Beaver: Practical Partial Snapshots for Distributed Cloud Services

Liangcheng Yu[♡] Xiao Zhang[♠] Haoran Zhang[♡] John Sonchack[♣] Dan Ports[♢] Vincent Liu[♡]
[♡]*University of Pennsylvania* [♠]*Shanghai Jiao Tong University* [♣]*Princeton University* [♢]*Microsoft / University of Washington*

Abstract

Distributed snapshots are a classic class of protocols used for capturing a causally consistent view of states across machines. Although effective, existing protocols presume an isolated universe of processes to snapshot and require instrumentation and coordination of all. This assumption does not match today's cloud services—it is not always practical to instrument all involved processes nor realistic to assume zero interaction of the machines of interest with the external world.

To bridge this gap, this paper presents Beaver, the first practical partial snapshot protocol that ensures causal consistency under external traffic interference. Beaver presents a unique design point that tightly couples its protocol with the regularities of the underlying data center environment. By exploiting the placement of software load balancers in public clouds and their associated communication pattern, Beaver not only requires minimal changes to today's data center operations but also eliminates any form of blocking to existing communication, thus incurring near-zero overhead to user traffic. We demonstrate the Beaver's effectiveness through extensive testbed experiments and novel use cases.

1 Introduction

The ability to capture a consistent, global view of a system is a powerful tool. For many tasks—deadlock detection, checkpoints and failure recovery, network telemetry, debugging of distributed software, and many others [3–5, 8, 9, 17, 33, 37, 39, 52, 54, 55, 58, 60]—a global view, and particularly a consistent one, is essential for correct operation. Without consistency, results are unreliable, and the value of associated tools is questionable.

The classic method for capturing consistent global states is the Chandy-Lamport snapshot algorithm that was proposed almost four decades ago and its subsequent variants [11, 26, 33–35, 41, 57, 60]. At a high level, these protocols flood snapshot initiation messages throughout the system, triggering local captures of state at every node they pass in a manner that guarantees causal consistency of the recorded values. Some versions (including the original) also include support of capturing messages that are in-flight at the time of the snapshot, i.e., channel state.

While these protocols have been simple, effective, and widely used for decades, they all rely on the fundamental assumption that the set of participants in the protocol is closed under causal propagation. In other words, if any node can both send and receive messages from participants in the protocol, it can propagate Lamport's 'happened-before' relation [35] and must also be a participant in the snapshot. For systems operating in isolation, ensuring full participation is trivial; however, modern cloud deployments are not so utopian.

Today's cloud services are often modular, e.g., structured as microservices, each of which might be developed and maintained by a different user, team, or organization or hosted on otherwise inaccessible infrastructure. Take, for instance, a managed pub/sub messaging layer like Amazon's Simple Notification Service (SNS). As a proprietary and black-box service, users cannot directly propagate snapshot initiation markers through the service. Further, while they might be able to add markers to the application-level content manually, with concurrency, replication, and reordering (e.g., due to prioritization), content-based markers are unlikely to track causal relationships accurately. Even when developers fully control all relevant servers, the clients of the service can also introduce hidden causal relationships, for example, when the user of a generative AI chatbot sends a follow-up message based on the response to the previous prompt. Ultimately, the nature of causal consistency means that a single non-participant can render all snapshots useless.

Observing this gap between classical assumptions and the practicalities of real-world deployments, we ask the question: Can we make distributed snapshots practical in modern cloud data centers, i.e., is it possible to capture a causally consistent snapshot when only a subset of the broader system participates? At first glance, this goal seems far-fetched: With partial participation, we cannot control the messaging behaviors nor instrument any coordination logic for machines external to those of interest. Complicating the issue is the fact that, to be practical, the protocol cannot block, e.g., by buffering or delaying user packets during a snapshot. In essence, this means that hidden causal relationships between participants and external communication partners are unavoidable.

This work presents Beaver[1], the first 'partial' snapshot protocol that extends the capability of distributed snapshots to cloud services with external interactions. Beaver provides the same basic abstraction as other snapshot protocols—for any event whose effects are observed in the snapshot, all other

[1]The animal species known for their engineering expertise in constructing dams using locally available materials such as rocks and tree branches.

events that ‘happened-before’ are also included. It achieves this even when the target service communicates with an arbitrary number of external, black-box entities, regardless of their scale, semantics, or placement, and despite potential multi-hop propagation of causal dependencies. Beaver does all of this without blocking or delaying user requests. Beaver tackles this seemingly impossible problem by:

1. Relying on two features found in all of today’s largest cloud data centers: (a) Layer-4 Software Load Balancers (SLBs) that interpose on a subset of inbound traffic [15, 28, 49, 50] and (b) servers with low time strata or otherwise stable clocks [13, 25, 29, 36, 38, 42, 44, 45].
2. Eschewing the enforcement of causal consistency in favor of simply *detecting* when violations may have occurred, a mechanism we call Optimistic Gateway Marking (OGM).

Note that for (1b), Beaver does not rely on the traditional notion of clock synchronization that other recent systems [13, 38, 42] are founded upon, which requires that the clocks of distinct machines have bounded drift. Instead, it uses a much weaker property [25, 40] over the frequency drift of a single machine[2]. Also note that (2) implies a tradeoff: snapshots are not always successful, but users can be assured of their correctness when they are and retry when they are not.

At a high level, Beaver’s approach is based on the observation that when examining the causal consistency of a snapshot, only inbound traffic is relevant and only a small subset therein. More specifically, we can divide inbound traffic into messages that are ‘causally irrelevant’ (e.g., triggered asynchronously and, thus, are not a part of any transitive causal relationships) and messages that are ‘causally relevant’ (e.g., triggered by post-snapshot outbound traffic but may not carry any markers of that fact). Beaver’s OGM mechanism is an approximate but full-recall detector of causally relevant traffic.

Our prototype[3] of Beaver demonstrates that not only is it possible to build an OGM mechanism, but by leveraging the aforementioned features of today’s cloud data centers, we can render the possibility of rejected snapshots minimal (near-zero in many cases). To summarize, this paper makes the following contributions:

- To the best of our knowledge, we are the first to detail the gap between classical assumptions of distributed snapshots and the practicalities of real-world clouds.
- We propose Beaver, the first partial distributed snapshot primitive for modern cloud services. Beaver presents a unique design point by tightly coupling the protocol with the regularities of the underlying data center environment.
- We evaluate Beaver through end-to-end implementation on a real-world testbed aligned with the production data center settings. We also show that the causally consistent view provided by Beaver enables a spectrum of use cases.

[2]Bounded clock drift to a low-stratum reference server is sufficient to guarantee bounded local frequency drift, but not necessary.

[3]The prototype is available at https://github.com/eniac/Beaver.

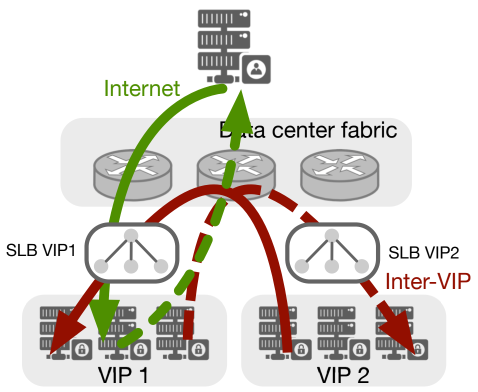


Figure 1: Today’s public cloud services place SLBs to handle the external traffic to its VIP in the inbound direction (solid lines to VIP 1). The response to inbound messages (dotted lines from VIP 1) typically bypasses its SLB to minimize the SLB traffic load.

2 Background and Motivation

We begin by describing the structure of today’s cloud services and the data centers in which they reside before we discuss the application of distributed snapshots to these services.

2.1 Communication in Public Cloud Data Centers

Today’s cloud data centers are massive collections of servers connected by a network fabric that host user services of diverse sizes and scopes. In this context, we can abstract user services as a set of virtual or bare metal machines managed as a single logical entity. Each service is typically assigned a public Virtual IP (VIP) address, and each physical machine a private Direct IP (DIP) address [15, 50].

Software Load Balancers (SLBs). A set of dedicated servers or programmable devices is responsible for translating between VIPs and DIPs. We refer interested readers to prior work [15, 50] for full details, but at a high level, these layer-4 devices act similarly to traditional Network Address Translators (NATs), allocating a new mapping for every new connection and rewriting the headers of every passing packet according to the mapping. In cloud systems, these devices are distributed, replicated, and serve an additional purpose as software load balancers that spread requests over available backend servers. A single service/VIP typically has a dedicated set of SLBs based on its scale (e.g., ∼7–20, including replication).

The path of packets in public clouds. In the presence of SLBs, packets can take different paths depending on the relationship between their source and destination (Figure 1):

Internet traffic: Incoming packets from the Internet are always routed through an SLB to translate from the service’s publicly visible VIP to a relevant internal DIP [15, 21, 22, 43, 50, 63]. Unlike most other NAT-like mechanisms, response packets are usually sent back directly, bypassing the SLB using techniques like Direct Server Return (DSR) [15, 50].

Inter-service traffic: Inbound traffic from other services within the same provider also passes through SLBs [15, 21, 22, 43, 50, 63], which still need to perform the same VIP-to-DIP translation. This is true even if the two service's servers are physically adjacent. Note that, like with Internet traffic, outbound traffic can bypass the responder's SLB; however, even in this case, the packet will still need to pass through the SLB responsible for the destination VIP(s), as shown in Figure 1. Note that while cached DIPs have been suggested to bypass inbound SLBs on the fast path [50], this optimization is currently disabled for major classes of production traffic due to load imbalance and cache management issues. The implication is that, at least for public clouds, this need to interpose on all inbound traffic is ubiquitous [10, 15, 63].

Intra-service traffic: Finally, messages between sources and destinations belonging to the same VIP are sent directly, bypassing the SLBs entirely.

Typical service communication patterns. In parallel to the above, we note that modern cloud services rarely operate in isolation. Frontend services typically rely on a wide array of backend services, e.g., to handle storage, analytics, and learning, thus triggering inter-service traffic. The rise of managed cloud service offerings and microservice design patterns have further encouraged modularity and the associated growth in the number of distinct services involved in processing a single user request. At a more basic level, most cloud services take requests from and return responses to external clients, each with its own internal, causality-carrying logic.

2.2 Revisiting the Chandy-Lamport Snapshot

The ability to capture a consistent snapshot of a cloud service's global state is a powerful tool. Indeed, many problems in distributed systems boil down to determining the global state across machines, including distributed logging and debugging, network telemetry, checkpointing and recovery, and deadlock detection [3, 11, 17, 33, 56, 60].

