

A LIGHTWEIGHT CACHING FRAMEWORK

Sameera Dinusha Nelson

148232M

Degree of Master of Science

Department of Computer Science and Engineering

University of Moratuwa

Sri Lanka

May 2018

A LIGHTWEIGHT CACHING FRAMEWORK

Sameera Dinusha Nelson

148232M

Dissertation submitted in partial fulfillment of the requirements for the degree Master
of Science in Computer Science and Engineering

Department of Computer Science and Engineering

University of Moratuwa

Sri Lanka

May 2018

DECLARATION

I declare that this is my own work and this MSc Research Project does not incorporate without acknowledgement any material previously submitted for a Degree or Diploma in any other University or institute of higher learning and to the best of my knowledge and belief it does not contain any material previously published or written by another person except where the acknowledgement is made in the text. Also, I hereby grant to University of Moratuwa the non-exclusive right to reproduce and distribute my thesis, in whole or in part in print, electronic or other medium. I retain the right to use this content in whole or part in future works.

.....

S.D. Nelson

.....

Date

I certify that the declaration above by the candidate is true to the best of my knowledge and that this project report is acceptable for evaluation for the CS6997 MSc Research Project.

.....

Dr. Indika Perera

.....

Date

ACKNOWLEDGEMENTS

I would like to convey my heartfelt gratitude to Dr. Indika Perera, my supervisor, for the excellent supervision, support and advice given throughout to make this research a success. I would also like to express my deep gratitude to Mr. Aruna Dissanayake, Head of Engineering, Persistent Software Lanka Pvt Ltd for helping to plant this project concept. Moreover, I would like to convey my sincere appreciation to the staff of the Computer Science and Engineering, University of Moratuwa for their generous support towards the completion of this task. I would also like to convey my appreciation to my family for their continuous support and motivation given to make this research project a success.

ABSTRACT

Node management techniques especially designed for the modern systems are becoming more and more advanced and complex with the research involvement from the industry and institutions today. Almost all type industries with the computing systems are trending towards the distributed environments now. In this context node management plays a very significant and important role which helps to fulfill the main aspect of the distributed-ness. Clustering is the process of managing, maintaining and binding different nodes together sharing common set of configurations to work together for achieving a common goal.

In this project, we design and implement an approach towards a lightweight caching framework for a distributed environment. Here, we evaluate several protocols and choose WebSockets protocol for implementing the node management of the cluster. In our approach, each node maintains node server and node client set in a multithreaded environment to cater the node management. We have detailed out the design and implementation aspects on chapter 4. Then we demonstrate the performance achievements of the caching framework by using it on an application, having small to medium scale cluster. Then through a simulated use-case of the caching cluster framework, the system stability is monitored while up scaling the system step by step. Finally, the above-mentioned use case is extended to a simulated experimental evaluation making a comparison with a reference framework. For that experiment, a reference framework which based on TCP protocol is implemented with the help of Netty framework. We have concluded by highlighting the significance of our caching cluster framework by the outstanding behaviors and features such as efficiency, lightweight-ness, and scalability, stability on scaling and very low system overhead.

TABLE OF CONTENTS

DECLARATION.....	iii
ACKNOWLEDGEMENTS.....	iv
ABSTRACT	v
CHAPTER 1	1
1 INTRODUCTION.....	1
1.1 Overview.....	2
1.2 Scope	3
1.3 Issues.....	3
1.4 Cluster Management for Caching.....	4
1.4.1 Consistency.....	4
1.4.2 Static Mode	5
1.4.3 Dynamic Mode.....	5
1.5 Configuration Management	5
1.6 Fault Tolerance	6
1.7 Distributed Caching	6
1.8 Research Problem.....	6
1.9 Motivation	7
1.10 Objectives	7
CHAPTER 2	8
2 LITERATURE REVIEW	8
2.1 Overview.....	9
2.2 History and Background.....	9
2.3 Related Work	9
2.4 Group Communication Protocols for Node Management.....	10

2.4.1	Multicasting Protocols.....	10
2.4.2	Gossip Protocols.....	12
2.5	Client- Server Protocols for Node Management.....	15
2.5.1	HTTP Protocol.....	15
2.5.2	WebSockets Protocol.....	16
2.6	Caching, Distribution and Replication	18
	CHAPTER 3	19
	3 METHODOLOGY	19
3.1	Overview.....	20
3.2	Cluster Membership.....	22
3.3	Caching.....	22
3.4	Cluster Setup Steps.....	23
	CHAPTER 4	25
	4 DESIGN AND IMPLEMENTATION	25
4.1	Tools Used for the implementation work	26
4.1.1	Netty Framework.....	26
4.1.2	WebSockets for Cluster Management	28
4.1.3	Framework Thread Model	30
4.1.4	Message Model.....	32
4.1.5	Security measurements in the Framework.....	35
4.2	Node Implementation	36
4.3	Server Implementation	37
4.4	Client Implementation	37
4.5	Node Registry	38
4.6	Managing Cluster Memberships.....	39
4.6.1	Phase 1: Cluster Setup	39

4.6.2	Phase 2: Cluster Sync up.....	41
4.6.3	Detecting Node Failures	42
4.6.4	Node Recovery	42
4.7	Caching	42
	CHAPTER 5	44
	5 EVALUATION AND RESULTS	44
5.1	Tests and Results: Using Cache Cluster Framework.....	45
5.1.1	Node Server Startup.....	45
5.1.2	Node Client Startup.....	46
5.1.3	Client Failure Sync up.....	46
5.1.4	Messaging	47
5.2	Assessment of the Advantages of Cache Framework.....	48
5.3	Feature Evaluation	49
5.3.1	Ease of usage	49
5.3.2	Lightweight-ness of the Framework	51
5.3.3	Scalability & Stability	54
5.3.4	Fault Tolerance.....	56
5.4	Performance Evaluation and Analysis	57
5.4.1	Application Use case.....	57
5.4.2	Experimental Evaluation	62
	CHAPTER 6	65
	6 CONCLUSION & FUTURE WORK.....	65
	REFERENCES	67

TABLE OF FIGURES

Figure 3.1: Purposed High-level component diagram	20
Figure 3.2: Responsibilities of the Components	21
Figure 3.4: Steps of adding a node to the cluster	23
Figure 4.1: Class Hierarchy Diagram for Socket Channels	28
Figure 4.2: NioServerSocketChannel usage in Server	29
Figure 4.3: NioSocketChannel usage in Client	29
Figure 4.4: Thread Model of the Cluster Framework	30
Figure 4.5: Handshaker usage through handlers	32
Figure 4.6: Client/ Server Handshake	33
Figure 4.7: Client Handshake Request [40]	33
Figure 4.8: Server Handshake Response [40]	34
Figure 4.9: Types of WebSocketFrame.....	34
Figure 4.10: lcf4j.properties Configuration File	36
Figure 4.11: Server Module Work Flow	37
Figure 4.12: Client Module Work Flow.....	37
Figure 4.13: Node Registry Implementation.....	38
Figure 4.14: Cluster Membership Management.....	39
Figure 4.15: Cluster Setup.....	40
Figure 4.17: Node Registry Update with Success Response	41
Figure 5.7: Thread Pool Configuration	50
Figure 5.8: Cluster Sync up Message.....	50
Figure 5.9: CPU Load on System Idling.....	51
Figure 5.10: CPU Load on Cluster Setup.....	52
Figure 5.11: CPU Load after Cluster Setup	52

Figure 5.12: Network Utilization on System Idling.....	53
Figure 5.13: Network Utilization on Cluster Setup	53
Figure 5.14: Network Utilization after Cluster Setup	54
Figure 5.15: Scaling the Cluster to 12 Nodes.....	55
Figure 5.16: Extending to 24 Nodes	55
Figure 5.17: Stable Cluster with 18 nodes	56
Figure 5.18: Stable Cluster after Node Failures.....	56
Figure 5.19: CPU Load on System Idling.....	58
Figure 5.20: Memory Usage on System Idling	58
Figure 5.21: Network Utilization on System Idling.....	59
Figure 5.22: CPU Load with Node Count.....	59
Figure 5.23: Memory Usage with Node Count	60
Figure 5.24: Network Utilization with Node Count.....	61
Figure 5.25: CPU Load, Web Sockets vs. TCP	62
Figure 5.26: Memory Usage, Web Sockets vs. TCP.....	63

LIST OF ABBREVIATIONS

Abbreviation	
P2P	Peer to Peer
NIO	Non-Blocking Input Output
SSL	Secure Socket Layer
HTTP	Hypertext Transfer Protocol
TLS	Transport Layer Security
TCP	Transmission Control Protocol
WS	WebSocket
WSS	WebSocket Secure

CHAPTER 1

1 INTRODUCTION

1.1 Overview

With the enhancement of the high-speed networks, commodity hardware usage and cheap storage, clustering in distributed computing is becoming a norm to guarantee more performance as well as availability of current distributed systems.

However, most of the frameworks currently available are mainly focused on one or few specific business requirements targeted for internal usage or towards enterprise business. It is becoming a trend in the software industry, that such products are released to the community allowing it to be adopted to general purpose usage as well with limited feature sets or partial capabilities.

On the other hand, such frameworks are not scaled well in all conditions, mostly making it hard to be used with small to medium clusters efficiently in lightweight mode. That to say, those frameworks often add a considerable amount of overhead and/ or complexity while fulfilling the clustering requirement even though the user is not expected or interested in such additional features.

Also, most of the distributed systems, especially for small to medium scale systems, node management has become a prominent issue, where the simplicity and the flexibility are always appreciated rather adding unexpected overheads or the complexities with the underlying clustering framework. That's to say, those frameworks are not implemented not having the node management in mind, but to achieve a different enterprise feature set.

To achieve the initial purpose, it is often required to find an accurate and adequate mechanism to handle node memberships of the cluster in a very efficient, lightweight manner as well as with a minimal communication overhead. Also, it is very important to find out the performance penalties which can be occurred on the underlying system by such mechanisms or frameworks.

Here, the caching framework with node management can be embedded as an underlying framework in to the system. Hence, it can be considered as an independent management layer or module of the system, which is completely decoupled from the business logic of the user application.

To cater the above requirement, the caching cluster framework should be implemented with a well-defined API, which can be integrated to the application seamlessly. Also having mostly required distributed feature such as configuration management will be adding additional benefits to the system without sacrificing the fundamental requirements, if they are made easily pluggable in to the system.

1.2 Scope

Defining an efficient, self-managing, fault tolerant, consistent and efficient cache clustering mechanism is still an open research topic, though both the enterprise software communities and the distributed systems communities have been utilizing clustering for a long time. Therefore, the focus of this project is to design and implement a very efficient, lightweight and fault tolerant cluster framework for distributed node management, which allows an application to seamlessly perform application specific operations with cluster awareness [1] after integration. Also, it is expected to provide a SCM [3] and distributed caching behavior, which is meant to be an added advantage of using this framework.

With the above features, the user application can benefit from simple-ness and lightweight-ness on the underlying topology, especially from the node management and related functions which allows to focus more on the business requirements intended for the application implementation.

1.3 Issues

One of the major concern in clustering is to maintain the node availability, cache & configuration synchronization among the nodes. This concern leads to a very common and arguable challenge, achieving a fine-tuned clustering mechanism while generating a minimal network/ system overhead and achieving a low latency. There are several issues to be considered and minimized while finding an optimum solution on node management as it helps to improve the availability, reliability and overall performance of the caching system along with the consistency. Clustering or simply maintaining nodes in a distributed environment is always a complex, costly and challenging process in software implementations. It always pays off the consistency or the availability of the system causing the possibilities of intermediate inconsistencies. For example, if a

time frame is considered, a single node may be unavailable and then come back to live, it is considered as unavailable till the next sync up or till the heart beat happens in a polling type protocol.

Here, the system is inconsistent till all the nodes are synchronized with individual status updates. This is an example of a critical condition which needs to be carefully handled in a clustered environment. Especially for data replication and distributed tasks, the above scenario is critical as the system may not behave as expected resulting in duplicates or mismatching data. Also, this can be appeared due to attaching a non-updated node to the system which contains invalid or outdated data. Hence, it is required to maintain a higher consistency level on all replicas to enable accurate cache data sets in the nodes in this type of framework.

Also, if the system is set to wait for the sync up to be happened, before serving the client request, the response time is increased impacting the performance on the overall platform. This is mostly an unacceptable scenario on most of the application use-cases.

1.4 Cluster Management for Caching

Cluster management or node management is the unique identification of each node, logical location, instantaneous status, current property set and their capabilities. Cluster membership is a significant factor to consider when dealing with the strict consistent model [2]. As a result, it is required to populate an update on each active node, making an additional requirement to maintain a registry of nodes. There are two acceptable approaches for this, static node configuration and dynamic node configuration. Node membership, failure handling and configuration sync up can be considered as the sub tasks of the cluster management.

