JCSE Journal of Computing Science and Engineering

Regular Paper

Journal of Computing Science and Engineering,
Vol. 11, No. 3, September 2017, pp. 92-99

A Data-Consistency Scheme for the Distributed-Cache Storage of the Memcached System

Jianwei Liao
College of Computer and Information Science, Southwest University of China, Beibei, Chongqing, China
liaojianwei@il.is.s.u-tokyo.ac.jp

Xiaoning Peng*
College of Computer Science and Engineering, Huaihua University, Huaihua, Hunan, China
hhpxn@163.com

Abstract

Memcached, commonly used to speed up the data access in big-data and Internet-web applications, is a system software of the distributed-cache mechanism. But it is subject to the severe challenge of the loss of recently uncommitted updates in the case where the Memcached servers crash due to some reason. Although the replica scheme and the disk-log-based replay mechanism have been proposed to overcome this problem, they generate either the overhead of the replica synchronization or the persistent-storage overhead that is caused by flushing related logs. This paper proposes a scheme of backing up the write requests (i.e., `set` and `add`) on the Memcached client side, to reduce the overhead resulting from the making of disk-log records or performing the replica consistency. If the Memcached server fails, a timestamp-based recovery mechanism is then introduced to replay the write requests (buffered by relevant clients), for regaining the lost-data updates on the rebooted Memcached server, thereby meeting the data-consistency requirement. More importantly, compared with the mechanism of logging the write requests to the persistent storage of the master server and the server-replication scheme, the newly proposed approach of backing up the logs on the client side can greatly decrease the time overhead by up to 116.8% when processing the write workloads.

Category: Cloud Computing / High Performance Computing

Keywords: Memcached; Data consistency; Buffering logs on client; Overhead; Timestamp-based recovery

I. INTRODUCTION

Memcached is a general-purpose distributed-memory caching system [1]. It adopts a typical distributed client-server architecture, and commonly used to save the requisite time for data retrieval, as it caches the frequently accessed data and objects in the memory of the Memcached servers [2, 3]. Specifically, the system consists of the Memcached servers (Memcached instances) that are for data-saving, and client libraries that offer application programming interfaces (APIs) to applications over a network or local file socket connection for the server interactions [4]. In fact, the great popularity of Memcached (or with certain application-specific modifications) means that it has been employed by many Web service-delivery companies including Facebook [3], Twitter [5], and YouTube [6] to decrease the latency in providing Web or database data to consumers.

Open Access http://dx.doi.org/10.5626/JCSE.2017.11.3.92 http://jcse.kiise.org

Received 29 August 2017; **Revised** 08 September 2017; **Accepted** 10 September 2017

*Corresponding Author

pISSN: 1976-4677 eISSN: 2093-8020

Memcached, however, stores all of the data of key–value pairs in the non-persistent memory by design, so that in the event of a failure, all of the stored data become unavailable [7]. That is, it fails to meet the semantic of data consistency, and this means that all read operations must return the data from the latest completed write operation [8]. Although Memcached can asynchronously flush memory data to the persistent storage of the servers, this not only causes a system-performance degradation, but the risk of losing uncommitted in-memory data is also present. To overcome this problem, the feature of *data replication* has been introduced to Memcached to achieve an enhanced data throughput, as well as a high availability, to ensure data consistency; such implementations include Couchbase [9], R-Memcached [10], and repcached [11]. These approaches, however, require handling the issue of data consistency at the expense of the synchronization latency, especially in cases where considerable data-update requests are made. In addition, many distributed file systems employ logging- or journaling-update operations, so a complete, up-to-date data snapshot can be yielded by resorting to the logs or journals. To be specific, because these logs are stored in the nonvolatile storage devices, when the former active server has unexpectedly crashed, it is possible to restore the lost uncommitted changes (in-memory) by replaying them. However, logging all of the update operations to the disk will definitely result in a time latency on the server side.

For the purpose of reducing the overhead caused by ensuring the data-consistency semantic, a scheme of backing up the write requests (i.e., `add` and `set` requests) on the side of the Memcached client for the benefit of a possible future data recovery is proposed in this paper. To this end, besides sending the requests to the server, the Memcached client backs up the update requests in the memory as well. Alternatively, the latest data are supposed to be asynchronously flushed to the disk, e.g., a database, thereby leading to the clients' removal of the related buffered logs. If the master Memcached server becomes unavailable, the rebooted server can replay the cached write requests that are forwarded by the clients based on a snapshot of the flushed data to recover the lost updates.

