

# **ZStore - A distributed key-value storage system**

Core Database System Group - <http://research.vng.com.vn>

## **Technical Report**

**(December 30<sup>th</sup> 2010, Continuing update)**

**Nguyen Quoc Minh,**

Research manager, VNG Corp,

E-mail: [minhng@vng.com.vn](mailto:minhng@vng.com.vn)

**Nguyen Trung Thanh,**

Technical leader, VNG Corp,

E-mail: [thanhnt@vng.com.vn](mailto:thanhnt@vng.com.vn)

**To Trong Hien,**

E-mail: [hientt@vng.com.vn](mailto:hientt@vng.com.vn)

**Pham Trung Kien,**

E-mail: [kienpt@vng.com.vn](mailto:kienpt@vng.com.vn)

**Cao Minh Phuong,**

E-mail: [phuongcm@vng.com.vn](mailto:phuongcm@vng.com.vn)

**Nguyen Quoc Nhan,**

E-mail: [nhannq@vng.com.vn](mailto:nhannq@vng.com.vn)

**Pham An Sinh,**

E-mail: [sinhpa@vng.com.vn](mailto:sinhpa@vng.com.vn)

Abstract

1. Introduction

2. Related Works

4. High Scalability

4.1. Consistent Hashing

4.2. Golden Ratio

4.3. Adaptive Partition

4.4 Elasticity

5. Availability

5.1 Hinted handoff

6. Reliability

6.1 Replication

6.2 Fault tolerance

Hinted handoff: handling temporary failure

Merkle tree: handling permanent failure

6.3 Failure Detection

7. Evaluation

7.1 Partitioning

7.2 Hinted Handoff

7.3 Merkle Tree

7.4 Failure Detection

8. Conclusion

References

# Abstract

[ZingMe](#) is the biggest social network in Vietnam, which currently serves 25 million users and would reach **41 million customers** in **2014**. At this scale, there are many challenges for VNG's backend system such as high scalability and reliability, the most challenging issues at large scale distributed systems.

Motivated by how Dynamo, a distributed key-value storage system from Amazon, solved trade-off problems in a way that assured great availability; my team at [VNG](#) started designing a Dynamo-like system with the following features: persistent storage, memory caching, high scalability and fault-tolerance. ZStore's design is some how like Dynamo's. We make extensive use of consistent hashing and replication scheme to achieve our design goal. However, because ZStore system is used for services of ZingMe's social network and it's online games, our design concentrates on high availability in read more than write.

# 1. Introduction

ZingMe has approximately 1.7M daily active users; however, this number may reach 20M in 2014. Therefore, the need of scaling the the ZingMe's backend is urgent. There are two ways to scale a system: scaling-up and scaling-out. Currently, ZingMe's backend uses scaling up method by adding more resources such as CPUs or RAM to a single node. This method leads two most concern problems which are high cost and limited scalability. Scaling out is the solution for solving these problems. However, this method may cause to some other problems such as increasing complexity of programming, management and latency as well.

When designing a distributed web services, there are three properties that are commonly desired:

1. **Consistency:** all nodes see the same data at the same time
2. **Availability:** node failures do not prevent survivors from continuing to operate
3. **Partition tolerance:** the system continues to operate despite arbitrary message loss

According to CAP theorem [7] , a distributed system can satisfy any two of these properties at the same time, but not all. We observed that many components need only primary key access such as: user profiles, feeds, game items, etc. Thus we have developed Zstore - a multi purposes distributed key-value storage system for future Zing's backend services such as: feed ranking, social games, etc. Moreover, current ZingMe's backend rely on MySQL which fall in RDBMS is not easy to scaling-out. For all of these reasons, we believe that Zstore will be used for ZingMe in the the future.

ZStore is a distributed key-value storage system which bases on Dynamo with the following features:

- High scalability
  - Peer-to-peer, easy to scale using commodity machine.
- High availability
  - No single point of failure.
  - No lock and always readable.
- Fault tolerant
  - Hinted handoff to handle temporary failure.
  - Merkle tree to handle permanent failure.
  - Gossip protocol to quickly detect failure.
- Eventually consistent
  - Replicas are loosely synchronized.

We are currently testing the ZStore with feed ranking's services.

## 2. Related Works

### Cassandra

Cassandra is a column-family based data storage; the data model of Cassandra is BigTable. Like Amazon's Dynamo, Cassandra provides structured key-value storage with eventual consistency. The most interested features of Cassandra is no single point of failure, fault-tolerance, tunable consistency and elasticity.

### Dynamo

In Dynamo, to achieve high availability and tolerance partition:

1. Dynamo provides a simple primary-key only interface
2. Dynamo replicates its data on multiple hosts.
3. To achieve high availability for write, Dynamo use Vector clocks with reconciliation during reads.
4. Sloppy Quorum and hinted handoff to provides high availability and durability guarantee when some of the replicas are not available.

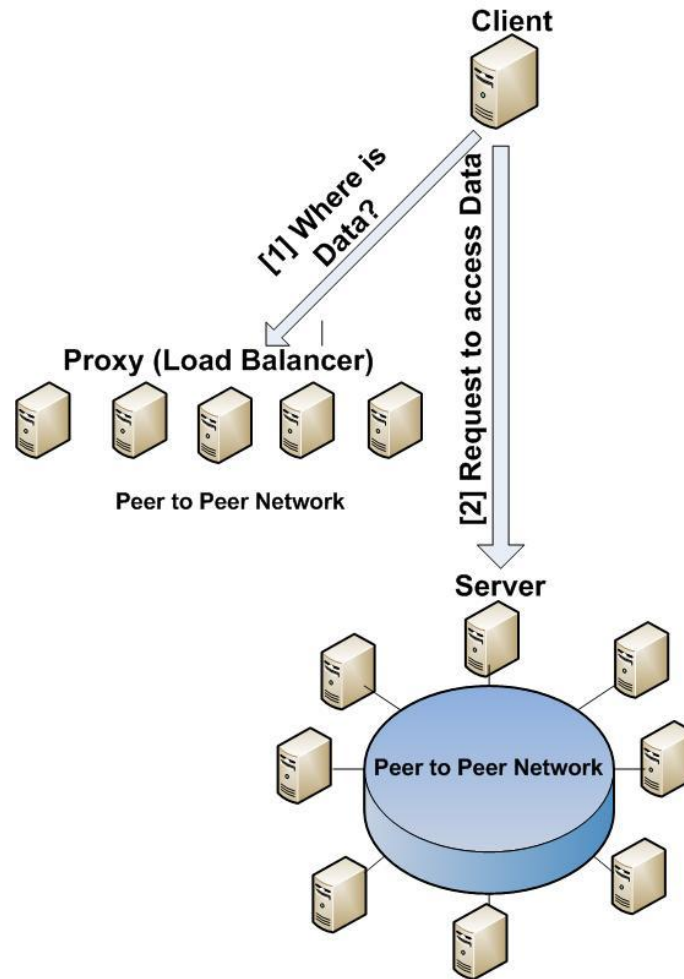
To achieve tolerance partition Dynamo uses:

1. Gossip-based membership protocol and failure detection.
2. Anti-entropy using Merkle trees to synchronizes divergent replicas in the background.
3. Sloppy Quorum and hinted handoff

### Redis

Redis is key-value storage which supports strings and several abstract data types: **Lists** of strings, **Sets** of strings, **Sorted sets** of strings, **Hashes** where keys are strings and values are either strings or integers. Besides, Redis supports high level atomic server side operations like intersection, union, and difference between sets and sorting of lists, sets and sorted sets. Redis is amazingly fast because it's operations is performed in an asynchronous ways and it does not supports transactions.

### 3. ZStore System Architecture



#### 1. ZStore

Consists of N nodes that we put together in a logical *ring*, each node responsible for a range of key and serve as replica for other node.

#### 2. Get put operations

1. `get(key)`: return value from local storage.
2. `put(key, value)`: save value to local storage, automatically replicate to other nodes.

#### 3. Client visibility

1. Send a request to any of proxy to know which node contains the key.
2. The request is forwarded to node associated with the key.

#### 4. Proxy

- Contains a routing table: map key to corresponding node.
- Each proxy aware of other proxies state in the network.

The solutions to three most challenging problems of distributed storage system is summarised as in following table.

<b>High Scalability</b>	Partitioning <ol style="list-style-type: none"><li>1. Consistent Hashing Strategy.</li><li>2. Golden Ratio Strategy.</li><li>3. Adaptive Partition Strategy.</li></ol>
<b>Availability</b>	Hinted Handoff
<b>Reliability</b>	<ol style="list-style-type: none"><li>1. Replication</li><li>2. Failure Detection<ol style="list-style-type: none"><li>a. Gossip-based membership protocol.</li></ol></li><li>3. Fault-tolerance<ol style="list-style-type: none"><li>a. Temporary failure: Hinted Handoff</li><li>b. Permanent failure: Anti-entropy using Merkle trees to synchronizes divergent replicas in the background.</li></ol></li></ol>

## 4. High Scalability

The partitioning module distributes the load across multiple storage hosts. We have three schemes for partitioning which are Consistent Hashing (like Dynamo) and two other new ones for specific purposes. Golden Ratio scheme addresses load balancing problem and Adaptive Partition scheme solves problems with Feed like Applications, such as “new keys” are highly-access keys and read is much more than write.

