PaxosStore: High-availability Storage Made Practical in WeChat

Jianjun Zheng[†] Qian Lin[§†*] Jiatao Xu[†] Cheng Wei[†]
Chuwei Zeng[†] Pingan Yang[†] Yunfan Zhang[†]

[†]Tencent Inc. [§]National University of Singapore

[†]{rockzheng, sunnyxu, dengoswei, eddyzeng, ypaapyyang, ifanzhang}@tencent.com
[§]linqian@comp.nus.edu.sg

ABSTRACT

In this paper, we present PaxosStore, a high-availability storage system developed to support the comprehensive business of WeChat. It employs a combinational design in the storage layer to engage multiple storage engines constructed for different storage models. PaxosStore is characteristic of extracting the Paxos-based distributed consensus protocol as a middleware that is universally accessible to the underlying multi-model storage engines. This facilitates tuning, maintaining, scaling and extending the storage engines. According to our experience in engineering practice, to achieve a practical consistent read/write protocol is far more complex than its theory. To tackle such engineering complexity, we propose a layered design of the Paxos-based storage protocol stack, where PaxosLog, the key data structure used in the protocol, is devised to bridge the programming-oriented consistent read/write to the storage-oriented Paxos procedure. Additionally, we present optimizations based on Paxos that made fault-tolerance more efficient. Discussion throughout the paper primarily focuses on pragmatic solutions that could be insightful for building practical distributed storage systems.

1. INTRODUCTION

WeChat is one of the most popular mobile apps with 700 million active users per day. Services provided by WeChat include instant messaging, social networking, mobile payment, third-party authorization, etc. The comprehensive business of WeChat is supported by its backend which consists of many functional components developed by different teams. In spite of the diversity of business logic, most of the backend components require reliable storage to support their implementation. Initially, each development team randomly adopts off-the-shelf storage systems to prototype the individual component. However, in production, the wide variety of fragmented storage systems not only costs great effort for system maintenance, but also renders these systems hard to scale. This calls for a universal storage system to serve the variety of WeChat business, and PaxosStore is the second generation[1] of such storage system in WeChat production.

The following requirements regarding storage service are common among WeChat business components. First, the well-known three V's of big data (volume, velocity, variety) are a real presence. On average, about 1.5 TB data are generated per day carrying assorted types of content including text messages, images, audios, videos, financial transactions, posts of social networking, etc. Application queries are issued at the rate of tens of thousands queries per second in the daytime. In particular, single-record accesses dominate the system usage. Second, high availability is the first-class citizen regarding the storage service. Most applications depend on PaxosStore for their implementation (e.g., point-to-point messaging, group messaging, browsing social networking posts). Availability is critical to user experience. Most applications in WeChat require the latency overhead in PaxosStore to be less than 20 ms. Such latency requirement needs to be met at the urban scale.

Along with building PaxosStore to provide high-availability storage service, we face the following challenges:

- **Effective and efficient consensus guarantee.** At the core, PaxosStore uses the Paxos algorithm [19] to handle consensus. Although the original Paxos protocol theoretically offers the consensus guarantee, its implementation complexity (e.g., the sophisticated state machines needs to be maintained and traced properly) as well as the runtime overhead (e.g., bandwidth for synchronization) render it inadequate to support the comprehensive WeChat services.
- **Elasticity and low latency.** PaxosStore is required to support low-latency read/write at the urban scale. Load surge needs to be handled properly at runtime.
- **Automatic cross-datacenter fault-tolerance.** In WeChat production, PaxosStore is deployed over thou-

[*]Part of this work was done while Qian Lin was an intern at Tencent.

[1]The first generation of WeChat storage system was based on the quorum protocol (NWR).

Proceedings of the VLDB Endowment, Vol. 10, No. 12
Copyright 2017 VLDB Endowment 2150-8097/17/08.

sands of commodity servers across multiple datacenters all over the world. Hardware failure and network outage are not rare in such large-scale system. The fault-tolerant scheme should support effective failure detection and recovery without impacting the overall efficiency of system.

PaxosStore is designed and implemented as the practical solution of high-availability storage in the WeChat backend. It employs a combinational design in the storage layer to engage multiple storage engines constructed for different storage models. PaxosStore is characteristic of extracting the Paxos-based distributed consensus protocol as a middleware that is universally accessible to the underlying multi-model storage engines. The Paxos-based storage protocol is extensible to support assorted data structures with respect to the programming model exposed to the applications. Moreover, PaxosStore adopts the leaseless design, which benefits improved system availability along with fault-tolerance.

The key contributions of this paper are summarized as follows:

- We present the design of PaxosStore, with emphasis on the construction of the consistent read/write protocol as well as how it functions. By decoupling the consensus protocol from the storage layer, PaxosStore can support multiple storage engines constructed for different storage models with high scalability.
- Based on the design of PaxosStore, we further present the fault-tolerant scheme and the details about data recovery. The described techniques have been fully applied in a real large-scale production environment, enabling PaxosStore to achieve 99.9999% availability[2] for all the production applications of WeChat.
- PaxosStore has been serving the ever growing WeChat business for more than two years. Based on this practical experience, we discuss design trade-offs as well as present results of experimental evaluation from our deployment.

The remaining content of this paper is organized as follows. We first present the detailed design and architecture of PaxosStore in Section 2, and then describe its fault-tolerant scheme and data recovery techniques in Section 3. We discuss the key points of PaxosStore implementation in Section 4. In Section 5, we conduct a performance study of PaxosStore and collect measurements from the running system. We discuss related research in Section 6, and finally conclude the paper as well as list the lessons learned from building PaxosStore in Section 7.

2. DESIGN

2.1 Overall Architecture

Figure 1 illustrates the overall architecture of PaxosStore which comprises three layers. The *programming model* provides diverse data structures exposed to the external applications. The *consensus layer* implements the Paxos-based storage protocol. The *storage layer* consists of multiple storage engines implemented based on different storage models to fulfill various kinds of performance requirements. The architecture of PaxosStore differs from the traditional design

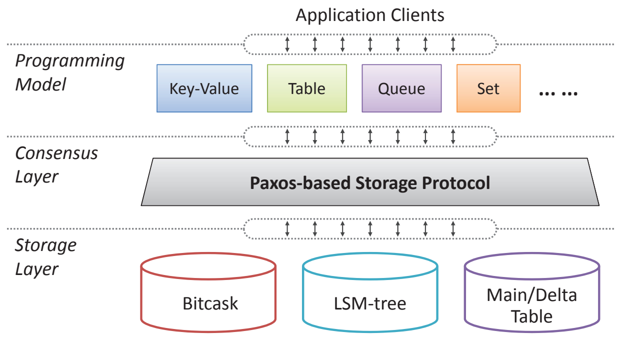


Figure 1: Overall architecture of PaxosStore.

of storage system mainly in that it explicitly extracts the implementation of the consensus protocol as a middleware and universally provides data consensus guarantee as a service to all the underlying storage engines.

Conventional distributed storage systems, more often than not, are built based on their individual single storage model and tweak the design and implementation to fulfill the various application demands. However, these tend to introduce complicated trade-offs between different components. Although assembling multiple off-the-shelf storage systems to form a comprehensive system can meet the diversity of storage requirements, it usually makes the entire system difficult to maintain and less scalable. Moreover, each storage system embeds its implementation of data consensus protocol into the storage model, and the consequential divide-and-conquer approach for consensus could be error-prone. This is because the consensus guarantee of the overall system is determined by the conjunction of consensus achievements provided by the individual subsystems. Furthermore, applications with cross-model data accesses (i.e., across multiple storage subsystems) can hardly leverage any consensus support from the underlying subsystems, but have to separately implement the consensus protocol for such cross-model scenario by their own.

PaxosStore applies a *combinational storage* design such that multiple storage models are implemented in the storage layer with the only engineering concern of high availability, leaving alone the consensus concern to the consensus layer. This facilitates each storage engine as well as the entire storage layer to scale on demand. As the implementation of the consensus protocol is decoupled from the storage engines, universal service of providing data consensus guarantee can be shared among all the supported storage models. This also makes the consensus of cross-model data accesses easy to achieve.

While the design and implementation of the programming model layer are straightforward in terms of engineering, in this section, we mainly present the detailed design of the consensus layer and storage layer of PaxosStore.

