Timestamp as a Service, not an Oracle

Yishuai Li
Alibaba Cloud
Shanghai, China
liyishuai.lys@alibaba-inc.com

Yunfeng Zhu
Alibaba Cloud
Hangzhou, China
yunfeng.zyf@alibaba-inc.com

Chao Shi
Alibaba Cloud
Beijing, China
chao.shi@alibaba-inc.com

Guanhua Zhang
Alibaba Cloud
Hangzhou, China
sengquan.zgh@alibaba-inc.com

Jianzhong Wang
Alibaba Cloud
Hangzhou, China
zelu.wjz@alibaba-inc.com

Xiaolu Zhang
Alibaba Cloud
Beijing, China
xiaoluzhang.zxl@alibaba-inc.com

ABSTRACT

We present a logical timestamping mechanism for ordering transactions in distributed databases, eliminating the single point of failure (SPoF) that bother existing timestamp "oracles". The main innovation is a bipartite client–server architecture, where the servers do not communicate with each other. The result is a highly available timestamping "service" that guarantees the availability of timestamps, unless half the servers are down at the same time.

We study the fundamental needs of timestamping, and formalize its availability and correctness properties in a distributed setting. We then introduce the TaaS (timestamp as a service) algorithm, which defines a monotonic spacetime over multiple server clocks. We prove, mathematically: (i) *Availability* that the timestamps are always computable, provided any majority of the server clocks being observable; and (ii) *Correctness* that all the computed timestamps must increase monotonically over time, even if some clocks become unobservable.

We evaluate our algorithm by prototyping TaaS and benchmarking it against state of the art timestamp oracle in TiDB. Our experiment shows that TaaS is indeed immune to SPoF (as we have proven mathematically), while exhibiting a reasonable performance at the same order of magnitude with TiDB. We also demonstrate the stability of our bipartite architecture, by deploying TaaS across datacenters and showing its resilience to datacenter-level failures.

PVLDB Reference Format:
Yishuai Li, Yunfeng Zhu, Chao Shi, Guanhua Zhang, Jianzhong Wang, and Xiaolu Zhang. Timestamp as a Service, not an Oracle. PVLDB, 17(5): 994 - 1006, 2024.
doi:10.14778/3641204.3641210

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://zenodo.org/doi/10.5281/zenodo.10467611.

This work is licensed under the Creative Commons BY-NC-ND 4.0 International License. Visit https://creativecommons.org/licenses/by-nc-nd/4.0/ to view a copy of this license. For any use beyond those covered by this license, obtain permission by emailing info@vldb.org. Copyright is held by the owner/author(s). Publication rights licensed to the VLDB Endowment.
Proceedings of the VLDB Endowment, Vol. 17, No. 5 ISSN 2150-8097.
doi:10.14778/3641204.3641210

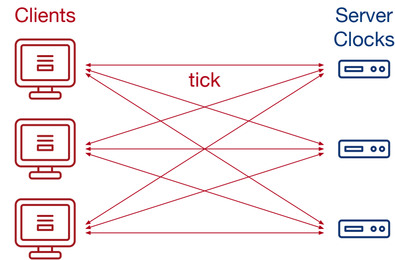


Figure 1: Bipartite architecture of TaaS.

1 INTRODUCTION

1.1 Motivation towards a timestamping service

1.1.1 Monotonicity. Distributed databases are concurrent in nature, processing transactions in parallel to exploit scalability. The transactions should be processed in a correct order, for atomicity, consistency, and isolation, so called "concurrency control" [12].

To order transactions in a way that meets the users' intention, an intuitive way is to timestamp them with logical clocks [14], which extends the precedence relation "$\prec$" among transactions into a consistent total ordering "$\leq$" among timestamps.

1.1.2 Simplicity. The vanilla logical clock by Lamport [14] requires carrying timestamps along every inter-process communication. In other words, such clocks only capture causality by the "passing of timestamps", but not the "flowing of time", making them impractical for distributed systems, whose participants may communicate via arbitrary backchannels.

To avoid modifying all the backchannels just for adapting with logical clocks, industrial systems may deploy a centralized logical clock that serves as the "wall time": If transaction τ_1 precedes τ_2, then the timestamp assigned for τ_1 is smaller than that of τ_2.

1.1.3 Availability. The centralizd clock should be *fault-tolerant*. For example, PolarDB-X [3] backs up its timestamp oracle (TSO) with a Raft cluster; and OceanBase [26] utilizes Paxos. If the leader clock becomes irresponsive, then the cluster re-elects a new leader to continue serving timestamps.

However, the leader-based TSO exhibits blackout periods—i.e., the cluster cannot serve any timestamp during the re-election period, until the new leader takes office. Such single point of failure (SPoF) makes the TSO an "oracle", rather than a "service".

1.2 Contribution

This paper presents *Timestamp as a Service* (TaaS), a distributed algorithm that computes logical timestamps from a consensusless cluster of clocks. The idea is to observe "multiple clocks" simultaneosly, rather than focusing on "a distinguished clock".

As shown in Figure 1, our timestamp service consists of clients and servers that are bipartite—without client–client or server–server communications. The server clocks work independently without synchronizing with each other. The clients compute timestamps by ticking multiple server clocks.

The result is a highly availabile timestamping service that is immune to SPoF: Suppose the cluster consists of $N = 2M - 1$ clock servers, then any M servers being "up" (i.e., responsive to client ticks) guarantees the client to get a timestamp.

In addition to the high availability, the TaaS algorithm also features low latency: If all the N servers are up, then the client can get the timestamp within 1RTT (round-trip time) to the farthest server. If some servers are down or too slow, then the latency is no greater than 2RTT to the median (i.e., M-th nearest) server.

This paper is structured as follows: §2 discusses how today's databases timestamp transactions and live with the limitations of "timestamp oracles". We introduce the TaaS theory in §3, and discuss how to apply the theory to real-world practices in §4. We prototype and evaluate the TaaS algorithm in §5, simulating failures of servers or the entire datacenter. We discuss related and future works in §6, and conclude in §7.

2 STATE OF THE ART

2.1 Clocks in distributed systems

The concept of timing distributed computations was first formalized by Lamport [14], and developed into four flavors of clocks that suit different scenarios:

2.1.1 Scalar logical clocks. The original logical clock by Lamport [14] linearizes the "happen before" relation $\prec$ into a total order $\leq$ or scalar timestamps, and exhibits *completeness* when ticked properly:

Definition 1 (Timestamp completeness). A timestamping mechanism is *complete* if: For all pairs of events where σ "precedes" τ, the timestamp assigned to σ is less than that assigned to τ:

$$\forall\sigma, \forall\tau, (\sigma \prec \tau \implies \text{timestamp of } \sigma < \text{timestamp of } \tau)$$

2.1.2 Vector logical clocks. Mattern [16] extends the scalar clocks into vector clocks that yields a lattice with partial order $<$, achieving *soundness* in addition to completeness:

Definition 2 (Timestamp soundness). A timestamping mechanism is *sound* if: For all pairs of events σ and τ, if the timestamp assigned to σ is less than that assigned to τ, then σ must "precede" τ:

$$\forall\sigma, \forall\tau, (\text{timestamp of } \sigma < \text{timestamp of } \tau \implies \sigma \prec \tau)$$

2.1.3 Hybrid logical clocks (HLC). Kulkarni et al. [13] implement scalar logical clocks with the system's crystal oscillator, to associate timestamps with the "time in physical world". Such physical and logical timing technique was implemented by CockroachDB [23], MongoDB [25], and YugabyteDB [27], and enables synchronizing transactions across databases.

2.1.4 Synchronized clocks. An alternative to "synchronizing different clocks" is to deploy "one clock that synchronizes all", either physically or logically:

TrueTime. Spanner [4] timestamps transactions by the TrueTime API, where the "timeslave daemon" polls multiple "time masters" equipped with GPS receivers and atomic clocks. The result is a highly accurate timer with uncertainties less than 10 milliseconds.

Centralized timestamping. To avoid the cost of atomic clocks (which aren't cheap enough as of 2023), serveral databases generate timestamps from a centralized "timestamp oracle", e.g., CORFU sequencer [2], PolarDB-X TSO [3], TiDB placement driver [10], Percolator TSO [19], Postgres-XL global transaction manager [20], Omid TO [21], and OceanBase global timestamp service [26].

Centralized timestamping drops the dependency on highly-precise hardware clocks. The lightweight and simple design makes it popular among databases deployed within the same datacenter, or across co-located datacenters—where the round-trip overhead for fetching timestamps does not significantly affect the performance.

Scope. This paper focuses on centralized timestamping. We are especially concerned about the *availability* of the timestamp "oracles", which we further discuss in §2.2.

2.2 Availability of timestamp oracles

As its name suggests, the "centralized" timestamp oracle plays a critical part in the database system: If the oracle is down, then nobody can propose any transactions. Therefore, all timestamp oracles to our knowledge are backed up with a failover cluster organized by consensus—more specifically, leader-based consensus such as Multi-Paxos [15] and Raft [18]—and thus inherit their drawbacks.

2.2.1 Leader, single point of failure. Leader-based timestamp oracles are bottlenecked in both availability and performance:

(1) When the leader crashes or hangs, all existing timestamp oracles stop service, until the cluster re-elects a new leader that continues to serve timestamps. Such blackout period can be tuned with consensus algorithm parameters, but never eliminated in a leader-based setting.
(2) The latency of fetching timestamps depends on the network connection between the client and the leader. If the leader's network stack is overloaded, then the clients might immediately experience downgraded performance.

