Agent-Based Fault-Tolerance Mechanism for Distributed Key-Value Database

Wu Hui-jun, Lu Kai, Li Gen, Jiang Jin-fei, Wang Shuang-xi
Science and Technology on Parallel and Distributed Processing Laboratory
National University of Defense Technology
Changsha, PR China
(whj_nudt@foxmail.com)

***Abstract*—Distributed key-value database is widely used in Web 2.0 applications and cloud computing environments. It overcomes the weak performance and bad scalability of traditional relational database. But fault in distributed system will lead to errors, then the high performance is useless. So we should build a fault tolerance mechanism. On the other hand, in many application scenarios, transactional operations are inevitable. Some existing key-value databases utilize two-phase commit protocol or optimistic concurrency control in transaction processing. But the problems are sing-node failure and high overhead in protocol processing. Meanwhile, users' programming becomes more error-prone. This paper designs a fault tolerance and recovery mechanism on DStageDB, which is a distributed key-value database. We design an agent-based transaction processing mechanism. The transaction processing speed is improved and less user intervention is needed.**

Keywords—fault-tolerance, key-value database, agent, transaction

I. INTRODUCTION

With the development of Web2.0 applications and cloud computing, the limitations of traditional rational databases appear. Many NoSQL databases are designed to solve the problems of performance, scalability in rational databases. Key-Value database[1] is the most popular one. Due to CAP theory[2], existing key-value databases like Amazon Dynamo[3], Cassandra, etc. sacrifice the strong consistency to get a high performance. But most systems haven't support transaction. Some systems like Google Cloud Storage have support for single-item consistency, but not for multi-items. Many databases throw this problem to users. It is very error-prone.

Existing methods to support transaction can be classified into two main ways. They are supporting transaction processing at server end and using client libraries to support transactions. The typical system for the former type is Spanner, while much more systems belong to the later type.

Spanner's strong transaction mechanism and consistency is based on its precise GPS/atomic clock. FoundationDB [4] is a key-value database which supports transaction mechanism in client-side. It need not to lock the resources before transaction processing. The conflicts among transactions are detected by a cluster before the transactions are sanded to the database servers. When conflicts are detected, the failure will be returned to clients.

In conclusion, there are two major deficiencies in existing systems. First, single-node failures are existed in protocols like 2-phase commit. The offline of both coordinator and participants will lead to a wait. Handling this problem properly and assuring the consistency are very complicated. Secondly, sending transaction failures to users is error-prone. Although systems like foundationDB doesn't need lock support, the behavior that clients continuously try to send transaction is similar to lock too.

On the other hand, every node in the databases could be offline at every moment. It is a critical point to backup the data. The recovery of a physical node needs a relatively long time. Before new nodes can provide service, it is unwise to refuse every request from clients.

This paper designs a fault-tolerance mechanism for DStageDB, which is a distributed key-value database. We design an agent-based transaction processing. An agent is a combination of a sequence of operations, lock mechanism and conflicts processing mechanism. Once the users send an agent to the DStageDB server, it need not to wait for its completeness. All the operations will be completed in the background. Agents negotiate by messages to assure that all transactions are processed in right order.

The contributions in this paper are the following:

- We propose a fault-tolerance mechanism in distributed key-value database.
- We present the virtual node-based data recovery in DStageDB which improves the usability of data during the recovery time.
- We design an agent-based transaction processing mechanism. It reduces the influence of sing-node failure and avoids complex protocol processing. Transaction processing speed has been accelerated.

II. BACKGROUND

This section introduces the framework of DStageDB. Then we formulate the possible errors and the fault-tolerance requirement. We define the focused error type in this paper. At the end of this section, we'll talk about the problems in

978-1-4799-4284-8/14 $31.00 © 2014 IEEE
DOI 10.1109/ICDH.2014.58

fault-tolerance and transaction processing in existing key-value databases.