2.2 Consensus Layer

The distributed storage protocol stack of PaxosStore consists of three bottom-up layers as shown in Figure 2.

2.2.1 *Paxos*

The storage protocol relies on the Paxos algorithm [19] to determine the value of data object that is replicated over

[2]Such result comes from the statistics for six months of operating data.

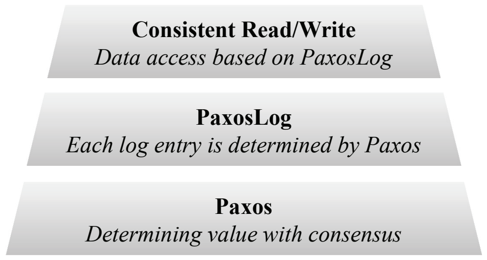

Consistent Read/Write
Data access based on PaxosLog
PaxosLog
Each log entry is determined by Paxos
Paxos
Determining value with consensus

Figure 2: Storage protocol stack.

multiple nodes. In theory, a Paxos procedure contains two phases in sequence: the *prepare* phase for making a preliminary agreement, and the *accept* phase for reaching the eventual consensus. By convention, the Paxos procedure is depicted using state machines. However, the excessive states and complicated transitions not only defeat elegant implementation but also lead to low-performance runtime.

In the development of PaxosStore, we deprecate the state machine based Paxos interpretation, and alternatively interpret the Paxos procedure using *semi-symmetry message passing*. To this end, we first define symbols that are essential for our algorithmic interpretation. We define a proposal $\mathcal{P}$ as

$$\mathcal{P} = (n,\ v)$$

where n is the proposal number and v is the proposal value.

Let N_X denote a node whose node ID is X. Let r be a data record and r_X be the r replica at node N_X. Let $\mathcal{S}_Y^X$ be the *state* of r_Y from the view of N_X, and it is defined as

$$\mathcal{S}_Y^X = (m,\ \mathcal{P})$$

where m is the least proposal number that r_Y promises to accept, and $\mathcal{P}$ is the latest proposal that r_Y has accepted. In other words, any proposal with $n < m$ will be rejected by r_Y. Specially, we term $\mathcal{S}_Y^X (X \neq Y)$ the *view state* and $\mathcal{S}_X^X$ the *actual state*. Note that the view states and the actual state of a data replica could be different under certain situations, and synchronization of the states relies on the Paxos procedure [19].

In PaxosStore, consensus between data replicas is achieved by mutually exchanging the view states and the actual states, and the corresponding message $\mathbb{M}_{X \to Y}$ sending from N_X to N_Y is defined as

$$\mathbb{M}_{X \to Y} = \left\{ \mathcal{S}_X^X,\ \mathcal{S}_Y^X \right\}$$

That is, node N_X sends the actual state of r_X and its view state of r_Y to node N_Y. Unlike other Paxos implementations which employ many types of messages, PaxosStore uses a universal message format as defined above in its Paxos implementation.

Algorithm 1 summarizes the Paxos implementation in PaxosStore. Initially, when a node (e.g., N_A) receives a write request, it starts the prepare phase by invoking `Issue(`m_i`)` where m_i is a node-specific intention proposal number. N_A pushes the actual state of r_A and its view state of r_X to each remote node N_X. Whenever a node (including N_A) receives a message, the function `OnMessage()` will be invoked to update the local states with respect to the actual state of remote replica contained in the received message (i.e., `UpdateStates()`). Apart from the conditional update

Algorithm 1: Paxos implementation in PaxosStore.

```
    Input: intention proposal number $m_i$
 1: Procedure Issue($m_i$):                         /* invoked at $N_A$ */
 2:     $\mathcal{S}_A^A \leftarrow$ actual state of $r_A$
 3:     if $\mathcal{S}_A^A.m < m_i$ then
 4:         $\mathcal{S}_A^A.m \leftarrow m_i$
 5:         write $\mathcal{S}_A^A$ to the PaxosLog entry of $r_A$
 6:         foreach remote replica node $N_X$ do
 7:             send $\mathbb{M}_{A \to X}$

    Input: proposal $\mathcal{P}_i$ with $\mathcal{P}_i.n = m_i$
 8: Procedure Issue($\mathcal{P}_i$):                /* invoked at $N_A$ */
 9:     $\mathcal{S}_A^A \leftarrow$ actual state of $r_A$
10:     $\mathbb{S}^A \leftarrow$ all the states of $r$ maintained at $N_A$
11:     if $\left|\left\{\forall \mathcal{S}_X^A \in \mathbb{S}^A \mid \mathcal{S}_X^A.m = \mathcal{P}_i.n\right\}\right| \times 2 > \left|\mathbb{S}^A\right|$ then
12:         if $\left|\left\{\forall \mathcal{S}_X^A \in \mathbb{S}^A \mid \mathcal{S}_X^A.\mathcal{P}.v \neq \mathit{null}\right\}\right| > 0$ then
13:             $\mathcal{P}' \leftarrow$ the proposal with maximum $\mathcal{P}.n$ in $\mathbb{S}^A$
14:             $\mathcal{S}_A^A.\mathcal{P} \leftarrow (\mathcal{P}_i.n,\ \mathcal{P}'.v)$
15:         else
16:             $\mathcal{S}_A^A.\mathcal{P} \leftarrow \mathcal{P}_i$
17:         write $\mathcal{S}_A^A$ to the PaxosLog entry of $r_A$
18:         foreach remote replica node $N_X$ do
19:             send $\mathbb{M}_{A \to X}$

    Input: message $\mathbb{M}_{X \to Y}$ sent from $N_X$ to $N_Y$
20: Procedure OnMessage($\mathbb{M}_{X \to Y}$):     /* invoked at $N_Y$ */
21:     $\mathcal{S}_X^X,\ \mathcal{S}_Y^X \leftarrow \mathbb{M}_{X \to Y}$
22:     UpdateStates($Y$, $\mathcal{S}_X^X$)
23:     if $\mathcal{S}_Y^Y$ is changed then
24:         write $\mathcal{S}_Y^Y$ to the PaxosLog entry of $r_Y$
25:         if IsValueChosen($Y$) is true then commit
26:     if $\mathcal{S}_Y^X.m < \mathcal{S}_Y^Y.m$ or $\mathcal{S}_Y^X.\mathcal{P}.n < \mathcal{S}_Y^Y.\mathcal{P}.n$ then
27:         send $\mathbb{M}_{Y \to X}$

    Input: node ID $Y$, actual state $\mathcal{S}_X^X$ of $r_X$
28: Function UpdateStates($Y$, $\mathcal{S}_X^X$):   /* invoked at $N_Y$ */
29:     $\mathcal{S}_X^Y \leftarrow$ view state of $r_X$ stored in $N_Y$
30:     $\mathcal{S}_Y^Y \leftarrow$ actual state of $r_Y$
31:     if $\mathcal{S}_X^Y.m < \mathcal{S}_X^X.m$ then $\mathcal{S}_X^Y.m \leftarrow \mathcal{S}_X^X.m$
32:     if $\mathcal{S}_X^Y.\mathcal{P}.n < \mathcal{S}_X^X.\mathcal{P}.n$ then $\mathcal{S}_X^Y.\mathcal{P} \leftarrow \mathcal{S}_X^X.\mathcal{P}$
33:     if $\mathcal{S}_Y^Y.m < \mathcal{S}_X^X.m$ then $\mathcal{S}_Y^Y.m \leftarrow \mathcal{S}_X^X.m$
34:     if $\mathcal{S}_Y^Y.m \leq \mathcal{S}_X^X.\mathcal{P}.n$ then $\mathcal{S}_Y^Y.\mathcal{P} \leftarrow \mathcal{S}_X^X.\mathcal{P}$

    Input: node ID $Y$
    Output: whether the proposals in $N_Y$ form a majority
35: Function IsValueChosen($Y$):                     /* invoked at $N_Y$ */
36:     $\mathbb{S}^Y \leftarrow$ all the states of $r$ maintained at $N_Y$
37:     $n' \leftarrow$ occurrence count of the most frequent $\mathcal{P}.n$ in $\mathbb{S}^Y$
38:     return $n' \times 2 > \left|\mathbb{S}^Y\right|$
```

of states, the `OnMessage()` routine involves checking whether the proposal value can be chosen based on the criterion of reaching the majority (i.e., `IsValueChosen()`). In addition, the receiver node will send back its states if any local state update is found to be newer. Note that N_A counts for a timeout to check whether the proposing m_i reaches the majority. If the majority cannot be reached within the time-

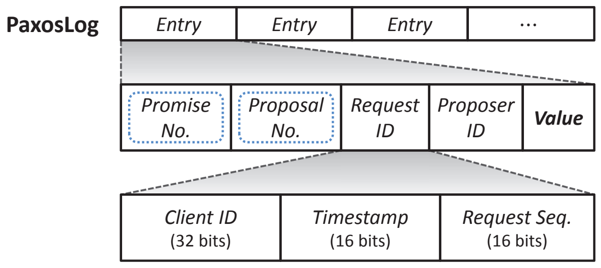


Figure 3: The PaxosLog structure.