Goal 1. We want to eliminate the bottleneck, and provide a *leaderless* timestamping service that is resistant to single-point failures.

2.2.2 Performance vs Completeness vs Availability. Apart from the SPoF issue, timestamp oracles face another problem that: Given a leader-based consensus, how to request timestamps from it? (i) "Write through" all timestamps to the quorum, or (ii) Use the leader as a "write back" cache?

"Writing through" bases completeness on the consistency of consensus. The leader propagates every requested timestamp to its followers. Upon re-election, the new leader serves on top of the previously issued timestamp, and thus guarantees completeness.

To reduce the performance overhead of propagating each request, real-world oracles choose the write-back approach (ii), where the

leader allocates a *range* of timestamps, and propagates the allocated range to the followers. Upon client request, the leader may immediately respond with an in-range timestamp, without propagating the response to its followers. When the leader fails, the new leader starts from above the allocated range, i.e., above the upper bound of the old leader's timestamps.

The outperformance of write-back over write-through requires careful maintenance of leader uniqueness [18]. Otherwise, the old and new leaders may serve timestamps simultaneously, which breaks monotonicity. Therefore, every write-back timestamp oracle must implement a lease-based leadership [8]: Before re-election, the cluster must wait for the old leader's lease to expire. This results in a longer blackout period during failover.

Goal 2. We aim for optimal performance and rigorous completeness, while keeping the service continuously available.

These goals motivate us to think outside the "consensus" box, and design a "consensusless" mechanism where the servers work collaboratively (i.e., each contributes a subset to the result) yet independently (i.e., without communicating with other servers).

3 ALGORITHM

This section presents the TaaS algorithm. We begin with the easiest scenario in §3.1, where all the servers are always up and responsive. We then handle fault tolerance in §3.2: if some servers are down, then compute the timestamp from the remaining subset of servers. We prove the correctness and availability of TaaS in §3.3, and illustrate some edge-case examples in §3.4.

3.1 Assuming all the servers are up

3.1.1 Motivation. We first address correctness before tolerating faults, by formalizing the correctness of timestamping services:

Definition 3 (Client sessions). A *session* is a series of client interactions for computing a timestamp. For a session named σ, we say it begins at time Q_σ, and ends at time A_σ (as in Query&Answer).

Here we restrict that each client process executes only one session at a time. A computer may execute multiple sessions simultaneosly, by launching parallel client processes.

Definition 4 (Correctness). A timestamping service is *correct* if it is session-complete: For all pairs of sessions σ and τ, if the end of σ precedes the beginning of τ (written as "$A_\sigma \prec Q_\tau$"), then the result timestamp computed by session σ is less than that of session τ:

$$\forall\sigma, \forall\tau, (A_\sigma \prec Q_\tau \implies \text{result of } \sigma < \text{result of } \tau)$$

Note that sessions σ and τ may be executed by the same client or by different clients. The sessions may or may not depend on each other, which we system designers cannot tell, as the clients may implement arbitrary backchannels for synchronization. Instead, we capture the fact that causality implies ordering of time, and define correctness for all pairs of sessions that are disjoint in time.[1]

Next, we will show how the clients interact with servers and compute the timestamps correctly. At this stage, we assume fast and reliable network connection between the clients and the servers.

[1] Here the definition of "time" may be relative [5]—i.e., the clients and servers needn't be synchronized with any specific clock—as the observation of causality is absolute.

Table 1: Result upon different choices of parameter M.

M	$M@\check{\alpha}$	$M@\check{\beta}$	$M@\check{\gamma}$
1	1	1	2
2	1	2	3
3	2	3	3

3.1.2 Logical clocks. Each server implements a logical clock by Lamport [14], i.e., a logical counter that advances monotonically:

```
TS c; // TS for TimeStamp
TS tick (TS t) {
  c = c ⊕ t;
  return c;
}
```

A client may "tick" a server with a timestamp. When ticked by timestamp t, the server clock advances from c to "$c \oplus t$"—a timestamp greater than the client's input t and greater than the server's previous timestamp c:

$$c \oplus t > c \quad \wedge \quad c \oplus t > t$$

For example, when the timestamps are integers, we may implement $(c \oplus t)$ as $(\max(c, t) + 1)$. Note that our theory does not require the timestamps to be integral, i.e., does not assume that $c \oplus t \geq c + 1$.

3.1.3 Syntax. We assume that timestamps form a total order $\leq$, and define an operation called "M-th smallest": Let Σ be a set of cardinality $\mathbf{card}(\Sigma)$, then the M-th smallest element in Σ (written as "$M@\Sigma$") is no less than M elements of Σ, and no greater than $\mathbf{card}(\Sigma) - M + 1$ elements of Σ:

$$\mathbf{card}\{x \in \Sigma \mid x \leq M@\Sigma\} \quad \geq M$$

$$\mathbf{card}\{x \in \Sigma \mid M@\Sigma \leq x\} \quad \geq \mathbf{card}(\Sigma) - M + 1$$

We also define a bottom timestamp (written as "$\bot$") that is no greater than any timestamp:

$$\forall t, \bot \leq t$$

3.1.4 Semantics. A client begins a session by broadcasting $\bot$ to all servers. It then waits for all the servers to respond. Let Σ be the set of responses from all the servers, then the client concludes with $M@\Sigma$, where M is some parameter shared by all the clients.

For example, consider two clients interacting with three servers in Figure 2. This example consists of three sessions: Client Cv does not launch session γ until the conclusion of session α—i.e., $A_\alpha \prec Q_\gamma$—while client Cw runs session β in parallel.

The result of each session depends on the choice of parameter M, as listed in Table 1. We can see that the M-th smallest response in session α (written as "$M@\check{\alpha}$") is always less than $M@\check{\gamma}$, regardless of what M we choose. Here we sketch such correctness proof that "earlier sessions conclude with smaller timestamps", with underlying lemmas proven in the appendices.

Theorem 1 (Basic correctness). For all pairs of sessions σ and τ, if σ ends earlier than τ begins, then for any choice of parameter M, the M-th smallest response received in session σ is always less than the M-th smallest received in session τ:

$$\forall\sigma, \forall\tau, (A_\sigma \prec Q_\tau \implies \forall M, M@\check{\sigma} < M@\check{\tau})$$

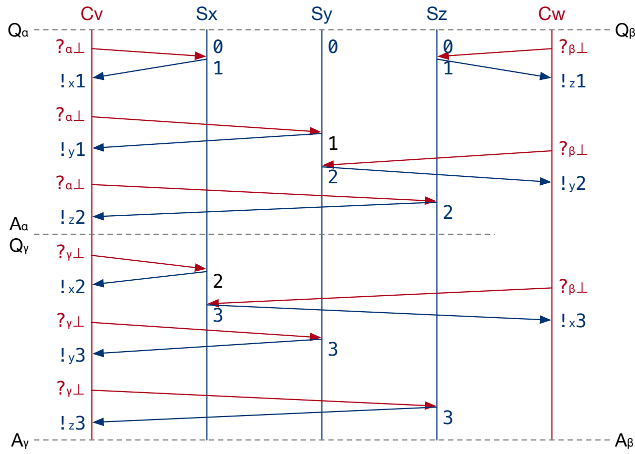


Figure 2: Example where all three servers (Sx, Sy, and Sz) are always up, serving two clients (Cv and Cw) for three sessions (α, β, and γ). Each session begins at time Q and ends at time A. The servers' timelines are annotated with their internal states, i.e., logical timestamps. The clients' timelines record their send and receive events: "$?_\alpha\bot$" is pronounced "sending bottom timestamp in session α"; while "$!_x 1$" means "receiving timestamp 1 from server x".

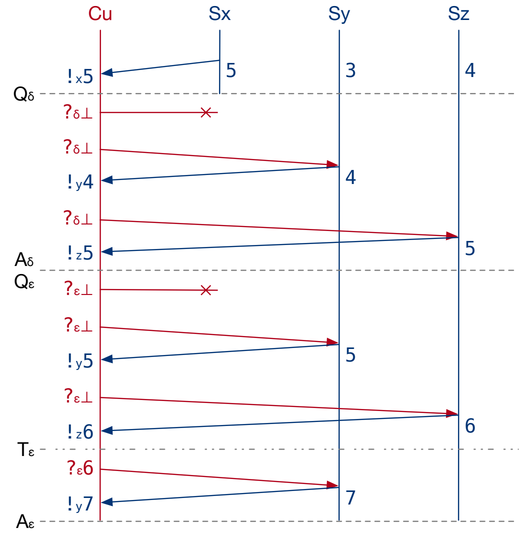


Figure 3: Example where server Sx is down since session δ.