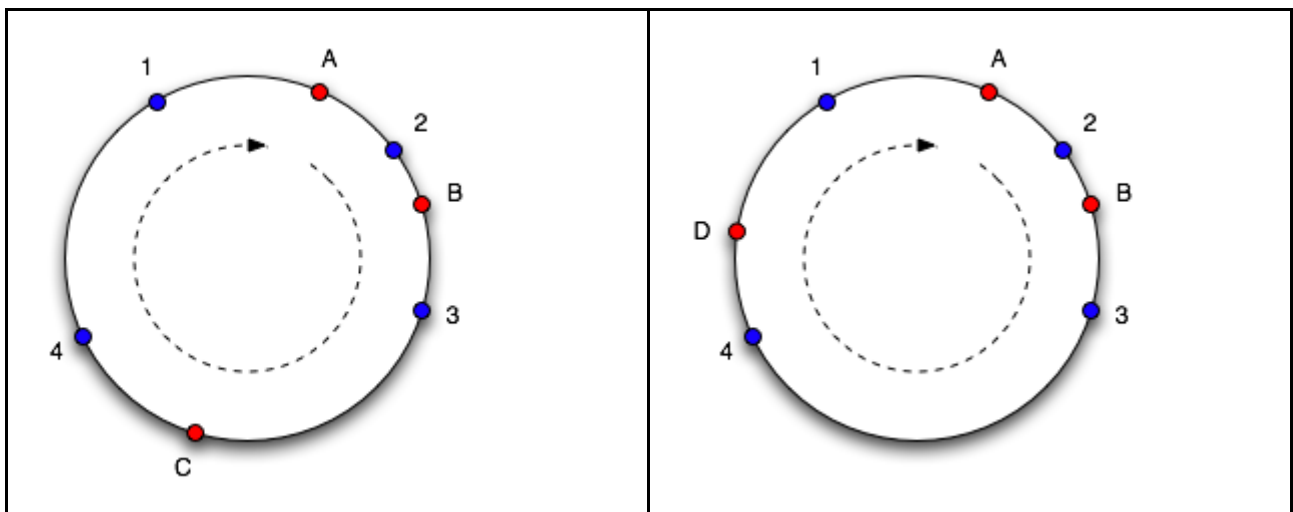
### 4.1. Consistent Hashing

ZStore currently uses Consistent Hashing (CH) scheme to increase scalability. The purposes of CH is to partition and replicate data (which will be discussed in the next section). The output range of a hash function is treated as a fixed circular space or range (for example AB, BC, etc) in the ring. Each node in the system is assigned a random value within this space which represents its position.

Each data item identified by a key is assigned to a node by hashing the data item's key to yield its position on the ring (1, 2, 3, 4). Each node become responsible for the region in the ring between it and its predecessor node on the ring. The advantage of CH scheme is that departure or arrival to a node only affects its immediate neighbors and other nodes remain unaffected. On the other hand, CH means - **consistently** maps objects to the same cache machine, as far as is possible, at least.

For example:

- Object 1 and 4 belong in node A, object 2 belongs in B and object 3 belongs in C.





- Consider what happens if C is removed: object 3 now belongs in A, and **all the other object mappings are unchanged**. If then another node D is added in the position marked it will take objects 3 and 4, leaving only object 1 belonging to A.

The results of Consistent Hashing has been proved in [7] and [8].

## 4.2. Golden Ratio

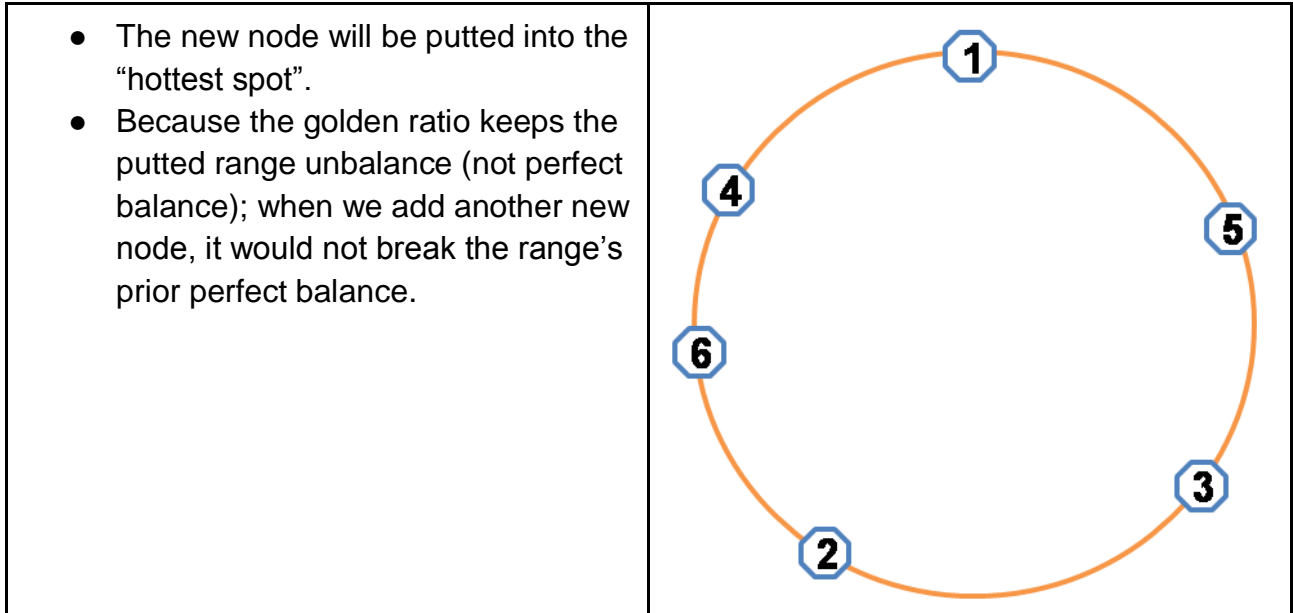
### Analysis Load Balancing

If the existing system is balanced and you add new nodes to the ring, the system may be unbalanced. For example, existing system has N nodes; if you need perfect balance, you have to add N other nodes at a time (each old node will share its load with a new node). If your current system has 100 nodes and you want add 120 new nodes, let add 100 new nodes first before adding the last 20 nodes.

Due to the cost, however, adding many nodes at a time is not feasible in practice. We want to add only one node at a time, whilst still keeping good load balancing. We have two strategies:

**Manual strategy:** one way to get perfect balance is to compute range size for every node and assign them to each node manually by command. We therefore always put the new node into the “hottest spot” - the range with high load.

**Golden Ratio (GR) strategy:** when add a new node, the ring will automatically computed the largest range and put the new node into it. Then, the new node gets its position by dividing this range into two ranges according to golden ration (the below figure). GR strategy has satisfied our requirements:



## 4.3. Adaptive Partition

Adaptive Partition (AP) is an adaptive scheme for key distribution in Feed Applications.

**Motivation:** Dynamo currently uses two schemes for key distribution: Random Partition and Order Preserving Partition.

Random Partition (RP)	Order Preserving Partition (OPP)
+ Prop: Good for load balancing + Cons: Not good for multi request	+ Prop: Efficient for multi request and range queries + Cons: Can cause unevenly distributed data

Each scheme has its own advantages and disadvantages. However, no scheme is designed for a particular application - Feed Applications, in which “new keys” are highly accessed keys. This fact may cause unbalance with the Random Partition of Dynamo. We therefore designed a new partition named Adaptive Partition which has two characteristics. Firstly, not only speed of single request is improved significantly than Dynamo’s models but also we can exploit multi-request which is much faster than single request. Secondly, load balancing is still good for Feed like applications in which “read” is much more frequent than “write” and “new keys” are highly-accessed keys. A new problem is raised: **How to balance between speed (range queries) and load balancing?**

**Objectives of AP:** trade off between OPP and RP.

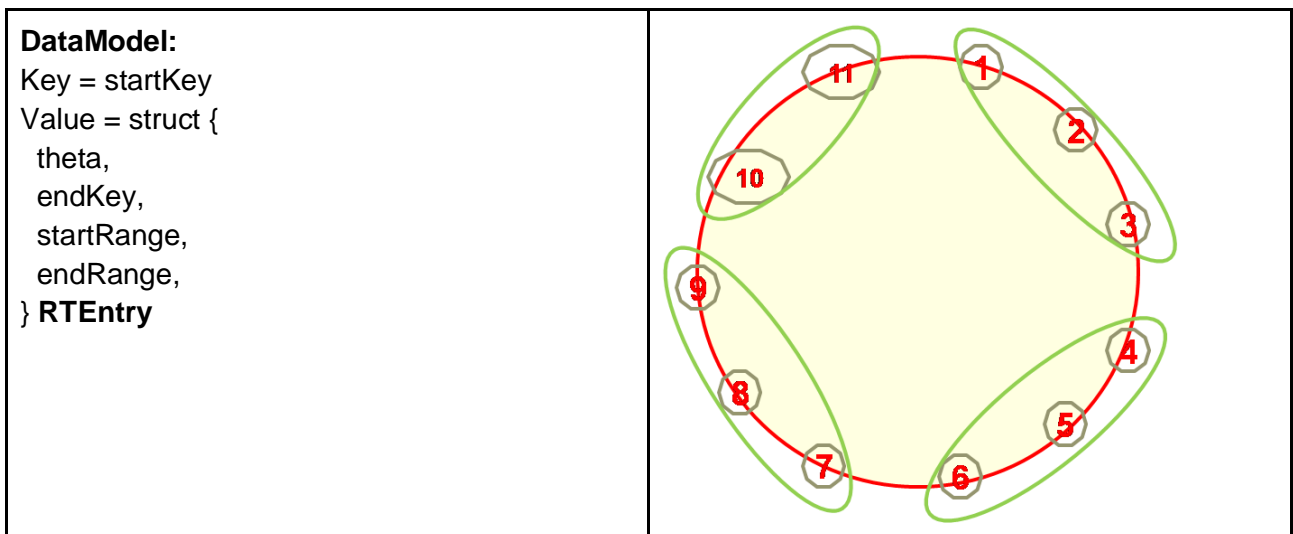
- O1: Improve single-request, multi-request: multi-get and get-range return keys stored in as small nodes as possible.
- O2: Increase load balancing with “new keys” problem.

### AP Model

To satisfy O1: Only one node is written at the period of time until it receives Theta single requests (Theta is a variable number which we need to optimize to obtain the good performance as well as load balancing; we also use Theta to indicate a period of time in which only one node is written). Therefore, multi-get and get-range will return only one node or a small successive nodes (1, 2, 3 in the figure below).