Intuitively, a snapshot is a collection of local states captured from the processes of a system. For simplicity, we omit channel states in our definitions, but the analysis is similar. The snapshot is deemed consistent if the captured states at each process 'cut' the timeline of events in a way that respects the following definition:

DEFINITION 1. *(Consistent Snapshot [11, 56]). For a snapshot, let C be the set of events on every process that occurs before the 'cut'. C is causally consistent iff $\forall e \in C$, if $e' \rightarrow e$, then $e' \in C$, where $x \rightarrow y$ denotes that x 'happened before' y.*

The seminal Chandy-Lamport algorithm was the first to present a solution for this problem. We refer the interested readers to the original paper [11] or a distributed systems textbook [33, 56] for complete details, but we give a simplified description of the model and the protocol below:

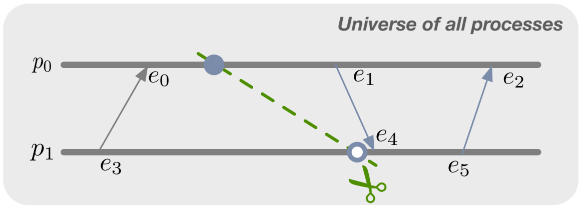


Figure 2: A minimal example of a consistent cut for 2 processes p_0, p_1 and 6 events $e_{0,1,\ldots,5}$. The global snapshot formed from the collection of ○ and ● is a 'causal cut' of the event timelines for all processes, where ● and ○ indicate snapshot initiations triggered out-of-band or by receiving marker messages, respectively.

- *Model:* A system involves a set of asynchronous processes $P = \{p_0, p_1, \ldots, p_{N-1}\}$ that interconnect with each other through FIFO message channels. Each process p_i holds state of interest, s_i, that may change in response to local events (e.g., local computation, message sends or receives, etc.). A global snapshot involves a union of states $\{s_i\}$ recorded at different times for all processes.
- *Protocol state machine:* The protocol requires coordination in *all* processes $p \in P$. An initiator process first records its local state and then sends a marker message to all others. The captured state is application-dependent and can range from a single bit representing the state of a lock to all of local memory. When any other process p_i receives a marker message for the first time, it records its state s_i and, to ensure consistency, sends marker messages immediately through all other channels.

Later variants refine the basic algorithm to generalize channel assumptions, allow for concurrent initiation, or reduce message complexity [26, 33, 34, 41, 57, 60]. In particular, the Lai-Yang algorithm permits non-FIFO and lossy channels by having processes piggyback a single marker bit in every sending message [34] rather than sending separate marker messages as in the original protocol. Upon receiving a message with a marker bit set, the receiving process *first* records the local state, processes the payload, and sets the bit for future sending messages. Additional bits can be used to support concurrent snapshots. Figure 2 shows a consistent cut with the Lai-Yang algorithm.

2.3 A Case for Partial Snapshots

The above snapshot algorithm makes a fundamental and implicit assumption that *all* processes that can communicate with processes in P are themselves in P. Unfortunately, as previously mentioned, today's cloud services are frequently interconnected, with efforts toward modular design and managed solutions promoting increasing complexity in the dependency graph over time. As a rough indication of severity,

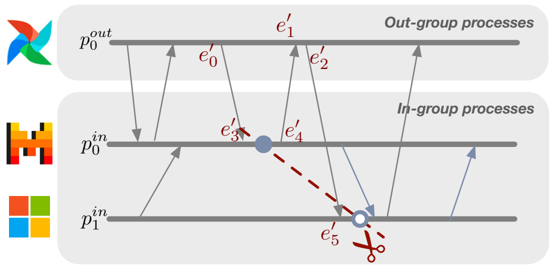


Figure 3: An application where a distributed serving system is accessed by an external user (e.g., an Apache Airflow workflow). The out-group process p_0^{out} imposes a hidden causal relationship $e'_4 \to e'_5$ between events e'_4 and e'_5, rendering a traditional snapshot of only the serving system inconsistent.

previous studies have shown that inter-service traffic comprises 10–50% of total traffic in the data center, and Internet traffic accounts for 5–25% [21, 22, 50].

Consider, for instance, a HuggingFace-like ML inference service [2] that hosts a collection of models that can be accessed from external clients. As they are externally visible, the models are frequently used in larger jobs, e.g., as part of an interactive chatbot (where clients submit requests based on prior responses) or more complex Apache Airflow workflows.

The inference service might want to capture a service-wide statistic (e.g., tracking the maximum number of in-flight requests) to decide on the number of servers to provision. Any analysis of the developer's application that does not consider the potential dependencies introduced by external services or clients will miss important causal dependencies.

Figure 3 shows a simple example of this, where a single external Airflow job makes requests to multiple models hosted by the inference service such that only one request is outstanding at any given time. Occasional internal messages are for monitoring and coordination. Although there is at most one outstanding request at any given time, a traditional distributed snapshot that only considers the inference service will not respect that bound.

For example, in Figure 3, the depicted cut 'observes' two inflight messages because it fails to capture the external interactions ($e'_5 \in C$, yet $e'_4 \notin C$). In fact, for a single client that issues a single request at a time, an n-server snapshot can 'observe' any number of in-flight requests [0, n]. These arbitrary results can cause the developer to waste money and resources on redundant provisioning. More broadly, while the frequency and consequences of consistency violations are application-dependent, there is often a meaningful difference between 'correct' and 'incorrect'.

Although converting all cloud services into participants of the snapshot protocol might be possible given either (a) a well-resourced developer who can implement and manage everything (even if machines are geo-distributed or on the broader Internet) in-house or (b) support from the cloud provider to propagate snapshot markers on all packets, these approaches are not always feasible. For (a), the popularity of managed services demonstrates their importance to low-cost and agile development. For (b), forced instrumentation can lead to overhead and fragmentation for users not involved in the snapshot. Even worse, if the external source of dependencies is a human (e.g., accessing your service through a browser), incorporating her into the snapshot is impractical.

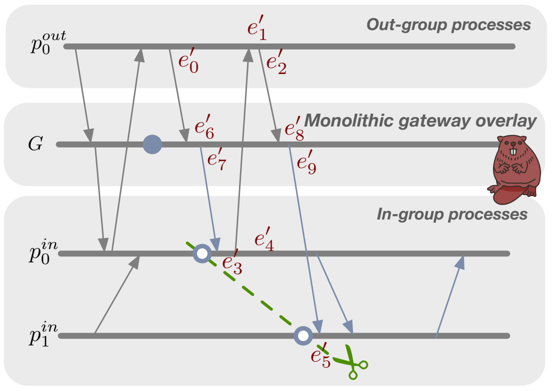


Figure 4: With the gateway indirection, Beaver's MGM results in a new frontier at the in-group process p_1^{in} that precedes rather than succeeds the event e'_5 (as in the scenario of Figure 3), converging to a consistent partial snapshot.

A formal definition of partial snapshots. We seek the design and implementation of a partial snapshot. In a partial snapshot, processes are divided into two groups. The first, *in-group processes* P^{in}, are the machines of the VIP(s) of interest. The second, *out-group processes* P^{out}, includes all other machines, whether in the same data center or the broader Internet.

Given these sets, we refine Definition 1 to obtain a definition of consistent partial snapshots:

Definition 2. *(Consistent Partial Snapshot). Consider a universe of processes* $P = P^{in} \cup P^{out}$, $P^{in} \cap P^{out} = \emptyset$. *Let* C_{part} *be the set of pre-snapshot events for* P^{in}. C_{part} *is causally consistent iff* $\forall e \in C_{part}$, *if* $e'.p \in P^{in} \wedge e' \to e$, *then* $e' \in C_{part}$.

Similar to traditional snapshots, for a set of in-group processes P^{in}, if a consistent partial snapshot includes the effect of an event e, it must include any event e' at $p \in P^{in}$ that leads to it. Like traditional snapshots, the 'happened before' relation, $\to$ is transitive and defined over events in the universe of processes. Unlike traditional snapshots, however, the included events only account for in-group events.

3 Gateway Marking

This paper introduces Beaver, a partial snapshot primitive that captures a causally consistent collection of state for cloud services sitting behind one or more operator-specified VIPs.

Fundamentally, the nodes in P^{out} are uncontrollable and, as a result, can introduce arbitrary hidden causal relationships, disrupting the consistency of traditional snapshots. At

Symbol	Description
P	Set of all processes.
P^{in}	Set of in-group processes with states of interest.
P^{out}	Set of out-group processes without any control.
G	Set of gateways handling inbound traffic for P^{in}.
C	Set of pre-snapshot events for a snapshot 'cut'.
e	Event tuple $e = (p, m, t)$.
$e.p$	The process at which an event e occurs.
$e.m$	The message involved in an event e, if any.
$e.t$	Global wall clock time, for ease of discussion.
e^{ss}_{gmax}	The event when the last gateway is in a new snapshot.
e^{ss}_{gmin}	The event when the first gateway is in a new snapshot.
e^{ss}_{g}	The event when $g \in G$ enters a new snapshot.
e^{ss}_{p}	The event triggering $p \in P^{in}$ to enter a new snapshot.
$d(p,q;V)$	One way delay from p to q with intermediate nodes $v \in V$ $(p,q \in (P \cup G), V \subseteq (P \cup G))$ in sequence.
τ_{min}	Min time for an external causal chain to occur.

Table 1: Summary of notations in Beaver.

the core of Beaver is a primitive called Optimistic Gateway Marking (OGM), which allows Beaver to detect when such causality violations may have occurred. As we show later in §5, by combining this primitive with common-case features of today's cloud data centers, Beaver can provide:

- Partial deployability where *only* the in-group machines for the target VIP(s) participate while ensuring high-rate, consistent partial snapshots for the target service(s).
- Minimal cost for data center infrastructure, for example, without switch reconfiguration or additional SLB replicas.
- Near-zero impact on existing data center service traffic.

In this section, we first introduce a strawman version of the primitive before discussing practicalities and how Beaver addresses them with OGM in §4.

Strawman: Monolithic Gateway Marking (MGM). Beaver starts with a simple idea: for all packets originating from out-group nodes and destined for in-group nodes, route them through a gateway. The gateway is responsible for two tasks:

1. Tagging incoming packets to in-group nodes with snapshot markers.
2. Initiating snapshots by tagging all subsequent inbound messages accordingly.

After the gateway initiates a snapshot, the protocol proceeds as a traditional snapshot among the in-group nodes. For the strawman, assume that the gateway is implemented by a single monolithic node. Figure 4 shows an example execution using the above protocol and the same application-level communication pattern as Figure 3. In contrast to Figure 3, indirection and marking via a gateway cause p^{in}_1 to take the snapshot at the correct time. In a way, the gateway node in this protocol can be seen as a stand-in for all nodes in P^{out}. We can prove that MGM produces a consistent partial snapshot.

