



VAASAN AMMATTIKORKEAKOULU
UNIVERSITY OF APPLIED SCIENCES

Yang Zhou

Development of Distributed Cache Strategy
for Analytic Cluster in an Internet of Things
System

Technology and Communication
2016

FOREWORD

This is my final undergraduate thesis in the Degree Programme in Information Technology, at Vaasa University of Applied Sciences, Vaasan Ammattikorkeakoulu.

Sincerely I want to thank my thesis supervisor, Dr. Yang Liu, for his various help and great guidance both in the fields of academic research and thesis writing. Benefiting from his rich professional knowledge and conscientious mentoring I have built up scientific researching skills and finally finished the writing of this thesis. Furthermore, I appreciate greatly his support and inspiration during my study period, which boosted my confidence and efficiency.

Moreover, I would like to express my great appreciation to my mentors in the industry, Tuomas Ritola and Mathias Grädler, for their great, effective and various instructions and support for achieving the final successful result in an engineering perspective. Also I am grateful to my company Wapice Ltd for providing me this valuable and interesting thesis topic.

Besides, I am also very thankful to other professors and staffs in Vaasan Ammattikorkeakoulu, including Mr. Santiago Chavez, Dr. Menani Smail, Dr. Ghodrat Moghadampour, Mr. Jani Ahvonen, Dr. Chao Gao, Mr. Antti Virtanen and all other teaches and staffs who have kindly helped and guided me with their wisdom and knowledge during my undergraduate study, which forms the foundation for this thesis work, and my further studies and career.

Finally, I want to express my biggest gratitude to my family and friends for their understanding and generous support. They always are my strongest backup when I am pursuing my dream.

Yang Zhou
Vaasa, Finland
5/2/2016

VAASAN AMMATTIKORKEAKOULU
UNIVERSITY OF APPLIED SCIENCES
Degree Program in Information Technology

ABSTRACT

Author	Yang Zhou
Title	Development of Distributed Cache Strategy for Analytic Cluster in an Internet of Things System
Year	2016
Language	English
Pages	84 + Appendices
Name of Supervisor	Yang Liu

This thesis discusses the development of a distributed cache strategy for an analytic cluster in an IoT system. In this thesis, LRU and Proactive Cache and essential distributed system related concepts are discussed. The study about the approaches for performance optimization, nodes and data distributing in the IoT system are also included.

In the IoT system, the cluster for data analysis involves large volume of data and some specific processes such as streaming processing raises a need for real-time data querying, which brings a heavy load to the data center for catering the request. Besides for expanding the capability of the data center, a well-organized cache layer is required, which will noticeably enhance the performance and scalability of the data center.

This thesis mainly contributes to the development of a cache strategy based on the distributed system. In the cache strategy session, LRU and Proactive Cache, and the optimizing of them for the IoT system are discussed. Next, a set of topology model of nodes is introduced and compared. After that, the algorithm of data partitioning is discussed and examined. Redis is used as the main database in the study, Apache Kafka is involved for enabling Proactive Cache and Spring Boot is adopted as the platform for the implementation of algorithms and business logic included in this thesis. The result of this thesis is tested using local development machine and the development server in Wapice Ltd.

It can be concluded that the distributed cache strategy developed in this thesis is functional and well-extendable. It fulfills the need of high performance data querying from analytic cluster and reduces the load from data center in the IoT system.

Keywords	Caching, distributed system, IoT, data partitioning, cluster topology
----------	---

CONTENTS

FOREWORD

ABSTRACT

1. INTRODUCTION	11
1.1 Purpose.....	11
1.2 Overall structure of this thesis	11
1.3 Background of IoT Ticket and Wapice Ltd	13
1.3.1 Wapice Ltd	13
1.3.2 IoT-Ticket	13
1.4 Introduction of Data Cache	15
1.5 Redis	16
1.6 Apache Kafka.....	17
1.7 Introduction of Microservice	18
1.8 Spring Boot	19
2. DATA MODEL.....	21
2.1 Data model in target IoT system.....	21
2.2 Data model in cache.....	22
2.2.1 Fundamental data structure	22
2.2.2 Key-Value based data structure	25
3. CACHE STRATEGY.....	28
3.1 LRU in Redis	28
3.1.1 Approximated LRU algorithm	29
3.1.2 The configuration for Approximated LRU in Redis	32
3.2 Optimizing for LRU in Redis	34
3.2.1 TTL	34
3.2.2 Placeholder	34
3.3 Proactive Caching	39
3.3.1 Zookeeper and Kafka	41
3.3.2 Simulated data source	42
3.3.3 Consumer in Kafka	45
3.4 Aggregation.....	47
4. CLUSTER TOPOLOGY MODEL.....	48
4.1 Master – Master Model.....	48

4.2	Master – Slave Model	49
4.3	Master – Master – Slave Model	50
4.4	Implementation	52
4.4.1	Sub-cluster of master nodes	52
4.4.2	Sub-cluster of Master - Slave	53
5.	DATA PARTITIONING ALGORITHM	57
5.1	Simple hash solution	58
5.2	Consistent Hashing solution	59
5.2.1	Hash Ring	59
5.2.2	Mapping of objects	60
5.2.3	Mapping of nodes	60
5.2.4	Adding and removing of nodes	62
5.2.5	Virtual node	64
5.3	Pseudo-code of Consistent Hashing	65
6.	MICROSERVICE	68
6.1	Configuration service	68
6.2	Discovery Service	69
6.3	Cache Service	69
7.	TESTING AND ANALYSIS	70
7.1	Cache strategy test	70
7.2	Master sub-cluster test	73
7.3	Slave node test	74
7.3.1	Insert to master and read from slave test	76
7.3.2	Consistency test	77
7.3.3	Consistency test 2	78
8.	DISCUSSION AND FURTHER DEVELOPMENT	80
8.1	The failure tolerance in the cluster	80
8.2	Grouping cache data by different industry	80
8.3	Test in real distributed environment	81
9.	SUMMARY	82
	REFERENCES	83

APPENDICES

LIST OF ABBREVIATIONS

IoT	Internet of Thing
Ltd	Limited
LRU	Least Recent Used
WRM	Wapice Remote Management
MRU	Most Recently Used
PLRU	Pseudo-LRU
RR	Random Replacement (RR)
SLRU	Segmented LRU
SQL	Structured Query Language
NoSQL	Not only SQL
MID	Manufacturer Identification
TTL	Time To Live
JSON	JavaScript Object Notation
DI	Dependency Injection
SYNC	Synchronize
ASYNC	Asynchronous
DHT	Distributed Hash Table
MIT	Massachusetts Institute of Technology

P2P	Peer-to-peer
CARP	Cache Array Routing Protocol
WWW	World Wide Web
IP	Internet Protocol
REST	Representational State Transfer
ms	Millisecond

LIST OF FIGURES AND TABLES

Figure 1.	IoT-Ticket, overview of communication(monitored)/2/	p. 14
Figure 2.	IoT-Ticket, overview of communication(controlling)/2/	p. 14
Figure 3.	IoT-Ticket Analytics screenshots/2/	p. 14
Figure 4.	Redis major data structure /6/	p. 16
Figure 5.	Model of Apache Kafka /7/	p. 17
Figure 6.	Anatomy of a Topic of Apache Kafka /7/	p. 18
Figure 7.	An example of microservice architecture /10/	p. 19
Figure 8.	An example of microservice based of Spring Boot /12/	p. 20
Figure 9.	Lists data type in Redis /14/	p. 24
Figure 10.	Sets data type in Redis /14/	p. 24
Figure 11.	Hashes data type in Redis /14/	p. 25
Figure 12.	Data structure design in cache	p. 27
Figure 13.	Traditional LRU caching schema/15/	p. 28
Figure 14.	Flow chart of sampling in Approximated LRU	p. 30
Figure 15.	Graphical comparison of Approximated LRU and LRU /16/p.	31
Figure 16.	Inconsecutive data generating	p. 34
Figure 17.	Flowchart of “Placeholder” optimizing	p. 37
Figure 18.	Traditional cache pattern /17/	p. 40

Figure 19.	Proactive Caching pattern /17/	p. 40
Figure 20.	Zookeeper service is running	p. 42
Figure 21.	Kafka service is running upon the Zookeeper	p. 42
Figure 22.	List of created topics in Kafka	p. 43
Figure 23.	Aggregation of cache strategy	p. 47
Figure 24.	Master – Master topology mode/14/	p. 48
Figure 25.	Master – Slave topology mode /14/	p. 50
Figure 26.	Master – Master – Slave topology model	p. 51
Figure 27.	Hash Ring	p. 59
Figure 28.	Objects are mapped in the Hash Ring	p. 60
Figure 29.	Nodes and Objects are mapped in the same Hash Ring	p. 61
Figure 30.	Each object is store in node by Consistent Hashing	p. 62
Figure 31.	One node is removed from cluster	p. 63
Figure 32.	One node is added to cluster	p. 64
Figure 33.	Virtual node	p. 65
Figure 34.	Brief microservice system structure	p. 68
Figure 35.	Query without cache strategy deployment	p. 70
Figure 36.	Query with partial cache strategy deployment	p. 71
Figure 37.	Query with full cache strategy deployment	p. 71
Figure 38.	3 master nodes were up running in development machine	p. 73
Figure 39.	Print of the test for master sub-cluster	p. 74

Table 1.	ProcessData(simplified) data model	p. 21
Table 2.	Data type mapping in ProcessData(simplified) data model	p. 22
Table 3.	Time-based performance of Redis collection data types/13/	p. 23
Table 4.	Eviction policy list and description /16/	p. 32
Table 5.	Redis configuration options for slave node	p. 53
Table 6.	Collected data from the strategy test	p. 71
Table 7.	Average time consumption of strategy test	p. 72
Chart 1.	Collected from strategy test data in line chart	p. 72

LIST OF APPENDICES

APPENDIX 1. Java Implementation of Consistent Hashing

1. INTRODUCTION

1.1 Purpose

This thesis discusses the development of a distributed cache strategy for an analytic cluster in the IoT system. The objective of studied cache strategy is to provide a high performance, highly extendable and highly stable cache solution in the distributed system on IoT environment to reduce the load of center databases and enhance the data query performance. There are mainly five parts in the study of this thesis. In the first part, the data model and cache strategy, which are the main focus in this thesis, are discussed in detail. In the second part, the nodes topology model which indicates how the nodes are connected and how they collaborate is discussed. In the third part, a data partitioning algorithm which describes how the data is distributed into the cluster is introduced. In the fourth part, the micro-service based architecture which adopted in this thesis is depicted. In the final part, the core components are examined and the further development is discussed.

1.2 Overall structure of this thesis

This thesis are mainly divided into 9 chapters and ordered by the logical flow of the study. In chapter 1, the overall information of this thesis and the background of involved components are introduced. In chapter 2, the model of sensor data and cache data are described. In chapter 3, the cache strategy of LRU, Proactive Cache and the approaches for optimizing in IoT system are discussed. Chapter 4 focuses on the topology models of nodes. Chapter 5 discusses the data partitioning. In chapter 6, the microservice based architecture which is adopt by the study of this thesis is represented. Chapter 7 and 8 discuss about the result of test and the further development of this study. Chapter 9 summarizes up the content of this thesis.

The project in this thesis mainly consists of 5 parts: data model, cache strategy, cluster topology model, data partitioning algorithm and microservice.

1. Data model

Data model consists of two parts: model of sensor data and model of cache data. Since the cache discussed in this thesis is designed for the analytic cluster, not all fields from sensor data are needed to be cached, a converting process is required. Furthermore, for effective data storing and querying with Redis database, a dedicated data model is developed.

2. Cache strategy

Cache strategy is the primary topic in this thesis. Cache strategy contains two layers. The first layer is in the database server, in which the LRU solution is adopted. The second layer is in the cache service layer, in which Proactive Caching is deployed. Besides, the optimizing approaches for the IoT system are discussed.

3. Cluster topology model

Along with the development of algorithm, cluster topology model describes how the nodes are connected to assemble the cache cluster. In the cluster topology model, Master – Master, Master – Slave and Master – Master – Slave topology models are discussed.

4. Data partitioning algorithm

For distributing the data into the cache cluster, a discussion about the data partitioning algorithm is raised. Consistent Hashing algorithm which introduces a concept of Hashing Ring and maps both data and nodes in the same hash space is introduced in the data partitioning algorithm session.

5. Microservice

Microservice is a system architecture adopted in the study of this thesis in the cache service implementation by using Spring Boot. It brings great flexibility and scalability in the system level.

1.3 Background of IoT Ticket and Wapice Ltd

Wapice Ltd is the sponsor of this thesis work and IoT-Ticket is the target IoT system for which this thesis is developed.

1.3.1 Wapice Ltd

Wapice Ltd is a private and independent company. It was started in 1999 and the headquarters is in Vaasa, Finland. Now there are over 300 software specialists are working in Wapice, and the company is keeping growing. Wapice is based in Vaasa, and also has deployed the offices in Helsinki, Tampere, Seinäjoki, Hyvinkää, Oulu and Jyväskylä. /1/

Wapice Ltd is an AAA-rated company in which the software production is based on a certified ISO 9001:2008 quality management system and ISO 14001:2004 environmental management system. /1/

The three of Wapice major products: IoT-Ticket, Summium and EcoReaction have got the Key Flag granted from the organization of Finnish work. /1/

1.3.2 IoT-Ticket

- Background

IoT-Ticket is a modern Internet of Thing system. IoT-Ticket is developed by Wapice Ltd and received its name in beginning of 2015, but has been developed and promoted since 2005 under the name Wapice Remote Management platform (WRM platform). /2/

The development target of IoT-Ticket was and is to create a modern, easy to use, web-based platform which allows remote monitoring and control of assets with a powerful report creation and analytics possibilities taking advantage of a secure Big-data system. Enabling regulatory reporting, supervisory monitor-

ing & control, operational efficiency & KPI tracking and condition monitoring is the goal to be achieved. /2/

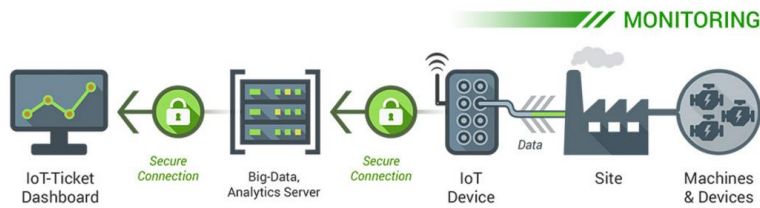


Figure 1. IoT-Ticket, overview of communication (monitoring) /2/



Figure 2. IoT-Ticket, overview of communication (controlling) /2/

- IoT-Ticket Analytics

IoT-Ticket Analytics supports users to analyze and dig value based on data collected over a long period of time. The web-based and easy to use Analytics package seamlessly integrates to the IoT-Ticket Dashboard.



Figure 3. IoT-Ticket Analytics screenshots /2/

1.4 Introduction of Data Cache

Cache is a generic term of the technologies which temporarily stores some specific data elsewhere from the main data center for enhancing the performance of data querying. The definition of the Cache is: In the computer system, the cache is a component as a temporary data storage for enhancing the performance in further work when the stored data is involved in. The data is stored in the cache can be the middle or final result of the previous work, or just the duplication of some pieces of data. /3/

When the requested data can be found in a cache, the cache hit happens. When the cache hit occurs, the stored data can be provided for the request directly avoiding recalculating or reading from a slower data store, which could significantly enhance the performance in further calculation. /3/

In this research, the function of cache is not only increasing the speed of data query, but also reducing the load of data center to promote a better stability and scalability.

- Cache algorithm

The Cache algorithms are the sort of algorithms for ruling the way how the cache program works, like when to evict the stored data, which data to be evicted and how to locate the data which should be evicted.

The examples of frequently used cache algorithms are: Least Recently Used (LRU), Most Recently Used (MRU), Pseudo-LRU (PLRU), Random Replacement (RR) and Segmented LRU (SLRU). /4/

- Distributed Cache

In contrast to with the traditional single node cache, a distributed cache may span multiple servers so that it can grow in size and in transactional capacity. The development of distributed cache based on the increasing internet speed and the decreasing price of both the memory and servers. Distributed cache enable the possibility to handle voluminous data by using cheap cluster consists of commodity computers. /3/ /5/

1.5 Redis

Redis is adopted as main database in the cache cluster in this thesis. Redis is an open source project (BSD licensed) which was started form 2009. It is a C based single thread NoSQL database. For bringing the high performance, redis uses the in-memory storage mechanism. Redis is a key-value based database which has fairly simple data structure; furthermore this is one of the factors which leads a better speed in data operation for Redis. Redis supports data structures such as strings, hashes, lists, sets, sorted. Redis also has various noticeable features like built-in replication, Lua scripting and LRU eviction which boost its popularity. Redis brings also great stability and scalability which is one of the key considerations of its adoption.

The Redis version adopted in the study of this thesis is 2.8.

