Towards Scalable and Reliable In-Memory Storage System: A Case Study with Redis

Shanshan Chen*†, Xiaoxin Tang*, Hongwei Wang*, Han Zhao‡ and Minyi Guo*
*Department of Computer Science and Engineering, Shanghai Jiao Tong University, China
†School of Overseas Education, Nanjing University of Posts and Telecommunications, China
‡School of Computer Science & Technology, Huazhong University of Science and Technology, China
Email: *{moistcss, tang.xiaoxin, 77877156, myguo}@sjtu.edu.cn, ‡zhaohan.miven@gmail.com

***Abstract*—In recent years, in-memory key-value storage systems have become more and more popular in solving real-time and interactive tasks. Compared with disks, memories have much higher throughput and lower latency which enables them to process data requests with much higher performance. However, since memories have much smaller capacity than disks, how to expand the capacity of in-memory storage system while maintain its high performance become a crucial problem. At the same time, since data in memories are non-persistent, the data may be lost when the system is down.**

In this paper, we make a case study with Redis, which is one popular in-memory key-value storage system. We find that although the latest release of Redis support clustering so that data can be stored in distributed nodes to support a larger storage capacity, its performance is limited by its decentralized design that clients usually need two connections to get their request served. To make the system more scalable, we propose a *Client-side Key-to-Node Caching* method that can help direct request to the right service node. Experimental results show that by applying this technique, it can significantly improve the system's performance by near 2 times.

We also find that although Redis supports data replication on slave nodes to ensure data safety, it still gets a chance of losing a part of the data due to a weak consistency between master and slave nodes that its defective order of data replication and request reply may lead to losing data without notifying the client. To make it more reliable, we propose a *Master-slave Semi Synchronization* method which utilizes TCP protocol to ensure the order of data replication and request reply so that when a client receives an "OK" message, the corresponding data must have been replicated. With a significant improvement in data reliability, its performance overhead is limited within 5%.

***Index Terms*—In-Memory, Key-Value, Storage System, Scalability, Reliability, Key-to-Node Caching, Semi Synchronization.**

I. Introduction

Due to the increasing capacity and high throughput of main memory, in-memory computing has become a new trend for today's Big Data processing. Especially in recent years, in-memory key-value storage systems have been widely deployed in many real-time and interactive tasks, e.g., context-aware peer classification[1], geographical feature analysis[2], program constraints analysis[3], [4], cloud-based evolutionary algorithms[5], social graph analysis[6], presidential election analysis[7], wireless communications[8], etc.

However, due to the relatively smaller capacity of main memory compared with disks, single-node in-memory storage systems do not have a big enough capacity for today's Big Data challenge. Although key-value storage model, or non-relational database (or NoSQL, not only SQL), has made it easy to scale the storage systems out, they still face the challenges like single-node failure, data non-consistency, etc. To overcome these limitations, the systems have to make tradeoffs in their design and implementation.

Meanwhile, since data in main memories are non-persistent, data may be lost during unexpected system crash, which makes the system unreliable. To overcome this problem, there are several techniques that can be applied. For example, the system can periodically save the data as image files, or save update logs onto disks to ensure data safety. However, disk operations may significantly hurt the performance. The system can also use data replication which duplicates data on other nodes to improve system reliability. Nevertheless, the data consistency issue between master and slave nodes could also cause performance degradation.

In this paper, we make a case study with Redis[9], which is a very popular in-memory key-value storage system[1], [2], [3], [5], [7], [8]. Although Redis can provide stable performance as a single-node service, its clustering service still faces the problems of low scalability and reliability. By adding more nodes to the clustering service, we expect the system's performance to be improved linearly. However due to a decentralized design, it may also hurt the performance because of an inefficient data indexing mechanism. As in a distributed environment, node failure could happen quite easily and the system has to take it into considerations carefully. Yet we still find that the current design get a high chance of losing data during system crash. This paper addresses the above problems with the following novel contributions:

- We propose a **Client-side Key-to-Node Caching** method which enables clients to cache the key-to-node mapping status. This can help direct data requests to the right service node and reduce the overhead caused by false connections. Experimental results show that it can improve the system's throughput by up to 2 times;
- We propose a **Master-slave Semi Synchronization** method to ensure reliable data replication with limited overhead. This method can not only guarantee that no data would be lost during system crash, but also can limit the performance overhead by no more than 5%;