THEOREM 1. *With MGM, a partial snapshot C_{part} for $P^{in} \subseteq P$ is causally consistent, that is, $\forall e \in C_{part}$, if $e'.p \in P^{in} \wedge e' \to e$, then $e' \in C_{part}$.*

PROOF. *Let $e.p = p^{in}_i$ and $e'.p = p^{in}_j$. There are 3 cases:*

1. *Both events occur in the same process, i.e., $i = j$.*
2. *$i \neq j$ and the causality relationship $e' \to e$ is imposed purely by in-group messages.*
3. *Otherwise, the causality relationship $e' \to e$ involves* at least *one $p \in P^{out}$.*

In cases (1) and (2), the theorem is trivially true using identical logic to proofs of traditional distributed snapshot protocols. We prove (3) by contradiction.

Assume $(e \in C_{part}) \wedge (\exists e' \to e)$ but $(e' \notin C_{part})$. With (3), $e' \to e$ means that there must exist some e^{out} (at an out-group process) satisfying $e' \to e^{out} \to e$. Now, because $e' \notin C_{part}$, we know $e^{ss}_{p^{in}_j} \to e'$ or $e^{ss}_{p^{in}_j} = e'$, that is, p^{in}_j's local snapshot happened before or during e'. Combined with the fact that the gateway is the original initiator of the snapshot protocol, we know that $e^{ss}_g \to e' \to e^{out} \to e$.

We can focus on a subset of the above causality chain: $e^{ss}_g \to e$. From the properties of the in-group snapshot protocol, $e^{ss}_g \to e$ implies $e \notin C_{part}$.

This contradicts our original assumption that $e \in C_{part}$! □

Theorem 1 implications: Beyond correctness, the strawman exhibits several valuable properties:

1. *Obliviousness to out-group semantics:* The proof treats the internals of the out-group processes as a black box. In fact, the protocol remains correct, even if the causal dependency results from multiple network hops through distinct out-group nodes or if an element of the out-group chain is a human.
2. *Obliviousness to outbound messages:* The gateway only needs to observe messages inbound to in-group processes without requiring any visibility or tagging of outbound messages. MGMs achieve this by initiating the snapshot at the gateway, which—as a stand-in for P^{out}—obviates the need to track dependencies carried to the out-group.

SLBs as a candidate for gateway marking. The SLBs described in §2.1 are a convenient candidate for implementing gateway marking as VIPs are a natural granularity for service-specific partial snapshots, and SLBs already interpose on all incoming traffic to a VIP—regardless of whether it is from the Internet or a different service. MGM's obliviousness to outbound messages helps here as well, making the system amenable to DSR.

Of course, assuming that a single server can handle all incoming traffic to a service is not feasible. The scale of modern SLBs serves as proof that even for simple gateway processing incoming requests for a single service, multiple servers are necessary to handle typical data volumes, load balance among SLBs, and provide fault tolerance.

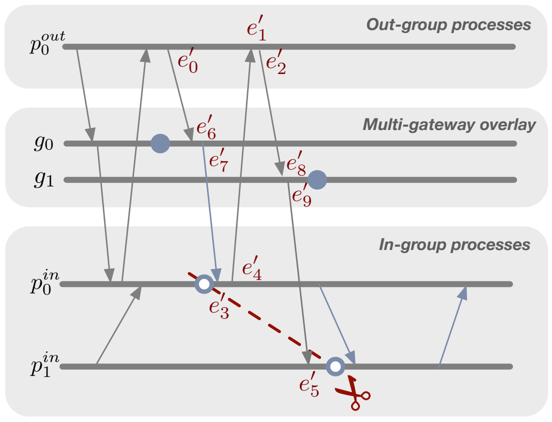


Figure 5: An inconsistent partial snapshot using two asynchronous SLBs g_0, g_1. When $e'_8.m$ arrives at g_1, g_1 has not initiated the new snapshot mode to mark the message, thus triggering the violation.

4 Optimistic Gateway Marking (OGM)

Beaver extends gateway marking to practical, distributed environments using OGM. First, to see why asynchronous SLBs could break the consistency guarantee, consider a simple scenario in Figure 5 where two SLBs, signaled by an out-of-band controller, initiate a new snapshot. When g_0 initiates snapshot mode and marks $e'_6.m$, it triggers a snapshot at p_0^{in}. However, a new message $e'_2.m$ from p_0^{out} is routed to a different gateway g_1[4], which has not yet entered snapshot mode. This leads to inconsistency: while $e'_5 \in C$ and $e'_4 \to e'_5$, $e'_4 \notin C$.

To block or not to block? An obvious solution would be to block inbound packets at SLBs during a snapshot and only resume forwarding them after all SLBs have 'committed' to the new snapshot. Unfortunately, this method introduces large overheads—not only to the applications, whose response times will spike while the SLB is blocking requests but also to the cloud providers, where the SLB would require large buffers and overprovisioned capacity to drain said buffers after a snapshot.

Rather than trying to enforce consistency, Beaver seeks a method to (a) detect *in*consistency, (b) reject snapshots when they are potentially inconsistent, and (c) minimize the rejection rate. It seeks to do this with near-zero overhead for applications and cloud infrastructure.

4.1 Causal Relevance and Irrelevance

A key idea in Beaver is that, even among the incoming traffic to the in-group, only a subset of that traffic is causally relevant. Using Figure 5 to illustrate, an incoming message, m, is **causally relevant** only when (1) an initiated SLB (g_0) sends a marked message to an in-group node (e.g., p_0^{in}), (2) that node interacts directly or indirectly with an out-group node

[4]This is typical in ECMP routing, where connections even from the same source may reach different SLBs.

(e.g., p_0^{out}), and (3) that out-group node sends m back to a different in-group node via an uninitiated SLB (e.g., g_1). Other communication patterns, e.g., an m triggered by an uninitiated process, are **causally irrelevant**.

In essence, causally relevant messages are only produced if the message loop: $GW_A \to IN_A \to OUT \to GW_B$ all occurs within the window of time in which the gateways are propagating snapshot initiation. More formally:

Theorem 2. *In a system with multiple asynchronous gateways, let the wall-clock time of the first and last gateway initiating snapshots be $e^{ss}_{gmin}.t = \min_{e^{ss}_g}(e^{ss}_g.t)$ and $e^{ss}_{gmax}.t = \max_{e^{ss}_g}(e^{ss}_g.t)$, $\forall g \in G$, respectively. Also let $\tau_{min} = min(d(g,g';\{p,q\}))$, $\forall g,g' \in G$, $p \in P^{in}$, and $q \in P^{out}$. If $e^{ss}_{gmax}.t - e^{ss}_{gmin}.t < \tau_{min}$, then the partial snapshot is causally consistent.*

Proof. *We extend the proof of Theorem 1 to a distributed setting. Similar to Theorem 1, there are three cases, with (3) being the one that differs. We again prove it by contradiction. Assume $(e \in C_{part}) \wedge (\exists e' \to e)$ but $(e' \notin C_{part})$. As before, there must be some chain $e' \to e^{out} \to e^g \to e$. Because $e' \notin C_{part}$, we have $e^{ss}_{p_j^{in}} \to e'$ or $e^{ss}_{p_j^{in}} = e'$, that is, p_j^{in} must have been triggered directly or indirectly by an inbound message. Denote the arrival of this inbound message at its marking gateway as $e^{g'}$. By the definition of τ_{min}, we have $e^g.t - e^{g'}.t \geq \tau_{min} > e^{ss}_{gmax}.t - e^{ss}_{gmin}.t$. Thus, at event e^g, the gateway must have already initiated the snapshot and will mark $e^g.m$ before forwarding. This results in $e \notin C_{part}$, a contradiction!* □

Theorem 2 implications: Informally, this theorem suggests that if the time gap between the first and last SLB snapshot initiations ($e^{ss}_{gmax}.t - e^{ss}_{gmin}.t$) is sufficiently small, or the minimum time for a message to revisit a gateway (τ_{min}) is long enough, causally relevant messages are impossible and the concerned partial snapshot is provably consistent[5].

Causally relevant messages are rare in the real world. Intuitively and with anecdotal evidence, the inequality $e^{ss}_{gmax}.t - e^{ss}_{gmin}.t < \tau_{min}$ can be satisfied with an exceedingly high probability in real-world contexts:

For the LHS ($e^{ss}_{gmax}.t - e^{ss}_{gmin}.t$): This time gap is essentially the difference in one-way delays between the controller and each of the SLBs. As SLBs share a region with the target service, a well-placed initiator (e.g., equidistant from all target SLBs or one whose messages are forced to travel to the root of the data center fabric) can simultaneously ensure reactive snapshot initiation and $e^{ss}_{gmax}.t - e^{ss}_{gmin}.t$ of near zero.

For the RHS (τ_{min}): This value includes multiple network hops, extending from an SLB to in-group nodes, then to out-group nodes, and back to an SLB. Particularly when out-group nodes are in other data centers or are end-host clients,

[5]In principle, another sufficient condition is when the in-group snapshot completes quickly enough. We do not rely on this because it has worse scaling properties than SLB convergence, but it can be added as an optimization.

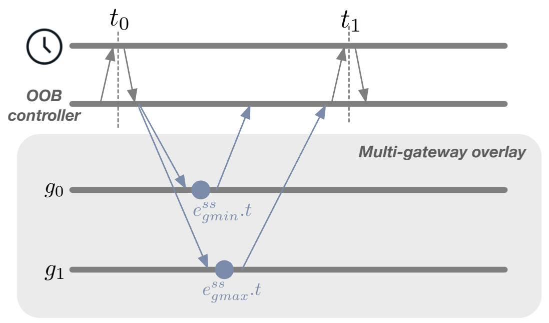


Figure 6: The time difference $t_1 - t_0$ as a safe upper bound for $e^{ss}_{gmax}.t - e^{ss}_{gmin}.t$ by querying a single hardware clock source with bounded frequency drift.

this value can be orders of magnitude higher than typical values for the LHS—on the order of milliseconds or tens of milliseconds. However, even when the out-group nodes are in the same region or data center as the in-group, we can still expect that this value is higher than any observed delta between initiator-to-SLB one-way delays as it includes *at least three trips* through the data center fabric in addition to processing time at the in/out-group network stacks[6].

4.2 Efficiently Verifying Causal Irrelevance

The primary technical challenge of OGM is minimizing the LHS of the above inequality and efficiently/confidently verifying that the resulting inequality held for a given snapshot, even in the presence of message drops, delays, and other sources of unexpected latency. The cloud provider can compute the two sides of the inequality separately.

4.2.1 Computing a Lower Bound for the RHS

For the RHS, the value can be determined statically, as dynamic network conditions like failures and congestion can only add to the latency of the message sequence. The latency is then equivalent to the sum of each hop's minimum propagation, transmission, and processing delays. These values depend on the relative placements of the in-group nodes, SLBs, and out-group communication partners, but all of those are known at runtime. To ensure a conservative lower bound, operators can and should assume that application-level processing and transmission delays are zero (Figure 13).

