ADDRESSING NON FUNCTIONAL REQUIREMENTS THROUGH SYSTEM RE-ENGINEERING

By

ANKIT JAIN

(Under the Direction of Krzysztof J. Kochut)

ABSTRACT

Scalability, availability and failover are some of the major problems of large software systems today. The Web is growing at a fast pace and websites are experiencing high traffic volumes and often are not able to scale up to meet the increased demand. This thesis presents an approach to making a Web-based system more available and scalable to the end-users. Our solution has been based on reusing the existing code and architecture to create a distributed and scalable environment to handle high load levels. We show how we applied open source technology to a deployed application to make it more available and scalable. Our aim was to re-engineer the system in such a way so that it would fulfill the new non-functional requirements through its refactoring. Additionally, we introduced support for failover to the entire application. We have re-architected the existing system to achieve the stated performance goals without a complete system redesign. Subsequently, we have conducted performance comparison of the re-engineered system with the original one, which showed that the improved system achieved the stated performance requirements.

INDEX WORDS: Distributed Systems, Performance, Scalability, Availability, Failover, Re-engineering, Non-Functional Requirements.

ADDRESSING NON FUNCTIONAL REQUIREMENTS THROUGH SYSTEM RE-ENGINEERING

By

ANKIT JAIN

B.E. Institute of Engineering and Technology, India 2007

A Thesis Submitted to the Graduate Faculty of The University of Georgia in Partial

Fulfilment of the Requirements for the Degree

MASTER OF SCIENCE

ATHENS, GEORGIA

2012

ADDRESSING NON FUNCTIONAL REQUIREMENTS THROUGH SYSTEM RE-ENGINEERING

By

ANKIT JAIN

DEDICATION

This thesis is dedicated to my elder brother Ajay Jain who is the source of my inspiration. This thesis is dedicated to him for his support, encouragement and belief in me. He is the reason I am writing this thesis.

ACKNOWLEDGEMENTS

First, I would like to thank Ned Kalinovic for being my mentor, for his guidance support and encouragement. I mean he really helped and supported me by coming out of the box. He is like biggie in our company but whenever I go to him for help, guidance and support, he always snatch the time out from his busy schedule for me. This thesis is totally dedicated to him. He gave me the opportunity to work on some awesome projects and during this course of time I learned a lot from him, every time I go to him I come like all fired up and always get my basket filled with some or other knowledge. Secondly, I would also like to thank my major professor Dr. Krys Kochut for his constant support through the past few years. He always supported me and he gave me an open opportunity to pick something or some topic or project at work, which has a potential to be considered as a masters thesis, and eventually his idea and advice worked. I would also like to thank Dr. Budak Arpinar and Dr. Thiab Taha for consenting to serve on my committee. I would also like to thank my teammates who supported me through all these knowingly and unknowingly. I am also thankful to my mother, elder brother and my beloved wife for all their unconditional love, support and encouragement. Last thanks to my friend Ankit Singh Sengar for supporting me in all endeavours of my life since we met. He is such an incredible friend and person.

TABLE OF CONTENTS

LIST OF TABLES

Page

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

1.1 CHARACTERISTICS OF ONLINE SYSTEM

A large amount of research has been conducted in software re-engineering within the past several years. Many techniques and tools have been implemented to support re-engineering and system reconstruction. There are usually many reasons for organizations to perform software re-engineering or reconstructions. Some of the examples include:

- Addressing non-functional requirements (NFRs).
- Re-documenting the architecture of existing system.
- Improving the quality of the system.
- For outsourcing situations.
- Providing system's service to other clients.

At present, concept of live online bidding of the merchandise or services while seating on a chair with a mouse click and getting it shipped to the home drives the attention to the concept called online auctioning system. These days system provides a turnkey solution that includes a team of in-house buyers that assess the seasonality and popularity of the auction items, while also incorporating any of the customized preferences. System tracks review and provide enhancement recommendations, ensuring we meet and exceed desired program goals.

The more interesting point is the concept of extended time auctioning. Extended play auctions have a unique feature that resets the timer by 5-10 seconds, when more than one unique bid is placed during the final seconds. So this makes the auction more interesting when multiple users are trying to win it. In the final seconds they are likely to bid thousands of times and this makes this whole online auction system more interesting and engaging.

1.2 ONLINE AUCTION SYSTEM

An online Auction System is customizable; it increases the rewards currency (rewards points, Airline miles or points) demands for any company, which offers rewards program for their client. The system can have wide variety of prizes to engage end customers; system can also analyze and report from anything to everything related to each and every single item auctioned at a particular date and time. This system is exciting and engaging. It drives the end users interest and makes them to earn more and more rewards currency or points to participate in online auction. This further increases the sales of that particular client and drives more and more users to enroll and engage into the Rewards Program.

The online auction system we discuss in this thesis is novel (In fact there are not many live auctions applications available where thousands of users can bid every second, placing thousands of bids.

In our work, an initial the Online Auction System has been developed for a limited number of users. When it was being developed the focus was only on functional requirements and to accomplish and develop the system as soon as possible and the non-functional requirements were neglected entirely. However, the Non-functional requirements should be given equal importance as the functional requirements. Later when this online system was subjected to high loads, we realized that neglecting or not counting the non-functional requirements was a huge mistake, because at high demand loads, the system lacked scalability and availability. We noticed degradation in the system performance with more users coming in and using the system. Scalability, availability, performance, and efficiency become critical issues, which needed to be addressed. The system needs to be re-engineered to address all the lacking non-functional requirements. Re-engineering should focus on addressing non-functional requirements like scalability, availability, improved performance and efficiency, speed, distributed load and last but not least is to have failover.

We started our study and research about how to re-engineer the existing system to address the lacking non-functional requirements. After rigorous study, we learned that it could be done in multiple ways with using existing open source servers available at no cost, which could help in addressing the non-functional requirements and to improve performance.

In our case we are applying a NFRs driven approach to re-engineer the system to perform better and achieve list of NFRs (scalability, availability, failover and self distributed load mechanism). The goal of the system is to make system more available and scalable. We also wanted to have failover and distributed load mechanism in the re-engineered system. So our research work revolved to address the lacking NFRs and to have failover. So we re-engineered the system with the introduction of open source Redis which took care of lacking NFRs and also gave us other bonuses with less developing effort and cost.

The rest of the thesis is organized as follows. Chapter 2 includes background, necessary techniques, frameworks or any other information needed to understand the project. It also describes the Redis server and the Redis cluster we used. It explains how Redis can be used and configured with the existing system to add failover and scalability. Chapter 3 discusses closely related work and research being done in that needed context. In Chapter 4, we detailed our work to the design part of the system and also talk about re-architecture being done. Chapter 5 describes how we introduced the Redis server to the system and explains the previous system and then shows how we solved the problem and the new system is described as well. Evaluation is presented in Chapter 6. Finally, Chapter 7 contains conclusion and future work.

CHAPTER 2

Background

In this section, we describe general concepts needed to understand the project. We explain a little bit about non-functional requirements (NFRs) framework, system re-engineering.

2.1 SOFTWARE ARCHITECTURES

Software architecture of a computing system is the organization of the software system, which comprises software components, the properties of those components and relationship between them. It is a blueprint or plan that software will follow during the development and implementation. It is a overall design and make-up of the system.

2.1.1 MASTER SLAVE RELATIONSHIP

Master slave is a model of communication where devices communicate to other devices in the group and one among them is called Masters and rest are called as Slaves. In database language the master database is the main source to hold all data and the slaves are synchronized to it by required means and programming. Once the master – slave relationship is established, the main source to serve the system is always master but if something happens to master like failover or crash then slave is brought into picture and it changes its role from slave to master. Sometime this change of role is temporary until the master is running up back again but sometimes slaves' remains master until it fails or crashes. In our system we have the later master-slave relationship, where slaves becomes master when master fails and remain master until it fails and master will be back running up and it will become slave.

2.1.2 SYSTEM RE-ENGINEERING

System re-engineering is the modification of a software system to add new functionality or to correct errors. Reverse engineering is the initial examinations of the system and re-engineering is the subsequent modifications. Legacy system often needs to be re-engineered for various reasons. Re-engineering is frequently performed to improve maintainability.

There are no specific criteria for re-engineering a system. However, there are a few aspects that should be taken into an account before thinking of any type of transformation. [26] Paper talks about few patterns. First for the restructuring transformations, they have considered a ‘‘Primitive Structural’’ design pattern namely, the Composite design pattern that is the most popular one as it is used by four other design patterns and is refined by three other design patterns. The Composite pattern describes how to build a class hierarchy that is made up of different kinds of objects. In this category, another design pattern for re-architecting a software system into subsystems and helping minimize the communication and data flow dependencies between subsystems is the Facade design pattern. This pattern shields clients from the subsystem components, thereby reducing the number of objects that clients deal with and making the subsystems easier to use and maintain.

Software quality is a subjective term (for some its also includes NFR) and everybody has his or her own ideas of what quality is. So it is difficult to decide a unique list of criteria to re-engineer the software system. Main problem one has to face in quality and NFR driven re-engineering where only the non- functional requirements are addressed.

The goal of our research is not to re-engineer or reconstruct the every aspect of architecture but to provide the lacking failover mechanism and to address the non-functional requirements (scalability, availability, maintainability)

2.2 NON-FUNCTIONAL REQUIREMENTS (NFRs)

Due to enormous pressure towards deploying and releasing software as fast as possible, functional requirements gets all the focus of software development and implementations at the expense of NFRs such as performance, security, scalability and availability, etc. In enterprise applications NFRs got neglected to such a great extent that sometimes project leads to failure even though it completes all functional requirements. So organizations and development team have to stop focusing only on functional requirements and should give equal importance to NFRs, as well.

However, several definitions of NFRs exist. IEEE defines non-functional requirements as “ a software requirement that describes not what the software will do, but how the software will do". Many enterprises neglected the inclusion of NFRs during the software development life cycle; it is because normally satisfying one may affect another. Despite the very much importance of NFRs it is being neglected and left for verification after the complete implementation is done. And once the complete implementation is done than achieving those missed NFRs is a tedious task which gives rise to system re-engineering to address the set of NFRs specific to application.

It has been observed that many software projects focus on delivering software, which meets certain functionalities, whereas Non-Functional-Requirements are frequently neglected. Examples of NFRs, which are also known as Quality Requirements, can be reliability, scalability, availability, security, maintainability, portability and accuracy. Major problem of NFRs is that under different situations they can be interpreted differently and further it is not easy to formalize and generalize them. As software complexity and demands grows NFRs can no longer be ignored. NFRs should be considered as an important part of software development life cycle but many software companies have ignored that and now they are facing the issue with that which either leaves them with two choices. First, either design the new system from scratch with that NFRs been taking care of or re-engineer the existing system to take care of those with less human effort and cost. Proposed system is taking care of the later choice. Our work can be used as a case study to re-engineer the existing software application to address NFRs like scalability, availability, failover, distributed load and maintainability.

CHAPTER 3

Related work

Ideally we should address NFRs during the initial phase of software development life cycle. There are plenty of formal and informal (semi-formal) approaches to address NFRs depending on the software system. Adapting the informal is in great use, as it does not require a high human expertise. NFR framework is the most popular approach in these regards. The quality attributes taxonomy [24] is another semi-formal work in the list. Agile development to address the specification and testing of performance is an important type of NFR. After each sprint or iteration development team can identify and specify performance requirements incrementally, which eventually leads to desired level of detail description.

3.1 NON-FUNCTIONAL REQUIREMENT (NFR) FRAMEWORK

The NFR Framework [24] is one significant step towards making the relationships between quality requirements and design decisions explicit. In the NFR Framework, quality requirements are treated as potentially conflicting or synergistic goals that need be achieved, and are used to model and rationalize the various design decisions to be taken during system/ software development. Accordingly, the NFR Framework introduces the concept of soft-goals whose achievement is judged by the sufficiency of contributions from other (sub) soft-goals. In this context, a soft-goal interdependency graph is used to support the systematic modeling of the design rationale. The first step is to identify the NFRs and then main NFRs are treated as soft goals to be achieved. The framework uses NFRS in order to support architectural design and to model the impact of design alternatives. Given a quality constraint for a re-engineering problem, one can look up the soft-goal interdependency graph for that quality, and examine how it relates to other goals, and what are additional transformations that may affect the desired quality positively or negatively. Transformations are also represented as soft-goals, which are fulfilled when they are included in the re-engineering process.