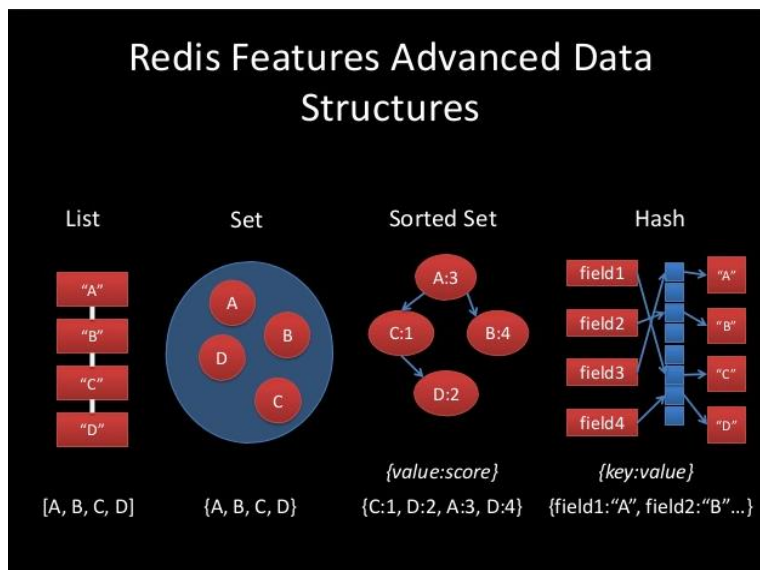


Figure 4. Redis major data structure /6/

1.6 Apache Kafka

Apache Kafka is a message broker service with the advanced feather of distributary, partitioning and replication. It's an open-source message queue developed by Apache Software Foundation based on Scala. In Kafka the work flow is organized by several modules mainly includes: Topic, Producer, Broker and Consumer. Topic is the categories Kafka maintains feeds of messages in. Producer is the process in Kafka that publishes messages. Consumer is the process that subscribes to topics and receives the published messages from Produce via the subscribed topics. Kafka message queue is hosted in a single or multiple nodes formed cluster. Each node of the cluster is called a Broker. Generally the work flow could be depicted as that producer sends messages over the network to the Kafka cluster which delivers the messages to the consumer. /7/

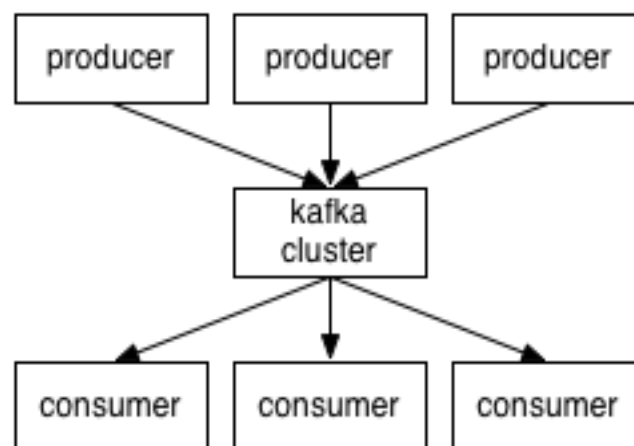


Figure 5. Model of Apache Kafka /7/

Kafka cluster processes a partitioned queue for each involved Topic. Every partition is group of messages in an ordered queue which is unmodifiable. The new committed message is appended to the tail of the queue. Every message in the queue is assigned a unique id according to its offset for the identification in the

partition. And every published message not matter whether has it been consumed are kept in a configurable period of time by the Kafka Cluster. Since the Kafka cluster only holds the essential details of data, the big volume of retained data will not lead to problems.

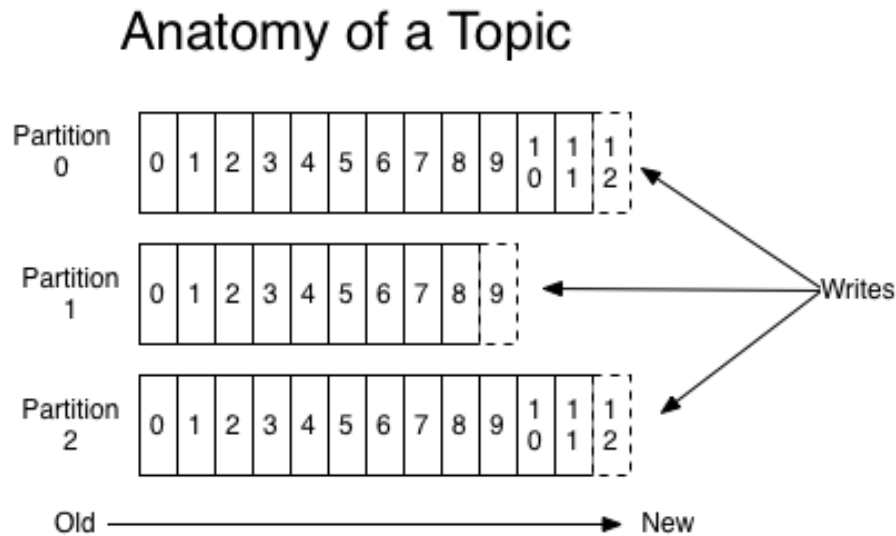


Figure 6. Anatomy of a Topic of Apache Kafka /7/

1.7 Introduction of Microservice

In software development, microservices are small, independent processes that communicate with each other to form complex applications which utilize language-agnostic APIs. These services are small building blocks, highly decoupled and focused on doing a small task, promoting a modular approach for the system level architecting. The microservices architectural style is becoming the standard for building continuously deployed systems. /8/

The term "Micro-Web-Services" was introduced by Dr. Peter Rodgers in a presentation in Cloud Computing Expo in 2005. In that event, the presentation of "Software components are Micro-Web-Services" was stated. And In May 2012, the "microservices" was decided as the most appropriate name for aforementioned system architecture. /8/ /9/

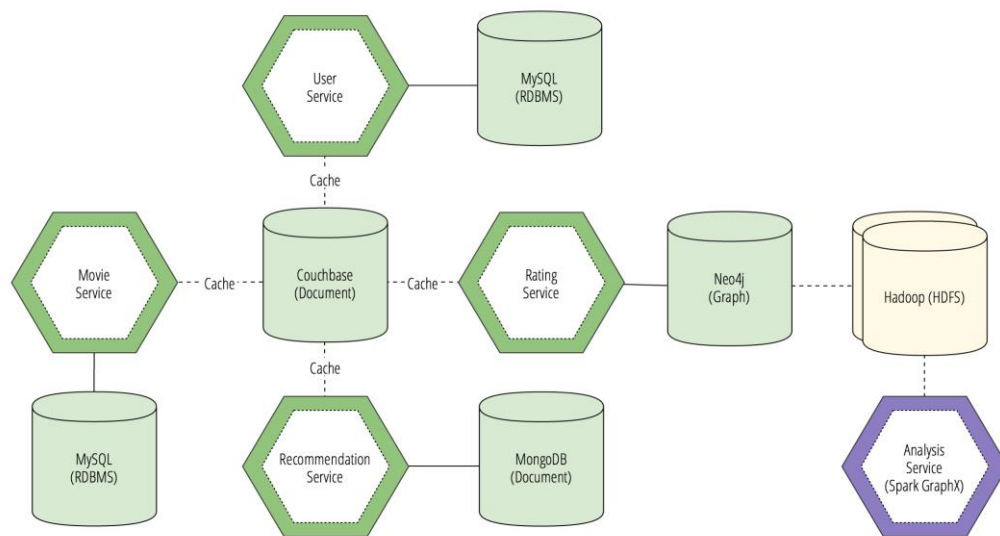


Figure 7. An example of microservice architecture /10/

1.8 Spring Boot

Spring Boot is an open source project under Spring community which was initiated in 2002. Spring Boot provides the facilitation to create stand-alone, production-grade Spring based Applications. Spring Boot can embed Tomcat, Jetty or Undertow directly which distinguishing from the traditional server-dependency model and makes Spring Boot application can be run independently. Spring Boot uses Maven or Gradle as dependency management tool and an opinionated 'starter' configuration file is provided for simplifying the work. Besides, Spring Boot also provides the production-ready features for example health checks. And it can be guaranteed that within Spring Boot, the code generation and XML configuration is completely avoided, which makes the development more convenient and efficient. /11/

Based on the features mentioned in the proceeding figure, Spring Boot is frequently used for construction of microservice architecture based system.

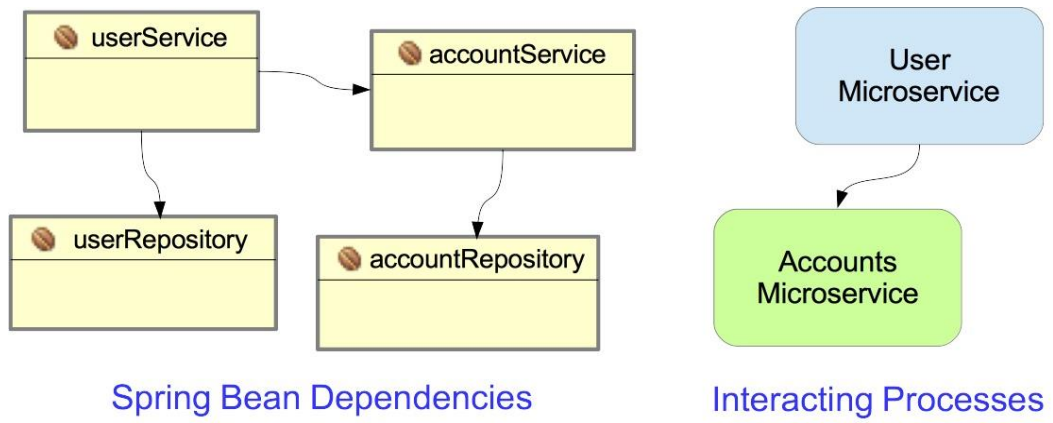


Figure 8. An example of microservice based of Spring Boot /12/

2. DATA MODEL

The cache layer in this study is built upon the existing IoT system, IoT-Ticket. The data model in this thesis follows the solution in the target IoT system. Since this study only focus on providing the data caching for analytic cluster, only the related data model is discussed. The data model session consists of two parts, the data model in IoT-Ticket and the data model in cache layer of the study.

2.1 Data model in target IoT system

In the target IoT system, the data model “ProcessData” is the model for raw sensor data which is the input for the analytic cluster. In this study, the discussion and development of data model is performed based on “ProcessData”.

Data Type	Field Name
Long Integer	mid
Integer	type
Long Integer	epochHour
Long Integer	timestamp
[dataType]	data

Table 1. ProcessData(simplified) data model

In “ProcessData” data model, Manufacturer Identification (mid) is stored in Long Integer format as the identification for distinguishing the origin of the data record. “timestamp” is stored in Long Integer format in which the time when this data record is generated is kept in millisecond scale. “epochhour” is another field for recording generating time as “timestamp”, it is calculated from “timestamp” and is in hour scale. For example, if one data record keeps “timestamp”

“1462436089149”, the corresponding “epochHour” is “1462435200000”. The formula is depicted below:

$$\begin{cases} V_{epochHour} = V_{timestamp} - (V_{timestamp} \% Hour) \\ Hour = 3600000 \text{ (ms)} \end{cases}$$

“type” indicates the data type of the data which kept in current data records. An Integer value is provided by “type”, each Integer value stands for a different data type.

Integer Value	Data Type
0	Double
1	Integer
2	String
3	Boolean
4	Binary

Table 2. Data type mapping in ProcessData(simplified) data model

The data in the “ProcessData(simplified)” model which is discussed in this study is sensor data, which is generated from sensors in the IoT system. It is strictly ordered by the timeline, and is also static, namely the generated data will never be updated or individually deleted. And it is obvious that the volume of this category of data is large and increasing fast. Therefore, it can be concluded that the main feature of this category of data can be summarized as chronological, static and voluminous.

2.2 Data model in cache

2.2.1 Fundamental data structure

The model in cache is designed based on the Redis in-built data structure and optimized for the data usage of the analytic cluster. In the work flow of the analytic

cluster, the sensor data is queried and processed by the minimum unit of hours, namely the data should be grouped by the hour when they have been generated. Hence in the cache layer, a data structure of collection should be adopted. In Redis there are 4 data types that support collection data structure: Lists, Sets, Sorted Sort and Hashes. For the sake of better performance, firstly the time-based performance of related operations between these four data types is compared.

Operation Data type	Add	Multiple Add	Get Single element by index	Get All
Lists	O (1)	O (n)	O(S)	O (N+1)
Sets	O (1)	O (n)	-	O (N)
Sorted Sets	O (log(N))	O (log(N))	O (log(N)+1)	O(log(N)+N)
Hashes	O (1)	O (n)	O (1)	O (N)

Table 3. Time-based performance of Redis collection data types /13/

In the table above, O notation is a means to classify the algorithm (data structure) on its performance when the data grows. O (1) means time taken by the command on a data structure is constant. O (N) and O (n) mean time taken by the command on a data structure scales linearly on the amount of data it contains. O (log (N)): Time taken by the command on a data structure is logarithmic in nature. Where N is the number of elements, n is the number of elements to be added or got, S is the number of elements before the target element in a list.

From the data of the table above, it can be observed that the sorted Sets is firstly out of considering because of obvious disadvantage of performance; Lists, Sets, Hashes has very similar time-based performance in the operations including: Add, Multiple Add and Get All. Furthermore, even though there is quite big difference of time-based performance between those of getting a specific element from col-

lection, but it cannot be strong evidence since this operation will not be frequently used in the target cache system.

Hence, in next step the structure and use scenario of these three data types are compared.

- Lists

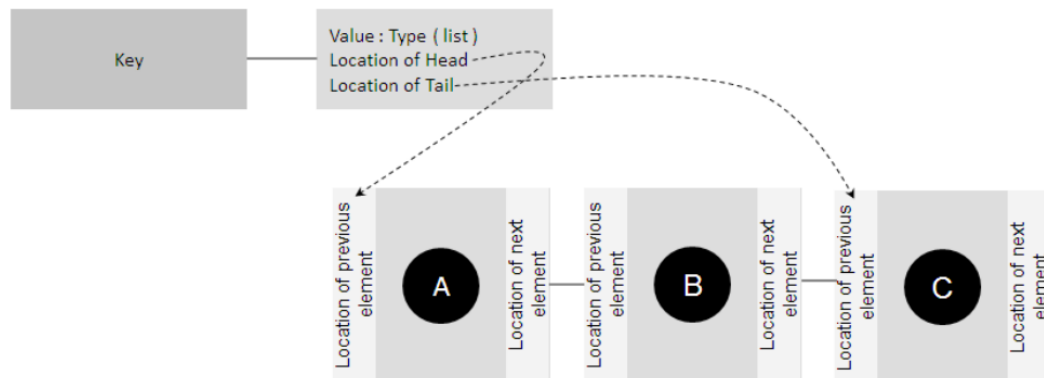


Figure 9. Lists data type in Redis /14/

Lists is designed to store data sequentially in a Redis server where the write speeds are more desirable than read performance. For example, log messages. /14/

- Sets

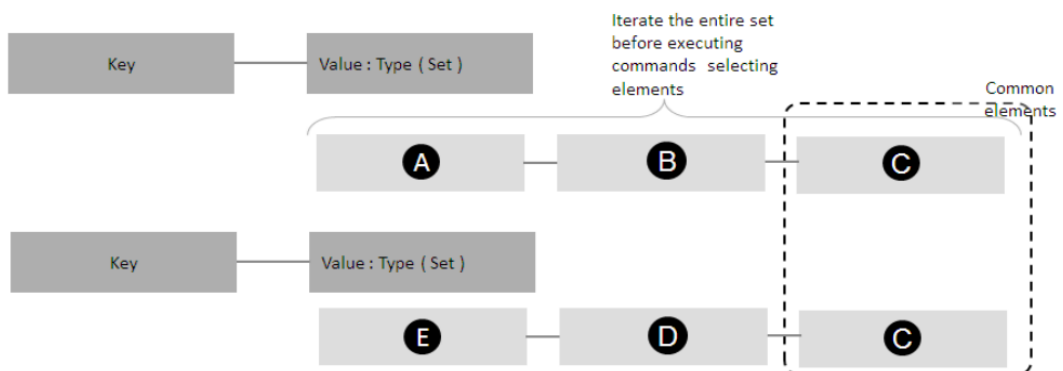


Figure 10. Sets data type in Redis /14/

Sets is designed to store data as a set in Redis server. The use cases for set data would be more for an analytics purpose, for example for the analysis of how many people browse the product page and how many end up purchasing the product. /14/

- Hashes

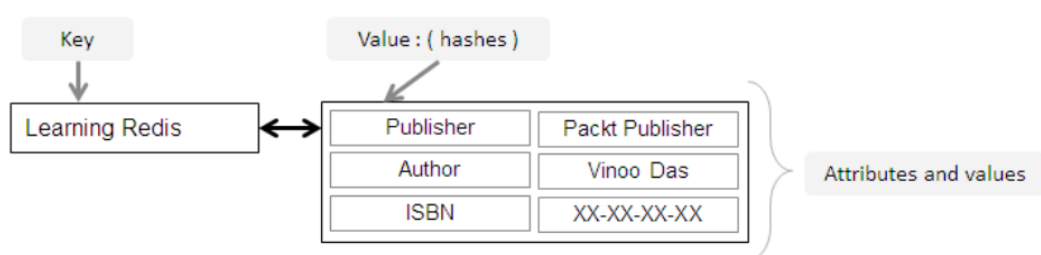


Figure 11. Hashes data type in Redis /14/

Hashes is designed to store simple and complex data objects in the Redis server. For example, user profile. /14/

For the considering of optimizing the performance for data query, the data structure in the cache layer in this study should be designed both for speedy writing and fast reading with clear organization and minimum converting operation.