Proof. Let "$M@(\Omega : T)$" be the M-th smallest among the servers' internal states at time T.[2] We then have:

(1) The M-th smallest response received in session σ is less than or equal to the M-th smallest server state at the conclusion of session σ, per Lemma 5 in the appendix:
$$M@\check{\sigma} \leq M@(\Omega : A_\sigma)$$
(2) The M-th smallest server state at the conclusion of session σ is less than or equal to the M-th smallest server state at the start of session τ, per Lemma 6:
$$M@(\Omega : A_\sigma) \leq M@(\Omega : Q_\tau)$$
(3) The M-th smallest state at the start of session τ is less than the M-th smallest response received in session τ, per Lemma 7:
$$M@(\Omega : Q_\tau) < M@\check{\tau}$$

Therefore: $M@\check{\sigma} \leq M@(\Omega : A_\sigma) \leq M@(\Omega : Q_\tau) < M@\check{\tau}$ □

Note that different sessions might conclude with the same result, e.g., $1@\check{\alpha} = 1@\check{\beta} = 1$. In other words, our TaaS kernel is not linearizable to "ticking" a logical clock, but to "looking at" a clock on the wall. To conclude each session with a unique result, we may have the servers to produce disjoint timestamps, with details described in §4.2.

Summary. So far, we have presented the core algorithm of the TaaS client. The main idea is to bound the conclusion of each session σ within the range of $(M@(\Omega : Q_\sigma), M@(\Omega : A_\sigma)]$, which we will revisit in the fault tolerance design in §3.2.

[2] As mentioned in Footnote 1, our theory is relativity-consistent. The "time" here may be relative to any frame of reference that observes $A_\sigma \prec Q_\tau$.

3.2 When some servers are down

The core mechanism in §3.1 assumes the client–server connection to be fast and reliable. Whereas in reality, the requests and responses might be indefinitely delayed en route. This section shows how to tolerate the potential packet losses/delays, and compute timestamps using responses received from a subset of servers.

3.2.1 Motivation. Before showing our fault tolerance algorithm, we exemplify it in Figure 3, with parameter $M = 2$.

Consider session δ: Suppose client Cu remembers a response $!_x 5$ received in a previous session, then the client believes that Sx's response to $?_\delta\bot$ (if it ever arrives) must be greater than 5. Therefore, if the client has observed $!_y 4$ and $!_z 5$ in session δ, then it can safely conclude that $2@\check{\delta} = 5$ without hearing from Sx.

Now suppose server Sx remains irresponsive in session ϵ, and the client has received $!_y 5$ and $!_z 6$. The client cannot assert the response from Sx to be no less than 6 (as our theory doesn't assume the timestamps to be integral), and thus cannot conclude based on its observation up to time T_ϵ.

3.2.2 Analysis. Let's take a step back and ask: Why do we conclude with "the M-th smallest among all the responses"?

Because Theorem 1 bases correctness on the conclusion of each session σ being: (i) greater than $M@(\Omega : Q_\sigma)$; and (ii) no greater than $M@(\Omega : A_\sigma)$. Taking the M-th smallest response guarantees the conclusion within range $(M@(\Omega : Q_\sigma), M@(\Omega : A_\sigma)]$.

Consider session δ: Why may we conclude session δ with 5, without waiting for the response from server Sx?

Because the conclusion is guaranteed: (i) greater than $2@(\Omega : Q_\delta)$—whose upper bound is less than $!_z 5$—needless to hear from Sx; and (ii) no greater than $2@(\Omega : A_\delta)$—whose lower bound is no less than $!_x 5$ and $!_z 5$—based on the monotonicity of logical clocks.

Now let's revisit time T_ϵ: Having observed $!_y 5$ and $!_z 6$, the client is confident that $2@(\Omega : Q_\epsilon) < 6$. It only needs to make sure that $2@(\Omega : A_\epsilon) \geq 6$ to justify 6 as a correct conclusion.

Therefore, to raise the lower bound of $2@(\Omega : A_\epsilon)$ without assuming that Sx would ever respond, the client can advance the state of Sy by sending $?_\epsilon 6$ to it. If Sy responds, then its state must have become greater than 6. Upon hearing from Sy, the client gains confidence that $2@(\Omega : A_\epsilon) \geq 6$, and thus concludes with 6.

3.2.3 Solution. Our algorithm is shown in Figure 4 as pseudo C++. The client is a function that interacts with the server and returns a timestamp. It maintains two mappings from servers to timestamps:

(1) A local mapping $\breve{\sigma}$—pronounced "session"—that stores the smallest response sent by each server for the current session, initiated in Line 4 to "$\top$", a top timestamp (e.g., infinity) that is greater than all other timestamps:
$$\forall t, t \leq \top$$
(2) And a global mapping $\hat{\$}$—pronounced "cache"—that stores the largest response ever sent by each server, regardless of for which session, initiated to $\bot$ in Line 1.

Syntax. The local and global mappings are manipulated during runtime by the `update` function defined in Line 40. Function `update`(f, k, v) maps k to v, and maps all other keys by f:

$$\texttt{update}(f,k,v)(arg) \triangleq \begin{cases} v & arg = k \\ f(arg) & \text{otherwise} \end{cases}$$

Semantics. Each session σ begins by launching the `client()` function in Line 3, and concludes by returning in Line 21.

The client broadcasts $?\bot$ as the first hop, and waits for the servers' responses (by polling the set of `acks`). For each `response` received from server `src`: If it is greater than the cached value, then the client updates $\hat{\$}$ increasingly in Line 36; If it is less than the previous response from `src` in the current session[3] (or trivially less than $\top$ if `src` has not responded yet), then the client updates $\breve{\sigma}$ decreasingly in Line 15.

Candidacy. If the client has received responses from at least M servers—i.e., there exist at least M servers whose local record in $\breve{\sigma}$ were decreased from $\top$ in Line 18—then the client takes $M@\breve{\sigma}$ as a *candidate* conclusion, as it is guaranteed greater than $M@(\Omega : Q_\sigma)$:
$$M@(\Omega : Q_\sigma) < M@\breve{\sigma} = \texttt{candidate}$$
The client considers the candidate "conclusive" if it is no greater than $M@\hat{\$}$ in Line 20. Let "$\texttt{CC}_\sigma$" be the conclusive candidate of session σ, then at the conclusion of σ, there exist at most $M-1$ servers whose states are less than $\texttt{CC}_\sigma$:
$$\texttt{CC}_\sigma \leq M@\hat{\$} \leq M@(\Omega : A_\sigma)$$

Promoting the candidate. If the client has processed all the responses in Line 22 but still finds the `candidate` inconclusive, then it "promotes" the candidate by broadcasting it as a second hop. The second hop only targets "slow" servers—whose cache records are below the candidate—to make $M@\hat{\$}$ catch up with the candidate. After broadcasting the second hop, the client continues to wait for responses to arrive at the input channel, until reaching a conclusion.

If a client cannot conclude within a certain timeout period, then the client may abort the starved session and start a new one, using a fresh local mapping and reusing the current global mapping.

[3]Suppose in a session, a client sends two requests to a server. The server's responses must monotonically increase, but the first response might be delayed en route, and arrive at the client later than the second response.

```
1   function $̂ = [](Server _) { return ⊥; };
2
3   TS client() {            // Q_σ
4     function σ̆ = [](Server _) { return ⊤; };
5     Set acks;
6
7     // broadcast the first hop
8     for (dst: servers) {
9       thread _(snd_rcv, ⊥, dst, acks);
10    }
11
12    while (true) {
13      // wait for the next response
14      !_src response = acks.next();
15      σ̆ = update(σ̆, src, min(σ̆(src),response));
16
17      // find a candidate
18      if (M@σ̆ < ⊤) {
19        TS candidate = M@σ̆;
20        if (candidate <= M@$̂) {
21          return candidate; // A_σ
22        } else if (acks.empty()) {
23          // promote the candidate
24          for (dst: servers) {
25            if ($̂(dst) < candidate)
26              thread _(snd_rcv, candidate, dst, acks);
27          } // end for (servers)
28        }   // end if  (acks.empty())
29      }     // end if  (M@σ̆ < ⊤)
30    }       // end while
31  }         // end client
32
33  void snd_rcv(Request qst, Server srv, Set &acks) {
34    send(?qst, srv);
35    Response rsp = recv(srv);
36    $̂ = update($̂, srv, max($̂(srv),rsp));
37    acks.insert(!_srv rsp);
38  }
39
40  function update(function f, Server k, TS v) {
41    return [](Server arg) {
42      return arg == k ? v : f(arg);
43    };
44  }
```

Figure 4: TaaS client algorithm, handling server failures by broadcasting a promotional second hop. In each session, let $\breve{\sigma}$ store the smallest response sent by each server. If the client has received at least M responses to the first hop in Line 18, then it takes $M@\breve{\sigma}$ as a candidate, and decides whether this `candidate` is ready to conclude or not in Line 20. For the servers whose cache (i.e., largest response ever received by the client, stored in "$\hat{\$}$") is less than the `candidate`, the client broadcasts the `candidate` as the second hop to keep those servers up to date in Line 26.

3.3 Proof of correctness and availability

Theorem 2 (Functional correctness). The TaaS client algorithm in Figure 4 satisfies the correctness with respect to Definition 4.

Proof. We show correctness by proving that for each pair of sessions σ that ends before τ begins, their conclusive candidates $\mathtt{CC}_\sigma$ and $\mathtt{CC}_\tau$ satisfy the following inequation:

$$\begin{aligned}\mathtt{CC}_\sigma \le M@(\hat{\$}^\sigma : A_\sigma) &\le M@(\Omega : A_\sigma) \\ &\le M@(\Omega : Q_\tau) < M@\check{\tau} = \mathtt{CC}_\tau\end{aligned}$$

Here syntax "$\hat{\$}^\sigma : A_\sigma$" is pronounced "the global mapping at time A_σ, observed by the client that executes session σ".[4]

(1) $\mathtt{CC}_\sigma \le M@(\hat{\$}^\sigma : A_\sigma)$ is the `if` condition in Line 20.
(2) $M@(\hat{\$}^\sigma : A_\sigma) \le M@(\Omega : A_\sigma)$ is proven as Lemma 5.
(3) $M@(\Omega : A_\sigma) \le M@(\Omega : Q_\tau)$ is proven as Lemma 6.
(4) $M@(\Omega : Q_\tau) < M@\check{\tau}$ is proven as Lemma 7.
(5) $M@\check{\tau} = \mathtt{CC}_\tau$ is the candidate's definition in Line 19.

