that the portion of empty queues decreases as the load grows, as expected. We see two important observations as follows. First, even at low loads, the queue may not be empty. This explains why NetClone shows higher latency than C-Clone at low loads in Figure 7. Second, likewise, queues do not always build up even under very high loads. This is the reason why cloning happens at not only low loads but also high loads. To check the efficiency of cloning at high loads, we run experiments with the baseline and NetClone 10 times at 0.9 of load and get the average tail latency and their standard deviations. Figure 13 (b) shows the results. As expected, we see that NetClone may cause worse latency than the baseline occasionally. However, by considering the average and the standard deviation, we can conclude that NetClone generally provides better latency than the baseline even at very high loads.

5.6.2 Impact of Service-Time Variability. Figure 14 shows the experimental results for synthetic workloads with a low variability of $p = 0.001$. The Y-axis of Figure 14 (a) and (b) is in the log-scale. We can see that NetClone can decrease tail latency even if the service-time variability is low. The trend of experimental results is similar to Figure 7. One difference is that performance improvement slightly decreases. However, it is not surprising since the benefit of request cloning comes from masking service-time variability.

5.6.3 Impact of Redundant Response Filtering. We now inspect the impact of redundant response filtering. To do this, we turn off the response filtering function and compare its performance against the baseline and NetClone. Figure 15 plots the result. We have the following observations. First, at low loads, redundant responses barely harm performance since the client has enough capability to handle redundancy. However, as the system load grows, the latency gets worse. The performance is even worse than the baseline at high loads if NetClone does not use response filtering. This means that filtering redundant responses plays an important role to optimize the performance of NetClone.

5.6.4 Performance under Switch Failures. In this experiment, we evaluate the resilience of NetClone to switch failures. Figure 16 shows the throughput for 25 seconds. The switch was stopped at 5 seconds and manually reactivated at 7 seconds. The throughput recovers after approximately 10 seconds. The downtime is not a

result of NetClone, but rather depends on the switch architecture. Thus, we can say that NetClone is robust to switch failures. Note that NetClone does not incur permanent misbehavior since the switch stores only soft states.

6 RELATED WORK

We briefly discuss existing works related to NetClone in terms of request cloning, server-level solutions, and in-network computing solutions.

Request cloning. Vulimiri *et al.* [39] investigates the tradeoff of client-based request cloning. They identify the threshold load and the client-side overhead. Gardner *et al.* [17, 18] provide rigorous theoretical analysis for cloning. Dolly [5] and RepFlow [40] utilize the cloning technique for mitigating stragglers in MapReduce clusters and multi-path routing in data center networks, respectively. LÆDGE [38] performs dynamic cloning using the coordinator but lacks low latency overhead and scalability. NetClone is the first dynamic request cloning system for microsecond-scale RPCs.

Server-level solutions for microsecond-scale RPCs. There are line of works that reduce the latency of RPCs at the server level in hardware and software. ALTOCUMULUS [42] avoids the scheduling overhead using direct register-level messaging. RPC-Valet [13] bypasses slow PCIe buses using shared caches when dispatching RPCs to CPU cores. nanoPU [19] bypasses the cache and memory hierarchy to provide a fast path from NIC to applications using a hardware accelerator. eRPC [26] improves the performance of small messages by optimizing common cases with various software techniques. IX [7], ZygOS [37], and Shinjuku [24] are data plane OSes that provide efficient CPU scheduling for microsecond-scale RPCs. For example, Shinjuku [24] implements a preemptive scheduling algorithm by re-queueing long-lasting RPCs if the runtime exceeds a given threshold. The above works address the RPC latency at the server level, whereas NetClone tries to optimize RPC latency at the cluster level. Since NetClone does not restrict the server-side mechanism to a specific solution, NetClone is orthogonal to the existing works.

In-network computing for microsecond-scale RPCs. The capability and flexibility of programmable switch ASICs trigger the emergence of in-network computing. NetCache [23], Pegasus [32], DistCache [34], Harmonia [43], NetLR [30], P4DB [20], and Transaction Triaging [21] are solutions to accelerate distributed storage. NetClone can improve the latency of GET queries and is a more generic solution. RackSched [44] is an in-network request scheduler for microsecond-scale RPCs that performs the JSQ load balancing. NetClone is orthogonal to RackSched since NetClone does not specify its load balancing algorithm.

