Computing (2014) 96:455–478
DOI 10.1007/s00607-013-0355-8

A novel clustered MongoDB-based storage system for unstructured data with high availability

Wenbin Jiang · Lei Zhang · Xiaofei Liao · Hai Jin · Yaqiong Peng

Received: 31 January 2013 / Accepted: 26 September 2013 / Published online: 14 November 2013

Abstract More and more unstructured data are produced and consumed over network. How to maintain these data and improve the availability and scalability of the storage systems has become a considerable challenge. Although some NoSQL systems such as Dynamo, Cassandra, MongoDB have provided different advantages for unstructured data management, no one can provide flexible query functions like MongoDB, meanwhile guarantee the availability and scalability as Cassandra simultaneously. This paper presents a new high available distributed storage system called MyStore based on an optimized clustered MongoDB for unstructured data. Consistent hash is used to distribute data on multiple MongoDB nodes by applying virtual node method. NWR mode is applied to provide automatic backup operation and guarantee data consistency. And a gossip protocol is taken for exchanging information of failures in the system. Moreover, a user-friendly interface module and an efficient cache module are designed for improving the usability of the system. Based on above strategies, the system can realize high availability for unstructured data storage, while providing complex query functions like rational databases. Moreover, it is applied in a multi-discipline virtual experiment platform named VeePalms that has run practically. Experimental evaluation shows that the methodology is powerful enough not only to enhance the data availability, but also to improve the server's scalability.

Keywords Unstructured data · High availability · Distributed storage · NoSQL

Mathematics Subject Classification 68P20 · 68M14 · 68P05

W. Jiang · L. Zhang · X. Liao (✉)· H. Jin · Y. Peng
Services Computing Technology and System Lab,
Cluster and Grid Computing Lab, School of Computer Science and Technology,
Huazhong University of Science and Technology, Wuhan 430074, China
e-mail: xfliao@hust.edu.cn

1 Introduction

Computer and information technologies have brought us into the era of data explosion. Massive data produced as well as required by Grid computing, Ubiquitous computing, Cloud computing, etc. are demanded to be arranged efficiently and effectively. However, the complexity and the vast quantity of the data makes the management and maintenance become more and more complicated and harder to guarantee their availability. Especially, more and more unstructured data are produced and consumed over network, which are usually diversified in format, size, and content. According to the survey of the Gartner Group [1], 80 % of business is conducted on unstructured information and unstructured data doubles every three months.

How to manage such massive unstructured data and construct an efficient, user-friendly data storage platform is still a hard nut to crack. Many companies and communities have been suffering from the rushing growth [2,3]. It is urgent to improve the capability of data accessing and transaction processing to provide a uniform, stable storage platform for unstructured data with high availability.

Traditionally, there are following ways to store unstructured data using existing technologies or the combination of them.

The first one is storing unstructured data in a local file system, with maintaining an index table in memory or a relational database. This method can speed up reading/writing operations and provide high throughput. However, maintaining the index table is a tough task. It is hard to guarantee the integrity and consistency between the original data and their index information. Moreover, some additional authentication is needed to guarantee the security of a local file system.

The second one is storing unstructured data in a relational database system, always represented as BLOB (*Binary Large Object*) field. This method fully takes the advantages of modern database technologies, such as transaction processing, data arrangement and security issues. While, under the circumstance of the roll-in and roll-out of large amount of small files, the relational database has limited performance of query. The optimization space is also relatively small. Moreover, the relational database is hard to make scale-out, for complex table designs and many join operations.

Recently, more and more NoSQL (*Not Only SQL*) [4] databases are presented for unstructured data storage, such as Dynamo [5], Bigtable [6], Cassandra [7], MongoDB [8]. Different databases can provide different advantages for diversified applications. However, they also have different limitations for unstructured data storage, which will be discussed in details in the next section.

Here, the objective of our research is to design and implement an efficient, user-friendly and always-available storage system by overcoming some limitations of these existing NoSQL databases. Especially, we pay main attention to improve the availability of the storage system.

How to maintain unstructured data and improve their availability has become a tough job since we must accept that certain storage nodes are unstable and maybe become unavailable sometimes [9].

At present, there is an emerging consensus in large-scale data storage that the traditional focus on ACID (*atomicity, consistency, isolation, durability*) has become more and more inappropriate in today's digital world with explosive data increasing

[10,11]. The availability and partition tolerance have become at least as important as consistency [12]. In some applications or systems, the availability becomes much more important and urgent than consistency [13]. In cloud computing, there are strict operational requirements on 7-days, 24-h service in terms of availability, while supporting continuous data growth. Dealing with all kinds of failures in a system comprised of thousands of computer nodes has become a normal and frequent operation. So, the software system needs to be constructed in a manner which treats failures as normal happenings rather than exceptions or errors. It should recover from failing state automatically [14,15]. To meet the needs of data availability and scalability, a new storage system called MyStore is developed. Moreover, it has been deployed in a remote network education platform: VeePalms.

VeePalms is a virtual experiment education platform for multi-disciplines. It needs to manage and store a lot of various data, such as XML (*Extensible Markup Language*) experiment components, scenes, guideline videos, experiment reports. These data are stored and archived in MyStore. If the data storage system fails, VeePalms will break down. Moreover, the network latency is also an important factor for data storage. If a storage request is not responded in an acceptable time, the request will be considered to be failed.

MyStore is a distributed data storage system comprised of a batch of nodes. Some new mechanisms are used to achieve high availability and scalability. It is also easy to add a node or remove a node in the system to make it scalable.

The rest of the paper is arranged as follows. Section 2 gives the background knowledge related to this work including an overview of basic theories and the NWR architecture. Section 3 talks about some technical background issues. Section 4 describes the architecture of MyStore. Section 5 focuses on the data storage module of the system, some key technologies and issues are discussed. In Sect. 6, evaluations both in functionality and performance of the system presented are given. Finally, Sect. 7 draws some conclusions and gives some directions for future work.

2 Related work

The performance and the availability of unstructured data management have been studied widely in file systems and databases. There are existing considerable NoSQL systems using distributive policies to improve data availability, such as Dynamo, Bigtable, Cassandra, MongoDB.

Dynamo is a highly available key-value storage system introduced by Amazon EC2 [16] to provide always-on service. The design of Dynamo is to avoid failures coming from network, hardware, or software exceptions, as well as to reduce centralized control as much as possible, which brought lots of consistency-related problems [17]. The data is stored in node, which is decided by consistent hashing algorithm. Consistent hashing is a special kind of hashing. In traditional hashing tables, change of size of array causes nearly all keys to be remapped. By using consistent hashing, only K/N keys need to be remapped on average, where K is the number of keys, and N is the size of array. Consistent hashing could decrease the number of keys to be remapped,

and avoid lots of data migration operations in distributed data storage. Also consistent hashing provides incremental scalability.

In Dynamo, NWR is another important technique for solving the problem of data consistence. NWR is short for three parameters for read and write operations. N is the number of replicas in a global space. W is for synchronous writing. The number of successful writings in a storage system should be greater than W. R is for synchronous reading, which denotes the minimum of successful reading. Configuring these three parameters could get varied effects of data consistency.

Unlike Cassandra, which stores data in string, Dynamo stores data in binary format with MD5 encryption. Data access must use key values, which makes queries harder. This limitation blocks it to be applied in many applications that require some relative complex queries. In Dynamo, to make load balance, some information about the scale of the storage servers is required, which reduces its scalability to some extent.

Bigtable is a well-known infrastructure introduced by Google, which is a distributed storage system for managing structured data. It maintains a sparse, distributed, persistent multi-dimensional sorted map. However, Bigtable is owned by Google. It is hard for the third part to use it. Fortunately, there are some Bigtable-based open source projects available.