Therefore:

$$\forall\sigma, \forall\tau, (A_\sigma \prec Q_\tau \implies \mathtt{CC}_\sigma < \mathtt{CC}_\tau) \qquad \square$$

Theorem 3 (Availability of timestamps). Let N be the cardinality of servers. For each session σ, if there exist $\max(M, N - M + 1)$ servers that are up, then the session will eventually conclude.

Proof. We need to show that, provided $\max(M, N - M + 1)$ servers being responsive: (1) The client can find a candidate; and (2) The candidate will eventually become conclusive.

(1) Since all the up servers will respond to the first hop, the client can receive at least M responses in σ, thus candidate $M@\check{\sigma}$.
(2) Consider the cache record of $N - M + 1$ up servers: Some of them might be less than the candidate, but will become greater after they respond to the promotinal second hop. Therefore, at most $N - (N - M + 1) = M - 1$ cache records may remain below the candidate, thus $M@\hat{\$} \ge \mathtt{candidate}$.

Therefore, any $\max(M, N - M + 1)$ servers being up guarantees the session to conclude. $\square$

Corollary 3.1 (Recommendation on parameter M). While the choice of parameter M does not affect correctness, here we recommend M to be $\lceil \frac{N+1}{2} \rceil$, for the following reasons:

(1) The system allows at most $\min(N - M, M - 1)$ downs while continuing its service. Therefore, for maximum fault tolerance, we should take $M \approx \frac{N+1}{2}$, i.e., *M*edian.
(2) If a client hasn't heard from at least $N - M + 1$ servers when looking for a candidate, then the client should not naïvely believe that at least $N - M + 1$ servers would ever respond to the second hop that promotes the candidate. Therefore, the client should wait for $M \ge N - M + 1$ servers to respond before choosing the candidate, to gain confidence in promoting it.

For the rest of this paper, we assume that $M = \lceil \frac{N+1}{2} \rceil$, unless otherwise mentioned. This allows us to discuss liveness in a simpler way: "A session would eventually conclude *if and only if* there exist M servers being up."

[4]In this theorem, sessions σ and τ may be executed by the same client or by different clients, so they may observe the same or different global mappings. Therefore, we explicitly specify the mapping(s) they observe, as well as the time of observation.

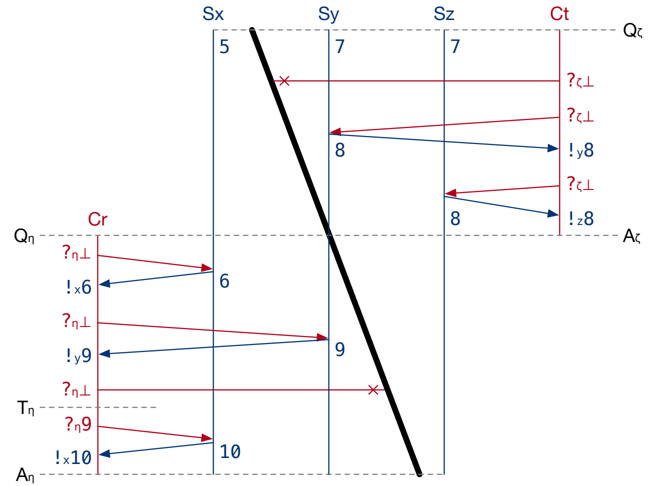


Figure 5: Network partition example.

3.4 Additional examples

To better explain our client algorithm, we illustrate some edge cases, featuring network partition and message delays.

3.4.1 *Network partition.*

As a consensusless protocol, TaaS servers do not communicate with each other. In this case, the concept of "partition" refers to disrupted client–server connections. For example, in Figure 5, client Ct cannot reach server Sx, and client Cr loses connection to server Sz.

First consider session ζ: When Ct has received $!_y8$ and $!_z8$, the client updates its local and global mappings to:

$$\check{\zeta} = \{x \mapsto \top; y \mapsto 8; z \mapsto 8\}$$
$$\hat{\$}^\zeta = \{x \mapsto \bot; y \mapsto 8; z \mapsto 8\}$$

The client then candidates $2@\check{\zeta} = 8$, and finds it conclusive, i.e., $2@\hat{\$}^\zeta \ge 8$. So the client concludes with 8 without waiting for Sx or broadcasting a second hop.

Now consider session η: The client first broadcasts $?_\eta\bot$ and starts processing the responses, i.e., enters the `while` loop in Line 12 of Figure 4. After processing $!_x6$ and $!_y9$, the client has:

$$\check{\eta} = \{x \mapsto 6; y \mapsto 9; z \mapsto \top\}$$
$$\hat{\$}^\eta = \{x \mapsto 6; y \mapsto 9; z \mapsto \bot\}$$

The client candidates $2@\check{\eta} = 9$ in Line 19, and finds it inconclusive, i.e., greater than $2@\hat{\$}^\eta = 6$. Since the connection to Sz is cut, the client observes no more responses, i.e., the `acks` set becomes empty. So the client enters the `if` branch at Line 22, and immediately promotes the candidate 9, without waiting for Sz to respond. Upon receiving $!_x10$, the client returns the conclusive candidate $\mathtt{CC}_\eta = 9$, which is greater than $\mathtt{CC}_\zeta = 8$.

3.4.2 *Delayed messages.*

Instead of waiting for any specific server to respond, the TaaS algorithm asynchronously processes the responses from all the servers. The client may conclude a session after processing a subset of the responses, so as to optimize latency. Then what happens when some responses arrive too slow?

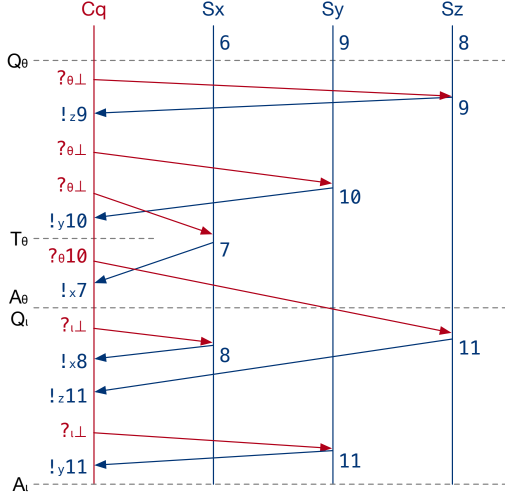


Figure 6: Delayed messages example.

Consider sessions θ and ι in Figure 6: At time T_θ, client Cq has received $!_z 9$ and $!_y 10$, while the response by Sx was delayed en route. The client's local and global mappings are:

$$\check{\theta} : T_\theta = \{x \mapsto \top; y \mapsto 9; z \mapsto 10\}$$

$$\hat{\$} : T_\theta = \{x \mapsto \bot; y \mapsto 9; z \mapsto 10\}$$

The client candidates $2@(\check{\theta} : T_\theta) = 10$, and promotes the candidate by broadcasting $?_\theta 10$ as a second hop. The client then receives the delayed first-hop response $!_x 7$, and updates its mappings to:

$$\check{\theta} : A_\theta = \{x \mapsto 7; y \mapsto 9; z \mapsto 10\}$$

$$\hat{\$} : A_\theta = \{x \mapsto 7; y \mapsto 9; z \mapsto 10\}$$

So the client recalculates the candidate $2@(\check{\theta} : A_\theta) = 9$, and finds it conclusive, i.e., $2@(\hat{\$} : A_\theta) \geq 9$. Thus, the client concludes with 9 without waiting for the response to $?_\theta 10$, in the same way as §3.1 that takes the M-th smallest among all the first-hop responses.

The client then launches session ι and receives $!_x 8$ and $!_y 11$. Meanwhile, the asynchronous thread sending $?_\theta 10$ wasn't killed, but received $!_z 11$. Recall our definition in §3.2.3 that—$\check{\iota}$ represents "the smallest response by each server for session ι", and $\hat{\$}$ stores "the largest response ever sent by each server, regardless of for which session", these mappings become:

$$\check{\iota} = \{x \mapsto 8; y \mapsto 11; z \mapsto \top\}$$

$$\hat{\$} : A_\iota = \{x \mapsto 8; y \mapsto 11; z \mapsto 11\}$$

So the client candidates $2@\check{\iota} = 11$, and finds it conclusive, i.e., no greater than $2@(\hat{\$} : A_\iota)$. Thus, it concludes session ι with 11.

Summary. So far, we have explained the client side logic for tolerating down servers. The main idea is to bound the "conclusive candidate" of each session σ strictly greater than $M@(\Omega : Q_\sigma)$, and no less than $M@(\Omega : A_\sigma)$, using the following inequation:

$$M@(\Omega : Q_\sigma) < M@\check{\sigma} = \mathtt{CC}_\sigma \leq M@(\hat{\$} : A_\sigma) \leq M@(\Omega : A_\sigma)$$

This inequation relies on the servers to advance monotonically, despite errors like power outage and disk corruption. In §4.1, we present two ways to protect the monotonicity of each server.