4.2.2 Determining an Upper Bound for the LHS

The LHS is harder to compute statically as failures and congestion mean a true upper bound may not exist[7]. Instead, we need to measure an upper bound online for the observed difference between gateway timestamps ($e^{ss}_{gmax}.t - e^{ss}_{gmin}.t$) for the snapshot in question.

[6]Even with the detection criteria later described in §4.2, LHS entails only 2 trips and encompasses a simpler data path with SLB stacks that are heavily optimized for minimal processing latency and jittering [15, 18, 22, 63].

[7]Beyond the heat death of the universe or at least the life of a data center.

	Rubidium	JILA Sr	Quartz	Quartz (calibrated)
Δf	± 0.05 ppb	$\pm 2.1 \times 10^{-18}$	± 100 ppm	± 100 ppb

Table 2: Frequency drift (Δf) uncertainty range of today's clocks, ppb (parts per billion) $= 10^{-9}$, ppm (parts per million) $= 10^{-6}$.

The typical method of measuring time gaps on different machines is via clock synchronization. Although today's clock synchronization techniques can achieve microsecond or sub-microsecond precision, fundamentally, they rely on frequent cross-machine messaging to correct the offset, which is sensitive to congestion and failures, thus impacting the bound on clock drift in the worst case [23, 62]. Data center services like TrueTime provide a reliable interface to query time points and calculate their differences. However, a general timing service incurs higher overhead and a typical clock uncertainty range of 1–7 ms [13], much greater than the timescales relevant for Beaver detection.

Synchronization-free approach. Beaver adopts an alternative, customized approach using a single hardware clock to calculate the elapsed time. As depicted in Figure 6, the controller queries the start time at t_0 from this clock source with a read t_0^r before initiating a new snapshot. Once the final ACK from the SLBs arrives, it reads the end time t_1^r at t_1 from the source, where t_0, t_1 represents the global wall clock time, and t_0^r, t_1^r the actual clock reads. This hardware clock can be a local hardware clock from either a COTS PCIe NIC [46] or from one equipped with an atomic clock, which are increasingly deployed in production data centers [42, 48].

Note that $t_1 - t_0$ is an upper bound on the LHS as $t_1 > e^{ss}_{gmax}.t$ and $t_0 < e^{ss}_{gmin}.t$. Thus, if $t_1 - t_0 < \tau_{min}$, the partial snapshot under examination is consistent. In practice, the time difference $t_1^r - t_0^r$ is adjusted to account for the maximum frequency drift Δf according to the clock data sheet, to determine an upper bound estimate for the corresponding elapsed time $t_1 - t_0$, thus the detection criteria $(t_1^r - t_0^r) \times (1 + \Delta f) < \tau_{min}$. This method, which relies solely on a single hardware clock to calculate time differences, eliminates issues common in traditional clock synchronization approaches, such as cross-machine message congestion and errors stemming from delays in clock readings due to software interrupts. The frequency drift of a single clock is relatively low and is mainly deterministically affected by temperature, which has low variance in modern data centers [38, 44, 59]. Standard quartz crystal oscillators in production data centers typically drift by ± 100 ppm, or 0.01% error [13, 25, 36, 38, 44]; recent studies are able to reduce this drift of quartz clocks in commodity data center servers to ± 100 ppb (10^{-7} error) by calibrating the offset due to temperature variations. More advanced oscillators (e.g., atomic clocks) can reduce this frequency drift by further orders of magnitude [29, 45] (Table 2).

Snapshot invalidation. While ensuring correctness (i.e., no false negatives), our proposed upper bound adds an additional margin to the original time gap. This margin comprises the clock query latency and the RTT between the controller and the SLBs, which may lead to false positives. In practice, however, we note that many devices support precise hardware timestamping along with the packet data path (i.e., when sending the first notification and when receiving the last notification). Our evaluations on a cloud data center in §7 reveal that the resulting snapshot invalidation rate is $< 5\%$ for typical SLB scales today, even in worst-case scenarios when the out-group nodes are in the same data center and under stressed snapshot operation frequencies.

In the end, false positives—while leading to the invalidation of potentially consistent snapshots—are of little concern due to our system's efficient snapshot operations and its ability to achieve a high snapshot rate.

5 Beaver's Partial Snapshot Protocol

As mentioned previously, Beaver's snapshot 'quantum' is a single VIP—Beaver can provide snapshots for one or more such VIPs within a single region.

Operation. At a high level, Beaver's partial snapshot protocol distinguishes itself from traditional snapshots in two aspects: (1) its lightweight SLB marking logic for inbound traffic and (2) the snapshot verification process at the controller.

In-group processes: Among in-group processes, Beaver inherits its coordination logic (and the omitted, optional recording of in-flight messages) from prior snapshot algorithms [34, 60] that piggyback 'marker' information per message to handle non-FIFO and lossy channels[8]. Figure 7 depicts the core logic: upon receiving a packet, either from an SLB or another in-group process, the current in-group process evaluates if $pkt.sid > csid$. If true, it signals a new snapshot operation: it records the relevant state, updates the local $csid$, and asynchronously notifies the controller of completion. For outgoing packets, if the destination address falls within the scope of in-group processes, the process updates $pkt.sid$ to its current $csid$.

SLBs: As discussed in §4, Beaver instantiates the gateway overlay with the SLBs. For the set of SLBs handling the target in-group process traffic, Beaver embeds logic for marking inbound messages. On receiving an inbound packet, an SLB first checks if the destination VIP is for the in-group [line 16]—since operators may multiplex a single SLB server for multiple VIPs—and modifies the snapshot ID field accordingly. On the control path, the SLB initializes a new snapshot upon receiving an 'INIT' notification from the controller and subsequently sends the acknowledgment to the controller. This process happens out-of-band to avoid biases in the snapshot verification process. Combined, Beaver's

[8]Optional broadcast of marker messages from SLBs to in-group processes may accelerate the snapshot convergence when service traffic is infrequent.

```
• $csid$: Current snapshot ID state for $p \in P^{in}$ or $g \in G$.
  – $pkt.sid$: Snapshot ID ($N$b) in SLB encapsulation header.
  – $pkt.dst$: Destination address of a user packet.
  – $pkt.src$: Source address of a user packet.
1  function IN-OnReceive (pkt):
2      /* Signaled a new snapshot */
3      if $pkt.sid > csid$ then
4          Record the state of interest;
5          Send FIN for $csid + 1, \ldots, pkt.sid$ to the controller;
6          $csid \leftarrow pkt.sid$;
7  function IN-OnSend (pkt):
8      if $pkt.dst \in P^{in}$ then
9          $pkt.sid \leftarrow csid$;
10 function SLB-OnReceive (INIT):
11     if $INIT.sid > csid$ then
12         $csid \leftarrow INIT.sid$;
13         ACK for $csid + 1, \ldots, pkt.sid$ to the controller;
14 function SLB-OnReceive (pkt):
15     /* Mark inbound packet from out-group */
16     if $(pkt.dst \in P^{in}) \wedge (pkt.src \notin P^{in})$ then
17         $pkt.sid \leftarrow csid$;
18     Forward packet to $pkt.dst$;
```

Figure 7: Logic for partial snapshots at in-group processes and SLBs. All control plane operations are asynchronous.

gateway logic requires minimal processing and can be incorporated into existing SLB data planes at line rate, including hardware-accelerated ones.

Controller: With Beaver, operators can designate any server with direct or indirect access to a stable clock source, preferably located near the pertinent SLBs, as the controller. The core logic to initiate snapshots, shown in Figure 8, involves continuously sending INIT commands to SLBs to initiate new snapshots. The protocol maintains the number of snapshots in flight and controls the snapshot frequency. The detection of invalid snapshots follows the methodology outlined in §4.2: The controller queries the clock read for t_0 before sending notifications [line 5] and uses the clock reads upon receiving the last ACK to determine the snapshot's validity [line 20]. It the local NIC supports hardware time-stamping capabilities, `queryClock()` can occur along the data path during the send of the first INIT notification and the receive of the last ACK response.

Handling packet loss, delay, and reordering. Beaver is robust to faults in data- and control-plane communications.

Data plane: Unlike the original Chandy-Lamport protocol, which relies on separate marker messages, Beaver draws inspiration from subsequent variants [34, 60] to incorporate marker information by piggybacking it into existing traffic. This piggybacking makes Beaver inherently resilient to 'marker' losses and reordering on the data path, whether these occur within the network core or the host networking stacks.

Control plane: Although timely and reliable delivery of control messages can be beneficial (e.g., through an alternate port that is dedicated to control tasks) Beaver does not de-

```
• $csid$: The next snapshot ID to initiate at the controller.
• $receivedFIN[sid][p]$: If received FIN from $p \in P^{in}$ for $sid$.
• $receivedACK[sid][g]$: If received FIN from $g \in G$ for $sid$.
• $t_0[sid]$: Timestamp $t_0$ for $sid$.
– $FIN.p$: The source process sending the FIN.
– $FIN.sids$: The associated $sid$(s) of the FIN.
– $ACK.g$: The source SLB sending the FIN.
– $ACK.sids$: The associated $sid$(s) of the ACK.
1  function Controller-OnSnapshot():
2      $num\_inflight\_ss = 0$, $csid = 0$;
3      while $num\_inflight\_ss < 2^{N-1} - 1$ do
4          /* Optional rate-limiting for less greedy snapshots */
5          $t_0[csid] = queryClock()$;
6          Send $INIT$s ($INIT.sid = csid$) to all $g \in G$;
7          $num\_inflight\_ss$ += 1, $csid$ += 1;
8  function Controller-OnReceive(FIN):
9      for $sid \in FIN.sids$ do
10         $receivedFIN[sid][FIN.p] = 1$;
11         /* Check all FINs received with bitwise negation */
12         if $\sim receivedFIN[sid][\cdot] == 0$ then
13             $num\_inflight\_ss$ −= 1;
14             $receivedFIN[sid][\cdot] = 0$;
15 function Controller-OnReceive(ACK):
16     for $sid \in ACK.sids$ do
17         $receivedACK[sid][ACK.g] = 1$;
18         /* If all ACKs received */
19         if $\sim receivedACK[sid][\cdot] == 0$ then
20             if $(queryClock() - t_0[sid])(1 + \Delta f) < \tau_{min}$ then
21                 /* Accept the snapshot */
22             else
23                 /* Invalidate the snapshot */
24             $receivedACK[sid][\cdot] = 0$;
```

Figure 8: Main controller logic for continuous snapshots.

pend on it for its core functionality. It operates effectively even with unreliable transport protocols such as UDP and it requires only a negligible number of control messages: $|P^{in}|$ FIN messages (or less as members of the in-group, P^{in}, can batch updates in a single ACK on the increments in prior snapshots), $|G|$ INIT commands, and $|G|$ ACK responses for each snapshot.

While delays or losses of the above messages might slow down the snapshot rate—a minimal impact as observed in our evaluation—they do not compromise the correctness of Beaver. The controller, in response to any delays or losses, simply invalidates the affected snapshot.