The problem of coping with NFR framework during re-engineering has been experimentally tackled by developing a number of tools that met particular quality requirements. First, a tool has been developed to perform the re-engineering task, then a trial-and-error strategy was used to select a particular set of transformations which ensured that the re-engineered code satisfied given quality constraints.

However, not much effort has been made to provide a solution for large enterprise applications to re-engineer the existing system lacking some non-functional requirements and to re-engineer the existing product at architectural level. In this context we believe that our case study will help organizations to re-model their existing system at the architectural level to overcome set of non-functional requirements (scalability, availability, maintainability and failover with distributed load among servers)

3.2 SYSTEM RE-ENGINEERING AND RE-FACTORING

In software re-engineering a system is restructured to conform to satisfy Functional and Non-Functional Requirements (NFR). However the main part of software reengineering is still driven by the Functional Requirements, although the NFRs are just as crucial to the success of the system and it should be taken into account as serious as functional requirements. [Rebecca Tiarks] A lot of research was done on the field of quality-driven refactoring but there is no systematic way of building quality into software automatically. One main problem is that each decision made in the development process typically affects more than just one quality issue. Further it is difficult to integrate these desired qualities into the re-engineering process.

It is necessary to model and identify non-functional requirements in such a way that their dependencies among each other should be clear so that it will be easy to establish out which refactoring plan will help to reach the desired NFR.

The main aim of system and software re-engineering is that it should address the needed requirements, which could be addressing functional or non-functional requirements. Original system should be analysed properly to define the re-engineering process. In the paper [24] they described the software re-engineering model that is driven by specific non-functional requirements (NFRs) and addresses issues related to the evolution of the system requirements and software architecture. There are several research areas, which should be investigated before re-engineering the system, some of them are:

- List of software re-engineering techniques, which address the particular software qualities and non-functional requirements.
- The impact of those techniques on the non-functional requirements.
- There should be a software metrics, which keeps track of re-engineering impact on each and every non-functional requirement.

The International Organizations for Standardization introduced taxonomies of quality attributes, which divide quality into six characteristics namely: functionality, reliability, usability, efficiency, maintainability and portability. Complementary to the product-oriented approaches, the NFR Framework [24] takes a process-oriented approach to dealing with quality requirements. The NFR Framework is one significant step towards making the relationships between quality requirements and design decisions explicit. The framework uses NFR to drive the design to support architectural design level and to deal with the changes.

As the re-engineering of the enterprise application has become a major concern in today's software industry, most re-engineering efforts were focused towards the analysis of quality, performance and security.

3.2.1 SOFTWARE REFACTORING PROCESS

To meet non-functional requirements (NFRs) an iterative process is necessary. Usually the goals cannot be reached by a single transformation, but by a sequence of transformations. The 4 steps defined in [25] were:

- Setting up a goal-reasoning model, quantifying satisfaction or denial attributes of each soft goal in a metric.
- Quantitatively measuring software qualities so as to establish which alternative soft goal should be applied first.
- Picking an effective refactoring method among various transformations that contribute to the claimed soft goal.

- Applying the selected refactoring technique, which leads to iterative evaluations, leading back to step 2. Repeat that step until the soft goal is reached.

3.3 OTHER ONLINE AUCTION SYSTEM

There are dozens of online auction systems where users can go online and use either their miles and rewards points or real money to bid on a product and win. For example, eBay has an online auction system, United Airlines has launched the online auction using their miles, and they offer exciting merchandise to their customers for the frequent travel miles they have accumulated. Some auctions systems extend auctions usually to eliminate sniping, and to extend the availability of the auction. In our system we allow user to increase the bid by only one increment to engage more and more user with a chance and hope of winning an auction (rather than outbid others who are willing to spend). So, this extended auction feature with only one incremental bid is the main feature of the system we are discussing here. This change increases the demand for this system to make it more scalable, available and performance efficient. That is why Redis has been introduced to overcome all these non-functional requirements (scalability, availability, efficiency) with failover mechanism.

CHAPTER 4

PERFORMANCE REQUIREMENTS

4.1 PROBLEM STATEMENT

Current auction module has some flaws when it comes to scalability, availability and failover. With the successful launch of auctions, and increased demands placed on our systems, we wanted to re-architect the auctions module to overcome the drawbacks it has when it is exposed to big clients with millions users. The re-architecture shall focus on scalability, availability, and try to maintain as many prototype components as possible. Current Auction architecture has no fail-over mechanism and load balancing, lacking distributed mechanism and scalability. Current system has only one auction manager handling the entire bid request, serving everything needed and if that goes down the whole application is crashed. So we were looking to modify the existing system to be intelligent enough to balance load, overcome failure, more scalable and available when exposed to bigger chunk of users bidding at the same time. So technically we want our Auction Module to be smart enough to equally distribute the load among them.

We want our system to be capable of:

- Multiple Auction Managers running at the same time.
- Single Authority for Auctions.
- Scalability (Splitting Queries and filtering at many levels)
- Availability (Multiple Web servers, load balance) (Master-Slave relationship)

4.2 IMPROVING PERFORMANCE VIA SYSTEM RE-ARCHITECTING

These days many organizations are dependent on computer aided software to improve performance because there is too much human effort involved to re-engineer the existing systems to address the required non functional requirements. [23] Paper presents a framework that allows specific NFR such as performance and maintainability to guide the re-engineering process. Understanding the architecture of an existing system assists in predicting the impact evolutionary changes may have on specific quality characteristics of the system [23]. In order to understand and to re-engineer the existing system, it is necessary to capture its current design, architecture and the relationship between its different entities of implementation. In sum, a preliminary model is required in order to document the existing system before actually start thinking of re-engineering the product.

For this research work, we were particularly interested to investigate design patterns and their relationships as a means to restructure an existing online auction system so that the new system conforms and addresses all non-functional requirements such as scalability, availability, distributed load and failover. For achieving this goal, we needed to develop a list of specific design patterns and refactoring operations that can be used to enhance specific software qualities during reengineering namely, performance, maintainability, scalability, availability and enhancements for the new re-engineered system.

4.3 DESIGN OF THE CURRENT SYSTEM

Initially online auction system was developed at the company (for confidentiality reasons name is not mentioned anywhere is the thesis document). The company provides incentive rewards program to their clients. Online auction is one of their developed modules. An existing online auction system is being re-engineered in order to conform to a quality requirement (i.e., scalability, availability, distributed load mechanism, and failover). After studying the code and the desired requirements, it has been concluded that the existing structure of the program makes the desired extension difficult to achieve, and that the application of some design patterns, or source code transformations would help to achieve the desired properties with the help of existing server, which could be implemented with the current system to overcome the missing NFRs. In this context, the aim was to provide support for the developer to decide what design patterns or transformations to apply towards achieving the specific quality requirement for the new system.

Initially, the Online Auction System was developed as a proof of concept prototype that could be used to gauge user interest in rewards auctions. With the successful launch of auctions, and increased demands placed on our systems, we were looking to re-architect the auctions module. The re-architecture had to focus on scalability, availability, and tried to maintain as many prototype components as possible. Current Auction architecture has no fail-over mechanism and load balancing, lacking distributed mechanism and scalability. Only one auction manager handles the entire bid request, serving everything and in case of its failure the whole application crashed. What we needed was multiple level of load balancing, multiple Auction managers running to equally distribute the load. We want Auction Manager to be smart enough to equally distribute the load among auctions managers.

We wanted our system to be capable of running multiple auction managers at the same time. It should have a single authority for auctions. System should possess Scalability (Splitting Queries and filtering at many levels) and Availability (Multiple Web servers, load balance) (Master-Slave relationship). So, we created a design, which involved the introduction of the Redis to the system. We chose Redis after researching a lot because it demanded less coding, less re-architecting, less time and most importantly thing is that it was an open source, so it would be free to use. We used the Jedis Java-based API to communicate with Redis but it accepts data in some different format so we have to format the data in Jedis Specific format which is converting the data into bytes and converting it back while reading from Jedis serialized Auction Hash.

4.4 MIDDLEWARE STORAGE SYSTEMS EVALUATION

We researched many middleware systems and techniques available in the market; some are Redis, VoltDB, JBoss-Cache, Memcached, TerraCotta, and Hazelcast. In this section we will explain that why we chose Redis on above all these.

4.4.1 VOLTDB

VoltDB is an in-memory database. It is a blazingly fast New SQL database system specifically designed to run on modern architectures based around fast, inexpensive servers connected via high-speed data networks. VoltDB is aimed at a new generation of database applications – real-time feeds, sensor-driven data streams, micro-transactions, low-latency trading systems – requiring database throughput that can reach millions of operations per second. Moreover, the applications that use this data must scale on demand, provide flawless fault tolerance and enable real-time visibility into the data that drives business value. VoltDB is more than an ultra-fast database but it does not provide us with the entire thing, which we looked for. It provides scalability but at the cost of database transactions even though its high speed. No failover is provided with VoltDB. Failover can be achieved but then we have to code to achieve that in case if we move forward with VoltDB. So we ruled out this option.

4.4.2 MEMCACHED

Memcached is a free & open source, high-performance, distributed memory object caching system, it is generic in nature, but intended for use in speeding up dynamic web applications by alleviating database load. Memcached is an in-memory key-value store for small chunks of arbitrary data (strings, objects) from results of database calls, API calls, or page rendering. Memcached is simple yet powerful. Its simple design promotes quick deployment, ease of development, and solves many problems facing large data caches. Its API is available for most popular programming languages. It is perfect for what we are looking for but it lacks failover mechanism and locking mechanism, which is the important key in this system.

4.4.3 TERRACOTTA

Terracotta is a commercial open source company. Big-Memory is one of the most successful commercial products of Terracotta. BigMemory stores “big” amounts of data in machine’s main memory for high-speed access.

Terracotta is one of the best systems for our needs (offers good technical support, failover, and multiple servers running together with synchronized data). However, it is expensive (they charge for each server) and since our organization is small, we are always careful about the money we spend. Consequently, we ruled out this.

4.4.4 JBOSS CACHE

JBoss Cache is designed to cache frequently accessed java objects in order to improve the performance of the applications. JBoss Cache’s goal is to provide enterprise-grade clustering solutions to Java-based frameworks, application servers or custom-designed Java applications. JBoss Cache is a replicated cache; state is always kept in synchronize form with other servers in the cluster. This makes any state stored in JBoss Cache resilient to server crashes or restarts, achieving high availability. JBoss Cache is an advanced, 'enterprise-grade' data grid solution, providing features such as transactions, eviction, and cache loading in addition to replication. JBoss cache also good provides us high availability; fewer database transactions, synchronized clustered servers, but lack the failover mechanism. As a result we have eliminated it from considerations.

4.4.5 HAZELCAST

Hazelcast is an open source clustering and highly scalable data distribution platform for Java, which is:

1. Lightning-fast; thousands of operations/sec.
2. Fail-safe; no losing data after crashes.
3. Dynamically scalable as new servers are added.
4. Easy to use included as a single jar.

Hazelcast stores data into a java heap, which is subject to garbage collection. As the heaps get bigger, garbage collection might cause the application to pause for significant amount of time, badly affecting the application performance. So, even with terabytes of cache in-memory with numerous updates, garbage collection will have almost zero effect, resulting in more predictable latency and throughput.

4.4.6 REDIS

Redis is an advanced persistent Key-Value Store, also referred as a data structure server since keys can contain strings, hashes, lists, sets and sorted sets. Redis is an in-memory advanced persistent key-value store.

<u>Benefits of Using Redis:</u>

- Multiple auction managers - clustered under a load balancer.
- All auction managers connect to redis master through the load balancer with stickiness. (Load balancer basically sticks to one server below it. It uses only that server till that goes down)
- Slave auto SYNCs with master after background data save by master.
- In case master redis server goes down, bid requests are sent to one of the slave. Slave marked is not a slave anymore
- Auction Updates are retrieved from the Slave
- Many Auction Managers can handle load efficiently – One auction manager to two had a significant increase in response time.
- Failover at Redis server.
- Redis is open-source and free to all users.
- It offers good technical support.
- It satisfies all our needs.