1.4.1 Consistency

Consistency is the process of maintaining the data consistent among all the nodes in the cluster. There are several types of consistency models available from strict consistency model [2] to relaxed consistency models such as eventually consistent model.

There are various types of implementations with various consistency models catering specific applications or set of applications needs. More importantly recent trends on the community are towards more relaxed consistency models focused on ecommerce platforms gaining higher level of performance with enormous data sets. However, still there are strict consistency models catering specific applications which are more restrictive but mandatory for application needs. Consistency model can be freely maintained on top of the clustering framework, where the cluster framework is reliable and responsible enough to be up-to-date with accurate statistics of the nodes.

1.4.2 Static Mode

In static mode, nodes need to be added and removed manually, or at least via an application configuration mechanism. In this scenario, failure handling it is much complex as the node need to be removed manually or ignored till it recovers.

1.4.3 Dynamic Mode

Dynamic clustering is much more complex to implement due to it dynamic behavior. Mainly it is required to scan through multicast which needs dynamic discoverable capabilities as well. However, it has a drawback on high network utilization making the network congested due to the multicasting for node discovery. However dynamic mode allows the systems to self-maintained with self-healing.

1.5 Configuration Management

Software configuration management [3] is the process that defines the procedure of configuring and maintaining the configuration among the application nodes. It is beneficial to all over the software development, maintenance and quality assurance phases of the application. Basically, SCM includes configuration management, release management and change management etc.

In this project, we will be more focusing on configuration management and replication to make sure the all the nodes are synchronized accurately without any configuration conflicts or mismatches.

1.6 Fault Tolerance

Fault tolerance [4] refers to the implementation of safe guard mechanisms in a system to identify the possibilities occurring a failure, detecting them and proving workarounds or fixing them before the system get effected. In this project it is more focused towards the detection [5] and recovery rather than avoidance and prevention. Because the failure in a clustered environment can be more transient and complex which are almost impossible to get isolated for tracking purpose.

1.7 Distributed Caching

Caching [6] is the mechanism of maintaining a copy of frequently required data in a fast and seamlessly accessible storage. In software caches, mostly the specific application memory becomes the storage for cache rather a persistent medium, where the accessibility is very convenient. Basically, the node local caching is a straight forward process. In contrast, distributed caching [7], [8] is more complex and needs much more attention to cater towards the distributed-ness. Especially, it is required to replicate the cache updates, evictions and invalidations real time among the cluster nodes to avoid conflicts of data and maintaining consistent replicas of the cache among the cluster.

1.8 Research Problem

The research problem of this project is to design and implement an efficient and lightweight caching framework with distributed node management. Even though the initially identified requirements of this research project is to design and implement an efficient cluster framework, the framework was extended include industry ready feature to add configuration management.

However still the focus of this project was to research a suitable discovery mechanism for node membership management. Specially to introduce a lightweight synchronization methodology which seamlessly updates the status of the nodes and the cache in a node/ cache registry. Also, it will avoid connect/ replication attempts to failed nodes, minimizing the network traffic generated by the node management.

1.9 Motivation

An efficient and lightweight caching framework is a common expectation in the software industry. More importantly it is expected to be lightweight as it really should be, without significant overheads and resistances to scale up or down. Mostly, the existing solutions are targeting for scaling up or for the large-scale clusters as well as to contain in-build overheads like memory leaks and generating higher network congestion. This happens due to the repetition of metadata and message broadcasting basically. Mostly these behaviors are inherited from the basic design of such systems which are performing very poorly when they are used on general purpose applications. More importantly, having small number of nodes does not mean a non-production ready or poorly design system in software industry, it is yet another system designed as per a requirement, tuned very well for the purpose and should be scaled towards a large-scale system with minimum overheads and limitations.

Finding a cache cluster framework for such system is really a challenging task. By involving mostly on this area of software development, lack of such framework was noticed and motivated to implement such. Designing and implementing cluster membership management [9], with value added feature like configuration management, the application can be taken ahead one step towards fulfilling the industry capabilities. Also, this will provide a great exposure on the trending topics in distributed computing environment.

1.10 Objectives

The objective of this research project is to design and implement an efficient and lightweight node management framework with replicated caching and configuration management. Finding a methodology which helps to manage cluster memberships of the nodes of the system while maintaining the network traffic on the clustering to a minimum level can be described as a key objective among them. In this research a very efficient and lightweight cache cluster framework is to be implemented. Beyond the initial expectations of the cache framework, it is also expected to add a value addition through configuration replication without sacrificing the efficiency or lightweight-ness of the framework.

CHAPTER 2

2 LITERATURE REVIEW

2.1 Overview

Topics like cache management, cluster management, failure handling, distributed caching etc., are hot topics in distributed computing and are being discussed often and being updated since a long period by the community as well as the industry. Many leading and famous software entities have been involved in to this research area to find solutions to specific problems, mostly to their own internal concerns, but also have been contributing to the community as well by introducing kind of general purpose modules, sections or even complete solutions. Even though it is not possible to apply the research outcome of such projects out of the box to all the other general projects all the time, often it is possible to use the components, methodologies or mechanisms towards resolving a common research problem.

In this project, analysis of such related projects is much important and advantageous to find the optimum solution for the research problems considering the appropriate concepts for node management etc. Also, that kind of analysis is helpful to find out any suitable solutions as building blocks for this kind of general purpose solution.

2.2 History and Background

Consistency is a main aspect in cluster management. In past, strict consistency is the norm of distributed systems. However, with the limitations and especially on performance bottlenecks, the consistency models become more and more flexible, which is flexible enough for a specific requirement or a purpose. Node management with strict consistency is very complex with the transient issues and behaviors in the networks. As a result, the industry is moving towards eventual consistency paradigms step by step. In practice, this approach is not much easy and require significant effort in fault tolerance, message synchronization, data validation and data comparison techniques.

2.3 Related Work

Skeleton based tools have a higher demand in distributed computing, especially as building blocks to develop scalable and maintainable applications. Basically, the expectation of a framework is to get the low-level tasks done via a convenient API, so

the application can also be layered or modularized, clearly separating out the tasks on each component.

Ferreira et al. [10] have introduced a Java Skelton based framework, JaSkel. It is an inheritance-based framework and supports various compositions such as orthogonal and hierarchal. Through this composition, JaSkel has been achieved complex parallel applications, such as multi-level farms as well. Also, JaSkel has been focused more towards grid computing [11] with above features.

Borg [12] is a framework that is developed for large scale cluster management in Google. There are three main benefits which are expected to achieve through Borg. Firstly, it hides complexities of the resource management and failure handling behaviors. Then it tries to achieve a very high level of reliability and availability. Lastly, distributing load as micro tasks across thousands of nodes, though the concepts of jobs and tasks. Borg has a very convenient scheduling mechanism which queues the jobs and assign them to the nodes with sufficient resources. To make the processing mode consistent, it provides a naming and monitoring service to be tracked for the work done. Again, Borg focused more on very large and complex cluster management, such as Google like deployments. Hence, most of the features and enhancements are related to large clusters which mostly miss the issues and problems appearing in the applications with small to medium scale clusters.

Initially it is required to look for an efficient node management methodology to implement the cache framework on top of it.

2.4 Group Communication Protocols for Node Management

2.4.1 Multicasting Protocols

Multicasting is a communication mechanism which sends a message from a point to multipoint. Multicasting heavily contributes to the process of dynamic discovering the network resources in a network. A group is considered as the main concept behind the multicasting. A multicast message is expected to be transferred to a group of hosts which also has a multicast group id. Whenever a message is sent out, the destination points are specified by the multicasting group.

Kasera et al. [49] has presented an approach for providing reliable, scalable multicast communication, using multiple multicast groups for reducing receiver processing costs in a multicast session. Here the original transmission of packets is done by a simple multicast group where retransmission is done by separate multicast groups. They have shown that the processing overheads at the receiving end can be considerably reduced by using infinite number of multicast groups. However, they have taken that experiment to a next level by using a negative acknowledgment-based protocol with smaller number of multicast groups achieving the same reduction. At the end they have presented a filtering scheme for reducing the signaling with multiple multicasting groups.

2.4.1.1 Multicasting based Frameworks

JGroups is a well-known, reliable group communication toolkit. JGroups provide an extension of the Java distributed object model based on the group communication paradigm [45]. Distributed object frameworks such as CORBA [46] and Java Remote Method Invocation [47], [48] act as a middleware platform which facilitate the distributed objects to communicate among them via a client/server approach.

The above-mentioned group communication paradigm [45] offers a very reliable communication mechanism which enables reliable and highly available applications using replication. The paradigm is oriented with the concept of a group which consist of set of members. The set of members share a common goal and maintain a replicated status.

The key mechanism in the core architecture of the group communication paradigm is a group membership service integrated with a reliable multicast service. JGroups uses IP multicasting protocol to provide the above behavior. Using IP multicasting, JGroups consistently inform the nodes about the current memberships of the groups to make the nodes cluster aware.

It can be easily used to create clusters that the nodes can communicate with each other using JGroups. Through that, JGroups framework can be used to create as well as delete clusters. The messaging allows to identify the node behaviors such as joining, leaving, membership detections and notification on status of the cluster nodes.

2.4.2 Gossip Protocols

Using Gossip based mechanisms is a preference in distributed database domain for cluster membership management [13]. Mostly these mechanisms are much probabilistic [14], which utilize less resources and distribute the load to the all the nodes making it more efficient as well as failure resilient by avoiding single point of failures.

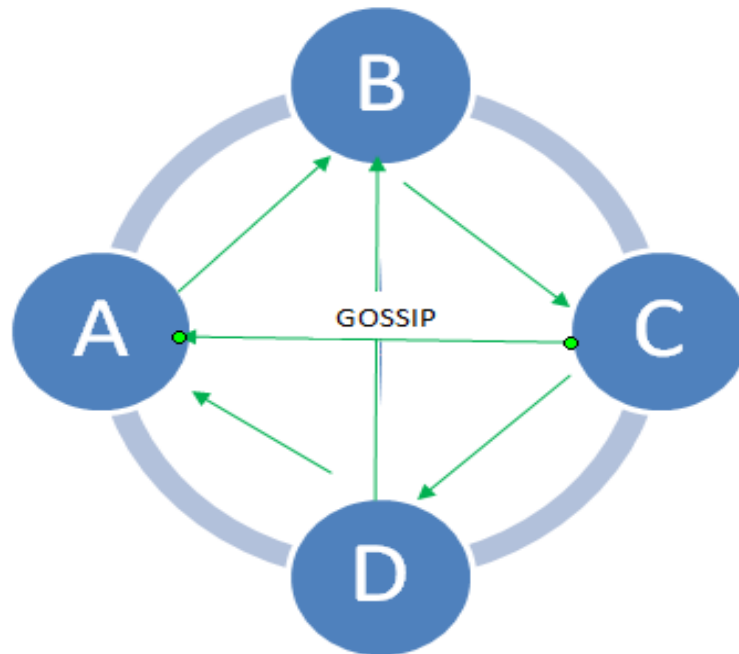


Figure 2.1: Gossip Protocol

There are two types of gossip mechanisms as anti-entropy and rumor-mongering protocols. Anti-entropy protocols continue gossip process till the information is outdated. This mechanism has been used in Cassandra [15] as well as in Dynamo [16]. However, rumor-mongering protocol gossips for a specific period, assuming the configured time is sufficient to gossip on all or configured number of nodes. Anti-entropy has been more popular and used in many cases among them.

Eugster et al. [17] present a decentralized lightweight probabilistic algorithm based on partial views. More importantly they discuss two approaches on membership tracking in gossip-based algorithms. The first method is bimodal multicast [18] which relies on two phases best effort multicast model and peer-based retransmission. The second

method is directional gossip [19] which is targeted for wide area networks. This method maintains a weight for each neighbor which represents the connectivity. Redundant transmission is reduced selecting higher weighted node with higher probability. This algorithm is based on partial views [17], which maintain single gossip server in hierarchical manner.

To enhance the gossip mechanism more scalable and efficient, it is often noticed that a specific methodology is defined namely partial views. Partial view is introduced to make the system more scalable while reducing the cost of maintaining the complete membership entries up-to-date. Partial view is a subset of the entire memberships of the cluster which establish neighboring associations among nodes. A node is responsible for selection a fixed number of nodes as its partial view. However, there is a major drawback of this system. If a large number of node failures happen, the particular nodes as well as the nodes listed in each partial views are disconnected. Then it requires several rounds of effort to reestablish the lost node to the cluster and to refresh the views.