Cassandra is another unstructured data storage system aiming at availability. It is originally developed by Facebook. Cassandra is an open source project of Apache. It combines the data model of Google's Bigtable and the distributed architecture of Amazon's Dynamo, which makes it also called Dynamo2. Due to referring the data model of Bigtable, this combination makes Cassandra more flexible compared with Dynamo. Cassandra uses a gossip protocol [18] to communicate with other physical nodes. The gossip protocol is inspired by the form of gossip seen in social networks. Modern distributed systems often use gossip protocols to solve problems that might be difficult to be solved in other way. Gossip protocol makes central nodes unnecessary. The positions of all storage nodes are equal. The failure of some nodes will not cause disastrous consequences. Cassandra is a complete NoSQL database oriented to unstructured data. It cannot provide complex query and combined query, unlike relational databases.

There are also existing diversified document oriented storage systems, such as CouchDB [19] and MongoDB [20]. CouchDB is a document-oriented database that can be queried and indexed in a MapReduce fashion using JavaScript. It is written in Erlang. While MongoDB is a scalable, high-performance, open source, document-oriented database, which is written in C++ and based on Bigtable. Both of above are schema-free. This kind of architecture is suitable for sparse and document-like data storage. But CouchDB is not mature enough. So we choose MongoDB as our research base.

MongoDB can provide more complex query functions compared with Cassandra and Dynamo. It is a database system between relational database and non-relational database. On one hand, it can provide data organization by Collection that is like database and table in relational database. On the other hand, it does not restrict the data model and structure, which makes MongoDB can store almost all of unstructured data. The above features make MongoDB provide complex queries for unstructured data. However, MongoDB just uses simple master/slave mechanism for data replication, which reduces the data availability obviously. Moreover, the scalability of this system is also limited.

The aim of our research is to build a new NoSQL system to overcome the shortages of these systems mentioned above based on clustered MongoDB, which can provide perfect query functions like MongoDB as well as provide high data availability and scalability like Cassandra and Dynamo.

3 Background

3.1 Basic concepts

Traditional database provides ACID guarantees which have poor availability [21]. According to Eric Brewer's CAP [22] (*Consistency, Availability and Partition Tolerance*) theorem, it is impossible for a distributed system to provide all the three guarantees of CAP simultaneously. In a distributed system, partition tolerance is a basic requirement. Therefore, the design process of a distributed system is a balance between consistency and availability. MyStore is designed to be an always-available service. We choose Eventually Consistent [23], since there is a time window between users' each operations and it is hard for end-users to be aware of the inconsistency. We simplify the merge of concurrent *write/update* of identical data using *last write wins* policy.

3.2 REST architecture

REST (*Representational State Transfer*) is a style of software architecture for distributed hypermedia system. The largest REST architectural system is World Wide Web. MyStore is also designed as a REST architectural system. RESTful interfaces provide uniform and user-friendly access methods for system users and end-users. Meanwhile, RESTful interfaces are stateless, and there is no context stored on the server between requests.

3.3 Data model

MyStore is also a key-value storage system. The basic unit of writing is a record. The record is a BSON (*Binary JavaScript Object Notation*) document similar to MongoDB. BSON is a data interchange format for data storage and network transfer. In MyStore, the record is designed composed of five key-values according to the requirements of the system. The following is an example:

{"*_id*" : *ObjectId*("4*ee*4462739*a*8727*afc*917*ee*6"),
"*self-key*" : "*Resistor*5",
"*val*" : *BinData*(0, "*dGhpcyBpcyB*0*ZXN*0*IGRhdGEgZm*9*yIHJlYWQ* = "),
"*isData*" : "1",
"*isDel*" : "0"}

- The first key *_id* is a private key. Its value is automatic generated by UUID (*Universally Unique Identifier*) algorithm. The private key is the main evidence for identifying two different records.
- The second key is *self-key*, the value is decided by user. The *self-key* is used for indexing and searching in read operation.
- The third key *val* is data, which is the data entity for writing.
- The last two keys *isData* and *isDel* are flags. The former is a flag indicates whether the record is a copy, and the latter is a flag for data deleting. If the record is deleted, just update the flag and not physically remove the record from disk.

4 Architecture of MyStore

Briefly, MyStore, the new NoSQL system presented in this paper, contains four main modules, which are user interface, distribution module, cache module and data storage module. Its framework and data flow are shown in Fig. 1.

The user interface is designed as a REST architectural system. The HTTP (*Hypertext transfer protocol*) methods corresponding to relational database's CRUD (*Create, Read, Update, and Delete*) operations are:

- GET method: retrieve a representation of addressed data, expressed in an appropriate Internet media type.
- POST method: create or update a new entry.
- DELETE method: delete the addressed data.

The access of unstructured data in the system fully takes advantage of HTTP, it also exposes three operations: GET, POST,DELETE. GET operation locates unstructured data with the key in cache or database (if it gets a cache miss, it will switch to database

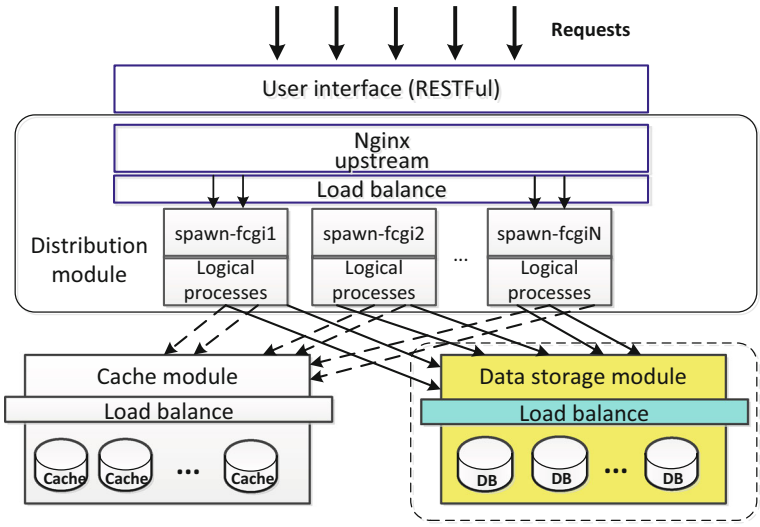

Fig. 1 Architecture of MyStore

and the returned value will be inserted to cache). If user executes a POST operation without key, it will create a new item in database and return a key value to user, this key will be set to cache. If user provides a key for POST operation, the data item in cache and database will be updated. DELETE operation must have a key. The item with this key will be deleted from cache and set to be unavailable in database.

For distribution module, Nginx is chosen to distribute requests from users to spawn-fcgi that is responsible for maintaining the lifecycle of logical processes. The distribution is based on round-robin algorithm. Logical processes are used to process concrete data resource requests including POST, GET, and DELETE, which are implemented in Python language.

Cache module is an independent memory cache system consisting of several cache servers, which are responsible for different partitions of data resources. Their load balances are based on the hash of resources' keys. Unstructured data items in cache are stored in {*key: value*} format using LRU (*Least Recently Used*) algorithm for age-out. The *key* is assigned as a universal ID, while the *value* is the unstructured data item. Items which are read/inserted/updated recently will be inserted into cache.

Finally, the most important part of the storage system is the data storage module for the persistence of the distributed unstructured data. This module focuses on distributing data for eventual consistency, incremental scalability, state transferring, failure detection and failure recovery, while providing functions of complex query like some relational databases and MongoDB. These features are required by many massive users-oriented applications including VeePalms. It will be discussed in the next sub-section minutely.

Additionally, for secret access, it is impossible to authorize data access through session or cookie since RESTful interfaces mentioned are stateless. The only way left is to consider URI-based digital signature. Clients and servers have an agreement on secret key creation and mechanism on how to generate the signature and authenticate it. The mechanism is shown in Fig. 2. The secret key and the token are known to both user and servers. The difference between them is that the secret key is a string to identify unique user and the token is a string to identify a single request. MD5 hash is applied to generate signature.