In brief, this paper makes the following two contributions:

- *Introduction of a scheme comprising the backing up of the write requests on the Memcached client side.* For decreasing the overhead that results from ensuring data consistency, such as the write-request logging on the disk storage of the master server and the synchronization of the replicas, buffering update requests in the memory of the Memcached client side is introduced. Specifically, the Memcached client side buffers the write requests until the relevant data changes have been flushed to the disk of the Memcached server. In fact, these logged requests are supposed to be replayed to restore the (in-memory) lost updates in the case of a failure of the master server.
- *Presentation of a timestamp-based data-recovery technique.* This approach seeks to efficiently regain the lost-data changes on the rebooted Memcached server by replaying the cached write requests that are forwarded by the clients on the basis of the latest flushed-data snapshot. Moreover, by following a time order, this mechanism can guarantee that only the recently uncommitted write requests will be replayed.

The rest of this paper is structured as follows: Section II discusses the background knowledge of Memcached, as well as the related works about the replication of Memcached and the log-based replay for the recovery of the lost data; the specifications of the proposed scheme are presented in Section III; Section IV describes the evaluation methodology and reports the relevant results; and lastly, this paper is concluded in Section V.

II. BACKGROUND AND RELATED WORKS

This section introduces the background knowledge regarding Memcached and its replication feature, as well as the typical data-consistency approaches for which the logged requests are replayed.

A. Memcached and Replication for Consistency

As discussed, Memcached is an open-source, multithreaded, distributed, key-value caching software system that is widely adopted for the purposes of delivering software-as-a-service (SaaS) applications and decreasing the service latency and network traffic to the database or computational servers [7]. Memcached deals with items comprising metadata, i.e., a key and a value. Specifically, these items are managed by using a number of unique data structures, such as a Slab Allocation for storage, an Association Array for quick access, and an LRU (Least Recently Used) list for an eviction from the cache [12].

As previously mentioned, nevertheless, the native Memcached does not provide an improved data availability, since the data-replication characteristic is not supported; this also indicates that the cached data becomes unreachable with the failure of the Memcached-server node, thereby resulting in a loss of the in-memory changes [10]. Accordingly, the aim of numerous research proposals is the incorporation of the replication functionality to the original Memcached to achieve an improved data availability.

Specifically, repcached is a middleware for Memcached that utilizes the master–slave replication solution to address the single point of failure problem in Memcached [11]. In order to simplify the data synchronization, the write operations can merely be processed in the master node of repcached, and by default, each data item can comprise only two replicas. More importantly, repcached employs the manner of *SYNC* synchronization to maintain the strong data consistency of the replicas. All of the write updates must be broadcast to any other relevant server nodes with the same data copy, and this definitely results in a longer write latency.

Lu et al. [10] proposed R-Memcached to enable data replication in Memcached. Different from repcached, R-Memcached offers three protocols for the retention of the cache consistency in a variety of application contexts. It cannot, however, eliminate the data-loss risk in the case where the master Memcached server becomes unresponsive, as the latest data are asynchronously flushed to a database, even though the data availability and reliability have been improved to some extent. Ristov et al. [12] proposed the Memcached-based cache system called as LogMemcached that supports state replication via the Remote Direct Memory Access (RDMA) [13] to offer an increased systemic availability and improved failure resilience.

In addition, Cocytus takes advantage of a hybrid replication scheme to yield both efficiency and availability for Memcached [14]. Memcachedb [15] and Couchbase [9] are Memcached-compatible distributed key-value storage systems that are designed for the persistent storage, and they can offer a high-availability of data storage with a replication functionality; but, both systems may cause a considerable synchronization overhead due to flushing the cached data to the persistent storage.

All of the previously mentioned replication implementations either employ the *SYNC*-synchronization scheme to maintain a strong data consistency that then causes a synchronization latency, or use the *ASYNC*-synchronization scheme to yield an improved systemic performance, but this is at the cost of a loss of certain data updates. In brief, none of the existing replication schemes with an acceptable overhead can ensure a strong data consistency without update losses.

B. Logging Schemes for the Restoration of Lost Data

The traditional approach of the logging of update requests means it is necessary to flush such logs to the nonvolatile storage before responding to the client who issued the request. This kind of mechanism is quite straightforward and can ensure a proper data consistency, as it can recover the lost data by simply replaying the recorded logs on the basis of the previous consistent state.