Handling failures. One important problem is how to handle failures of the SLBs and backend servers. Fortunately, most public clouds today already apply central management mechanisms that ensure fault tolerance and state consistency during changes in membership of machines for each VIP[9] [10, 15, 22, 50]. Operating on top of the abstraction, Beaver's controller coordinates with the SLBs and backend servers belonging to the requested VIP (as indicated by the current central state), incurring minimal additional costs and deployment complexity. To handle failure events during a snapshot, Beaver incorporates a single ACK mechanism (Figure 8): if the controller does not receive the ACK from an SLB or an in-group process, Beaver simply invalidates the snapshot or drops affected states while guaranteeing correctness.

[9]Unlike the DIP caching feature in §2.1, the consistency mechanism was originally absent in [50], but later incorporated as an essential component.

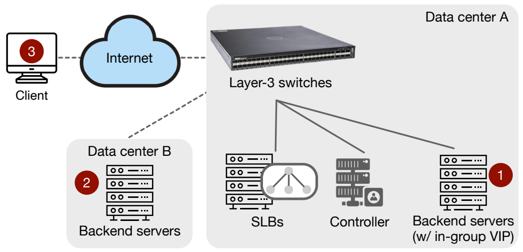


Figure 9: Evaluation setup considering three different out-group locations: within the same data center, data center of a different region, or on the Internet (from a local laptop).

Supporting parallel snapshots. Many cases, such as event-driven or telemetry tasks, require higher-frequency state capture [60, 61]. Rather than waiting for the completion of one snapshot before initiating another, limiting the snapshot rate to the slowest component in the snapshot convergence process, Beaver can initiate snapshots concurrently. The controller ensures that the number of packets in flight remains within $2^N - 1$ [line 3 in Figure 8], the maximum concurrent snapshots supported by the header field *sid*. The extra $^{-1}$ in the exponent is to eliminate ambiguities in comparator operations at in-group processes [line 3 in Figure 7] under worst-case wrap-around conditions.

Beaver also supports parallel snapshots for distinct groups of VIPs without needing extra metadata. This is facilitated by the SLBs' ability to naturally segregate operations based on VIP information. Consequently, the same *sid* header space can be utilized for simultaneous snapshots across groups with non-overlapping VIPs.

6 Implementation

We implement a Beaver prototype on a cloud data center [14] (Figure 9) that aligns with a production setup [15, 28, 50].

Supporting SLB-associated functionalities. We implement an end-to-end workflow to mirror the behaviors associated with SLBs in production data centers [15, 28, 50]. Additionally, our system facilitates automated service discovery operations through an out-of-band controller server.

SLB implementation: Our setup configures DELL EMC PowerSwitch S4048-ON [1] for layer-3 ECMP forwarding based on service VIPs to SLBs. Emulating prior work [15, 28], we implement the core SLB functions with DPDK [19], involving around 1860 lines of C/C++ code. Each SLB maintains an in-memory connection flow state, employs consistent hashing on the 5-tuple of each packet to determine the appropriate backend server, and caches the decision for future decisions. Then, the SLBs encapsulate the inbound packet's header and forward it to the backend server with the destination DIP. To maximize utilization of SLB servers, we perform load balancing across different CPU cores using RSS.

Backend servers: To maintain transparency for the upper-layer applications, we implement the re-computation of checksums, NAT caching in a shared eBPF map, and the de-encapsulation of incoming packets from the SLB via XDP [27]. For outbound packets, we instrument the Linux `tc` to look up the NAT entries and perform the header transformations to replicate Direct Server Return (DSR). In total, they involve 1040 lines of C/C++ code.

Topology. Our testbed supports typical communication patterns, encompassing a variety of out-group positions, including other VIPs within the same data center, VIPs in other data centers, and Internet clients—all through the layer-3 switches and SLBs, along with DSR on the return path. We scale up to 16 SLB servers, each capable of supporting 64 in-group processes, due to limits in resource availability. Our current testbed servers are equipped with Intel(R) Xeon(R) CPU E5-2640 v4 @ 2.40GHz and dual-port ConnectX-4 Lx NICs.

Integrating the Beaver protocol. We implement Beaver's partial snapshot protocol from §5. The SLBs append a snapshot ID to inbound packet headers that encapsulate the destination DIP and the source SLB IP. The in-group processes and SLBs embed Beaver's snapshot logic from Figure 7 through XDP and DPDK. The additional logic involves 68 lines of C++ for SLB data-path logic and 102 lines of C codes for eBPF at in-group processes. The controller server, following Figure 8, automates the initiation, control, collection, and verification of snapshots. We use UDP for bi-directional control messages with SLBs and unidirectional messages from in-group servers. The controller currently exploits local NIC hardware timestamping (`SOF_TIMESTAMPING_RAW_HARDWARE`) for precise timing of INIT and ACK messages on their data path [47].

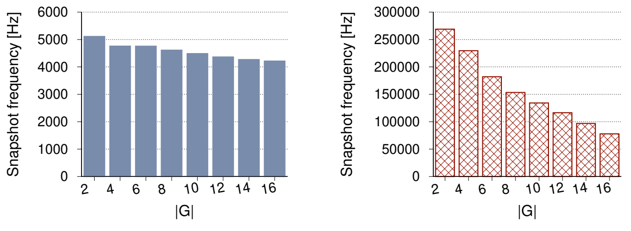
7 Evaluation

Our evaluation focuses on exploring the following questions.

- Can Beaver sustain fast snapshot rates? How does the scale of the in-group nodes and SLBs affect? (§7.1)
- What about effective snapshot rates? How often do Beaver invalidate snapshots in cloud data centers? (§7.2)
- Does Beaver's distributed coordination affect the existing service traffic? (§7.3)
- How does Beaver help real-world services? (§7.4)

7.1 Beaver Supports Fast Snapshot Rates

To stress-test Beaver, unless otherwise specified, our evaluation runs Beaver at very high snapshot frequencies. To further ensure that our performance/overhead results are conservative, state capture in the snapshots are NOPs. Real local record



(a) w/o parallelism (b) w/ parallelism

Figure 10: Beaver's sustained snapshot frequency versus a strawman approach with blocking operations at varying scales of SLBs and backend processes.

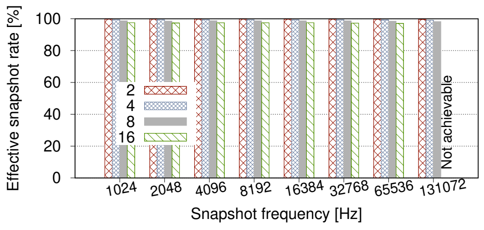


Figure 11: Beaver's effective snapshot rates under varying snapshot frequencies and in-group process scale.

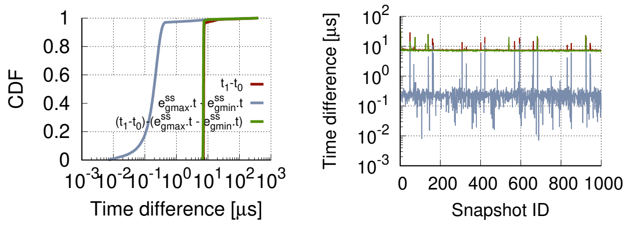
operations (which are application-dependent and orthogonal to the study of distributed snapshot protocols) will only result in less contention and overhead.

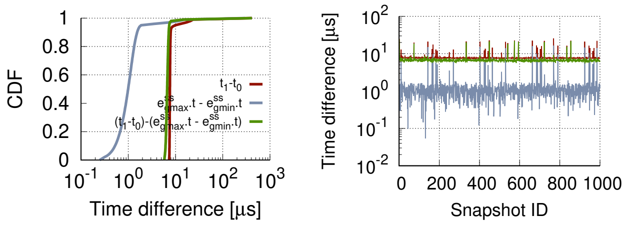
As a measure of Beaver's efficiency and scalability, even at these high rates, Beaver exhibits good performance. Figure 10 shows the maximum snapshot rate compared to a strawman approach, which waits for completion before initiating another. The maximum rate is determined by increasing the snapshot frequency until we observe backlogs in the ACK and INIT message notification queue. We vary the number of gateways ($|G|$) up to 16, aligning with typical values for SLBs assigned to a VIP.

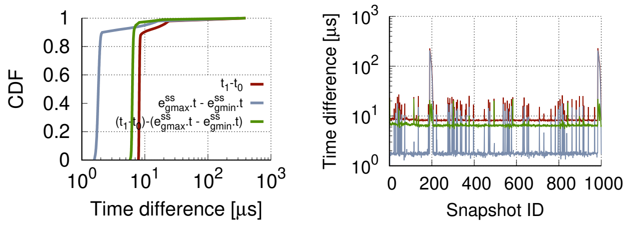
The baseline is limited by the snapshot convergence time, which depends on factors such as scale, traffic pattern, and topology. In contrast, Beaver's parallel snapshot capability significantly enhances the rate and shifts the bottlenecks to the processing power of the controller's CPU. Even at the maximum scale, Beaver reaches a snapshot rate of > 77000 Hz, $> 18\times$ that of the strawman. In practical applications, leveraging a more powerful processor or scaling the controller server could further improve its speed.

7.2 Beaver Invalidates Snapshots Infrequently

With a high snapshot frequency, how does Beaver perform in terms of effective snapshot rates? Recall in §4.2, Beaver uses an upper bound $t_1 - t_0$ for the time gap between SLB initiations ($e^{ss}_{gmax}.t - e^{ss}_{gmin}.t$) to eliminate the need for time synchronization, it invalidates a snapshot if the bound is greater than τ_{min}, the minimum time to for an external causal chain to occur. While this upper bound ensures correctness, it may reject snapshots and reduces the effective snapshot rate.


(a) $|G| = 2, |P^{in}| = 128$


(b) $|G| = 4, |P^{in}| = 256$


(c) $|G| = 8, |P^{in}| = 514$

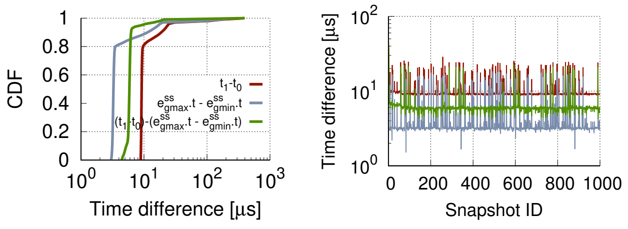

(d) $|G| = 16, |P^{in}| = 1024$

Figure 12: CDF of Beaver's upper bound $t_1 - t_0$ with the ground truth $(e^{ss}_{gmax}.t - e^{ss}_{gmin}.t)$ for > 10M snapshots and a zoom in to its snapshot series, under stressed scenario with 65536 Hz snapshot frequency and varying number of SLBs/processes.