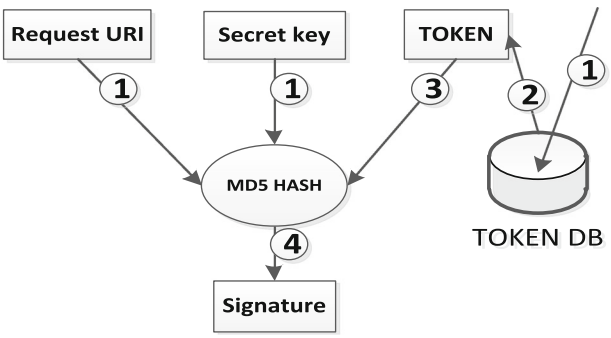

Fig. 2 The process of generating signature

Once users need to request data, the first thing is to get TOKEN from TOKEN DB, which will be combined with resource's URI and the secret key that can be obtained from web interface. The next step is to use the TOKEN, the request URI and the secret key to generate digest signature string with MD5 hash method. With the signature, TOKEN and request URI, a new authorized request URI can be formed. The servers will use the same method to authorize the digest signature and data request. The performance of the whole signature process is proven to be effective.

5 The data storage module

The data storage module is the most important part of MyStore. It uses clustered MongoDB for data persistence. An updated gossip mechanism and a revised NWR mode are used to combine multiple MongoDB cluster nodes to improve the availability and scalability of the system. An efficient consistent hashing algorithm is presented to distribute data on multiple nodes. A revised virtual node method is also applied to address the limitation of the number of physical storage nodes. First of all, we discuss the framework of this module. Then some key technologies will be introduced in details.

5.1 The framework of the data storage module

Figure 3 shows the brief architecture of the data storage module of MyStore. The module has three layers.

The upper layer is the message framework developed based on a non-blocking network event driven framework called Netty, which is a NIO (*New I/O*) client/server framework which enables quick and easy development and deployment of network applications. It is used here as TCP (*Transmission Control Protocol*) server for messages delivery and distribution, which provides a good event-driven, asynchronous

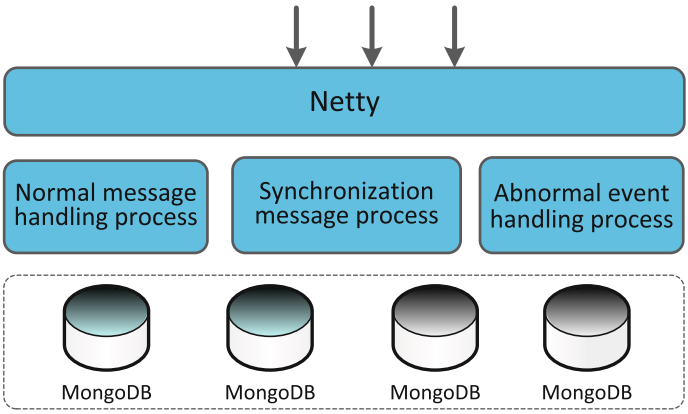

Fig. 3 The data storage module

programming architecture for the query management. It can support high concurrent requests from massive users.

The middle layer consists of some important components for message handling and event dealing, which are the normal message handling process, the abnormal event handling process and the synchronization message process. They also are the emphases of the module.

- The normal message handling process is in charge of the normal request messages from users and delivers them to appropriate MongoDB nodes.
- The abnormal event handling process is responsible for handling abnormal events such as the failures and departures of some storage nodes, network exceptions, disk errors.
- The synchronization message process exchanges the status messages among different storage nodes to let them update their status timely, and creates various events caused by the changes of these statuses.

The lower layer is a cluster of MongoDB nodes, which are combined by the middle layer to provide high availability for storage. Here, MongoDB database is responsible for data persistence.

The storage module wraps the interfaces mainly using three kinds of operations: *Connect, Get* and *Put*. *Connect* is corresponding to the creation of data connection to MongoDB. *Get* and *Put* are corresponding to two kinds of methods (GET/POST) the system provides to users. For simplifying the interface, *Put* is also in charge of the request from DELETE operation of users.

Typically, a *Get/Put* request firstly gets access to the module by the sockets provide by Netty. Then, the module assigns the request to the proper node in the storage cluster. The node finds out and determines the replicas for the key. For *Get* operation, the node uses the key to search results from some data nodes and waits for a quorum of replicas from them.

In original MongoDB, there is no real TCP connection created when a request is forwarded to the database, which limits the capacity of handling exception of the system. To meet the requirement of handling exception, a function of connection testing is added to the wrapped *Connect* operation in MyStore. Only when the connection to the database is built really, the *Connect* will return *true*, otherwise *false*. So the *Connect* operation mainly includes the following three steps:

(1) To create a database connection pool: The system uses a connection pool to realize the MongoDB connections management. The main principle of connection pool is to create a certain amount of connections in memory in advance. The connection pool can reduce the time of the connection establishment and release significantly and improve access speed. Here, the connection pool is implemented as a singleton, which means only one instance of the connection pool exists in the system.
(2) To configure parameters of connections: Connection parameters are divided into two types: ones for connection pool and ones for database. For the former, the parameters that mainly affect the performance of the connection are the following three ones: *connecttimeoutms*, *sockettimeoutms* and *autoconnectretry*. The *connecttimeoutms* denotes the value of the connection timeout, the *sockettimeoutms*

is the socket overtime and the *autoconnectretry* decides whether re-connection should be made when a connection failed. For the latter, the main parameters of the database include the IP address of database server, monitoring port, database name.

(3) To test database connection: The test of database connection is based on the interface of getting version that the MongoDB provided. First, a connection is obtained from the connection pool. Then, a query is made for obtain the version of the database specified in the step (2). If the operation is finished successfully without any exceptions thrown, the test passes. Otherwise, it fails.

For *Get* and *Put* operations, another important point is the exception handling. Because the results of *Get* and *Put* operations affect the availability of the system directly, the exceptions caused by the two kinds of operations must be captured and handled: the connection exception and the reading/writing exception. The connection exception is that the connection breaks during the process of *Get/Put*, which results in the failure of reading/writing. The reading/writing exception is a kind of I/O anomaly. For *Get* operation, when some exception happens, an exception message will be created and released from servers. No more enforced reading operation is required, because the failure of reading is a kind of exception that the system can tolerate. Benefiting from replication mechanism that will be discussed in the coming sub-section, the reading failure of a single node will not cause the read failure of the whole system. For *Put* operation, because the system cannot tolerate writing failure, when some exception happens, the system must find other storage node, and try to write several times to guarantee the success of writing, while send some notifying messages to the exception handling process. It also will be discussed in details in the next sub-section.

5.2 Some key issues in the storage module

5.2.1 Distribution

The first key problem of the storage module is how to route the *Get/Put* requests to the proper nodes (MongoDB nodes). For *Get*, the node routed should be the closest to the data's replicas. For *Put*, the replicas should be evenly distributed to the system. At last, when nodes are added or removed, the amount of data migration in the system should be as little as possible.

Based on these requirements, this module distributes data across the cluster nodes using consistent hashing [24] (seen in Fig. 4). In consistent hashing, the output of hash function is a ring. Each physical node is assigned a random value that presents the position on the ring. Each data to be written is identified by a string key (that is self-key specified by client). The hash function maps the string key to a random value, which also represents on the ring. Then by walking the ring clockwise, the node with a position larger than the data's position is found. This node is the proper node to store the data entity. The hash function used for generating a random value is the Ketama hashing algorithm [25]. Thus, each node becomes responsible for the region on the ring and each data key falls into regions of the ring.