Based on this consideration, it can be analyzed that Lists has a fast writing speed but it takes a longer time for the reading operation; sets is designed for data operation in Redis than data storage; both Lists and Sets have flat data structure which may bring more converting operation in practice; Hashes has both speedy writing and reading capacity and brings a multiple layers structure which is very suitable for the need of the study in this thesis. Consequently Hashes is a best choice and it is adopted.

2.2.2 Key-Value based data structure

Redis is a Key-Value based NoSQL database. Naturally a Key-Value based data structure is desired for the cache system upon it.

As stated in the preceding, the data record is distinguished by the Manufacturer Identification (mid) and grouped by the hour when they are generated (epochHour). One effective solution for the key of a data collection is to compose mid and epochHour. Therefore, the reference key is designed in string format as:

$$\text{mid} - \text{epochHour} \quad (1)$$

For example: “33156-1462435200000”.

The value part in a Key-Value pair is discussed in previous session: a Hashes data type is adopted. Therefore the data structure can be depicted as:

$$[\text{mid} - \text{epochHour}]_{\text{key}} - [\text{Hashes}]_{\text{value}} \quad (2)$$

In the studied cache system, the desired output should contains values of queried mid, epochHour, timestamp and processed data, and the latter should be grouped by epochHour. The mid and epochHour are organized in the reference key in the first layer of data structure, next the timestamp and processed data should be considered. Hashes is also a collection which stores Key-Value pairs in which the “key” position is perfect for the timestamp and the “value” position is filled by actual data. The structure of Hashes here can be depicted as:

$$[\text{timestamp}]_{\text{key}} - [\text{data}]_{\text{value}} \quad (3)$$

For example: “1462436156558-65.0235”

The complete data structure design is:

$$[\text{mid} - \text{epochHour}]_{\text{key}} - \left[\begin{array}{l} (\text{timestamp})_{\text{key}} - (\text{data})_{\text{value}} \\ (\text{timestamp})_{\text{key}} - (\text{data})_{\text{value}} \dots \end{array} \right]_{\text{value}} \quad (4)$$

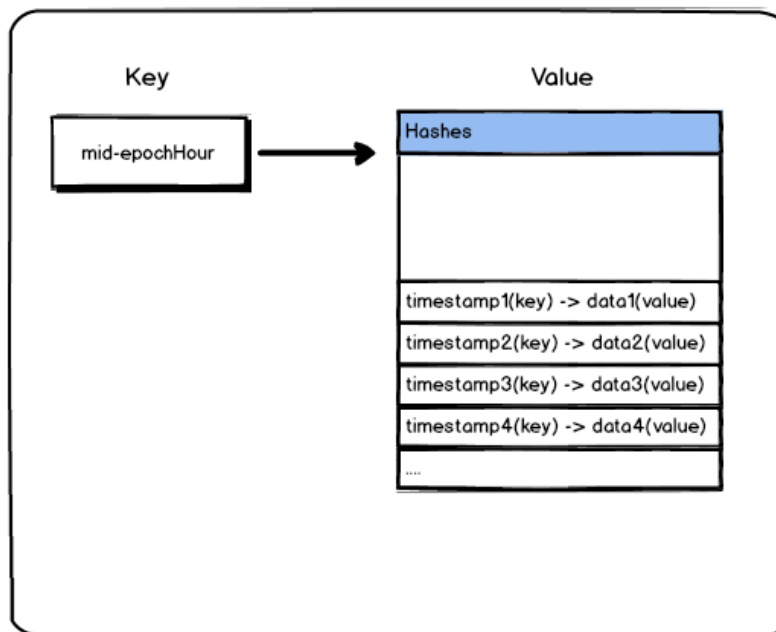


Figure 12. Data structure design in cache

With this data structure, the needs of the analytic cluster are fulfilled and the data converting operations are kept to a minimum degree. It is an effective design for the cache system in the study of this thesis.

3. CACHE STRATEGY

The cache strategy is the major focus in this thesis. The cache strategy is developed for managing the cached data and enhancing the efficiency of the usage of the caching storage. In this study, the cache strategy can be mainly divided into two layers. The first layer is upon the Redis database level, the in-built LRU implementation is deployed. The second layer is built on the microservices in this study on which the various cache algorithms can be used. In this thesis scope, the Proactive Caching is introduced and implemented.

3.1 LRU in Redis

Least Recent Use (LRU) is an extensive use cache algorithm. Typically, in LRU the least recently used items are firstly discarded when the storage is full. In original LRU solution, a track of when the stored data is used is desired, it could be expensive, if it needs to be ensured that the algorithm always discards the actual least recently used item. Namely the operation could consume much system resources. /3/

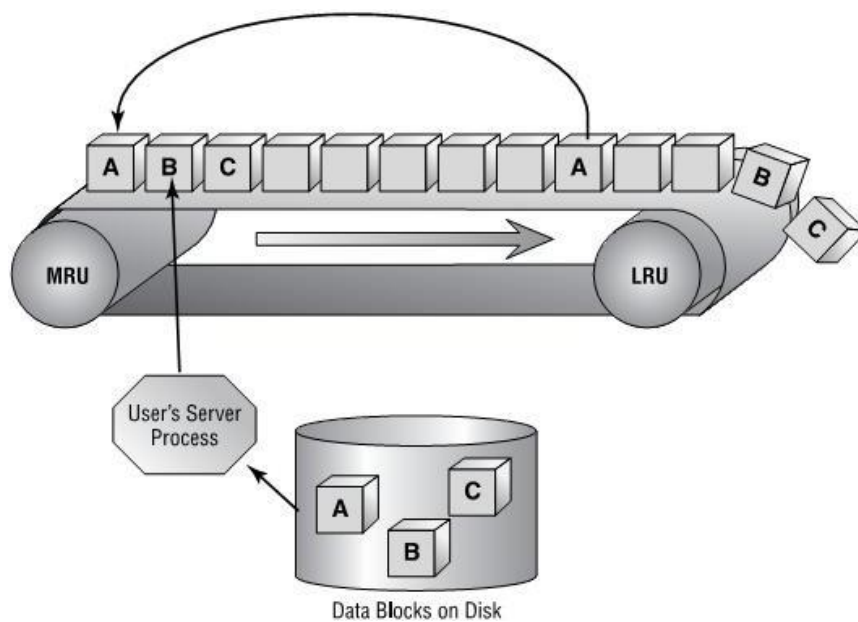


Figure 13. Traditional LRU caching schema /15/

3.1.1 Approximated LRU algorithm

In this study, the LRU implementation provided by Redis is adopted. The Redis implementation is not a strict LRU version that can be called as “Approximated LRU”, in which a concept of sampling is introduced. The reason why the real LRU is not used is that it is too expensive of the system resource for maintaining the “cache-line” and evicting the precise least recently used one. This means that it does not work to find and pick the best candidate that is least recently used for eviction. Instead, a sampling program will be run via which a small group of keys will be selected, and the one with the oldest access time within this group will be evicted.

- The pseudo-code

```
BEGIN

SAMPLE_AMOUNT <- 10 // Amount of samples
sample_group <- empty set //Initiating the sample group
/*Populating the sample group by random algorithm*/
index <-0
REPEAT
  sample_group[index] = pick data via randomized algorithm
  index ++
UNTIL index >= SAMPLE_AMOUNT

lru_data <- null // For recording LRU data
/*Looking for the LRU data in sample group*/
index <-0
REPEAT
  IF lru_data == null ||
    lru_data.timestamp > sample_group[i].timestamp
  {
    lru_data = sample_group[i]
  }
  index ++
UNTIL index >= sample_group.length

/*Evicting*/
remove lru_data
```

END

- The Flow chart

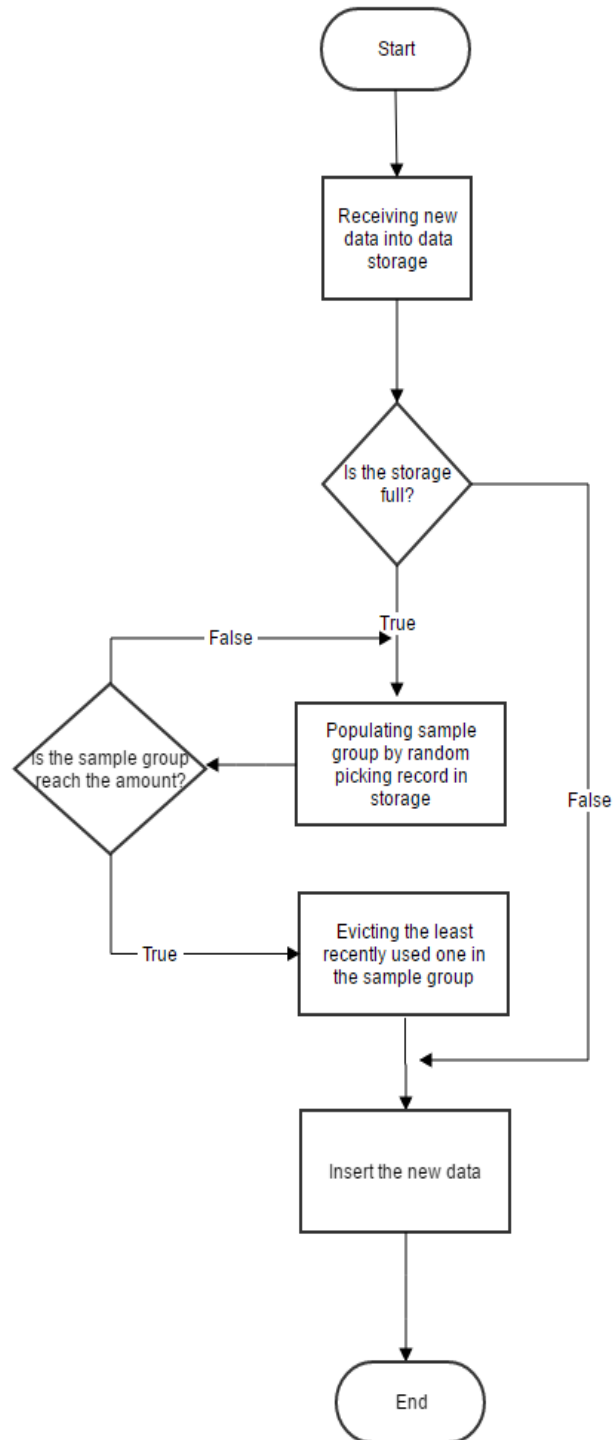


Figure 14. Flow chart of sampling in Approximated LRU

- Comparing Approximated LRU and LRU

Although Approximated LRU is not a strict LRU version, in practice the approximation is virtually equivalent to the application using Redis. A test of the difference between the LRU approximation used by Redis and the true LRU is performed, the result is represented graphically below.

The figures below show the test which filled a Redis DB with a set of data. The data were wrote in sequence, therefore the first set of inserted data has highest priority to be evicted in a strict LRU algorithm. /16/

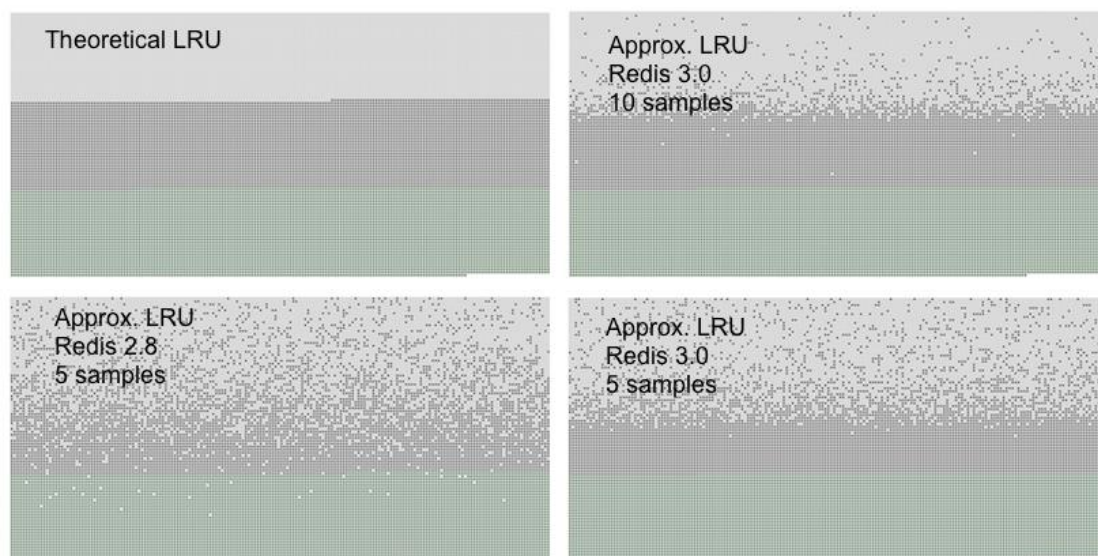


Figure 15. Graphical comparison of Approximated LRU and LRU /16/

There are three colors of points in the figures above, creating three separate area. It can be observed that the ash grey parts stand for the keys which were evicted. The gray parts stand for the keys that were not evicted. The green parts stand for the keys that were added. It is expected that in a theoretical LRU implementation, the first half within the older keys will be evicted, as showed on the first figure.

And in the Approximated LRU algorithm will instead only probabilistically evicting the older keys. /16/

From the figures of comparison above it can be observed that, the more samples are taken the better precision is achieved in the Approximated LRU, correspondingly the more system resource will be consumed. It also can be found that when the number of samples reaches 10, in Redis 3.0 the result of approximation LRU is already very close to the theoretical LRU solution.

3.1.2 The configuration for Approximated LRU in Redis

For enabling the Approximated LRU in Redis database level, there are several options should be configured from the “redis.conf” file within the Redis server.

- maxmemory

“maxmemory” is a configure option for setting the maximum memory the corresponding Redis instance can use. For example, in this study, it is expected that 2 Gigabits of the memory should be used, then:

maxmemory 2gb

- maxmemory-policy

“maxmemory-policy” is a configure option for setting the policy of data eviction after the current data amount exceeds the limit which is set by “maxmemory”. The current supported policies are listed below:

Policy Name	Description
noeviction	Errors will be returned errors when the memory is full and the client is still trying to perform the sort of operation which causes the more use of the memory.
allkeys-lru	All of the existing keys is in the sampling scope for the LRU algorithm for eviction which create room for new joined data.

volatile-lru	Only the keys with expiring set of the existing keys is in the sampling scope for the LRU algorithm for eviction which create room for new joined data.
allkeys-random	Random keys will be evicted for the sake of making space for the new data added.
volatile-random	Random keys will be evicted for the sake of making space for the new data added, but the only the keys with expiring set will be involved in the randomized scope.
volatile-ttl	Only keys with an expiring set will be evicted when the memory is full, and the eviction of keys will follow the rule of a shorter time to live (TTL) first.

Table 4. Eviction policy list and description /16/

In the study of this thesis, the “allkeys-lru” is adopted, then:

maxmemory – policy allkeys – lru

- maxmemory-samples

“maxmemory- samples” is a configure option for setting the amount of samples which will be collected by a randomized algorithm for the Redis data storage, and the collected samples will be examined to find the least recently used one among the samples when the storage size reaches the limit that is set by “maxmemory”. In the study of this thesis, the number 10 is firstly adopted and it might be modified in the further development if needed, then:

maxmemory – samples 10

3.2 Optimizing for LRU in Redis

There are two points which are developed in this study for optimizing the performance of the LRU solution in the first layer of the cache strategy. These optimizations are developed based on the characters of sensor data in the target IoT system which are chronological, static and voluminous.

3.2.1 TTL

Time to live (TTL) is an operation that is setting a limit of lifespan for an item in the system. Within the scope of this study, the mentioned item is the cached data. Setting of TTL is an optional optimizing point for near real time data query from client, such as stream processing. By setting TTL, the data will be expired and evicted in a given time period which will proactively clean the cache storage for reducing the chance of eviction sampling operation in Redis and leading a more efficient cache storage usage.

3.2.2 Placeholder

As mentioned in the preceding, in the target IoT system, the data is normally generated from sensors. This category of data is generated following the timeline and stored as logs for further use which will never be updated or individually deleted. It can be summarized that it is strictly chronological and invariable. Moreover, in practice, it is possible that in some period of time there is no data generated in some sensors. Therefore the data is not always sequential. In this thesis, the data is grouped by hour when they are generated and therefore it can be analogized that the data is generated as a series of block following the timeline, and at some period of time, there is no block existing and the absence lasts permanently.