2324-9013/16 $31.00 © 2016 IEEE
DOI 10.1109/TrustCom.2016.253

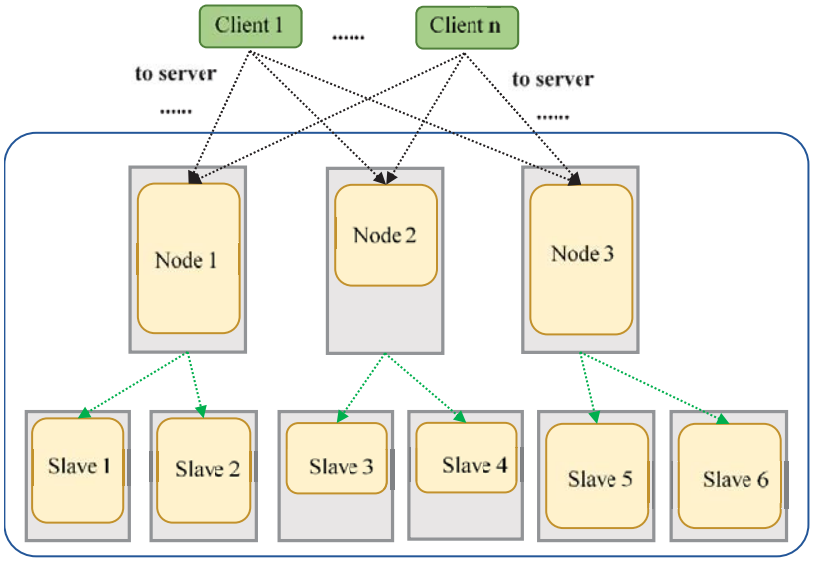

Fig. 1. Overview of a typical client-server structure for Redis distributed storage service.

This paper is organized as follows: Section II gives an introduction to the background of Redis and explains our motivation in detail; Section III describes our proposed methods and explains how we are able to improve system scalability and reliability; Section IV gives the experimental results of our methods; Section V introduces the related work of this paper; Finally, Section VI concludes this paper.

II. Background

According to the famous CAP theory[10], a system can only hold two of the three properties at the same time: consistency, availability and partition-tolerance. Although a distributed design of a storage system is the key to its scalable capacity, its performance scalability and data reliability are very challenging to handle properly. It is very important to achieve a good balance between the CAP properties. But not until recently does Redis release its stable cluster version (Redis 3.0 Beta), which shows how hard it is.

Fig. 1 gives a typical topology of a distributed Redis storage service, in which we mainly focus on how the system can be scaled out with good performance and how the system can ensure data safety. There are usually multiple machines that are providing data storage services together, which are called "Node". Each node may take certain amount of memory space on each machine and is responsible for storing parts of the data. Different clients can visit these nodes to get and set (read and write) data independently. To ensure data safety, there are also at least one slave node (two in the figure) for each master node to replicate data so that when the master is down, the slave node can replace it to restore the service. We will give a more detailed description in the following sub-sections.

A. Data Distribution Design of Redis

For a given distributed storage system, we need to first solve how data are distributed on different nodes. Since Redis is a key-value storage system, the main problem becomes how to schedule different keys to nodes. To build a stable and dynamically adjustable system, Redis uses a key-slot-node mapping strategy to maintain the data distribution status.

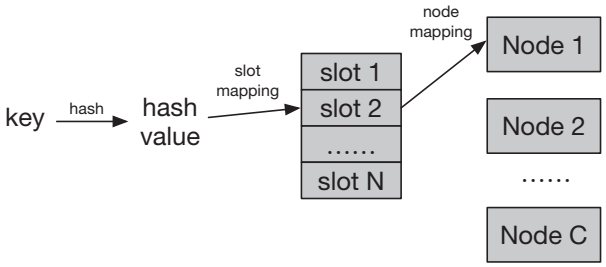

Fig. 2. A three stage mapping from key to node.

Fig. 2 shows how the keys are mapped to each node. In the hash stage, for a given key, Redis will calculate its hash value. In Redis, all hash values are evenly assigned to each slot and there are 16382 slots in total. Then in the slot mapping stage, we can find the corresponding slot for the key based on its hash value. In Redis, all slots can be dynamically assigned to different nodes. In this case, if any node is under heavy workloads, we can move parts of its slots to other nodes or even new nodes to improve system's overall performance. Thus in the third step, we can find the corresponding node of the given key based on the slot-node mapping status.

B. System Scalability Design of Redis

In addition to the data distribution status, there are other important info that needs to be maintained like number of nodes, their IP address and port numbers, current working status, etc. These info can be stored on a central node of the cluster, which could simplify the design of the distributed system. But when this node is down or its performance is slow, the whole cluster may fall in trouble.

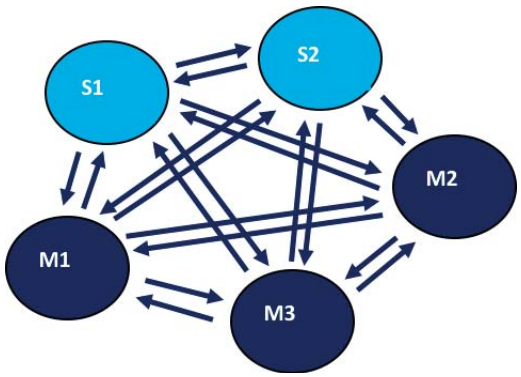

Fig. 3. A decentralized design for Redis by using Gossip protocol.

To avoid this single-node failure issue, Redis takes a fully decentralized design, which uses the Gossip protocol[11] to maintain these important info on all nodes. As shown in Fig. 3, all nodes in the system are fully connected and know the current state of the system. Every time when the system's status is changed, this new info will be propagated to every node. Nodes will also randomly send PING messages to other nodes and expect to receive PONG messages to prove that the cluster is working correctly. If any node is found not working properly, all other nodes will start a vote to use a slave node to replace the crashed master node.

With this design, the system is highly reliable and clients can connect to any of the master nodes for data storage service. When a master node receives a key-value request from a client, it will check whether this key belongs to this node by checking the key-slot-node mapping mentioned above. If yes, it will process this request and return the results back to the client. Otherwise, it will return a MOVED error which contains the right node for this request. Then after receiving this error, the client will know which node it should send the request to.