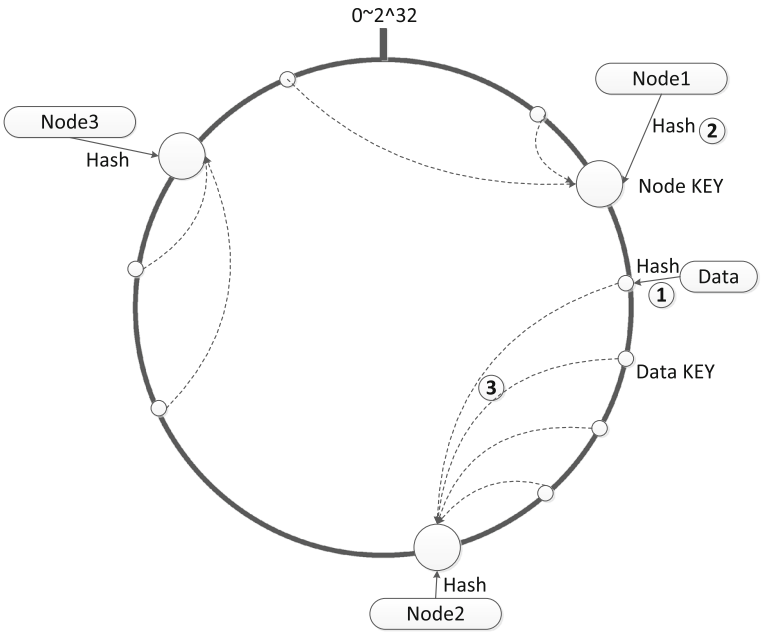

Fig. 4 The consistent hashing

Consistent hashing usually takes MD5 as the function of hashing, due to its good hashing performance. The formula is:

$$Y = \min(n \in N, md5(n) - md5(X)),\ md(n) > md5(X) \tag{1}$$

N is the set of all nodes. X is current KEY value of the data for hashing. Finally, the data will be delivered to the n node that satisfies the above equation (1).

Traditionally, the following algorithm is applies for the distribution [26]:

$$Y = hash(X) \text{mod } N \tag{2}$$

The principal advantage of consistent hashing is that departure or arrival of a node only affects its neighbor nodes, and other nodes remain unaffected. It is especially useful when MyStore makes data migration from one node to another. It greatly reduces the amount of data in migration.

But the basic consistent hashing presents some challenges. The value for node or data may not be equal probability on the ring, especially when the number of nodes in the system is limited. Our system refers to the *virtual node* method from Amazon Dynamo and revises it to address the issue. Each physical node is divided into multiple *virtual nodes*. The number of *virtual nodes* is determined by the performance of physical node. More powerful means more *virtual nodes*. The *virtual node*'s random key on the ring is decided by the physical node's key. *Virtual nodes* on the ring make the region of nodes smaller. When the data is distributed, it has a higher probability

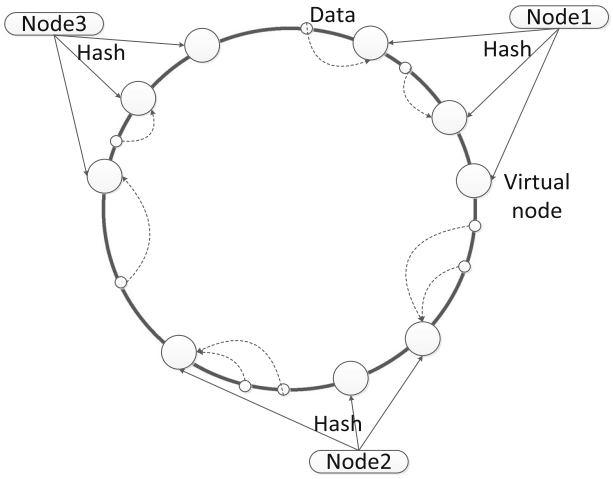

Fig. 5 Consistent hashing and *virtual node*

of being placed on other nodes. Consistent hashing and *virtual nodes* are shown in Fig. 5.

5.2.2 Replication

Our system uses replication to achieve high availability. Each data record is replicated at N physical nodes, where N is the replication factor configured in advance. Each *Put* request with key k is assigned to a coordinator node. The coordinator is responsible for the replications of the data records that fall within its range. The node firstly stores the data records locally, and then replicates the data to other $N-1$ nodes on the ring. The $N-1$ nodes are found from ring clockwise, and these nodes are physical nodes. The replications are run concurrently. Our system counts the amount of successful replications. If the number is greater than W that is configured as threshold of writing, a *Put* operation is considered to be successful, otherwise it is failed. For *Get*, there is also a threshold R. If the number of successful *Get*s is greater than R, the reading is completed successful. In addition, the *Get* operation gets all replications of the specified key, and checks the number of replication. If replications are less than N, maybe because of the departure of some nodes, some more replications are supplemented to achieve N ones in the system. The *Get/Put* latency is decided by the slowest replication. If one node fails, the system writes to the next node on the ring, makes each writing success.

In a distributed system, N is usually greater than 3. According to the CAP theory, it is impossible to provide all the three guarantees simultaneously. In a distributed system, partition tolerance is a basic requirement.

Therefore, the design process of a distributed system is a balance between consistency and availability. If the system needs high consistency, then configures $N = W$ and $R = 1$. This relationship provides low availability. If the system needs high availability, configures $W = 1$ and $R + W < N$. This configuration makes low writing latency and improves writing performance.

5.2.3 State transfer

There are many states in the system. These states are aroused by random events, such as adding or removing nodes, network exceptions, and disk errors. Some of events are triggered when *Get/Put* operations are executed. Others are triggered by some schedule tasks. For example, heartbeat detection is a schedule task for periodically testing the nodes in the network.

These states are aroused and transferred from one node to another in the system. So these nodes could sense each other and work together well. Our system uses a gossip protocol for transferring states. Gossip is a style of computer-to-computer communication. Gossip spreads information in a manner similar to the spread of a virus in a biological community. In the gossip protocol, each node has a group of states, which are key-value pairs. Each state is appended a version number. The greater of version number means newer states. There are three methods of gossip message processing: Push-Gossip, Pull-Gossip and Push-Pull-Gossip.

- Push-Gossip is the process that node A sends the group of states to node B, node B compares states to its local states, and then returns the Cartesian set of A and B.
- Pull-Gossip is that node A sends a digest (including key and version) to node B, B compares and returns some states that A needs to update.
- Push-Pull-Gossip is similar to Pull-Gossip, but the difference is when node B returns states that A needs to update, B requests the old states of A at the same time.

In MyStore, the Push-Pull-Gossip is used for transferring states. Figure 6 shows the communication process between two nodes by gossip. This communication includes three types of messages:

- *GossipDigestSynMessage*
- *GossipDigestAck1Message*
- *GossipDigestAck2Message*

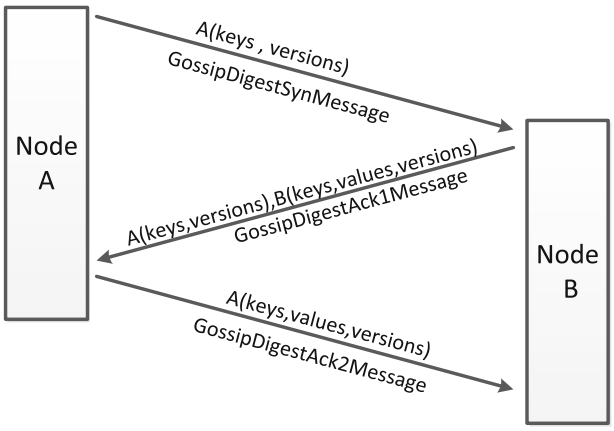

Fig. 6 Gossip between two nodes

The detail state transferring process of two nodes *A* and *B* is as follows:

(1) Firstly node *A* collects states with key and version and then sends it to node *B*.
(2) Node *B* compares the version number with its local states and starts to create the *GossipDigestAckMessage*. If local version number is greater than the remote version number, node *B* appends its local states (including key, value and version) to *GossipDigestAckMessage*. Otherwise, node *B* appends the remote states' key to *GossipDigestAck1Message*, and asks *A* for the latest version of states.
(3) Node *A* traverses every neighbor nodes and decides whether to update its local status. Further, it creates some events about this change. The abnormal event handling process will deal with them. If *B* needs the status values of *A*, *A* will response *GossipDigestAck2Message* to *B* including them.
(4) After receiving *GossipDigestAck2Message*, *B* updates its status, and triggers various status change events.