A. DStageDB

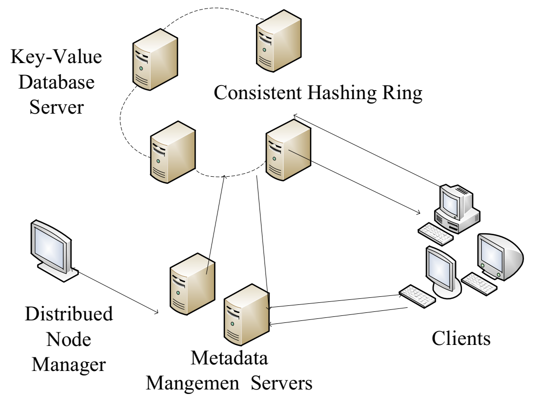

Fig. 1. DStageDB's Framework

Considering that the implementation of DStageDB[5] isn't the focus of this paper, readers interested in that can find details in our pre-work. DStageDB includes four parts. They are key-value database clusters, clients, metadata management clusters and distributed node manager. Key-value database servers are organized by the structure of consistent hashing ring. Metadata management clusters are implemented on the basis of zookeeper[6], which is an open source distributed service framework. The system manager can manage the whole system. Clients ask the information of servers from metadata management clusters and send the requests to a certain server. Each DStageDB server processes users' requests by a pipeline, so it has a relatively high throughput. On Xeon E5 platform, the single-node performance can reach about 60K operations per second. Interfaces like get, set, delete and append are provided both in synchronous and asynchronous way.

B. Errors and Falut-Tolerance in DStageDB

The first step to design a fault-tolerance mechanism is to define the faults and errors. There are four kinds of faults and errors in distributed systems. First, every node can be offline and reboot at any time. Meanwhile, the time needed for recovery is also random. Second, errors in network devices may divide the network into several isolated parts. Third, network packages might be lost. Each package could be lost during the transmission. But this fault can be removed by avoiding using UDP protocol. Fourth, the messages may be delayed due to the complicated network status and congestion.

In DStageDB, data is distributed to different server nodes. Because of this point, different nodes are responsible for different service. The second fault type won't be avoided due to the design. Package loss can be avoided by using TCP protocol. And in high-speed LAN, message loss isn't that serious. So the focus of this paper is the offline and reboot of server nodes.

Considering that each server node's data size is over several GBs, it is time-consuming to recover. If all users' requests cannot get response during the recovery time, the service delay is too long. As we introduce backup in DStageDB servers, it is critical to assure the consistency among different data copy.

C. Existing Methods for Transaction Processing

There are two main methods in distributed transaction. One is two-phase commit protocol[7] . There are coordinators and participants in the protocol. The coordinator asks each participant if it could commit. Only when all the participants' response is ok, the coordinator asks all nodes to commit. Or the transaction will be aborted. The other method is concurrency control. A frontier checks all transaction request from each node. If conflicts exists, the transaction related to the conflicts will be declined. Meanwhile, a message of failure will be sended to the correspond client. It is obvious that these methods are complex. Overhead is a critical criteria. Leaving the failures to clients makes user-side programming error-prone.

III. METHODOLOGY

This section presents the design of fault-tolerance and transaction processing mechanism in DStageDB. Firstly, we introduces how to build a global total-ordering relationship. Then we discuss the backup and recovery mechanism. Finally, we present the design of agent-based transaction processing mechanism.

A. Global Total Ordering Relationship

In order to assure the correctness of program execution, it is important to build a global total ordering relationship in distributed database. In this paper, we utilize zookeeper to do that. We set a znode at the root catalog of zookeeper. A timestamp is stored in the znode. Onces event like read, write, transaction or node offline happen, the timestamp increases. In this way, every event in the whole system has a unique timestamp. The order of the events is determined. All this is supported by ZAB (Zookeeper Atomic Broadcast) protocol [5]. ZAB protocol assures the atomic operation for znodes, so a certain timestamp won't be assigned to two events.

B. Backup and Recovery in DStageDB

DStageDB uses two-node backup for each data. We calculate the probability of liveness for different system (see TABLE 1).

TABLE I. PROBABILITY OF LIVENESS FOR DIFFERENT SYSTEM

Total Nodes	*Required Nodes*	*Probability of liveness*
1	1	95.00%
3	2	99.27%
5	3	99.88%

If we assume the failure rate is 5%, the probability of liveness for 2-node backup will be 99.27%. The recovery of a server node is around several minutes. So the MTBF will be over 10K hours.

DStageDB utilizes pipeline to process users' request. Its persistence uses SStable[8] to accelerate the performance. So the write overhead is not that high. We propose (3,3,1) backup mechanism. Each data has two backups. A read operation only needs to read the master node. Meanwhile, a write operation can only be completed when all the three nodes have been written successfully. In the second section, we mentioned that the main fault we cared in this paper was nodes' offline and reboot. We classify the fault into two types. The running time fault and recovering time fault.