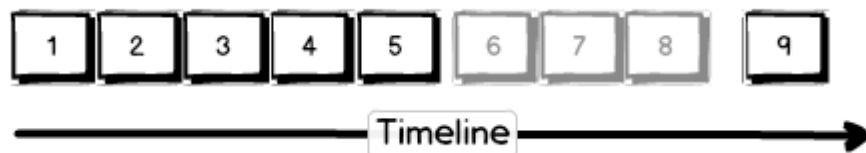


Figure 16. Inconsecutive data generating

As shown in the figure above, the dark blocks stand for the existing data, the grey ones stand for the lacking data in the timeline.

In the first layer of cache strategy, cache server receives data after the query operation from client, for example when client queries the data in the timespan from timestamp 1 to timestamp 9 as depicted in figure 16, the cache will receive the data block 1, 2, 3, 4, 5 and 9, and they will be kept in the cache storage. Next time, when client queries from the same timespan (timestamp1 to timesptamp9), the cached data will be returned. But there is an issue raised here, since the data block at timespan (grouped by hour) 6, 7, 8 are not existing in the cache, a query operation from data center will still be triggered. Obviously this operation gets no data. However, it is evitable. Since the data within this study is strictly chronological and invariable, the absence of data in some timespan will never be filled, the aforementioned database query operation should be prevented for performance optimization.

For preventing the redundant query operation described above, one concept of “placeholder” is introduced here. It can be described as: when populating the cache with queried data, filling the timespan (by hour) which return no data a placeholder in the cache. The placeholder is only a mark in the storage of cache which only marks the place is fulfilled and takes a small and constant room.

The formula of stored cache data collection is:

$$\sum_{i=0}^L f(i) = \begin{cases} P, & len(i) \leq 0 \\ H, & len(i) > 0 \end{cases}$$

Where L is the count of hours in the timespan of the incoming query, i stands for the index of data block in the return data, P stands for the placeholder, H stands for the Hashes data type in Redis, len(i) stands for the amount of data record in the data block under index i.

On the implementation perspective, for coherence of cache storage, the placeholder is also formed by a Hashes data type with the key of “mid-epochHour”. Because of the constraint from the Redis database, an empty Hashes cannot be kept in the database. Hence, inside the corresponding Hashes, instead of the real data one and only one pair of the constant string “EMPTY_TAG”:“EMPTY_TAG” will be set. Then, on the caching reading phase, one filter for filtering the placeholder is added which distinguishes placeholder according to both length of the examined Hashes and the mark which is given “EMPTY_TAG”.

- Flowchart of “Placeholder” optimizing:

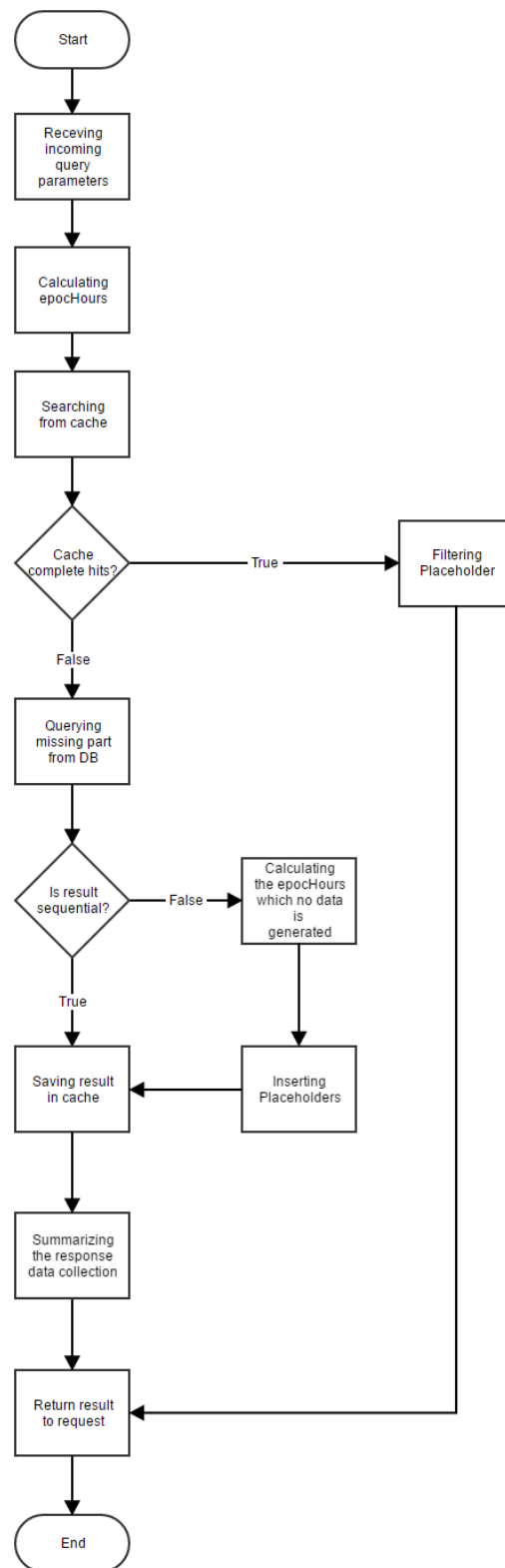


Figure 17. Flowchart of “Placeholder” optimizing

- The pseudo-code

```
BEGIN

/*Initiating system resource*/
HOUR <- 1000 * 60 * 60
EMPTY_TAG <- "EPT"
db <- Data_center_connection
redisCache <- Cachec_luster

/*Initiating query parameters*/
mid <- MID
start <- start timestamp
end <- end timestamp

/*Calculating epochHours in the time range from query*/
epochhours <- calculating epochhours from start and end
length <- length of epochHours
index <- 0

/*storing the epochour under which data is missing*/
epochHours_lack <- empty colletion

/*storing data found from cache*/
hit_data_from_cache <- empty colletion

/*Search in cache*/
REPEAT
  key <- mid+"-"+epochhours[index]
  data <- search reqlt from redisCache by key

  /*If cache hit, filtering the placeholder
  and add result to return data*/
  IF data is found
  {
    filtering the Placeholder by examining EMPTY_TAG
    IF data is not a Placeholder
    {
      hit_data_from_cache add data
    }
  }

  /*If cache miss, mark the missing epochour*/
  ELSE
  {
    epochHours_lack add epochhours[index]
```



```

}
index ++
UNTIL index >= length

/*If no missing epochhour is marked,
return the result*/
IF epochHours_lack.length <= 0
{
RETURN hit_data_from_cache
}
/*If some data missing, query from DB*/
ELSE
{
data_set_from_DB <- query from db

/*Inserting placeholder if return data is not sequential*/
IF data_set_from_DB is not sequential
{
REPEAT
/*Assembling placeholder*/
placeholder_key <- mid+"-"+missing epochhour
placeholder_value <- Hashes: [EMPTY_TAG:EMPTY_TAG]
placeholder <- placeholder_key:placeholder_value
insert placeholder into redisCache
UNTIL all missing epochhours is inserted as placeholder
}

/*merge data set queried for cache and DB*/
return_data_set <- data_set_from_DB + hit_data_from_cache

/*Return final result*/
RETURN return_data_set
}
}
END

```

3.3 Proactive Caching

Proactive caching is firstly introduced by Microsoft SQL Server Analysis Services for optimizing the performance of caching in use case of near real time data housing and business intelligence. This is also an ideal idea for enhancing the caching performance in the analytic cluster in an IoT system when the near real time data

querying is required, for example, stream data processing. In traditional cache mechanism, the cache layer retrieves data reactively in which the data is queried from database and filled in cache when a request comes and corresponding data is missing in the cache.

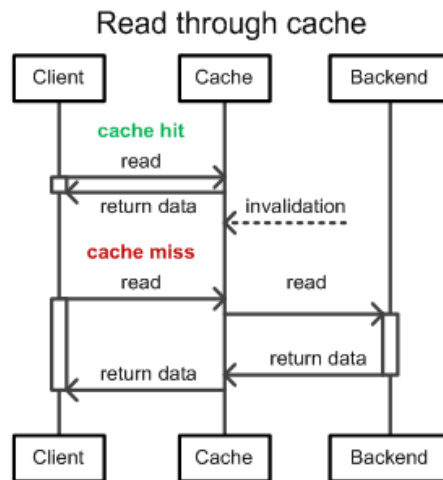


Figure 18. Traditional cache pattern /17/

Contrast to traditional cache pattern, in Proactive Caching pattern, the fresh data is regularly queried from the database and updated in the cache, which allows the clients read near real time data directly from cache. By using Proactive Caching, the caching performance of near real data housing is enhanced and the load of center database in concurrence is reduced.

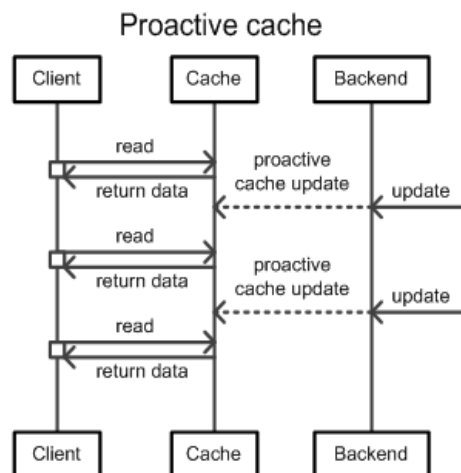


Figure 19. Proactive Caching pattern /17/

Proactive Caching is adopted in the second layer of the cache strategy. And in the target IoT system, Apache Kafka message queue is adopted as message broker between the data source (sensors) and consumers (center database and other backend components). Based on this architecture, the solution of Proactive Caching used in this study is even more aggressive that instead of regularly querying for center database, the data is collected directly from the data source by registering a consumer in the Kafka message queue and listening to the topics of given sensors for which the Proactive Caching is enabled.

3.3.1 Zookeeper and Kafka

In the current phase of development, it is preferred to establish a separate Kafka message queue then directly using the one in production server. For starting the Kafka message queue, a Zookeeper and a Kafka server are required.

Zookeeper is a configuration service, which is designed for the distributed system. In the general practice, the Apache Kafka is deployed upon the ZooKeeper which notably reduce the complex for the development and maintenance. /18/

For enabling the Zookeeper and Kafka service, firstly they should be downloaded and installed in the development machine. Then, under the windows folder of Kafka server, the commands below should be run:

```
zookeeper – server – start. bat config\zookeeper. properties
```

And

```
kafka – server – start. bat config\server. properties
```

```

[2016-05-09 13:35:41.015] INFO Server environment:java.io.tmpdir=C:\Users\yzhou\AppData\Local\Temp\ (org.apache.zookeeper.server.ZooKeeperServer)
[2016-05-09 13:35:41.015] INFO Server environment:java.compiler=<NA> (org.apache.zookeeper.server.ZooKeeperServer)
[2016-05-09 13:35:41.015] INFO Server environment:os.name=Windows 8.1 (org.apache.zookeeper.server.ZooKeeperServer)
[2016-05-09 13:35:41.015] INFO Server environment:os.arch=amd64 (org.apache.zookeeper.server.ZooKeeperServer)
[2016-05-09 13:35:41.015] INFO Server environment:os.version=6.3 (org.apache.zookeeper.server.ZooKeeperServer)
[2016-05-09 13:35:41.015] INFO Server environment:user.name=yzhou (org.apache.zookeeper.server.ZooKeeperServer)
[2016-05-09 13:35:41.015] INFO Server environment:user.home=C:\Users\yzhou (org.apache.zookeeper.server.ZooKeeperServer)
[2016-05-09 13:35:41.015] INFO Server environment:user.dir=C:\Kafka\kafka_2.11-0.9.0.1 (org.apache.zookeeper.server.ZooKeeperServer)
[2016-05-09 13:35:41.020] INFO tickTime set to 3000 (org.apache.zookeeper.server.ZooKeeperServer)
[2016-05-09 13:35:41.020] INFO minSessionTimeout set to -1 (org.apache.zookeeper.server.ZooKeeperServer)
[2016-05-09 13:35:41.021] INFO maxSessionTimeout set to -1 (org.apache.zookeeper.server.ZooKeeperServer)
[2016-05-09 13:35:41.068] INFO binding to port 0.0.0.0/0.0.0.0:2181 (org.apache.zookeeper.server.NIOServerCnxnFactory)

```

Figure 20. Zookeeper service is running

```

GroupCoordinator)
[2016-01-01 15:40:12.404] INFO [ExpirationReaper-0], Starting (kafka.server.DelayedOperationPurgatory$ExpiredOperationReaper)
[2016-01-01 15:40:12.435] INFO [Group Metadata Manager on Broker 0]: Removed 0 expired offsets in 16 milliseconds. (kafka.coordinator.GroupMetadataManager)
[2016-01-01 15:40:12.420] INFO [ExpirationReaper-0], Starting (kafka.server.DelayedOperationPurgatory$ExpiredOperationReaper)
[2016-01-01 15:40:12.513] INFO [ThrottledRequestReaper-Producer], Starting (kafka.server.ClientQuotaManager$ThrottledRequestReaper)
[2016-01-01 15:40:12.529] INFO Will not load MX4J, mx4j-tools.jar is not in the classpath (kafka.utils.Mx4JLoader)
[2016-01-01 15:40:12.529] INFO New leader is 0 (kafka.server.ZooKeeperLeaderElector$LeaderChangeListener)
[2016-01-01 15:40:12.529] INFO [ThrottledRequestReaper-Fetch], Starting (kafka.server.ClientQuotaManager$ThrottledRequestReaper)
[2016-01-01 15:40:12.545] INFO Creating /brokers/ids/0 (is it secure? false) (kafka.utils.ZKCheckedEphemeral)
[2016-01-01 15:40:12.560] INFO Result of znode creation is: OK (kafka.utils.ZKCheckedEphemeral)
[2016-01-01 15:40:12.576] INFO Registered broker 0 at path /brokers/ids/0 with addresses: PLAINTEXT -> EndPoint: :9092,PLAINTEXT (kafka.utils.ZkUtils)
[2016-01-01 15:40:12.571] INFO kafka version : 0.9.0.0 (org.apache.kafka.common.utils.AppInfoParser)
[2016-01-01 15:40:12.591] INFO Kafka commitId : fc7243c2af4b2b4a (org.apache.kafka.common.utils.AppInfoParser)
[2016-01-01 15:40:12.591] INFO [Kafka Server 0], started (kafka.server.KafkaServer)
[2016-01-01 15:50:12.418] INFO [Group Metadata Manager on Broker 0]: Removed 0 expired offsets in 0 milliseconds. (kafka.coordinator.GroupMetadataManager)

```

Figure 21. Kafka service is running upon the Zookeeper

The Screenshot above shows that Zookeeper is up running at port 2181, Kafka connects to Zookeeper and is running of port 9092. After the Kafka is running correctly, the step can be move to next.

3.3.2 Simulated data source

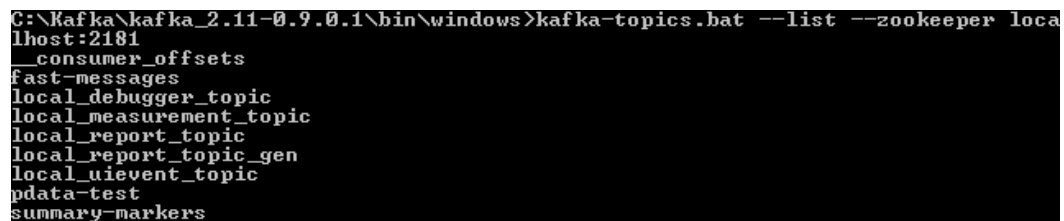
After the Kafka message queue is available on development machine, a simulated data source can be developed for further use. The simulated data source is fairly simple and it consists of two parts: A topic in Kafka message queue and a single thread application acts as producer and sends a message via given topics. For creating a topic in Kafka message queue, the following command should be run under the windows folder of Kafka server:

```
kafka -- topics.bat --create --zookeeper localhost: 2181 --replication
-- factor 1 --partitions 1 --topic pdata -- test
```

The command above creates a topic “pdata-test” with 1 partition and 1 replication. The result of topic creation can be tested by the command:

```
kafka -- topics.bat --list --zookeeper localhost: 2181
```

It can be checked in the screen that the topic “pdata-test” is created.



```
C:\Kafka\kafka_2.11-0.9.0.1\bin\windows>kafka-topics.bat --list --zookeeper loca
lhost:2181
__consumer_offsets
fast-messages
local_debugger_topic
local_measurement_topic
local_report_topic
local_report_topic_gen
local_uievent_topic
pdata-test
summary-markers
```

Figure 22. List of created topics in Kafka

The simulated producer is an independent Spring Boot application which connects to the Kafka message queue and generating dummy data and publishes them via the topic which is created in the last step. The simulated producer uses the official provided java library “kafka-clients-0.9.0.0”.

Firstly the configuration for connecting to development Kafka message queue is set:

```
bootstrap.servers=127.0.0.1:9092
acks=all
retries=0
batch.size=16384
auto.commit.interval.ms=1000
linger.ms=0
key.serializer =
org.apache.kafka.common.serialization.StringSerializer
```

```
val-  
ue.serializer=org.apache.kafka.common.serialization.StringSe  
rializer  
block.on.buffer.full=true
```

Secondly the application imports the class “KafkaProducer” and “ProducerRecord” from library “kafka-clients-0.9.0.0”, loads the configuration, connects to the development Kafka service, continually generates dummy messages with “mid”, “timestamp”, “valdouble” in JSON format simulating a sensor does and publishes them via topic “pdata-test”.