Let give an example about the most “recent news” in Facebook or ZingMe. The most 20 “recent news” of each user are likely posted in the last 12 hours to several days. Because during Theta (hours), only one node is written. The number of successive used nodes is therefore:  $12/\text{Theta}$  nodes. In the next part we will discuss how to calculate Theta to satisfy two constrains: Theta will be large enough so that multi-request will return a small number of successive nodes and Theta must not be too large to reduce overhead of the node which is being written (We can optimize Theta because in Feed Applications, “read” is much more than “write”).

We use A Routing Table (K, V) to map between key ranges to ranges of nodes.



### Routing Table

**$\mu$  (theta):** number of new keys/node (in a period of time).

**nk:** number of nodes per range (a number of successive nodes). We use “node range” to reduce the size of the Routing Table; therefore, reduce look up time.

### Routing table: $\mu=10^5$

$\mu$	$n_k$	Key		Range		Range Size
		Start	End	Start	End	
$10^5$	3	1	$3 * 10^5$	1	3	3
$10^5$	3	$3 * 10^5 + 1$	$6 * 10^5$	4	6	3
$10^5$	3	$6 * 10^5 + 1$	$9 * 10^5$	7	9	3
$10^5$	3	$9 * 10^5 + 1$	$10^6 + 2 * 10^5$	10	1	3
$10^5$	6	$10^6 + 2 * 10^5 + 1$	$10^6 + 8 * 10^5$	2	7	6
$10^5$	6	$10^6 + 8 * 10^5 + 1$	$2 * 10^6 + 3 * 10^5$	8	2	6

### Overhead at “Active node”

“Active node”: *the only node which is being written.*

- Not only being written but also being read with highest speed when comparing with the other nodes
- To reduce overhead of the active node, we need to reduce Theta. But It means that multi-get will return more nodes (We scarify speed in multi-get).

The previous question now became a new problem: **How to calculate Theta?**

### Load of “Active node”:

Load = Load (Get) + Load (Put)

$$= (T * \lambda) * \Psi + T * \Psi = T \Psi (\lambda + 1)$$

- $\Psi$ : “new keys” access rate.
- $\lambda$ : get/put rate.

Each node’s load has a constant upper bound:  $L > T \Psi (\lambda + 1)$ .

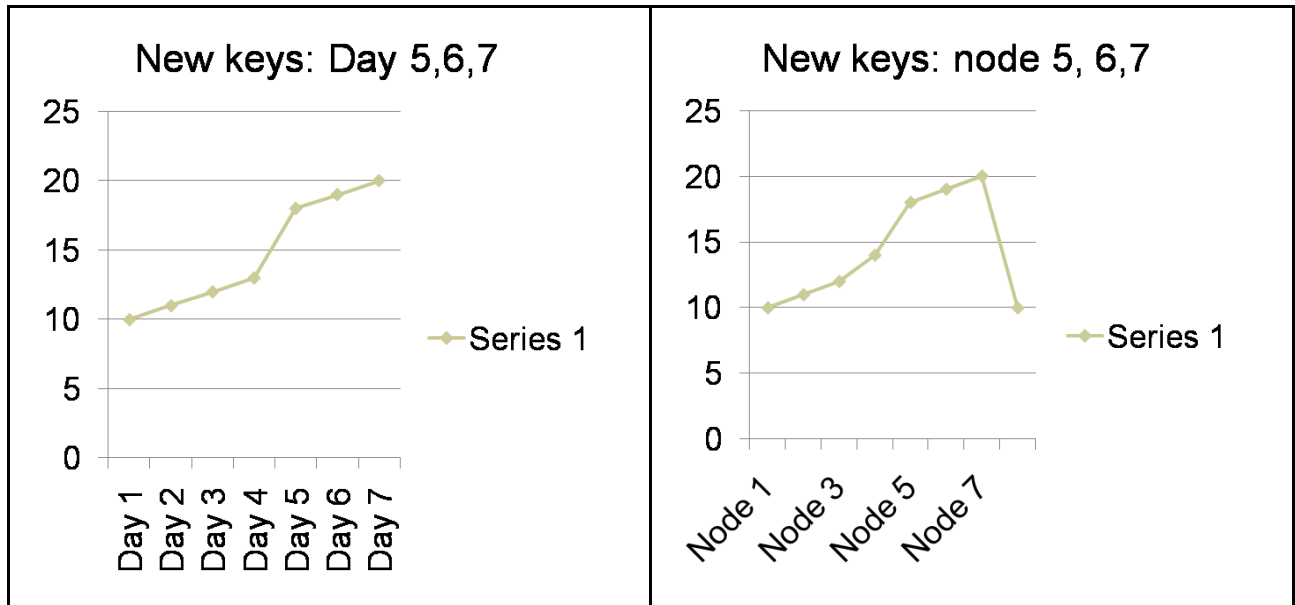
X: number of “new keys”.  $T > X/N$  (N: number of nodes)

$$X/N < T < L/(\Psi(\lambda+1)).$$

Theta has an upper bound and a lower bound. We need to change Theta in this range to get an optimum for speed or an acceptable SLA in average latency of the system.

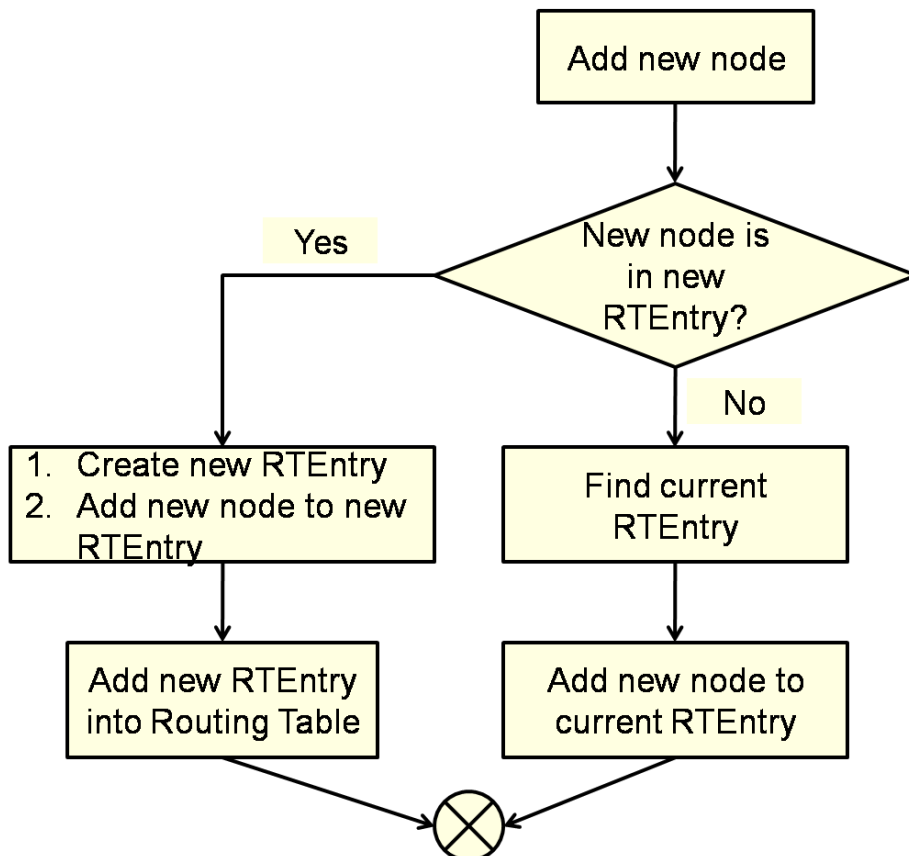
**What is “new keys”:** There are two ways to define “new keys”. One is based on load per day and the other is based on load per node. In the first approach, new keys are the keys which is written in the most recent days. In the second approach, new keys are the keys which is written to the most recent active nodes.

Base on load per day	Based on load per node
----------------------	------------------------