C. Data Reliability Design of Redis

Due to many possible failures in a distributed environment, Redis has to ensure that the data would not be lost during node crash. There are mainly two ways to keep data safe in Redis: data persistence and data replication. Data persistence keeps data safe by writing the data onto disks while data replication keeps data safe by replicating the data on other nodes.

Redis supports two types of data persistence: RDB and AOF. The RDB persistence performs point-in-time snapshots the data in Redis memory at specified intervals and put the compact file on disk. RDB is a good choice for data backups, disaster recovery and fast service restart since the compact file can be transferred easily. It also brings less overhead during persistence since Redis can fork one child process to do the job and the father process would never involve any disk I/O operations. However, since Redis can not frequently do snapshot, it still get the chance of losing data after the last snapshot.

The AOF persistence logs every write operation received by the server. During restoring the service, these operations will be played again. The main advantage of AOF over RDB is that it is more flexible since the system can do logging more frequently than snapshot. The disadvantage is that AOF may brings more overhead since the main process needs to do logging by itself. It also takes more space for logs and cost longer time to restore the service.

Compared with data persistence on disks, data replication gets the following advantages: first, since data is replicated on another node, the crashed service can be restored very quickly by simply replacing the master node with the slave node while for data persistence it takes longer time to read data from disks especially when the data size is big; second, data replication mainly involves network I/O, which has advantages on latency and bandwidth over disks, especially for small and random data read and write. However, data replication still faces the tradeoff between data reliability and performance since it will cost longer time to ensure that all data are fully replicated. The current design of Redis takes an asynchronous replication strategy and we find that it still has the risk of losing data.

D. Motivation

Based on the above introduction, we can find two major problems for Redis. One is caused by the decentralized design that a client needs to take two connections to get its request served: one connection to ask which node the key belongs to and the other connection to send the request to the right node and get it served. If the client is lucky, it can get the request served by using the first connection only if the key belongs to the first node. However, when there are many nodes in the cluster, the chance for the first node to own that key is quite low. In order to improve the performance, we need to find a way to help the client so that it can have a better chance to get the request served in the first connection.

The second problem is how to ensure that no data would be lost without signicantly hurting the performance. AOF is a candidate solution for this problem. However, it involves too many disk I/O operations that can bring more performance overhead and may take longer time to restore the service. Thus in this paper, we mainly consider data replication as a possible solution for this problem. The current implementation of Redis is a good start point since its master-slave synchronization method can partly ensure data safety. However, due to performance consideration, it may still lose data during our test. We will try to find a way to modify the codes to achieve our goal without making too many changes to Redis.

III. Design and Implementation

In this section, we will give details on how we solve the problems mentioned above. We propose two new designs, which are called **Client-side Key-to-Node Caching** and **Master-slave Semi Synchronization**. We will show their details and how they are implemented in the following subsections.

A. Client-side Key-to-Node Caching

Since each time when client send a key to a service node, the node will check whether this key belongs to it or not. If not, it will send a MOVED error back which could guide the client to the right node. Thus, if we can cache this mapping status after receiving this error, then next time this client should be able to find the service node correctly for all keys belong to the same slot.

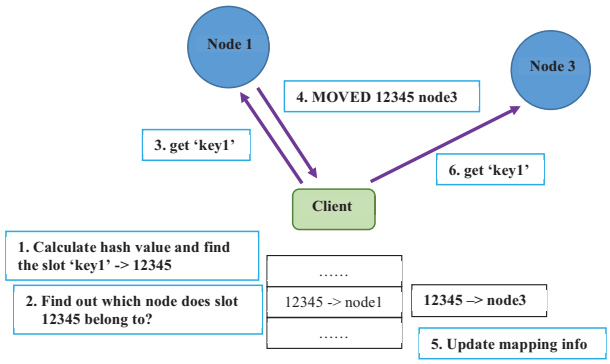

Fig. 4. A use case for client-side key-to-node caching.

Fig. 4 shows how this method works. Initially, we set up a table which stores the slot-to-node mapping status. An initial node is set as the default node for all slots. For a given "key1", the client calculates its hash value and then decide which slot this key belongs to in step 1. Here let us assume it belongs to slot "12345"; then the client searches the table to see which node this slot belongs to in step 2. Here the table shows it belongs to "node1"; Then the client sends the request to "node1" in step 3; However, "node1" finds that this slot actually belongs to "node3" and send back a MOVED error to the client in step 4; the client receives this error and update the table to remember that slot "12345" belongs to "node3" in step 5; Finally, the client sends the request to "node3" to gets it served.

Algorithm 1 Pseudocode for client-side key-to-node caching.

```
Require: Node cache[16384];
Ensure: reply
  slot = getSlotByKey(cmd);
  node = cache[slot];
  reply = send(cmd, node);
  if reply is a MOVED error then
    extract slot and node from reply;
    cache[slot] = node;
    reply = send(cmd, node);
  end if
```

Although it takes two connections for the client to get the request served for the first time, it only takes one connection for all keys that belong to this slot to get served in the next time until this slot is moved to another node. The pseudocode for the caching method is shown in Algorithm 1. Since network communication takes most of the time for each request, our caching method is expected to improve the performance by up to 2 times while it only takes very little extra space and computation to do the caching job. Thus, this method is a lightweight but useful supplement to the decentralized design of Redis.