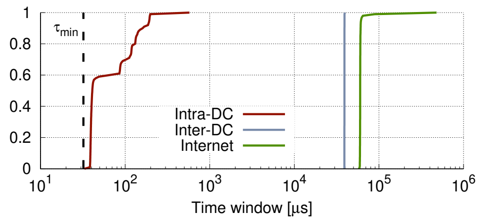


Figure 13: Measurement of the minimum time window for a external causal chain to occur under worst case conditions.

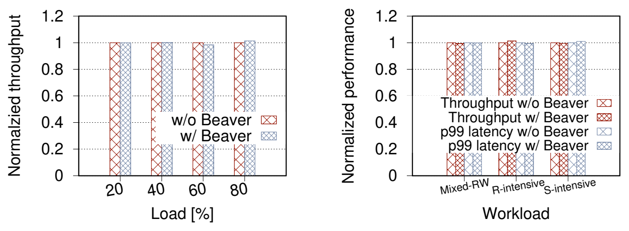
To measure the time for an external causal chain to occur, we consider three distinct scenarios for out-group process locations in Figure 9. In each scenario, we set up a worst-case condition where, immediately following an SLB's snapshot initiation, the SLB forwards an inbound packet to the closest in-group node. The in-group node then loopbacks an immediate message to out-group node with the shortest path, which bounces the packet back to any SLB. Figure 13 shows that the intra-DC scenario results in the shortest time window, resulting in τ_{min} as $33\mu s$. This value is robust because, even though varying cloud conditions often cause latency spikes, they primarily affect the tail rather than the minimum.

To stress test Beaver's performance, we focus on the worst-case scenarios with out-group processs located within the same data center. For other scenarios, τ_{min} is significantly greater, leading to 100% effective snapshot rates across 10M snapshot operations. We execute Beaver in various experimental settings, including scale and snapshot frequencies. For each configuration, we calculate the effective rate based on more than 10 million snapshots. The results, as in Figure 11, reveal that the proportion of snapshots invalidated by Beaver is remarkably low even under the maximum operating frequencies and scales of our testbed.

To better understand the results, we compare the recorded upper bound estimation of $t_1 - t_0$ with the true ground truth $e^{ss}_{gmax}.t - e^{ss}_{gmin}.t$. As the two events e^{ss}_{gmax} and e^{ss}_{gmin} occur on separate SLB machines, we synchronize the clocks of all SLBs to controller's PTP master clock over symmetric paths without contending traffic, which reports maximum 50 ns offsets during the ground truth measurement. This step, meant solely to understand the behavior, should not be confused with Beaver's clock-synchronization-free approach. Figure 12 shows the comparison over > 10M snapshots when Beaver operates at a frequency of 65536 Hz. Overlapping tails of $e^{ss}_{gmax}.t - e^{ss}_{gmin}.t$ and the heads of $t_1 - t_0$ are expected—the cdf of the pairwise calculation of $(t_1 - t_0) - (e^{ss}_{gmax}.t - e^{ss}_{gmin}.t)$ for each snapshot clearly demonstrates that the upper bound is strictly higher than the ground truth SLB initiation time gap. The observed outliers in $t_1 - t_0$ are typically due to queueing in our manager's processing queue at high rates or asynchrony in SLB initiations. Furthermore, the margin introduced by $t_1 - t_0$ over $e^{ss}_{gmax}.t - e^{ss}_{gmin}.t$ is due to the RTT between the controller and the SLBs, which is used to ensure the theoretical upper bound without clock synchronization.

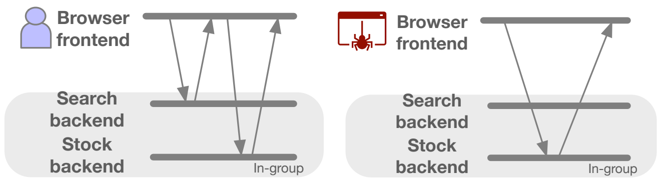
7.3 Beaver Incurs Near-zero Impact

We also stress test the overhead of Beaver on user traffic. Figure 14a compares throughput with and without Beaver under the 65536 Hz snapshot frequency and the max scale of our testbed. `iperf` clients send traffic with varying degree of the total consumed bandwidth capacity of the 16 SLBs. We also run YCSB benchmark workloads [12] with varying mix of read, update, and scan operations, as shown in Figure 14 for backend servers running CassandraDB [6]. The requests follow zipfian distributions, and the scan length adheres to the uniform distribution.



(a) Stressed workloads (b) YCSB benchmarks

Figure 14: Performances with and without Beaver's overhead, normalized to the value without Beaver.



(a) Benign (b) Bot access

Figure 15: Example benign and bot access patterns.

The results of various performance metrics are almost identical, confirming that Beaver has a near-zero effect on service traffic. This is because Beaver, by design, eliminates any delay or blocking operations on the data path for distributed coordination, and the lightweight control path messages are orthogonal.

7.4 Use Cases

We also examine several use cases of Beaver. These examples are intended as instruments through which we can understand its potential utility, differences versus traditional snapshots, and the semantics of its causal consistency guarantee under partial deployments that were previously impossible.

7.4.1 Detecting Anomalous Access

Web applications often feature a JavaScript browser frontend for user interaction and a backend providing service APIs. Consider a legitimate user access in an e-commerce application (Figure 15a). The frontend calls a Search API `fetch("example.com/api/v1/search")`, followed by a Stock API `fetch("example.com/api/v1/get_stock")` for product details. However, malicious traffic, such as web scrapers, might bypass the initial search stage and directly query the stock backend, potentially overwhelming the server. This type of traffic can be challenging to detect as it differs from legitimate traffic in intent rather than content [16, 31].

Beaver can help detect such anomaly patterns, as its partial snapshot can capture the external dependency of these requests, even though it occurs through communication with the Internet. To illustrate, we run a varying mixture of benign and illegal bot clients on our testbed. The backend servers maintain per-client request count in a BPF map through double buffering, so as to 'freeze' the current state through a single switch of the pointer and minimize the impact of blocking local record calls. Table 3 shows detection results calculated against the ground truth. We find that Beaver can accurately recognize the interdependence between the accesses. For example, when all clients are benign, Beaver consistently results in true negatives, aligning with the ground truth. However, a polling-based approach and traditional snapshots (L-Y) can result in false positives due to interpretations of erroneous capture of higher counts at the Search backend than at the Stock backend.

Method	Bot ratio = 0% TP, FP, TN, FN	Bot ratio = 5% TP, FP, TN, FN	Bot ratio = 10% TP, FP, TN, FN
Polling	0, 0.005, 0.995, 0	0.005, 0.062, 0.874, 0.059	0.069, 0.136, 0.666, 0.129
L-Y	0, 0.005, 0.995, 0	0.001, 0.058, 0.886, 0.055	0.011, 0.105, 0.783, 0.101
Beaver	0, 0, 1, 0	0.053, 0, 0.947, 0	0.113, 0, 0.887, 0.001

Table 3: Beaver's detection accuracy versus (1) polling-based approach using time synchronization, and (2) Lai-Yang algorithm, a state-of-the-art global snapshot protocol.

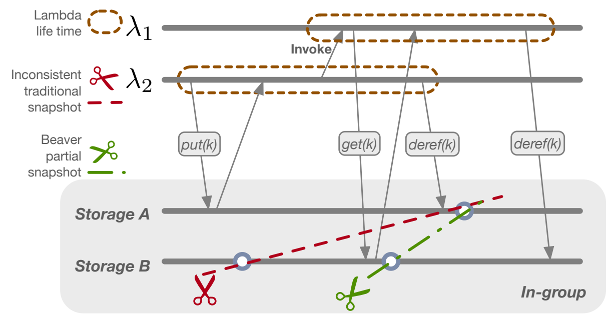


Figure 16: Garbage collection for the ephemeral storage for serverless analytics.

7.4.2 Serverless Garbage Collection

Backend services that support serverless applications are also a natural fit, as requests to serverless functions rely on schedulers and logic that are not visible to the backend services or the serverless functions themselves. Consider an application that provides storage for a serverless analytics job and uses reference counting for garbage collection [32]. The storage service deploys multiple servers for scalability and supports three primary APIs: `get()`/`put()`, which fetch/upload the object and increment the reference counter, and `deref()`, which indicates that the previously fetched object is no longer in use and decrements the reference counter.

Beaver's consistent partial snapshots can support safe garbage collection decisions. To illustrate, we instantiate two serverless functions through [30] that follow the workflow of Figure 16 on our testbed. The backend storage maintains an in-memory state of reference counters for each KV object. When a reference counter reaches 0 in a snapshot, the controller informs the backends to recycle the correspond-

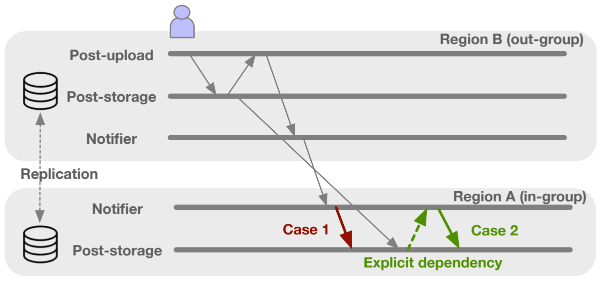


Figure 17: A simplified example of geo-distributed social media application [17] which includes distinct services such as post-upload, post-storage, and notifier.

ing object. During invocations, we also record the incident counts of invalid `get()` access or `deref()` calls. We find that, across invocations, the Lai-Yang algorithm may produce inconsistent snapshots (shown in Figure 16) that indicate *no* open references to the object—however, λ_1 is still keeping a reference to it. This leads to unsafe decisions to recycle the object associated with the key and results in an observed invalid call percentage of 23–29%. In contrast, Beaver's partial snapshots guarantee causal consistency even in the presence of external communication that ensures safe reclamation of the object and consistently results in 0 invalid calls.

7.4.3 Integration Testing

Integration testing, commonly used in CI/CD pipelines [24], extends the coverage of testing to inter-service logic. Unfortunately, applying it to distributed applications can be challenging. Consider the example shown in Figure 17, a violation of the application specification occurs when followers in a region receive a notification and request the storage DB (case 1) before the cross-region protocol actually replicates the post data. Recent solutions [17, 53] address the inconsistencies by forming explicit dependencies (case 2). However, the involvement of auxiliary services and additional dependencies make it difficult to capture a holistic snapshot.

Beaver offers a practical abstraction to test distributed applications by enabling partial deployment and capturing causal dependencies relevant to the local service. By snapshotting states in post-storage and notifier services, developers can write test cases to verify the crucial invariant above: the presence of a post in the storage must always precede its corresponding notification in the notifier service. In particular, Beaver's guarantee of causal consistency means that if a canary solution is correct, a partial snapshot observing a log in the notifier must have captured the data entry of the corresponding version in post-storage. Therefore, a single violating test case will suggest the presence of bugs.