A holding of the data in the main memory allows the in-memory databases to significantly decrease the I/O cost of the transaction processing. To guarantee data consistency, however, the in-memory systems need to flush the log to the nonvolatile storage, which incurs a substantial number of I/Os [16]. Although many logging approaches have been proposed, including command logging [17] and advanced logging [16, 18], for the purpose of reducing the number of logging messages, the disk I/O operations cannot be avoided, and this definitely imposes negative effects onto the server machine.

Besides, a major part of the conventional distributed and parallel file systems such as the Gfarm [19] and Google file systems [20] employ this kind of mechanism to ensure data consistency. The disk logs can be redone to restore the lost data updates if the active server crashes unexpectedly.

Thus, for the purpose of decreasing the overhead that is caused by a retention of data consistency during the use of the conventional replication and logging schemes, a novel approach for Memcached is proposed here for the achievement of a data-consistency retention with a preferable overhead.

Instead of flushing the update logs to the nonvolatile storage, the clients of Memcached are responsible for buffering their sent update requests, and these requests cannot be removed until the relevant changes have been flushed to the disk on the Memcached servers. In the case of a server failure, these buffered logs are supposed to be forwarded to the rebooted Memcached server to recover the lost in-memory data updates through replaying them.

III. SPECIFICATIONS OF THE SYSTEM DESIGN

This section illustrates the specifications of the proposed data-consistency scheme for Memcached. First, a high-level architectural overview of Memcached is shown. Then, the details of the backing-up of the update requests on the side of the Memcached clients will be described. Finally, the algorithm of the timestamp-based recovery that is employed to recover the lost updates is presented with additional information.

A. High-Level Systemic Overview of Memcached

Memcached consists of the following two components: *client component* and *server component*. That is, the Memcached server manages the cached data with *key* and *value* pairs. The Memcached-client component is normally deployed in the application-server node, and responsible for communicating with the Memcached server to obtain the cached data, and this occurs after the receipt of the data request from the application.

Fig. 1 shows the functional overview of Memcached that is deployed for an Internet web application. Specifically, there are typically four steps in the workflow of reading a data item, between sending a data request and obtaining the needed data:

1. The web application issues a data request that will be intercepted by the Memcached client. Then, the Memcached client uses consistent hashing to calculate the location of the pair that is cached at the server side (i.e., the bucket, as well as the IP and the port of the server) and corresponds to the key in the request.
2. After that, the client sends a `get` request to the related Memcached server to obtain the desired data value.

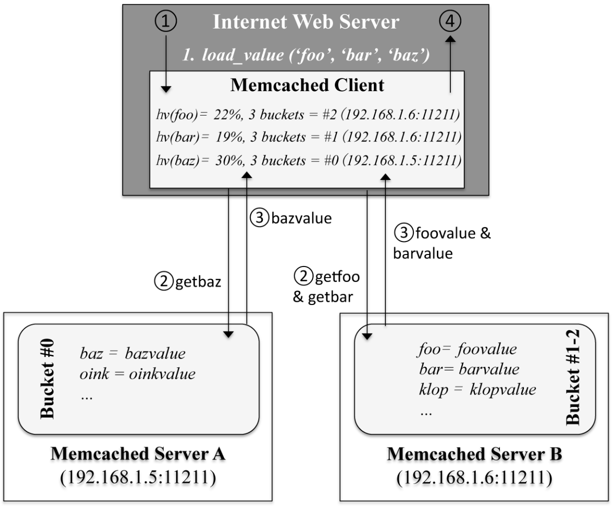


Fig. 1. High level system overview of Memcached.

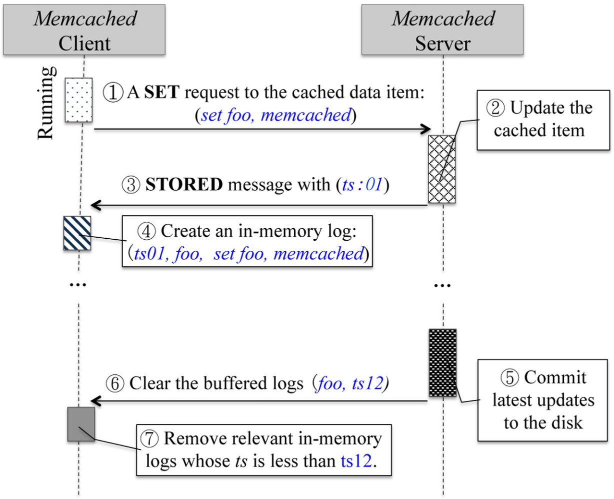

Memcached Client
Memcached Server
Running
① A SET request to the cached data item: (set foo, memcached)
② Update the cached item
③ STORED message with (ts : 01)
④ Create an in-memory log: (ts01, foo, set foo, memcached)
...
...
⑥ Clear the buffered logs (foo, ts12)
⑤ Commit latest updates to the disk
⑦ Remove relevant in-memory logs whose ts is less than ts12.