4.5 ARCHITECTURE MODIFICATION

In this section we will describe what re-architecture we have performed and how we incorporated redis to our system. In the previous system (see the Figure 1) the auction manager accessed the static auction hash, which communicated with the database and provide updates and access to Auction manager. So whenever a write request on that static hash was issued the complete hash acquired a lock and prevented further processing until lock is released, which resulted in decreased performance, deadlocks.

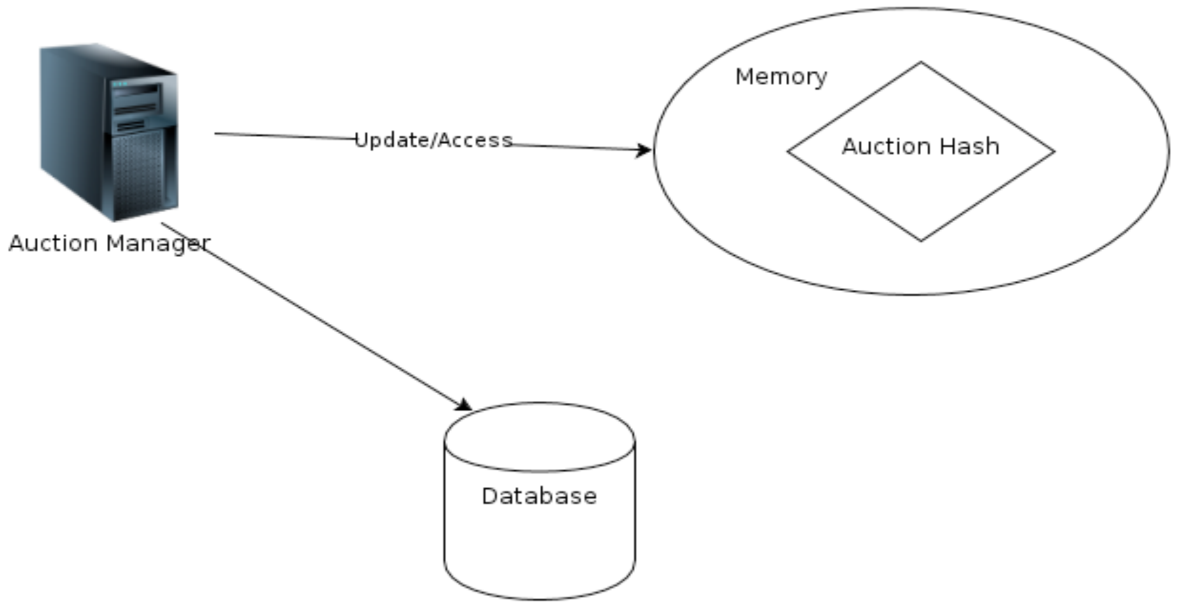


Figure 1: Previous architectural design of the system

Now we incorporated Redis (Jedis serialized Auction Hash Map) where we store all key-value pairs. The auction manager communicates with Redis via Jedis API, which is used specifically for Java based systems. Now the auction manager communicates with Auction manager client, who in turn communicate with database and get the latest required results and then the same auction manager client also convert that data into bytes for Jedis (see Figure 2). That conversion is called marshaling of data into Jedis specific format. While reading the data from Jedis by auction manager that data is again being converted into an array by un- marshaling.

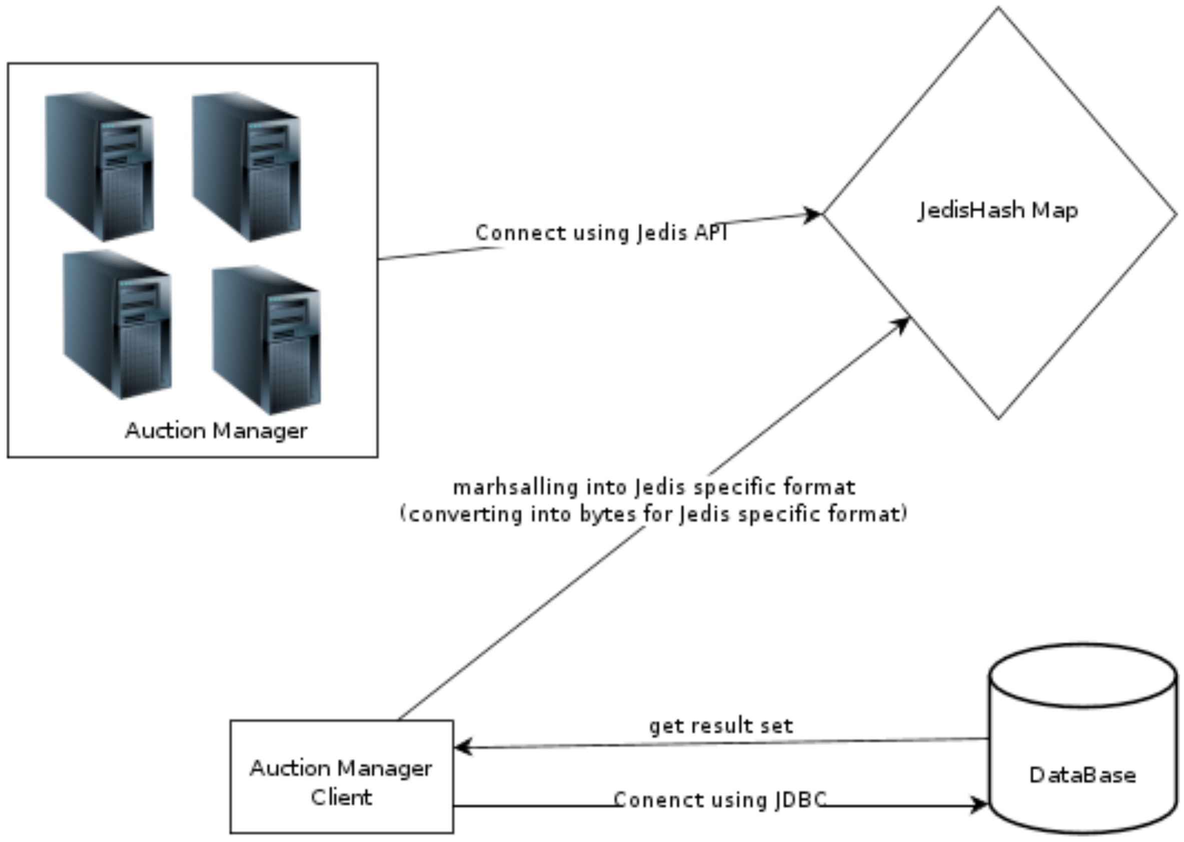


Figure 2: Re-architected design of the system (Redis Introduction)

We replaced the static hash map by the Jedis serialized Auction Hash Map, because the static hash map acquires a lock on the entire hash map and prevents any further access to the hash map until lock is released. However, with the Jedis serialized Auction Hash Map, we acquire a lock to a particular entry and all the remaining the entire hash map available for read and write. This way, the performance is improved and no reads and writes have to wait until the lock is released if it is other than that locked particular entry. So, by incorporating Redis we improved the responsiveness of the system. The system is faster, and more efficient and responsive than before.

CHAPTER 5

IMPLEMENTATION

5.1 PREVIUOS SYSTEM

Initially we offered this Auction Module to small clients and did not realize its shortcomings. As soon as we introduced it to high volume clients we realized that we needed more robust and scalable module. We had to create an architecture providing failover and most importantly multiple Auctions Managers running at the same time. We wanted to build our own backing mechanism but at this point we want to re-architect our system using a large number of servers, which could serve our purpose.

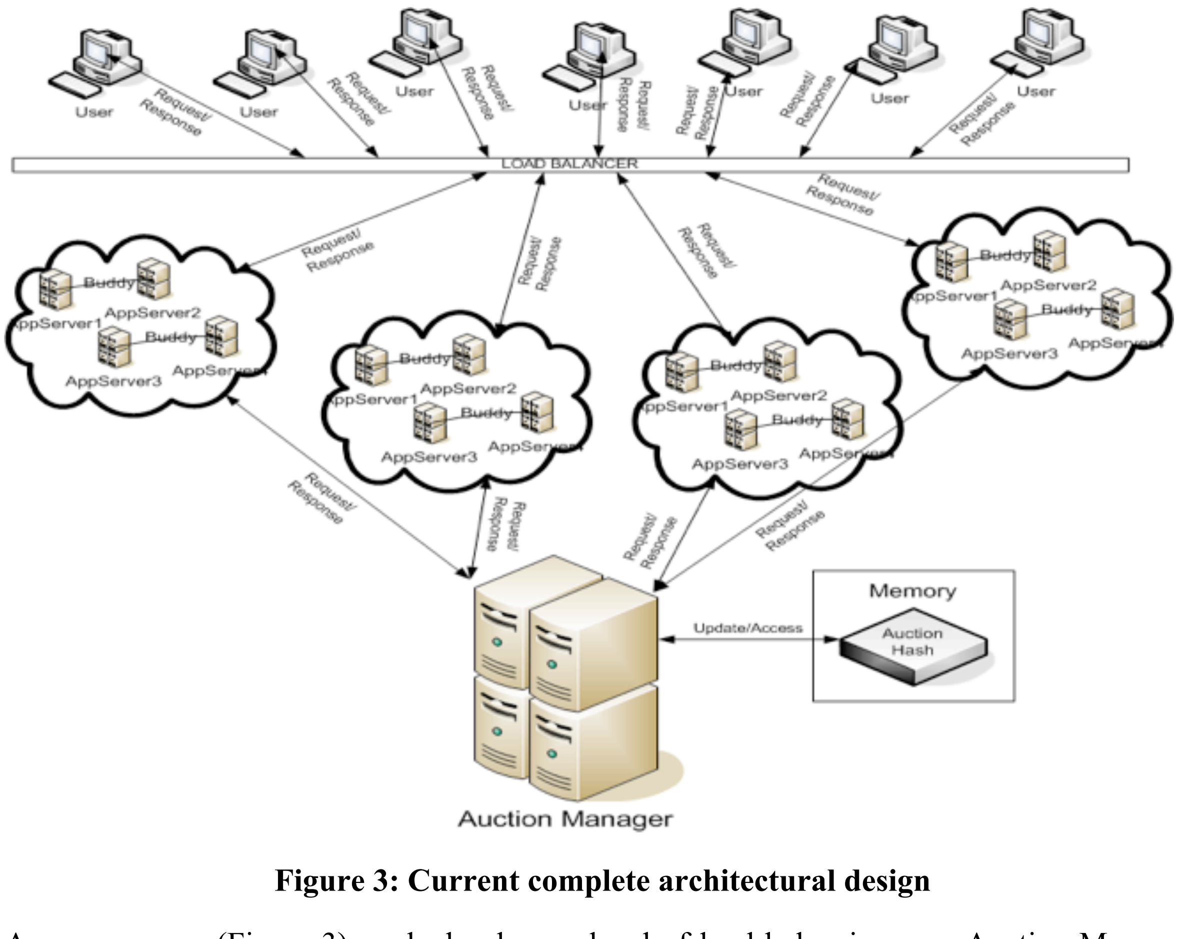


Figure 3: Current complete architectural design

As you can see (Figure 3) we had only one level of load balancing, one Auction Manager, and an Auction Hash, which contains all the data.

We wanted to create a system with single authority not implemented in a database. We needed multiple levels of filters to serve bid request and auction updates. Essentially, we were looking for a distributed caching mechanism like memory-based key/value stores. Thus, vertical scale can be achieved by provisioning more RAM to the machine. A more sustainable scale is possible by cloning and replicating nodes and partitioning the key space. We started our research on existing open source severs which can solve the purpose at a low cost or no cost to the organization. Our research was focused on re-engineering the existing system to address the missing non-functional requirements.

5.2 INCORPORATING REDIS TO THE SYSTEM

The online auction module has been developed for almost a year now and uses only one auction manager to serve all bids and updates. We were using the static auction hash to store all the information; data in auction hash should be same across all requests, so queued up requests compete for one synchronized Auction Hash object. There is no failover mechanism. The system is dependent only on a single auction manager, which performs everything. It is a single point of failure. We incorporated Redis to this online auction system to address the failover, to make the system more scalable and available to thousands of users.