B. Master-slave Semi Synchronization

In this paper, we mainly consider the problems of using data replication to keep data safe. Different synchronization strategies may have influences on the system performance and data reliability. Here we give an introduction to three possible strategies.

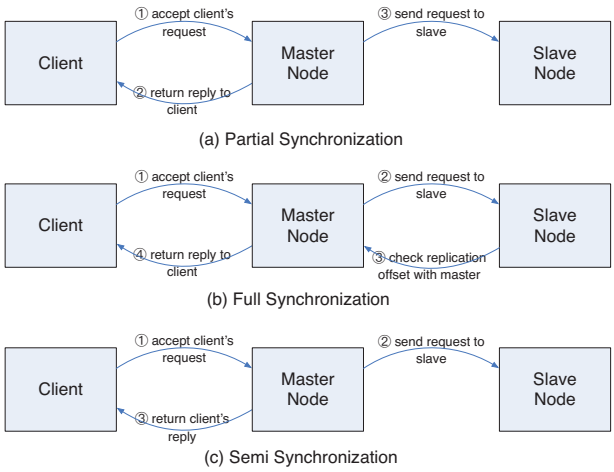

Fig. 5. Master-slave synchronization strategies.

As shown in Fig. 5(a), Redis currently uses the partial synchronization strategy. For a given data write request from the client in step ①, the master node processes the request immediately and then returns the reply back to client in step ②. After that, the master node sends the replicating request to the slave in step ③. The advantage of this strategy is that its performance is higher since data service and data replication can be processed independently. The client can get the reply as soon as possible without being influenced by data replication. The disadvantage is that it may bring the risk of losing data. For example, if the system crashes after step ② but before step ③, the client would think that its request has been processed and stored properly but the truth is the request data has not been replicated yet! In this case, this request data is lost.

To avoid this problem, a full synchronization strategy shown in Fig. 5(b) could ensure that when the client gets the reply, the request data has already been properly replicated. Instead of sending reply back to client immediately after the request is processed, master node sends the request to slave node in step ②. Then, the slave node would process this request and check replication offset with master node in step ③. If their offsets are not the same, it means there are still unprocessed requests in slave node. The master node will wait until its offset is the same as the slaves'. Finally, the master sends the reply back to client in step ④. With this strategy, if the system crashes, the client would not receive any reply, in which case client would be able to handle this error. However, due to a full synchronization between master and slave, the system's performance would be significantly influenced.

To ensure data safety while minimize the performance influence, we propose the semi synchronization strategy which is shown in Fig. 5(c). Unlike the full synchronization strategy which needs to check the synchronization status between master and slave to ensure they are fully synchronized, semi synchronization only ensures that all latest requests have been sent to the slave nodes in step ②, which is guaranteed by reliable TCP data transferring. When this is done, the master would send the reply back to the client. The assumption behind this strategy is that as long as all latest requests have been received by slaves, we can treat them as having already been replicated properly since these requests have already been successfully processed on the master node and it is just a matter of time for the slaves to process them. This strategy combines the advantages in performance and data reliability of both partial and full synchronization strategy, which is why we call it semi synchronization.

The detailed implementation of semi synchronization strategy depends on the event processing framework of Redis which is responsible for handling the time events and file events in the system. We have reused many existing APIs and added no more than 50 lines of codes to Redis. Due to space limitation, we do not give the details on how it is implemented.

IV. Experimental Results

A. Test Environment

In our experiment, we use Redis 3.0.3 as a baseline for performance evaluation. Our proposed methods are also implemented based on this version of Redis. We use our own benchmark implemented in C which is similar to YCSB (Yahoo! Cloud Serving Benchmark)[12] to test the performance of Redis and evaluate our proposed methods.

The hardware and software configurations for each service node are listed in Table I. During the experiment, we mainly use QPS (Queries per Second) to evaluate the throughput. Here, QPS represents the average number of queries processed by the system in each second. We set the length of each query as 100 bytes and set each client to send 10,000 queries to the server node. Then, we observe how the performance is changed when the number of clients vary from 1 to 512.

TABLE I
Hardware and Software Configurations.

CPU	Intel(R) Xeon(R) E5645 CPU 2 X 6 @ 2.40 GHz
Memory	64 GB DDR3 @ 1333 MHz
Disk	3 x 1 TB SATA disk
Network	Intel(R) PRO/1000 Network Connection @ 1 Gbps
OS	CentOS 6.2

B. Client-side Key-to-Node Caching

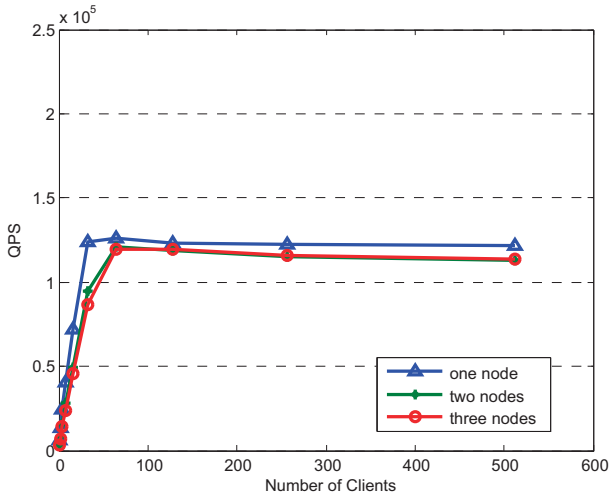

Fig. 6. System throughput of the original Redis cluster.