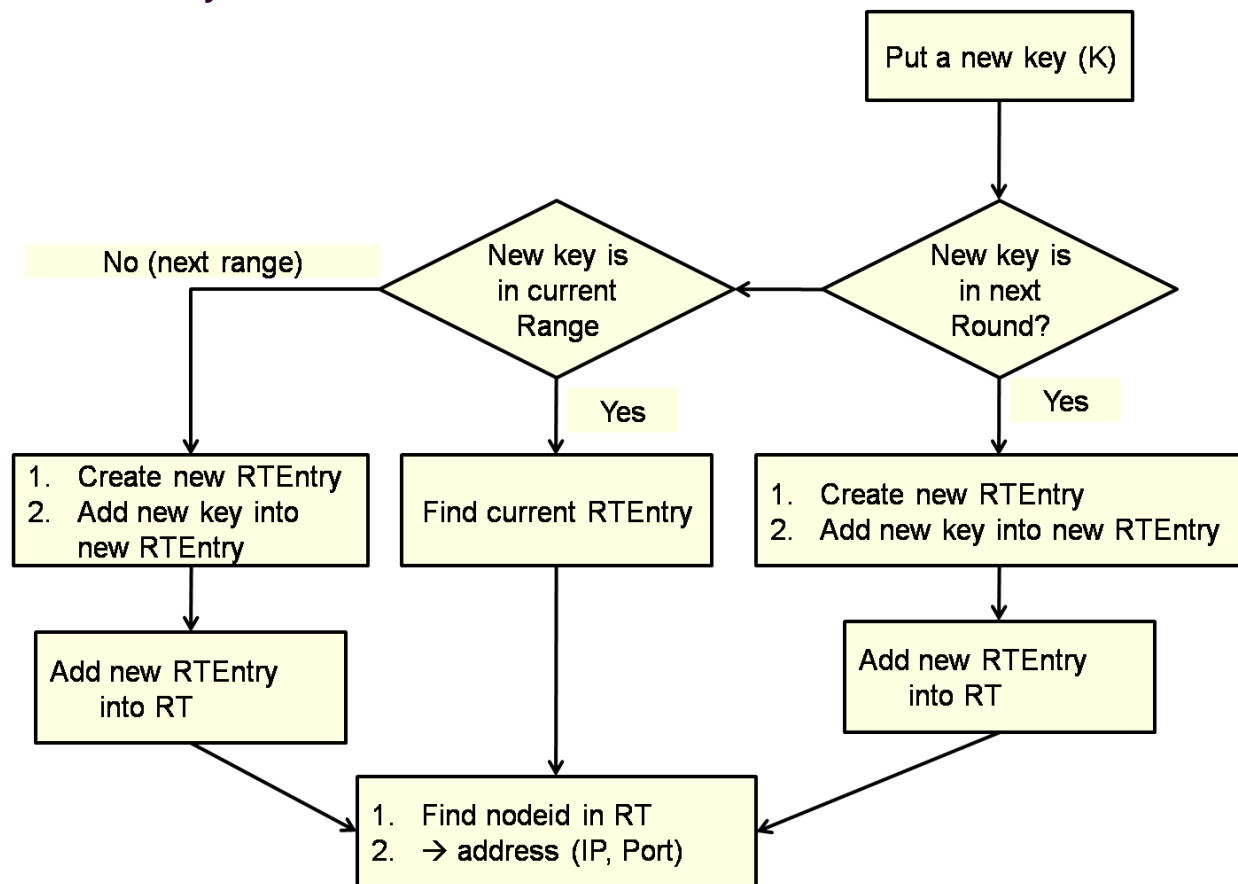


### Operations with adaptive partitioner:

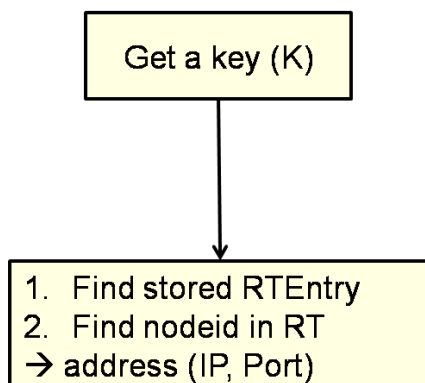
**Add/Remove nodes:** We have a list to store all nodes. Add/Remove nodes at the end of the list.



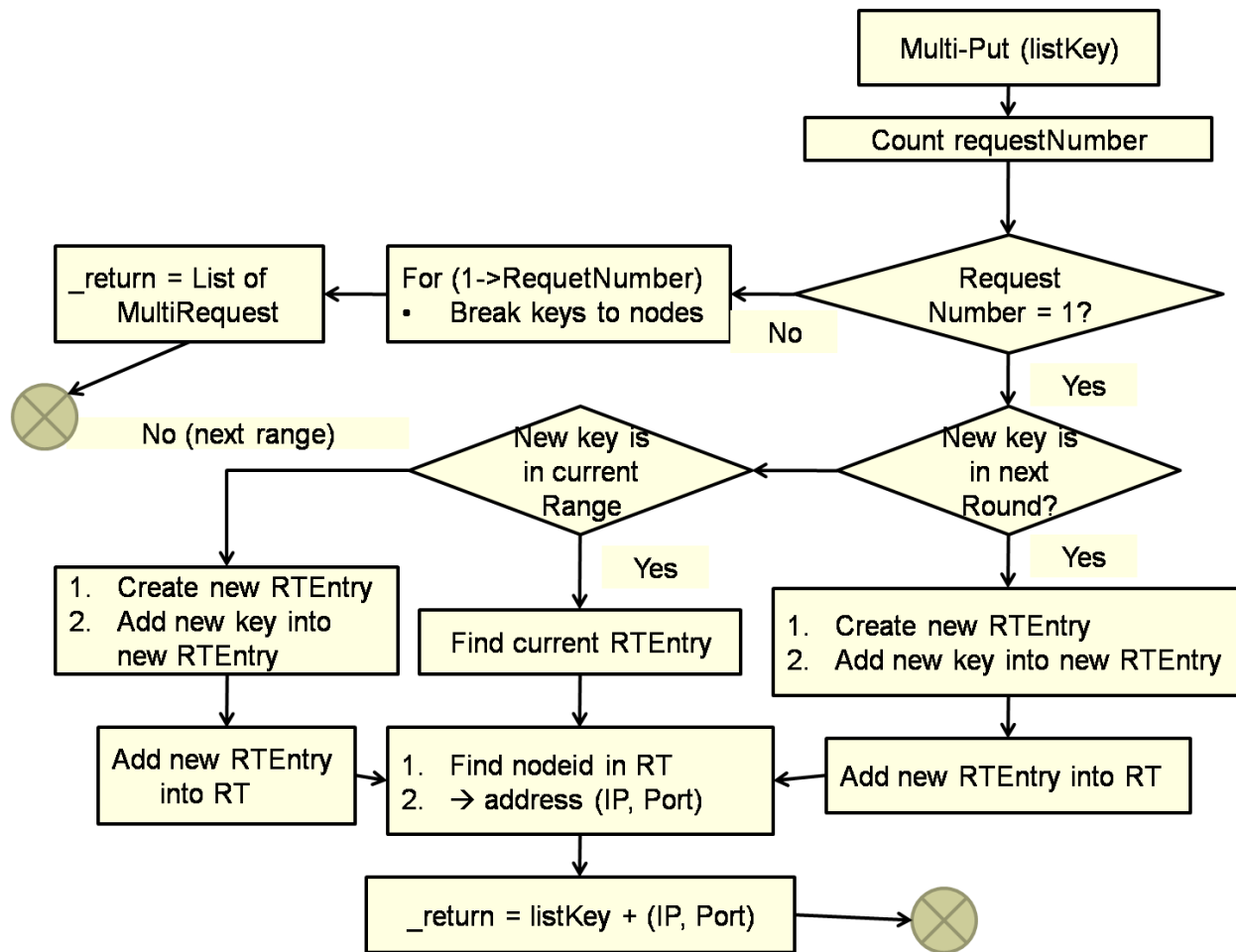
### Put a new key:



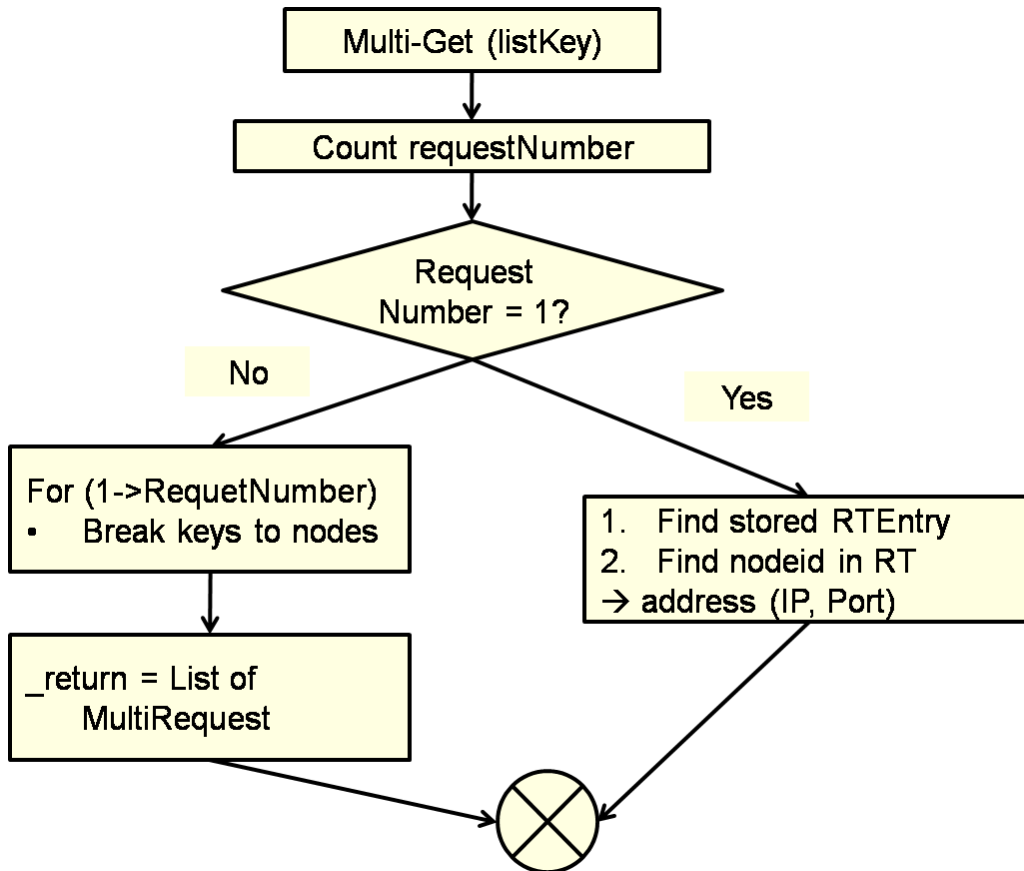
### Get a key-value:



### Multi-put:



### Multi-get:



The results of adaptive partition will be discussed in the experiment part.

## 4.4 Elasticity

In ZStore, the process of scaling out or scaling up does not affect availability of the system. In another word, the responds for request of accessing data are not interrupted when a node temporarily stop to add more resources or a node get into//remove from the system.



## 5. Availability

### 5.1 Hinted handoff

#### Analysis

How to ensure system durable in failure situation, that is, put and get requests will not failed due to temporary node or network failure? Traditional approach using master-slave or backup node may leaves a *dead time*, while healthy node begin switch to take role of dead node request will be lost.

#### Our approach

Previous studies from Amazon Dynamo[1] presents efficiency of hinted handoff technique in handling short time outage. That is a healthy node will cover requests for its temporary failed neighbor node. When the neighbor node is up, all the “hinted” data are “handoff” from the healthy node to neighbor node. In our system, we define four state of hinted handoff module when failure happens to handle put and get requests.

#### System state

State	Name	Action taken
1	Standard strategy.	Put to local storage. Replicate if needed.
2	HH on.	Put to buffer. Put to local storage. Replicate if needed.
3	HH on.	Put to buffer. Replicate if needed.
4	HH off.	Flush buffer. Replicate if needed. Switch to state 1 when done.

#### Changing state

States are switched automatically depends on events which happen.

- Default system state is 1 when system is operating normally.
- State from 2 to 4 only active when failure happens.
- State 4 is switched to state 1 when it flushes all buffer.