Fig. 2. Backing up write requests on the Memcached client.

3. The Memcached server is able to respond to the client with the requisite value by accessing the cached data item, instead of reading the data from the nonvolatile storage.
4. Lastly, the Memcached client forwards the received data to the web application for an eventual response from the end-users.

B. Backing up the Write Requests on the Client Side

Logging write requests (i.e., `set` and `add` operations in Memcached) to the persistent storage of the master server is a widely used method to maintain data consistency. From another perspective, if the master server crashes for any reason, the logged write requests will be replayed to recover the lost updates from a previous consistent state on the rebooted master server. But, it must record the logs corresponding to every write request to the disk, resulting in extra disk I/O operations.

To address this problem, a novel scheme to buffer the write requests on the client side of Memcached is introduced here; that is, a unique, incrementing timestamp is introduced on the side of the Memcached server to identify a specific write request to the data item. As a result, the server can respond to the clients with a normal acknowledgment message, as well as a particular timestamp that is composed of 8 bytes by default. Therefore, the client can generate the corresponding log of the write request that can be recognized by its timestamp and the key. In contrast to a flushing of the update logs to the disk, the buffering of these logs on the client side can greatly shorten the write latency, and Section IV will present more results on this topic.

Fig. 2 illustrates an example of an interactive workflow between a Memcached server and a client during the processing of a write request (e.g., a `set` request). Different from the interactive procedure in the native Memcached, the server responds to the client with not only the data value in Step 3, but also a relevant timestamp of *ts0*. On the other side, the client creates an in-memory log to back up this `set` request after it receives a *STORED* acknowledgment message from the server, as shown in Step 4. These backed-up logs are actually supposed to be used for the possible data recovery when the server with the latest data version crashes at a future time. To minimize the number of logs with respect to every data item, the proposed mechanism only keeps the latest log. Namely, the maximum of one buffered log is related to each key-identified data item.

In the case where the master server has committed the in-memory changes to the persistent-storage system, e.g., a disk file system, the relevant buffered logs will be removed by the client. This process is demonstrated in Steps 5–7, as shown in Fig. 2.

In fact, maintaining write requests in the memory of clients intends to decrease the time overhead caused by making disk logs for the requests. Even though the Memcached clients may access the same server by following an interleaved mode, and the clients only manage the uncommitted update logs sent by themselves, the newly proposed scheme can satisfy the correctness requirement. This is because the Memcached server can sort all of the uncommitted write requests, which are forwarded by a variety of clients, by referring to their timestamps. Consequently, the server can obtain a time-ordered list of uncommitted update requests, which is the same as the conventional disk-logging scheme on the server side.

Moreover, to effectively manage the buffered logs on the side of the Memcached clients, the logs are managed as a linked list, which can be flexibly adjusted according to the different operations (e.g., insert and remove) on the log records. Fig. 3 demonstrates the data structure for storing the recorded logs on the client side. The element in the Key List array indicates a key that has been

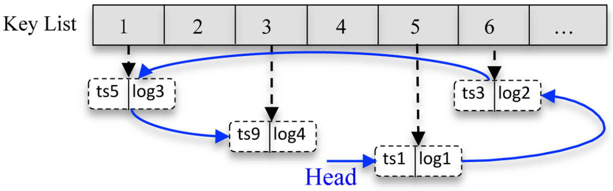


Fig. 3. Data structure for the management of buffered logs on the client side.

requested by the application on the client side, and this facilitates a quick location of the log record for which a key referral is used. More importantly, another (blue) link relationship that exists among the logs links the logs according to their timestamps.

As seen in Fig. 3, the linked list can be tuned efficiently if there is a new record, or if the buffered records are supposed to be removed. For instance, a data-structure adjustment case will be performed to remove the relevant nodes in the list when the in-memory changes that occur prior to the *ts9* timestamp have been flushed to the disk. Then, except for the log belonging to Key 3, the other relevant buffered logs are supposed to be removed from the list.