J. Leitao, J. Pereira, and L. Rodrigues [20] purposed a novel approach to implement a gossip-based protocol as well as a membership protocol to minimize the impact of the above drawback. They present two different views called active and passive views, where passive views are backup views which can be promoted to active views upon a notice of a failure.

Avinash Lakshman, Prashant Malik [15] has introduced a decentralized structured storage solution called Cassandra. Cassandra uses a very efficient anti entropy gossip-based mechanism called Scuttlebutt [21] for cluster membership with efficient resource and network channel utilization. The gossip mechanism in Cassandra tracks the existence of the nodes in the cluster directly contacting the node as well as indirectly, contacting the nodes through other nodes.

DeCandia et al. [22] present a more systemic approach for gossiping by maintaining a routing table for nodes in the cluster. Each node directly gossips based on full routing table to other nodes. However, this model has advantages as well as a major drawback. The advantage is that it allows to identify any failure or new addition with less latency

for a small to medium scale cluster. But this would propagate a considerable overhead for maintain a massive routing table with the size of the cluster. Due to that, this approach is suitable to be used with a switchable configuration.

The problem of group membership has been addressed in a different approach by maintaining a view of the group membership in [23]. In the model, view defines the set of processes a member believes are a part of the group at any given time. A member process will always have itself in its view. The group view may also contain application-specific shared state information. A process updates the shared state by using epidemic [24], [25] communication to propagate changes to members currently in its group view. Views also contain internal information for use by the group membership protocols, such as status and timestamps.

2.4.2.1 Gossip Frameworks

Gossip frameworks are built based on the commonly defined gossip protocols and enable to easily utilize those protocols on application level. Even though there are number of gossip protocols defined, only few gossip frameworks available to be used commonly.

Gossip frameworks such as T-Man [26] and Proactive Gossip Framework [27] have been specifically designed to support gossip protocols. Both offer a solution for overlay topology management. Proactive Gossip Framework works on heartbeat synchronization. It works as a P2P network which is an open system usually having the possibility of free-riding. Karakaya M et al. [28] have described the possibility of free riding behavior in P2P system. More often, a peer can be using the P2P network services, but reluctant to contribute back to the network or to the other peers in an acceptable manner. This behavior can be highly decreasing the overall performance level if it's happening heavily.

On the other hand, event driven communication systems like Ensemble [13] and Cactus [29] are not developed to specifically support gossip protocols. Even though they are not entirely supporting gossip protocols, there are many advantages of using event driven behavior in the gossip context. The individual micro-protocols can be used for constructing related gossip protocols. Then by loading the above micro-

protocols dynamically and rebinding them on event handlers, configurability can be taken in to a great extent. However due to the lack of domain specific support such as communication patterns, common structure etc., for each protocol type.

Lin, Shen et al. [30] have presented a hybrid model gossip framework, GossipKit. GossipKit is a gossip-oriented middleware framework for developing gossip-based application development. More importantly it provides an event driven framework with feature rich yet extensible platform and designed to support gossip protocols. Here, it has been defined to overcome the drawbacks of above different models aggregating the beneficial behaviors. It offers a common component architecture with adding module types and connection bindings between modules which is independent from the implemented protocols.

2.5 Client- Server Protocols for Node Management

Client server protocols define the model for data communication between two nodes, defined as a client and a server. Here a node can be any or both above, where the data communication happens as per the request/ response behavior. HTTP is the main client server protocol that in wide use. Several client-server communication methodologies have been defined based on HTTP such as HTTP polling and HTTP long Polling. WebSockets [31] protocol is the trending protocol in client server communication with extended capabilities beyond HTTP.

2.5.1 HTTP Protocol

- **HTTP Polling**

HTTP polling is looking for the server by client in pre-defined time intervals to fetch up-to-date information if available. In HTTP polling, client send a request to a server. Then the server responds with any new messages. If there are no new messages available, the server responds with an empty or pre-agreed format. This flow is repeated by the configured polling interval. HTTP polling generates a significant network overhead in the system as it requires the repetition of the headers in each request.

Therefore, HTTP polling performs comparatively well, if the update receiving frequency is fixed. If the update receiving frequency is high or the update receipt is random or non-predictable, HTTP polling may lead to a communication overhead. Hence, the main drawback of the HTTP polling is sending number of unnecessary requests to the server even there are no new updates to fetch.

- **HTTP Long Polling**

HTTP long polling [32] is a variation of polling technique defined to overcome the above-mentioned drawbacks of simple polling. It is defined to efficiently handle the data transmission between the server and the client. The difference of the long polling is that, it doesn't send an empty response immediately, when there are no new messages. Instead of that, the server waits for a new message to respond or the request is timed out. This offers more efficient solution to the polling, especially when there are less new messages in the system, reducing the number of client requests. Again, if connection timeouts are appearing beyond response, still HTTP long polling is not very advantageous. Hence, HTTP long polling has not resolved all the drawbacks appeared by HTTP polling.

2.5.2 WebSockets Protocol

WebSockets protocol was introduced as an extension, to enhance the client-server communication with several value additions. Victoria Pimentel and Bradford Nickerson [33] have highlighted the significance of WebSockets beyond HTTP polling and HTTP long polling. In real-time data exchange, WebSockets performs well with less communication overhead, providing efficient and stateful communication between client and the server.

WebSockets protocol allows to maintain long term TCP socket connections between clients and servers. More importantly web sockets enable the messages to be instant delivered with a negligible overhead to the system, in real time, bi-directionally and full duplex manner. WebSockets exhibit a unique feature of traversing through firewall and proxies. It can detect the presence of proxy server and automatically sets up a tunnel to pass through the proxy.

Lubbers P. [34] has introduced the WebSockets usage on a web browser capability as HTML 5 web sockets as well, to deliver real time data seamlessly. Especially the web browsers can be take the advantage of full duplex mode to send data in either direction at the same time. Also, the WebSockets provide a significant reduction of network congestion and overheads than a regular duplex system which maintain two connections with polling, by operating thought a single socket over the web.

With the enhanced capabilities of the WebSockets protocol, it is often suitable for network application frameworks to provide more of consistency, allowing massive scalability on real time platforms. Also, it has been emerging as a very reliable, highly performing, real-time ready and promising protocol for internode communication as well, making it a reliable methodology for cluster management.

2.5.2.1 Client- Server Frameworks

Client-server frameworks are another specific type of network application framework build on top of client-server mostly having additional tools, which are useful to develop highly performing and seamlessly scalable applications easily. Further, maintainability, efficient resource utilization and acceptable latency should be major concerns of such frameworks.

There are several network application frameworks build upon client server frameworks, having their own significant capabilities and user advantages. Most of the client server frameworks were used to use HTTP/ TCP protocols for data communication setup. But now they are more of moving towards adding the support of latest protocols like WebSockets to include the features and the value additions provided by them.

The Netty project [35], has been developed to provide an asynchronous event-driven network application framework based on the client server protocols. It has been added WebSockets protocol support recently. It allows the general-purpose applications to be communicated with each other, allowing to develop highly maintainable, performing and scalable protocol servers and clients.

More importantly Netty is a NIO client server framework which heavily simplifies and streamlines network programming. Netty supports WebSockets by an implementation on the Web Sockets API [39].

Yang, J. et al. [43] have also designed and implemented a consistency detection system based on Netty framework. They also have highlighted the advantages on Netty with NIO beyond traditional IO. Similarly, Apache MINA [36] is also a network application framework. It is useful to develop highly performing and scalable network applications easily. More importantly Apache MINA provides an abstract event-driven and asynchronous API over various transports using Java NIO.

2.6 Caching, Distribution and Replication

Caching is a very popular mechanism used by almost all the applications in current days in any of the forms. Caching is the mechanism of storing most frequently required data in quick accessible location. In practice, there are several software caching solutions available. But finding a compatible solution is always tricky when it comes to java in a distributed manner.

JCache [37] is the Java specification for caching mechanism on Java related applications. JCache is a very simple yet straightforward mechanism which fits in to most of the application use cases.

Infinispan [38], Ehcache [42], Hazelcast [41] etc. are the most popular implementations of JCache API, which are most trusted and used by the industry. However most of the above implementations offer local node level caching in community editions but distributed caching as a premium feature.

From that, Infinispan [38] differentiate between a distributed cache over a replicated cache. Replicated cache replicate the cache data in to all the nodes. In distributed caching, few number of additional copies are maintained to provide redundancy and fault tolerance. Infinispan prefers distribution as new caching mode as it provides a far greater degree of scalability than a replicated cache. Also, it is also able to transparently look for the keys in the cluster.

CHAPTER 3

3 METHODOLOGY

3.1 Overview

The figure 03 shows the purposed high-level component diagram for the cache framework.

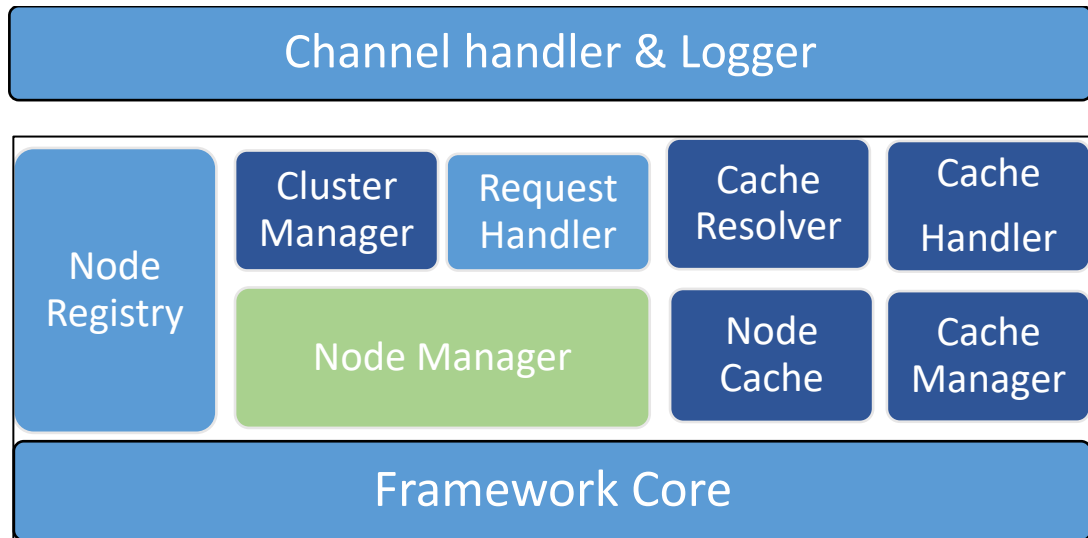


Figure 3.1: Purposed High-level component diagram

Channel handler is responsible for handling individual connection requests through channels. All the steps from connection handshaking client requests are handled by the channel handler. Meanwhile the integrated logger will be logging the connection details.

Node manager is responsible for two main operations in the framework. Firstly, to maintain the server/client behavior in the individual node. It allows the individual node to behave as a server for other client requests while being a client which tries to connect to the other nodes. Then the node manager maintains the cluster memberships which eventually contribute to the failure detection of the system. Node manager performs its tasks with the help of the channel manager which basically monitor and handle the inter-connectivity among nodes. The implementation on this node manager would be based on a Netty framework, which introduced in chapter 2. Also, the node manager is responsible for maintaining required records and attributes of the nodes for clustering. Additionally, it is required perform the cluster information sync up on node addition as well.

Cache handler looks for cache related messages on message flow, filter them out and hand them over to the cache manager. Cache manager is responsible for maintaining a valid cache on the system.

On arrival of cache related message, Cache manager validates the freshness of the cache request. First it checks the content changes of the cache, if there is not any, the message is discarded. If it is different, timestamps are compared to identify the fresher copy. Then the local node cache is updated or discarded accordingly.

Also, the cache manager is responsible for cache warm up on new node addition. Cache warm-up protocol and the cluster info sync up for fault tolerance has been integrated to the framework seamlessly without significant performance tradeoffs.

Component	Description
Channel Handler	Individual connection is created on channels and channel handler is responsible for handling the connection requests received through channels.
Node Manager	Node manager is responsible for maintain the client/ Server behaviors on individual node.
Node Registry	Node registry maintains the status records of the individual nodes of the current cluster.
Cache Handler	Cache handler is responsible for accepting the cache related requests and handover to the manager.
Cache Resolver	Cache resolver is responsible for verifying the cache content before it is being processed.
Cache Manager	Cache Manager perform the cache related operations such as cache validation and update. (and cache replica warmup).
Node Cache	Node cache is the caching store for the individual node.