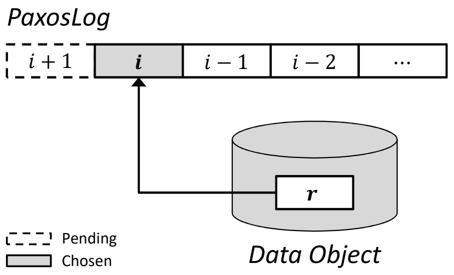


Figure 4: PaxosLog + Data object.

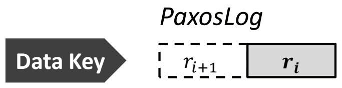


Figure 5: PaxosLog-as-value for key-value storage.

out, N_A will invoke `Issue(`m_i`)` again to restart the Paxos procedure with a larger m_i. Only after the majority of the proposing m_i is reached before timeout, N_A will invoke `Issue(`$\mathcal{P}_i$`)` to start the accept phase with the proposal $\mathcal{P}_i$, where $\mathcal{P}_i.n = m_i$ is held and $\mathcal{P}_i.v$ is the value given by the write request. Subsequently, the same process of state exchange through message passing operates until the majority condition is satisfied (i.e., chosen).

As can be seen, the uniform processing routine (i.e., performed by `OnMessage()`) with respect to the identical format of messages autonomously drives the Paxos procedure to operate till consensus reached. This not only simplifies the Paxos implementation but also benefits the locality of message processing. In PaxosStore, the Paxos protocol depicted as Algorithm 1 is implemented in about 800 lines of C++ code, with robustness proven by its successful deployment in WeChat production.

2.2.2 *PaxosLog*

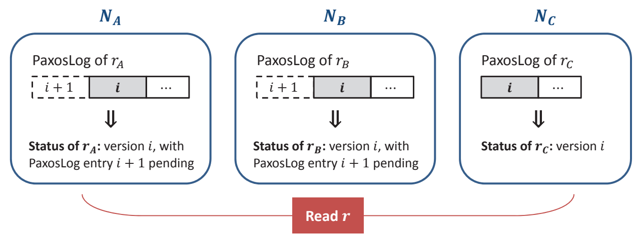
PaxosLog serves as the write-ahead log [3] for data update in PaxosStore. Figure 3 shows the data structure of PaxosLog, where each log entry is determined by the Paxos algorithm and indexed by an immutable sequence number, namely *entry ID*, that increases monotonically within the PaxosLog. The maximal entry ID of a PaxosLog entry that has been applied to the physical storage represents the version of the associated data replica. Conceptually, a PaxosLog instance could accommodate infinite amount of log entries that are generated along with the Paxos procedures. In practice, obsolete log entries are asynchronously evicted by following the LRU policy. Apart from the data value, each PaxosLog entry contains the Paxos-related meta data of the promise number m and the proposal number n, which are used for the Paxos procedure as discussed in Section 2.2.1 but discarded once the log entry is finalized (marked by dashed boxes, indicating the mutability and volatility). In addition, as multiple replica hosts of the same data may issue write requests simultaneously but only one of these requests will be eventually accepted, a proposer ID is attached to each chosen value to indicate the value proposer. The proposer ID is a 32-bit machine ID that uniquely identifies a node in the datacenter. It is used for the *pre-preparing optimization* [2] where the current write can skip the prepare/promise phase if it shares the same origin of request with the previous write, in anticipation of the locality of write requests. Moreover, a request ID is constructed to uniquely identify the write request associated with the data value. It is used for preventing duplicated Paxos procedures caused by false positive of failover (details are discussed in Section 3.3). Specifically, a request ID consists of three segment fields: a 32-bit client ID (e.g., IPv4 address) identifying the external client that issued the write request, a 16-bit timestamp in seconds assigned by the client's local clock for rough time reference, and a 16-bit sequence number assigned by the client's request counter for distinguishing the write requests within the same second.

According to the mechanism of write-ahead log for data update [3], the PaxosLog entry must be settled before the corresponding value change is materialized to the storage. In general, the PaxosLog storage and the data object storage are separated, as shown in Figure 4. Consequently, for each data update, two write I/Os need to be completed in ordered: one for writing the PaxosLog entry, followed by the other one for updating the data object.

PaxosLog for Key-Value Data. Considering the key-value storage, we make two optimizations to simplify the construction and manipulation of PaxosLog.

First, we cancel out the data value storage and make key-value access through PaxosLog instead, namely *PaxosLog-as-value*. In the Paxos-based key-value storage, one data key is mapped to one data value and associated with one PaxosLog. Such one-to-one mapping renders storing both the PaxosLog entry and the data value redundant, as each PaxosLog entry already contains the entire image of data object (i.e., the single data value). Therefore, we can refer to the PaxosLog entry for the data value and get rid of storing the data value separately. This makes only one write I/O (i.e., to PaxosLog only) for every key-value update and thus saves the I/O bandwidth.

Second, we trim the PaxosLog for key-value data by retaining two log entries only: one is the latest chosen log entry, and the other is the pending log entry with respect to the ongoing Paxos procedure. In other words, log entries older than the latest chosen one are considered obsolete and therefore can be thrown away. The resulting PaxosLog structure is illustrated in Figure 5. As can be seen, PaxosLog for key-value data becomes compact and of constant length, i.e., only two entries are maintained in each PaxosLog. Since the PaxosLog does not grow in size, memory and disk space allocation for PaxosLog along with data updates can be eliminated, yielding storage and computation resources. Moreover, key-value data recovery can be simplified and consequently efficient. As the latest chosen PaxosLog entry contains the entire image of data object (i.e., the data value),



N_A
N_B
N_C
PaxosLog of r_A
PaxosLog of r_B
PaxosLog of r_C
$i+1$
i
...
⇓
Status of r_A: version i, with PaxosLog entry $i+1$ pending
Status of r_B: version i, with PaxosLog entry $i+1$ pending
Status of r_C: version i
Read r

Figure 6: Sample scenario of read processing in PaxosStore.

it can be referred to for the recovery of data value when failure happens. In contrast, data recovery of normal data object needs to redo all the historical log entries since the last checkpoint maintained in the general PaxosLog (i.e., as shown in Figure 4), especially for collection-oriented data structures such as list, queue and set.

2.2.3 *Consistent Read/Write*

PaxosStore is designed as a multi-homed system [13] that runs actively on multiple datacenters around the clock, and meanwhile employs the Paxos protocol for operational consensus in a leaseless fashion, i.e., each node can handle data updates equally[3]. Specifically, each data access (i.e., read/write) is constantly processed through the cooperation of all its replica nodes and relies on the PaxosLog to achieve strong consistency of data.

Consistent Read. Consistent read depends the criterion of returning the value of data replica with the maximum version that necessarily reaches the majority among all the related replicas. In other words, for a data object r, system reads its value from any of the up-to-date r replicas and meanwhile these up-to-date replicas are required to dominate the total replicas of r. For data that are read-frequent, the above criterion tends to be satisfied most of the time, and thus the corresponding read processing could be very lightweight with rare failures. Specifically, for a node storing a replica of data object r, after it receives the read request on r and figures out the current version of its r replica is i, it queries all the other replicas whether their PaxosLog entries of entry ID $i+1$ are inactive, i.e., either absent or $\mathcal{P}.n = 0$ if entry does exist. If the majority of the $i+1$ PaxosLog entries are inactive, then the node can immediately respond to the read request with its r replica of version i.