C. Timestamp-based Recovery

For the purpose of recovering the uncommitted, in-memory changes in the case where the Memcached server has crashed for some reason, a timestamp-based recovery mechanism is proposed in this study. After the rebooting of the server, the client may try to communicate with it to obtain the latest cached data copy. However, the reloaded cached copies from the disk are likely to be obsolete, as the latest in-memory versions have been lost due to the server crash. Therefore, to restore the lost changes, we have proposed an algorithm of the timestamp-based recovery, which replays the buffered logs that saved on the client side.

In fact, to support the newly proposed timestamp-based recovery approach, Memcached was slightly modified in terms of the `get`-processing operations. That is, the client is supposed to include its retained timestamp of the target data item (referred with the key) to indicate that the latest version of the data is desired. On the other side, the server also keeps an up-to-date timestamp that corresponds to a cached item; then, it compares the timestamp in the get request with its retained timestamp, and the cached data will be responded to in the case where the latter one is not less than the former one. Otherwise, the server indicates the lost changes for the cached data, triggering the recovery process.

Fig. 4 shows the Memcached clients forwarding their buffered write requests to the rebooted server to start the timestamp-based recovery for the purpose of restoring the lost updates for the cached data on the server side. Specifically, in the process of the timestamp-based recovery, the Memcached clients are supposed to send their cached logs to the server upon their receipt of the replay-collaboration request that is sent by the server. Next, the server organizes the collected logs and sorts them in the order of the timestamps. The server will subsequently redo these update requests to regain the lost updates. Eventually, the goals of the recovery of the uncommitted data and the retention of data consistency can be satisfied.

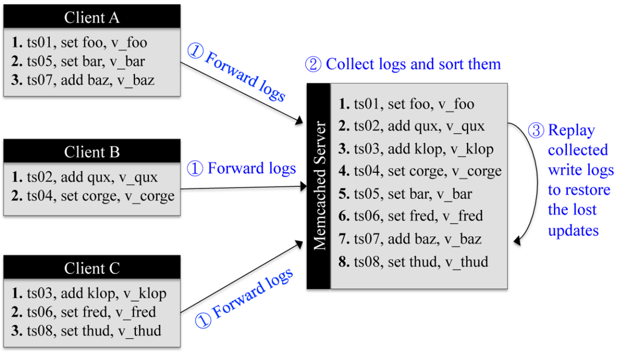

Client A
1. ts01, set foo, v_foo
2. ts05, set bar, v_bar
3. ts07, add baz, v_baz
Client B
1. ts02, add qux, v_qux
2. ts04, set corge, v_corge
Client C
1. ts03, add klop, v_klop
2. ts06, set fred, v_fred
3. ts08, set thud, v_thud
① Forward logs
① Forward logs
① Forward logs
② Collect logs and sort them
Memcached Server
1. ts01, set foo, v_foo
2. ts02, add qux, v_qux
3. ts03, add klop, v_klop
4. ts04, set corge, v_corge
5. ts05, set bar, v_bar
6. ts06, set fred, v_fred
7. ts07, add baz, v_baz
8. ts08, set thud, v_thud
③ Replay collected write logs to restore the lost updates

Fig. 4. Timestamp-based recovery in Memcached after the server sends a replay collaboration request to all relevant clients.

IV. EXPERIMENTS AND EVALUATION

This section first describes the experimental platform that was employed for the evaluation of the proposed mechanism and its comparison counterparts. It then reports the experiment results and provides relevant discussions. Lastly, a summary of the key points of the newly proposed scheme is presented.

A. Experimental Platform

To conduct the designed experiments, a six-node cluster was employed. Each node is composed of four CPUs of the Intel E5410 2.33 G and 4 GB of RAM, and all of the nodes are connected with a 10-GbE Ethernet. Four nodes in the cluster were run as servers, and the remaining two were employed to execute the client processes (threads).

B. Experiment Results

The throughput and the average latency of the proposed mechanism were tested using Memcached with disk logging and repcached with only one additional copy for the cached data. Beyond that, the native Memcached implementation *Baseline* was included to show the side-effects that are caused by different schemes in ensuring data consistency. Furthermore, evaluations of the recovery overhead when the proposed timestamp-based recovery and the conventional approach of the disk-log-based recovery are employed were conducted.

In the tests, each of the client threads forms 10,000 open-close connections with the server, so that the server deals with the `set` and `get` requests as different tests but at the same concurrency. In each request, the key is set as 64 bytes, and the value occupies 64 bytes as well.

1) Average Latency

A measurement of the average latency for the processing of a read/write operation was performed during the use of the selected schemes, and Fig. 5 presents the relevant experiment results.