7 CONCLUSION

In this paper, we presented NetClone, a new request cloning system that dynamically and quickly replicates requests to reduce the tail latency of microsecond-scale RPCs at scale. Unlike traditional client-based or coordinator-based cloning approaches, NetClone performs request cloning in the network switch using programmable switch ASICs. Various technical challenges to design and implement NetClone in the switch data plane were addressed. We have implemented a prototype of NetClone with an Intel Tofino switch and a cluster of commodity servers. The experimental results showed that NetClone effectively improves the tail latency of RPCs for both synthetic and real-world application workloads. We believe that network switches would play a key role as domain-specific hardware for microsecond-scale RPCs. We emphasize that there are remaining problems to realize the vision of in-network computing, which include fully synthesizing existing in-network solutions for microsecond-scale RPCs and integrating NetClone with existing RPC frameworks.

Ethics: This work does not raise any ethical issues.

ACKNOWLEDGEMENT

We would like to thank our shepherd, Andreas Haeberlen, and the anonymous SIGCOMM reviewers for their insightful comments and constructive feedback. This research was sponsored by the National Research Foundation of Korea (NRF) grants funded by the Ministry of Science and ICT (No. RS-2023-00240029). Gyuyeong Kim is the corresponding author.

REFERENCES

[1] 2019. Cavium XPliant Ethernet switch. https://www.openswitch.net/cavium/.
[2] 2021. NVIDIA Messaging Accelerator (VMA). https://docs.nvidia.com/networking/spaces/viewspace.action?key=VMAv940.
[3] 2023. Intel Tofino Programmable Ethernet Switch:. https://github.com/barefootnetworks/Open-Tofino.
[4] 2023. Redis Key-Value Store. https://redis.io/.
[5] Ganesh Ananthanarayanan, Ali Ghodsi, Scott Shenker, and Ion Stoica. 2013. Effective Straggler Mitigation: Attack of the Clones. In *Proc. of USENIX NSDI*. Lombard, IL, 185–198.
[6] Luiz Barroso, Mike Marty, David Patterson, and Parthasarathy Ranganathan. 2017. Attack of the Killer Microseconds. *Commun. ACM* 60, 4 (mar 2017), 48–54.
[7] Adam Belay, George Prekas, Mia Primorac, Ana Klimovic, Samuel Grossman, Christos Kozyrakis, and Edouard Bugnion. 2016. The IX Operating System: Combining Low Latency, High Throughput, and Efficiency in a Protected Dataplane. *ACM Trans. Comput. Syst.* 34, 4, Article 11 (dec 2016), 39 pages. https://doi.org/10.1145/2997641
[8] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, and David Walker. 2014. P4: Programming Protocol-independent Packet Processors. *SIGCOMM Comput. Commun. Rev.* 44, 3 (July 2014), 87–95.
[9] Pat Bosshart, Glen Gibb, Hun-Seok Kim, George Varghese, Nick McKeown, Martin Izzard, Fernando Mujica, and Mark Horowitz. 2013. Forwarding Metamorphosis: Fast Programmable Match-action Processing in Hardware for SDN. In *Proc. of ACM SIGCOMM* (Hong Kong, China). 99–110.
[10] Maury Bramson, Yi Lu, and Balaji Prabhakar. 2010. Randomized Load Balancing with General Service Time Distributions. In *Proc. of ACM SIGMETRICS* (New York, New York, USA). 275–286.
[11] Inho Cho, Ahmed Saeed, Joshua Fried, Seo Jin Park, Mohammad Alizadeh, and Adam Belay. 2020. Overload Control for μs-Scale RPCs with Breakwater. In *Proc. of USENIX OSDI*. USA, Article 17, 16 pages.
[12] Sharad Chole, Andy Fingerhut, Sha Ma, Anirudh Sivaraman, Shay Vargaftik, Alon Berger, Gal Mendelson, Mohammad Alizadeh, Shang-Tse Chuang, Isaac Keslassy, Ariel Orda, and Tom Edsall. 2017. dRMT: Disaggregated Programmable Switching. In *Proc. of ACM SIGCOMM* (Los Angeles, CA, USA). 1–14.
[13] Alexandros Daglis, Mark Sutherland, and Babak Falsafi. 2019. RPC-Valet: NI-Driven Tail-Aware Balancing of µs-Scale RPCs. In *Proc. of ACM ASPLOS* (Providence, RI, USA). New York, NY, USA, 35–48. https://doi.org/10.1145/3297858.3304070
[14] Jeffrey Dean and Luiz André Barroso. 2013. The Tail at Scale. *Commun. ACM* 56, 2 (Feb. 2013), 74–80.
[15] Brad Fitzpatrick. 2004. Distributed Caching with Memcached. *Linux J.* 2004, 124 (Aug. 2004), 5–.
[16] Yu Gan, Yanqi Zhang, Dailun Cheng, Ankitha Shetty, Priyal Rathi, Nayan Katarki, Ariana Bruno, Justin Hu, Brian Ritchken, Brendon Jackson, Kelvin Hu, Meghna Pancholi, Yuan He, Brett Clancy, Chris Colen, Fukang Wen, Catherine Leung, Siyuan Wang, Leon Zaruvinsky, Mateo Espinosa, Rick Lin, Zhongling Liu, Jake Padilla, and Christina Delimitrou. 2019. An Open-Source Benchmark Suite for Microservices and Their Hardware-Software Implications for Cloud & Edge Systems. In *Proc. of ACM ASPLOS* (Providence, RI, USA). 3–18.