Table 2: Differentiating timestamps by server identifiers.

M	$M@\check{\alpha}$	$M@\check{\beta}$	$M@\check{\gamma}$
1	1.x	1.z	2.x
2	1.y	2.y	3.y
3	2.z	3.x	3.z

4 PRACTICAL CONSIDERATIONS

4.1 Monotonic clocks via stable storage

Theorem 3 allows some servers to be down without disrupting the service. We want to *resume* the down servers up, before there are too many of them that make the service unavailable.

The "downs" of servers can be classified as follows:

(1) A *disconnection* is a server being unable to receive requests from and/or send responses to some clients or any client. A disconnected server works fine (i.e., monotonically) as a logical clock if it ever gets ticked, and can be safely resumed by fixing the client–server connection.
(2) A *failure* is a server forgetting the largest timestamp it has ever sent, i.e., forgetting the largest state it has ever reached. For example: (i) The server stored the timestamp in non-persistent RAM, and lost it in a power outage; or (ii) The server wrote its state to a hard disk, but the disk was broken in an earthquake. A failed server must be *recovered* before resumed. To keep monotonicity that the server's next response is greater than all its previous responses, we need to set the server to a *resumption state* that is greater than its pre-failure state.

To recover a failed server without breaking monotonicity, we need to compute the *upper bound* of the server's pre-failure state, and use this upper bound as the resumption state. Here we introduce two flavors of solutions:

(1) Cloud style: trust the disk. The clock server may periodically allocate a range of timestamps that it can serve, and store the range on a reliable disk, e.g., our triplicate disks on Alibaba Cloud [1] that provides 99.9999999% data durability.
(2) Garage style: bring your own cluster. To deploy TaaS on bare metal, a simple solution is to allocate the timestamps on a conceptual "metadata node"—a highly-available cluster that replicates timestamps on multiple disks. By exploiting the redundancy of consensus, such architecture may achieve durability comparable to cloud storages.

As mentioned in §2.2.2, CORFU [2], PolarDB-X [3], TiDB [10], Percolator [19], and Omid [21] all preserve timestamps on consensus facilities such as etcd [7] and Apache Zookeeper [11]. Upon failure, the server simply resumes from above the previously-allocated range, thus guarantees its monotonicity.

We adopt this allocation-based strategy from existing timestamp oracles into our TaaS prototype evaluated in §5. To prevent the cloud disk from bottlenecking the server, our servers allocate new ranges of timestamps before exhausting the old ones.

Table 3: Experiment variables. The Availability experiment measures throughput and latency qualitatively (Qual.). The Stability experiment measures the latency quantitatively (Quant.) and disregards the throughput (N/A).

RQ	Availability	Scalability	Stability
# of Datacenters	1,3	1	1
# of Servers (N)	5	3,5,7,9	5
# of Clients	100	1,2,4,8,16	1
Interference	Outage%	N/A	Delay%
Throughput	Qual.	Quant.	N/A
Latency	Qual.	Qual.	Quant.

4.2 Unique timestamps via disjoint clocks

As mentioned in §3.1.4, TaaS by default does not linearize to sequentially ticking a logical clock, but to looking at a clock on the wall, where simultaneus observations may result in the same conclusion.

To make the conclusions unique, we may annotate each timestamp with its sending server's identifier. For the example in Figure 2, the annotated timestamps are shown in Table 2. Timestamps of the same counter (higher bits) are ordered by their server identifers (lower bits), e.g., $1.x < 1.y < 1.z$.

As a result, when $M = 1$, sessions α, β, and γ would conclude with timestamps $1.x$, $1.z$, and $2.x$, respectively, which are distinguishable by their lower bits.

Theorem 4 (Uniqueness of Conclusions). No two sessions may result in the same server-identified conclusion.

Proof. For each session, the candidate must be among the responses received within that session. A server never responds with the same counter twice, so no two sessions may receive the same annotated response. Therefore, each session's candidate (and thus conclusion) is unique. □

5 EVALUATION

5.1 Experiment Design

The empirical evaluation of the TaaS algorithm is to answer the following research questions (RQ's):

- RQ1 (Availability): Does TaaS exhibit—in practice—the high availability we proved mathematically in Theorem 3?
- RQ2 (Scalability): How does TaaS behave when scaled up?
- RQ3 (Stability): Is TaaS prone to network interference?

These RQ's motivate us to compare TaaS against state of the art timestamp oracle qualitatively and quantitatively. The baseline we choose is TiDB, a hybrid transactional and analytical processing (HTAP) database. TiDB's placement driver (PD) module can "generate about a million timestamps per second" [10].

Our TaaS prototype is implemented by injecting the client and server algorithms into the TiDB-PD codebase. We reuse the remote process call (RPC) interface in TiDB, so as to evaluate the algorithms using the most similar facilities possible.

We compare TaaS with TiDB in three experiments, as shown in Table 3. We explain the experiments' variables and invariables in §§5.1.1–5.1.2, and present the results in §§5.2–5.4.

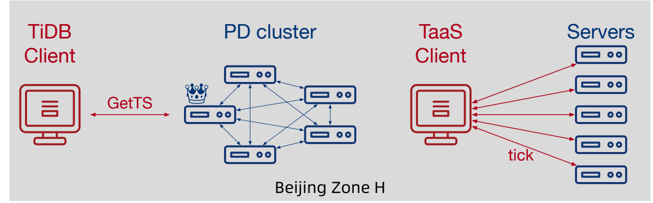
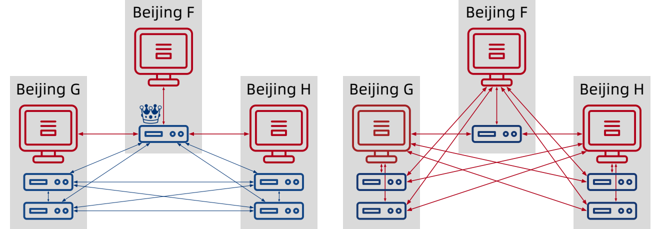


Figure 7: Single-datacenter experiment architecture.



(a) TiDB-PD **(b) TaaS**

Figure 8: Tri-datacenter experiment architecture.

5.1.1 Variables. To answer the different RQ's, we manipulate the timestamping systems in various directions, emphasizing outages (RQ1), parallelism (RQ2) and delays (RQ3).

Servers. To evaluate the availability of TaaS with different levels of outages (server-level and datacenter-level), we deploy the servers in two settings: Figure 7 puts all servers in the same datacenter. Figure 8 distributes the servers into three datacenters—namely Zones F, G, and H—all in Beijing.

The scalability and stability experiments are conducted within Beijing Zone H. To measure the performance of TaaS as the system scales up, we change the number of servers $N \in \{3, 5, 7, 9\}$, with parameter $M \in \{2, 3, 4, 5\}$, as recommended by Corollary 3.1.

Clients. As mentioned in Definition 3, a computer may launch parallel client processes that execute sessions simultaneously. For example, TiDB-PD benchmarks its timestamp throughput by running 100 clients per machine. We replicate this level of parallelism in the availability experiment, and analyze the performance of running $\{1, 2, 4, 8, 16\}$ parallel clients in the scalability experiment.

The single-datacenter experiments hosts all client(s) on the same machine. The tri-datacenter availability experiment puts one client-hosting machine per zone, each running 100 clients in parallel.

Workload. The original TiDB experiment by Huang et al. had the clients to request timestamps as quickly as possible, to measure the maximum throughput they can achieve by exhausting the machines.

When replicating such flooding workload on TaaS, we observe a significant increase of latency, and blame the concurrent nature of TaaS clients for it: Unlike the TiDB client that only sends *one* `GetTS` request to the cluster's leader *synchronously*, the TaaS client interacts with *all* servers *asynchronously*, and thus involves heavier concurrency for both the operating system and the network stack.

Table 4: Single-zone outage schedule.

Time	TaaS	TiDB-PD
0'00"	All five servers up	
1'00"	Stop a random server	Stop the leader
2'00"	Stop another server	Stop the then leader
3'00"	Resume a random server	
4'00"	Resume the remaining server	
5'00"	End of experiment	

Table 5: Tri-zone outage schedule.

Time	TaaS	TiDB-PD
0'00"	All three zones up	
1'00"	Stop a random zone	Stop the leader's zone
2'00"	Resume the stopped zone	
3'00"	Stop another zone	Stop the then leader's zone
4'00"	Resume the stopped zone	
5'00"	Stop the third zone	Stop the then leader's zone
6'00"	Resume the stopped zone	
7'00"	End of experiment	

We then doubt: To answer RQ's in §5.1, is it really worth measuring the latency at the extreme workload (approximately 600,000 requests per second) that overloads the client machine?

When benchmarking on TPC-C [24], TiDB "requests at most 6000 timestamps per second per server[5]". So in the availability experiment, we choose to overload TiDB and TaaS by five times, i.e., requesting a fixed rate of 30,000 timestamps per second per client. Our scalability experiment studies how the TaaS client gets exhausted, by flooding it with requests at unlimited rate.

Outages. We demonstrate the high availability of TaaS by stopping arbitrary servers. Our deployment consist of five servers that are all up initially. We then reduce the number of ups, by randomly choosing servers and killing their processes. We expect the service to be continuously available provided any three servers being up.