As seen in Fig. 5(a), repcached needs the least time to complete a `get` request, since two servers hold the same data copy, so it can concurrently satisfy the client threads. The other three schemes resulted in the read latency without a noticeable difference, since the read operations do not influence the consistency of the data semantic. Besides, it is evident that the newly proposed *Client backup* approach requires slightly more time for the completion of a **get** request; this is because the client threads additionally access the log list to obtain the latest timestamp of the target data item.

A more interesting clue is shown in Fig. 5(b), as it is regarding an attractive write-latency improvement of up to 116.8% that is a result of the newly proposed mechanism, compared with the repcached and the *Disk log-based* mechanisms. The reason for this improvement is that, to ensure data consistency, the latter two approaches are supposed to either synchronize the write contents to other replicas or write the update logs to the disk. But, the newly proposed scheme can also fulfill this requirement by backing up the logs in the memory of the Memcached server, which does not cause a discernible time overhead.

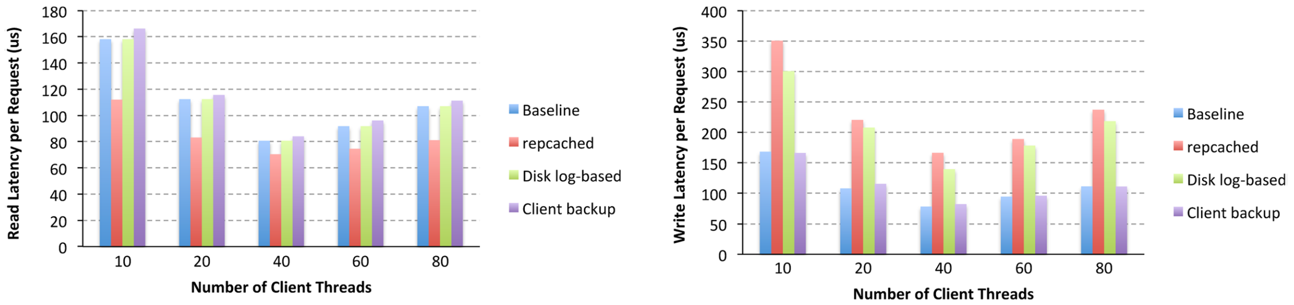
2) Transaction Throughput

The transaction throughput metric implies the number of operations that can be processed per second on the server side, and the higher transaction throughput indicates an improved systemic performance. Fig. 6 reports the statistics on this metric in terms of the evaluation experiments. In the case of the read-workload processing, and except for repcached, the other three schemes yielded nearly the same results, as shown in Fig. 6(a). Furthermore, similar to the results shown in Fig. 5(b), the proposed *Client backup* scheme outperformed the repcached and *Disk log-based* mechanisms in the processing of the write workloads, as shown in Fig. 6(b).

An emphasis was placed on the best performance of the *Memcached* baseline when a large amount of write requests are being handled, but the data consistency is not ensured here. On the other side, another three schemes were able to maintain data consistency in the case where the server has crashed due to some reason.

3) Logging and Recovery Overheads

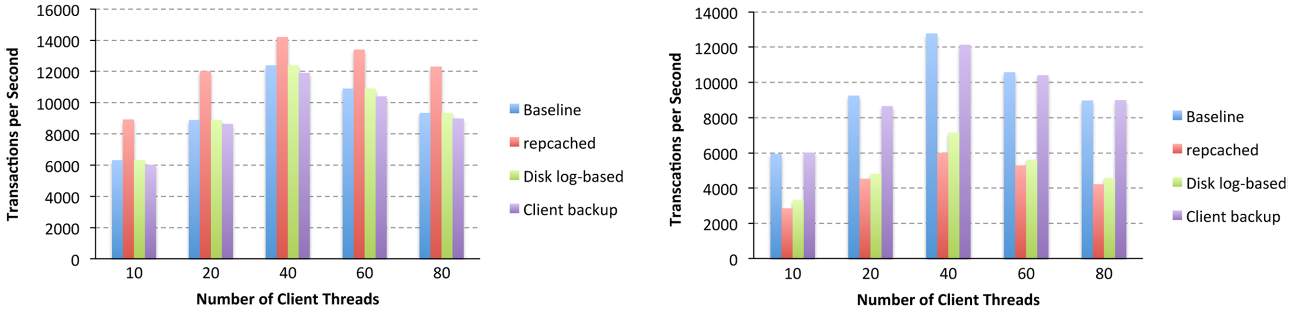
Because the replication scheme does not require performing data recovery, this section only reports the recovery overhead regarding the utilization of the timestamp-based and disk-log-based schemes. In fact, the number of requests



(a)

(b)

Fig. 5. Time latencies for a **get** operation (a) and a **set** operation (b).