#### Implementation details

We have done implementation *one node failure scenario*, these events as follow:

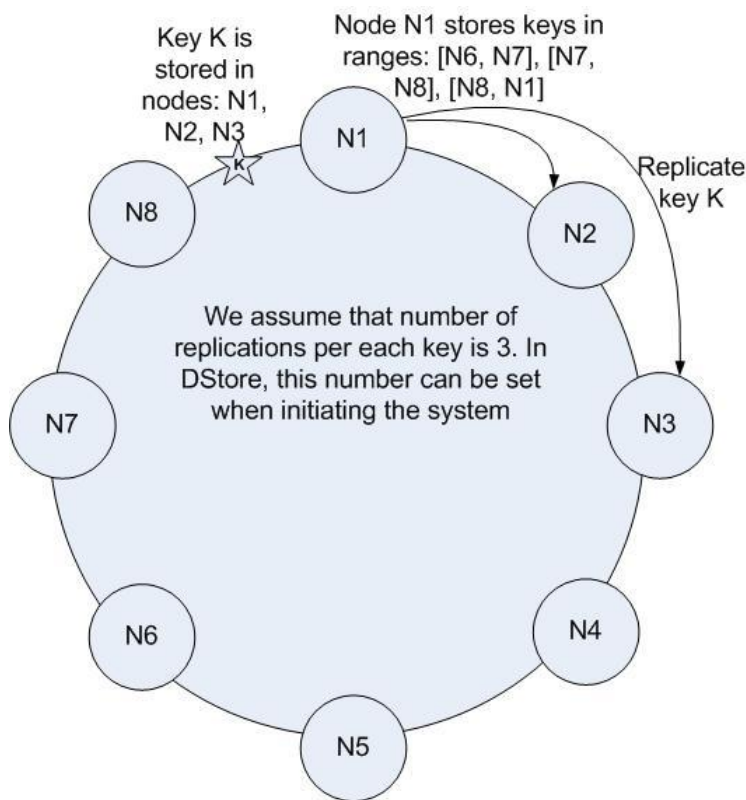
- Gossip service is notified of failure node.
- Proxy is notified that there is a failed node.
- Proxy remove the failed node from system.
- **Hinted handoff** is triggered.
- Failed node is up and rejoined the system.
- Proxy add the newly joined node to the system.
- Data from from buffer is transferred to newly joined node.

**Hinted handoff** is deactivated.

## 6. Reliability

### 6.1 Replication

We replicate keys on multiple nodes to achieve reliability. The below figure denotes how we combine replication scheme with consistent hashing model based on the design of Dynamo system.



In this figure, we consider the keys in range [N8, N1] which are stored in 3 consecutive nodes N1, N2, N3. N1 is in charge of “master node” for keys in this range. The term “master” indicates node N1 will be the first node responding for the requests of accessing to keys in the range. If the requests are adding a new key, N1 will replicate this key to next nodes N2, N3 called “slave” node. This “master” node contains a list of “slave” nodes. It is easy to see that each node in the network plays the role of both of “master” node for a range of keys and a “slave” node for another range. This guarantees the load balance between each node.

## 6.2 Fault tolerance

Fault tolerance is vital in large scale system. That is because there is a high rate probability that less reliable components from the system can be down. Many kinds of failure can lead to this problem; for example: list something. In ZStore system, fault tolerance is automatic. When Gossip service detect any fault node, it notifies this information to a proxy and then the proxy handles the process of fail-over. In this case, proxy executes 3 following steps in order:

1. Remove the fault node out of the system by changing its routing table and then synchronize to other proxies.
2. Change the lists of “slave” nodes in the relevant nodes; replace the fault node in these lists by other node.
3. Change the state of hinted handoff in the “temporary” node standing at the next position of the fault node by clockwise. Within this state, the “temporary” node creates a buffer to store all of keys falling into it. Thus, the “temporary” node play the same role with the fault. When the fault node is resumed, it automatically requests the “temporary node” to get these keys.
4. When the fault node is ready again, we only need to run this node in resume mode in order to make it get into the system. The process of resuming a node performs in 5 steps.

### Hinted handoff: handling temporary failure

We make use of hinted handoff as described in section 5 to ensure put/get handling when temporary failure happens.

### Merkle tree: handling permanent failure

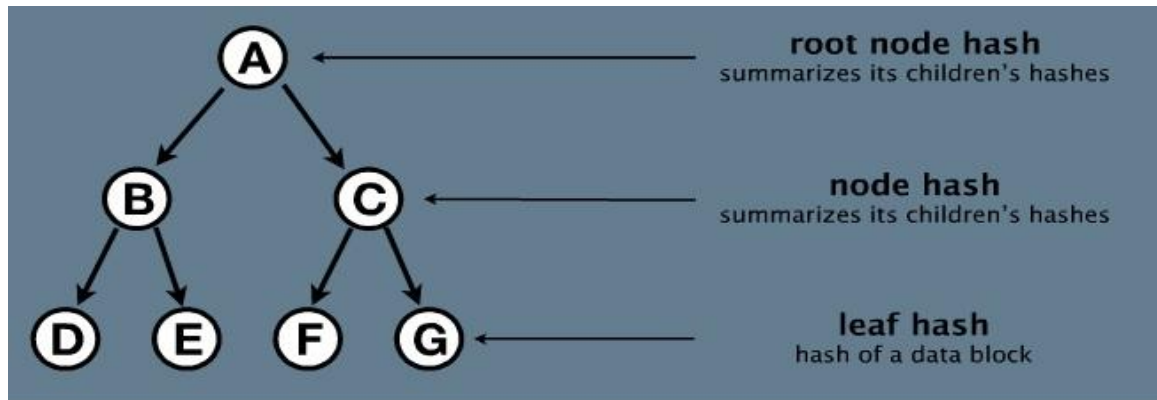
#### Analysis

Hinted handoff works best if the node failures are transient. If the hinted replicas become unavailable before they can be returned to the original replica node, it's very difficult and time consuming to detect and repair the difference of data between two nodes. To handle this and other threats to durability, we use an replica synchronization method to keep the replicas synchronized.

#### Our approach

To detect the inconsistencies between replicas faster and to minimize the amount of transferred data, we use Merkle trees. A Merkle tree is a complete binary tree with a k bit value associated to each node such that each interior node value is a one-way function of the node values of its children. In our system, each value of a (key, value) pair will be hashed to a k bit value, it's value of leaf in the Merkle Tree. And each inner node is a hash of it's respective children.

Each data node in our system has a Merkle Tree for each key range (the set of keys hold by it's replica nodes) it hosts. So, to compare the data in two node, we only compare two Merkle Tree, it will reduce the amount of data that needs to be transferred while checking for the inconsistencies among replicas. If the hash values of the root of two trees are equal, then the values of the leaf nodes are also equal (the data in two data node are same) and these data nodes require no synchronization. In contrast, it indicates that the data of some replica are different.



Merkle tree creation

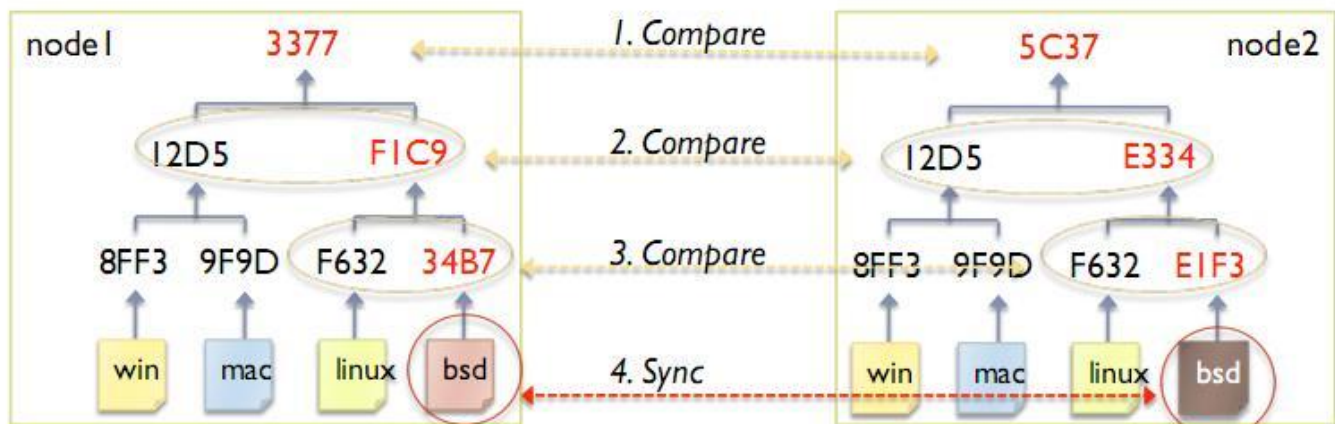
## Operations

### - Make tree:

To make tree, we use a cryptographic hash function SHA-256 to hash the value of all keys. This function guarantees there is no collision on a node. With a replica node, we hash all keys in a predefined key range. We combine the key and the hash value of value to a pair (key, hash value) and send all of pairs to master nodes in a predefined order. We only send the data of (key, hash value) to reduce the transferred data.