- Pseudo codes:

```
BEGIN  
  
/*Simulated senensor identification*/  
MID <- 30979  
  
/*Setting configuration info for connection*/  
Properties <- Configuration_Info  
  
/*Importing helper class from "kafka-clients-0.9.0.0"*/  
IMPORT KafkaProducer  
IMPORT ProducerRecord  
  
/*Initiating KafkaProducer by loading configuration*/  
producer <- KafkaProducer.load(Properties)  
  
/*Repeating sending message with JSON format via given topic  
simulating a sensor*/  
REPEAT  
  /*Creating simulating data*/  
  jsonObj <- {MID, current timestamp, random value}  
  /*Testing topic*/  
  topic <- "pdata-test"  
  kafkaMessage <- ProducerRecord(topic, jsonObj)  
  /*Publishing the message*/  
  producer.SEND(kafkaMessage)  
UNTIL TERMINAL  
  
END
```

3.3.3 Consumer in Kafka

In this study, a Kafka consumer performs as bridge between data source (producer) and cache layer. It is built as a separate Spring Boot component which can be started and terminated according to configuration information and does not influence other threads. In the same way as the simulated producer, the consumer uses the officially provided java library “kafka-clients-0.9.0.0”.

Firstly the configuration for connecting to Kafka message queue for development is set:

```
bootstrap.servers=localhost:9092
group.id=test
enable.auto.commit=true
key.deserializer=org.apache.kafka.common.serialization.StringDeserializer
value.deserializer=org.apache.kafka.common.serialization.StringDeserializer

session.timeout.ms=10000

# These buffer sizes seem to be needed to avoid consumer
# switching to a mode where it processes one bufferful
# every 5 seconds with multiple timeouts along the way.
fetch.min.bytes=50000
receive.buffer.bytes=262144
max.partition.fetch.bytes=2097152
```

Next, the application imports the class “KafkaConsumer”, “ConsumerRecord” and “ConsumerRecords” from library “kafka-clients-0.9.0.0”, loads the configuration and connects to the development Kafka service. A reference of cache service which maintains the cache layer is injected in running time by using the Spring feature “Dependency Injection” (DI).

After the connection with the Kafka service is established, the consumer subscribes to the given Topic “pdata-test” and listens to the coming message. When

the message comes, it will be decoded from the JSON format into values and updated to the cache layer.

- Pseudo codes:

```
BEGIN

/*Simulated senensor identification*/
TOPIC <- pdata-test

/*Setting configuration info for connection*/
Properties <- Configuration_Info

/*Importing helper class from "kafka-clients-0.9.0.0"*/
IMPORT KafkaConsumer
IMPORT ConsumerRecord
IMPORT ConsumerRecords

/*Import Cache_service by Dependency Injection
in running time*/
IMPORT Cache_service

/*Initiating KafkaConsumer by loading configuration*/
consumer <- KafkaConsumer.load(Properties)

/*consumer subscribe to the given topic*/
consumer.subscribe([TOPIC])

/*consumer listens to the message from Kafka service*/
REPEAT

    consumer listen to the subscribed topic

    IF message comes
    {
        /*Messages comes as a set*/
        ConsumerRecords <- message.getRecords
        REPEAT
            /*Iterating the message set*/
            ConsumerRecord = ConsumerRecords.next

            /*Decoding*/
            data <- DECODE_JSON(ConsumerRecord)

            /*Updating the cache via Cache_service*/
            Cache_service.update_cache(data)
```



```

    UNTIL all records are processed
  }
UNTIL TERMINAL
END

```

3.4 Aggregation

It can be summarized that the cache strategy in this study contains 2 independent layers. The first layer is an optimized LRU solution which performs as basic cache policy and which is deployed across the database layer and service layer. The second layer, Proactive Caching solution, acts as supplement cache policy for optimizing the cache performance in near real time data process scenario. It is located in an individual service component and it can be initiated and terminated via configuration according to the requirement. The two layers collaborate and maintains the same cache database cluster.

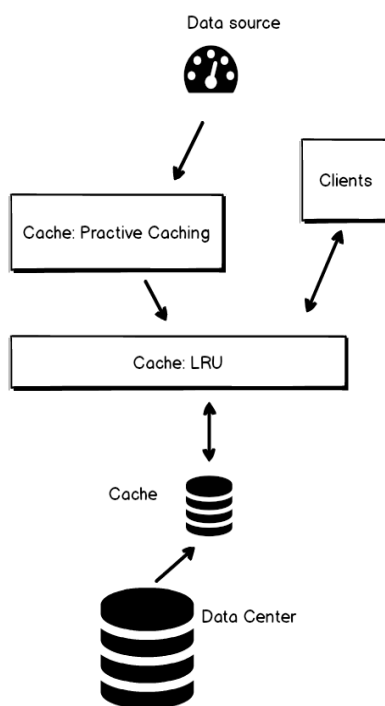


Figure 23. Aggregation of cache strategy

4. CLUSTER TOPOLOGY MODEL

For the sake of stability and scalability, it is important that data is kept in a replicated manner. In the study of this thesis, instead of using the single node database, Redis database cluster is involved. A consideration of how the nodes in the cluster are connected and what is the way they collaborate is raised. The targets of the cluster topology model design are two points:

1. Splitting dataset among multiple nodes
2. Continuous operations when a subset of the nodes are failures

Different to most common cache solutions which have low write and high read scenario, the cache in this study needs to cater for both high write concurrency from Proactive Caching mechanism and high read concurrency from client which is analytic cluster.

4.1 Master – Master Model

In the built-in solution for replication provided by Redis which is a simple Master – Slave replication, the slave node only has read capacity. Therefore, for catering for the high concurrency for write, a Master – Master topology model is introduced.

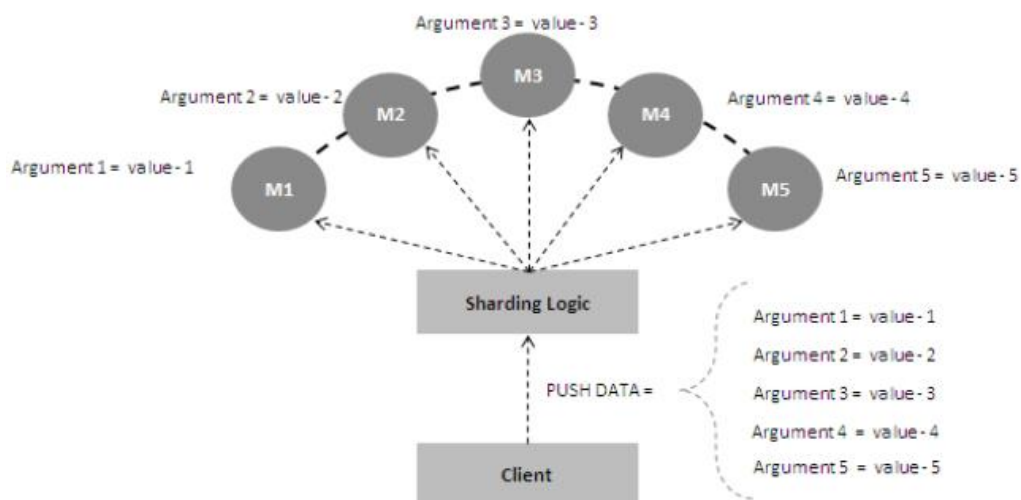


Figure 24. Master – Master topology mode /14/

A Master – Master topology model connects multiple Redis databases forming a master cluster in which every node carries both read and write capabilities. This solution solves the need of high concurrency in write. In the Master – Master topology model, the dataset is partitioned among the master nodes within the cluster according to a given rule.

Many databases come with the in-built capability for sharding the data across nodes. The advantage of in-built dataset sharding is that, apart from high concurrency in writes, it provides a mechanism of partial failure tolerance. Namely, even if one of the nodes in the cluster goes down, but the cluster can still maintain the response for request correctly and continuously.

Redis does not natively provide the solution for aforementioned data sharding. However, it is applicable to build logic upon the Redis cluster, which performs to shard and store the dataset which will still enable the capability for catering high concurrency in write. In the study of this thesis, the sharding logic can be deployed on the Spring Boot based service, which will be described detailed in the following implementation session and Data Partitioning Algorithm chapter.

4.2 Master – Slave Model

Besides the high concurrency in write, the high concurrency in read is another requirement which should be satisfied on the cache layer in the study of this thesis. For archiving this target, a Master – Slave model is considered.

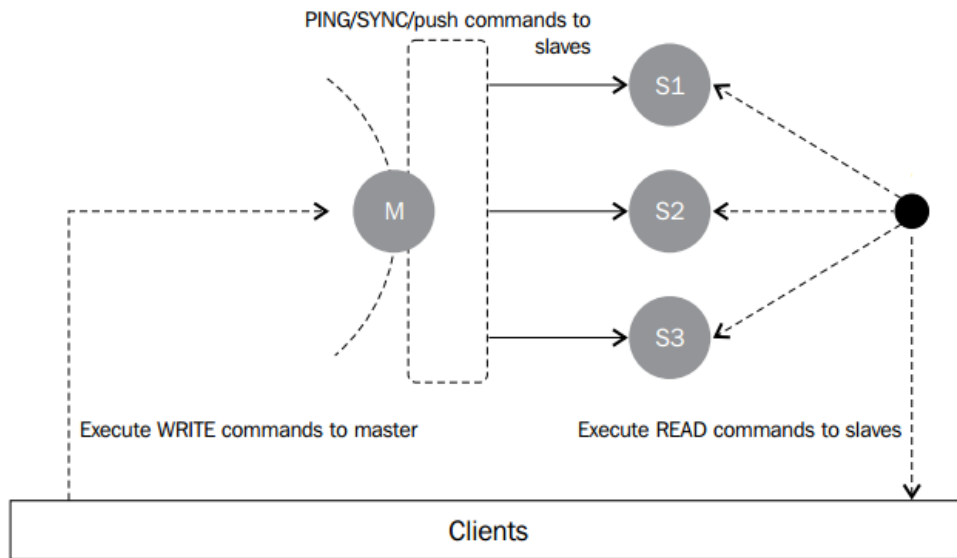


Figure 25. Master – Slave topology mode /14/

In a Master – Slave model, master is the node for data writing and then the data is replicated into all the slave nodes which response for the data read request. The replication performances are asynchronous, namely when the master node receives a data writing, the slaves are not written synchronously. But a separate process performs the replication asynchronously; in other words eventual consistency. This mechanism brings lower impact of performance. In contrast, if the synchronous replication is deployed instead, the master will update all the slaves synchronously when a data is written and then only all updates are performed, this data writing operation can be marked as a success and finished. Obviously the more performance penalty will be brought.

Redis provides an in-built Master – Slave solution in database level which brings in production performance, failure-tolerance and linear scalability up to 1,000 nodes. Therefore it is adopted in the study of this thesis.

4.3 Master – Master – Slave Model

After the introduction of both Master – Master and Master – Slave topology model, an integration of them that is able to cater high concurrency in both write and read can be made. It's called “Master – Master – Slave” topology model.

In the “Master – Master – Slave” topology models, a set of master nodes connect together forming a sub-cluster which consumes all of the write operation and the dataset is sharded among the master nodes according to the data sharding algorithm, Consistent Hashing. The data partition in every master node is replicated into a group of slave nodes in which each slave node keeps a complete copy of data from corresponding master node. And the slave nodes response all of the read request. Besides the combination of the advantage of Master – Master and Master – Slave topology model, Master – Master – Slave solution also brings an extra flexibility when the cluster is extending. When the concurrency in write is increasing, the more nodes can be added in Master sub-cluster, when the concurrency in read is increasing, the more slaves nodes can be added.

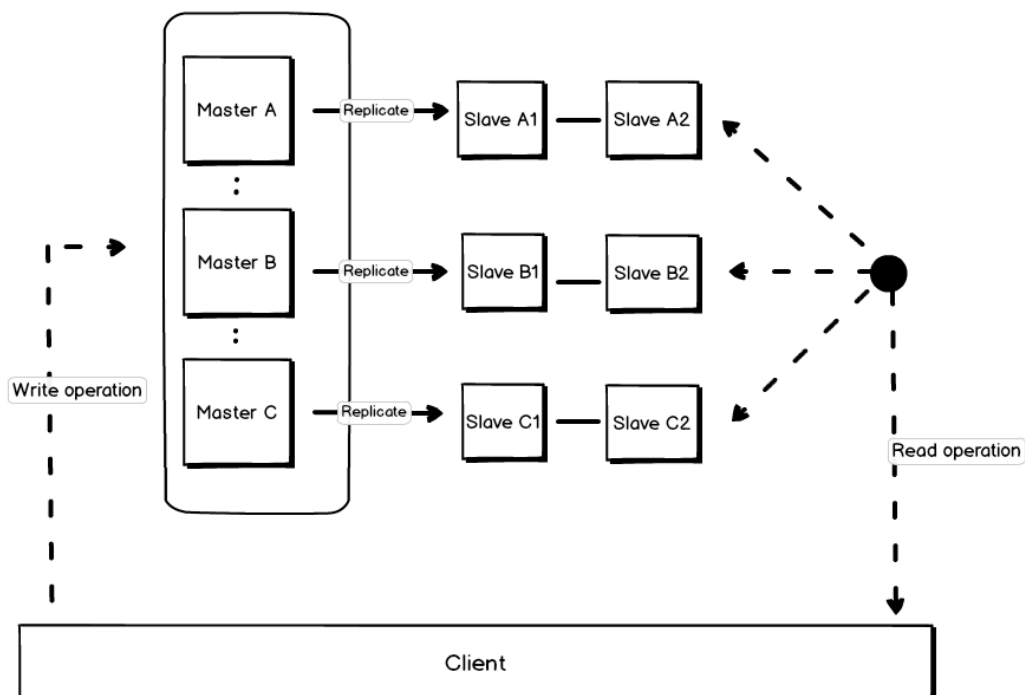


Figure 26. Master – Master – Slave topology model

4.4 Implementation

For enabling the Master – Master – Slave topology model in the Redis cluster, there are mainly two tasks. The first one is connecting the set of master nodes virtually by the sharding algorithm, the second one is up running and configuring a given set of slave nodes for each master nodes.

4.4.1 Sub-cluster of master nodes

Since the Redis does not originally support the Master – Master topology, the connection of master nodes should be maintained by additional service with the implementation of a data partitioning algorithm. In this research the service is formed by a Spring Boot application and it acts like a hub for every request to the cache cluster.

- Pseudo codes:

```
BEGIN

/*Initial the sub-cluster of master*/
REPEAT
Nkey <- Hash(masterNode)
master_pool.add(Nkey:masterNode)
UNTIL all master nodes are added

/*Only read and write operation is required*/
REPEAT
  listen to the event

  /*If a write request comes, the value
  is distributed to mapped node*/
  IF event comes && event = "write"
  {
    Vkey <- Hash(value)
    Nkey <- Mapping_by_algorithm(Vkey)
    masterNode <- master_pool.get(Nkey)
    masterNode.addValue(value)
  }

  /*If a read request comes, the value
  is read from a slave node*/
  IF event comes && event = "read"
```

```

{
  Vkey <- Hash(key)
  Nkey <- Mapping_by_algorithm(Vkey)
  slaveNode <- master_pool.get_a_slave(Nkey)
  response <- slave.read(key)
  return response
}

UNTIL TERMINAL

END

```

4.4.2 Sub-cluster of Master - Slave

After the arrangement of the master nodes, the corresponding slave nodes can be connected to each of them. This step can be done by using Redis in-built Master – Slave mechanism. A “redis.conf” file which contains the configuration information in Redis server plays important role here, in which the slave node related options is configured.