Read fails if the up-to-date replica version cannot account for the majority, especially when data contention is high. In particular, data replicas may be in asynchronous states due to the ongoing Paxos procedure when the read request is processed. For example, in Figure 6, a read request on data object r is being processed while its replicas residing in nodes N_A, N_B and N_C are being updated as per another write request on r. Specially, the three replicas r_A, r_B and r_C are all of version i, but both r_A and r_B have PaxosLog entries with entry ID $i+1$ corresponding to the in-progress Paxos procedure for the write request. As a consequence, whether version i is the latest valid version cannot be determined from the majority since some $i+1$ PaxosLog entry may be chosen but not detected. To address this issue, an immediate approach is to query all the replica versions through a trial Paxos procedure issued with $m_i = i+1$. The trial Paxos procedure does not correspond to any substantive write operation, i.e., leaving $\mathcal{P}.v$ to be a blank *no-op* value. If the trial Paxos procedure completes with success, all the $i+1$ PaxosLog entries of r replicas are inferred to be inactive, and thus version i is validated to be up-to-date. Using the trial Paxos procedure to resolve the ambiguity due to read-write contention can guarantee strong-consistency read operations in PaxosStore.

Consistent Write. Consistent write relies on the Paxos procedure as described in Algorithm 1. Although Paxos guarantees the consensus once reached, its liveness could be an issue—it may not terminate in some cases due to write-write contention. To tackle the problem of livelock, PaxosStore pessimistically prevents different replicas of the same data from issuing Paxos procedures simultaneously. To this end, when a replica receives a Paxos message sent from a remote replica indicating an ongoing Paxos procedure, a time window is set to the local replica such that the replica promises not to issue any concurrent Paxos procedure within the time window. In other words, succeeding write requests to the replica will be postponed for its processing till the expiry of the time window. This is in anticipation of the processing of the preceding write request completed within the time window.

Towards performance gain in practice, PaxosStore further enforces a constraint such that only the prespecified replica nodes (e.g., N_A and N_B but not N_C) can process the read/write requests. Therefore, only those prespecified replicas need to be checked for their versions and status. Moreover, those unspecified replica nodes (e.g., N_C) apply the chosen PaxosLog entries to the physical storage in a batch manner, rather than applying them timely and individually. We term this as *PaxosLog-entry batched applying*. Note that batching is only performed on log entries belonging to the same PaxosLog. The strategy of PaxosLog-entry batched applying can effectively reduce the cost of applying PaxosLog entries to the storage when the workload appears to be write-frequent.

2.3 Storage Layer

Applications in WeChat production have raised kinds of data access requirements as well as performance concerns. After rounds of restructuring our previous generation storage system, we realized that it was essential to support multiple storage models in the storage layer. This motivates the design of storage layer of PaxosStore to involve multiple storage engines built based on different storage models. In particular, Bitcask [31] and LSM-tree [26, 30] are the two primary models used in PaxosStore. Both models are designed for key-value storage: the Bitcask model is preferable to point queries, whereas the LSM-tree model is superior in range queries. We implement all the storage engines in the storage layer of PaxosStore with the main concerns of availability, efficiency and scalability, excluding consensus (as handled by the consensus layer). Extracting the consensus protocol out of the storage layer and making it as a middleware greatly ease the development, tuning and maintenance of storage engines in the storage layer of PaxosStore. The storage engines can not only be tuned individually but also work collaboratively so that they support more flexible

[3]In contrast, lease-based Paxos systems only allow those elected leader nodes with time-based leases to issue the Paxos procedure. Leader election is performed periodically, or bypassed if lease renewal is granted to the same leader.

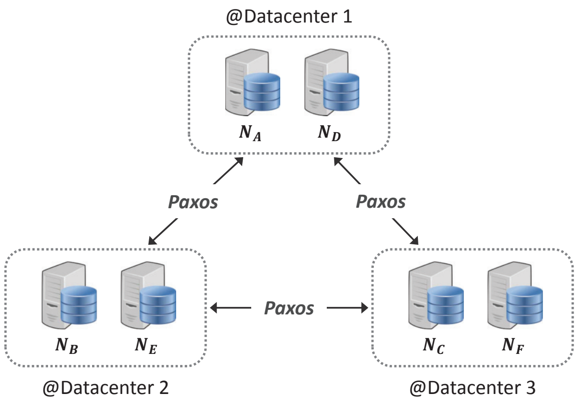


Figure 7: Sample intra-region deployment of PaxosStore, with $\mathcal{C} = 2$.

options to meet the application/business specific demands.

The most frequently used physical data structures[4] in PaxosStore are key-value pair and relational table. Key-value storage is naturally supported by the Bitcask and LSM-tree based engines. Fundamentally, tables in PaxosStore are also stored as key-value pairs where each table represents a value and is indexed by a unique key. However, naively storing tables as normal key-value pairs would cause performance issue. This is because most of the tables are read-frequent in reality. In particular, each table usually contains thousands of tuples[5], but most of the WeChat applications often touch only one or two tuples within one-time access. As a consequence, tables thrashing between disk/SSD and memory along with read/write operations can degrade system performance significantly. To address this issue, PaxosStore adopts the *differential update* technique [33, 15, 9] to reduce the overhead of table update. To this end, each table is split into two meta tables: a read-optimized *main-table* which manages the majority data of table, and a write-friendly *delta-table* which manages table changes (e.g., update, insertion and deletion) that have been made recently. As a consequence, table access is through the view of the main-table with all corresponding differences from the delta-table combined on-the-fly. Furthermore, in order to keep the delta-table small (i.e., to retain the write-friendly property), its containing changes are periodically merged into the main-table [33].

3. FAULT TOLERANCE AND AVAILABILITY

Hardware failure and network outage are the two main sources of failure in WeChat production. In particular, with PaxosStore deployed at the scale of thousands of nodes, the failure rate of storage nodes is 0.0007 per day on average, and planned and unplanned network outages happen every few months.

[4]Here "physical" refers to the actual storage in the storage layer. This is to distinguish it from "semantic" data structure in the programming model. Note that, in PaxosStore, different semantic data structures may be backed by the same physical data structure in terms of implementation.

[5]In WeChat applications, tables are generally created as per user account.

3.1 Fault-tolerant Scheme

As a globally available mobile app, WeChat relies on PaxosStore to support its wide accessibility of service with low latency. In WeChat production, PaxosStore is deployed across multiple datacenters scattered over the world. Datacenters are grouped into disjointed sets, each of which consists of three distant datacenters that cover a geographical region of WeChat service. PaxosStore employs consistent hashing [32] for inter-region data placement. For intra-region deployment (e.g., in a megalopolis), each of the three datacenters maintains one replica of every data object hashed to the region. This infers the replication factor of PaxosStore equals 3 under such deployment setting[6]. Inside a datacenter, PaxosStore nodes are partitioned into uniform-size mini-clusters. We define the size of a mini-cluster to be $\mathcal{C}$. Each mini-cluster in a datacenter is uniquely correlated to two remote mini-clusters separately hosted by the other two datacenters. Figure 7 illustrates a sample mini-cluster group of PaxosStore, where each mini-cluster (denoted by a dashed box) comprises two nodes (i.e., $\mathcal{C} = 2$). With such correlation of mini-clusters, PaxosStore enforces data of one node to be replicated to the two remote mini-clusters with uniform distribution within each mini-cluster. In other words, replicas of a data object are distributed over three distant mini-clusters with node combination randomly chosen under the constraint of the mini-cluster grouping. For example, in Figure 7, replicas of a data object r could be placed at nodes N_A, N_B and N_C, while replicas of another data object r' belonging to the same mini-cluster group may be placed at nodes N_A, N_E and N_F. As a consequence, data hosted by N_A are replicated to N_B and N_E (resp. N_C and N_F) evenly, i.e., each of N_B and N_E (resp. N_C and N_F) stores statistically half of the data replicas mirrored to N_A. Similar scheme also applies to other nodes in the same mini-cluster group.