Figure 3.2: Responsibilities of the Components

3.2 Cluster Membership

The expectation is to use a simple and very lightweight internode communication mechanism considering the solutions currently available the community. In cluster membership, dynamic discovery is a main expectation. Here, a hybrid internode communication model is purposed. Initially the available nodes will be communicated, and the cluster is formed via multiact at bootstrap. After registering nodes in the cluster membership, the node management task will be handed over to WebSockets. This way it is possible to provide the dynamic discoverable cluster framework using the multicast while maintaining them using WebSockets to avoid the higher network utilization by multicasting especially when the nodes are failing. So the challenge is to find some building block applications which supports both protocols.

Therefore, Netty framework is considered for the cluster membership management due to its simplicity, flexibility and the ability to develop scalable and maintainable applications as discussed in literature review. More importantly, Netty supports both multicasting and WebSockets. Also, it is expected to add a proper replica synch up mechanism to allow the system to be more stable while nodes addition and removal. This will help to handle temporary as well as the permanent node failures along with the boot strapping.

3.3 Caching

It is expected to purpose a cache warm-up protocol to maintain a valid and synchronized caching store on each node. This will help to contain a reliable local cache on boot strapping which can be immediately qualify for cluster membership. For an example when a new node is added with an out dated cache, it is required to sync the cache updates of the existing system to the node before exposing it to the real-time user requests. This behavior is exactly similar when adding a node from failure or after a manual removal.

On cache messages retrievals, the cache validation happens, and the local node cache is updated accordingly.

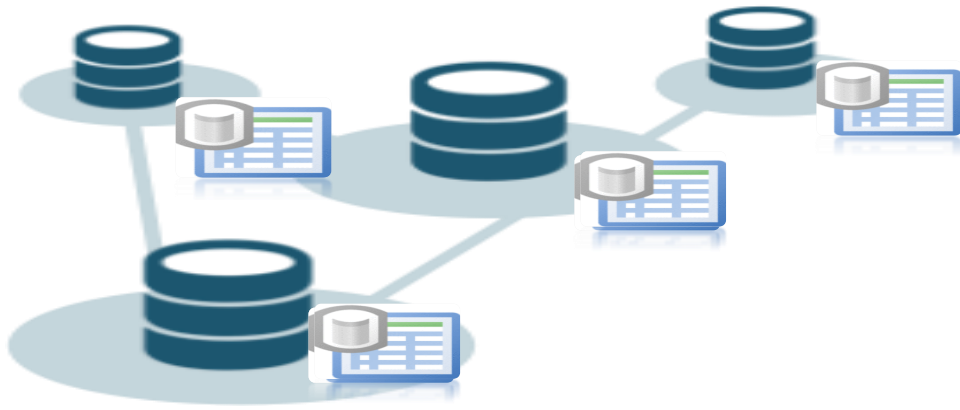


Figure 3.3: Replicated Caching with Node Cache

3.4 Cluster Setup Steps

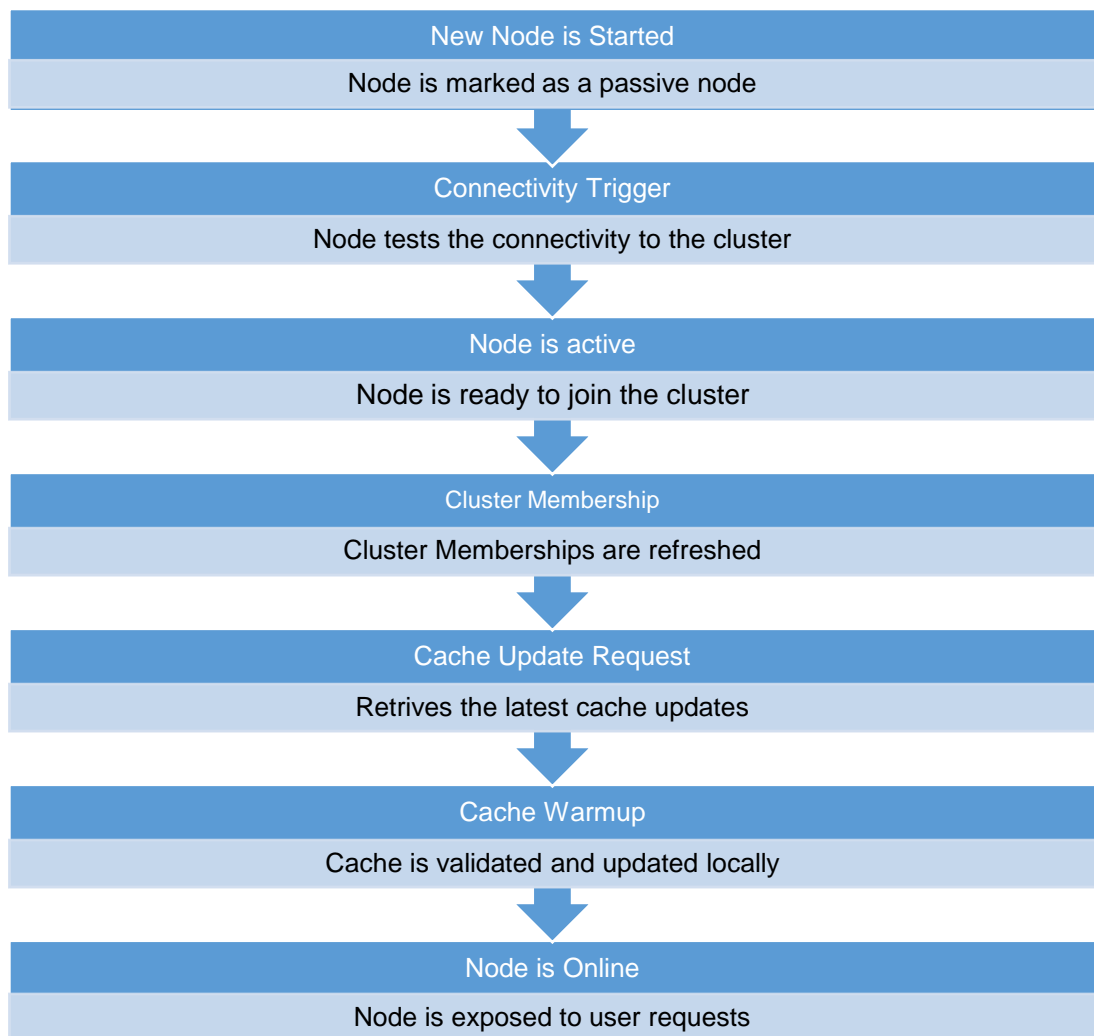


Figure 3.4: Steps of adding a node to the cluster

The high-level new node addition steps are implemented as follows. First the node is marked as “PASSIVE” when it is started. It indicated that the node is started but not ready for serving as it is with outdated information.

Secondly it is required to test the cluster connectivity to make sure the correct cluster information is configured, or the cluster is an active joinable cluster.

After the above step, the node is promoted to “ACTIVE” status making it ready to be added to the cluster. Then the node requests for the cluster membership as a new node to the cluster with a logical identification and a location. If the node is a restarted or a returning node, the cluster membership record is updated accordingly with earlier entries.

Upon the cluster membership retrieval, the node then sends cache update requests asking for the up to date cache data. Then the cache details from multiple nodes are compared, validated and then stored locally. After completing the above steps, the node is marked as “ONLINE” and ready to accept user request.

CHAPTER 4

4 DESIGN AND IMPLEMENTATION

This chapter details out the overview of design and implementation work for the cache cluster framework. These tasks are proceeded with the described methodology in chapter 3.

4.1 Tools Used for the implementation work

Usage of well-suited tools and technologies is a significant fact in this type of research as it directly affects to the behavior of the result. For an example the tools and the protocols need to be well evaluated to extract the optimal outcome of the solution, unless the outcome will contain the unexpected drawbacks or the tradeoffs. Each selected technology needs to be well suited for the purpose as well as they must be compatible among each. That helps to integrate such tools smoothly and allow the platform to be work seamlessly.

In the protocol level, we had to make the exact selection for the communication protocol from the available solutions. As per the characteristics presented in the literature review in chapter 2, we have selected the Web Sockets protocol due to its unique and outstanding behavior from the rest of the protocols, HTTP, P2P or Gossips. With the tools which supports Web Sockets protocol, Netty is having very specific yet powerful feature set on flexibility, integration and usage.

This is the list of toolsets used for the implementation work on this project.

Java Based Technology

- IntelliJ IDEA – Java integrated development environment
- WebSockets – Communication Protocol
- Netty – NIO Client Server framework which supports WebSockets
- Log4j – Logging Framework

4.1.1 Netty Framework

Developing network applications like protocol servers and clients with simple networking aspects is a challenging task in network programming. Especially when providing simple and flexible APIs on top of basic socket related programming

models, it is required to hide the complexities of such low-level behaviors, while offering the highly demanding features of them.

Non-blocking, I/O is a behavior, from what we can benefit a lot in performance aspects especially as we are highly concerning about the lightweight-ness and the simplicity of the purposed framework. NIO allows the application to proceed without blocking by buffering or queuing the request. This NIO behavior is achieved through an implementation on channels. When looking at the API of Netty, we can notice that not only it supports various transport/ protocol types, but also has unified them for usage in both blocking and non-blocking Modes. Design details of the usage of NIO socket channels for our clatter framework is planned for an oncoming section.

For the caching cluster framework, a usage of proper thread model is a mandatory factor. Thread model directly impacts on the performance of the application which uses the framework. For the clustering framework we need to use several thread pools for managing specific servers and the clients. The thread model design of the cluster framework is detailed in a next section. In addition to the above-mentioned design qualities, Netty provides various software properties to be adopted to the client applications, in our case, for the cache cluster framework.

- Security - In production environment, security is a key consideration. Especially in transport layer, secure connection and communication such as SSL/TLS is a must.
- Performance - Less resource (processing/ memory) consumption with low latency and high throughput is a general expectation of an application.
- Stability - facts such as memory leaks or resource misuse tends to unstable a system and make them hang, not catering the service requests.
- Flexibility - It is often needed to change or update the things and integrate easily with other components.
- Scalability - Again in production environments, scalability is a main concern as the applications grow with the customer requirements often.
- Maintainability - Maintainability is a key fact which decides the life time of an application. It is a challenge with application scaling/ upgrading.

4.1.2 WebSockets for Cluster Management

4.1.2.1 Usage of NIO Socket Channels

In both server and client implementations of the cluster framework, NIO socket channels are used. NIO socket channel is an implementation of socket channels which is a logical representation of TCP/IP socket in network programming. Netty also supports blocking I/O as well through a derivative called OIO Socket channel. However, in our project, we do not consider using that as we are more preferred in to the NIO behavior due to the facts mentioned above. More specifically two different NIO socket channel implementations are used in our project for the server and for the client implementations. The following diagram illustrate the class hierarchy diagram for the socket channel classes used in this project.

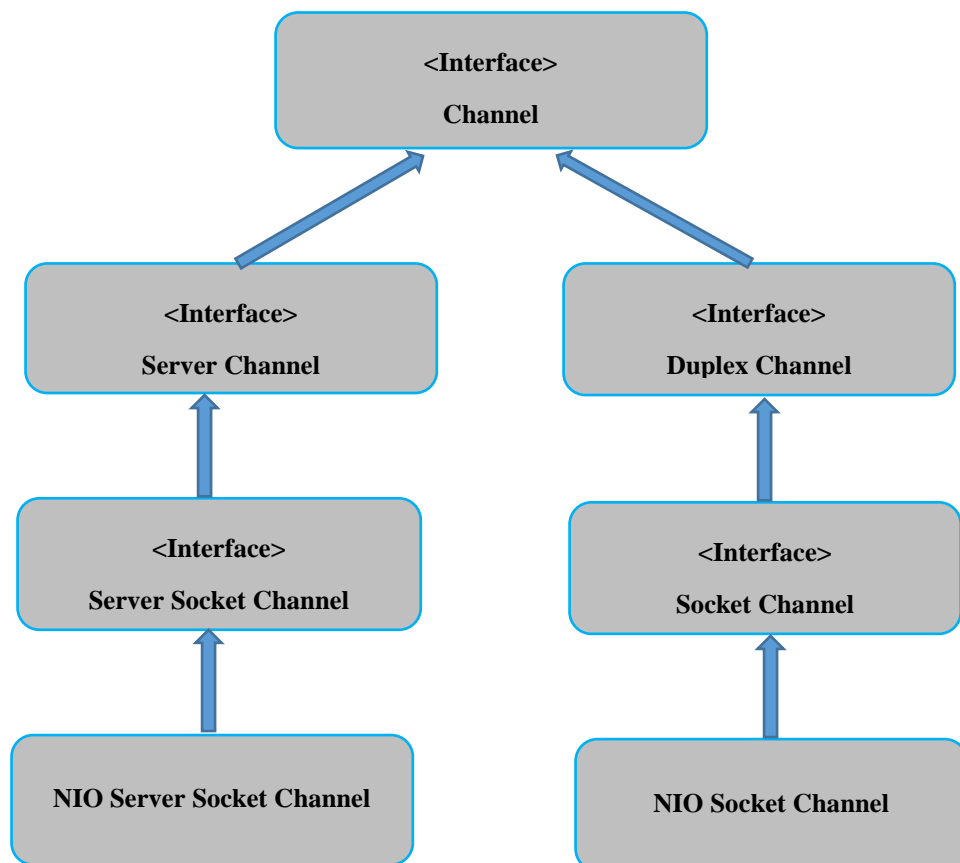


Figure 4.1: Class Hierarchy Diagram for Socket Channels

- **NioServerSocketChannel for Server**

Server needs to allow new connections from the clients as it is needed for cluster setup and management. NioServerSocketChannel is used for the server implementation as it allows to accept new connections.

```
try {
    ServerBootstrap b = new ServerBootstrap();
    b.group(bossGroup, workerGroup)
    .channel(NioServerSocketChannel.class)
    .handler(new LoggingHandler(LogLevel.DEBUG))
    .childHandler(new WebSocketServerInitializer(sslCtx));

    Channel ch = b.bind(PORT).sync().channel();
    NodeRegistry.addNode(new NodeData(ch.id().toString(), hostName));
}
```

Figure 4.2: NioServerSocketChannel usage in Server

Additionally, it is also an extension of the AbstractNioMessageChannel class, which allows to operate on messages which are passing through. As a result, it is helpful for accessing and manipulation the messages transmitted as well.