As for TiDB-PD, we show its single point of failure by deliberately killing the leader, and expect no timestamp to be served during the failover period, until the cluster re-elects a leader.

We simulate server-level outages in the single-zone experiment, and simulate zone-level outages (where the entire datacenter becomes unavailable) in the tri-zone experiment. The per-minute schedules are listed in Table 4 and Table 5, respectively.

Delays. To emulate interferences that may delay network packets, we use NetEm [9] to add network delays at nearly[6] exponential distribution [22] with standard deviation of 1 millisecond.

When testing TaaS, we measure the latency when $\{0, 1, 2, 3, 4, 5\}$ out of 5 servers are interfered with the delay. As for TiDB-PD, we find that delaying the followers does not affect the latency, so we compare the latency under the leader being interfered vs intact.

[5]Here the "server" corresponds to a "TiDB client" in Figure 7.

[6]NetEm can only produce delays within 4 times of the standard deviation, so the added delay ranges between $[0, 4\sigma]$ rather than $[0, +\infty)$.

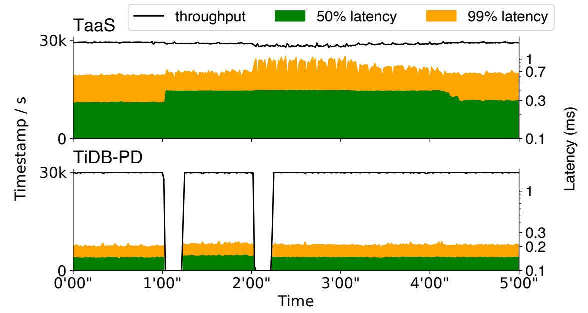


Figure 9: Single-zone: throughput and latency upon outage.

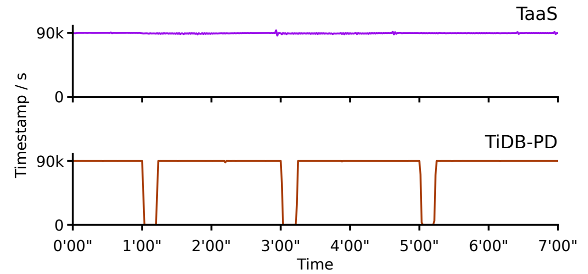


Figure 10: Tri-zone: total throughput upon outage.

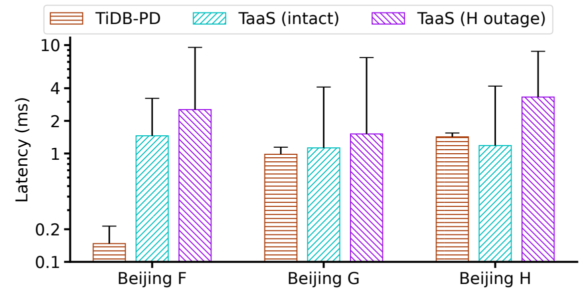


Figure 11: Tri-zone: median and 99% latency in each zone.

5.1.2 *Invariables.* Software wise, we re-use the benchmarking tool for TiDB-PD. We inject the TaaS algorithm as a replacement module in the TiDB codebase, switched on/off via command line.

Hardware wise, we run all clients and servers on Alibaba Elastic Compute Service, using instance type `ecs.c5.8xlarge` equipped with 32-virtual-core Intel Xeon Platinum CPUs at 2.5GHz, RAM of 64GB, and network of bandwidth 10Gbps. The round-trip time (RTT) is 0.1–0.2ms within datacenter, or 1–2ms across datacenters.

5.2 Upon server outage

5.2.1 *Within the same datacenter.* The TaaS and TiDB clients count the number of timestamps they fetched per second, and measures the latency distribution.

As shown in Figure 9, when the leader is stopped, TiDB-PD stops serving timestamps for nearly 10 seconds. We believe the blackout

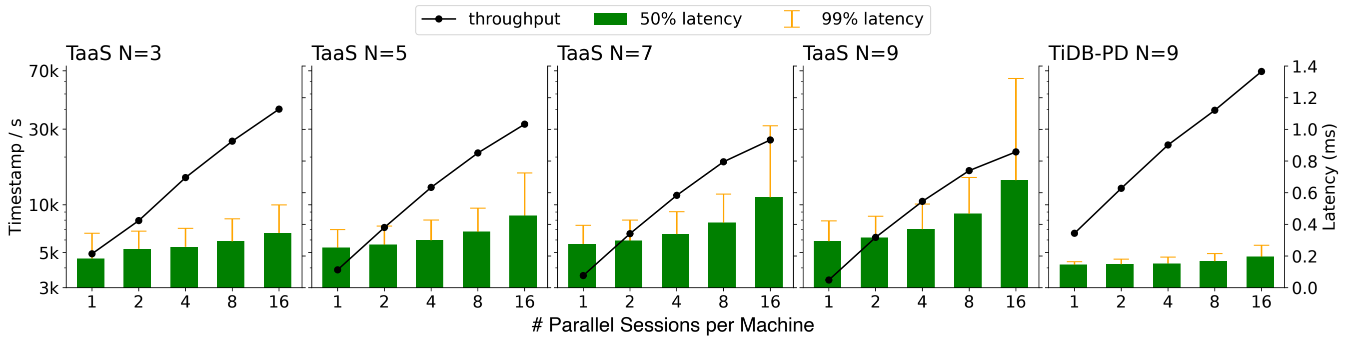


Figure 12: Throughput and latency at different parallelism.

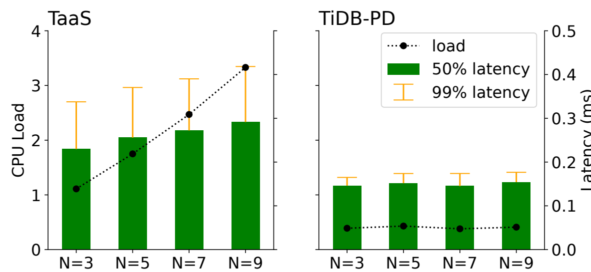


Figure 13: Client-side overhead running one session at a time.

period can be shortened by tuning the parameters of the consensus algorithm, but never eliminable due to the SPoF nature of the leader.

In comparison, the consensusless design keeps TaaS immune from SPoF. Stopping one or two servers increases the median latency by 0.1ms—i.e., one additional RTT for promoting the candidate—but never disrupts the timestamp service.

5.2.2 Across datacenters. We compare the total throughput between TaaS and TiDB, by summing up the number of timetamps fetched by the three clients, each requesting 30,000 per second.

As shown in Figure 10, during normal execution, both TaaS and TiDB meet the throughput expectation of 90,000 per second. When the TiDB leader's zone fails, the service discontinues for nearly 10 seconds, in the same way as Figure 9. In contrast, the TaaS service remains stable, no matter which zone fails.

Latency. TiDB clients in different zones experience distinct latency: The client in the same zone as the leader can conclude within 0.1–0.2ms, and clients in different zones need to wait for 1–1.5ms.

The latency of TaaS is less distinct among different zones, as shown in Figure 11. When all the servers are up, the clients in all three zones exhibit a similar latency of 1–1.5ms.

Upon zone-level outages, e.g., Beijing H,[7] all clients need to tick all three remaining servers, one in F and two in G. The client in H is impacted most (median 3.3ms), followed by F (2.5ms). The impact on G is less significant (1.5ms), due to colocation with two servers.

[7]We simulate zone-level failures by stopping all the TaaS servers, rather than shutting down our datacenter. So the clients are still alive and analyzing the performance.

5.3 As the system scales

As shown in Figure 12, running more sessions in parallel increases the total throughput of the client machine, but also increases the latency of each timestamp. And when deploying more TaaS servers, the latency significantly increases as more clients run in parallel.

We locate this scalability issue by measuring the CPU load of the client machine running one session at a time. As shown in Figure 13, the TaaS client's CPU load linearly increases with the number of servers, which matches our expectation: To compute a timestamp, the TaaS client needs to tick all servers once, or tick a majority of the servers twice, i.e., at complexity $O(N)$. In comparison, TiDB-PD only needs to tick the leader once, i.e., at complexity $O(1)$. We view the client-side complexity as a trade-off for higher availability.

As a result, running multiple TaaS sessions may overload the CPU (e.g., 16 parallel sessions with $N = 9$), resulting in excessive latency and sublinear throughput. Note that real-world clients do not fetch timestamps floodingly as the benchmarks do, but only on demand per transaction. The sub-millisecond latency of TaaS is negligible for database transactions that take dozens of milliseconds.

On the server side: When serving 90,000 timestamps per second (i.e., 15 times of TiDB's TPC-C throughput) in the tri-datacenter experiment, the peak CPU load varies among TaaS servers, ranging from 7 to 23 (during outage).[8] In comparison, the TiDB-PD leader exhibits a constant CPU load around 7.

5.4 Under network interference

As shown in Figure 14, when a minority of the servers are interfered by network delays, the latency of TaaS increases by 0.1ms, i.e., one additional RTT. If the interference expands to a majority ($\geq M$), then the latency significantly increases, as the client needs to tick at least M servers before reaching a conclusion.

In comparison, the latency of TiDB-PD only depends on the client–leader connection: If the leader is interfered by the network delay, then the latency expectation exceeds 1 millisecond.