The above strategy of data replication based on mini-cluster grouping is to mitigate the burden of instant soaring workload due to load distribution upon node failure. When a node fails, its workload in anticipation has to be distributed over the $2\mathcal{C}$ nodes in the related remote mini-clusters, each of which takes over about $1/2\mathcal{C}$ of the extra workload. For example, if N_A fails in Figure 7, its workload (i.e., serving query requests) will be handed over to N_B, N_C, N_E and N_F with each taking about 25% of the load. By doing so, occurrence of failure tends not to subsequently saturate or even overload the load receiver for the sake of fault tolerance in PaxosStore. Obviously, effectiveness of the above strategy is affected by the configuration of the mini-cluster size apart from the replication factor. Although larger $\mathcal{C}$ benefits higher tolerance of load surge, it also degrades the effectiveness of performing batched network communication, where a node's outgoing data with common destination node are periodically sent in a batch manner. Hence, the choice of appropriate $\mathcal{C}$ is dependent on the trade-off between the tolerance of load surge and the overhead of inter-node communication.

Under the aforementioned deployment scheme, consistent read/write operations with fault-tolerant guarantee follow the Paxos-based mechanism as described in Section 2. By

[6]Data replication in PaxosStore is constrained within the region. Network latency between intra-region datacenters is $1 \sim 3$ ms on average.

default, for each data object r, we enforce a constraint such that all the read/write requests for r are routed to a pre-specified replica (e.g., at node N_A) for processing and only transferred to another replica (e.g., at N_B) if a fault or failure is detected at N_A. Comparing with routing request to arbitrary replica at random, this strategy can greatly reduce the write-write conflicts due to contention[7].

3.2 Data Recovery

Unlike traditional DBMS synchronously checkpointing the holistic storage snapshot, checkpointing in PaxosStore is fine-grained. In fact, checkpointing in PaxosStore is implicit, as data recovery is based on data version contained in the PaxosLog or the data image in the persistent storage. To elaborate, let us suppose a data record r is replicated to three nodes N_A, N_B and N_C with the corresponding r replicas denoted by r_A, r_B and r_C respectively. Now suppose node N_A fails and later resumes, and r is updated (maybe multiple times) during the recovery of N_A. Note that the update of r can still be completed in spite of the failure of N_A, because the majority of r replicas (in N_B and N_C) stay alive. As a consequence, the version of r_A lags behind that of r_B and r_C, i.e., r_A considered to be obsolete.

In PaxosStore, version of data replica (i.e., the maximal entry ID of PaxosLog entry that has been applied to the physical storage) represents a checkpoint. Let $\nu(r)$ be the version of data r. In the above example, suppose $\nu(r_A) = i$ and $\nu(r) = \nu(r_B) = \nu(r_C) = j$ with $i < j$. Figure 8 demonstrates the options of data recovery approaches in PaxosStore. First, if the PaxosLog of r_B (or r_C) contains the log entries of versions from $i + 1$ to j, then these delta log entries will be sent to N_A to synchronize r_A up-to-date (i.e., to version j). Second, if the above log entries are discarded in both r_B and r_C, then r_A has to refer to the r image in N_B or N_C for its recovery. In particular, for data objects whose elements are append-only[8], we extract the delta updates from r_B (or r_C) with respect to r_A, and send them to N_A to supplement r_A, making r_A up-to-date. Third, if the incremental PaxosLog entries are absent and meanwhile no append-only property of data object can be exploited, then the whole data image of r_B (or r_C) will be sent to N_A to replace the obsolete r_A. For the special case of key-value storage, we only need to send the PaxosLog entry of version j and ignore those log entries of version less than j. This is because, for key-value data, each PaxosLog entry contains the entire value of the data object. This further corroborates the optimization of concise PaxosLog for key-value storage as described in Section 2.2.2.

The above fine-grained checkpointing scheme further enables autonomous data recovery in case of data replica loss due to node failure. Specifically, the recovery of data replica is actualized by the consistent read/write semantics. After a failed node is restored to its state before the crash, data replicas contained in the node may be obsolete or even missing. PaxosStore does not immediately recover the invalid data replicas upon node recovery, but recover them when the corresponding data are subsequently accessed. For each data access in PaxosStore, the Paxos-based distributed storage protocol guarantees all the active replicas of a data record are synchronized to the latest version upon the read/write operation. We term such recovery strategy *lazy recovery*. Let us consider again the aforementioned example with data r instantiated as three replicas r_A, r_B and r_C where r_A is corrupted at the time of N_A just resumed. According to the lazy recovery, PaxosStore does not recover r_A until the next access to r. In other words, r_A will be automatically recovered to the latest version upon the first access to r after N_A has revived. Lazy recovery guarantees all the replicas belonging to the same data record are always synchronized to the up-to-date version as long as the data record can be successfully read or written. Meanwhile, it amortizes the recovery overhead to individual data accesses, leading to the avoidance of system downtime during data recovery and thus benefiting high availability of the system.

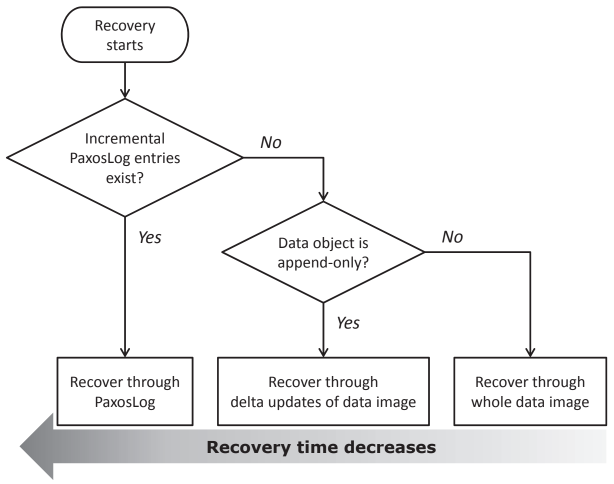

Recovery starts
Incremental PaxosLog entries exist?
No
Yes
Data object is append-only?
No
Yes
Recover through PaxosLog
Recover through delta updates of data image
Recover through whole data image
Recovery time decreases

Figure 8: Data recovery approaches in PaxosStore.

3.3 Optimizations

Failover Reads. When an abnormal read due to obsolete or lost data replica is detected (i.e., for r_A), PaxosStore will trigger an immediate recovery for r_A with a time τ deadline where τ is configurable in tuning (e.g., shorter than the promise of specific response time with respect to the application). If the recovery completes within τ, the healed r_A is used to respond to the read request. Otherwise, the read request will be forwarded and handed over by another data replica (i.e., r_B); meanwhile, the recovery of r_A continues in an asynchronous process.

Preventing Duplicated Processing. Duplicated processing may occur in subtle cases, most of which are due to message loss or network delay. Taking a data record r replicated to three nodes N_A, N_B and N_C for example, consider the following events happening in order:

E1. The client issues a write request of r, which is routed to N_A for processing.

[7]For some applications where data are read-frequent and the access locality is clearly known upfront, we allow each corresponding WeChat backend server prioritizes PaxosStore nodes within the same datacenter for its read accesses to PaxosStore, and alternatively redirects the read requests to remote datacenters in case of failure occurring at the local datacenter. This could save the bandwidth of inter-datacenter communication to some extent, but may increase the chance of potential read-write conflict.

[8]Data insertion is by appending data element to the head of data object. And the obsolete data elements may be garbage collected from the tail of data object.

E2. Upon receiving the write request, N_A launches a Paxos procedure for updating r, and send back the response to the client immediately after the update of r is chosen (i.e., successful completion of the Paxos procedure). Unfortunately, the response message is delayed in network.

E3. As the client obtains no response within timeout, it reissues the request by sending it to N_B.

E4. As requested, N_B runs another Paxos procedure to update r and then informs the client upon the update completion.

E5. The client eventually receives both response messages from N_A and N_B. Meanwhile, r in the storage was updated twice accordingly.

For non-idempotent operations (e.g., insertion and deletion), duplicated processing could ruin the data. As can be seen from the above example, the duplicated processing is caused by the unexpected network delay (in E2) and the consequent false positive of resending the request (in E3). To address this problem, PaxosStore employs a simple but practical approach to prevent duplicated processing. It uses the request ID (as shown in Figure 3 and described in Section 2.2.2) to filter out duplicated requests at the storage side. The request ID is generated by the client and sent together with the write request, and checked by the PaxosStore node to detect potential duplication of actual request handling. To elaborate, let us consider again the above example. In E4, when N_B receives the client's request, it scans the $PaxosLog$ of r_B to check whether the request ID exists among the entries. If the request ID is found, this implies another node (e.g., N_A) already completed the Paxos-based update with respect to the received request. Therefore, N_B will not trigger another Paxos procedure but directly reply to the client. Otherwise, N_B can confirm no previous Paxos procedure has ever been done for this request, and consequentially runs a Paxos procedure to handle the update. As the request IDs are materialized to the PaxosLog and the PaxosLog entries are shared among replicas with strong consensus, duplicated processing can be effectively eliminated by the above strategy.