The running time fault occurs when the server node is running. The node which becomes offline can be a master or a slave. If the master is offline (see Fig.2 (a)), there are also different conditions. If the last request has already been completed and no new request comes when the node is offline, the offline can be sensed by zookeeper. The consistency between master and slaves isn't damaged. Then the following read operations need to be responded by slaves while write operations will return failure.

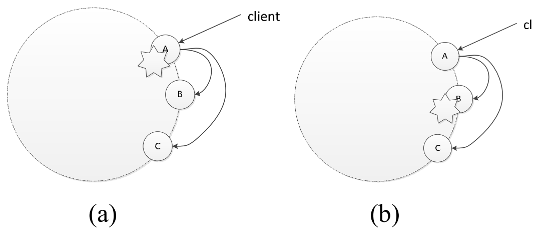

Fig. 2. Running time fault

What's more, if last read request is still flying, the user will discover the offline by zookeeper. Then the user turns to slaves to read. But if the request flying is write and it has already done on master node, the backup-writes on slaves have not finished. To solve this problem, the write request's completeness is iterative(see Fig.3). The backup write is sended to B from A and then it is sended to C from B. When C is finished, the return will be sended reversely. So if A is offline, the write request won't be finished.

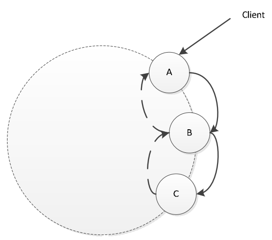

Fig. 3. Iterative completeness of write request

If one of the slaves is offline (see Fig.2 (b)), the read request can still be finished on master. But for write request, the return-link will be broken, so the write won't be finished. The system could discover the offline by the message from zookeeper and it allocates a new node to be a slave.

Another type of fault is recovering time fault. No matter master or slave is offline, the system will allocate a new node to execute the recovery. As the master and slaves are always in consistency, every node can be used to recover the data to new-allocated node. But fault may be occurred. If the new allocated node is offline, although it isn't likely to happen.(see Fig 4(a)). The only way is to allocate another one. If the node that is responsible for copying is offline (see Fig 4(b)), because of the consistency, arbitrary node can be used to copy.

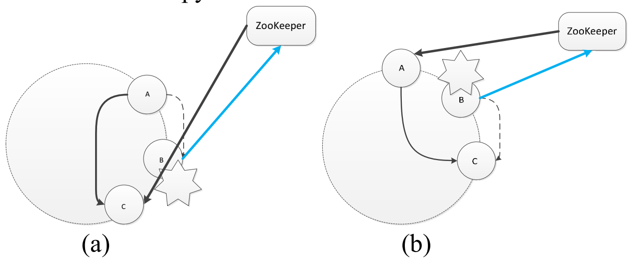

Fig. 4. Recovering time fault

We have mentioned that zookeeper maintains the status of nodes. There are five status for each node. They are preparing, prepared, service, recovering and offline. The status switching is shown in Fig.5.

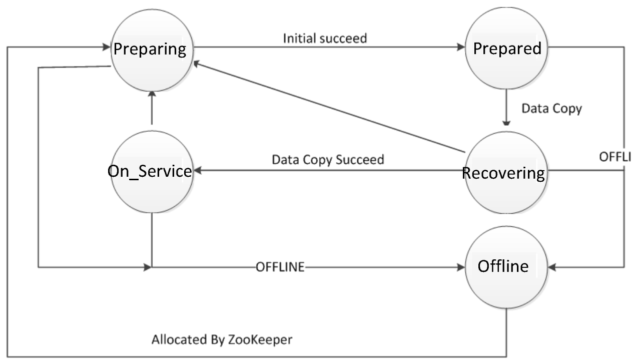

Fig.5. Node status switching model

In preparing status, the node do some initializations like create database. After that, it switches into prepared status. When the node that is responsible for copying data discovers this status, it begins to copy data to the new-allocated node. During the copy, the node status is recovering. Once the copy is finished, the node can run into service status. When a node in service status is allocated to receive copy, it switches into preparing status. No matter what status a node is in, if it is offline, it switches into offline status.

C. *Virtual Node Based Recovering*

In DStageDB, each physical node's data size is around several hundred GBs. If we execute recovering on the granularity of physical node, the recovering time will be several minutes. The most serious thing is that during this period, the database could not provide service. In order to solve this problem, we propose virtual node-based recovering mechanism. That means the mechanism we introduces in section B will be executed on the granularity of virtual node. Once a virtual node is completely recovered, it can provide service.