In this experiment, we set the read-to-write ratio as 9:1, which means there are 9 "get" queries and 1 "set" query among every 10 queries. Data replication is disabled in this test. Fig. 6 shows the system throughput of the original Redis cluster. Here we build three clusters which contains one, two and three nodes respectively. As we can find, there is no performance improvement at all when the cluster contains more nodes. Meanwhile, the performance is even slightly worse when using two or three nodes in the cluster, which shows a poor scalability in performance of the system. This result is caused by the default setting of clients since each client is set to connect to one default node at the beginning. In this case, this default node becomes the bottleneck of the system.

To avoid this problem, we apply our proposed Client-side Key-to-Node Caching method, whose results are shown in Fig. 7. As we can find, when there is one node in the cluster, there is no performance improvement compared with the original Redis cluster. This is because in this case all data belongs to the same node which makes the caching method meaningless. However, when there are more nodes in the system, the performance improvement becomes quite significant. For example, the throughput with two nodes is almost 1.7 times over that with one node after using more

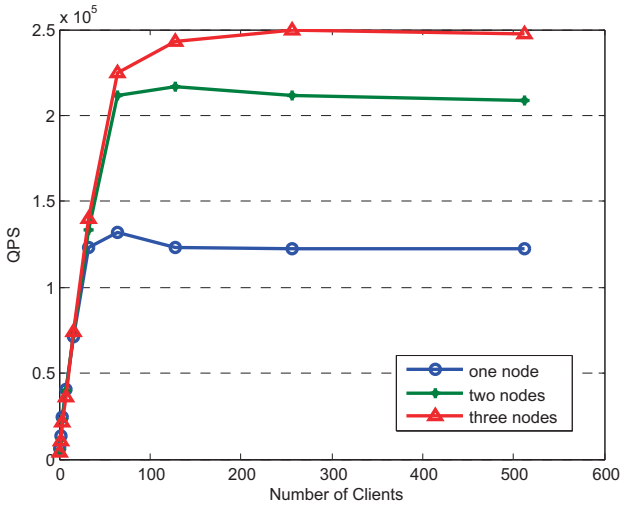

Fig. 7. System throughput of Redis cluster after applying client-side caching.

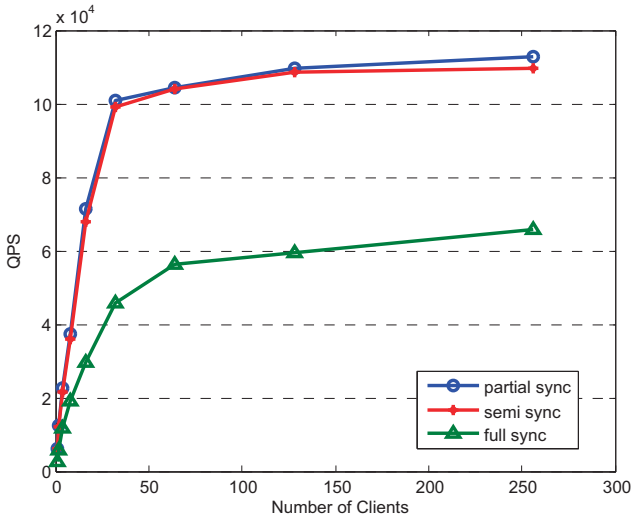

Fig. 8. System throughput of Redis with different synchronization strategies.

than 64 clients. When there are three nodes in the cluster, the throughput becomes 2 times over that with one node.

This result shows our caching method can improve system performance not only by reducing connection times but also by improving the scalability of the system since it can avoid the single-node bottleneck problem. Notice that when using more than 128 clients, the system's throughput becomes flat. This is due to the limitation that currently we put all clients on one node. Thus, the network I/O of the client node becomes the bottleneck. In our future work we will try to distribute the clients to different nodes to avoid this problem.

C. Master-slave Semi Synchronization

In order to show that our Master-slave Semi Synchronization method is able to ensure data safety with limited overhead, we also implement the full synchronization strategy to make a comparison. During the test, we set read-to-write ratio as 0:10, which means all queries are "set" operations. In this worst case scenario, we should be able to test the system's reliability to the limit. We let the system to have one master node and two slave nodes, as it is a common practice and the possibility for all three nodes to crash is very small.

To test whether the synchronization strategies are able to ensure data safety, we set up an experiment as follows: at the beginning we use 50 clients to continuously send "set" queries to the master node; then in the middle of the test, we use Linux's "kill" command to stop master node's process to simulate system crash; once the clients find that the master node is down due to the failed TCP connection, they will connect to the slave nodes to check whether the previously processed queries have been correctly replicated or not by comparing their values. We find that both full synchronization and semi synchronization have passed the test but partial synchronization has failed, which proves our previous analysis.