Redis is an open source, advanced persistent key-value store, also referred to as a data structure server since the keys can contain hashes, lists, sets and sorted sets. Redis uses hashing to optimize the memory and the key-value store designed using Redis is significantly more efficient.

5.2.1 REDIS REPLICATION

Redis replication is very simple to use and configure. It offers master-slave replication that allows slave Redis servers to be exact copies of the master servers.

PROPERTIES OF REDIS REPLICATION:

1. A master can have multiple slaves.
2. Aside from connecting number of slaves to masters, slaves can also be connected to other slaves in a graph like structure to have more peer-to-peer like structure to

support failover.

3. Redis replication is non-blocking on master side, which means that master will continue to serve queries when one or more slaves performs the very first synchronization. It is also non-blocking on the slave side, which means that when a slave is performing the synchronization it can also reply to queries using old version of data set. These configurations can be done in redis.conf file or can be configured to send an error message to client when the master is down.
4. Replication can be used for scalability in order to have multiple slaves for read only queries.
5. It is possible to avoid saving on master server, just a small configuration is needed.

REPLICATION

The slave upon connection sends the SYNC command. As soon as master receive the SYNC command it starts background saving, and collects all new commands received that will modify the existing dataset. When the background saving process is complete the master send the dataset to slaves, which saves dataset on disk, and then loads the dataset into the memory. The master then sends to the slaves all accumulated commands, and all new commands received from users that will modify the dataset. This is performed as a stream of commands.

Whenever the master-slave link fails, slaves try to reconnect to the master and if the master receives multiple connection requests from slaves it performs a single background save in order to serve all of them. As soon as the connection is established between the master and the slaves a full resynchronization has been performed.

5.2.2 REDIS PERSISTENCE

Redis provides different types of persistence:

1. RDB: It performs point-in-time snapshots of the dataset at specified intervals.
2. AOF: It logs every write operation received by the server.

Persistence can be disabled in case the data needs to be available only as long as server is running. It is also possible to combine RDB and AOF in the same instance. But in that case, when the redis server restarts the AOF file will be used to reconstruct the original dataset since it is guaranteed to be the most complete.

Table 1: Advantages and Disadvantages of RDB Persistence

S.No.	Advantages	Disadvantages
1.	RDB is a very compact single-point-in-time representation of your Redis data.	RDB is not good if you need to minimize the chance of data loss in case Redis stops working.
2.	It is very good for disaster recovery.	RDB needs to fork often in order to persist on disk. using a child process.
3.	It allows faster restarts with big datasets compared with AOF.	fork can be time consuming depending on the size of the dataset.

Table 2: Advantages and Disadvantages of AOF Persistence

S.No.	Advantages	Disadvantages
1.	Using AOF Redis is much more durable: different synchronization policies could be achieved.	AOF file is usually bigger than corresponding RDB file for the same dataset.
2.	The AOF log is an append only log, no seeks, no corruption problems if there is a power outage.	It is slower than RDB.
3.	Redis is able to automatically rewrite the AOF in background when it gets too big.	Bugs are possible in AOF but are almost impossible in RDF format.
4.	AOF contains a log of all the operations one after the other in an	Redis AOF works incrementally updating an existing state.

	easy to understand and parse format.	

It is advisable to use both persistence methods to provide a high degree of data safety.

5.2.3 BACKING UP REDIS DATA

Redis is backup friendly. It is possible to copy the files while database is running. The following steps can be taken to backup the files:

- Regular snapshots on an hourly or daily basis in different directories and server.
- Old snapshots or files should be deleted. Versioning of the files should be maintained.
- Data could be transferred to a safe location other than the regular data centre periodically.

5.2.4 DISASTER RECOVERY

Disaster recovery is the ability to backup or transfer data to several places. This way data is safe and secured in case something unexpected happens to the Redis server. Common disaster recovery techniques are transferring data to Amazon S3 in an encrypted form or doing SCP to the servers located far away at various different locations.

5.2.5 HIGH AVAILABILITY

Redis sentinel is a system designed to monitor the Redis resources. Its major task is to monitor whether master and all slaves functioning correctly, if not than it notifies the administrator that something is wrong with one of the nodes or instances. If the master does not function well than it promotes one of the slaves to behave as the master and remaining nodes in the cluster are reconfigured automatically to use the new master and the application using the redis server is informed to connect to the new address of new master.

5.2.6 REDIS CLUSTER

The redis cluster is a pragmatic approach to distribution. All nodes are directly connected to each other. They do not proxy queries. Clients talk to nodes. Nodes communicate to each other by PING/PONG communication protocol. PING/PONG packets contain enough information for the cluster to restart after it stopped. Each node has a unique ID (it lives forever and never changes for any given node ever) and config file.

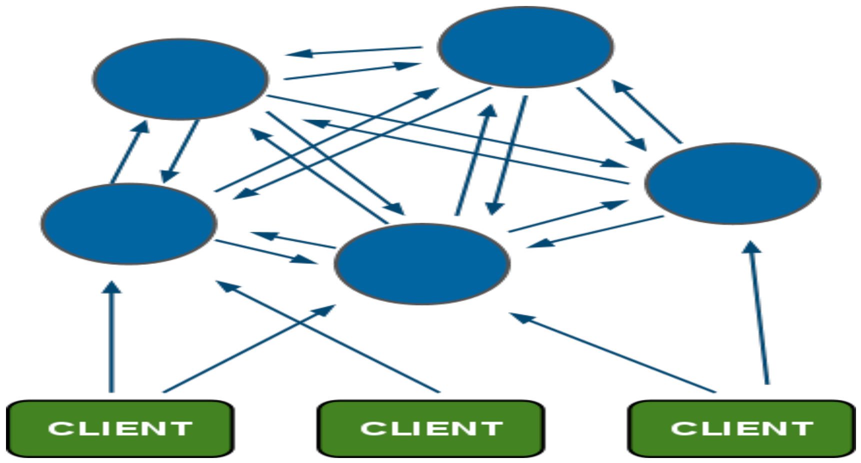


Figure 4: Redis node cluster

Hash slots: key space is divided into N hash slots. 10 hash slots in the below example are divided among nodes.

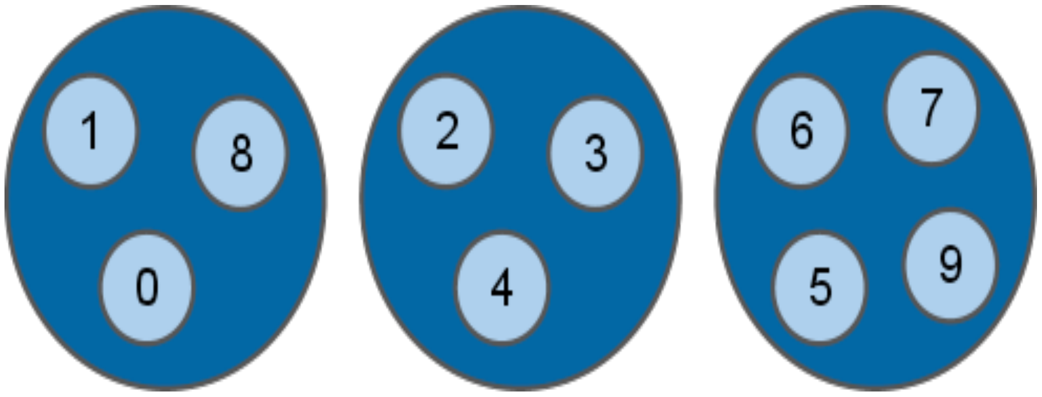


Figure 5: Redis node cluster with 10 hash slots

There are of two types of nodes: master and slave nodes. Their functionality is essentially same.

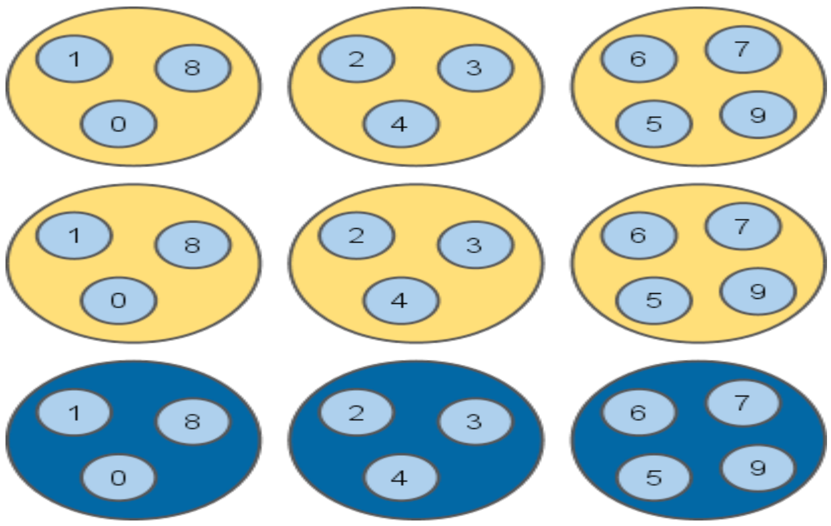


Figure 6: Redis master-slave node cluster

Above diagrams explains that every key exists only in one node and in its N replicas, which never receive, any sort of write operations. Replicas or slaves use redis replication to remain in sync. The best way is to allocate master and slave to the different physical server to avoid loss of data completely in case one physical server fails.

The cluster will continue working as long as there is at least one node, which is up and running. For example, in diagram below two nodes are down.

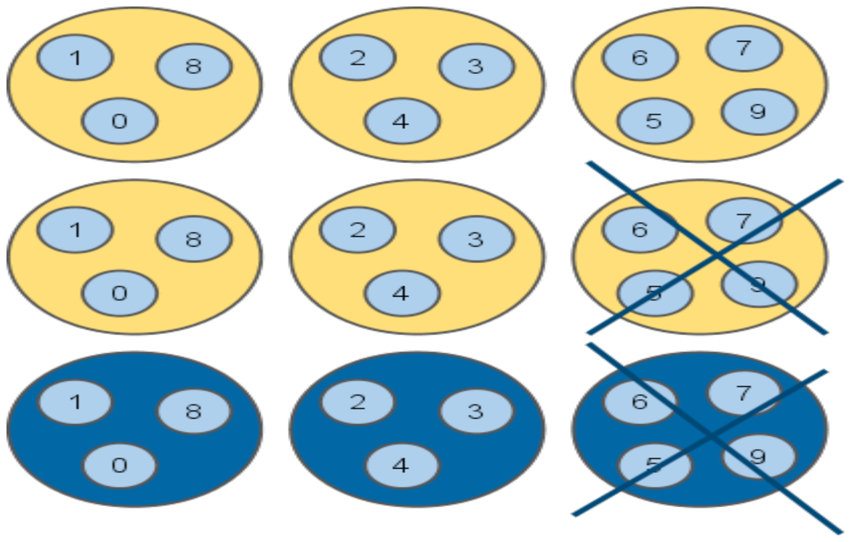


Figure 7: Redis master-slave node cluster with 2 slaves down

Re-sharding is a process of moving the hash slots from one node to a totally different or new node (master or slave) if the former experiences a heavy load. Below is the example of moving hash slot 7 from node C to node D.

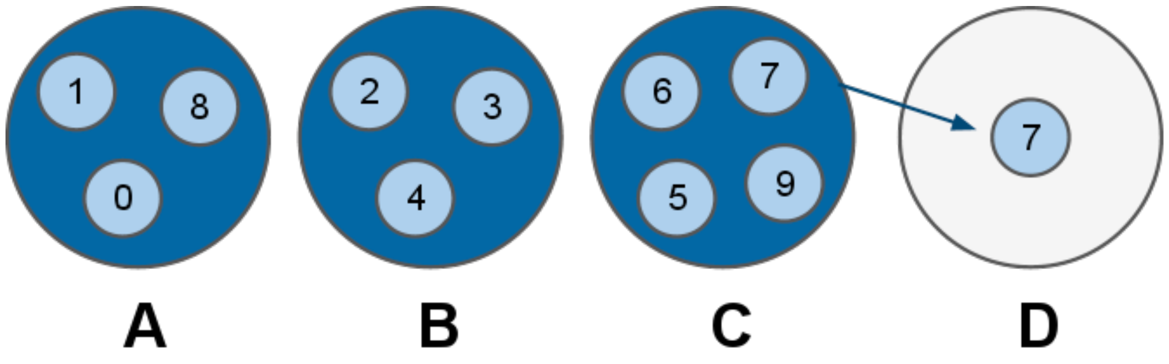


Figure 8: Redis node cluster showing resharding

Here, if node C receives any request related to slot 7 it will pass on to the node D. All the new keys for slot 7 will be created and updated in node D and will be moved or deleted from node C.