We also observe that the latency distribution of TiDB-PD (solid red line) exhibits a bell curve, which differs from the client–leader ping (dotted black line) that follows exponential distribution. This indicates that the timestamping latency of TiDB-PD should not be simply modelled as 1RTT, and is worth studying beyond this paper.

[8]We read this result qualitatively—Each session ticks different subsets of servers, based on (i) "what" the servers respond with and (ii) "when" the responses arrive.

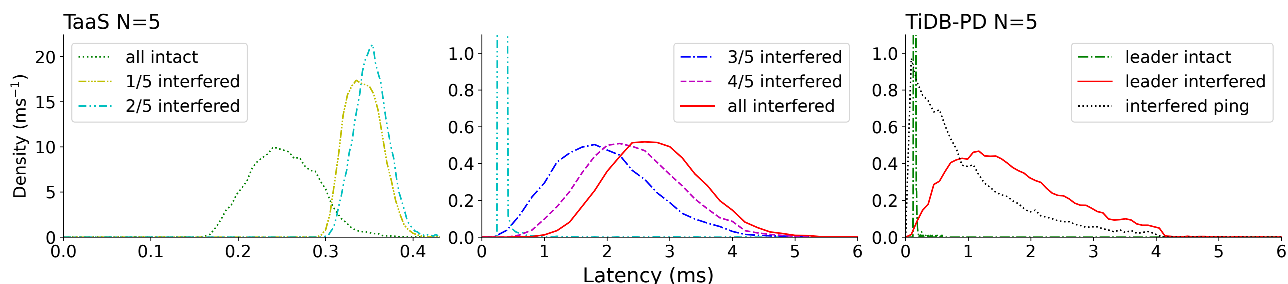


Figure 14: Latency distribution under network inference.

Summary. These results allow us to answer the RQ's in §5.1:

RQ1: TaaS is immune to server-level and zone-level failures, which reflects our mathematical proof in Theorem 3.

RQ2: As the number of servers increases, TaaS tolerates more outages, at the cost of higher client loads and increased latency.

RQ3: Upon network interference, TaaS exhibits a more predictable performance that depends on the number of affected servers, compared to TiDB-PD that varies with "which" server being slow.

6 RELATED AND FUTURE WORKS

This section locates TaaS in the area of timestamping transactions, and projects how to expand TaaS for more scenarios: §6.1 compares centralized and decentralized approaches of timestamping. Within the scope of centralized timestamping—which this paper studies—§6.2 compares the consensusless TaaS algorithm against consensus-based solutions. §6.3 then discusses how to use timestamps in end-to-end solutions—whether deploying it as standalone service, or embedding it into the protocol design.

6.1 Centralized / Decentralized

As mentioned in §2.1, clocks in distributed systems consist of: (i) "Centralized" ones that are ticked by all participants, e.g., TaaS and consensus-based oracles; and (ii) "Decentralized" ones that are maintained by each participant, e.g., TrueTime [4] and scalar/vector/hybrid logical clocks [13, 14, 16]. Here we discuss the pros and cons of centralized timestamping, and explore the possibility of combining centralized and decentralized techniques.

Pros. Centralized clocks are easy to understand and implement:

- Theoretically: All the participants tick the same clock, so the only logic is that "earlier ticks get smaller timestamps".
  In contrast, logical clocks introduces the concept of "advancing one clock to above another clock", and TrueTime involves uncertainty—both require careful reasoning.
- Practically: Centralized timestamps are requested "on demand", where the participant calls a stateless (except for caching) function that queries the clock service.
  Whereas, logical clocks require each participant to maintain its own timer state, and carry timestamps along all synchronizations, so as to track the causality among transactions. TrueTime lifts the demand on modifying the communication protocol, but relies on GPS satellites to keep its atomic clocks tightly synchronized.

Cons. Centralized timestamping is limited by lightspeed: When the data spreads across continents, the RTT for ticking the centralized clock might reach hundreds of milliseconds, which is unacceptable for a cloud service.

In contrast, decentralized clocks are colocated with their users, and thus impose less latency overhead: TrueTime clients only need to wait for the uncertainty range to expire, which takes less than 10ms. Logical clocks are continuously synchronized per communication, so the latency of ticking locally becomes negligible.

Next step: Combining centralized + decentralized timestamping. To exploit the locality of decentralized clocks, while keeping the simplicity of centralized clocks, the PolarDB-X database [3] partitions data across datacenters, and deploys a timestamp oracle per region that orders local transactions. Each region also equipts a hybrid logical clock for synchronizing cross-region transactions.

We will explore the possibility of substituting the region-specific timestamp oracles with a TaaS-per-region for higher availability, in combination with the decentralized HLC, to achieve optimal latency for both local and global transactions.

6.2 Consensus / Consensusless

One innovation of TaaS is the bipartite architecture, where the server only connects to the clients—unlike consensus-based mechanisms that synchronize among servers. So what makes TaaS different? What do we mean by "consensusless"?

Consistency. We read the concept of "consistency" (i) generically as "refining a single representation for the convenience of reasoning"; and (ii) specifically to the consensus world, "converging all replicas to the same representation".

Now we can define "consensusless" as "achieving consistency without converging representations": The "representation" that TaaS reasons about is $M@\Omega$—the M-th smallest server state—that deliminates the lower and upper bounds of each conclusion. Provided any M servers being up, the client can deduce and advance this representation, so as to achieve timestamp completeness.

Partition. The requirement of "convergence" results in the completely interconnected architecture of consensus-based systems. But for the timestamping problem, we find "completeness" (Definition 1) a weaker yet sufficient goal, that can be satisfied by a bipartite architecture whose servers are naturally partitioned. In this paper, "partition" refers to "disconnecting some clients from some servers", as shown in Figure 5.

Availability. As proven in Theorem 3, a TaaS client can conclude if and only if it can reach any $\max(M, N - M + 1)$ out of N servers. As a result, for any pair of clients that are simultaneously available, they must reach some server in common, per pigeon-hole principle. Such restriction motivates us to formalize the concepts of CAP in a consensusless setting, which we leave as future work.

6.3 Standalone / Embedded

Distributed systems utilize timestamps in two styles:

(1) Deploying a *standalone* service that assigns a timestamp for each event, e.g., TrueTime and centralized timestamp oracles.
(2) *Embedding* timestamps into the concurrency control protocol, e.g., two-phase commit [3, 26], timestamp stability [6], logical clocks [13, 14, 16], and consensus protocols [15, 18].

Comparison. Standalone timestamping *reduces* the order of events to the order of their timestamps. It keeps the communication protocol simple that all participants query the timestamp service, rather than negotiating with each other. Therefore, mainstream products choose the standalone style whenever it is affordable—e.g., deploying within the same region (where the latency overhead is acceptable), or having sufficient budget for TrueTime.

Embedded timestamping *deduces* the event order by reasoning on the timestamps transmitted among the events. Such methods avoid the overhead of explicitly fetching timestamps, but requires modifying the communication protocol to carry timestamps during all synchronizations. As a result, embedded timestamping is widely used for systems that care about reducing the round-trips of transactions, e.g., databases that scale geographically [3, 23, 25–27].

Next step: Embedding the TaaS theory. The TaaS client *deduces* timestamps from the collection of server states, in a deterministic process that doesn't require conflict detection. Such simplicity motivates us to expand the standalone TaaS service into an embedded mechanism for ordering transactions.

For example, we may squeeze the "timestamp and then commit" processes into one phase, by smuggling data payloads along with the ticks. This results in a consensusless distributed journal whose append operation normally takes only one (client–server) hop—regardless of conflicts—unlike leader-based consensus [15, 18] that take two hops (i.e., client–leader–followers) or leaderless solutions [6, 17] that require detecting and resolving conflicts. Our next paper will introduce the TaaS-inspired journaling mechanism.

7 CONCLUSION

This paper shows how to generate logical timestamps correctly, quickly, and smoothly. We study the fundamental need for timestamps, and define a monotonic spacetime over a cluster of logical clocks. We tolerate single-point errors by computing timestamps from the entire cluster, rather than prioritizing any specific server.

TaaS introduces an alternative to consensus for fault-tolerance: We demonstrate how to build a consistent service, without synchronizing among the servers. TaaS inspires us to think out of the "consensus" box when (re)designing distributed databases.

Appendices

The correctness of the TaaS algorithm proven in Theorem 1 and Theorem 2 are based on the following lemmas:

Lemma 5 (Lower bound of ending server states). By the end of session σ: if the client has received—during any session—responses that are no less than X from at least $N - M + 1$ servers, then at the end of session σ, there exist at least $N - M + 1$ servers whose states are no less than X:

$$\forall M, \forall \sigma, \forall X, M@(\hat{\$} : A_\sigma) \geq X \implies M@(\Omega : A_\sigma) \geq X$$
$$\text{i.e., } M@(\Omega : A_\sigma) \geq M@(\hat{\$} : A_\sigma)$$

Proof. Consider the $N - M + 1$ servers that responded with no less than X before the end of session σ: Their states have been advanced to no less than X when sending the responses.