- **NioSocketChannel for Client**

```
Bootstrap b = new Bootstrap();
b.group(group)
.channel(NioSocketChannel.class)
.handler(new ChannelInitializer<SocketChannel>() {
    @Override
    protected void initChannel(SocketChannel ch) {
        ChannelPipeline p = ch.pipeline();
        if (sslCtx != null) {
            p.addLast(sslCtx.newHandler(ch.alloc(), host, port));
        }
    }
});
```

Figure 4.3: NioSocketChannel usage in Client

NioSocketChannel is an implementation of socket channel which supports duplex channel. Duplex channel allows to shut down the two sides of the channel independently. So that makes this channel variant a great fit for the client implementation.

4.1.3 Framework Thread Model

Developing the framework on a multi-threaded environment is a key concern in this project. Therefore, selecting the exact thread model was really challenging. As this is a multi-node, client/ server and network application, the requirement of defining and maintaining various thread pools was highlighted. For the application each node would have a single server but one or more clients. That is one server to serve the client requests from the other nodes and multiple clients for sending connection requests to other servers (nodes).

Netty provides various set of convenient EventLoopGroup implementations for different purposes. EventLoopGroup is a multi-threaded event executor group which allows registering the channels. In this project we used NioEventLoopGroup as it is a multithreaded event loop with I/O operation support on channels.

Client/ Server Thread Pool vs. Task Thread Pool

It is needed to differentiate between the client/ server thread pool and the task thread pool. Client/ server thread pool is maintained to handle each server and client's instances by individual threads. Here, each client and server will maintain individual thread pools to perform their own communication tasks as described in section 4.1.3.4 and 4.1.3.5.

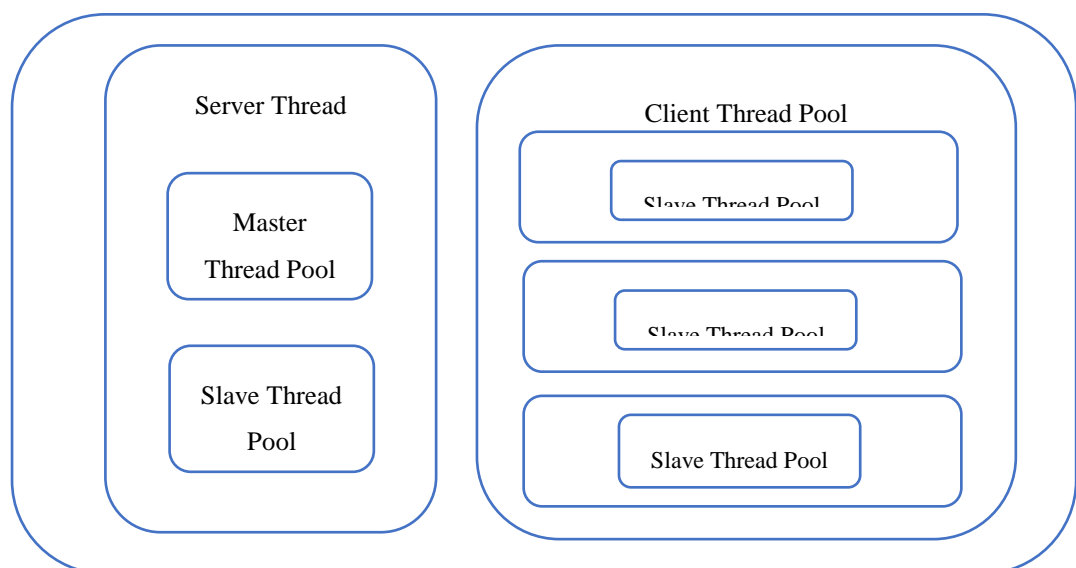


Figure 4.4: Thread Model of the Cluster Framework

Server Thread

Server instances are not created with a thread pool. Instead the main thread of the application creates a separate thread and assign it to start the server instance. Because for a single node, there will be only a one server instance, so a single thread is adequate.

More importantly, the main thread is not utilized to execute the server instance as the server needs to run in an infinite loop, waiting for the communication requests. To cater that main thread initiates a new dedicated thread for the server instance and returned to the main application execution.

Client Thread Pool

Unlike the server, a single node will contain number of client instances each for the other nodes in the cluster. As a result, it is required to maintain a thread pool for client instances. Usually one client instance is started by a single thread from the thread pool and start performing the client related tasks such as connecting to the servers and message communications.

Client thread pool size is not a configurable value, but the application creates a single thread for each configured node in the configuration file.

Server Event Thread Pools

There are two event loop groups defined for a server implementation, master event loop group and a slave event loop group.

Master event loop group is responsible for handling the incoming client requests and registering them to a slave thread. As the reconnections are comparatively small after cluster setup. The pool size can be a minimal standard value.

Slave event loop group come in to the play when the client requests are registered and assigned to the slave threads. Slave threads handles the message transfer traffic of the connected clients. As this is an ongoing process with channel communication, this pool size needs to be carefully configured for optimal performance.

Especially, if the cluster node size is high, it is required to setup a thread pool with higher number of threads to proceed the application functionalities smoothly. Both

thread pool sizes are configurable in `lcf4j.properties` file, which is the application configuration file for the cluster framework where all the custom configuration is setup. `server.master.threadpool.size` defines the Master event loop group which is configured to 2 threads by default. `server.salve.threadpool.size` is for Slave event loop group which is default to 10 thread initially.

Client Event Thread Pool

Client has only a single event loop group, a slave event loop group which is responsible for handling the message traffic of the clients. A master event loop group is not required here as accepting incoming request task is not relevant for the client. We have configured `client.salve.threadpool.size` to have thread count defaults to 2 threads initially.

4.1.4 Message Model

Handshaking

Handshaking happens on both server and client sides in web sockets but initiated by the client. Netty has provided two factories to support Web sockets handshaking protocol for called `WStServerHandshakerFactory` and `WSClientHandshakerFactory`. Above factories pick the correct handshake implementation based on the handshake request.

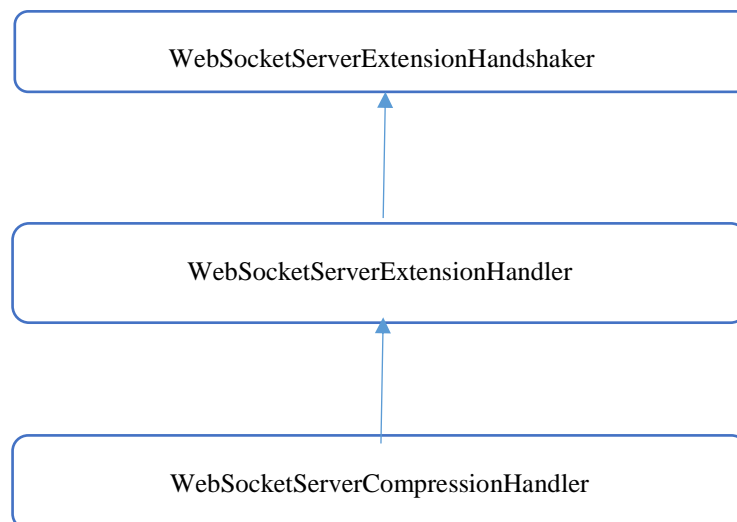


Figure 4.5: Handshaker usage through handlers

In our project, we have used the server handshaker, setting a referencing handler to the SocketChannel on Channel initialization through a ChannelPipeline. Then the handshaker is being called through a handler as shown in the figure 4.5.

In client implementation, the above handshaker usage with handler is similarly implemented. Always a new handshaker is created and set through the WebSocketClientHandler additionally.

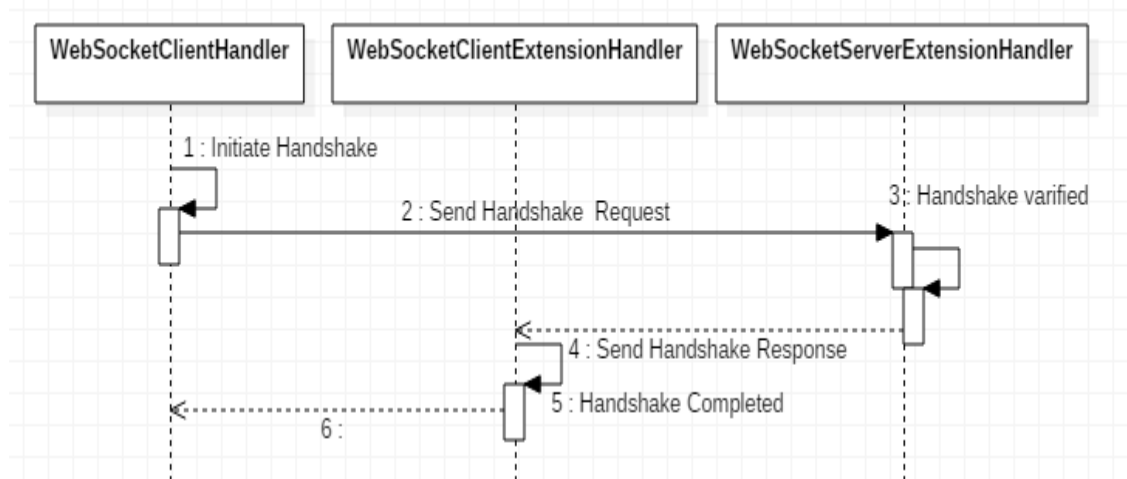


Figure 4.6: Client/ Server Handshake

This is because, as shown in the above diagram, when a new connection request is sent, the client initiate the handshaking. For that we need to create a new handshaker using the handshaker factory first. Then the server responds back with server handshaker handler and the handshaking completed receiving and handling the above response by client using the client handshake handler.

A sample client handshake request is as the figure 4.7.

```

GET /chat HTTP/1.1
Host: example.com:8000
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Key: dGh1IHNhbXBsZSBub25jZQ==
Sec-WebSocket-Version: 13
  
```

Figure 4.7: Client Handshake Request [40]

A sample Server handshake response is as figure 4.8,

```
HTTP/1.1 101 Switching Protocols
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Accept: s3pPLMBiTxaQ9kYGzzhZRbK+xOo=
```

Figure 4.8: Server Handshake Response [40]

Message Passing

In WebSockets protocol, the data is transferred between client and server by using an entity called Web Socket frames. For each specific messaging, it has been implemented different variants of the Web Socket frames. In this project we have used the defined frame type for our communication process which is described below in the coming sections.