(a)

(b)

Fig. 6. Transactions per second for the **get** operation (a) and **set** operation (b).

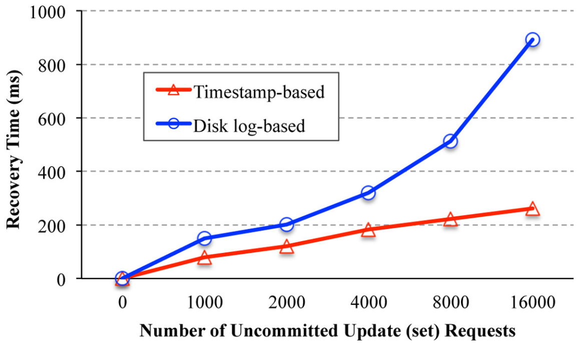


Fig. 7. Time overheads for the restoration of the lost updates.

that need to be redone is a critical factor that affects the recovery process, regardless of which scheme is used.

Fig. 7 presents the time-consumption total of the data recovery. As can be seen, more time is required as the number of update requests is increased, as more time is required for the collection and replaying of the requests.

More importantly, the newly proposed scheme of the timestamp-based recovery outperformed the *Disk log-based* scheme. For instance, the proposed scheme can reduce the recovery time by more than 70.5% in the case of 16,000 uncommitted requests, compared with the *Disk log-based* scheme. This is because the timestamp-based scheme requires less time for the loading of the uncommitted requests, although both schemes consume the same time for the replaying of them in the restoration of the lost changes.

As has been discussed, the maintenance of the write requests on the client side definitely results in client time and memory overheads. Thus, measurements of the time and space overheads of this set of experiments were performed, the results of which are specifically presented in Table 1. By referring to the data in the table, it is argued here that the proposed mechanism does not cause excessive time and space overheads for backing up the update logs. Especially, the overhead did not increase dramatically when the number of recorded logs was increased. For instance, it consumed a memory of only 20.2 MB for the buffering of 16,000 update requests on the client side, as well as 502 ms for the generation of these in-memory records. This shows the sound scalability of the approach of the buffering of the write requests on the client side.

C. Summary

With respect to the comparison of the existing schemes for the achievement of the target of data consistency and the proposed mechanism, the following two key observations are emphasized in this paper. First, the buffering of the write requests on the client side can definitely reduce the time overhead during the processing of the write workloads, in contrast to the scheme where the logs are flushed to the disk. Second, the proposed approach of the timestamp-based recovery is able to restore the lost-data updates with an acceptable overhead.

Table 1. Time and space overheads of the logging in the clients

# of logs	Logging time (ms)	Memory (MB)
1,000	82	1.8
2,000	102	3.1
4,000	166	5.8
8,000	287	10.4
16,000	502	20.2

In brief, it is concluded that the proposed mechanism can significantly reduce the requisite time by guaranteeing data consistency in the distributed-cache mechanism of Memcached. Besides, it is believed that this idea can be naturally merged into other distributed-cache systems such as Redis [21].

V. CONCLUDING REMARKS

This paper contains the proposal of a novel mechanism that contains the scheme of a write-request buffering on the client side and the approach of a timestamp-based recovery for the purpose of maintaining data consistency in the Memcached system. That is, the component of the Memcached client is supposed to back up its write requests, including the `set` and `add` operations, which will be replayed for a possible data recovery. In the case where the Memcached server crashes, the rebooted server collects the buffer logs that have been saved on the client side, and it replays them according to the timestamp order to recover the uncommitted lost-data updates. Note that this newly proposed Memcached-applied mechanism is also limited by a data-loss risk in the case where the relevant Memcached client and server crash at the same time.

Compared with the existing approaches for ensuring data consistency, such as server replication and the conventional disk-log-based scheme, the newly proposed mechanism can significantly reduce the time latency during runtime by up to more than 50% when processing the update workloads. In conclusion, the proposed approach of this paper is an attractive option for the application contexts in which Memcached is leveraged to speed up the data access where data consistency is the issue.

ACKNOWLEDGMENTS

This work was partially supported by the National Natural Science Foundation of China (No. 61303038), Fundamental Research Funds for the Central Universities (No. XDJK2017B044), and Opening Project of State Key Laboratory for Novel Software Technology (No. KFKT2016B05).