Re-sharding can fail if a slave node or master who it is being transferred to is down. If it happens, the hash slot goes to the slave of the failing node, as shown in the Figure.

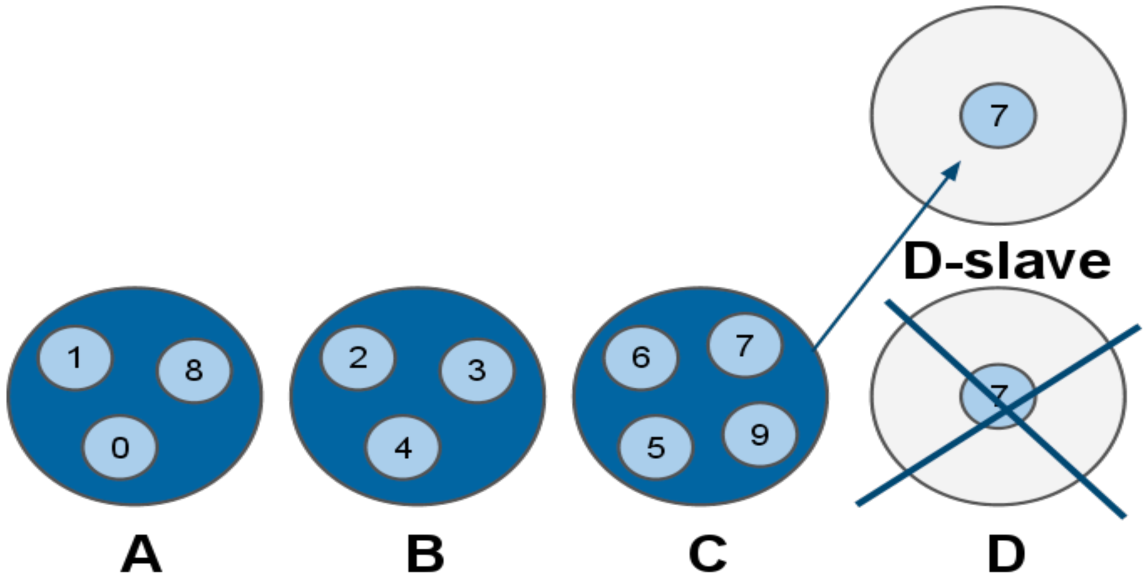


Figure 9: Redis node cluster showing resharding with master down

Fault-Tolerance: All nodes continuously ping other nodes and at any given time any particular node is marked as failed in the cluster if its timeout is longer than N seconds. In the redis cluster, every ping-pong contains gossip information. For example, if node A in a cluster thinks that node B is failing because PING request timed out, A cannot declare node B as failed by itself, with the use of the gossip action A will communicates with node C and if node C also thinks that node B is failing because of the time out then node A can mark node B as failed and convey this information to the whole cluster.

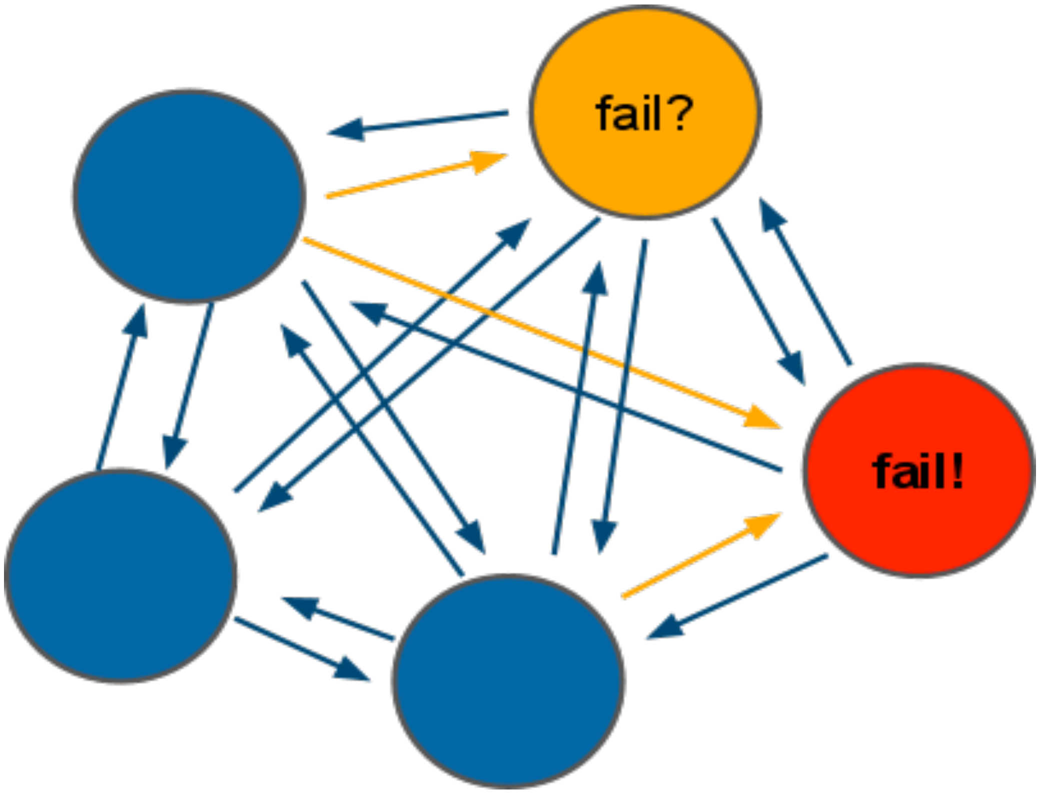


Figure 10: Redis fault-tolerance node cluster

Redis-trib: Its role is to setup a new cluster environment and checks regularly if the cluster is consistent or not. Its job is also to add new nodes to the cluster, either as master or as a slave to an existing master node or as blank nodes for resharding purpose to reduce the heavy load.

5.3 SYSTEM MODIFICATIONS

After researching key-value data stores we narrowed our choices down to **Terracotta** and **Redis**, Terracotta is more than what we needed and requires more time for implementation within the existing system and high cost is involved for its implementation. We decided to move forward with Redis, which is simple, provides persistence, free, open source and offered what was needed. We re-architected the code to work with redis functionalities to address the missing non-functional requirements we had in the existing system.

5.3.1 Hash Maps

Static Hash Map: Initially the static hash maps with the key was AuctionID and value was a complete Auction Summary Object (It has everything in it name of the Auction, Redemption type, starting big, client ID, auction status, start date, end date increment amount, display start date and end date, etc.) was used. The problem with the static hash map was that when it gets accessed the complete hash map acquired lock and prevented further processing until the lock was released (see Table 3). This resulted in decreased performance of the system.

Table 3: Static Hash Map (key-value)

Acquire Lock on entire hash map

Key	Value
100	Object (Auction Summary Object)
123	Object (Auction Summary Object)
456	Object (Auction Summary Object)
623	Object (Auction Summary Object)

Serialized Auction Hash Map: It is the same as the static hash map, which means AuctionID is the key and Auction Summary Object is a value. The only difference is that it is serialized and it acquires a lock to a particular entry and the remaining entire hash map is available to read and write. It has a locking mechanism and the lock is released once the process is complete.

Table 4: Serialized Auction Hash

Acquire Lock only on a particular entry and the remaining entire hash map is available to read and write

Key	Value
100	Object (Auction Summary Object)
123	Object (Auction Summary Object)
456	Object (Auction Summary Object)
623	Object (Auction Summary Object)

5.3.2 Code snippets (Jedis)

We used Jedis, since it is the most widely used API for Redis and it has a better support as compared to others (Jredis, Gedis)

- Jedis jedis = new Jedis (“localhost”); can also be connected to remote server with given port and timeout
- Jedis.connect(); //connect to Redis
- Jedis.set(“KEY”, “Value”); //set the value for the key
- String value = jedis.get(“KEY”); //get the value for the key which is case sensitive
- Jedis.disconnect(); //disconnect from Redis- an important step as performance deteriorates when multiple clients are connected even when they are not using the connection in any way.

Jedis is Java API for locking Redis objects and it uses SETNX to lock the object or a particular entry. Pseudo code:

```
While (AcquireTimeOut > 0) {
        If (setNX <lock key>, <current time + lock timeout +1> =1)
                Locked
        Else if (GET <lock key> less than Current Time)
                SET <current time + lock timeout +1>
                Locked
        Else
            Sleep
            Decrement AcquireTimeOut by total time taken
    for all the above steps
    }
```

After incorporating Redis within the Auction module, we observed significant change in response time and load balancing. During testing master was set to fail on purpose to observe and test the behavior of slave. This master failure posed no problems to end-users experience at all.

5.4 THE NEW SYSTEM

In this section, we present the design of the newly re-engineered system, which was the goal of this thesis. The new design shown below (Figure 12) explains the visual representation of the re-engineered online auction system with multiple web servers, application servers, and auction managers.

Below, we show the newly re-engineered system, which addresses all the missing non-functional requirements in addition it provides the bonus to have failover, which was the must requirement. This work can be used as a case study to address the set of missing non-functional requirements via system re-engineering.

New Auction Architecture:

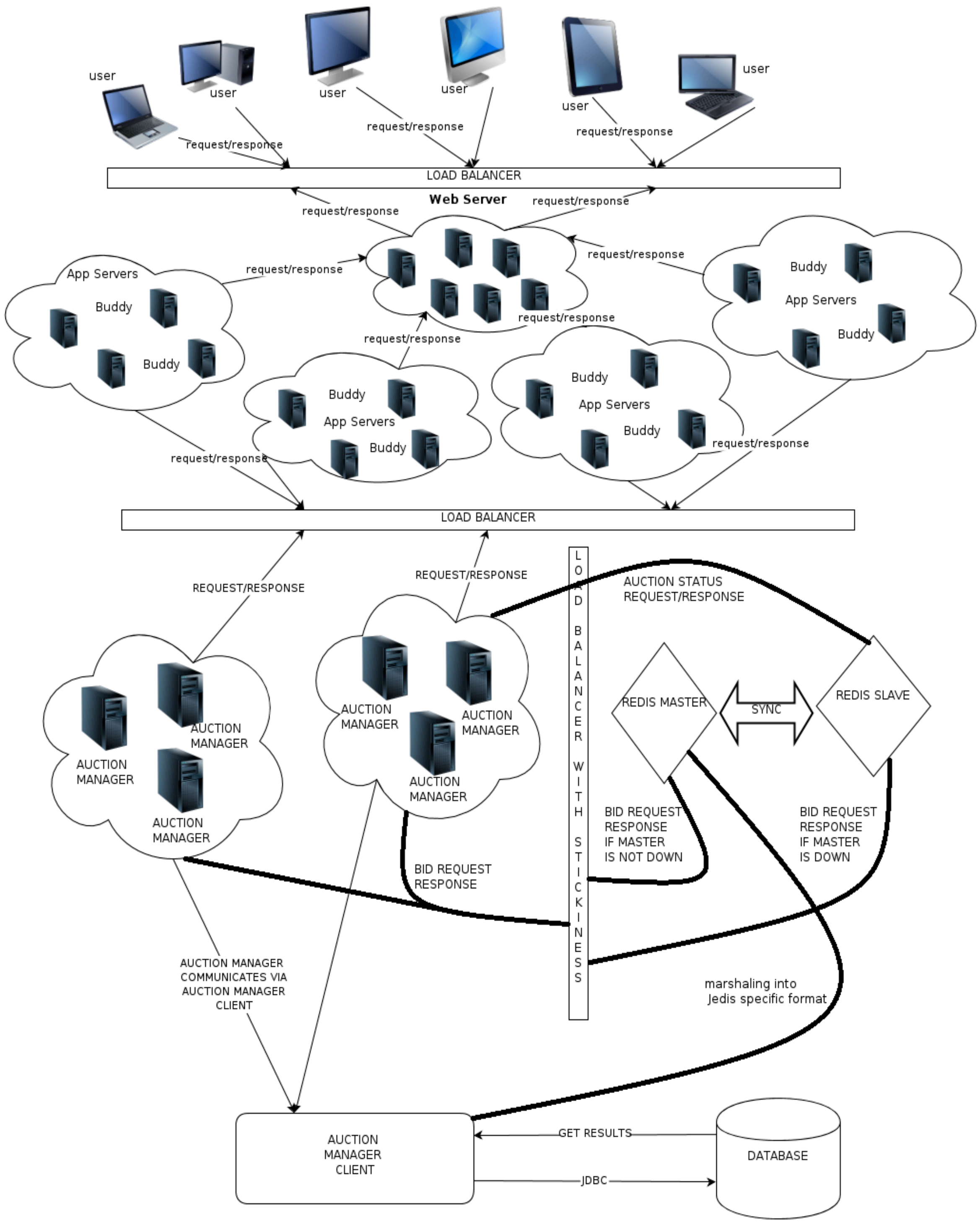
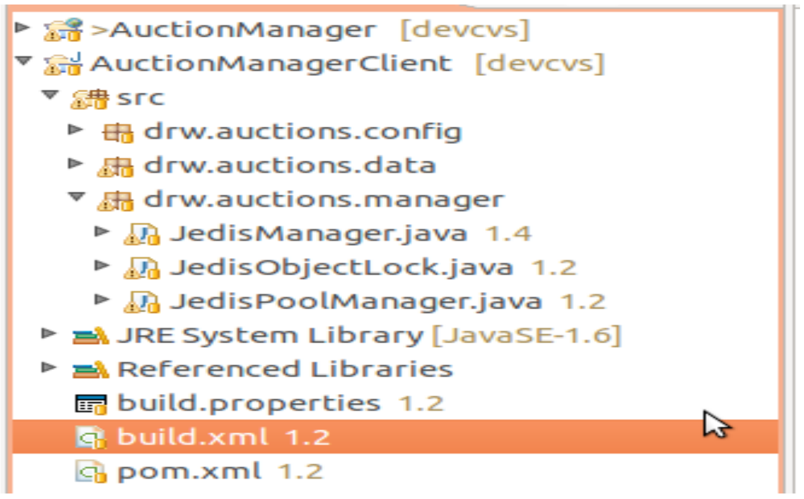