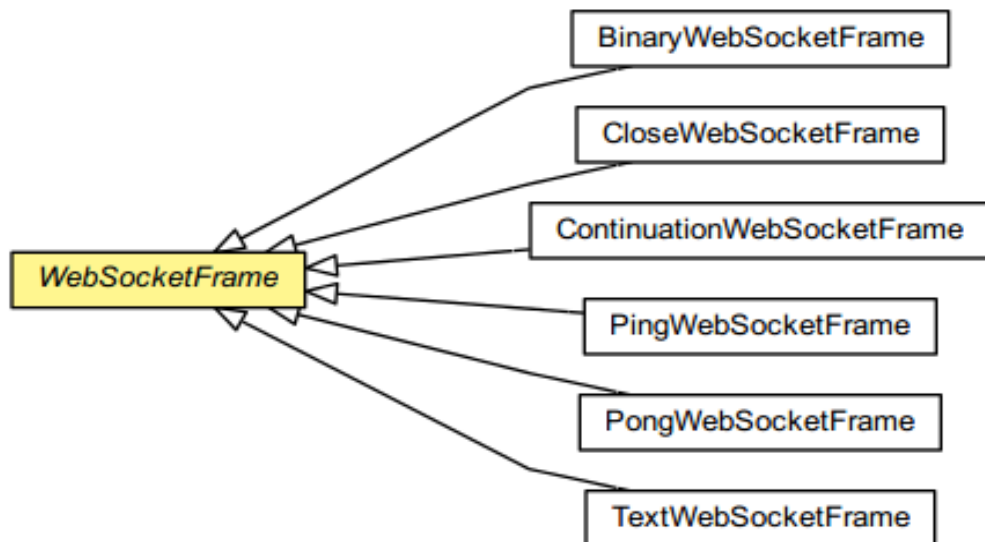


Figure 4.9: Types of WebSocketFrame

- **BinaryWebSocketFrame** – This is a data Web Socket frame containing binary data. We have not used this frame type in our cluster framework for cluster setup or maintenance.
- **CloseWebSocketFrame** – This is a control Web Socket Frame for closing the connection. As we are not supporting manual removal of nodes from the cluster, this frame type is also not used.
- **ContinuationWebSocketFrame** – This is used as a control frame, when the data sent fits to more than one frame, to indicate that continuation of data is available. This is also required for advanced behaviours so our cluster framework is not using this type.
- **PingWebSocketFrame** – Control Websocket frame containing binary data to indicate a heartbeat request. This frame type is used to verify the existence of available connections triggered after an idling timeout in our cluster framework.
- **PongWebSocketFrame** – We are using this control frame as a response to the ping heartbeat request to indicate that the connection is live and active.
- **TextWebSocketFrame** – This is the data basic frame used to transfer messages between the nodes. In our project we use this frame to exchange the active node list data, global node list data and to replicate cache data and metadata.

4.1.5 Security measurements in the Framework

Application security measurements are the next key concern on our project. Following are two main methodologies which can be used to setup security on the cluster framework.

SSL/ TLS

Netty supports SSL/ TLS out of the box. Therefore, we can use HTTP or even HTTPS connections on WebSockets handshaking. In our project we used only HTTP for the implementations and used the WebSocket Security instead of SSL/ TLS.

WS/ WSS

WebSockets work on two schemes plain WebSockets layer and the WebSockets Secure layer. More importantly, it is always possible to use HTTP and adopt to WSS

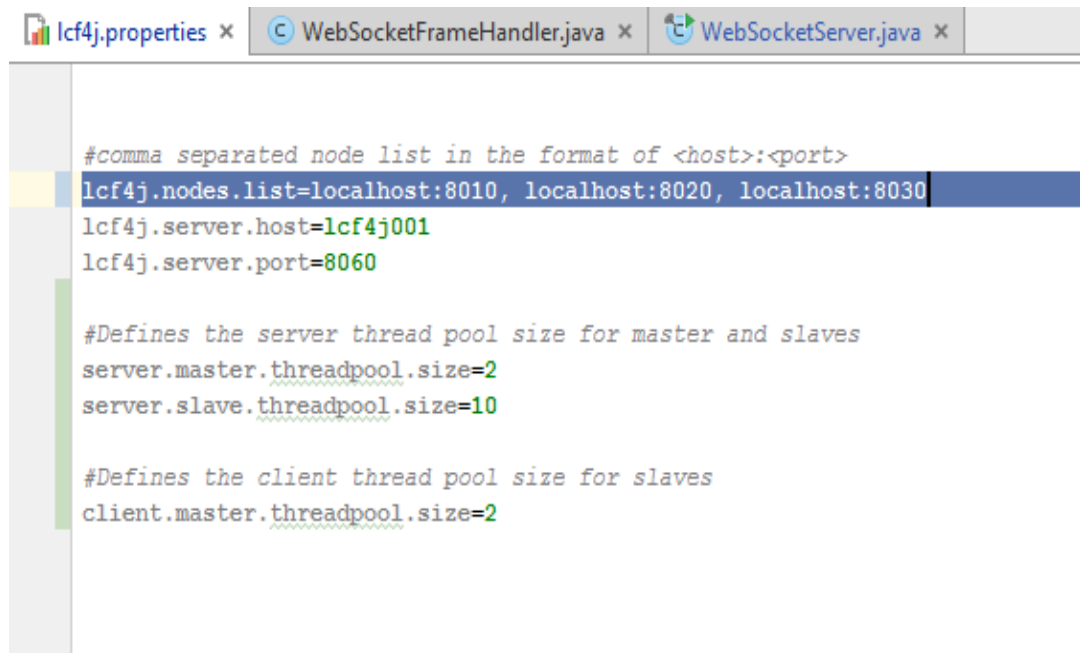
on top of that if required, which confirmed secure message transferring through encryption similar to HTTPS. WebSocket Secure scheme uses the same encryption as HTTPS (TLS/SSL).

In our project we used WSS though self-signed certificates. Mostly in production environments, self-signed certificates are not recommended, and it is required to setup a security key and configure it to be used with WSS.

4.2 Node Implementation

The framework defines two modules in each node, sever module and the client set module. Server Module contains a single instance of server behavior as it represents the node to cater the client requests.

Usually, a node needs to maintain a client set, of which it creates to connect to the listed servers in the configurations. Each server configuration will result in a client instance and the created client instance tries to connect to the above-mentioned server to form the cluster connectivity. Each node contains `lcf4j.properties` file, which is the configuration file for the cache framework application.



```
lcf4j.properties x WebSocketFrameHandler.java x WebSocketServer.java x
#comma separated node list in the format of <host>:<port>
lcf4j.nodes.list=localhost:8010, localhost:8020, localhost:8030
lcf4j.server.host=lcf4j001
lcf4j.server.port=8060

#Defines the server thread pool size for master and slaves
server.master.threadpool.size=2
server.slave.threadpool.size=10

#Defines the client thread pool size for slaves
client.master.threadpool.size=2
```

Figure 4.10: `lcf4j.properties` Configuration File

4.3 Server Implementation

Server module basically initiate as per the local node configuration settings and start listening the connection requests. First it handles the handshake requests sent by the clients. Then it handles the messages sent by the connected clients. The work flow of the server module implemented in the cache cluster framework is shown in figure 4.11.

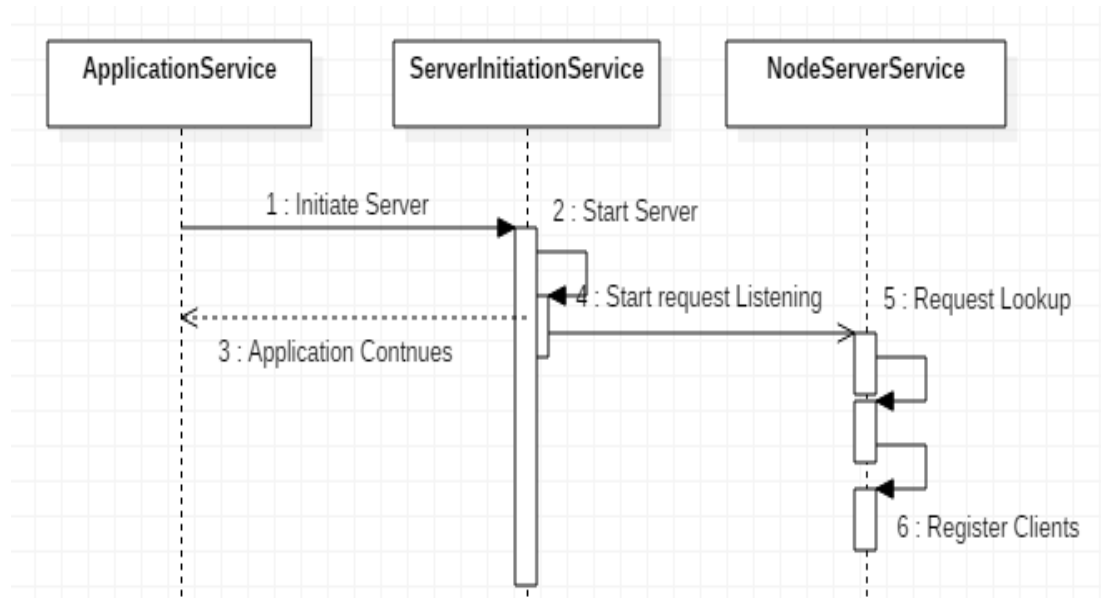


Figure 4.11: Server Module Work Flow

4.4 Client Implementation

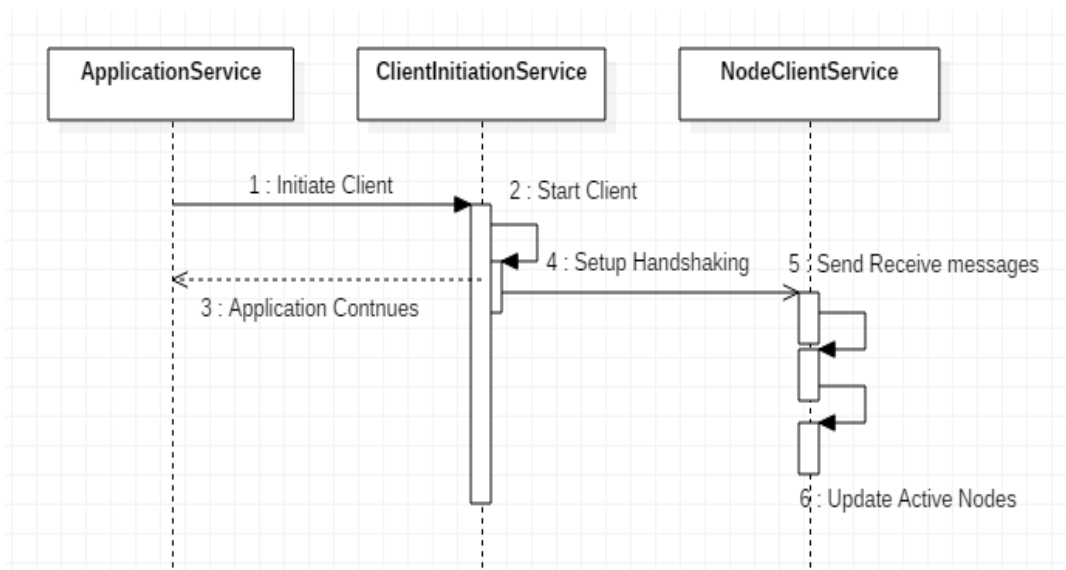


Figure 4.12: Client Module Work Flow

Client Module initializes a set of clients to communicate with the set of servers listed in the configurations. Then each client initiate handshaking with the relevant server. Once a success handshake is happened, the client adds the server node to the active node list, Node Registry.

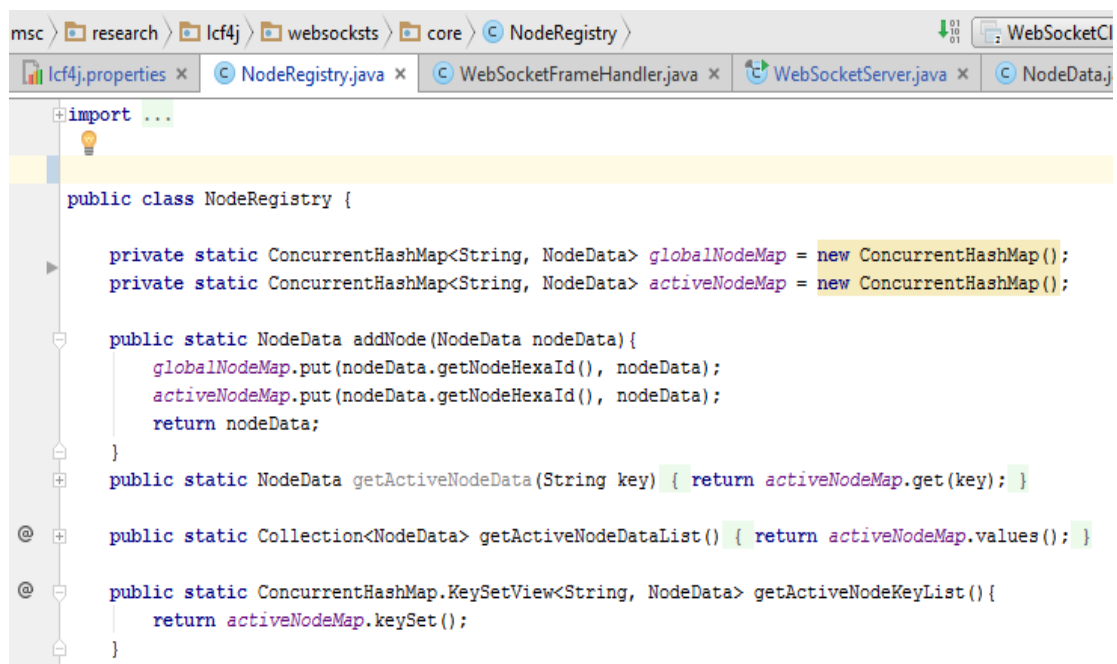
Usually clients often send ping frames as heartbeat messages to verify the node availabilities and update the Node registry.