[17] Kristen Gardner, Mor Harchol-Balter, Alan Scheller-Wolf, Mark Velednitsky, and Samuel Zbarsky. 2017. Redundancy-d: The Power of d Choices for Redundancy. *Operations Research* 65, 4 (2017), 1078–1094.

[18] Kristen Gardner, Samuel Zbarsky, Sherwin Doroudi, Mor Harchol-Balter, and Esa Hyytia. 2015. Reducing Latency via Redundant Requests: Exact Analysis. In *Prof. of ACM SIGMETRICS* (Portland, Oregon, USA). New York, NY, USA, 347–360.

[19] Stephen Ibanez, Alex Mallery, Serhat Arslan, Theo Jepsen, Muhammad Shahbaz, Changhoon Kim, and Nick McKeown. 2021. The nanoPU: A Nanosecond Network Stack for Datacenters. In *Proc. of USENIX OSDI*. 239–256.

[20] Matthias Jasny, Lasse Thostrup, Tobias Ziegler, and Carsten Binnig. 2022. P4DB - The Case for In-Network OLTP. In *Proc. of ACM SIGMOD* (Philadelphia, PA, USA). 1375–1389.

[21] Theo Jepsen, Alberto Lerner, Fernando Pedone, Robert Soulé, and Philippe Cudré-Mauroux. 2021. In-Network Support for Transaction Triaging. *Proc. VLDB Endow.* 14, 9 (may 2021), 1626–1639.

[22] Xin Jin, Xiaozhou Li, Haoyu Zhang, Nate Foster, Jeongkeun Lee, Robert Soulé, Changhoon Kim, and Ion Stoica. 2018. NetChain: Scale-Free Sub-RTT Coordination. In *Proc. of USENIX NSDI*. Renton, WA, 35–49.

[23] Xin Jin, Xiaozhou Li, Haoyu Zhang, Robert Soulé, Jeongkeun Lee, Nate Foster, Changhoon Kim, and Ion Stoica. 2017. NetCache: Balancing Key-Value Stores with Fast In-Network Caching. In *Proc. of ACM SOSP* (Shanghai, China). 121–136.

[24] Kostis Kaffes, Timothy Chong, Jack Tigar Humphries, Adam Belay, David Mazières, and Christos Kozyrakis. 2019. Shinjuku: Preemptive Scheduling for μsecond-scale Tail Latency. In *Proc. of USENIX NSDI*. Boston, MA, 345–360.

[25] Gopal Kakivaya, Lu Xun, Richard Hasha, Shegufta Bakht Ahsan, Todd Pfleiger, Rishi Sinha, Anurag Gupta, Mihail Tarta, Mark Fussell, Vipul Modi, Mansoor Mohsin, Ray Kong, Anmol Ahuja, Oana Platon, Alex Wun, Matthew Snider, Chacko Daniel, Dan Mastrian, Yang Li, Aprameya Rao, Vaishnav Kidambi, Randy Wang, Abhishek Ram, Sumukh Shivaprakash, Rajeet Nair, Alan Warwick, Bharat S. Narasimman, Meng Lin, Jeffrey Chen, Abhay Balkrishna Mhatre, Preetha Subbarayalu, Mert Coskun, and Indranil Gupta. 2018. Service Fabric: A Distributed Platform for Building Microservices in the Cloud. In *Proc. of ACM EuroSys* (Porto, Portugal). New York, NY, USA, Article 33, 15 pages. https://doi.org/10.1145/3190508.3190546

[26] Anuj Kalia, Michael Kaminsky, and David G. Andersen. 2019. Datacenter RPCs Can Be General and Fast. In *Proc. of USENIX NSDI*. USA, 1–16.

[27] Rishi Kapoor, George Porter, Malveeka Tewari, Geoffrey M. Voelker, and Amin Vahdat. 2012. Chronos: Predictable Low Latency for Data Center Applications. In *Proc. of ACM SoCC* (San Jose, California). New York, NY, USA, Article 9, 14 pages. https://doi.org/10.1145/2391229.2391238

[28] Antonios Katsarakis, Vasilis Gavrielatos, M.R. Siavash Katebzadeh, Arpit Joshi, Aleksandar Dragojevic, Boris Grot, and Vijay Nagarajan. 2020. Hermes: A Fast, Fault-Tolerant and Linearizable Replication Protocol. In *Proc. of ACM ASPLOS* (Lausanne, Switzerland). New York, NY, USA, 201–217.