Fig. 8 gives the system throughput results with different synchronization strategies. Compared with the original synchronization strategy used in Redis (partial sync), our proposed semi synchronization achieves a very similar performance. However for full synchronization, its throughput is almost reduced by half. Fig. 9 further gives the performance ratio of the two synchronization strategies over partial synchronization. As we can find, semi synchronization can limit its overhead within 5%, which is much better than that of full synchronization, i.e. 50%. These results have proved that our semi synchronization can ensure data safety with a very limited performance overhead.

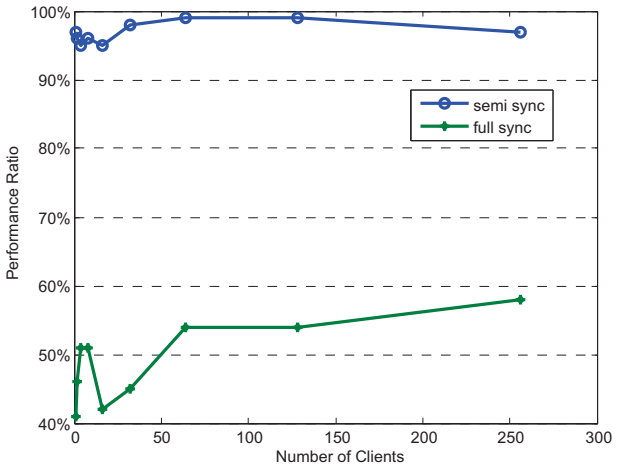

Fig. 9. Performance ratio of semi and full synchronization over partial synchronization.

V. Related Work

A. In-memory Storage Survey

There are several survey papers that readers can refer to to have an overview of in-memory storage systems. Tan *et al.*[13] have summarized the primary challenges and potential solutions from both software and hardware perspectives for in-memory databases. Zhang *et al.*[14] have provided a thorough review of a wide range of in-memory data management, processing proposals, systems and important technology in memory management. Cattell[15] have examined a number of SQL and NoSQL data stores on multiple dimensions. Carlson[9] gives a detailed introduction on how to use Redis.

B. System Analysis

System analysis is also very important for in-memory key-value storage studies. Atikoglu *et al.*[16] have collected detailed traces from Facebook's Memcached deployment and given a detailed analysis on many characteristics of these caching workload. Cooper *et al.*[12] present the Yahoo! Cloud Serving Benchmark (YCSB) framework with the goal of facilitating performance comparisons of the new generation of cloud data serving systems. Zhang *et al.*[17] analyze the performance of three in-memory systems including Memcached, Redis and Spark.

C. Optimizations for Performance

Performance design and optimization is one key to improve throughput for all these systems. A strategy[18] is implemented in Redis that epoll system call can be invoked to inform the kernel of the interest events in a manner adaptive to the current network load so that epoll system calls can be reduced and the events can be efficiently delivered. Thongprasit *et al.*[19] proposes a user-space TCP/IP stack that works on top of Intel DPDK for high performance KVS systems. Succinct[20] enables efficient queries directly on a compressed representation of input data and natively supports a wide range of queries. Zing Database[21] is presented as a high-performance persistent key-value store designed for optimizing reading and writing operations. Pilaf[22] is designed to take advantage of RDMA (Remote Direct Memory Access) to achieve high performance with low CPU overhead. Li *et al.*[23] even seeks for architecting solutions for high performance and efficient KVS platforms. HV-tree[24] uses a novel index structure that contains nodes of different sizes optimized for a level of memory hierarchy. It can dynamically adjust the node size to automatically exploit the best performance of all levels of different storage systems.

D. Data Reliability

HyperDex[25] employs a novel technique called value-dependent chaining and additional replication to provide strong consistency and fault tolerance of objects in the presence of concurrent updates. In ZHT[26] the primary replica and secondary replica are strongly consistent while other replicas are asynchronously updated after the secondary replica is complete, causing ZHT to follow a weak consistency model. Masstree[27] uses logging and check pointing with SSDs to ensure system consistency and durability.

E. Multicore and Manycore In-memory Store

Masstree[27] is a fast key-value database designed for SMP machines. It uses optimistic concurrency control which is a read-copy-update-like technique for lookup and local locking for updates. MICA[28] takes a holistic approach for multicore system that encompasses all aspects of request handling, including parallel data access, network request handling, and data structure design, which provide high throughput over a variety of mixed read and write workloads. Berezecki *et al.*[29] shows that the throughput, response time, and power consumption of a high-core-count processor operating at a low clock rate and very low power consumption can perform well when compared to a platform using faster but fewer commodity cores. Mega-KV[30] and MemcachedGPU[31] are two GPU-based high performance KVS systems. MASCOT[32] uses both CPU memory extended by SSDs and GPU memory as cache to accelerate SVM cross-validation problem and it also uses a caching strategy well-suited for its storage framework.

With the support of these high performance processors, it should be interesting to see how to maintain a good scalability and reliability for these systems.

VI. Conclusions

In this paper, we make a case study with Redis, which is one popular in-memory key-value storage system widely used in academic research and enterprise environment. Despite its success in providing single-node service, we find two major problems in its cluster deployment. The first one is caused by its decentralized design that a client usually needs two connections to get its request served. To avoid this problem, we propose a lightweight Client-side Key-to-Node Caching method that can help the client to find the right node to connect to. Our experimental results show that this method has made Redis more scalable and can improve its performance by up to 2 times. The second problem is caused by Redis's partial synchronization strategy which has the risk of losing data during system crash. To avoid this problem, we propose a Master-slave Semi Synchronization method that can ensure data safety with a limited overhead less than 5%.