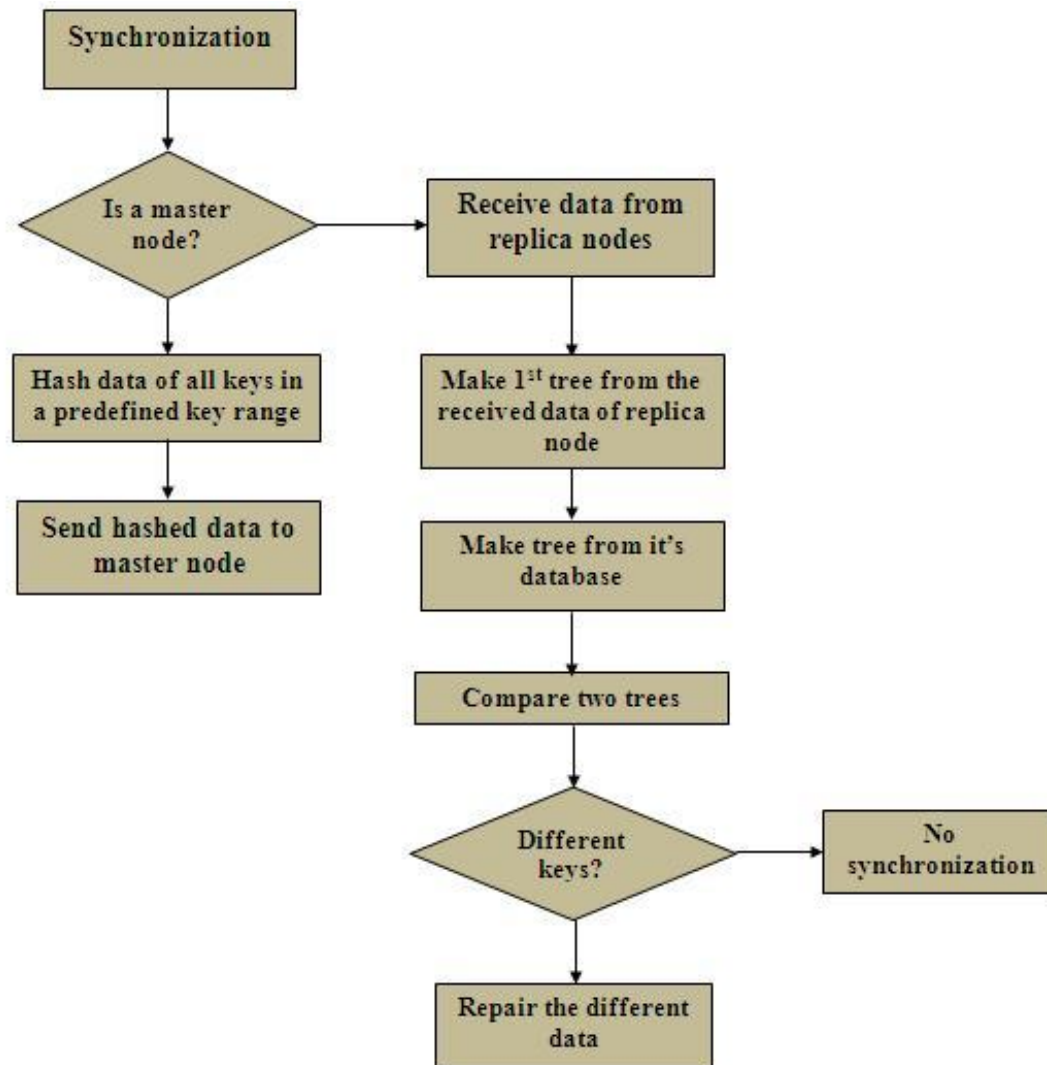
### - Compare tree:

When a master node received the comparison data from a replica node, it build the tree from the received (key, hash value) pairs and extract them to a key list. From the extracted key list, the master node read data from it's BDB database. If the key doesn't exist, it will be added to the different keys list of two nodes. If not, the value of this key will be hashed by SHA-256 function. Subsequently, the master node also build it's Merkle tree. Then, it compares two tree to find the different data between two nodes. If the hash values of the root of two trees are equal, then the data in two nodes are equal and the nodes require no synchronization. If not, we use the tree traversal algorithm to determine the different node between two tree and the different data between two nodes. Then, our system automatically repair these differences by putting the correct data from master node to replica node.



Comparison of hierarchical checksums in Merkle trees

## Implementation flows



## Conclusion

We only run Merkle tree for recovering data from permanent failure when a hinted replica fails or when we want to detect the consistency in our entire system. So, we don't need save the Merkle Tree of a node, it saves the memory of system. Moreover, it's final step in replication process so it will gurantee the eventual consistency of data in our system.

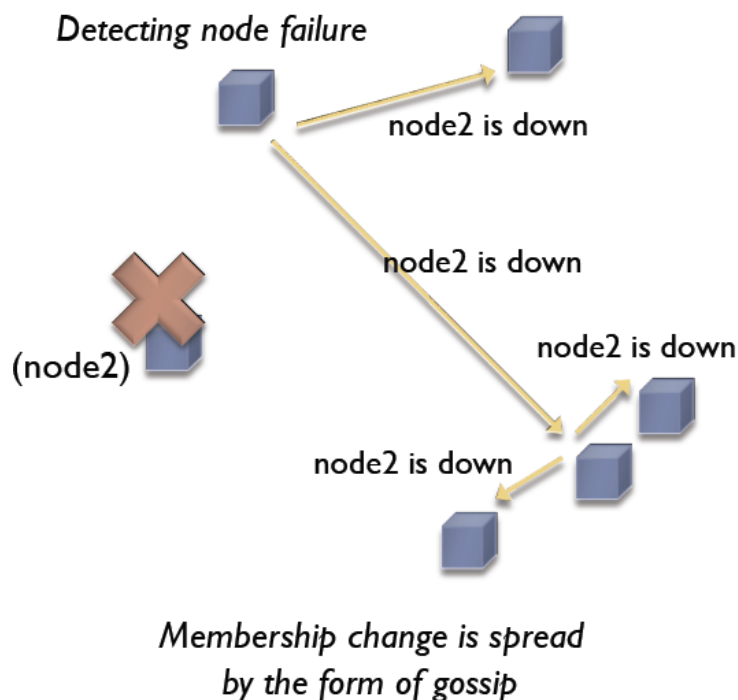
## 6.3 Failure Detection

### Analysis

Failure detection is used to prevent communication between client and unreachable nodes. When failure happens, all nodes are notified by a distributed dissemination protocol and node are being handle put/get request will trigger hinted handoff [section \_]. By doing this, we **guarantee fault-tolerance of the system**.

### Our approach

To date, various methods have been developed to detect failure such as centralized monitoring, distributed monitoring. One major drawback of the first approach is slow detection time when dealing with large number of nodes. We investigated in second approach by using gossip protocol to **quickly detect irresponsible nodes in cluster**, that is, liveness status of nodes are continuously propagated from node to node.



To integrate gossip with our system, each data node service is *attached* with a gossip service, and gossip services rumoring about data node service health status.

### Gossip-based protocol


Membership and liveness information are packed to message, then message is spread like a rumor. We exchange membership from node to node in every second and update membership whenever a node join/leave system.

Message format	Message reconciliation
<ul style="list-style-type: none"> <li><b>Gossiper unique ID.</b></li> </ul>	<ol style="list-style-type: none"> <li>Matching Gossip unique ID.</li> </ol>

<ul style="list-style-type: none"> <li>• <b>Array of [</b>  IP:Port<sup>1</sup>  Heartbeat version<sup>2</sup>  Liveliness of node<sup>3</sup>  <b>]</b></li> </ul>	<ol style="list-style-type: none"> <li>2. Compare heartbeat version between local and remote.</li> <li>3. If (version is outdated) <ul style="list-style-type: none"> <li>◦ “Merge” local heartbeat by remote heartbeat.</li> </ul> </li> <li>4. Else <ul style="list-style-type: none"> <li>◦ Propagate local heartbeat to remote.</li> </ul> </li> </ol>
---	--

Example:

A: 398725, live, 100 => ID = 398725, live, Heartbeat = 100

<b>A: 398725, live, 100</b> B: 462245, live, 200 C: 935872, live, 300		<b>D: 987521, live, 100</b> B: 462245, dead, 201 C: 935872, live, 300
A: 398725, live, 100 B: 462245, dead, 201 C: 935872, live, 300		<b>D: 987521, live, 100</b> B: 462245, dead, 201 C: 935872, live, 300
	...	
<b>A: 398725, live, 100</b> B: 462245, live, 202 C: 935872, dead, 301		<b>D: 987521, live, 100</b> B: 462245, dead, 201 C: 935872, live, 300
<b>A: 398725, live, 100</b> B: 462245, live, 202 C: 935872, dead, 301		<b>D: 987521, live, 100</b> B: 462245, live, 202 C: 935872, dead, 301

## Group Membership

We take care of two main action related to node membership in our system.

### 1. Join

A node join the system by sending MSG\_JOIN to a bootstrapper, which can be any active node already in the system. The bootstrapper will:

- Update cluster information to newly joined node.
- Propagate presence of new node to other node by gossiping.

### 2. Leave

<sup>1</sup>IP and Port which gossip service operates on.

<sup>2</sup>How many times node has been up?

<sup>3</sup>Identify node dead or alive?



- *Momentary leave*: in the case of software/hardware failure: Once an active node is unable to exchange message with failed node, this failed node is marked as *dead*. Then status of failing node will be propagated by gossiping.
- *Officially leave*: in the case of system maintenance: A node leave the system officially by sending MSG\_LEAVE to one of its active peer.

## Conclusion

Our approach has **two main advantages**, that is:

- Robust: No one can prevent message from spread.
- Exponentially rapid spread, near  $\log(n)$  time failure detection, where  $n$  is number of node.

In the future, we would apply gossip to build the following service.

- Configuration management: *quickly update configuration whenever it changes*.

Code distribution: *quickly update new version of code to entire system*.