[29] Daehyeok Kim, Zaoxing Liu, Yibo Zhu, Changhoon Kim, Jeongkeun Lee, Vyas Sekar, and Srinivasan Seshan. 2020. TEA: Enabling State-Intensive Network Functions on Programmable Switches. In *Proc. of ACM SIGCOMM* (Virtual Event, USA). 90–106.

[30] Gyuyeong Kim and Wonjun Lee. 2022. In-Network Leaderless Replication for Distributed Data Stores. *Proc. VLDB Endow.* 15, 7 (Mar. 2022), 1337–1349.

[31] Leslie Lamport. 1978. Time, Clocks, and the Ordering of Events in a Distributed System. *Commun. ACM* 21, 7 (jul 1978), 558–565. https://doi.org/10.1145/359545.359563

[32] Jialin Li, Jacob Nelson, Ellis Michael, Xin Jin, and Dan R. K. Ports. 2020. Pegasus: Tolerating Skewed Workloads in Distributed Storage with In-Network Coherence Directories. In *Proc. of USENIX OSDI*. 387–406.

[33] Hyeontaek Lim, Dongsu Han, David G. Andersen, and Michael Kaminsky. 2014. MICA: A Holistic Approach to Fast in-Memory Key-Value Storage. In *Proc, of USENIX NSDI* (Seattle, WA). USA, 429–444.

[34] Zaoxing Liu, Zhihao Bai, Zhenming Liu, Xiaozhou Li, Changhoon Kim, Vladimir Braverman, Xin Jin, and Ion Stoica. 2019. DistCache: Provable Load Balancing for Large-Scale Storage Systems with Distributed Caching. In *Proc. of USENIX FAST*. Boston, MA, 143–157.

[35] Sarah McClure, Amy Ousterhout, Scott Shenker, and Sylvia Ratnasamy. 2022. Efficient Scheduling Policies for Microsecond-Scale Tasks. In *Proc. of USENIX NSDI*. Renton, WA, 1–18.

[36] M. Mitzenmacher. 2001. The power of two choices in randomized load balancing. *IEEE Transactions on Parallel and Distributed Systems* 12, 10 (2001), 1094–1104.

[37] George Prekas, Marios Kogias, and Edouard Bugnion. 2017. ZygOS: Achieving Low Tail Latency for Microsecond-Scale Networked Tasks. In *Proc. of ACM SOSP* (Shanghai, China). New York, NY, USA, 325–341.

[38] Mia Primorac, Katerina Argyraki, and Edouard Bugnion. 2021. When to Hedge in Interactive Services. In *Proc. of USENIX NSDI*. 373–387. https://www.usenix.org/conference/nsdi21/presentation/primorac

[39] Ashish Vulimiri, Philip Brighten Godfrey, Radhika Mittal, Justine Sherry, Sylvia Ratnasamy, and Scott Shenker. 2013. Low Latency via Redundancy. In *Proc. of ACM CoNEXT*. 283–294.

[40] H. Xu and B. Li. 2014. RepFlow: Minimizing flow completion times with replicated flows in data centers. In *Proc. of IEEE INFOCOM*. 1581–1589.

[41] Zhuolong Yu, Yiwen Zhang, Vladimir Braverman, Mosharaf Chowdhury, and Xin Jin. 2020. NetLock: Fast, Centralized Lock Management Using Programmable Switches. In *Proc. of ACM SIGCOMM* (Virtual Event, USA). 126–138.

[42] Jiechen Zhao, Iris Uwizeyimana, Karthik Ganesan, Mark C. Jeffrey, and Natalie Enright Jerger. 2022. ALTOCUMULUS: Scalable Scheduling for Nanosecond-Scale Remote Procedure Calls. In *Proc. of IEEE/ACM MICRO*. 423–440. https://doi.org/10.1109/MICRO56248.2022.00039

[43] Hang Zhu, Zhihao Bai, Jialin Li, Ellis Michael, Dan R. K. Ports, Ion Stoica, and Xin Jin. 2019. Harmonia: Near-Linear Scalability for Replicated Storage with in-Network Conflict Detection. *Proc. VLDB Endow.* 13, 3 (Nov. 2019), 376–389.

[44] Hang Zhu, Kostis Kaffes, Zixu Chen, Zhenming Liu, Christos Kozyrakis, Ion Stoica, and Xin Jin. 2020. RackSched: A Microsecond-Scale Scheduler for Rack-Scale Computers. In *Proc. of USENIX OSDI*. 1225–1240.