The following is a detailed template a gossip message. A gossip message is a string like this:

HostAddress@VirtualNode;bootGeneration:bootGenerationVersion;heartbeat: heartBeatVersion;load:loadVersion.

There are four fields in the message:

- The first field is node's host name and the number of virtual nodes, which also is an ID for a node distinguished from other node's state.
- The second one is some node booting information such as booting generation and version.
- The third is the heartbeat information such as the failure times of heartbeat detection.
- The last one is the node load information.

Our system divides the physical storage nodes in the system into two types: the seed nodes and the normal nodes. The seed node is considered as the node that has more information about nodes in the system than the normal nodes. As for normal node, it communicates with seed nodes periodically and gets latest information from them. The seed node communicates with other seed nodes, keeping the states all over the system consistent. Figure 7 shows the communication among seed nodes and normal nodes.

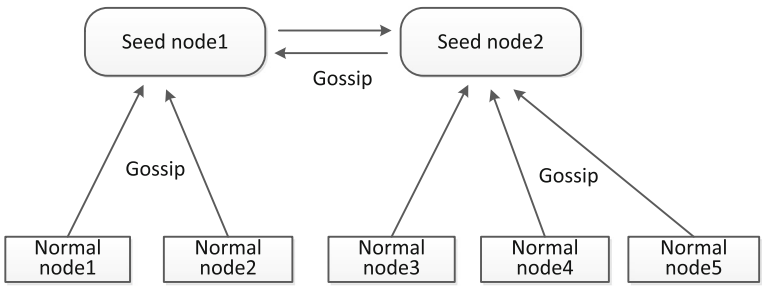

Fig. 7 Communication among seed nodes and normal nodes

5.2.4 Handling failures

There are two types of failures that our system should face and handle: short failure and long failure. Short failure not only means that the failure time is short, but also means that the failure could recover itself, such as network exception or the server process being blocked. Long failure is just the opposite of short failure. It usually means a failure that could not recover by itself automatically. In fact, it has not much to do with failure time.

Short failure usually affects the *Put* operation largely. Figure 8 shows the short failure and the handling process. When the data is to be replicated from node A to node B, if the node B becomes unconnected temporarily, it would turn to another temporary node C that is detected and found by heartbeat mechanism. Then A sends the replication request to C, while carrying the identifier of B. When C receives the request, it creates an index for the replication. At the same time, it detects the node B periodically by heartbeat service. When it finds that the B node is on-line again, the node C would write the data back to B.

Removing node is a good example of long failure. Removing node from the ring would affect the amount of replications of data that are stored in it. So after node removing, firstly the virtual nodes have to be removed too, then the replication should be complemented.

When a node is removed, two issues should be dealt with:

(1) The node synchronization mechanism should be taken to let all nodes in the system know that a node is in "long failure". All nodes should update their status information synchronously.

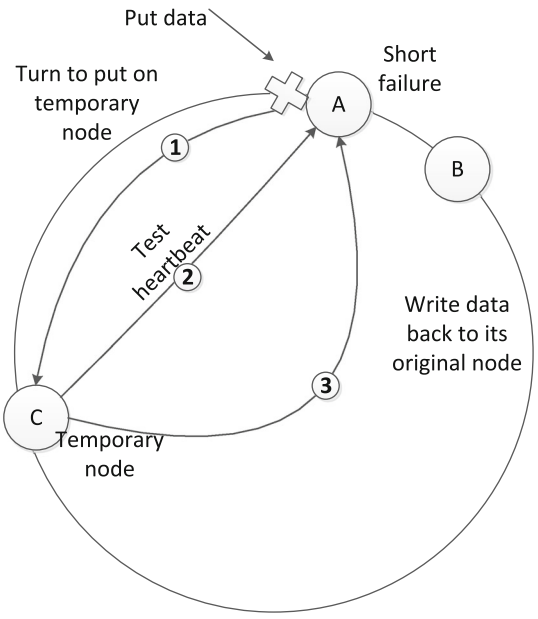

Fig. 8 Short failure and handling process

(2) Node removing will cause the number of the replications of data decreasing. So some new replicas should be created and distributed to other nodes.

For the first problem, the node synchronization mechanism should guarantee that all nodes in the system can receive the update message in specified time. In MyStore, the seed nodes are responsible for detecting "long failure" node, instead of normal. After all seed nodes have updated their statuses, all normal nodes can get the latest status information from their corresponding seed nodes.

For the second problem, the handling process is shown in Fig. 9.

Here, assume $N = 3$. Node C has three data replicas:

$$C : (H, A), (A, B), (B, C)$$

If node C leaves, some new replicas of the data should be complemented on other nodes, by taking a load balancing mechanism. Here,

(1) for node A, the replication of $(H, A]$ is complemented in node D;
(2) for node B, the replication of $(A, B]$ is complemented in node E;
(3) and for node C, the replication of $(B, C]$ is complemented in node F.

Further, adding node is another *failure* for system, although it has no impact on *Get/Put* operations. Adding node causes that partial data should be remapped and replicated to the new arrival node. Because the data to be replicated is stored on the next node of the ring before adding node, so the mapping and migrating operation are executed by the next physical node on the ring. The node fetches its data and the hashing value of each data is recomputed. If the value falls into the new node's region, then remove it from original node and add it to new node.

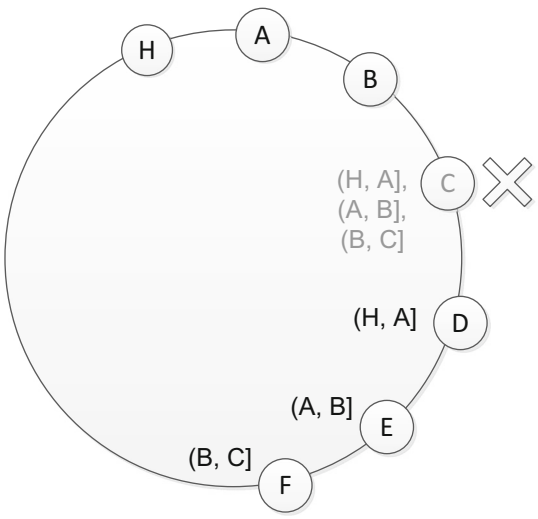

Fig. 9 Long failure and handling process

6 Experiments and performance

In this section, we do some experiments to evaluate the performance of the MyStore. The evaluation is divided into two aspects. One is the discussion about the performance of the whole system, and the other is deeper evaluation about the performance of the data storage module.

6.1 Performance of the system

This aspect of the evaluation consists of four tasks. The first is to evaluate the system average throughput of read operations (GET). The next is aimed to get average throughput of write operations (POST). The third one is a comparison with other storage patterns. Finally, we evaluate the scalability of the system.