7.4.4 In-flight Message Tracking

We also revisit the example in Figure 3. As mentioned in §2.2, a useful query is to estimate the number of concurrent

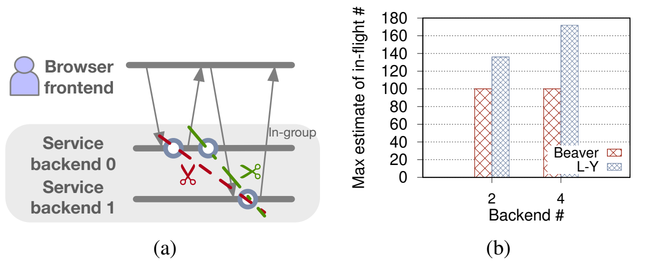


Figure 18: (a) Example snapshots for in-flight message tracking. (b) Comparison of estimated number of in-flight requests with and without Beaver.

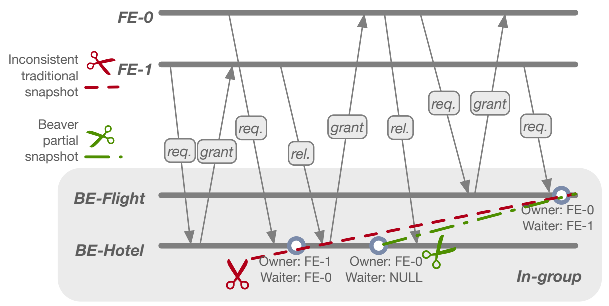


Figure 19: An example of deadlock detection using distributed snapshots.

requests, which can inform resource provision decisions. Figure 18a illustrates a scenario with only one active request. In theory, traditional snapshots, which fail to capture the causality between the client's follow-up request and the prior response, can give an overestimation of 2 in-flight messages (indicated by the cut in red). Beaver, in contrast, can capture the external causality and results in an estimation of no more than 1 message in flight (indicated by the cut in green).

To validate the behavior in practice, we run 100 clients concurrently that conform to the poisson arrival pattern on our testbed. Each backend process maintains a total request and response count value using a BPF map. Thus, the difference between the two counters indicates the number of messages in flight. The controller then collects the snapshot of counter values and then obtains the aggregate estimate. Figure 18b shows that traditional snapshots can overestimate the number of concurrent requests by more than 30%, while Beaver's result consistently matches the ground truth. Worse, a higher number of backends will lead to an overestimation further divorced from reality.

7.4.5 Distributed Deadlock Detection

A classic use of distributed snapshots is deadlock detection, a fundamental problem in distributed systems. Consider the scenario in Figure 19, where the machines of a frontend service interact with a reservation microservice to book flights and hotels on behalf of its clients. Here, a frontend server acquires a lock from the backend server for a target resource ID and

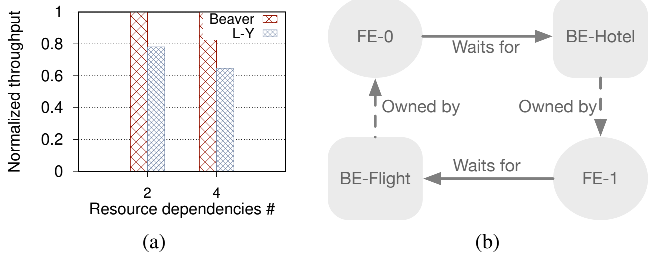


Figure 20: (a) Comparison of transaction throughput (normalized to Beaver). (b) WFG for the inconsistent snapshot in Figure 19.

releases it after completing its transaction. A deadlock may occur when a client requests resources that are held by others, forming a directed cycle in the resource dependency graph (known as Wait-For Graphs or WFGs). As these systems (such as those used by Airbnb and Uber) encompass thousands of microservices, each with its own sovereignty [20,64], global snapshots are challenging and expensive to enforce.

Beaver, however, is amenable to only taking partial snapshots of the reservation service. To illustrate, we run backend processes that maintain the ID list of client(s) currently owning/waiting for the local resources in memory. When the controller detects a deadlock based on a snapshot, it informs backend processes to abort the current transaction. We emulate clients that request backend resources in random order and measure the resulting transaction throughput. Figure 20a shows that the traditional snapshot algorithm can suffer from more than 20% throughput drops compared to Beaver. This is because, without accounting for the external message dependencies, it can render a snapshot that is inconsistent (Figure 19), which leads to false deadlocks (Figure 20b) and the unnecessary costs of deadlock resolution operations. Beaver, on the other hand, guarantees safe detection.

8 Discussion

Instantiating Beaver gateways. Beaver focuses on public clouds, which already contain SLBs, imposing minimal changes and costs to integrate its functionality. We argue that these are where partial snapshots are most important as smaller private clouds are easier to modify wholesale [51]. Without cloud providers' support, cloud tenants could also deploy their own Beaver-compatible gateways on virtual machines (e.g., Network Virtual Appliances (NVAs) [7]) to ensure consistency under external communication with clients and human users. This involves additional costs and complexities and can be suitable if NVAs are already in use, e.g., to provide firewall functionality.

Optimizing local record operations. Similar to classic distributed snapshot protocols (§2.2), Beaver is agnostic to the semantics of local record operations. An interesting problem—orthogonal to the core mechanism of Beaver—is to enable efficient local-state capturing mechanisms, especially when the user desires a large target state or a high snapshot frequency. Besides application-specific practices in §7.4, we postulate that a more generic and opportunistic approach may minimize their online impacts by focusing on state changes during IDLE times of the application. We leave a complete exploration for future work.

9 Related Work

Distributed snapshots. This work builds on the large array of classic distributed snapshot algorithms [11,26,33,34,41,56,57,60]. To the best of our knowledge, Beaver formalizes, designs, and implements the first partial snapshot primitive that extends their capabilities for practical usage.

Cloud data centers. Beaver is also related to works on various facets of cloud data centers, including layer-4 load balancers [10,15,22,43,50,63] and its clock services [13,23,25,36,38,42,44,48,59,62]. For the former, Beaver integrates its gateway marking logic based on the behaviors of SLBs fundamental to cloud data center services and implements a practical prototype aligned with today's setups. Meanwhile, Beaver builds on extensive measurement studies that highlight the reliable properties of frequency drifts of a single clock. Combined, Beaver presents a unique design without making any assumptions about clock synchronization that ensures consistent, high-rate partial snapshots under external interactions while incurring minimal changes and impacts to current operations and service traffic.

10 Conclusion

This paper rethinks the classic distributed snapshots and observes the mismatch of their assumptions with today's cloud services. With it, we present Beaver, the first partial snapshot primitive that advances the capabilities of existing snapshots for practical usage in distributed cloud services. Central to Beaver is the design and instantiation of a novel optimistic gateway marking primitive. Beaver presents a unique design point by tightly integrating the protocol with the regularities of data center networks. Our evaluation demonstrates that Beaver not only can capture partial snapshots at high speed, but it also incurs near-zero costs to existing service traffic.

Acknowledgments

We gratefully acknowledge our shepherd Nitin Agrawal and the OSDI 2024 reviewers for all their help and constructive comments. We also thank Behnaz Arzani, Linh Thi Xuan Phan, Wyatt Lloyd, and João Ferreira Loff for the valuable feedback. This work was funded in part by NSF grants CNS-1845749, CCF-2124184, and CNS-2321726.

References

[1] Dell emc powerswitch s4048-on support page. https://www.dell.com/support/home/en-us/product-support/product/force10-s4048-on/docs.

[2] Hugging face models. https://huggingface.co/models.

[3] Apache kafka. https://kafka.apache.org/, 2024.

[4] Remzi Can Aksoy and Manos Kapritsos. Aegean: replication beyond the client-server model. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 385–398, 2019.

[5] Apache Flink. Apache flink: Stateful computations over data streams. https://flink.apache.org/, 2024.

[6] Apache Software Foundation. Apache cassandra. https://cassandra.apache.org/, 2021.

[7] Aviatrix. Azure network virtual appliance. https://aviatrix.com/learn-center/cloud-security/azure-network-virtual-appliance/, 2024.

[8] Peter Bailis, Alan Fekete, Ali Ghodsi, Joseph M Hellerstein, and Ion Stoica. The potential dangers of causal consistency and an explicit solution. In *Proceedings of the Third ACM Symposium on Cloud Computing*, pages 1–7, 2012.

[9] Deepak Bansal, Gerald DeGrace, Rishabh Tewari, Michal Zygmunt, James Grantham, Silvano Gai, Mario Baldi, Krishna Doddapaneni, Arun Selvarajan, Arunkumar Arumugam, et al. Disaggregating stateful network functions. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 1469–1487, 2023.

[10] Tom Barbette, Chen Tang, Haoran Yao, Dejan Kostić, Gerald Q Maguire Jr, Panagiotis Papadimitratos, and Marco Chiesa. A high-speed load-balancer design with guaranteed per-connection-consistency. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 667–683, 2020.

[11] K Mani Chandy and Leslie Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Transactions on Computer Systems (TOCS)*, 3(1):63–75, 1985.

[12] Brian Frank Cooper et al. Yahoo! cloud serving benchmark (ycsb), 2021. Available at https://github.com/brianfrankcooper/YCSB.

[13] James C Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, Jeffrey John Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, et al. Spanner: Google's globally distributed database. *ACM Transactions on Computer Systems (TOCS)*, 31(3):1–22, 2013.

[14] Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, Aditya Akella, Kuangching Wang, Glenn Ricart, Larry Landweber, Chip Elliott, Michael Zink, Emmanuel Cecchet, Snigdhaswin Kar, and Prabodh Mishra. The design and operation of CloudLab. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, pages 1–14, July 2019.

[15] Daniel E Eisenbud, Cheng Yi, Carlo Contavalli, Cody Smith, Roman Kononov, Eric Mann-Hielscher, Ardas Cilingiroglu, Bin Cheyney, Wentao Shang, and Jinnah Dylan Hosein. Maglev: A fast and reliable software network load balancer. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, pages 523–535, 2016.

[16] Maryam Feily, Alireza Shahrestani, and Sureswaran Ramadass. A survey of botnet and botnet detection. In *2009 Third International Conference on Emerging Security Information, Systems and Technologies*, pages 268–273. IEEE, 2009.

[17] João Ferreira Loff, Daniel Porto, João Garcia, Jonathan Mace, and Rodrigo Rodrigues. Antipode: Enforcing cross-service causal consistency in distributed applications. In *Proceedings of the 29th Symposium on Operating Systems Principles*, pages 298–313, 2023.

[18] Daniel Firestone, Andrew Putnam, Sambhrama Mundkur, Derek Chiou, Alireza Dabagh, Mike Andrewartha, Hari Angepat, Vivek Bhanu, Adrian Caulfield, Eric Chung, et al. Azure accelerated networking: Smartnics in the public cloud. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 51–66, 2018.

[19] Linux Foundation. Data plane development kit (DPDK), 2015.