VII. Acknowledgment

The authors would like to thank the anonymous reviewers for their valuable comments. This work is partially sponsored by the National Basic Research 973 Program of China (No. 2015CB352403), the National Natural Science Foundation of China (NSFC) (No. 61261160502, No. 61272099, No. 61572263, No. 61272084), the Program for Changjiang Scholars and Innovative Research Team in University (IRT1158, PCSIRT), the Scientific Innovation Act of STCSM (No. 13511504200), and the EU FP7 CLIMBER project (No. PIRSES-GA-2012-318939).

REFERENCES

[1] M. Bardac, G. Milescu, and A. M. Florea, "Deploying a high-performance context-aware peer classification engine," in *The Seventh International Conference on Networking and Services-ICNS2011, Venice/Mestre, Italy*, 2011, pp. 268–273.

[2] H. Yu, Y. Liu, C. Tian, L. Liu, M. Liu, and Y. Gao, "A cache framework for geographical feature store," in *Geoinformatics (GEOINFORMATICS), 2012 20th International Conference on*, June 2012, pp. 1–4.

[3] W. Visser, J. Geldenhuys, and M. B. Dwyer, "Green: Reducing, reusing and recycling constraints in program analysis," in *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, ser. FSE '12. New York, NY, USA: ACM, 2012, pp. 58:1–58:11. [Online]. Available: http://doi.acm.org/10.1145/2393596.2393665

[4] X. Jia, C. Ghezzi, and S. Ying, "Enhancing reuse of constraint solutions to improve symbolic execution," *CoRR*, vol. abs/1501.07174, 2015. [Online]. Available: http://arxiv.org/abs/1501.07174

[5] M. Garcia-Valdez, A. Mancilla, L. Trujillo, J.-J. Merelo, and F. Fernandez-de Vega, "Is there a free lunch for cloud-based evolutionary algorithms?" in *Evolutionary Computation (CEC), 2013 IEEE Congress on*, June 2013, pp. 1255–1262.

[6] N. Bronson, Z. Amsden, G. Cabrera, P. Chakka, P. Dimov, H. Ding, J. Ferris, A. Giardullo, S. Kulkarni, H. Li, M. Marchukov, D. Petrov, L. Puzar, Y. J. Song, and V. Venkataramani, "Tao: Facebook's distributed data store for the social graph," in *Proceedings of the 2013 USENIX Conference on Annual Technical Conference*, ser. USENIX ATC'13. Berkeley, CA, USA: USENIX Association, 2013, pp. 49–60. [Online]. Available: http://dl.acm.org/citation.cfm?id=2535461.2535468

[7] M. Song, M. C. Kim, and Y. K. Jeong, "Analyzing the political landscape of 2012 korean presidential election in twitter," *Intelligent Systems, IEEE*, vol. 29, no. 2, pp. 18–26, Mar 2014.

[8] Z. Ji, I. Ganchev, M. O'Droma, and T. Ding, "A distributed redis framework for use in the ucww," in *Cyber-Enabled Distributed Computing and Knowledge Discovery (CyberC), 2014 International Conference on*, Oct 2014, pp. 241–244.

[9] J. L. Carlson, *Redis in Action*. Manning Publications Co., 2013.

[10] S. Gilbert and N. Lynch, "Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services," *SIGACT News*, vol. 33, no. 2, pp. 51–59, Jun. 2002. [Online]. Available: http://doi.acm.org/10.1145/564585.564601

[11] A. Demers, D. Greene, C. Hauser, W. Irish, J. Larson, S. Shenker, H. Sturgis, D. Swinehart, and D. Terry, "Epidemic algorithms for replicated database maintenance," in *Proceedings of the Sixth Annual ACM Symposium on Principles of Distributed Computing*, ser. PODC '87. New York, NY, USA: ACM, 1987, pp. 1–12. [Online]. Available: http://doi.acm.org/10.1145/41840.41841

[12] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with ycsb," in *Proceedings of the 1st ACM Symposium on Cloud Computing*, ser. SoCC '10. New York, NY, USA: ACM, 2010, pp. 143–154. [Online]. Available: http://doi.acm.org/10.1145/1807128.1807152

[13] K.-L. Tan, Q. Cai, B. C. Ooi, W.-F. Wong, C. Yao, and H. Zhang, "In-memory databases: Challenges and opportunities from software and hardware perspectives," *SIGMOD Rec.*, vol. 44, no. 2, pp. 35–40, Aug. 2015. [Online]. Available: http://doi.acm.org/10.1145/2814710.2814717

[14] H. Zhang, G. Chen, B. C. Ooi, K.-L. Tan, and M. Zhang, "In-memory big data management and processing: A survey," *Knowledge and Data Engineering, IEEE Transactions on*, vol. 27, no. 7, pp. 1920–1948, July 2015.