We deploy the test system on an application node following five database nodes (seen in Fig. 10). A test node is set to serve as load generator that produces massive concurrent requests. All the servers are located in a LAN with a gigabyte switcher. Four cache servers are deployed on four normal DB nodes. The memory usage for all of them is 1 GB. Each node is equipped with two quad-core 2.26 GHz Xeon processors, 16 GB of memory, two gigabyte Ethernet links and two 146 GB SAS disk, runs RHEL5.3 with kernel 2.6.18. Microsoft Web Application Stress Tool is used to simulate massive data requests.

Dataset for read and comparison evaluation consists of variety XML files with sizes between 3 and 600 KB. The amount of data items is 700,000, and total size is about 36 GB. The throughput of loading this dataset to this system is nearly 6 MB/s. The number of requests is 125 per second.

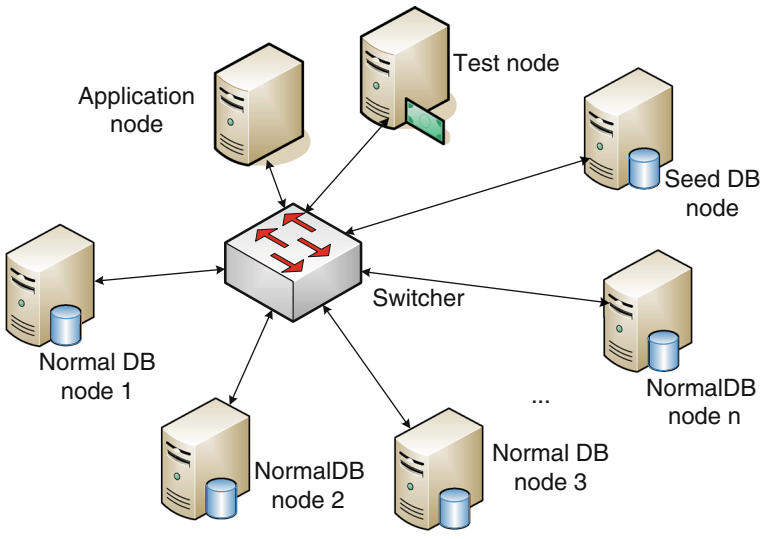

Fig. 10 The test topology of MyStore

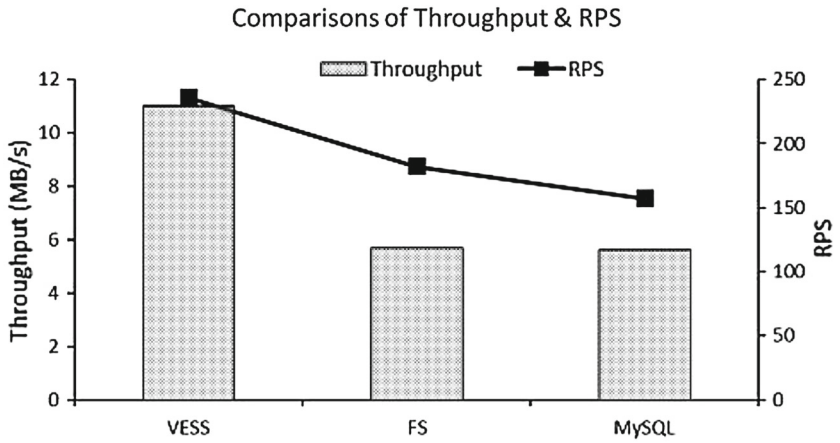

Fig. 11 Comparison of throughput and RPS in three systems

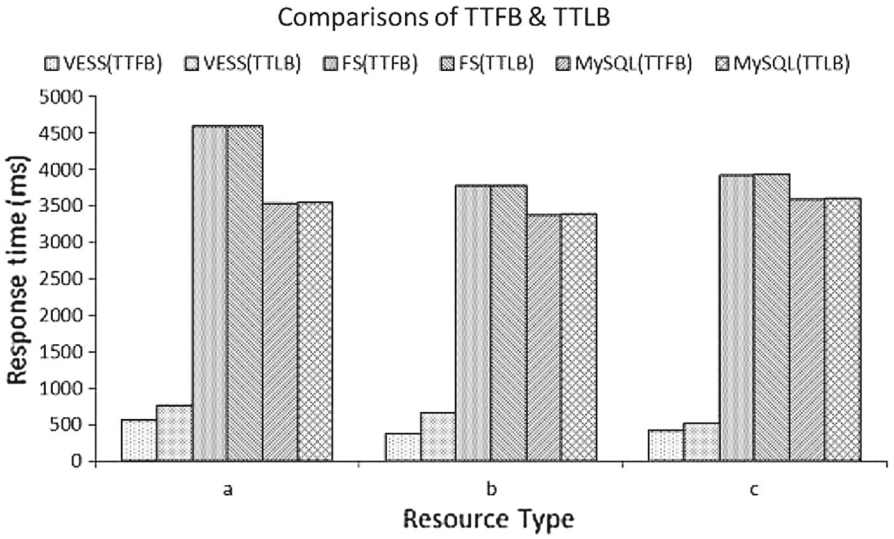

Fig. 12 Comparisons of TTFB and TTLB in three systems with different resource types

We keep running the system for 7 × 24 h under a heavy load by making simulation that 60,000 users generate requests within randomly delay between 0 to 500 ms. It performs stable enough both in functionality and performance. As shown in Fig. 11, the average throughput of read operations is nearly 11 MB/s, and the number of *requests per second* (RPS) are 236. Here, we compare MyStore with the ext3 file system and the master-slave structure MySQL relational database.

In order to make comparison of response time, we evaluate two factors: (1) *total time the first byte is received* (TTFB), (2) *total time the last byte is received* (TTLB). As shown in Fig. 12, where a, b, and c are three different types of resources.

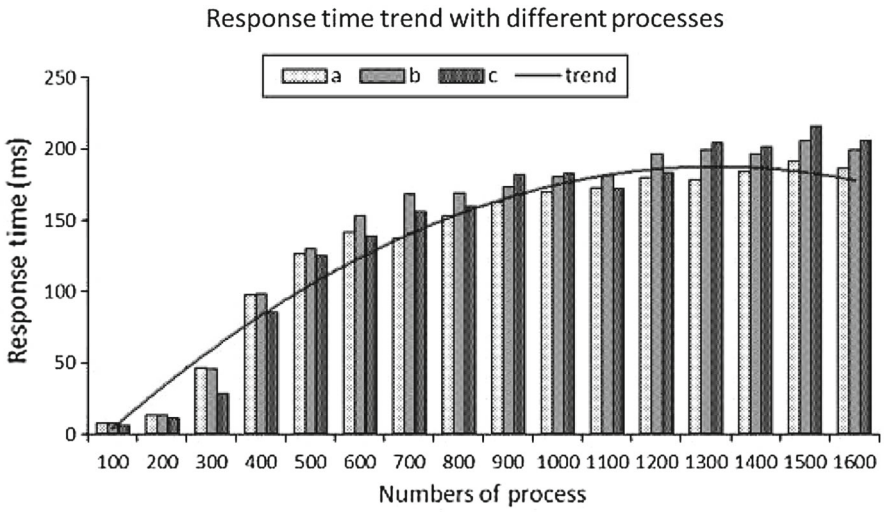

Fig. 13 TTFB trend with different processes

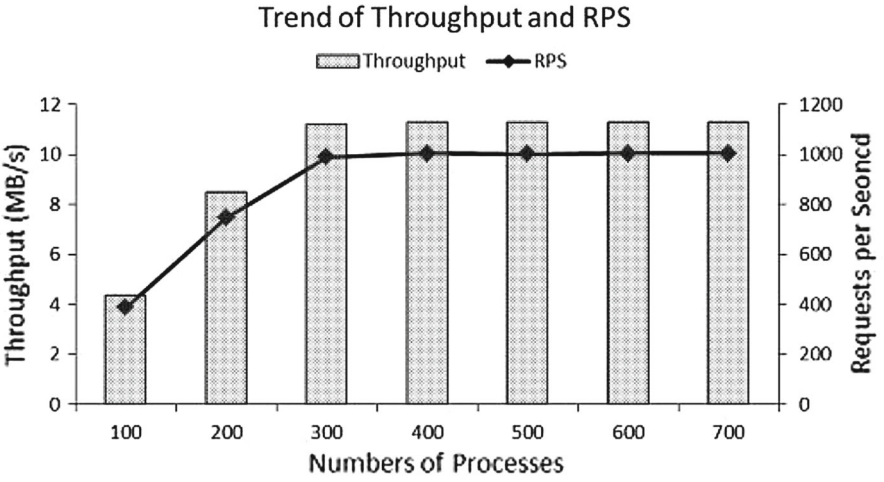

Fig. 14 Throughput and RPS with different processes

We can see that the system has a dramatic improvement on response time and much higher throughput. Additionally, we notice that the waiting for response from server spends most time of a request. Receiving data from server is rather quick. The three storage systems are all bounded to RESTful interfaces.