The major options about the configuration for slave node are listed in the table below:

Configuration option	Description
slaveof <masterip> <masterport>	Master-Slave replication. slaveof is used to make a Redis instance a replication of another Redis server.
slave-priority <number>	The slave-priority is used by Redis Sentinel for the selection of new master node from the slave nodes when the current master node is down. The slave-priority is represented with an integer number. The lower

	number represents the higher priority in selection. But the integer 0 is with a special setting. If a slave node is set slave-priority = 0, it will not be selected to be the master node.
masterauth <master-password>	Password for connection, if the master is password protected.
slave-serve-stale-data <boolean>	When the connection between the slave and its master is broken, or when the replication is still in progress, there are two options for the slave to perform: [no]: the slave will answer to all the sort of request of "SLAVEOF" and "INFO" with an error "SYNC with master in progress". [yes]: the slave will still answer with the existing data to client requests.
repl-ping-slave-period <number>	Slaves send PINGs to server in a predefined interval.
repl-timeout <number>	Timeout for: 3) Slave timeout in the masters' perspective. 2) Master timeout in the slaves' perspective. 1) Bulk transfer I/O during SYNC, in slaves' perspective.
repl-disable-tcp-nodelay <boolean>	After SYNC, Disabling the TCP_NODELAY on the slave socket: [no]: the less delay but the more bandwidth

	<p>will be used.</p> <p>[yes]: The smaller number of TCP packets and less bandwidth are used when sending data to slaves. However, in slaves' sides, the extra time is consumed when receiving these data.</p>
repl-backlog-size <size>	<p>Setting the replication backlog size which is a buffer for keeping the slave data when slaves are offline for some while. When the disconnected slaves go online, a full resync might be avoid, since a partial resync could be already enough, which just passes the portion of missed data while the slaves are disconnected.</p>
repl-backlog-ttl <number>	<p>Defining how long time the data will be buffered when the slaves are disconnected.</p>

Table 5. Redis configuration options for slave node /19/

As example, in this study, these values are set for establishing a slave node:

Current slave node will run at port 16380, with offset 1000 to its master node

port 16380

The master node for this slave node is located in development machine on port 6380

slaveof 127.0.0.1 6380

Because of the limit of Master – Master – Slave topology model at current phase, a slave should not be promoted to a master node when its master node is down.

```
slave – priority 0
```

The slave node should continuously reply to the client when its master node is disconnected for consistency.

```
slave – serve – stale – data yes
```

Slave sends PING to master node in every 10 seconds

```
repl – ping – slave – period 10
```

The timeout is 60 seconds for bulk SYNC, slave-master communication

```
repl – timeout 60
```

TCP_NODELAY on the slave socket after SYNC should be enabled

```
repl – disable – tcp – nodelay no
```

The replication backlog which acts as buffer, the size is set to 1mb

```
repl – backlog – size 1mb
```

The backlog will be freed after a master has disconnected with slaves for 1 hour,

```
repl – backlog – ttl 3600
```

5. DATA PARTITIONING ALGORITHM

For maintaining the data partitioning among the distributed cache cluster, a proper data partitioning algorithm should be deployed. The algorithm which is adopted in this study is Consistent Hashing.

Consistent Hashing is one of distributed hash table (DHT) algorithms. It was introduced by Massachusetts Institute of Technology (MIT) for relieving Hot Spots on the World Wide Web (WWW). Consistent Hashing solves the problems brought from CARP which performs simple hash algorithm, and deploys the DHT in P2P environment in practice. /20/

Consistent Hashing is defined with 4 properties:

- Balance

When given a specific view \mathcal{V} a set of items which are selected from the hash family by a randomly chosen function. If also it has high probability the fraction of items mapped to each bucket is $O(1/|\mathcal{V}|)$, a ranged hash family is balanced. /20/

- Monotonicity

When for all views $\mathcal{V}_1 \subseteq \mathcal{V}_2 \subseteq \mathcal{B}$, $f_{\mathcal{V}_1}(i) \in \mathcal{V}_1$ implies $f_{\mathcal{V}_1}(i) = f_{\mathcal{V}_2}(i)$, the ranged hash function f is monotone. Also, a ranged hash family is monotone, if all ranged hash function in which is. /20/

- Spread

Set $\mathcal{V}_1 \dots \mathcal{V}_v$ be a group of views, which consist of C different buckets and each of them individually having at least C/t buckets. The spread $\sigma(i)$ is the quantity $|\{f_{\mathcal{V}_j}(i)\}_{j=1}^v|$, for a ranged hash function and a specific item i . The maximum spread of an item is the spread of a hash function $\sigma(f)$. When with high probab-

ity, the spread of a hash family is σ . σ is also the spread of a random hash function from the family. /20/

- Load

A set of V views is defined as before. For a bucket b and a ranged hash function f , the load $\lambda(b)$ is the quantity $|\cup_{\mathcal{V}} f_{\mathcal{V}}^{-1}(b)|$. The hash function's load is the maximum load of a bucket. When with high probability the load of a hash family is λ , a randomly chosen hash function also has load λ . (Note that $f_{\mathcal{V}}^{-1}(b)$ is the set of items assigned to bucket b in view \mathcal{V}) /20/

Where \mathcal{I} is the group of items, \mathcal{B} is the group of buckets and i is item. A view (\mathcal{V}) is any subset of the buckets \mathcal{B} . $f(\mathcal{V}, i)$ is the bucket, to which item i is assigned in view \mathcal{V} and $f(\mathcal{V}, i) = f_{\mathcal{V}}(i)$. It is required that $f_{\mathcal{V}}(\mathcal{I}) \subseteq \mathcal{V}$ for every view \mathcal{V} because items should only be assigned to usable buckets. /20/

5.1 Simple hash solution

The formula of simple hash solution is

$$\text{hash}(o) \bmod N$$

Where o stands for the object which is distributed into the cluster, N stands for the amount of the node within the cluster.

For the sake of more understandable description, it is assumed that there are 3 nodes in the cluster, the indexes of them are n_1, n_2, n_3 , and there are 10 objects should be distributed among the cluster which is $o_i, i \in [0,10)$.

In the simple Hash solution, firstly each object will be hashed into a distinguish hash value, the hash value v is calculated:

$$v_i = \text{hash}(o_i), i \in [0,10) \quad (1)$$

Next, the index n of node for each value should be stored in can be calculated:

$$n_i = v_i \% N, N = 3 \quad i \in [0,10) \quad (2)$$

Now each object is distributed into the cluster in the corresponding node.

But the problem is raised when there is even a single node which is added or removed to the cluster, namely the range value N is changed. According to the formula $n_i = v_i \% N$, the distribution value n_i of all objects originally stored in this cluster is changed. It means that once the change of node happen, all of objects stored in the cluster should be re-distributed. Obviously it is too expensive in the maintaining perspective on a production environment.

5.2 Consistent Hashing solution

One of the solutions for the problem occurs in the simple hash is Consistent Hashing.

5.2.1 Hash Ring

It is known that the hash function actually maps objects and caches to a number range which is $[0, 2^{32} - 1]$. In Consistent Hashing, the hash space is treated as a closed ring, called Hash Ring.

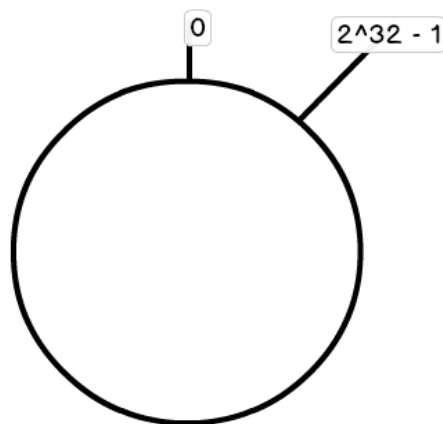


Figure 27. Hash Ring

5.2.2 Mapping of objects

As in the previous session, it is assumed that there are 3 nodes in the cluster, the index of them are n_1, n_2, n_3 , and there are 10 objects which should be distributed across the cluster which are $o_i, i \in [0,10)$.

Firstly each object is hashed into a distinguished hash value, and each object is mapped in the Hash Ring. The hash value v is calculated:

$$v_i = \text{hash}(o_i), i \in [0,10) \quad (1)$$

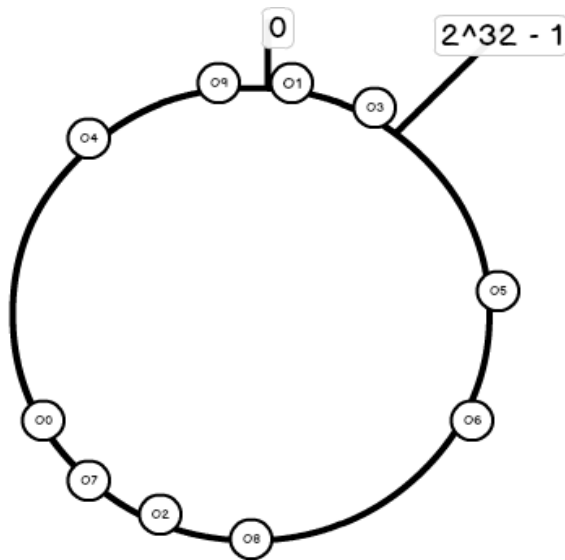


Figure 28. Objects are mapped in the Hash Ring

5.2.3 Mapping of nodes

In the same way as objects, each node is also hashed to a hash value by providing its identification value, for example IP address + port. Next, each node is mapped in the Hash Ring. The hash value V is calculated:

$$V_j = \text{hash}(n_j), j \in \{1,2,3\} \quad (2)$$

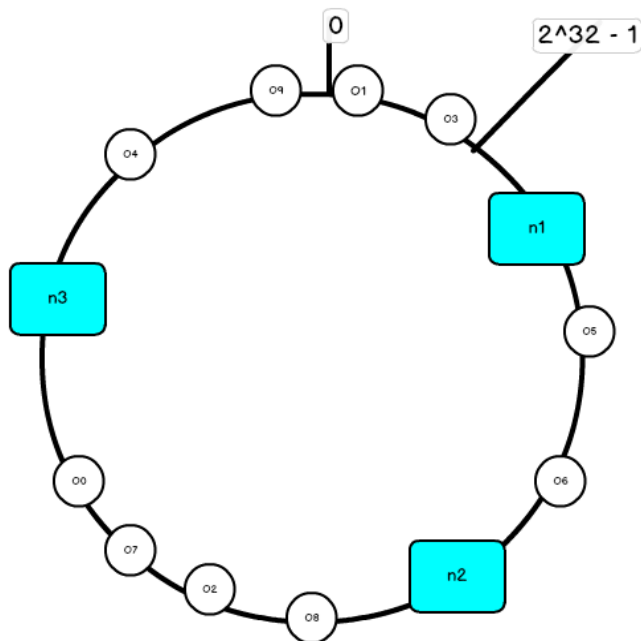


Figure 29. Nodes and Objects are mapped in the same Hash Ring

In this phase, it can be observed that nodes and objects are crossly mapped in the Hash Ring. Next, each object will be stored in the node which is closest to it in the clockwise:

$$n_i = v_i < V_j(\text{clockwise}), j \in \{1,2,3\} i \in [0,10) \quad (3)$$

Where the operator $<$ stands for the operation that, finding the nearest node V_j on the Hash Ring in clockwise for v_i .

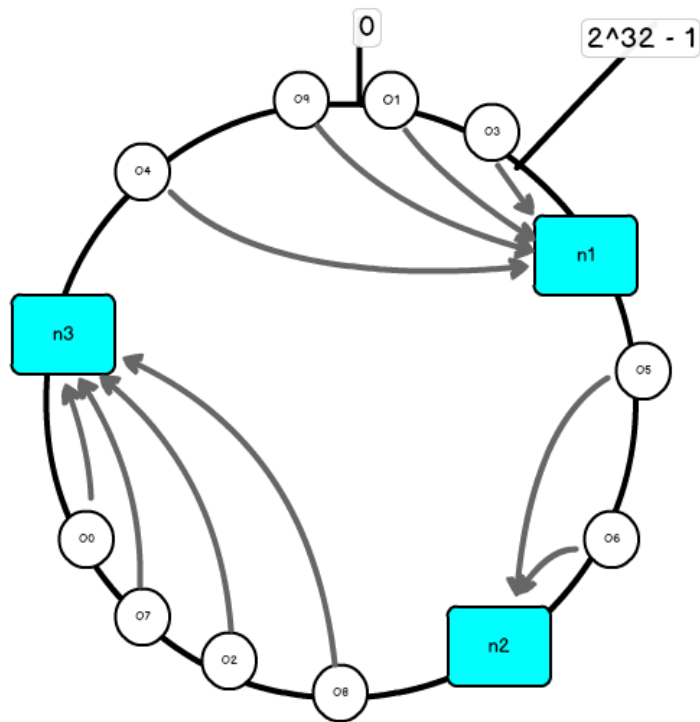


Figure 30. Each object is stored in the node by Consistent Hashing

The distribution of objects in the cluster can be observed:

$$n_1 = [1, 3, 4, 9],$$

$$n_2 = [5, 6],$$

$$n_3 = [0, 2, 7, 8]$$

5.2.4 Adding and removing of nodes

Consistent Hashing is a good solution for reducing the data migration when nodes added or removed from the cluster. For example, it is assumed that n_2 is removed from the cluster.

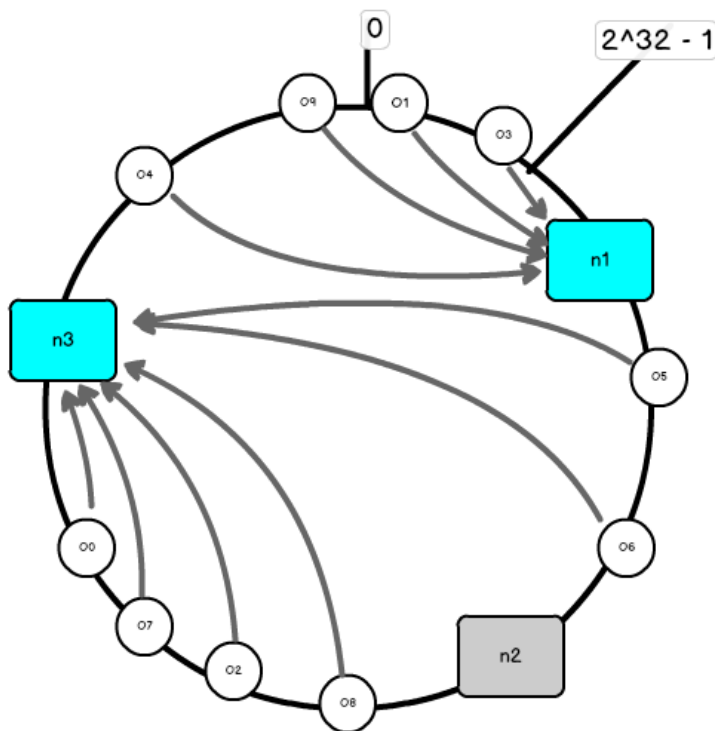


Figure 31. One node is removed from cluster

It can be observed that there are only objects o_5, o_6 which are originally stored in n_2 should be reallocated in n_3 , the others remains in same node. Therefore roughly there are $\frac{1}{N}$ fraction of data is influenced.

Next, it is assumed that there is a new node n_4 which is joined into the cluster. The V for the new node is also calculated: $V_4 = hash(n_4)$, and n_4 is mapped in the Hash Ring via V_4 .

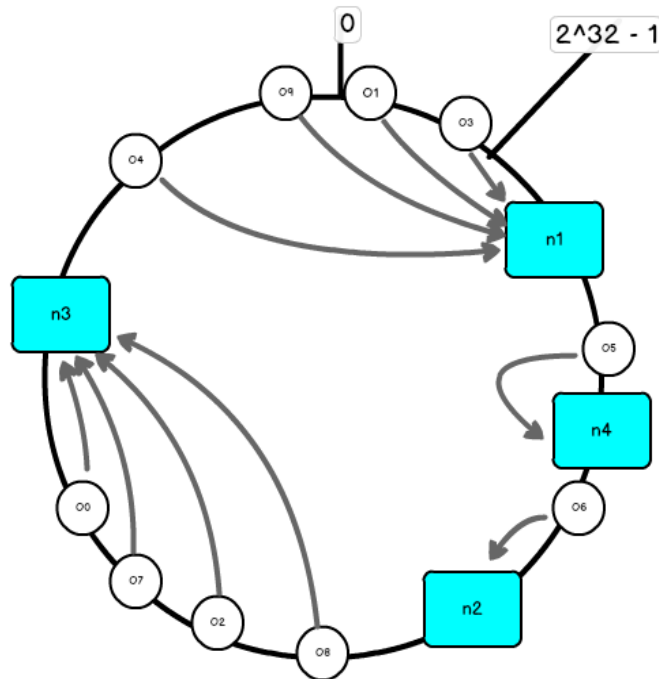


Figure 32. One node is added to cluster

It can be observed that when n_4 is joined, for which only object o_5 which is originally stored in n_2 should be reallocated in n_4 , the others remains in same node. Therefore there are roughly $\frac{1}{N+1}$ fraction of data which is influenced.

5.2.5 Virtual node

It is easy to find from the previous description that, the node and objects are randomly distributed in the hash space, the load balance among the nodes is not guaranteed. For instance, in the production environment, it is very possible that there are two nodes, n_x and n_{x+1} , which are very close, in this case most of the objects near them in clockwise will be stored in the n_x and n_{x+1} and gets rarely the chance. For enhancing the balance among the nodes, the concept of “Virtual node” is introduced. The idea is to replicate each node by multiple virtual nodes and these virtual nodes are spread in the hash space instead of the real node. When the object is matched with a virtual node, it will actually be stored in a corresponding real node. There are multiple ways to create virtual nodes for a node, for example, one of the easiest solutions can be adding a suffix to the identification of a node, as follows:

host: port#1

host: port#2 ..

And the hash value V_i of the virtual node for a node n can be calculated by:

$$V_{vi} = \text{hash}(n + s_i)$$

Where s stands for suffix.