Meanwhile the node registry data is replicated and validated among nodes to make the cluster more consistent and to identify any new nodes added beyond the initial configurations.

4.5 Node Registry

Node Registry contains two data lists, active node map and the global node map. Active node map contains real time up-to-date node information at a given time. However, the global node map contains all the nodes connected to the cluster from the initialization of the cluster.

ConcurrentHashMap data type was used to maintain the node data sets in node registry to make sure the thread safety of the application data.



```
import ...

public class NodeRegistry {

    private static ConcurrentHashMap<String, NodeData> globalNodeMap = new ConcurrentHashMap();
    private static ConcurrentHashMap<String, NodeData> activeNodeMap = new ConcurrentHashMap();

    public static NodeData addNode(NodeData nodeData) {
        globalNodeMap.put(nodeData.getNodeHexaId(), nodeData);
        activeNodeMap.put(nodeData.getNodeHexaId(), nodeData);
        return nodeData;
    }

    public static NodeData getActiveNodeData(String key) { return activeNodeMap.get(key); }

    public static Collection<NodeData> getActiveNodeDataList() { return activeNodeMap.values(); }

    public static ConcurrentHashMap.KeySetView<String, NodeData> getActiveNodeKeyList() {
        return activeNodeMap.keySet();
    }
}
```

Figure 4.13: Node Registry Implementation

4.6 Managing Cluster Memberships

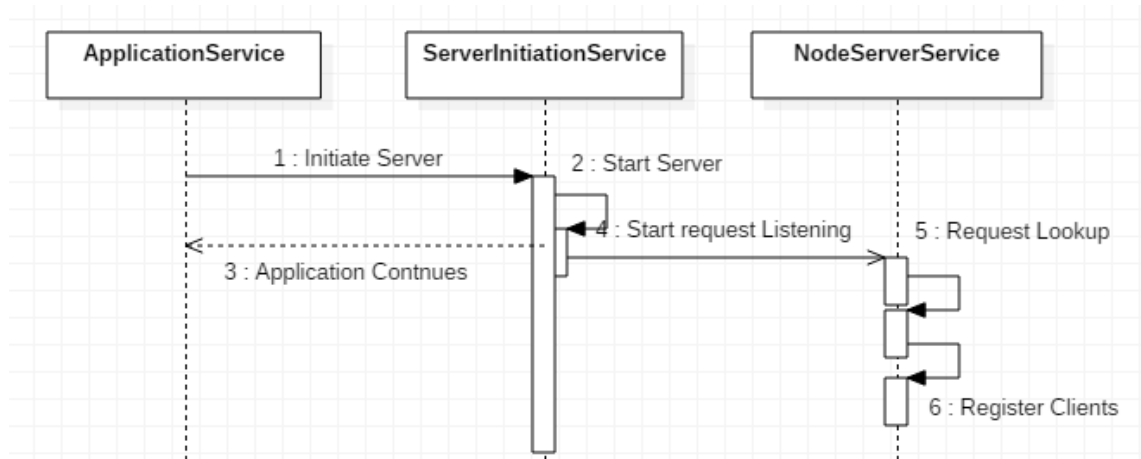


Figure 4.14: Cluster Membership Management

Cluster memberships of the cache framework is handled via the WebSockets protocol. Each node is configured with all the node details in the cluster as a configuration, so the node can individually communicate with other nodes.

Every node contains both server implementation and the client implementation. In that way, each node is responsible for acting as a server to serve for the connection requests as well as to scan for the other server nodes behaving as a client.

4.6.1 Phase 1: Cluster Setup

Node Server Setup for Cluster

Initially there are one or more nodes started and waiting to be added to or to form a cluster. In the first step, the cluster resolving service triggers the server handling. Each node has a node server module, where it should listen to the incoming client requests and mark the presence.

Here, the node is configured as per the configuration and start listening for the requests.

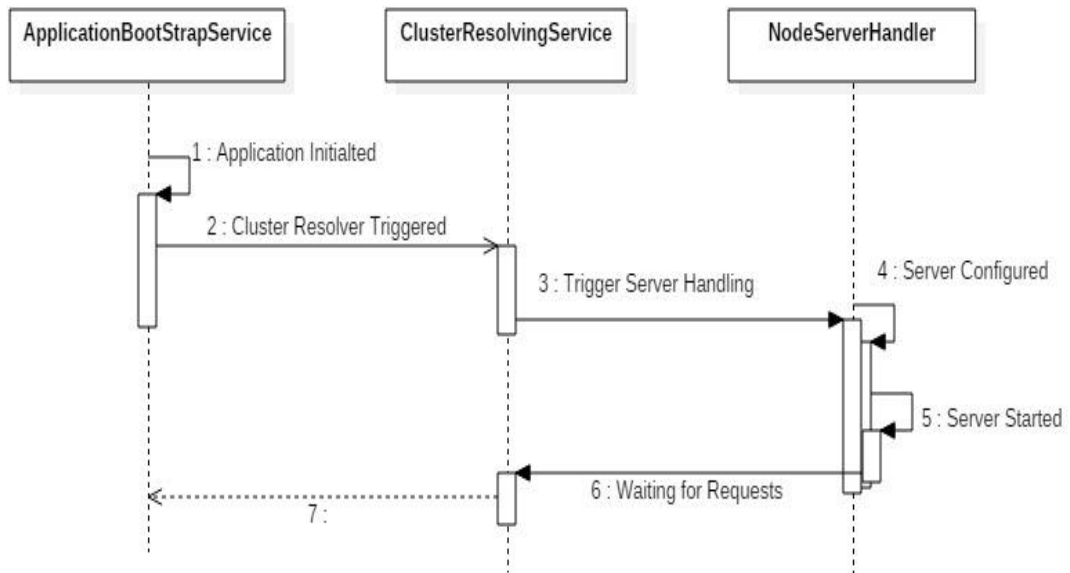


Figure 4.15: Cluster Setup

Cluster Sync Service

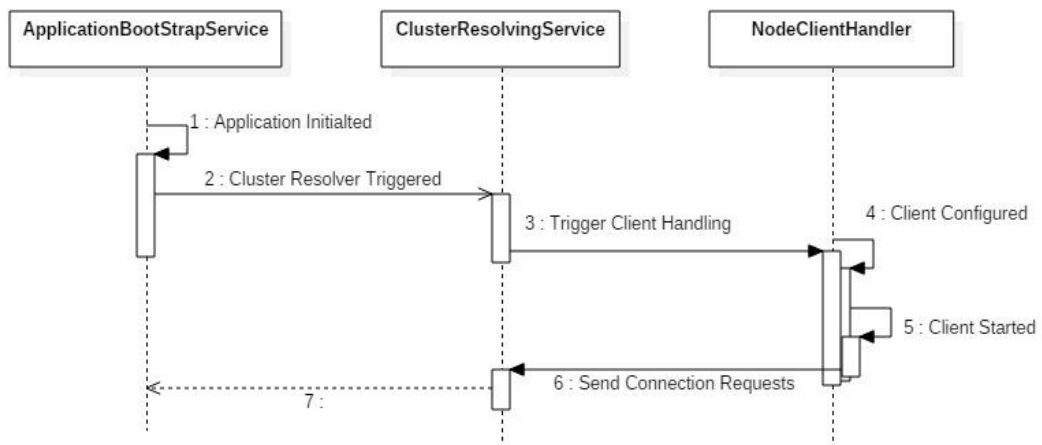


Figure 4.16: Cluster Sync up

Similarly, in the second step, cluster resolving service triggers the client handling. The Client handler configures each client as per the configurations per server.

Then the client is started and send connection requests to the servers to look for their presence.

Node Registry Update

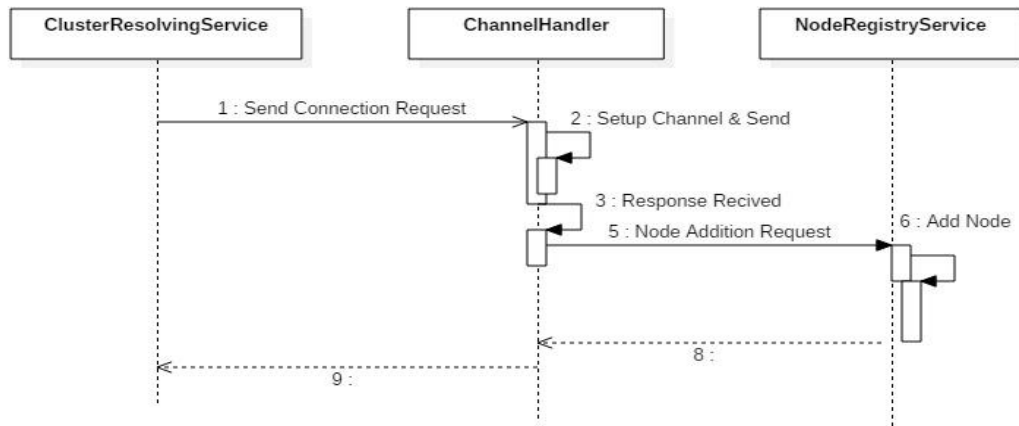


Figure 4.17: Node Registry Update with Success Response

The node registry is updated accordingly with the results from the clients. Also the connection requests are scheduled to repeat in configured intervals (*cluster.connect.interval* property) to track the presence real time. Each node maintains a node registry locally which contains the cluster metadata.

4.6.2 Phase 2: Cluster Sync up

Adding New Nodes

Adding a new node is a straightforward process in this framework. It just starts its node server and start scanning other nodes in the list. Through that method, the new node eventually adds to the current cluster seamlessly.

However even though the node is added to the cluster, it is being held in the cluster in active mode until it is heated up with the current cache set before moving it to online mode. A new node contains an empty cache set as it is not exposed to the user requests yet. Here, the cache resolution is bit easy as it does not have to deal with cache comparison on existing cache. It only needs to find the latest cache version from the retrieved cache sets.

After completing the above two steps, Node resolver service moves the node to active mode allowing to serve the real-time traffic.

4.6.3 Detecting Node Failures

Node failure detection is also a straight forward process to implement. When a node in the list is not responding, it is immediately removed from online node list. Basically, the re-connectivity process is continued with the configured interval. This is considered as a soft failure.

More importantly, after a configured number of retries, the node re-connectivity process is halted. If the failure continued till this step, it is considered as a hard failure.

4.6.4 Node Recovery

Once an existence of a node is detected, it is evaluated for a fresh node or a recovering node. This is decided by traversing through the node registry, if the node is found with the passive status, it should be a recovering node.

In the process of recovery from a soft failure, the node is active and cache warm-up is immediately started. However, a recovery after hard failure is bit different. The considered node will be added as a new node to the cluster initiating the cluster resolving sequence from the beginning. After successfully completing the above process, the node is moved to online mode and allowed to the real-time traffic.

4.7 Caching

Caching service is responsible for cache replication management after the initial cluster warm up. As the cluster nodes are fixed and recorded in node registry, cache needs to be replicated accordingly.

First, cache sync service sends a cache update request to fetch the cache records from the other nodes. Upon a response retrieval, cache resolver service compares, validates the cache.