For instance, let's assume that the data size of a physical node is 500GB. DStageDB utilizes levelDB as the underlying database engine. If we divide the physical node into 500 virtual nodes, each node contains 1GB data. Each virtual node is a logically independent database. When the virtual node is in on_service status, it can provide services to clients.

But when we are deploying a large-scale system, it is not realistic to have so many homogeneous machines. If machines with different performance share the same portion of key space, there will be imbalance. In order to overcome this problem, we introduce benchmarks to evaluate the performance of a certain node. The score of a node decides its workload. That is to say, before a node is added into servers, it has already been evaluated. The key space of a virtual node is fixed, but high performance machine will be divided into more virtual nodes. On the other hand, the benchmark per se isn't unique, it relies on the application type. For different applications, the users' behavior pattern is different either.

D. Agent-Based Transaction Mechanism

Agent-based transaction mechanism is to achieve a no-center and light transaction. A transaction agent includes three parts.

They are the sequence of operations, lock mechanism and conflicts processing mechanism. Once a user send an agent to servers, agent itself assure the completeness of transaction. Zookeeper maintains the global total ordering. Though clock on different nodes are not synchronous, transactions are processed by timestamps provided by zookeeper. If conflicts occur, agents negotiate to guarantee the transactions are processed in the right order.

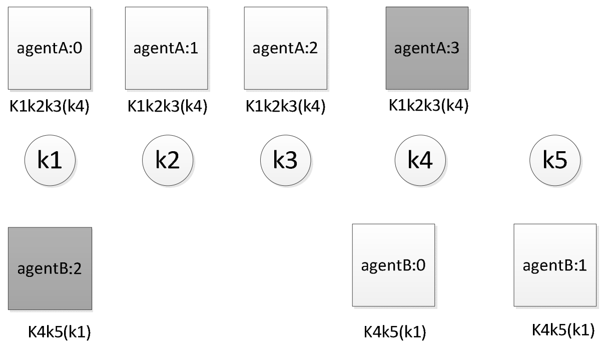

Fig.6 Agents' lock mechanism

In Fig. 5, k1, k2, k3, k4 and k5 are (key,value) pairs. Agent A and agent B are two transactions issued by two client A and client B. And agent A:0 is the main agent from client A, agent A:i(i≠0) are derived from agent A:0. Each agent is a single thread. Transaction A needs to lock k1, k2, k3 and k4. Meanwhile, transaction B needs to lock k4, k5 and k1. So agent A: 0 tries to lock k1 at first. An item is added to (key, value) pairs to show the information whether a certain pair has already been locked. If k1 is ready to be locked, then the thread agent A: 1 will be derived from POSIX [9] thread pool to detect k2. When k2 is locked, agent A: 0 could get this information from agent A: 1 by a message. Going on like this, if all the pairs are locked, the transaction can be executed. But sometimes there may be a deadlock. Let's suppose a condition like this. When agent A: 3 is detecting k4, it discovers that this pair has already been locked. At the same time, agent B: 2 is trying to lock k1. As k1 is locked by agent A: 0, agent B: 2 could not get the lock either.

To deal with deadlock, agents are able to avoid and handle the deadlocks by related algorithm. An advisable method is HRRF [10] algorithm, in which the transactions locked for a long time has a higher priority. So the fairness can be guaranteed. The most important thing is that the algorithm is implemented by the negotiation of agents.

This method is aimed at transactions that can be executed concurrently. For the transactions with data dependencies, they should be executed serially. To be specific, agents of these transactions will be put in order in transaction queues. Transactions from different queues can be concurrently processed while the transactions from a same queue should be executed serially.

IV. Evaluations

In this section, we test the performance of our design. Firstly, we test the overhead of different backup strategies. The result prove the effectiveness of the strategy we choose. Then we test the performance and usability improvement of virtual node based recovery mechanism.

A. Backup Stategies

We test (3,3,1),(3,1,2) and (3,2,1) strategies respectively. Strategy (3,3,1) emphasizes the strong consistency among master and slaves. Each read request only needs to read one node while the write request has to write all the three nodes. Strategy (3,1,2) aims at fast write. Only one node is needed to finish writing. As for read, the first two results are ordered to ruturn. Strategy (3,2,1) is a compromise. To make the experiment result more reliable, we utilize the key-value request model concluded by Berk Atikoglu [11], etc. from Facebook. Here are the results (see Fig.7).