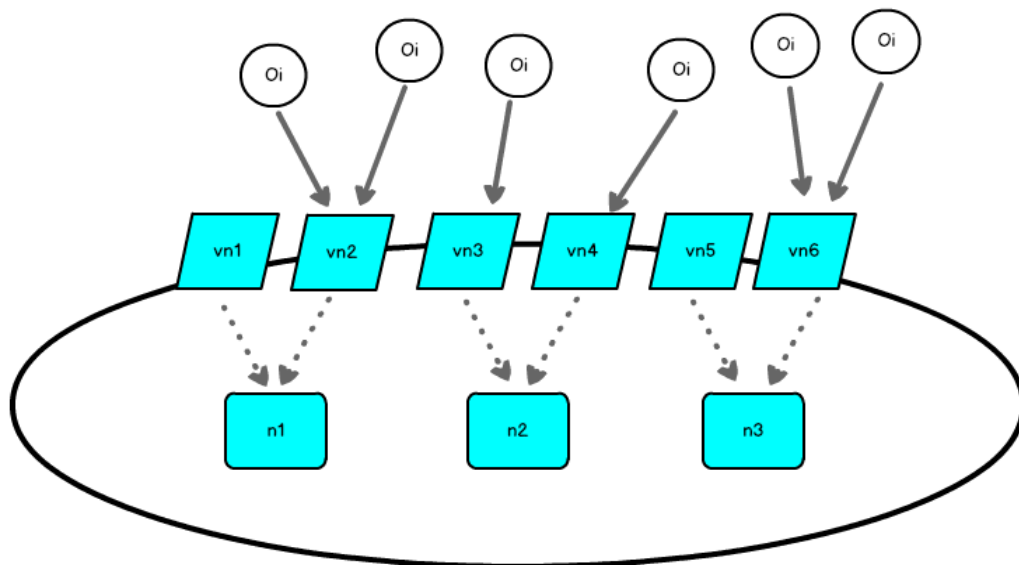


Figure 33. Virtual node

5.3 Pseudo-code of Consistent Hashing

```
BEGIN
numberOfReplicas <- number // Number of virtual nodes
circle <- HashMap //Hash Ring

/*Initiating by adding initiate nodes*/
INITIAL
{
  REPEAT
    add(node)
```

```

UNTIL all initial nodes are added
}

/*Function for adding node to Hash Ring*/
FUNCTION addNode(node)
{
  /*Adding Virtual nodes instead of real node*/
  index <- 0
  REPEAT
    /*Generating virtual nodes hash*/
    hash <- HASH(node.Identification +"#" +index)
    circle.put(hash, node);
    index ++
  UNTIL index >= numberOfReplicas
}

/*Function for removing node to Hash Ring*/
FUNCTION removeNode(node)
{
  /*Removing Virtual nodes*/
  index <- 0
  REPEAT
    hash <- HASH(node.Identification +"#" +index)
    circle.remove(hash, node);
    index ++
  UNTIL index >= numberOfReplicas
}

/*Function for setting object into cluster*/
FUNCTION setObject(key, object)
{
  hash <- HASH(key)

  /*Get subset of circle which hash values are bigger
  than the hash of incoming key*/
  tailCircle <- circle.getTail(hash)

  /*If the subset is empty, the nearest node in clockwise
  for incoming key should be the first node;
  otherwise, it should be first node in the subset*/
  IF tailCircle is empty
  {
    node_hash <- circle.firstKey()
  }
  ELSE
  {
    node_hash <- tailCircle.firstKey()
  }
}

```

```
}

/*Getting the node and saving the object*/
node <- circle.get(node_hash)
node saves the key:object
}

/*Function for getting object into cluster*/
FUNCTION getObject(key)
{
  hash <- HASH(key)
  tailCircle <- circle.getTail(hash)
  IF tailCircle is empty
  {
    node_hash <- circle.firstKey()
  }
  ELSE
  {
    node_hash <- tailCircle.firstKey()
  }

  /*Getting the node and getting the object*/
  node <- circle.get(node_hash)
  object <- node gets object by key
  RETURN object
}

END
```

6. MICROSERVICE

The analytic cluster for which this research is performed is constructed with microservice architecture based on the Spring Boot. Within a microservice system, each service is a small building block which is highly decoupled and focused on a specific task. And these services communicate with each other by a given approach, for example in this thesis, it is REST. The cache layer discussed in the thesis is also built as a service in the microservice system in target IoT system.

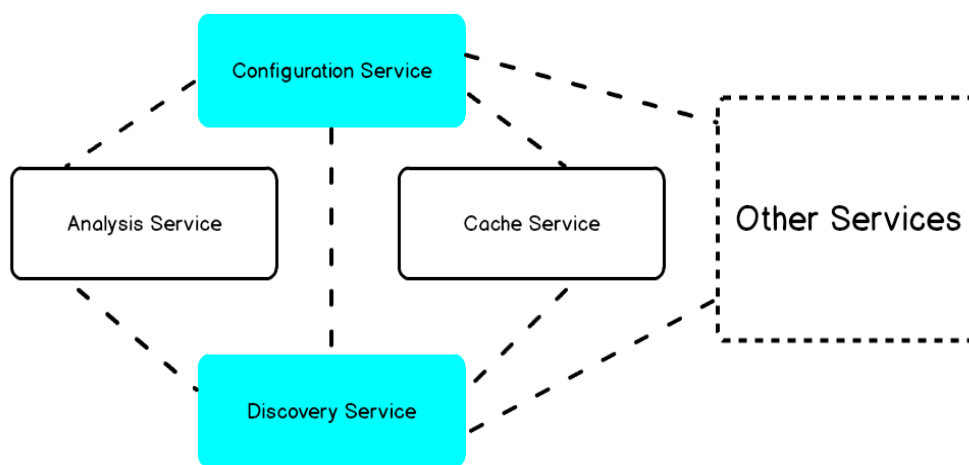


Figure 34. Brief microservice system structure

Since the cache service is only one part of the microservice system, the microservice will be briefly introduced in this thesis.

6.1 Configuration service

It is introduced by the twelve-factor app methodology that configurations for a microservice application should be stored in the environment, not in the project./21/ The configuration service is introduced for this purpose. Configuration service is a fundamental component in the microservice architecture, it works for the management of configuration of the system in running time.

In the involved microservice system in this study, the configuration is stored in several files in the dedicated Git repositories, when the configuration service is

started up, it will reference the path to those configuration files and begin to serve them up to the microservices that request those configurations. In this pattern, the configuration is both externalized and centralized in one place that can be version-controlled and revised without restarting if a configuration needs to be modified. In Spring Boot solution, when the change of configuration is made, a signal of refresh to discovery service can be issued that informing all microservices to update their configuration.

6.2 Discovery Service

The discovery service is also a core component of the microservice architecture. All of the instances of available service in the cluster are maintained by the discovery service. Eureka is adopted in the discussed microservice system for this purpose.

Eureka is a REST based service. The major functionality of it is locating services for the load balancing purpose and failover of middle-tier servers, which provides the solution to automatically discover and connect to other services within the cluster./22/

6.3 Cache Service

The cache layer which is developed in the study of this thesis is built as a microservice in the discussed microservice system. One of the salient advantages of its flexibility. It makes the cache strategy like Proactive Caching can be easily started and terminated without influencing other services. Furthermore, the extension like the deployment of multiple different cache service with different cache strategy can be more applicable in further development based on the flexibility provided by the microservice architecture.

7. TESTING AND ANALYSIS

Since the limitation of the current phase of development that the actual distributed environment for hosting the cache cluster is not ready yet, the tests described in this chapter were performed in the development machine. Redis version 2.8 is adopted.

7.1 Cache strategy test

Since there are several limitations including: the actual server for hosting cache cluster is not available yet, the cache service has to be launched from local development machine as the same as the test client and the network condition is not very stable. A precise testing does not seem applicable. Therefore a small and rough test is performed instead, and the result should be clear enough to conceptually prove the effectiveness of the cache strategy.

A cache strategy test was performed in development machine which hosted the virtual cache cluster and connected to the center database cluster for development use in Wapice Ltd. A 500 pieces scale of data set was used for testing. A Postman client was used as simulated client in the test.

In the test, the client continuously sent a request to the cache service querying data in a given time period (30 days) of a sensor, and the time consumptions in below 3 cases were collected, which are:

1. Query without cache strategy deployment
2. Query with partial cache strategy deployment (no “placeholder”)
3. Query with full cache strategy deployment

The request for each case was performed 10 times respectively.

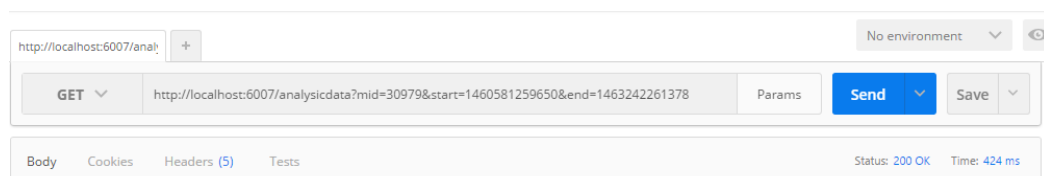
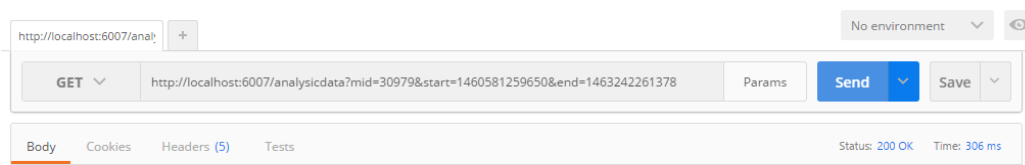
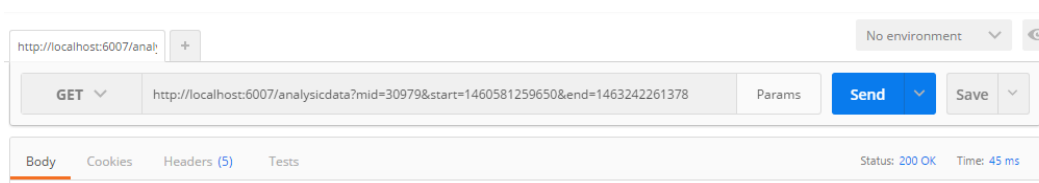


Figure 35. Query without cache strategy deployment**Figure 36.** Query with partial cache strategy deployment**Figure 37.** Query with full cache strategy deployment

The collected data of the test was time consumption, the unit is ms (millisecond), and it was collected in the table below:

Without cache strategy deployment(ms)	With partial cache strategy deployment (ms)	With full cache strategy deployment (ms)
455	269	43
424	498	48
434	344	41
439	373	42
411	322	40
416	382	39
389	357	40
387	363	42

397	358	45
433	365	42

Table 6. Collected data from the strategy test

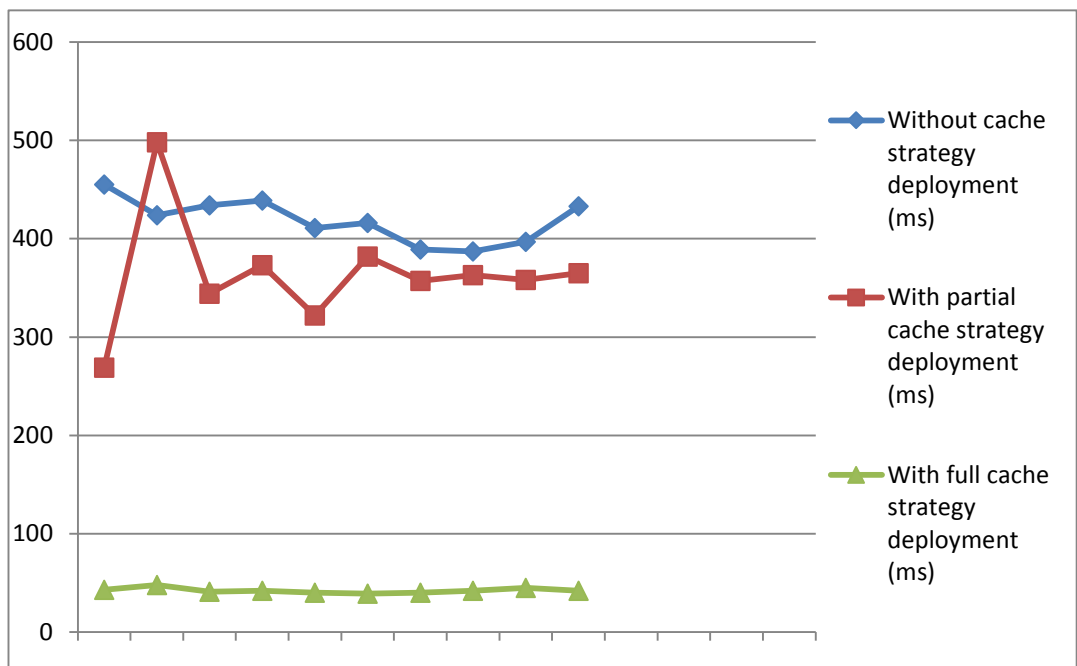


Chart 1. Collected from strategy test data in line chart

From the data in the table 6, the average time consumption in each case can be calculated:

	Without cache strategy deployment	With partial cache strategy deployment	With full cache strategy deployment
Average time consumption(ms)	418.5	363.1	42.2

Table 7. Average time consumption of strategy test

It can be observed from both average time consumption and the line chart of collected data that, generally the time consumption in cases of without cache strate-

gy T_n , with partial cache strategy deployment T_p , with full cache strategy deployment T_f is:

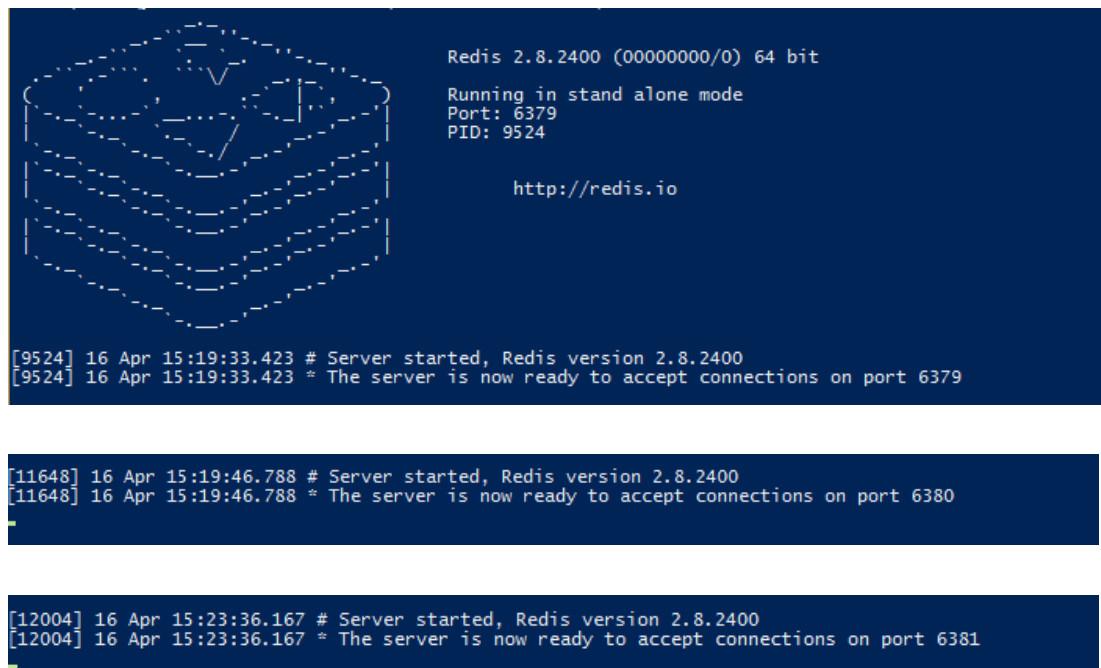
$$T_n > T_p > T_f$$

- Result

The cache strategy and the placeholder optimization developed in the study of this thesis effectively reduce the time consumption of the data query. And it can be deduced that the load of center database can be reduced.

7.2 Master sub-cluster test

In the test of Master sub-cluster, there were 3 master nodes were up and running on the port 6379, 6380 and 6381 which formed the master sub-cluster.



```

Redis 2.8.2400 (00000000/0) 64 bit
Running in stand alone mode
Port: 6379
PID: 9524

http://redis.io

[9524] 16 Apr 15:19:33.423 # Server started, Redis version 2.8.2400
[9524] 16 Apr 15:19:33.423 * The server is now ready to accept connections on port 6379

[11648] 16 Apr 15:19:46.788 # Server started, Redis version 2.8.2400
[11648] 16 Apr 15:19:46.788 * The server is now ready to accept connections on port 6380

[12004] 16 Apr 15:23:36.167 # Server started, Redis version 2.8.2400
[12004] 16 Apr 15:23:36.167 * The server is now ready to accept connections on port 6381

```

Figure 38. 3 master nodes were up running in development machine

In the test, 10 keys with value 0 to 9 were generated and written into the sub-cluster of master nodes. After the write operation, the 10 written values were read and printed with the port number of the node in which the value was kept.