## 7. Evaluation

### 7.1 Partitioning

#### Adaptive Partition

##### Speed experiments

**Experiment 1:** Compare multi-request with variable numbers of items and single-request.

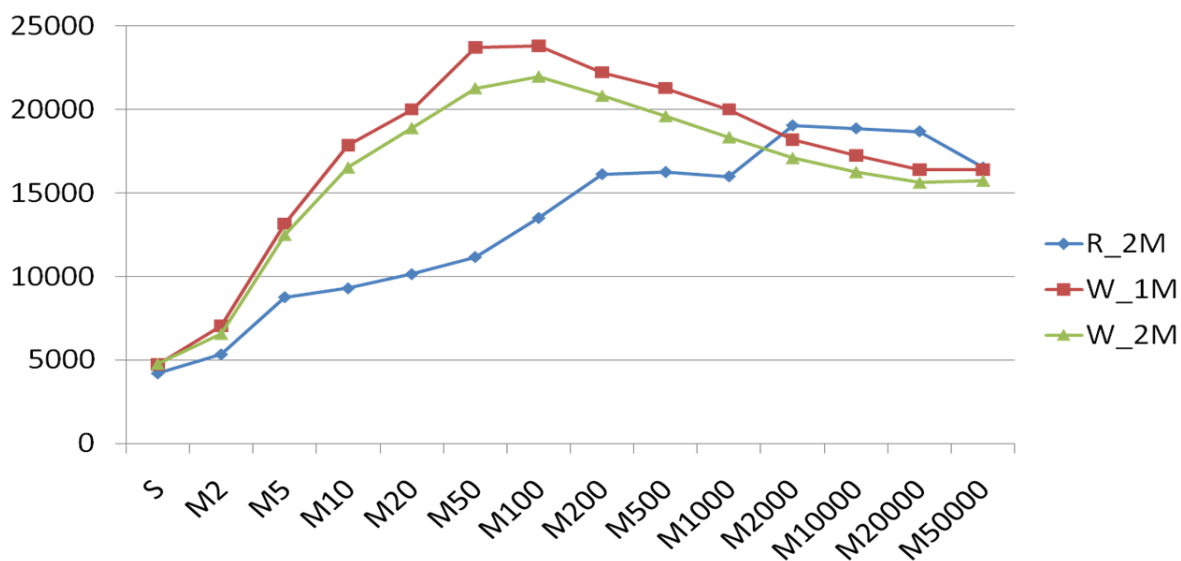
The system has:

- 3 database servers and 1 proxy in one physical machine.
- Theta is 100.000

We put into the system 2.600.000 Key-Values (the size of the routing table is 26, its height is 3), in which key type is a small string and value type is small object (~10 bytes).

In the following figure:

- R\_2M: average speed in getting the first 1.000.000 first key-values.
- W\_1M: average speed in putting the first 1.000.000 first key-values.
- W\_2M: average speed in putting the first 2.000.000 first key-values
- S: Single request (one key-value at a time)
- M\_n: Multi-request with n pair of key-values at a time



### Conclusion:

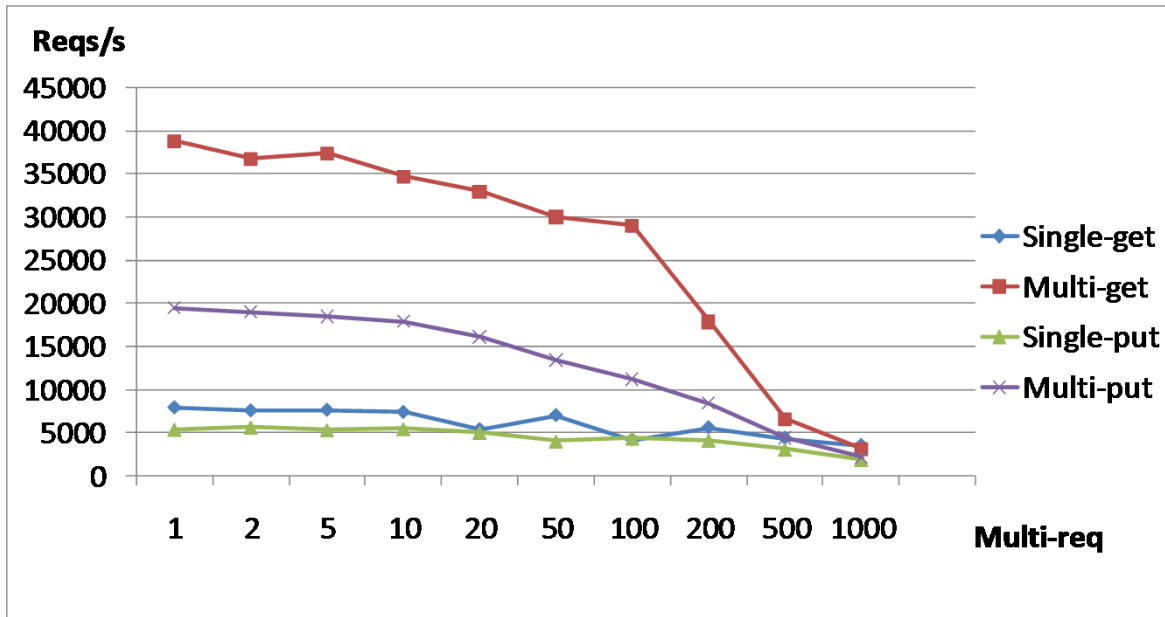
- Multi-request is much more faster than single request (about 4.5 times in case of small size of key-value and multi-request a hundred of items at a time).
- The line W\_1M and W\_2M show that multi-put with a number of items in the range 10-20.000 is good if the size of the value is small (1-10 bytes).
- The line R\_2M show that multi-get with a number of items in the range 200-20.000 is good.
- We therefore reach a conclusion that in the system in which read is much more than write, multi-request with a number of items in the range **200-2000** is good if the size of the value is small.
- **Single request is also faster** than consistent hashing model of Dynamo because we do not need to hash the key. (5000 reqs/s of AP and 3000 reqs/s of CH).
- *W\_2M is parallel with but lower than the line W\_1M because the size of its routing table is larger than the other line's size.*
- *R\_2M is lower than W\_2M because we have 3 database servers in only one physical machine.*

### Experiment 2: Multi-request with variable big value, fixed numbers of items (10)

The system has:

- 1 database servers and 1 proxy in one physical machine.
- Theta is 100.000

We put into the system 100.000 Key-Values (the size of the routing table is only 1), in which key type is a small string and value type is variable large object (1KB-1MB).



### Conclusion:

- The larger in size of the value, the worse in speed of multi-request. However, multi-request is still better than single request if the size of the value is smaller or equal to 500KB.
- Especially, multi-get is much faster (about 3-4 times) than single-get if the value's size is smaller or equal to 100KB.
- *The get lines is higher than put lines because we only have one database server in one physical machine.*

### Experiment 3: Multi-request with fixed size of value = 1KB, variable numbers of items

The system has:

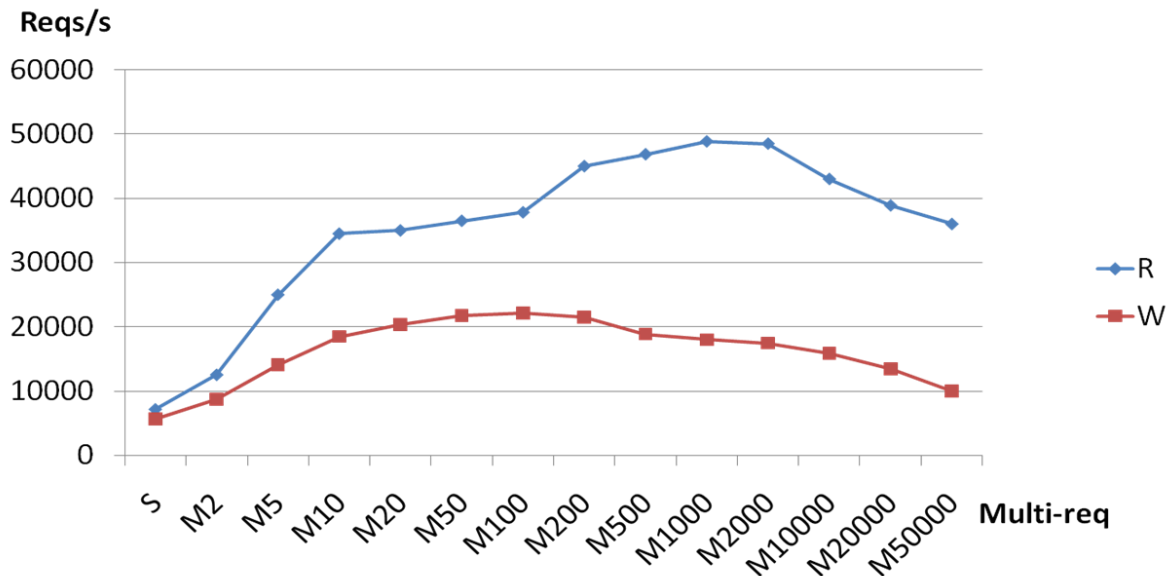
- 1 database servers and 1 proxy in one physical machine.
- Theta is 100.000

We put into the system 400.000 Key-Values (the size of the routing table is 4), in which key type is a small string and value type is a fixed size object (1KB). We change the number of items in a multi-request to get the best suitable number.

In the following figure:

- W: speed of multi-put
- R: speed of multi-get
- S: speed of single request
- M<sub>n</sub>: multi-request with n number of items. (the size of a multi-request is therefore

n KB).



#### Conclusion:

- The line R show that multi-get in the range **200-2000** is good, which is perfect match the case of small size of value (1-10 bytes) in the Experiment 1.
- The size of one multi-request may reach **2-20MB**, but it is still much better than single request (2.5 times).
- This chart has the same characteristic with the first chart in the Experiment 1: ***“multi-request is still much better than single-request in the case of the number of items in a multi-request is small (1-100 items). Especially, with the number of items is in range 1-10, the speed line increases sharply”***.