For optimization, each PaxosStore node buffers the recent request IDs upon completion of every Paxos procedure to avoid frequently scanning PaxosLog entries. The FIFO (first-in, first-out) based eviction policy of the buffering should guarantee that the buffered request IDs stay longer than the failover time. To fulfill such requirement in practice, it is advised to optimistically configure the buffering duration as long as the cost of memory space for buffering is affordable. For example, in WeChat production, we observed the failover time (including the failure detection time) is less than 1 second for over 90% of the failure cases and 5 seconds as maximum occurring in rare cases, and empirically configured the minimum buffering duration as 5 minutes.

4. IMPLEMENTATION

The implementation of PaxosStore in WeChat production involves many composite procedures that cooperate to perform diverse functionality. Towards high performance, the composite procedures are often carried out by the asynchronous threads. However, these asynchronous threads introduce tremendous amount of asynchronous states, which greatly complex the system in terms of engineering. Although we may alternatively implement the composite procedures in the synchronous manner to get rid of the above complexity, system performance would be degraded by orders of magnitude comparing with the asynchronous implementation. In order to ease the development of composite procedure with efficiency, the implementation of PaxosStore highly exploits the coroutine framework to program the asynchronous procedures in the synchronous paradigm. To this end, we have developed a fine-tuned coroutine library, namely `libco`, and made it open source[9]. The coroutine-based asynchronous programming not only saves the engineering effort, but also helps to ensure the correctness of asynchronous logic, improving both the productivity and quality of the PaxosStore development.

Note that coroutine is only suitable for asynchronous procedure that contains single branch of asynchronous execution, i.e., the asynchronous invocation being deterministic. Most of the asynchronous procedures in PaxosStore belong to this category, such as the invocations for consensus service, data materialization, and batched read/write operations. However, the Paxos implementation as described in Section 2.2.1 contains multiple branches of asynchronous execution. Therefore, it has to be programmed in an explicit asynchronous way. Except that, most of the other routines in the storage protocol stack and fault-tolerant scheme of PaxosStore (e.g., applying PaxosLog to the physical storage, and exploiting PaxosLog for data recovery) still leverage coroutine to simplify their implementation.

For inter-node communication, each server sends network packets in a batch manner, which can greatly save the network bandwidth (especially for the inter-datacenter network). Moreover, each server maintains multiple TCP sockets paired with the remote servers belonging to the same mini-cluster group. Each TCP socket takes turns to function for emitting the batched packets. By doing so, the workload of batched packet processing (e.g., serialization/deserialization) can be distributed over multiple threads, each of which corresponds to a distinct TCP socket. In WeChat production, we empirically configure the batch interval as 0.5 ms and the number of TCP sockets per server as 4.

5. EVALUATION

PaxosStore has been fully deployed in WeChat production for more than two years, and it has proven to be considerably more reliable than our previous generation system based on the quorum protocol [7]. During this period, PaxosStore has survived assorted outages of datacenters and different levels of load surge, with no significant impact on the end-to-end latency. In this section, we first provide detailed performance evaluation corresponding to the design and optimizations of PaxosStore presented in this paper. And then we conduct case studies of PaxosStore in WeChat production.

5.1 Experimental Setup

We run experiments on PaxosStore with its in-production deployment across three datacenters. Each node in the datacenter is equipped with an Intel Xeon E5-2620 v3 @ 2.4 GHz CPU, 32 GB DDR3 memory, and 1.5 TB SSD configured in RAID-10. All nodes within the same datacenter

[9]`https://github.com/Tencent/libco`

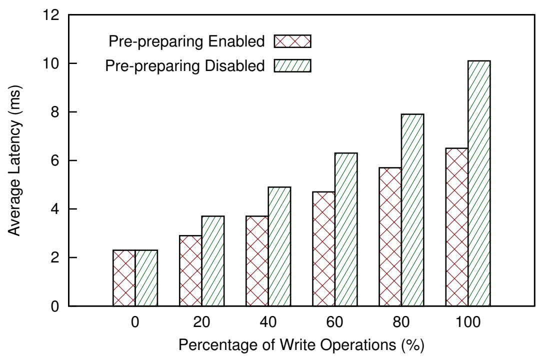


Figure 9: Overall read/write performance of PaxosStore and effectiveness of the pre-preparing optimization.

are connected by 1 Gigabit Ethernet. The inter-datacenter network latency is around 1 ms.

Experiments are run with synthetic workloads. Specifically, different mixture of read and write requests are issued on 10 million tables in the form of point queries. The size of time window for pessimistic prevention of write-write contention between data replicas is set to 5 ms, the same configuration adopted in WeChat production.

5.2 Latency

PaxosStore can afford to process tens of billions of queries per minute in real WeChat business. This is achieved by its design of supporting fast Paxos procedure as well as various optimizations. We measure the mean latency in a real deployment of PaxosStore and the results are shown in Figure 9. We run different workloads by varying the percentage of write operations for multiple testing instances, in each of which system is saturated by the workload to simulate the peak-hour scenario. In particular, we compare the performance with respect to the pre-preparing optimization which is enabled in PaxosStore by default. Figure 9 shows that PaxosStore can generally support consistent read/write with latency less than 10 ms, even under the extreme workload with 100% write operations. In WeChat business, most of the applications specify the acceptable latency incurred by the storage is 20 ms, and obviously PaxosStore can sufficiently satisfy such relaxed requirement. Moreover, the pre-preparing optimization effects latency decreasing. This is due to the intention design of bypassing the prepare phase and directly starting the accept phase in the Paxos procedure whenever possible (e.g, data accesses being clustered). In practice, PaxosStore implicitly prioritizes data replicas in terms of accesses so that read/write operations of any given data object are clustered to a stated replica most of the time, making the pre-preparing optimization pretty helpful.

For WeChat applications, PaxosLog is generally instantiated on the granule of user, as data accesses are based on the user ID most of the time. As a consequence, read-write contention in WeChat production is rare. This makes the read operation very efficient in the overwhelming majority of cases, since the trial Paxos procedure for read-write conflict resolution turns out to be trivial under such circumstances.

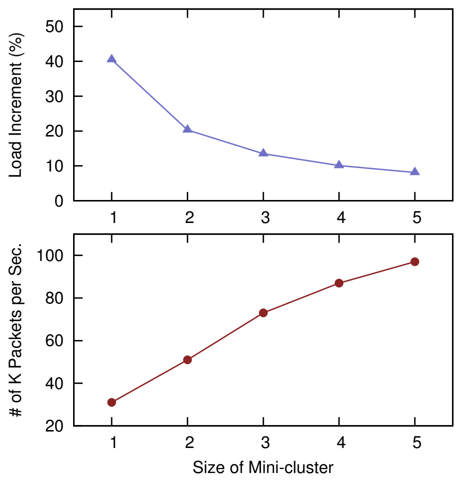


Figure 10: Scaling the size of mini-cluster.

5.3 Fault Tolerance

PaxosStore is designed for high-availability storage service. Its fault-tolerant scheme as described in Section 3.1 not only takes into account cross-datacenter data replication, but also makes the system affordable to instant load surge due to load distribution when node failure occurs. Specifically, proper choice of the size $\mathcal{C}$ of mini-cluster is essential for tuning the trade-off between the tolerance of load surge and the overhead of inter-node communication. Figure 10 illustrates the impact of different configurations of $\mathcal{C}$ for single-node load increment and amount of outgoing network packets. Note that the amount of outgoing network packets of a node is proportional to the overhead of network communication with batched networking enabled. By default, in WeChat production, each PaxosStore node is deployed to run with 30% $\sim$ 60% saturation in normal expectation. As can be seen, setting $\mathcal{C} = 1$ leads to single-node workload increases by more than 40% when a related node fails. Such single-node load increment is high but expected, since workload of the failed node is evenly distributed over the other two nodes within the mini-cluster group. Increasing $\mathcal{C}$ can effectively alleviate the impact of load surge caused by node failure. Figure 10 shows that the percentage of single-node load increment decreases to 15% and 12% when $\mathcal{C}$ is configured to 2 and 3 respectively. This is because workload of the failed node is distributed over 4 (resp. 6) nodes of the remote mini-clusters under the setting of $\mathcal{C} = 2$ (resp. $\mathcal{C} = 3$). Further increasing $\mathcal{C}$ can have even smoother load surge, but the decrease becomes marginal. On the other hand, the overhead of network communication increases along with the increment of $\mathcal{C}$. For example, the amount of outgoing network packets doubles when $\mathcal{C}$ changing from 2 to 4. This is due to the fact that larger size of mini-cluster results in lower opportunities and effectiveness of performing batched network communication. Based on the above results, we generally configure $\mathcal{C} = 2$ for PaxosStore in WeChat production.