REFERENCES

1. Memcached, http://memcached.org/.
2. B. Fitzpatrick, "Distributed caching with memcached," *Linux Journal*, vol. 2004, no. 124, pp. 72-76, 2004.
3. R. Nishtala, H. Fugal, and S. Grimm, M. Kwiatkowski, H. Lee, H. C. Li, et al., "Scaling memcache at Facebook," in *Proceedings of 10th USENIX Conference on Networked Systems Design and Implementation*, Lombard, IL, 2013, pp. 385-398.
4. P. Talaga and S. Chapin, "Reducing latency and network load using location-aware memcache architectures," in *Web Information Systems and Technology.* Cham: Springer International Publishing, 2012, pp. 53-69.
5. Twitter Engineering, "Memcached SPOF Mystery," 2010, https://blog.twitter.com/engineering/en_us/a/2010/memcached-spof-mystery.html.
6. Seattle Conference on Scalability: YouTube Scalability, https://www.youtube.com/watch?v=ZW5_eEKEC28.
7. A. Wiggins and J. Langston, "Enhancing the scalability of memcached," Intel Corporation, *Technique Report*, 2012.
8. H. Wada, A. Fekete, L. Zhao, K. Lee, and A. Liu, "Data consistency properties and the trade-offs in commercial cloud storage: the consumers' perspective," in *Proceedings of 5th Biennial Conference on Innovative Data Systems Research (CIDR)*, Asilomar, CA, 2011, pp. 134-143.
9. Couchbase: NoSQL database, https://www.couchbase.com/.
10. Y. Lu, H. Sun, X. Wang, and X. Liu, "R-memcached: a consistent cache replication scheme with memcached," in *Proceedings of the Middleware '14 Posters & Demos Session*, Bordeaux, France, 2014, pp. 29-30.
11. repcached, http://repcached.lab.klab.org.
12. S. Ristov, Y. Weinsberg, D. Dolev, and T. Anker, "LogMemcached: an RDMA based continuous cache replication," in *Proceedings of ACM Workshop on Kernel-Bypass Networks (KBNets'17)*, Los Angeles, CA, 2017, pp. 1-6.
13. P. Stuedi, A. Trivedi, and B. Metzler, "Wimpy nodes with 10GbE: leveraging one-sided operations in soft-RDMA to boost memcached," in *Proceedings of the 2012 USENIX Annual Technical Conference (USENIX ATC'12)*, Boston, MA, 2012, pp. 347-353.
14. H. Zhang, M. Dong, and H. Chen, "Efficient and available in-memory KV-store with hybrid erasure coding and replication," in *Proceedings of the 14th USENIX Conference on File and Storage Technologies (FAST'16)*, Santa Clara, CA, 2016, pp. 167-180.
15. S. Chu, "Memcachedb: the complete guide," 2008, http://memcachedb.org/memcachedb-guide-1.0.pdf.
16. S. Yao, D. Agrawal, and G. Chen, B. C. Ooi, and S. Wu, "Adaptive logging: optimizing logging and recovery costs in distributed in-memory databases," in *Proceedings of the 2016 International Conference on Management of Data (SIGMOD)*, San Francisco, CA, 2016, pp. 1119-1134.
17. C. Mohan, D. J. Haderle, B. G. Lindsay, H. Pirahesh, and P. M. Schwarz. "ARIES: a transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging," *ACM Transactions on Database Systems (TODS)*, vol. 17, no. 1, pp. 94-162, 1992.
18. W. Zheng, S. Tu, E. Kohler, and B. Liskov, "Fast databases with fast durability and recovery through multicore parallelism," in *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI'14)*, Broomfield, CO, 2014, pp. 465-477.
19. Gfarm File System, https://sourceforge.net/projects/gfarm/.
20. S. Ghemawat, H. Gobioff, and S. T. Leung, "The Google file system," in *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP)*, Bolton Landing, NY, 2003, pp. 29-43.
21. Redis: in-memory data structure store, https://redis.io/.

Jianwei Liao

Jianwei Liao received Ph.D. degree in computer science from the University of Tokyo, Japan in 2012. He joined the college of computer and information science, Southwest University of China in March 2012. Dr. Liao also works for Huaihua University, Huaihua, Hunan, China. His research interests are in system software, and high performance storage systems for distributed computing environments.

Xiaoning Peng

Xiaoning Peng researched as a visiting scholar in school of computer science of Birmingham University, Birmingham 2010-2011. Now Peng is a full professor at the school of computer science and engineering Huaihua University. His research interests include database systems and system software.