[20] Yu Gan, Yanqi Zhang, Dailun Cheng, Ankitha Shetty, Priyal Rathi, Nayan Katarki, Ariana Bruno, Justin Hu, Brian Ritchken, Brendon Jackson, et al. An open-source benchmark suite for microservices and their hardware-software implications for cloud & edge systems. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 3–18, 2019.

[21] Rohan Gandhi, Y. Charlie Hu, Cheng kok Koh, Hongqiang (Harry) Liu, and Ming Zhang. Rubik: Unlocking the power of locality and end-point flexibility in cloud scale load balancing. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*, pages 473–485, Santa Clara, CA, July 2015. USENIX Association.

[22] Rohan Gandhi, Hongqiang Harry Liu, Y Charlie Hu, Guohan Lu, Jitendra Padhye, Lihua Yuan, and Ming Zhang. Duet: Cloud scale load balancing with hardware and software. *ACM SIGCOMM Computer Communication Review*, 44(4):27–38, 2014.

[23] Yilong Geng, Shiyu Liu, Zi Yin, Ashish Naik, Balaji Prabhakar, Mendel Rosenblum, and Amin Vahdat. Exploiting a natural network effect for scalable, fine-grained clock synchronization. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 81–94, 2018.

[24] Boris Grubic, Yang Wang, Tyler Petrochko, Ran Yaniv, Brad Jones, David Callies, Matt Clarke-Lauer, Dan Kelley, Soteris Demetriou, Kenny Yu, et al. Conveyor: One-tool-fits-all continuous software deployment at meta. In *Proceedings of the 17th USENIX Symposium on Operating Systems Design and Implementation*, 2023.

[25] Eashan Gupta, Prateesh Goyal, Ilias Marinos, Chenxingyu Zhao, Radhika Mittal, and Ranveer Chandra. Dbo: Fairness for cloud-hosted financial exchanges. In *Proceedings of the ACM SIGCOMM 2023 Conference*, pages 550–563, 2023.

[26] Jean-Michel Helary. Observing global states of asynchronous distributed applications. In *Distributed Algorithms: 3rd International Workshop Nice, France, September 26–28, 1989 Proceedings 3*, pages 124–135. Springer, 1989.

[27] Toke Høiland-Jørgensen, Jesper Dangaard Brouer, Daniel Borkmann, John Fastabend, Tom Herbert, David Ahern, and David Miller. The express data path: Fast programmable packet processing in the operating system kernel. In *Proceedings of the 14th international conference on emerging networking experiments and technologies*, pages 54–66, 2018.

[28] Facebook Incubator. Katran: A high performance layer 4 load balancer. https://github.com/facebookincubator/katran, 2023.

[29] IQD Frequency Products. Iqrb-1 high-performance rubidium oscillators. https://www.digikey.com/en/product-highlight/i/iqd-frequency-products/iqrb-1-high-performance-rubidium-oscillators.

[30] Zhipeng Jia and Emmett Witchel. Nightcore: Efficient and scalable serverless computing for latency-sensitive, interactive microservices. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS 2021, page 152–166, New York, NY, USA, 2021. Association for Computing Machinery.

[31] Srikanth Kandula, Dina Katabi, Matthias Jacob, and Arthur Berger. Botz-4-sale: Surviving organized ddos attacks that mimic flash crowds. In *Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation-Volume 2*, pages 287–300, 2005.

[32] Ana Klimovic, Yawen Wang, Patrick Stuedi, Animesh Trivedi, Jonas Pfefferle, and Christos Kozyrakis. Pocket: Elastic ephemeral storage for serverless analytics. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 427–444, 2018.

[33] Ajay D Kshemkalyani, Michel Raynal, and Mukesh Singhal. An introduction to snapshot algorithms in distributed computing. *Distributed systems engineering*, 2(4):224, 1995.

[34] Ten H Lai and Tao H Yang. On distributed snapshots. *Information Processing Letters*, 25(3):153–158, 1987.

[35] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. In *Concurrency: the Works of Leslie Lamport*, pages 179–196. 2019.

[36] Ki Suh Lee, Han Wang, Vishal Shrivastav, and Hakim Weatherspoon. Globally synchronized time via datacenter networks. In *Proceedings of the 2016 ACM SIGCOMM Conference*, pages 454–467, 2016.

[37] Yiran Lei, Liangcheng Yu, Vincent Liu, and Mingwei Xu. Printqueue: performance diagnosis via queue measurement in the data plane. In *Proceedings of the ACM SIGCOMM 2022 Conference*, pages 516–529, 2022.

[38] Yuliang Li, Gautam Kumar, Hema Hariharan, Hassan Wassel, Peter Hochschild, Dave Platt, Simon Sabato, Minlan Yu, Nandita Dukkipati, Prashant Chandra, et al. Sundial: Fault-tolerant clock synchronization for datacenters. In *14th USENIX symposium on operating systems design and implementation (OSDI 20)*, pages 1171–1186, 2020.

[39] Yunhe Liu, Nate Foster, and Fred B Schneider. Causal network telemetry. In *Proceedings of the 5th International Workshop on P4 in Europe*, pages 46–52, 2022.

[40] Jennifer Lundelius and Nancy Lynch. An upper and lower bound for clock synchronization. *Information and control*, 62(2-3):190–204, 1984.

[41] Friedemann Mattern. Efficient algorithms for distributed snapshots and global virtual time approximation. *Journal of parallel and distributed computing*, 18(4):423–434, 1993.

[42] Meta. Precision time protocol at meta. https://engineering.fb.com/2022/11/21/production-engineering/precision-time-protocol-at-meta/, November 2022.

[43] Rui Miao, Hongyi Zeng, Changhoon Kim, Jeongkeun Lee, and Minlan Yu. Silkroad: Making stateful layer-4 load balancing fast and cheap using switching asics. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, pages 15–28, 2017.

[44] Ali Najafi and Michael Wei. Graham: Synchronizing clocks by leveraging local clock properties. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 453–466, 2022.

[45] Travis L Nicholson, SL Campbell, RB Hutson, G Edward Marti, BJ Bloom, Rees L McNally, Wei Zhang, MD Barrett, Marianna S Safronova, GF Strouse, et al. Systematic evaluation of an atomic clock at $2\times$ 10- 18 total uncertainty. *Nature communications*, 6(1):6896, 2015.

[46] NVIDIA Corporation. Connectx-6 dx ethernet adapter cards datasheet. https://www.nvidia.com/content/dam/en-zz/Solutions/networking/ethernet-adapters/connectX-6-dx-datasheet.pdf.

[47] NVIDIA Corporation. Time stamping - nvidia networking docs. https://docs.nvidia.com/networking/display/ofedv502180/time-stamping.

[48] Open Compute Project. Time appliance project. https://opencomputeproject.github.io/Time-Appliance-Project/.

[49] Tian Pan, Nianbing Yu, Chenhao Jia, Jianwen Pi, Liang Xu, Yisong Qiao, Zhiguo Li, Kun Liu, Jie Lu, Jianyuan Lu, Enge Song, Jiao Zhang, Tao Huang, and Shunmin Zhu. Sailfish: Accelerating cloud-scale multi-tenant multi-service gateways with programmable switches. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, SIGCOMM '21, page 194–206, New York, NY, USA, 2021. Association for Computing Machinery.

[50] Parveen Patel, Deepak Bansal, Lihua Yuan, Ashwin Murthy, Albert Greenberg, David A Maltz, Randy Kern, Hemant Kumar, Marios Zikos, Hongyu Wu, et al. Ananta: Cloud scale load balancing. *ACM SIGCOMM Computer Communication Review*, 43(4):207–218, 2013.

[51] Harshit Saokar, Soteris Demetriou, Nick Magerko, Max Kontorovich, Josh Kirstein, Margot Leibold, Dimitrios Skarlatos, Hitesh Khandelwal, and Chunqiang Tang. {ServiceRouter}: Hyperscale and minimal cost service mesh at meta. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, pages 969–985, 2023.

[52] Reinhard Schwarz and Friedemann Mattern. Detecting causal relationships in distributed computations: In search of the holy grail. *Distributed computing*, 7(3):149–174, 1994.

[53] Xiao Shi, Scott Pruett, Kevin Doherty, Jinyu Han, Dmitri Petrov, Jim Carrig, John Hugg, and Nathan Bronson. Flighttracker: Consistency across read-optimized online stores at facebook. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 407–423, 2020.

[54] Ticket News. Ontario man 'heartbroken' after ticketmaster cancels 'double-sold' seat. https://www.ticketnews.com/2018/03/paul-simon-fan-scored-floor-seats-had-them-revoked-by-ticketmaster-after-seat-was/, 2018.

[55] Peter Van Roy and Angel Bravo Gestoso. Saturn: A distributed metadata service for causal consistency. In *EuroSys 2017 Conference*, 2017.

[56] Maarten van Steen and Andrew S. Tanenbaum. *Distributed Systems*. distributed-systems.net, 3 edition, 2017.

[57] S Venkatesan. Message-optimal incremental snapshots. In *Proceedings. The 9th International Conference on Distributed Computing Systems*, pages 53–54. IEEE Computer Society, 1989.

[58] Nofel Yaseen, Behnaz Arzani, Ryan Beckett, Selim Ciraci, and Vincent Liu. Aragog: Scalable runtime verification of shardable networked systems. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 701–718. USENIX Association, November 2020.

[59] Nofel Yaseen, Behnaz Arzani, Krishna Chintalapudi, Vaishnavi Ranganathan, Felipe Frujeri, Kevin Hsieh, Daniel S Berger, Vincent Liu, and Srikanth Kandula. Towards a cost vs. quality sweet spot for monitoring networks. In *Proceedings of the Twentieth ACM Workshop on Hot Topics in Networks*, pages 38–44, 2021.

[60] Nofel Yaseen, John Sonchack, and Vincent Liu. Synchronized network snapshots. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, pages 402–416, 2018.

[61] Liangcheng Yu, John Sonchack, and Vincent Liu. Mantis: Reactive programmable switches. In *Proceedings of the 2020 ACM SIGCOMM 2021 Conference*, pages 296–309, 2020.

[62] Liangcheng Yu, John Sonchack, and Vincent Liu. Orbweaver: Using idle cycles in programmable networks for opportunistic coordination. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 1195–1212, 2022.

[63] Chaoliang Zeng, Layong Luo, Teng Zhang, Zilong Wang, Luyang Li, Wenchen Han, Nan Chen, Lebing Wan, Lichao Liu, Zhipeng Ding, et al. Tiara: A scalable and efficient hardware acceleration architecture for stateful layer-4 load balancing. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 1345–1358. USENIX Association, 2022.

[64] Zhizhou Zhang, Murali Krishna Ramanathan, Prithvi Raj, Abhishek Parwal, Timothy Sherwood, and Milind Chabbi. Crisp: Critical path analysis of large-scale microservice architectures. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, pages 655–672, 2022.