Therefore, at the end of session σ, these $N - M + 1$ server states are still no less than X, per monotonicity. □

Lemma 6 (Global monotonicity). For any pair of time T_1 that is earlier than T_2, the M'th smallest server state at T_1 is no greater than the M'th smallest server state at T_2:

$$\forall T_1, \forall T_2, (T_1 < T_2 \implies \forall M, M@(\Omega : T_1) \leq M@(\Omega : T_2))$$

Proof. Let $X = M@(\Omega : T_1)$ and $Y = M@(\Omega : T_2)$. Assume to the contrary that $X > Y$, then at time T_1, at least $N - M + 1$ servers have states no less than X. Therefore at time T_2, these $N - M + 1$ servers' states are still no less than X, thus greater than Y.

At time T_2, at least M servers have states no greater than Y. We then count the total number of servers—whose states either greater than or no greater than Y—no less than $(N - M + 1) + M = N + 1$, which contradicts the fact that there exist only N servers. □

Lemma 7 (Upper bound of beginning server states). For requests sent within session τ: if the client has received responses to those requests from at least M servers that are no greater than X, then at the beginning of session τ: there exist at least M servers whose states are less than X:

$$\forall M, \forall \sigma, \forall X, M@\check{\sigma} \leq X \implies M@(\Omega : Q_\sigma) < X$$
$$\text{i.e., } M@(\Omega : Q_\sigma) < M@\check{\sigma}$$

Proof. Consider the M servers that responded with no greater than X: Their states were less than X before processing the requests, which arrived after the beginning of σ.

Therefore, at the beginning of session σ, these M server states must be less than X, per monotonicity. □

ACKNOWLEDGMENTS

We thank the reviewers for paving the path towards this paper, Feiyu Yang for contributions to evaluating our TaaS prototype, and Qingda Lu for comments on earlier drafts.

REFERENCES

[1] Alibaba Cloud Documentation Center. 2022. Elastic Compute Service: Triplicate storage. https://www.alibabacloud.com/help/en/elastic-compute-service/latest/triplicate-storage.

[2] Mahesh Balakrishnan, Dahlia Malkhi, John D. Davis, Vijayan Prabhakaran, Michael Wei, and Ted Wobber. 2013. CORFU: A Distributed Shared Log. *ACM Trans. Comput. Syst.* 31, 4, Article 10 (Dec. 2013), 24 pages. https://doi.org/10.1145/2535930

[3] Wei Cao, Feifei Li, Gui Huang, Jianghang Lou, Jianwei Zhao, Dengcheng He, Mengshi Sun, Yingqiang Zhang, Sheng Wang, Xueqiang Wu, Han Liao, Zilin Chen, Xiaojian Fang, Mo Chen, Chenghui Liang, Yanxin Luo, Huanming Wang, Songlei Wang, Zhanfeng Ma, Xinjun Yang, Xiang Peng, Yubin Ruan, Yuhui Wang, Jie Zhou, Jianying Wang, Qingda Hu, and Junbin Kang. 2022. PolarDB-X: An Elastic Distributed Relational Database for Cloud-Native Applications. In *2022 IEEE 38th International Conference on Data Engineering (ICDE)*. 2859–2872. https://doi.org/10.1109/ICDE53745.2022.00259

[4] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, J. J. Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. 2013. Spanner: Google's Globally Distributed Database. *ACM Trans. Comput. Syst.* 31, 3, Article 8 (Aug. 2013), 22 pages. https://doi.org/10.1145/2491245

[5] Albert Einstein. 1905. Zur Elektrodynamik bewegter Körper. *Annalen der Physik* 322, 10 (June 1905), 891–921. https://doi.org/10.1002/andp.19053221004

[6] Vitor Enes, Carlos Baquero, Alexey Gotsman, and Pierre Sutra. 2021. Efficient Replication via Timestamp Stability. In *Proceedings of the Sixteenth European Conference on Computer Systems* (Online Event, United Kingdom) *(EuroSys '21)*. Association for Computing Machinery, New York, NY, USA, 178––193. https://doi.org/10.1145/3447786.3456236

[7] etcd Authors. 2023. etcd. https://etcd.io/.

[8] Cary G. Gray and David R. Cheriton. 1989. Leases: An Efficient Fault-Tolerant Mechanism for Distributed File Cache Consistency. In *Proceedings of the Twelfth ACM Symposium on Operating Systems Principles (SOSP '89)*. Association for Computing Machinery, New York, NY, USA, 202—-210. https://doi.org/10.1145/74850.74870

[9] Stephen Hemminger. 2005. Network emulation with NetEm. *Linux Conf Au* (May 2005).

[10] Dongxu Huang, Qi Liu, Qiu Cui, Zhuhe Fang, Xiaoyu Ma, Fei Xu, Li Shen, Liu Tang, Yuxing Zhou, Menglong Huang, Wan Wei, Cong Liu, Jian Zhang, Jianjun Li, Xuelian Wu, Lingyu Song, Ruoxi Sun, Shuaipeng Yu, Lei Zhao, Nicholas Cameron, Liquan Pei, and Xin Tang. 2020. TiDB: A Raft-Based HTAP Database. *Proc. VLDB Endow.* 13, 12 (Sept. 2020), 3072–3084. https://doi.org/10.14778/3415478.3415535

[11] Patrick Hunt, Mahadev Konar, Flavio P. Junqueira, and Benjamin Reed. 2010. ZooKeeper: Wait-Free Coordination for Internet-Scale Systems. In *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference* (Boston, MA) *(USENIXATC'10)*. USENIX Association, USA, 11.

[12] ISO/IEC. 1998. Distributed Transaction Processing. ISO/IEC 10026-1:1998.

[13] Sandeep S. Kulkarni, Murat Demirbas, Deepak Madappa, Bharadwaj Avva, and Marcelo Leone. 2014. Logical Physical Clocks. In *Principles of Distributed Systems*, Marcos K. Aguilera, Leonardo Querzoni, and Marc Shapiro (Eds.). Springer International Publishing, Cham, 17–32.

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

[15] Leslie Lamport. 1998. The Part-Time Parliament. *ACM Trans. Comput. Syst.* 16, 2 (May 1998), 133—-169. https://doi.org/10.1145/279227.279229

[16] Friedemann Mattern. 1988. Virtual Time and Global States of Distributed Systems. *Proceedings of the International Workshop on Parallel and Distributed Algorithms* (1988).

[17] Iulian Moraru, David G. Andersen, and Michael Kaminsky. 2013. There is More Consensus in Egalitarian Parliaments. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (Farminton, Pennsylvania) *(SOSP '13)*. Association for Computing Machinery, New York, NY, USA, 358–372. https://doi.org/10.1145/2517349.2517350

[18] Diego Ongaro and John Ousterhout. 2014. In Search of an Understandable Consensus Algorithm. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference* (Philadelphia, PA) *(USENIX ATC'14)*. USENIX Association, USA, 305–320.

[19] Daniel Peng and Frank Dabek. 2010. Large-scale Incremental Processing Using Distributed Transactions and Notifications. In *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation.*

[20] The PostgreSQL Global Development Group. 2023. Postgres-XL. http://www.postgres-xl.org/.

[21] Ohad Shacham, Yonatan Gottesman, Aran Bergman, Edward Bortnikov, Eshcar Hillel, and Idit Keidar. 2018. Taking Omid to the Clouds: Fast, Scalable Transactions for Real-Time Cloud Analytics. *Proc. VLDB Endow.* 11, 12 (Aug. 2018), 1795—-1808. https://doi.org/10.14778/3229863.3229868

[22] A.M. Sukhov, M.A. Astrakhantseva, A.K. Pervitsky, S.S. Boldyrev, and A.A. Bukatov. 2016. Generating a function for network delay. *Journal of High Speed Networks* 22, 4 (20 Oct. 2016), 321–333. https://doi.org/10.3233/JHS-160552

[23] Rebecca Taft, Irfan Sharif, Andrei Matei, Nathan VanBenschoten, Jordan Lewis, Tobias Grieger, Kai Niemi, Andy Woods, Anne Birzin, Raphael Poss, Paul Bardea, Amruta Ranade, Ben Darnell, Bram Gruneir, Justin Jaffray, Lucy Zhang, and Peter Mattis. 2020. CockroachDB: The Resilient Geo-Distributed SQL Database. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data* (Portland, OR, USA) *(SIGMOD '20)*. Association for Computing Machinery, New York, NY, USA, 1493–1509. https://doi.org/10.1145/3318464.3386134

[24] TPC. 1992. TPC Benchmark C. https://www.tpc.org/tpcc/.

[25] Misha Tyulenev, Andy Schwerin, Asya Kamsky, Randolph Tan, Alyson Cabral, and Jack Mulrow. 2019. Implementation of Cluster-Wide Logical Clock and Causal Consistency in MongoDB. In *Proceedings of the 2019 International Conference on Management of Data* (Amsterdam, Netherlands) *(SIGMOD '19)*. Association for Computing Machinery, New York, NY, USA, 636–650. https://doi.org/10.1145/3299869.3314049

[26] Zhenkun Yang, Chuanhui Yang, Fusheng Han, Mingqiang Zhuang, Bing Yang, Zhifeng Yang, Xiaojun Cheng, Yuzhong Zhao, Wenhui Shi, Huafeng Xi, Huang Yu, Bin Liu, Yi Pan, Boxue Yin, Junquan Chen, and Quanqing Xu. 2022. OceanBase: A 707 Million TpmC Distributed Relational Database System. *Proc. VLDB Endow.* 15, 12 (Aug. 2022), 3385–3397. https://doi.org/10.14778/3554821.3554830

[27] Yugabyte, Inc. 2023. YugabyteDB. https://www.yugabyte.com/.