Figure 11: Newly architected design of the system

In the Figure 11, the auction manager communicates with Redis via a separate module called Auction Manager Client (shown in the Figure), The Auction Manager has a dependency of a project called Auction Manager Client which includes all utility methods written and all Redis (Jedis) related methods such as configuration of Jedis, obtaining an instance of Jedis and its slaves and the locking mechanism. The role of Auction Manager Client is to communicate with the database to get the desired results and to store those results into the Jedis serialized Auction Hash (in Jedis specific format). The Auction Manager Client also has a utility method, which marshals the database results into Jedis Specific format.

It is obvious that the newly designed system have a load balancer at 3 levels, one is at the Web server (web servers) level and application servers are buddy replicated to support session failover. The second level is at the Auction Manager level (multiple cluster with multiple nodes of the auction manager). The third level is at Redis server level with stickiness (Load balancer sticks to a server below it until it fails). The new system has failover mechanism, multiple application servers running together with synchronized data. Auction application is smart enough to balance the load by itself. As a result, we have achieved all stated goals without having to purchase any commercially developer software and at a relatively small development effort.

Package Structure: We created a new package, which includes classes related to Jedis implementation (see Figure 12)



```
▶ >AuctionManager [devcvs]
▼ AuctionManagerClient [devcvs]
  ▼ src
    ▶ drw.auctions.config
    ▶ drw.auctions.data
    ▼ drw.auctions.manager
      ▶ JedisManager.java 1.4
      ▶ JedisObjectLock.java 1.2
      ▶ JedisPoolManager.java 1.2
  ▶ JRE System Library [JavaSE-1.6]
  ▶ Referenced Libraries
    build.properties 1.2
    build.xml 1.2
    pom.xml 1.2
```

Figure 12: Project and package representation

CHAPTER 6

EVALUATION

6.1 GOALS ACHIEVED

We have achieved significant performance improvements after the Redis was incorporated to the system. Now, the system has 4 clusters of buddy-replicated applications to support session failover. The system also possesses load balanced gateway access, 6 web servers (Load balanced). It has a cluster of multiple auction managers running on each node. When count of auction managers was increased from 1 to 4 linear curve of performance improvement was noticed. During load testing we observed that the Redis test server has a capacity for 100,000 SET/GET request for a 3KB data set are:

1. *48852 SET request/sec (0.0205 millisecond/request)*
2. *51894 GET request/sec (0.0193 millisecond/request)*

After that, increasing the number of auction managers did not improve the performance significantly. However, system supports for session failover, improved performance and efficiency. The system is more scalable and available.

6.2 RESULTS

Database transactions are reduced to almost 99%. Initially, everything was done via database transactions, which included all bid requests, rewards currency deductions, auction updates, etc. This resulted in millions of hits on the database. Now, with the redis-based implementation the database traffic has been reduced almost to zero.

Below, Figure is the chart for the actual bid page of the re-engineered auction system. Chart shows the response time in milliseconds on the Y-axis and number of users in thousands on the X-axis. Close examination of chart states that as the number of users increases from 50k to 100K, the response time increases and after 175K it decreases which means the system is more responsive for a large number of users. Scalability has been addressed in this new re-engineered system.

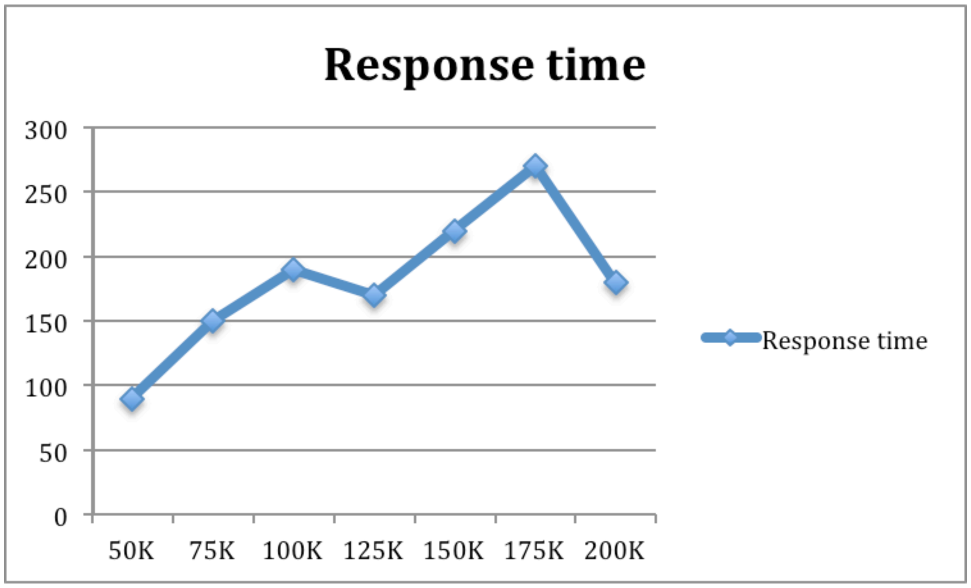


Figure 13: Chart Maximum users Vs. Response time for Auction Bid Page

Next, Figure is the chart for Latency versus hits/second for a auction bid gateway. As hits/second increased the latency increased and then it dropped suddenly. Latency limits the maximum rate that information can be transmitted, as there is often a limit on the amount of information that is "in-flight" at any one moment.

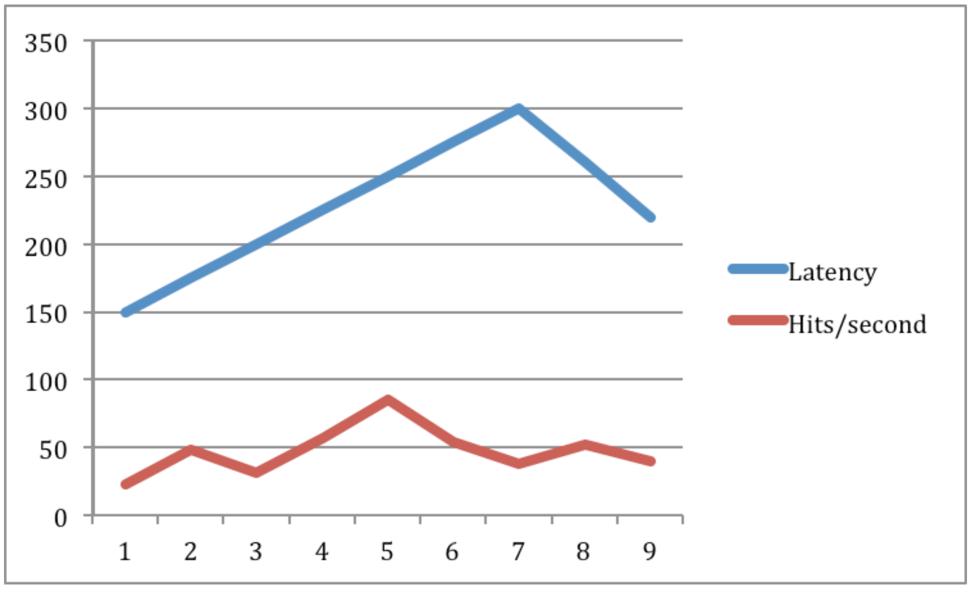


Figure 14: Chart Latency Vs. Hits/Second

Below,Figure is the chart for a response time versus users for a Auction System login page for users (in thousands) on the X-axis and response time (in milli-seconds) on the Y-axis. The system is scalable for login page access as well.

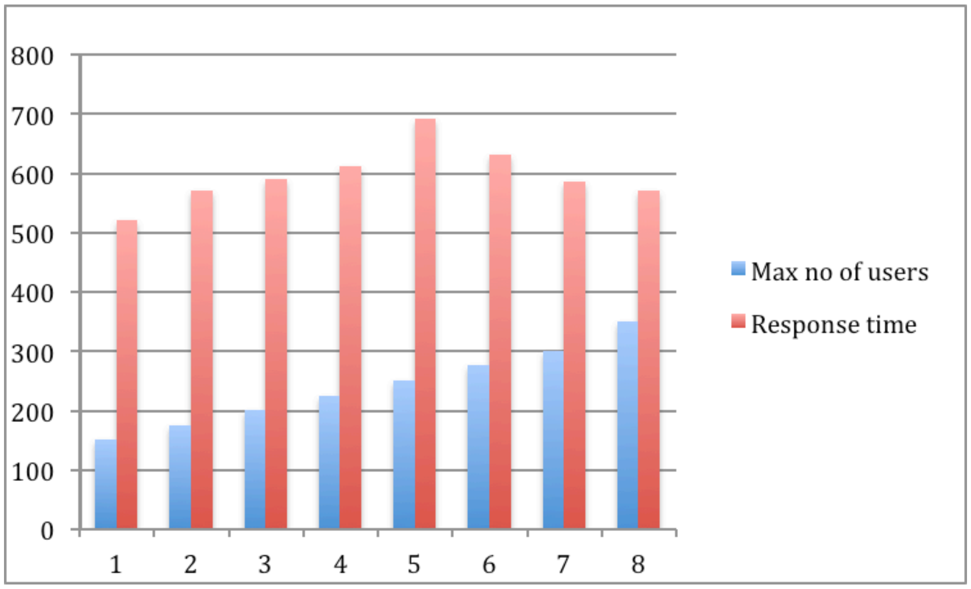


Figure 15: Chart Maximum users Vs. Response time for Auction Login Page

Below we presents screen-shots obtained during load testing on the auction application.