[15] R. Cattell, "Scalable sql and nosql data stores," *SIGMOD Rec.*, vol. 39, no. 4, pp. 12–27, May 2011. [Online]. Available: http://doi.acm.org/10.1145/1978915.1978919

[16] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny, "Workload analysis of a large-scale key-value store," in *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems*, ser. SIGMETRICS '12. New York, NY, USA: ACM, 2012, pp. 53–64. [Online]. Available: http://doi.acm.org/10.1145/2254756.2254766

[17] H. Zhang, B. M. Tudor, G. Chen, and B. C. Ooi, "Efficient in-memory data management: An analysis," *Proc. VLDB Endow.*, vol. 7, no. 10, pp. 833–836, Jun. 2014. [Online]. Available: http://dx.doi.org/10.14778/2732951.2732956

[18] X. Wu, X. Long, and L. Wang, "Optimizing event polling for network-intensive applications: A case study on redis," in *Parallel and Distributed Systems (ICPADS), 2013 International Conference on*, Dec 2013, pp. 687–692.

[19] S. Thongprasit, V. Visoottiviseth, and R. Takano, "Toward fast and scalable key-value stores based on user space tcp/ip stack," in *Proceedings of the Asian Internet Engineering Conference*, ser. AINTEC '15. New York, NY, USA: ACM, 2015, pp. 40–47. [Online]. Available: http://doi.acm.org/10.1145/2837030.2837036

[20] R. Agarwal, A. Khandelwal, and I. Stoica, "Succinct: Enabling queries on compressed data," in *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*. Oakland, CA: USENIX Association, May 2015, pp. 337–350. [Online]. Available: https://www.usenix.org/conference/nsdi15/technical-sessions/presentation/agarwal

[21] T. Nguyen and M. Nguyen, "Zing database: high-performance key-value store for large-scale storage service," *Vietnam Journal of Computer Science*, vol. 2, no. 1, pp. 13–23, 2015. [Online]. Available: http://dx.doi.org/10.1007/s40595-014-0027-4

[22] C. Mitchell, Y. Geng, and J. Li, "Using one-sided rdma reads to build a fast, cpu-efficient key-value store." in *USENIX Annual Technical Conference*, 2013, pp. 103–114.

[23] S. Li, H. Lim, V. W. Lee, J. H. Ahn, A. Kalia, M. Kaminsky, D. G. Andersen, O. Seongil, S. Lee, and P. Dubey, "Architecting to achieve a billion requests per second throughput on a single key-value store server platform," in *Proceedings of the 42Nd Annual International Symposium on Computer Architecture*, ser. ISCA '15. New York, NY, USA: ACM, 2015, pp. 476–488. [Online]. Available: http://doi.acm.org/10.1145/2749469.2750416

[24] R. Zhang and M. Stradling, "The hv-tree: A memory hierarchy aware version index," *Proc. VLDB Endow.*, vol. 3, no. 1-2, pp. 397–408, Sep. 2010. [Online]. Available: http://dx.doi.org/10.14778/1920841.1920894

[25] R. Escriva, B. Wong, and E. G. Sirer, "Hyperdex: A distributed, searchable key-value store," *SIGCOMM Comput. Commun. Rev.*, vol. 42, no. 4, pp. 25–36, Aug 2012. [Online]. Available: http://doi.acm.org/10.1145/2377677.2377681

[26] T. Li, X. Zhou, K. Brandstatter, D. Zhao, K. Wang, A. Rajendran, Z. Zhang, and I. Raicu, "Zht: A light-weight reliable persistent dynamic scalable zero-hop distributed hash table," in *Parallel Distributed Processing (IPDPS), 2013 IEEE 27th International Symposium on*, May 2013, pp. 775–787.

[27] Y. Mao, E. Kohler, and R. T. Morris, "Cache craftiness for fast multicore key-value storage," in *Proceedings of the 7th ACM European Conference on Computer Systems*, ser. EuroSys '12. New York, NY, USA: ACM, 2012, pp. 183–196. [Online]. Available: http://doi.acm.org/10.1145/2168836.2168855

[28] H. Lim, D. Han, D. G. Andersen, and M. Kaminsky, "Mica: A holistic approach to fast in-memory key-value storage," in *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*. Seattle, WA: USENIX Association, Apr 2014, pp. 429–444. [Online]. Available: https://www.usenix.org/conference/nsdi14/technical-sessions/presentation/lim

[29] M. Berezecki, E. Frachtenberg, M. Paleczny, and K. Steele, "Many-core key-value store," in *Green Computing Conference and Workshops (IGCC), 2011 International*, July 2011, pp. 1–8.

[30] K. Zhang, K. Wang, Y. Yuan, L. Guo, R. Lee, and X. Zhang, "Mega-kv: A case for gpus to maximize the throughput of in-memory key-value stores," *Proc. VLDB Endow.*, vol. 8, no. 11, pp. 1226–1237, Jul. 2015. [Online]. Available: http://dx.doi.org/10.14778/2809974.2809984

[31] T. H. Hetherington, M. O'Connor, and T. M. Aamodt, "Memcachedgpu: Scaling-up scale-out key-value stores," in *Proceedings of the Sixth ACM Symposium on Cloud Computing*, ser. SoCC '15. New York, NY, USA: ACM, 2015, pp. 43–57. [Online]. Available: http://doi.acm.org/10.1145/2806777.2806836

[32] Z. Wen, R. Zhang, K. Ramamohanarao, J. Qi, and K. Taylor, "Mascot: Fast and highly scalable svm cross-validation using gpus and ssds," in *2014 IEEE International Conference on Data Mining*, Dec 2014, pp. 580–589.