For the scalability of the system, we simulate users' concurrent requests with different numbers of processes increasingly. Figures 13 and 14 show the trends of TTFB, throughput, and RPS, respectively.

In Fig. 13, the response time increases almost linearly with the growth of the amount of processes used to generate concurrent requests, when it is less than 1,000. However,

when the amount of processes is more than 1,000, the response time almost does not change and be stable around 200 ms. Moreover, we can also get similar results from Fig. 14 that throughput and RPS will not change a lot after the amount of processes reaches a certain threshold, regardless of the increment of request processes.

In a summary, the system presented has a dramatic improvement on response time and much higher throughput, which can approve its efficiency clearly. When it reaches its peak capability, the throughput and RPS performs stable and the average response time is around some stable values. As the increment of backend application processes, cache servers and data servers, the system can get almost linear performance improvement. Moreover, it is easy for the system to scale-out.

Additionally, in the process of designing, implementing and maintaining the system, we gained some useful experiences. One of them is to use appropriate granularity of cache within different layers of the system and make users close to data where it is.

6.2 Performance of the data storage module

As mentioned above, to test the efficiency of data storage module, we deploy it on five DB nodes (seen in Fig. 10). The configuration (N, W, R) is (3, 2, 1). The service is running in Java virtual machine. The minimum of heap space is set to be 256 MB and maximum is 512 MB. The software environment and parameters for testing are listed in Table 1.

Here, Log4j is used to record the log data running and testing. Mongo is the Java driver of MongoDB server.

The dataset for *Get* and *Put* test consists of variety files with sizes between 18 and 7,633 KB (some large files are used to test the limitation of the module). These files include three types: experiment scenes, videos, and other documents (suffixed with doc, PDF, etc.). These files are sorted by their sizes and fetched to test system according to the Gaussian distribution of their sizes with parameters $\mu = 15$, $\sigma = 5$ that makes most of the sizes of the randomly selected files be got from the dataset. The amount of data items fetched is 10,000.

The evaluation has two steps:

- Firstly, generate a representative workload for it, and then measure the successful *Get*s/*Put*s per second.

Table 1 Software environment and parameters for data storage module

Software	Version	Parameters	Remark
MongoDB	1.6.3	Port: 27017	–
Java HotSpot Server VM	build 11.0-b15	–Xms 256M–Xmn 128M—Xmx512M	Mem. is 256M–512M
Netty	3.2.7	Port: 19870	–
Mongo	2.7.2	–	–
Log4j	1.2.16	log4j.rootCategory = ERROR, OUT	Record to files

Table 2 Probability of failures

	Type	Reason	Probability
1	Short-failure	Network exception	0.1
2	Short-failure	Disk IO error	0.002
3	Short-failure	Blocking processing	0.002
4	Long-failure	Node breakdown	0.001

- Secondly, inject some faults into the test system. There are two types of faults simulated: short failure and long failure. The network exceptions, disk IO errors, or blocking process may cause short failures. The long failure is that a node is off-line for a long time. Table 2 shows the happening probabilities of all failures assumed in this test system.

There are three topics evaluated here: the balancing performance, the performance comparison of *Put* operations without and with failures, and the *Put* performance comparison among different systems.

6.2.1 The balancing performance

Figure 15 shows the number of records in each physical node after all data replicated over all storage nodes.

The amount of data records is 10,000, and $N = 3$. So after all records are stored, there are 30,000 replicas in the system. Seen from Fig. 15, the average replicas of each node are 6,000. The total number of three nodes' replicas is 30,000. Although some random reasons cause the numbers of records of different nodes are different, this difference is negligible and acceptable. This shows that the module has a good performance on balancing and scalability.

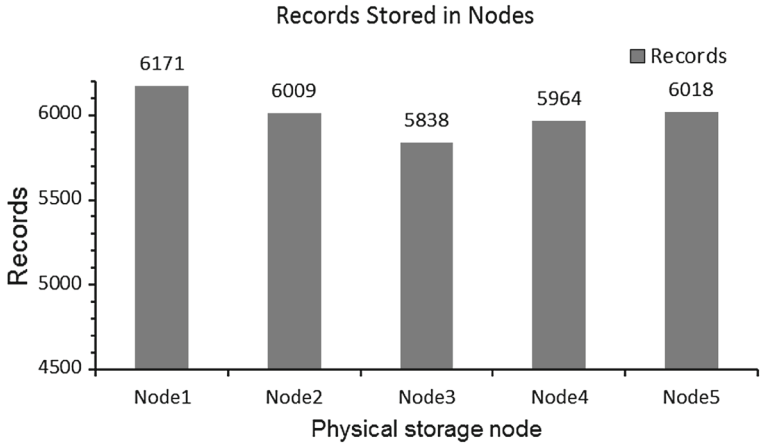

Fig. 15 Records in nodes

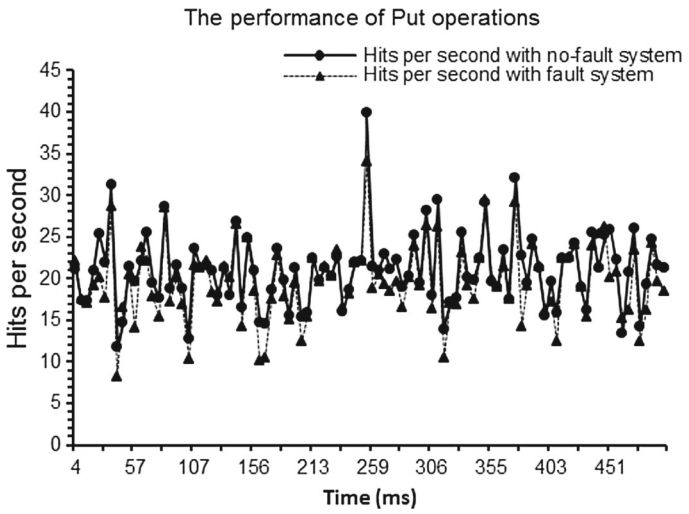

Fig. 16 Put performance of MyStore with no-fault and fault

6.2.2 The performance comparison of Put operations

Handling failures will decrease the performance of *Get/Put*. Since the performance of *Put* operation usually is affected by failures more largely, compared with the one of *Get*, so here, the performance of *Put* is considered for test. We compare the performance of *Put* operations under the same load with fault and no-fault system. Figure 16 show the successful hits per second under the two situations. It is obvious that the one with fault is lower than one with no-fault system because failure handling takes some time.

6.2.3 The Put performance comparison among different systems

Here we compare the *Put* performance by calculating the consuming time of every *Put* operation. For the convenience of statistics and show, all operations are sorted by their consuming time. Some representative operations are selected from all 10000 ones by interval of 100 operations.