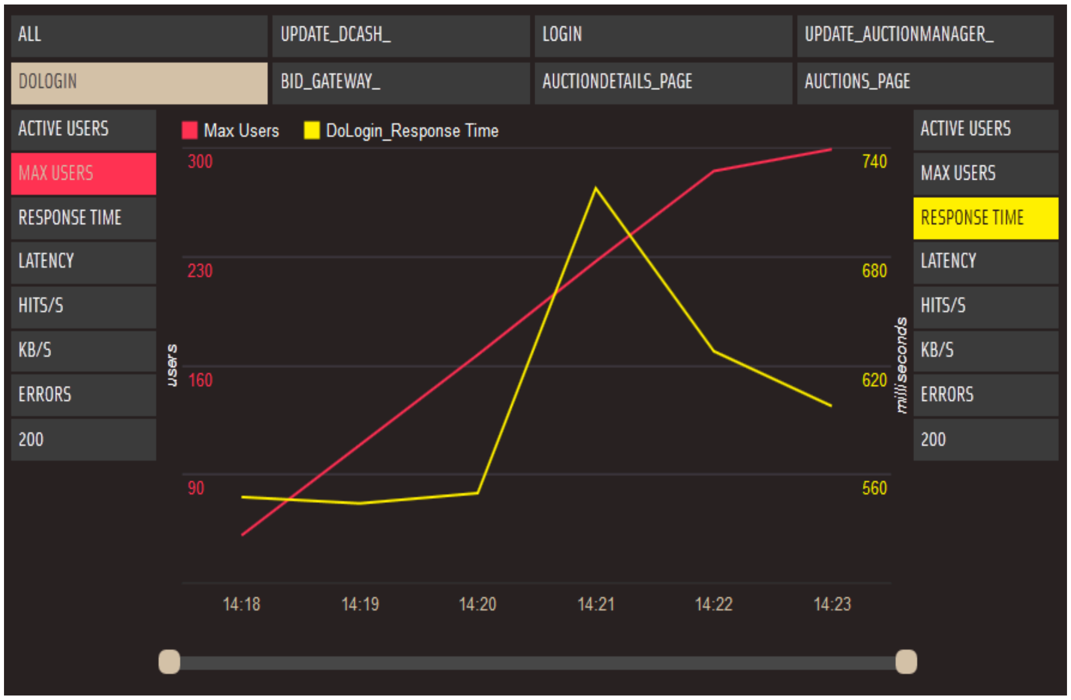


Figure 16: Graph for Users Vs. Response time for Login Page

The plot in Figure 16 shows the response time for the Login Page. This measurement was taken while load testing was being conducted and we noticed a significant improvement in the response time as compared to the old system. The approach used in this paper gives the noticeable amount of performance improvement on the server response time.

The graph in Figure 17 shows the response time of the main bidding page (the page where user can actually bid with the use of the “Place Bid” button). It shows the response time as a function of users vs. time (milliseconds).

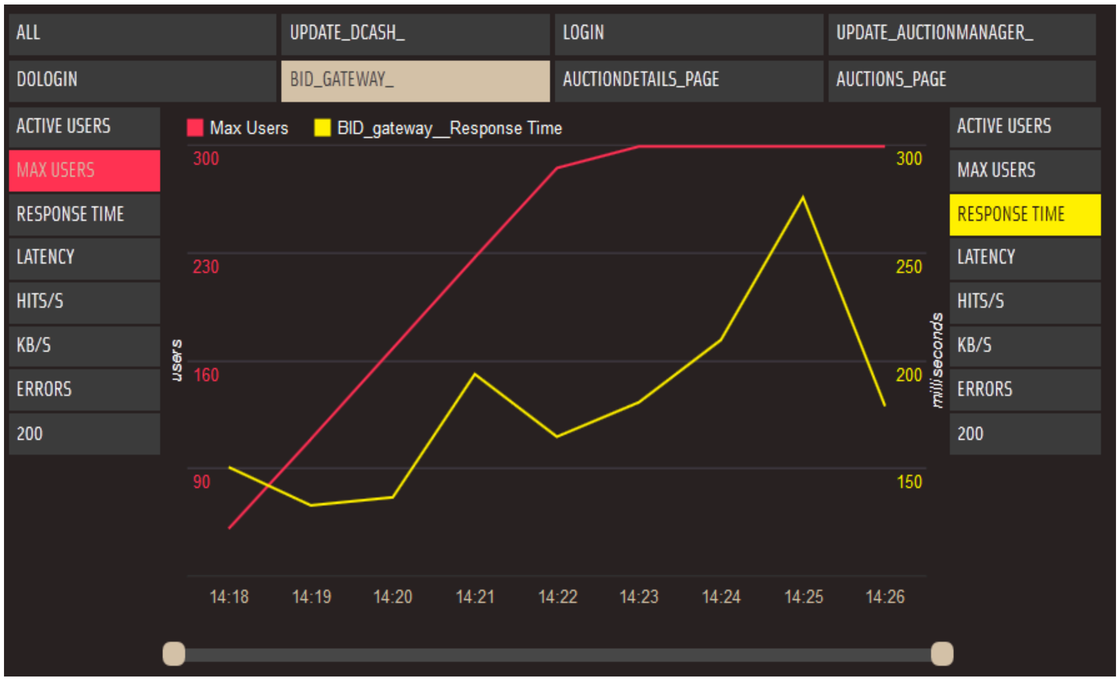


Figure 17: Graph for Users Vs. Response time for Bid Page

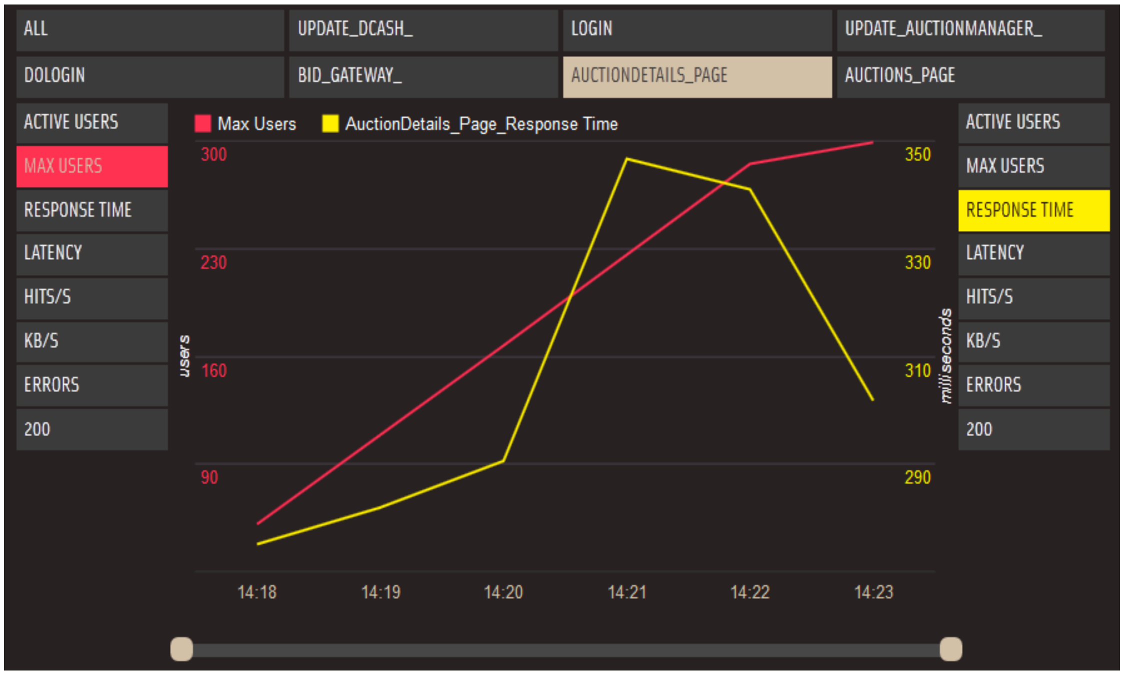


Figure 18: Graph for Users Vs. Response time for Auction Details Page

The graph in Figure 18 shows the Response time noticed on the server when user hits the Auction Details Page. This is the page where users can see minutes of auction details such as amount he has available to place bid, current highest bid, number of online bidders and the bid-history. This was a heavy load functionality, which usually takes longer time in the previous system. After the re-engineering it is reduced to a significant number.

The graph in Figure 19 shows the response time noticed on the server when user hits the Auction Page. This is the page where user can view all available auctions, upcoming auctions, dual currency auctions and carousel to the most fascinating and lucrative auctions and the auctions he participated in.

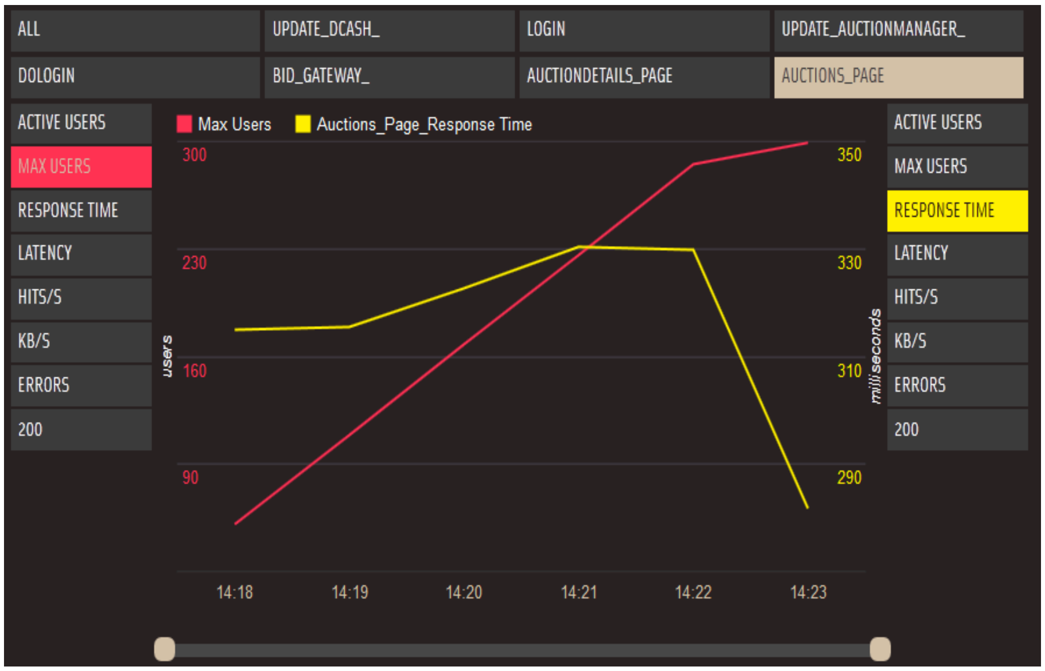


Figure 19: Graph for Users Vs. Response time for Auction Page

The graph in Figure 20 shows a variation between latency for the Auction Bid page versus hits-per-second.