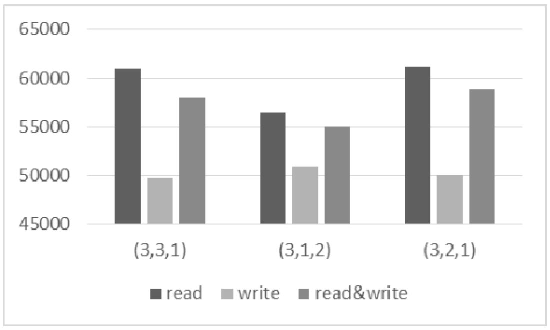

Fig.7 The performance of different backup strategies

It is obvious that strategy (3,1,2) has a bad performance. The main reason is the speed of read requests is determined by the slower one in the two results returned. Strategy (3,3,1) and strategy (3,2,1)'s performance are close. But there is a serious problem of consistency. Because the nodes we write may not be the nodes we read.

B. Virtual Node Based Recovery

We want to test the improvement of data usability and recovery performance in this subsection. Each node has a 256GB SSD. We divide the data size into 512 virtual nodes. The network here is Infiniband, the recovery time for a physical node is around 270 seconds. If we execute recovery at virtual node-grained, the time cost is about 280 seconds. A virtual node can provide service to users after itself is completely recovered.

That is to say, one virtual node can be recovered in 0.55 second. The data usability p_u can be calculated. If we suppose the recovery time of a physical node is t_r, the recovery time of a virtual node is t_{vr}, the number of virtual node is n.

$$p_u = \frac{t_{vr}}{t_r \times n}(1 + 2 + \cdots + n) = 50.3\%$$

By introducing virtual node based recovery, the data usability during the recovery has been improved from 0 to 50.3%.

C. Agent-based Transaction

We test the performance of agent-based transaction in this subsection. We build a 12-node system and two kinds of transaction. The first type is a transaction with 5 read requests and 5 write requests, the second one is 10 read requests.

We use two types of transactions as workload respectively. Not all requests are transactions. In fact, there are only small part of requests need to be finished atomically. The percentage in our test is 10%. In the all-read transaction test, the performance of database is 21.9K operations per second. While in the hybrid transaction test, the performance is 20.2K operations per second. The overhead introduced by agent-based transaction is about 8.3%.

ACKNOWLEDGMENT

We would like to thank the anonymous reviewers for the insightful comments to improve this manuscript. This material is based upon work partially supported by National High-tech R&D Program of China(863 Program) under Grants 2012AA01A301 and 2012AA010901, program for New Century Excellent Talents in University and National Science Foundation(NSF) China 61272142, 61103082, 61170261 and 61103193.

CONCLUSION

In this paper we have proposed a new approach for distributed key-value databases' fault tolerance. Compared with previous work, it has better performance.

The whole mechanism is based on agents, which are a combination of data and program binaries. The program can be executed at servers. So users need not care about the transaction after send it to the servers. On the other hand, although virtual node based recovery does not seem to be a complex technique, it improves the usability of servers a lot.

We have tested system performance. Our results show that we add fault tolerance support to key-value database which often focus on performance only.

REFERENCES

[1] M. Seeger, and S. Ultra-Large-Sites, "Key-Value stores: a practical overview," *Computer Science and Media, Stuttgart*, 2009.

[2] E. Brewer, "Pushing the CAP: Strategies for consistency and availability," *Computer,* vol. 45, no. 2, pp. 23-29, 2012.

[3] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, "Dynamo: amazon's highly available key-value store." pp. 205-220.

[4] " FounndationDB," https://foundationdb.com/key-value-store/white-papers.

[5] L. K. Wu Hui-jun, Li Gen. , " Design and Implementation of Distributed StageDB: A High Performance Key-Value Database."

[6] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed, "ZooKeeper: Wait-free Coordination for Internet-scale Systems." p. 9.

[7] R. L. Brockmeyer, R. Dievendorff, D. E. House, E. H. Jenner, M. K. LaBelle, M. G. Mall, and S. L. Silen, "Extension of two phase commit protocol to distributed participants," Google Patents, 1996.

[8] R. P. Spillane, P. J. Shetty, E. Zadok, S. Dixit, and S. Archak, "An efficient multi-tier tablet server storage architecture." p. 1.

[9] D. R. Butenhof, *Programming with POSIX threads*: Addison-Wesley Professional, 1997.

[10] L. I. Hai-cheng, "Research on HRRF Scheduling Strategy Based on TinyOS," *Computer Science* vol. 4, no. 20, 2010.

[11] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny, "Workload analysis of a large-scale key-value store." pp. 53-64.