Figure 17 shows the statistics of the sum of the representative *Put*s that are completed within specific consuming time among three situations respectively, which are MyStore with no-fault, MyStore with fault, and MongoDB with fault. Here, MongoDB is configured to be master-slave mode using three physical nodes.

The horizontal axis denotes the consuming time of *Put* operations. The vertical axis denotes the sum of all the *Put* operations whose consuming time is less than the consuming time specified by the horizontal axis.

It is clear that the module with fault is better than MongoDB with fault in the consuming time comparison. In one specified consuming time, the former has more

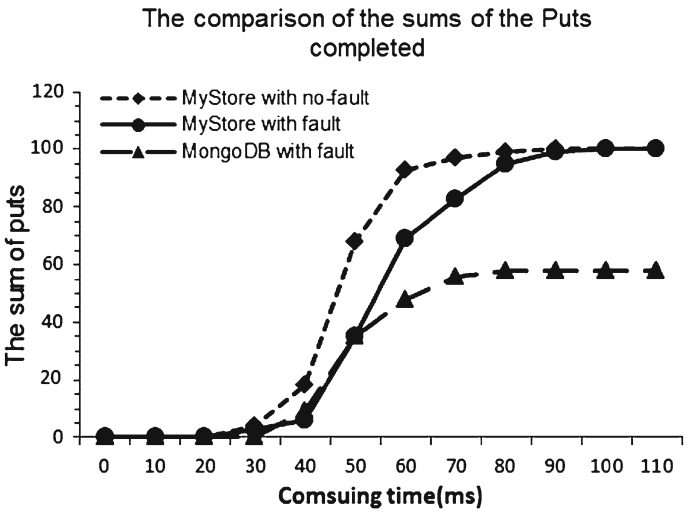

Fig. 17 Put performance comparison with MongoDB

operations finished than the latter. Of course, the one with no-fault has the best performance among these three.

In a summary, our system is competent to handle various failures, and the performance can be maintained in high level when failures take place. Additionally, it is a distributed system, and all physical nodes have open service interfaces over TCP, which lets clients can connect to any node in the system to get/put data, this is very useful when some nodes are busy for handling requests.

MyStore has been applied in the VeePalms for storing unstructured data such as scene files, files for guidelines, files for instructions. The VeePalms can support about 65,000 concurrent users to do online virtual experiments. It has run for about 2 years and the practice proves that MyStore has high availability and scalability.

7 Conclusion

In this paper, a high-available and scalable distributed storage system for the unstructured data management is presented. It has been applied in a virtual experiment platform that contains a large number of unstructured data such as video files for guidelines, experiment components, scenes in XML, documents in PDF and DOC. Experimental results show that MyStore dramatically improves the ability of handling failures and the availability, scalability of the system. Future work involves further improving performance of handling failures and solving problems on data's consistency. More attentions also will be paid to the segmentation, storage and schedule of large video files.

Acknowledgments This paper is supported by the project of National Science & Technology Pillar Program of China under grant No. 2012BAH14F02, China National Natural Science Foundation under grant No. 61272408, 61322210, CCCPC Youngth Talent Plan, and Natural Science Foundation of Hubei under grant No. 2012FFA007.

References

1. Gartner Inc. http://www.gartner.com/
2. Abadi J (2009) Data management in the Cloud: limitations andopportunities. IEEE Data Eng Bull 32(1):3–12
3. Lakshman A, Malik P (2009) Cassandra—a decentralized structured storage system. In: Proceedings of the 3rd ACM SIGOPS international workshop on large scale distributed systems and middleware (LADIS'09). ACM, New York, pp 35–40
4. Stonebraker M (2010) SQL databases v. NoSQL databases. Commun ACM 53(4):10–11
5. DeCandia G, Hastorun D, Jampani M (2007) Dynamo: Amazon's highly available key-value store. In: Proceedings of twenty-first ACM SIGOPS symposium on operating systems principles. ACM, New York, pp 205–220
6. Chang F, Deam J, Ghemawat S, Hsieh W, Wallach D, Burrows M, Chandra T, Fikes A, Gruber RE (2008) Bigtable: a distributed storage system for structured data. ACM Trans Comput Syst 26(2):1–26
7. Lakshman A, Malik P (2009) Cassandra: structured storage system on a p2p network. In: Proceedings of the 28th ACM symposium on Principles of distributed computing. ACM, New York, pp 5–5
8. Banker K (2011) MongoDB in action. Manning Press, USA
9. Ford D, Labelle F, Popovici F (2010) Availability in globally distributed storage systems. In: Proceedings of USENIX conference on operating system design and imlementation (OSDI'10). USENIX, Berkeley, pp 1–7
10. Pritchett D (2008) BASE: an acid alternative. ACM Queue 6(3):48–55
11. Jim G (1981) The transaction concept: virtues and limitations. In: Proceedings of the 7th international conference on very large databases (VLDB). IEEE, New York, pp 144–154
12. Vogels W (2009) Eventually consistent. Commun ACM 52(1):40–44
13. Bermbach D, Klems M, Tai S (2011) MetaStorage: a federated cloud storage system to manage consistency-latency tradeoffs. In: Proceedings of 2011 IEEE international conference on cloudcomputing (CLOUD'11). IEEE, Los Alamitos, pp 452–459
14. Ghandeharizadeh S, Goodney A, Sharma C (2009) Taming the storage dragon: the adventures of hoTMaN. In: Proceedings of the 35th international conference on management of data (SIGMOD'09). ACM, New York, pp 925–930
15. Sun X, Zhou L, Zhuang L, Jiao W, Mei H (2009) An approach to constructing high-available decentralized systems via self-adaptive components. Int J Softw Eng Knowl Eng 19(4):553–571
16. Garfinkel S (2007) An evaluation of Amazon's grid computing services: EC2, S3 and SQS. Tech. Rep. TR-08-07, Harvard University, Cambridge
17. Abadi D (2012) Consistency tradeoffs in modern distributed database system design: CAP is only part of the story. Computer 45(2):37–42
18. Shah D (2008) Gossip algorithms. Found Trends Netw 3(1):1–125
19. Yu Q (2010) Metadata integration architecture in enterprise data warehouse system. In: Proceedings of 2010 2nd international conference on information science and engineering (ICISE 2010). IEEE, Piscataway, pp 340–343
20. Verma A, Llora X, Venkataraman S, Goldberg DE, Campbell RH (2010) Scaling eCGA model building via data-intensive computing. In: Proceedings of 2010 IEEE congress on evolutionary computation (CEC'10). IEEE, Piscataway, pp 1–8
21. Fox A, Gribble SD, Chawathe Y, Brewer EA, Gauthier P (1997) Cluster-based scalable network services. ACM SIGOPS Oper Syst Rev 31(5):78–91
22. Brewer EA (2000) Towards robust distributed systems. In: Proc. ACM symposium on princinples of distributed computing (PODC'00). ACM, New York, p 7
23. Carstoiu B, Carstoiu D (2010) High performance eventually consistent distributed database Zatara. In: Proceedings of 2010 6th international conference on networked computing (INC'10) IEEE, Piscataway, pp 54–59
24. Lee B, Jeong Y, Song H, Lee Y (2010) A scalable and highly available network management architecture on consistent hashing. In: Proceedings of 2010 IEEE global telecommunications conference (GLOBECOM'10), pp 1–6
25. Petrovic J (2008) Using memcached for data distribution in industrial environment. In: Proceedings of 2008 3rd international conference on systems (ICONS'08). IEEE, Piscataway, pp 368–372
26. Byers J, Considine J (2003) Simple load balancing for distributed hash tables. Comput Sci 2735(2003):80–87