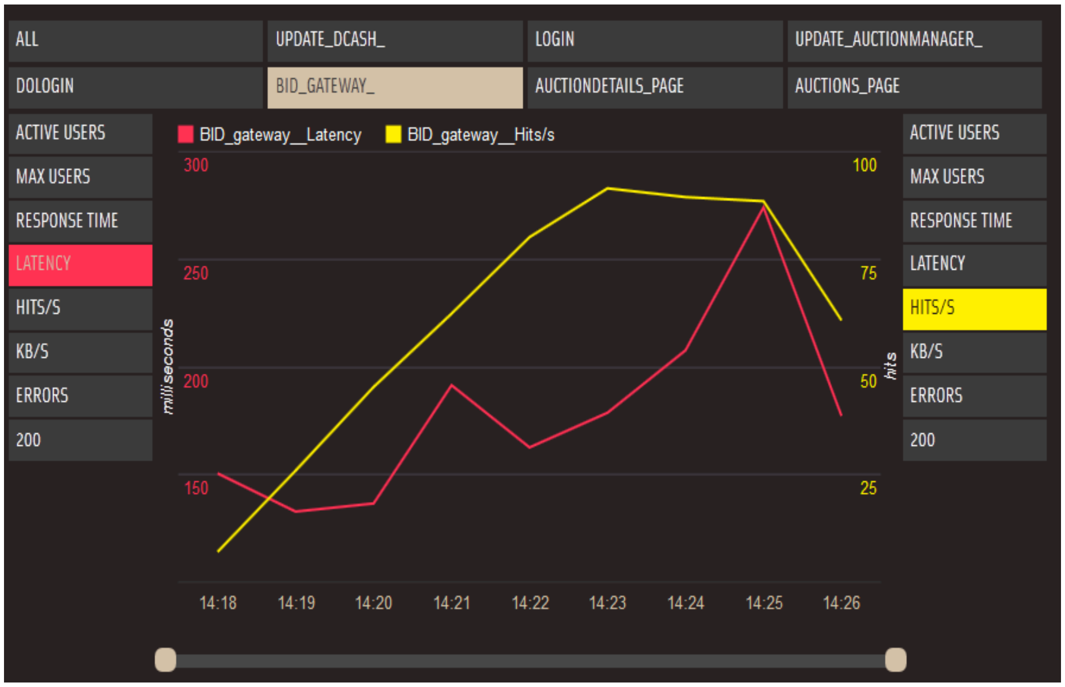


Figure 20: Graph for hits vs. latency for Auction Bid Page

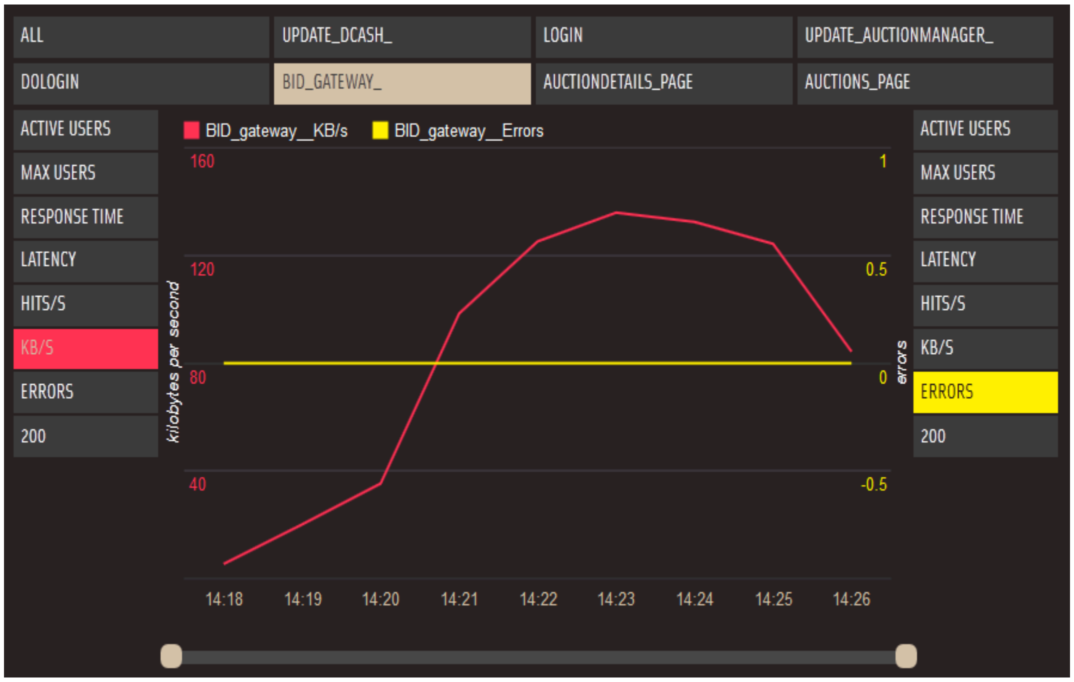
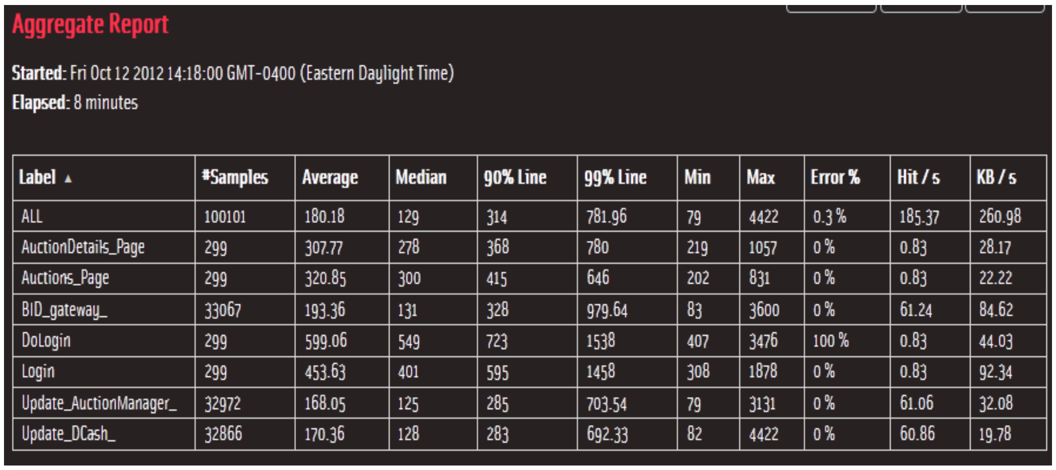


Figure 21: Graph for kilobytes/s vs. errors for Auction Bid Page

The graph in Figure 21 shows a kilobytes-per-second versus errors. And a close examination clearly states that zero error responses from the server were achieved. Previously, there were

130 to 140 errors due to a heavy load. By re-engineering the system it is reduced to zero.

The table in Figure 22 is the aggregate report for the various pages of the Auction system. It describes the overall performance and efficiency of the system. It includes many indications, including error percentage, hits/second, KB/second, Median, Average and Sample we took.



Aggregate Report

Started: Fri Oct 12 2012 14:18:00 GMT-0400 (Eastern Daylight Time)
Elapsed: 8 minutes

Label ▲	#Samples	Average	Median	90% Line	99% Line	Min	Max	Error %	Hit / s	KB / s
ALL	100101	180.18	129	314	781.96	79	4422	0.3 %	185.37	260.98
AuctionDetails_Page	299	307.77	278	368	780	219	1057	0 %	0.83	28.17
Auctions_Page	299	320.85	300	415	646	202	831	0 %	0.83	22.22
BID_gateway_	33067	193.36	131	328	979.64	83	3600	0 %	61.24	84.62
DoLogin	299	599.06	549	723	1538	407	3476	100 %	0.83	44.03
Login	299	453.63	401	595	1458	308	1878	0 %	0.83	92.34
Update_AuctionManager_	32972	168.05	125	285	703.54	79	3131	0 %	61.06	32.08
Update_DCash_	32866	170.36	128	283	692.33	82	4422	0 %	60.86	19.78

Figure 22: Aggregate Report

CHAPTER 7

CONCLUSION AND FUTURE WORK

Despite the fact that non-functional requirements are very difficult and expensive to deal with, the increasing software complexity and competition in the software industry has highlighted the need to consider NFRs as an integral part of software development process. However, they are still disregarded and many shortcomings of the developed system remain.

In this paper, we have presented an NFR driven approach for software re-engineering at the architectural level. The approach uses desirable qualities for the re-engineered code and design to define and guide the re-engineering process. The work offers a workbench where re-engineering activities do not occur in a vacuum, but could be evaluated and fine-tuned in order to address specific quality requirements for the target system. Specifically, the proposed re-engineered system would address the issues related to: scalability, availability, failover and distributed load balancing.

In this work, we have shown that how redis was incorporated and configured. Redis an open source system to improve performance, to gain scalability, availability and failover. We evaluated the performance of the newly architected system and noticed a vast improvement in terms of the server response time. This work describes in detail how Redis (which is persistence in-memory key-value storage) could be used for an existing system to achieve failover, to improve overall performance and how database traffic could be reduced to such a great extent by implementing single point of authority. Results indicates that the system with Redis implementation works even there is failure for master server and even if cluster or its individual node fails. This work proves that performance and quality can be achieved by using Redis in short span of time without implementing backing mechanism for a live running module for clients with large number of users. Implementing separate mechanism can be good but this introduction to redis to a live running module will beat that implementation on all grounds. We did thorough testing of the new system and it outperforms the previous system totally.

In this work Redis (Jedis API) mostly uses a single threaded design. This means that single process serves the entire users request, using a technique called multiplexing. Speed is achieved by writes not blocking the entire system (Hash in our case) calls like reading data to and from socket. It is sequential and serves single request at a time. Some threads are also used for I/O operations in the backend (mostly single threaded).

As a research continues on re-engineering or quality driven refactoring, it is necessary to formalize the NFRs to get a general knowledge base. Another problem is that quality is still neglected and has to come to developer's pocket to take NFRs as serious as functional requirements.

In the future we want to enhance the system capability more, currently when master fails over once to slave, we do not have the capability of automatically making the old master the slave of the new master. There is some manual intervention required to make sure the next switch happens smoothly. Moreover, we now create individual Redis instances as needed and disconnect after the usage. So in near future we are looking into a spring based Jedis implementation, which will help in pooling the Redis instances.

REFERENCES

1. Matti Paksula, Helsinki Finland-Persisting Objects in Redis Key-Value Database. Unpublished white paper.
2. Morris, R. 1968. Scatter storage techniques. Communication of the ACM 11, 1 (Jan. 1968), 38-44. DOI=http://doi.acm.org/10.1145/362851.362882.
3. Karl Seguin-The Little Redis Book. http://openmymind.net/2012/1/23/The-Little-Redis-Book.
4. Josiah L. Carison-Redis in Action. http://www.manning.com/carlson.
5. Tiago Macedo, Fred Oliveira-2011 Redis Cookbook, Practical Techniques for fast data manipulations. http://shop.oreilly.com/product/0636920020127.
6. M. Fayad and D. Schmidt, "Object-Oriented Application Frameworks," Communications of the ACM, vol. 40, October 1997.
7. Mateusz Berezecki, Eitan Frachtenberg, Mike Paleczny, and Kenneth Steele. Many-core key-value stores. In Proceedings of the Second International Green Computing Conference, Orlando, FL, August 2011.
8. N. Chohan, C. Bunch, S. Pang, C. Krintz, N. Mostafa, S. Soman, and R. Wolski. AppScale: Scalable and Open AppEngine Application Development and Deployment. In *Cloud Comp*, 2009.
9. L. Tahvildari, K. Kontogiannis, and J. Mylopoulos, "Quality-Driven Software Re-Engineering," Journal of Systems and Software, vol. 66, pp. 225-239, June, 2003.
10. R. Tiarks. Quality-driven refactoring. Technical report, University of Bremen, 2005.
11. International organizations for standardizations (ISO), 1996. Information Technology, software product evaluation, quality characteristics and guidelines for their use, ISO/IEC 9126, 1996.
12. Redis: http://redis.io
13. Memcached: http://memcached.org/

14. VoltDB: http://voltdb.com/

15. TerraCotta: http://terracotta.org/

16. Jboss Cache: http://www.jboss.org/jbosscache

17. Hazelcast: http://www.hazelcast.com/

18. I. Ivkovic and K. Kontogiannis, "A Framework for Software Architecture Refactoring using Model Transformations and Semantic Annotations," presented at the The Conference on Software Maintenance and Reengineering (CSMR'06), Bari, Italy, 2006.

19. R.J. Figueiredo et al., “A case for grid computing on virtual machines,” *23rd International Conference on Distributed Computing Systems*, 2003, pp. 550-559.

20. S. Shrivastava and V. Shrivastava, "Impact of Metrics Based Refactoring on the Software Quality: a Case Study," presented at the The 2008 IEEE Region 10 Conference (TENCON'08), Hyderabad, India, 2008.

21. L. Tahvildari, "Quality-Driven Object-Oriented Re-Engineering Framework," presented at the The 20th IEEE International Conference on Software Maintenance (ICSM'04), Chicago, IL, USA, 2004.

22. Y. Yu and J. Mylopoulos. Software refactoring guided by multiple soft-goals. The *First International Workshop on Refactoring: Achievements, Challenges, Effects (REFACE03)*, November 2003.

23. L. Grunske and R. Neumann, "Process Components for Quality Evaluation and Quality Improvement," presented at the The 2nd Workshop on Method Engineering for Object- Oriented and Component-Based Development (OOPSLA'04), The International Conference on Object Oriented Programming, Systems, Languages and Applications, 2004.

24. Bengtsson, P. and Bosch, J., Scenario-based Software Architecture Reengineering, *Proceedings of the 5th International Conference on Software Reuse* (ICSR5), IEEE, pp. 308-317, 2-5 June, 1998.

25. Matoussi, A., and Laleau, R.: ‘*A Survey of Non-Functional Requirements in Software Development Process*’, Technical report TR-LACL-2008-7, University of Paris 12, 2008.