5.4 Failure Recovery

Figure 11 reports a history of failure recovery of a PaxosStore node. The real-time performance, i.e., runtime throughput and counts of failed operations, are measured under the

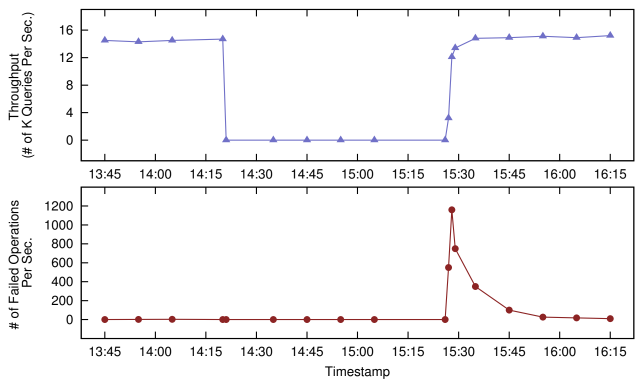


Figure 11: Monitoring failure recovery of a PaxosStore node in WeChat production.

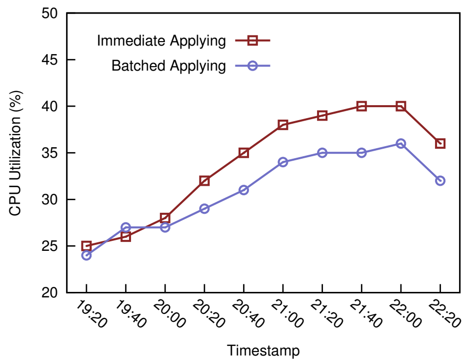


Figure 12: Effectiveness of PaxosLog-entry batched applying.

real workload in WeChat production, where the read/write ratio is 15 : 1 on average. As can be seen, the normal throughput is around 14.8 thousand queries per second when the node is stably running. Failure happens at 14:20, and the failed node is later restarted at 15:27. Note that, during the downtime of the failed node, other nodes in the related mini-cluster group function normally to process the incoming queries. After the failed node resumed, some of its data turn out to be obsolete. According to the lazy recovery technique in PaxosStore, those obsolete data are synchronized to the latest version when they are first accessed after revival of the failed node. The detection of data being obsolete as well as the subsequent replica synchronization are at the expense of failing some data operations. As can be seen from Figure 11, the number of failed operations rises rapidly in the first 3 minutes after the system is restored, and then starts to drop to normal. Accordingly, the node is restored to 95% of its normal throughput within 3 minutes. Such fast recovery in PaxosStore benefits from the fine-grained data recovery technique as presented in Section 3.2, whose effectiveness has been sufficiently validated in WeChat production.

5.5 Effectiveness of PaxosLog-Entry Batched Applying

The strategy of PaxosLog-entry batched applying, as described in Section 2.2.3, is enabled during rush hours in the real operational maintenance of PaxosStore. Figure 12 profiles the CPU utilization of one node during the rush hours[10]. In particular, when the strategy of PaxosLog-entry batched applying is turned off, all the PaxosLog entries will be applied to the physical storage immediately after they are chosen along the Paxos procedures. We name such situation as the immediate applying of PaxosLog entries. On the other hand, when PaxosLog-entry batched applying is enabled, log entries belonging to the same PaxosLog are applied to the storage in a batch manner. We configure the batch size as 10 in WeChat production. In the experiment with results shown in Figure 12, we monitor two individual nodes belonging to separated mini-cluster groups, with one node adopting the immediate applying of PaxosLog entries and the other node enabling the strategy of PaxosLog-entry batched applying during the period between 20:20 and 22:20. All the data accesses are supported by the main/delta table engine. As can be seen from the results, the strategy of PaxosLog-entry batched applying can effectively reduce CPU utilization by about 10% comparing with the immediate applying approach during the rush hours.

Like every batching-based optimization, the strategy of PaxosLog-entry batched applying inevitably introduces inconsistency with respect to the physical storage, since the chosen PaxosLog entries are applied to the physical storage with some delay. But in reality, such potential issue regarding data consistency exerts little impact on real applications. The reasons behind are twofold. First, the strategy of PaxosLog-entry batched applying is only enabled for the unspecified nodes in terms of primary read/write processing (as described in Section 2.2.3). Second, as long as the PaxosLog entries are not yet applied to the physical storage, they are retained in the PaxosLog and will not be garbage collected. Therefore, enabling the strategy of PaxosLog-entry batched applying is compatible to the mechanism of data recovery of PaxosStore as described in Section 3.2, since recovering from PaxosLog serves as the most preferential option during data recovery.

Benefit brought by PaxosLog-entry batched applying becomes marginal when the workload deviates from being write-frequent, e.g., during the normal hours. Moreover, always enabling the PaxosLog-entry batched applying may accumulate PaxosLog entries to an unexpected amount, which could eventually result in space overflow. This is because the chosen log entries of individual PaxosLogs (e.g., as per user specific) form the fixed-size batches slowly along with infrequent data updates. As a consequence, tremendous such "partially-formed" log entry batches could fail the garbage collection, since the related log entries are not applied to the physical storage with respect to the constraint enforced by the strategy of PaxosLog-entry batched applying. Hence, for a comprehensive consideration in WeChat production, we only enable PaxosLog-entry batched applying during the daily rush hours and turn it off by default.

5.6 PaxosStore for WeChat Business

PaxosStore has been deployed in WeChat production for more than two years, providing storage services for the core

[10]Empirically, the rush hours last from 20:30 to 22:00 in a day

businesses of WeChat backend including user account management, user relationship management (i.e., contacts), instant messaging, social networking, and online payment. In particular, instant messaging and social networking are the two main applications that dominate the usage of PaxosStore. Both applications rely on the key-value storage with one PaxosLog corresponding to a distinct user account. The amount of user accounts stored in PaxosStore is of billions.

5.6.1 Serving Instant Messaging

Content of WeChat instant messaging is temporarily stored in PaxosStore for message queuing. Once the content is delivered to user, it will soon be deleted by the server. Each PaxosStore server also periodically discards the contents in the message queue despite their delivery status. In other words, user's instant messaging contents is never permanently stored in PaxosStore or the WeChat backend, preserving user privacy.

The message queue of instant messaging content is stored by the LSM-tree-based storage engine. The data key of each message is represented by a sequence number which is strictly monotonically increasing. With such monotonically increasing property of data key, we perform early termination for traversing the LSM-tree to reduce the search cost of processing the open-end queries. Moreover, the monotonically increasing property entitles data recovery from partial updates of data images. Therefore, PaxosLogs for instant messaging can be more frequently garbage collected to yield storage space, while the performance of data recovery remains efficient.

5.6.2 Serving Social Networking

Another popular functionality in WeChat is the *Moments* (also known as *Friends' Circle* in its Chinese translation), the social networking platform of WeChat. User contents in the moments are logically organized as timelines per user stored by the key-value storage with indexes maintained by a set. The data key of each Moments post is partially constructed by the timestamps, and is augmented by adding certain prefixes to construct data keys for the related comments as well as reposts. Since the data of Moments are very huge but their accesses appear clear hot/cold distribution according to the timelines, we leverage the hierarchy of storage media to support fast data access, where hot data are stored in SSD and cold data are sunk to the archive storage (e.g., disks).