#### Experiment 4: Compare multi-get in only one node with multi-request in many nodes

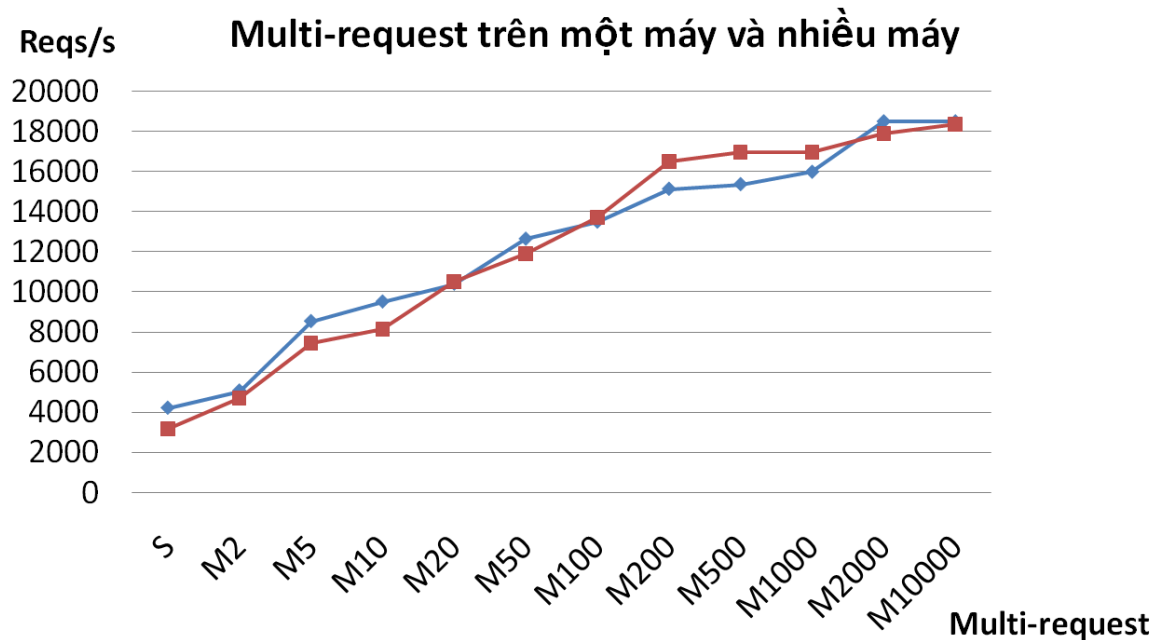
The system 1 has:

- 4 database servers and 1 proxy in one physical machine.
- Theta is 100.000

The system 2 has:

- 16 database servers and 4 client and 4 proxies in 4 physical machines.
- Theta is 100.000

We put into the system 1.100.000 Key-Values (the size of the routing table is 11), in which key type is a small string and value type is a small object (1-10 bytes).



**Conclusion:**

- With one client, multi-get in only one machine is like multi-get in many machines.
- We also test the system in case of adding more proxy and client (4 proxies and 4 clients), the ability of the system proved to increase **linearly or horizontally**.

**Experiment 5: Multi-request with discrete keys**

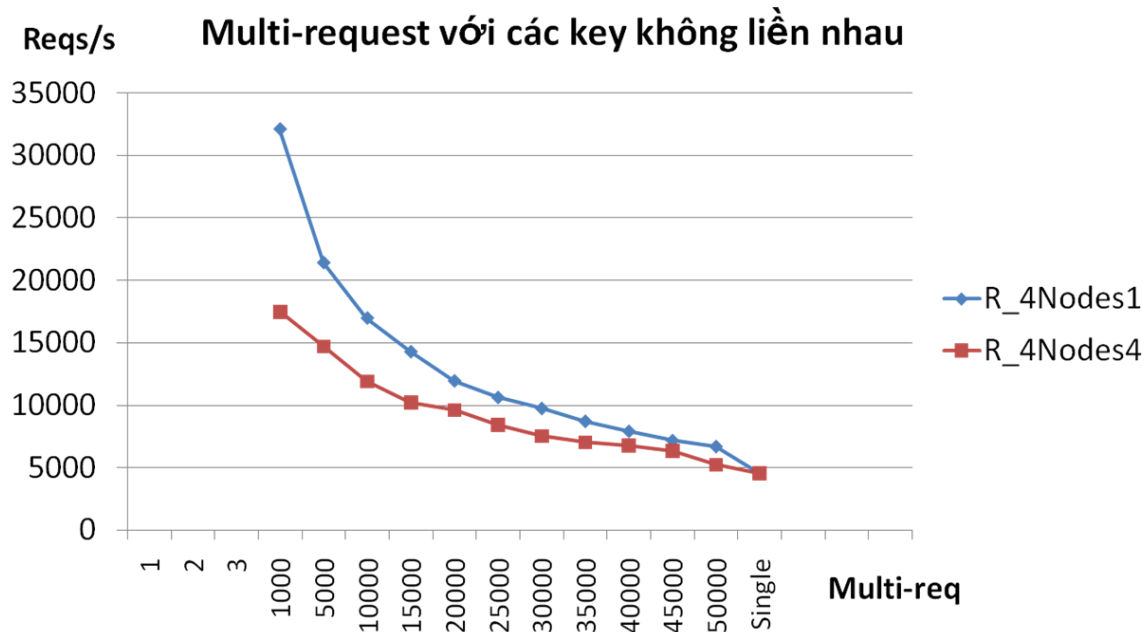
The system 1 has:

- 4 database servers and 1 proxy in one or four physical machines.
- Theta is 100.000
- Multi-get with the number of items is 10.

We put into the system 2.000.000 Key-Values (the size of the routing table is 20), in which key type is a small string and value type is a small object (1-10 bytes).

In the following figure:

- R\_4Nodes1: 4 database servers in 1 physical machines.
- R\_4Nodes2: 4 database servers in 4 physical machines.
- 1,2,3....50.000: are the gaps between two successive keys in a multi-get.



#### Conclusion:

- Both lines are going down because the larger gaps between two successive keys, the more database servers need to be get from.
- If the gaps are between 1000-5000, most of multi-gets are returned from only one database server. The speed is therefore very fast.
- If the gaps are between 10.000-50.000, multi-gets are returned from 2-6 database servers. (smaller than 10 in case of using multi-request with consistent hashing).
- In all the cases, multi-requests is still better than single-request, especially if multi-requests are returned from **1-4 database servers**.

## 7.2 Hinted Handoff

We observed hinted handoff technique performs perfectly in that case, the only drawback is the time from node failure until when proxy is aware of node failure. Note that, the failure detection time is depends on our gossip protocol. In the future, we would optimize gossip protocol to minimize detection time.

## 7.3 Merkle Tree

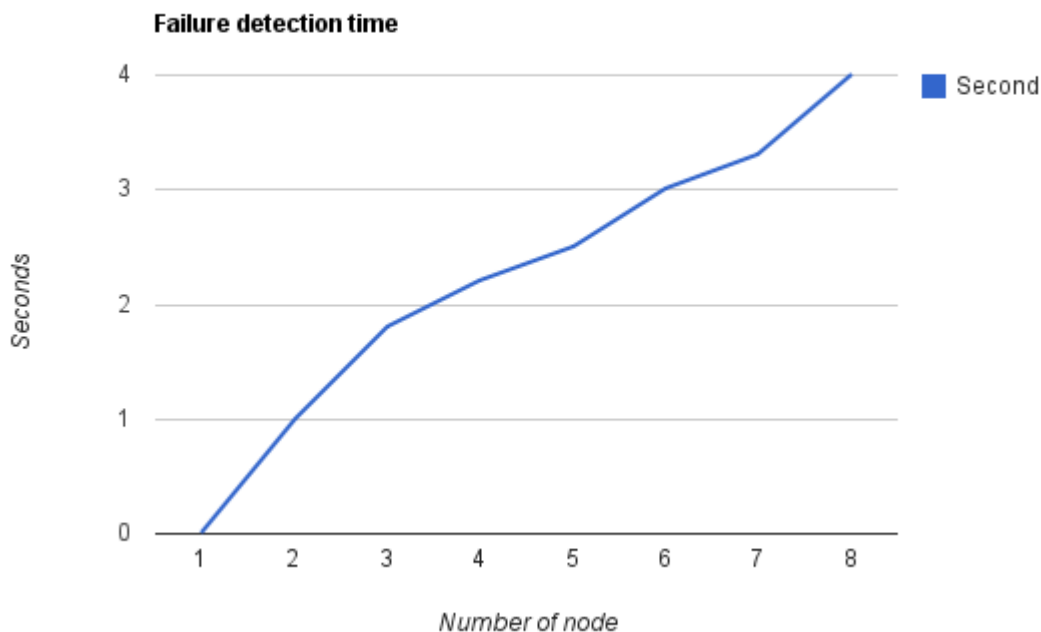
We have tested our system in two test case:

1. One node has no key and another node has a number of keys in the same key range
2. Two node have different keys and different value of keys in the same key range

With the two test case above, our system had indicated the difference of data between two nodes. The transferred data don't influence to the system bandwidth and the time to detect the difference between two nodes is fast. The time to compare and repair the difference of data of 1000000 key is about 4-5 minutes.

## 7.4 Failure Detection

Since mixed with random selection, the gossip protocol we implemented promises propagate time nearly  $\log(n)$ , where  $n$  is cluster size. We have tested on cluster of 8 nodes, the delayed time to propagate a node failure to entire nodes in cluster is about 4s.





## 8. Conclusion

In this paper, we have described ZStore , a scalable and highly available distributed key-value storage system, which will be used for storing data of many services in Zing social network. ZStore proves its incremental scalability and allows us to scale out depends on system's load. We have observed ZStore's ability to successfully handle two failure scenario: temporary failure and permanent failure.

We have designed and implemented two new partition strategies named Golden Ratio and Adaptive Partition. The first strategy can be used to replace Consistent Hashing model of Dynamo. In the second strategy, we not only improved speed of single request significantly than Dynamo's models but also exploit multi-request which is much faster than single request.

In the future, we would test as well as improve our system in several components. Firstly, testing new partition strategies with real applications, especially load balancing feature. Secondly, to improve failure detection ability by using Phi Accrual Failure Detection to adapt failure detection level for each service. We would like to hear your comment and suggestion to our work.

## References

- [1] Dynamo: amazon's highly available key-value store, G DeCandia 2007*
- [2] Skip Graphs, J Aspnes 2007*
- [3] Simple efficient load-balancing algorithms for peer-to-peer systems, DR Karger, DR Karger 2006*
- [4] Simple-load-balancing for distributed hash tables, J Byers, 2003*
- [5] Load balancing in structured P2P systems, A Rao, 2003*
- [6] Distributed Balanced Tables, Not making a hash of it All, P Ganesan 2003*
- [7] Brewer's Conjecture and Feasibility of Consistent, Available, Partition-Tolerance Web Services 2000*
- [8] Consistent Hashing and Random Trees: Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web, David Karger 1997*