The value for the key is value is 0
The port of the node keeping this key: 6379

The value for the key is value is 1
The port of the node keeping this key: 6379

The value for the key is value is 2
The port of the node keeping this key: 6381

The value for the key is value is 3
The port of the node keeping this key: 6379

The value for the key is value is 4
The port of the node keeping this key: 6381

The value for the key is value is 5
The port of the node keeping this key: 6381

The value for the key is value is 6
The port of the node keeping this key: 6380

The value for the key is value is 7
The port of the node keeping this key: 6381

The value for the key is value is 8
The port of the node keeping this key: 6380

The value for the key is value is 9
The port of the node keeping this key: 6380

Figure 39. Print of the test for master sub-cluster

From the print it can be found that the 10 values are distributed among the 3 master nodes. In the progress, when each data was inserted into the cluster, the hash of the key was calculated and the corresponding node for this data was matched by Consistent Hashing algorithm. It was proved that the master sub-cluster performs correctly.

- **Conclusion**

The Master sub-cluster performed correctly as design, test passed.

7.3 Slave node test

In the test of slave node, a slave node in development machine which replicates the master node at 127.0.0.1 port 6380 was involved. After the slave node started, the activities had been done were shown in the print from both master node and slave node.

It should be noticed that in this test, the PID of master node were 15964 and 7896 (after restarting) and the PID of slave node were 22008 and 20384. The PID can be used to distinguish the master and slave nodes in the console print.

The print from slave node:

```
[15964] 12 May 11:08:03.387 # Server started, Redis version 2.8.2400
[15964] 12 May 11:08:03.387 * DB loaded from disk: 0.000 seconds
[15964] 12 May 11:08:03.398 * The server is now ready to accept connections on port 16380
[15964] 12 May 11:08:04.387 * Connecting to MASTER 127.0.0.1:6380
[15964] 12 May 11:08:04.387 * MASTER <-> SLAVE sync started
[15964] 12 May 11:08:04.387 * Non blocking connect for SYNC fired the event.
[15964] 12 May 11:08:04.389 * Master replied to PING, replication can continue...
[15964] 12 May 11:08:04.389 * Partial resynchronization not possible (no cached master)
[15964] 12 May 11:08:04.396 * Full resync from master:
1d2c3b3580ffd1523bcfe8baa5faea6c0cf0bb44:253
[15964] 12 May 11:08:04.577 * MASTER <-> SLAVE sync: receiving 18 bytes from master
[15964] 12 May 11:08:04.581 * MASTER <-> SLAVE sync: Flushing old data
[15964] 12 May 11:08:04.581 * MASTER <-> SLAVE sync: Loading DB in memory
[15964] 12 May 11:08:04.582 * MASTER <-> SLAVE sync: Finished with success
```

The print from master node

```
[22008] 12 May 11:08:04.391 * Slave 127.0.0.1:16380 asks for synchronization
[22008] 12 May 11:08:04.391 * Full resync requested by slave 127.0.0.1:16380
[22008] 12 May 11:08:04.391 * Starting BGSAVE for SYNC with target: disk
[22008] 12 May 11:08:04.396 * Background saving started by pid 18824
[22008] 12 May 11:08:04.548 # fork operation complete
[22008] 12 May 11:08:04.574 * Background saving terminated with success
[22008] 12 May 11:08:04.578 * Synchronization with slave succeeded
```

It can be found that after initiating, the slave node raised an event of SYNC and tried to connect to the master node. The Master node answered the request and a full RE-SYNC was performed. The SYNC finished with success and slave node received the data in master node as replication.

7.3.1 Insert to master and read from slave test

In this test, a piece of data “msg - solid” was inserted into master node at port 6380 and it could be read from slave node at port 16380.

Print from client of master node:

```
127.0.0.1:6380> get msg
(nil)
127.0.0.1:6380> set msg solid
OK
127.0.0.1:6380> get msg
"solid"
```

Print from client of slave node:

```
127.0.0.1:16380> get msg
"solid"
```

In the progress, when a new data was inserted into the master node, it was synchronized to the slave node in a given interval. After that, this data was available in the slave node and could be read.

- Conclusion

A piece of data inserted in the master node, could be read from slave node, test passed.

7.3.2 Consistency test

In this test, the master node was turned down. When the master node was turned down, the slave should continuously answer the read request from client with its own dataset.

Firstly, the master node was terminated which simulated a master failure:

```
[22008] 12 May 12:27:49.639 # User requested shutdown...
[22008] 12 May 12:27:49.639 # Redis is now ready to exit, bye bye...
```

The slave node reported the error of disconnection of master node:

```
[15964] 12 May 12:32:47.934 * Connecting to MASTER 127.0.0.1:6380
[15964] 12 May 12:32:47.934 * MASTER <-> SLAVE sync started
[15964] 12 May 12:32:48.936 * Non blocking connect for SYNC fired the event.
[15964] 12 May 12:32:48.936 # Sending command to master in replication handshake: -Writing to
master: Unknown error
```

The client sent a read request, and got the data returned.

```
127.0.0.1:16380> get msg
"solid"
```

In the progress, when the master server was down, the slave node noticed that since the master node had not been answering the PIN for a given period. But the slave node still answered the read request according to the configuration.

- Conclusion

When the master node is turned down, the slave continuously answered the read request from the client with its own dataset, test passed.

7.3.3 Consistency test 2

In this test, the slave was turned down. If the slave node was turned down, in a period when the slave node was not performing, the data updates in the master node will be synchronized to the slave node when it restarts.

Firstly, the slave node was terminated which simulated a slave failure:

```
[15964] 12 May 12:38:26.258 # User requested shutdown...
```

```
[15964] 12 May 12:38:26.258 # Redis is now ready to exit, bye bye...
```

The master node was informed about the disconnection of the slave node 127.0.0.1:16380:

```
[7896] 12 May 12:38:26.258 # Connection with slave 127.0.0.1:16380 lost.
```

Now the data with key “msg” was updated:

```
127.0.0.1:6380> set msg "this is a new message comes when slave is down"  
OK
```

Next, the slave node was restarted with PID 20384 and synchronized with the master node:

```
[20384] 12 May 12:48:20.433 # Server started, Redis version 2.8.2400
```

```
[20384] 12 May 12:48:20.434 * DB loaded from disk: 0.000 seconds
```

```
[20384] 12 May 12:48:20.435 * The server is now ready to accept connections on port 16380
```

```
[20384] 12 May 12:48:21.438 * Connecting to MASTER 127.0.0.1:6380
```

```
[20384] 12 May 12:48:21.439 * MASTER <-> SLAVE sync started
```

```
[20384] 12 May 12:48:21.439 * Non blocking connect for SYNC fired the event.
```

```
[20384] 12 May 12:48:21.440 * Master replied to PING, replication can continue...
```

```
[20384] 12 May 12:48:21.440 * Partial resynchronization not possible (no cached master)
```

```
[20384] 12 May 12:48:21.446 * Full resync from master:
```

```
09f133f14e2c32e8bc2f85940122c68f118a6b73:1247
```

```
[20384] 12 May 12:48:21.519 * MASTER <-> SLAVE sync: receiving 72 bytes from master
```

```
[20384] 12 May 12:48:21.522 * MASTER <-> SLAVE sync: Flushing old data
```

```
[20384] 12 May 12:48:21.522 * MASTER <-> SLAVE sync: Loading DB in memory
```

```
[20384] 12 May 12:48:21.523 * MASTER <-> SLAVE sync: Finished with success
```


The master node was informed of the rejoined slave node:

```
[20384] 12 May 12:48:20.433 # Server started, Redis version 2.8.2400
[20384] 12 May 12:48:20.434 * DB loaded from disk: 0.000 seconds
[20384] 12 May 12:48:20.435 * The server is now ready to accept connections on port 16380
[20384] 12 May 12:48:21.438 * Connecting to MASTER 127.0.0.1:6380
[20384] 12 May 12:48:21.439 * MASTER <-> SLAVE sync started
[20384] 12 May 12:48:21.439 * Non blocking connect for SYNC fired the event.
[20384] 12 May 12:48:21.440 * Master replied to PING, replication can continue...
[20384] 12 May 12:48:21.440 * Partial resynchronization not possible (no cached master)
[20384] 12 May 12:48:21.446 * Full resync from master:
09f133f14e2c32e8bc2f85940122c68f118a6b73:1247
[20384] 12 May 12:48:21.519 * MASTER <-> SLAVE sync: receiving 72 bytes from master
[20384] 12 May 12:48:21.522 * MASTER <-> SLAVE sync: Flushing old data
[20384] 12 May 12:48:21.522 * MASTER <-> SLAVE sync: Loading DB in memory
[20384] 12 May 12:48:21.523 * MASTER <-> SLAVE sync: Finished with success
```

Now at the client of the slave node, it could be found that the value with key “msg” was updated:

```
127.0.0.1:16380> get msg
"this is a new message comes when slave is down"
```

- Result

If the slave node was turned down, in the period when the slave node was not performing, the data updated in the master node was synchronized to the slave node when it restarted, test passed.

8. DISCUSSION AND FURTHER DEVELOPMENT

8.1 The failure tolerance in the cluster

In the Master – Master – Slave topology model which is discussed in chapter 4, the failure tolerance is patricidal in current solution.

In a typical Master – Slave topology model, when the master node is down, a slave node will be promoted to be the new master, and when the old master node restarts, it will be joined as a slave node. But in the discussed Master – Master – Slave topology model, since Redis does not inherently support Master – Master topology model, the connection and data sharding logic is deployed in the service layer. On the other hand, the Master – Slave topology model is deployed by using Redis in-built solution in database layer. When a slave node is down, the rest of the cluster still performs, but when a master node is down, the slave node cannot be promoted to be the new master node. Because there is not a path via which service layer can communicate with database layer for the information of change about master node in current solution. Namely, the list of master nodes maintained in the service layer is relatively static. Therefore, in the current solution, when a master node is down, its slave nodes will not be promoted to be new master, instead, they will continuously respond to the read request with existing data till the failed master node is restarted.

Further development in related perspective can be performed by looking for the solution that is providing full failure tolerance in the Master – Master – Slave topology model, in which when a master node is down, a slave node can be promoted as a new master node and registered in the master – master sub-cluster, and when the failed master node is restarted, it is quitted from the master – master sub-cluster and rejoined as a slave node.

8.2 Grouping cache data by different industry

In current data sharding solution, the distribution of the data is strictly following the Consistent Hashing algorithm which is full random. Therefore the data from the same industry could be stored in every node among the cluster.

There is point of optimization that, by further developing the Consistent Hashing or adding a separate calculation, making the data from sensors which are from same industry allocated in a same node or a given sub-cluster. This solution will further improve the performance of the cache.

8.3 Test in real distributed environment

Because of the limitation in the current development phase, the real distributed environment for hosting the cache service which is developed in this research is not ready yet. Therefore, the testing has to be performed in the local development machine by simulating.

In the further development, when the real distributed environment is ready, in which the deployment and test should be performed. A more precise result will be produced and more factors can be considered.

9. SUMMARY

This thesis introduces the development of a distributed cache strategy for analytic cluster in an IoT system in detail.

For constructing a high performance cache strategy for analytic cluster in specific IoT system environment, firstly the related data model in the IoT system is analyzed, by that the data model which will be used in the cache layer and optimized for the target analytic cluster is designed. Next, the cache strategy is introduced. LRU as a proper solution is discussed and deployed using Redis in-built solution as the first layer in the cache strategy. For optimizing the LRU with the character of the target IoT system, a concept of “placeholder” is introduced. It is proved to be effective in the testing session. In another perspective, for a better performance of cache when the near real time data query is required by the target analytic cluster, a specified Proactive Caching solution based on the Apache Kafka is developed. Then, a Master – Master – Slave topology model is constructed iteratively, which describes how the nodes of database server are connected and how do they collaborate. Following, Consistent Hashing is introduced to answer the question that how the data is partitioned into the cluster built so far. Next, the microservice is introduced which is the system architecture the development of the target IoT system follows and the studied cache layer is built as a microservice in the system. It brings advanced scalability and flexibility in multiple purposes. Finally, the major opponents in the study are examined and the several points of further development are discussed.

It can be concluded that the studied cache strategy is well designed and optimized for the specific analytic cluster deployed in the IoT system in current phase. Moreover, the study can be improved by more effective testing and optimizing to achieve a production ready level.

REFERENCES

- /1/ Wapice Ltd. Accessed 2.5.2016.
<https://www.wapice.com/about-us/wapice>
- /2/ IoT-Ticket. Accessed 2.5.2016.
<https://www.iot-ticket.com>
- /3/ Wikipedia. Cache (computing). Accessed 2.5.2016.
[https://en.wikipedia.org/wiki/Cache_\(computing\)](https://en.wikipedia.org/wiki/Cache_(computing))
- /4/ Wikipedia. Cache algorithms. Accessed 2.5.2016.
https://en.wikipedia.org/wiki/Cache_algorithms
- /5/ Paul, S; Z Fei. Distributed caching with centralized control. 2001. Computer Communications 24,(2),256–268
- /6/ Tim, Palko. High-Volume Data Collection and Real Time Analytics Using Redis. 2013. Accessed 3.5.2016.
<http://www.slideshare.net/cacois/cois-palkostrata2013>
- /7/ Kafka 0.9.0 Documentation. Accessed 3.5.2016.
<http://kafka.apache.org/documentation.html#introduction>
- /8/ Wikipedia. Microservices. Accessed 3.5.2016.
<https://en.wikipedia.org/wiki/Microservices>
- /9/ James, Lewis; Martin, Fowler. Microservice. 2014. Accessed 4.5.2016.
<http://martinfowler.com/articles/microservices.html>
- /10/ Kenny, Bastani. Building Microservices with Polyglot Persistence Using Spring Cloud and Docker. 2015. Accessed 4.5.2016.
<http://www.kennybastani.com/2015/08/polyglot-persistence-spring-cloud-docker.html>
- /11/ Spring Boot official site. Accessed 6.5.2016.
<http://projects.spring.io/spring-boot/>
- /12/ Paul,Chapman. Microservices with Spring. 2015. Accessed 6.5.2016.
<https://spring.io/blog/2015/07/14/microservices-with-spring>
- /13/ Redis documentation. Accessed 6.5.2016.
<http://redis.io/commands>
- /14/ Vinoo, Das. 2015. Learning Redis. 1st ED. 35 Livery Street, Birmingham B3 2PB, UK. Packt Publishing Ltd.

/15/ Aaron, Toponce. The Adjustable Replacement Cache. 2012. Accessed 6.5.2016.

<https://pthree.org/2012/12/07/zfs-administration-part-iv-the-adjustable-replacement-cache/>

/16/ Using Redis as an LRU cache. Accessed 6.5.2016.

<http://redis.io/topics/lru-cache>

/17/ Alexey, Ragozin. Data Grid Pattern - Proactive caching. 2011. Accessed 7.5.2016.

<http://blog.ragozin.info/2011/10/grid-pattern-proactive-caching.html>

/18/ Zookeeper. Accessed 7.5.2016.

<http://zookeeper.apache.org/>

/19/ Redis 2.8 configuration. Accessed 7.5.2016.

<http://download.redis.io/redis-stable/redis.conf>

/20/ David Karger; Eric, Lehman; Tom, Leighton; Matthew, Levine; Daniel, Lewin; Rina, Panigrahy. Consistent hashing and random trees: distributed caching protocols for relieving hot spots on the World Wide Web. 1997. STOC '97 Proceedings of the twenty-ninth annual ACM symposium on Theory of computing. 10,654-663

/21/ Adam, Wiggins. Twelve-factor app methodology. 2012. Accessed 10.5.2016.

<http://12factor.net/config>

/22/ Eureka git repository. Accessed 10.5.2016.

<https://github.com/Netflix/eureka>

Appendix 1

Java Implementation of Consistent Hashing

```

import java.util.Collection;
import java.util.SortedMap;
import java.util.TreeMap;

public class ConsistentHash<T> {

    private final HashFunction hashFunction;
    private final int numberOfReplicas;
    private final SortedMap<Integer, T> circle =
        new TreeMap<Integer, T>();

    public abstract boolean add_object_to_node(T Node,
Object key, Object value);
    public abstract Object get_object_from_node(T Node,
Object key);

    public ConsistentHash(HashFunction hashFunction, int
numberOfReplicas, Collection<T> nodes) {

        this.hashFunction = hashFunction;
        this.numberOfReplicas = numberOfReplicas;

        for (T node : nodes) {
            add(node);
        }
    }

    public void add(T node) {
        for (int i = 0; i < numberOfReplicas; i++) {
            circle.put(hashFunction.hash(node.toString() +
i), node);
        }
    }

    public void remove(T node) {
        for (int i = 0; i < numberOfReplicas; i++) {
            circle.remove(hashFunction.hash(node.toString() +
i));
        }
    }
}

```

```

public Boolean setValue(Object key, Object value) {
    T thisNode = find_node(key);
    if(thisNode == null)
        return false
    else
        return add_object_to_node(thisNode, key, value);
}

public T getValue(Object key) {
    T thisNode = find_node(key);
    if(thisNode == null)
        return null
    else
        return get_object_from_node(thisNode, key);
}

public T find_node(Object key){
    if (circle.isEmpty()) {
        return null;
    }
    int hash = hashFunction.hash(key);
    SortedMap<Integer, T> tailMap = circle.tailMap(hash);
    int node_hash = tailMap.isEmpty() ? circle.firstKey() : tailMap.firstKey();
    T thisNode = circle.get(hash);
    return thisNode
}
}

```