6. RELATED WORK

The Paxos algorithm [19] for distributed consensus [22, 10, 12] has been proven in the theory of distributed systems [18]. Apart from the theoretical study of Paxos [8, 29, 23], many research efforts have been devoted to the study of practical Paxos implementation [24, 5, 17, 34] and Paxos variants [21, 20, 25, 27, 28], leading to diverse options for the development of Paxos-based high-availability systems such as [4], [16], [2], [11], [6], [1] and [14].

The essence of improved Paxos implementeation in PaxosStore is inspired by MegaStore [2]. MegaStore by design targets supporting low-latency reads through the coordination of reading data value from the closest replica. However, such coordination inversely complicates the read/write processing, since the capability of performing low-latency reads highly depends on the availability of the coordinator. Unlike MegaStore, read/write processing in PaxosStore is based on the remote procedure call (RPC) without centralized coordination. Each cluster of datacenters running PaxosStore is deployed at the urban scale, and the network latency between datacenters is typically in the range of $1 \sim 3$ ms. This facilitates PaxosStore to support decentralized strong-consistency data access through RPC without incurring significant performance penalty.

As a Paxos-based storage system, PaxosStore is similar to the Raft [27] based systems such as etcd[11]. PaxosStore mainly differentiates itself by adopting a leaseless storage protocol design. For the lease-based consensus protocol such as Raft, the elected master nodes with time-based leases are presumed to be active during their granted lease periods, in spite of their potential runtime failure. Once master failure occurs, it will inevitably incur a period of master switching time, during which the system service will be in the unavailable state. In contrast, the leaseless Paxos implementation of PaxosStore benefits fast failover without incurring system service downtime. PaxosStore by deployment anticipates the major availability of data replicas in real-world applications. Hence, for high system availability along with fault-tolerance, PaxosStore implements the leaseless Paxos protocol and further extends it to construct the storage protocol stack to support strong-consistency read/write with efficiency.

7. CONCLUSION

In this paper, we described PaxosStore, a high-availability storage system that can withstand tens of millions of consistent read/write operations per second. The storage protocol in PaxosStore is based on the Paxos algorithm for distributed consensus and further endowed with practical optimizations including PaxosLog-as-value and concise PaxosLog structure for key-value storage. The fault-tolerant scheme based on fine-grained data checkpointing enables PaxosStore to support fast data recovery upon failure without incurring system downtime. PaxosStore has been implemented and deployed in WeChat production, providing storage support for WeChat integrated services such as instant messaging and social networking.

Lessons Learned. In the development of PaxosStore, we have summarized several design principles and lessons below.

- Instead of supporting storage diversity through a compromised single storage engine, it is advised to design the storage layer supporting multiple storage engines constructed for different storage models. This benefits ease of targeted performance tuning with respect to the dynamics of operational maintenance.
- Apart from faults and failure, system overload is also a critical factor that affects system availability. Especially, the potential avalanche effect caused by overload must be paid enough attention to when designing the system fault-tolerant scheme. A concrete example is the use of mini-cluster group in PaxosStore.
- The design of PaxosStore makes heavy use of the event-driven mechanism based on message passing, which could involve a large amount of asynchronous state machine transitions in terms of the logical implementation. In

[11]`https://coreos.com/etcd/`

the engineering practice of building PaxosStore, we developed a framework based on coroutine and socket hook to facilitate programming asynchronous procedures in a pseudo-synchronous style. This helps eliminate the error-prone function callbacks and simplify the implementation of asynchronous logics.

Acknowledgments

We would like to thank the anonymous reviewers and shepherd of the paper for their insightful feedback that helped improve the paper.

8. REFERENCES

[1] R. Ananthanarayanan, V. Basker, S. Das, A. Gupta, H. Jiang, T. Qiu, et al. Photon: Fault-tolerant and scalable joining of continuous data streams. In *Proc. of SIGMOD*, 2013.

[2] J. Baker, C. Bond, J. C. Corbett, J. Furman, A. Khorlin, J. Larson, et al. Megastore: Providing scalable, highly available storage for interactive services. In *Proc. of CIDR*, 2011.

[3] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley Longman Publishing Co., Inc., 1987.

[4] M. Burrows. The chubby lock service for loosely-coupled distributed systems. In *Proc. of OSDI*, 2006.

[5] T. D. Chandra, R. Griesemer, and J. Redstone. Paxos made live: An engineering perspective. In *Proc. of PODC*, 2007.

[6] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, et al. Spanner: Google's globally-distributed database. In *Proc. of OSDI*, 2012.

[7] J. Cowling, D. Myers, B. Liskov, R. Rodrigues, and L. Shrira. Hq replication: A hybrid quorum protocol for byzantine fault tolerance. In *Proc. of OSDI*, 2006.

[8] R. De Prisco, B. Lampson, and N. Lynch. Revisiting the paxos algorithm. In *Proc. of WDAG*, 1997.

[9] F. Färber, N. May, W. Lehner, P. Große, I. Müller, H. Rauhe, et al. The sap hana database–an architecture overview. *IEEE Data Engineering Bulletin*, 35(1):28–33, 2012.

[10] M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty process. *JACM*, 32(2):374–382, 1985.

[11] L. Glendenning, I. Beschastnikh, A. Krishnamurthy, and T. Anderson. Scalable consistency in scatter. In *Proc. of SOSP*, 2011.

[12] J. Gray and L. Lamport. Consensus on transaction commit. *TODS*, 31(1):133–160, 2006.

[13] A. Gupta and J. Shute. High-availability at massive scale: Building google's data infrastructure for ads. In *Proc. of BIRTE*, 2015.

[14] A. Gupta, F. Yang, J. Govig, A. Kirsch, K. Chan, K. Lai, et al. Mesa: Geo-replicated, near real-time, scalable data warehousing. *PVLDB*, 7(12):1259–1270, 2014.

[15] S. Héman, M. Zukowski, N. J. Nes, L. Sidirourgos, and P. Boncz. Positional update handling in column stores. In *Proc. of SIGMOD*, 2010.

[16] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed. Zookeeper: Wait-free coordination for internet-scale systems. In *Proc. of USENIX ATC*, 2010.

[17] J. Kirsch and Y. Amir. Paxos for system builders: An overview. In *Proc. of LADIS*, 2008.

[18] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *CACM*, 21(7):558–565, 1978.

[19] L. Lamport. The part-time parliament. *TOCS*, 16(2):133–169, 1998.

[20] L. Lamport. Fast paxos. *Distributed Computing*, 19(2):79–103, 2006.

[21] L. Lamport and M. Massa. Cheap paxos. In *Proc. of DSN*, 2004.

[22] L. Lamport, R. Shostak, and M. Pease. The byzantine generals problem. *TOPLAS*, 4(3):382–401, 1982.

[23] B. Lampson. The abcd's of paxos. In *Proc. of PODC*, 2001.

[24] B. W. Lampson. How to build a highly available system using consensus. In *Proc. of WDAG*, 1996.

[25] I. Moraru, D. G. Andersen, and M. Kaminsky. Paxos quorum leases: Fast reads without sacrificing writes. In *Proc. of SoCC*, 2014.

[26] P. O'Neil, E. Cheng, D. Gawlick, and E. O'Neil. The log-structured merge-tree (lsm-tree). *Acta Informatica*, 33(4):351–385, 1996.

[27] D. Ongaro and J. Ousterhout. In search of an understandable consensus algorithm. In *Proc. of USENIX ATC*, 2014.

[28] D. R. K. Ports, J. Li, V. Liu, N. K. Sharma, and A. Krishnamurthy. Designing distributed systems using approximate synchrony in data center networks. In *Proc. of NSDI*, 2015.

[29] R. D. Prisco, B. Lampson, and N. Lynch. Revisiting the paxos algorithm. *TCS*, 243(1-2):35–91, 2000.

[30] R. Sears and R. Ramakrishnan. blsm: A general purpose log structured merge tree. In *Proc. of SIGMOD*, 2012.

[31] J. Sheehy and D. Smith. Bitcask: a log-structured hash table for fast key/value data. White paper, Basho Technologies, 2010.

[32] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proc. of SIGCOMM*, 2001.

[33] M. Stonebraker, D. J. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, et al. C-store: A column-oriented dbms. In *Proc. of VLDB*, 2005.

[34] R. Van Renesse and D. Altinbuken. Paxos made moderately complex. *CSUR*, 47(3):42:1–42:36, 2015.