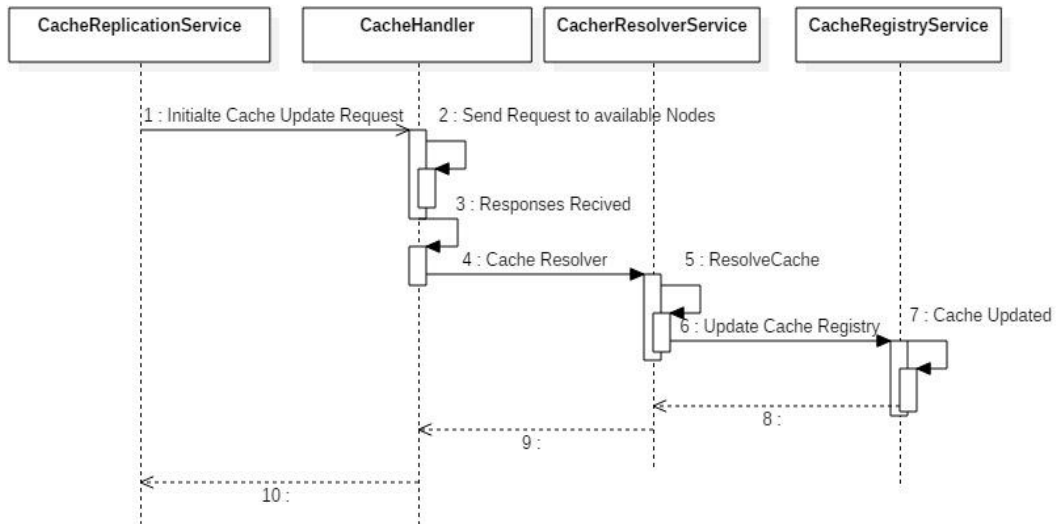


Figure 4.18: Cache Replication after Cluster Setup

Finally, the local node cache registry is updated making a synchronised cache in the cluster.

CHAPTER 5

5 EVALUATION AND RESULTS

5.1 Tests and Results: Using Cache Cluster Framework

We have exactly developed and tested the system by the design and implementation described in chapter 4 using the methodology presented in chapter 3. We have tested the cluster framework with following configurations,

- On Linux Cent OS Virtual Environment
 - Intel Core i7 CPU @ 2.80 GHz
 - 12GB Memory/ 4 CPU Cores assigned
- 6 Cluster Nodes
- Default Thread Pool Configurations
 - server.master.threadpool.size=2
 - server.slave.threadpool.size=10
 - client.master.threadpool.size=2

5.1.1 Node Server Startup

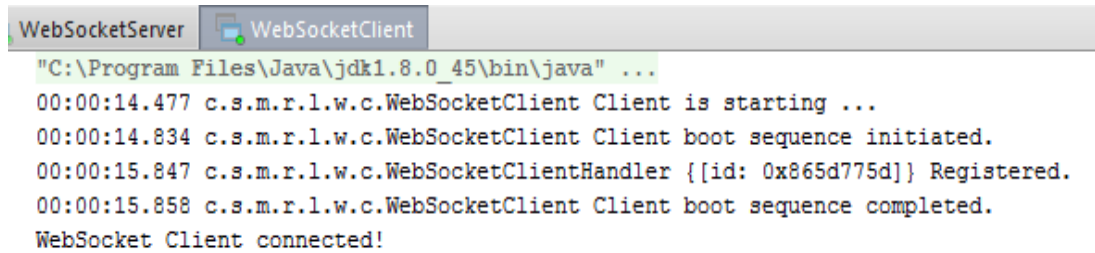
```
23:43:57.225 c.s.m.r.l.w.s.WebSocketServer Application Starting ...
23:43:57.231 c.s.m.r.l.w.s.WebSocketServer Property file loaded successfully.
23:43:57.551 c.s.m.r.l.w.s.WebSocketServer Server boot sequence initiated.
23:43:58.537 c.s.m.r.l.w.s.WebSocketServer Server boot sequence Completed.
23:43:58.553 c.s.m.r.l.w.s.WebSocketServer [ Node Count : 1] [ Active Node List : [NodeData{remo:
23:43:58.553 c.s.m.r.l.w.s.WebSocketServer [ Node Count : 1] [ Active Node List : [2f911419]]
23:43:58.554 c.s.m.r.l.w.s.WebSocketServer [ Node Count : 1] [ Global Node List : [2f911419]]
|
```

Figure 5.1: Node Server Startup

Once we start the application, the server boot sequence is initiated and completed. We could notice that this boot sequence process is completed less than a single second. This included resource preparation, setup, thread pool initialization, channel/ socket configurations, opening etc.

Therefore, this process is well optimized and tuned for better performance expectations. This could be achieved through the NIO capabilities provided by Netty framework which is adding a considerable value addition to our Cache Framework.

5.1.2 Node Client Startup



```
WebSocketServer  WebSocketClient
"C:\Program Files\Java\jdk1.8.0_45\bin\java" ...
00:00:14.477 c.s.m.r.l.w.c.WebSocketClient Client is starting ...
00:00:14.834 c.s.m.r.l.w.c.WebSocketClient Client boot sequence initiated.
00:00:15.847 c.s.m.r.l.w.c.WebSocketClientHandler {[id: 0x865d775d]} Registered.
00:00:15.858 c.s.m.r.l.w.c.WebSocketClient Client boot sequence completed.
WebSocket Client connected!
```

Figure 5.2: Node Client Startup

Similarly, the above log content shows that the client boot sequence is completed almost in a single second. The time taken for the client is slightly above the server boot sequence completion time.

However, this may look bit strange at once as a client, but when we look more in to the client's task sequence, we can see that the client needs to perform few more tasks than the server. Client needs to do the exact resource/ connection related preparations less the master thread pool setup. Then the client is also responsible for initiating the handshaking request for connectivity. Therefore, the client boot sequence is completed only after sending the handshake request as well.

Again, the client as well heavily benefiting from the NIO nature provided by Netty framework, which is adding huge performance advantage on our cluster framework.

5.1.3 Client Failure Sync up

To simulate this behavior, we used a `CloseWebSocketFrame`, triggered by a user interaction.

```
00:35:54.300 c.s.m.r.l.w.c.WebSocketClient Client disconnect initiated.
00:35:54.323 c.s.m.r.l.w.c.WebSocketClientHandler WebSocket Client received closing
WebSocket Client disconnected!
00:35:54.326 c.s.m.r.l.w.c.WebSocketClientHandler {[id: 0x4d604866]} Unregistered.
```

Figure 5.3: Client Dropout

```

00:35:54.323 c.s.m.r.l.w.s.WebSocketFrameHandler [a6712421 {127.0.0.1:54540} ] is INACTIVE.
00:35:54.324 c.s.m.r.l.w.s.WebSocketFrameHandler [ Node Count : 1] [ Active Node List : [NodeData{remoteAddress
00:35:54.324 c.s.m.r.l.w.s.WebSocketFrameHandler [ Node Count : 1] [ Active Node List : [3e9e74f5]]
00:35:54.326 c.s.m.r.l.w.s.WebSocketFrameHandler [a6712421 {127.0.0.1:54540} ] is OFFLINE.
00:35:54.326 c.s.m.r.l.w.s.WebSocketFrameHandler [ Node Count : 2] [ Global Node List : [3e9e74f5, a6712421]]

```

Figure 5.4: Server sync up on Client Dropout

As we mentioned, the Cache Cluster Framework performs well on failure handling. As presented on the above diagrams, the client dropout is noticed by the server by 23 milliseconds, which is an outstanding behavior of the framework. On the server side, the Node registry takes only 3 milliseconds to locally update its node status lists.

These values for failure identification can be considered as very significant as these measurements can be hardly achieved by any different protocols. For an example in HTTP Polling, the server will be notified on this node failure on next polling cycle. In gossip-based environment, it would be bit efficient due to the gossips but still need to wait till the next gossip happens. However, in either case, the above failure detection time may not be achieved.

5.1.4 Messaging

In our project, there are two main aspects of communicating between clients and the server unless control messages, for cluster Sync up and cache replication.

Cluster Sync up Messages

Cluster sync up messages are used for syncing up the nodes in the cluster in between. It is true that the nodes maintain their own client network for connectivity tests and maintain a node local registry. Here, the sync up messages provided a verification mechanism among nodes. For an example if a node needs to verify its content against other nodes, it can request for a cluster sync up message, then it can verify its contact against the response and make updates as required. This comes in to the play when a new node is added to the cluster which was not configured in the cluster configuration initially. Hence, the only method to get the other nodes informed on this new node is through cluster sync up messages.

Cache Replication Messages

Cache replication messages transfer the cache data and perform cache replication among nodes. Like the user case described above these messages are used to verify the cache content as well as to fetch the cache copy for a newly added node.

As per our tests, the message transfer time from the client to server averages to 80 milliseconds. However the server took only 6 - 11 milliseconds average to respond for the client request. This statistics are almost similar for both cluster sync up and cache replication messages.

5.2 Assessment of the Advantages of Cache Framework

As per the expectations of this research project, we could achieve all the initial concerns. As described in the literature review in chapter 2, most of the available frameworks contain fine grained cluster support for the application but they are coming with additional features and hence with a considerable amount of overhead. While we closely look in to above factors, we can notice that they are implemented not in a mind to give a flexible cache cluster framework, but to provide a distributed cache or similar enterprise feature set.

The rest is fine tuned for large to very large systems with several hundred to thousands of nodes. Those applications perform well in such environments but could not achieve adequate performance level on small to medium scale clusters. Therefore, our research is well focused on the above matter and the simplicity and the lightweight-ness was associated from the design phase.

The main advantage of this cluster framework is that, we can port this in to any Java application quickly to provide cluster support while the framework injects the cluster awareness to the application seamlessly.

Cluster aware-ness means that the knowledge of the cluster topology with node data and using it for the application related tasks to distribute the work load or make a separation of concerns. With all the above-mentioned aspects, our work is full filling a framework gap in the industry, which can be applicable for most of the distributed cluster supported application development.

5.3 Feature Evaluation

This section focuses on evaluating our work on the expected behaviors and common features. The application is also analyzed with the measurable output received throughout the evaluation.

5.3.1 Ease of usage

Evaluation of our work has clearly shown that the cluster framework is very simple in integration as well as in usage, which makes it very easily use for client applications. As we detailed out in the introduction section, the consumers of this framework would be small to medium scale applications, which needs a simple clustering solution for basic cluster related operations. We could achieve the ease of use in different aspects as follows,

Cache Framework Integration: Framework integration to a client application programmatically is a simple and straightforward task. The client application only need to hook the framework startup access point in application bootstrap trigger. Once it is attached, the cache framework will initiate the node cluster via server and relevant client set triggers.

Configuration: We have made the cluster configuration very simple by setting up the only the minimal settings with a default value for standard setup. The default configuration needs to be changed only if the cluster is setup for a huge number of nodes. In such case, the configuration needs to be set with fine-tuned values for maximum performance.

For an example, in such a scenario, slave thread pools size needs to be updated accordingly. All the other thread pool handling operations will be performed internally.

```

1
2
3  #comma separated node list in the format of <host>:<port>
4  lcf4j.nodes.list=localhost:8010, localhost:8020, localhost:8030
5
6  lcf4j.server.host=lcf4j001
7  lcf4j.server.port=8060
8
9  #Defines the server thread pool size for master and slaves
10 server.master.threadpool.size=2
11 server.slave.threadpool.size=10
12
13 #Defines the client thread pool size for slaves
14 client.master.threadpool.size=2

```

Figure 5.7: Thread Pool Configuration

Retrieve Cluster Data: In our work, local cluster data retrieval is a simplified via providing a simple API via the Node Registry. Application can directly get the node statistics calling the controlled API.

Perform Cluster Sync up: Cluster Control commands such as sync requests and responses can be triggered via NodeSyncService directly. Once triggered the service will sync up with the other nodes, update local Node Registry and returned the real-time cluster data to the calling application.

```

ress= null , registeredTime=Tue Dec 19 16:23:39 IST 2017, NodeData{remoteAddress= 127.0.0.1:35003 , registeredTim
}, 64857935, 7b225879]]
}, 64857935, 7b225879]]

ress='null', registeredTime=Tue Dec 19 16:23:39 IST 2017}, NodeData{remoteAddress='127.0.0.1:35003', registeredTim
}, 64857935, 7b225879, b932fbee]]
}, 64857935, 7b225879, b932fbee]]

ress='null', registeredTime=Tue Dec 19 16:23:39 IST 2017}, NodeData{remoteAddress='127.0.0.1:35003', registeredTim
}, e60832c6, 64857935, 7b225879, b932fbee]]
}, e60832c6, 64857935, 7b225879, b932fbee]]
received {[ Node Count : 6] [ Global Node List : [beed14e9, 7c2c1268, e60832c6, 64857935, 7b225879, b932fbee]]}

```

Figure 5.8: Cluster Sync up Message

5.3.2 Lightweight-ness of the Framework

Our work has demonstrated a very lightweight behavior on usage. This lightweight-ness is highlighted in both aspects such as application Load to the system and in the network overhead.

Minimum Impact on System Load: Our caching cluster framework shows very reasonable impact on the system load. More importantly, the load is applied on server start and the client handshaking. Interestingly after completing the startup and the handshakes, the system load is set to normal and both the system and the cluster framework is stable.

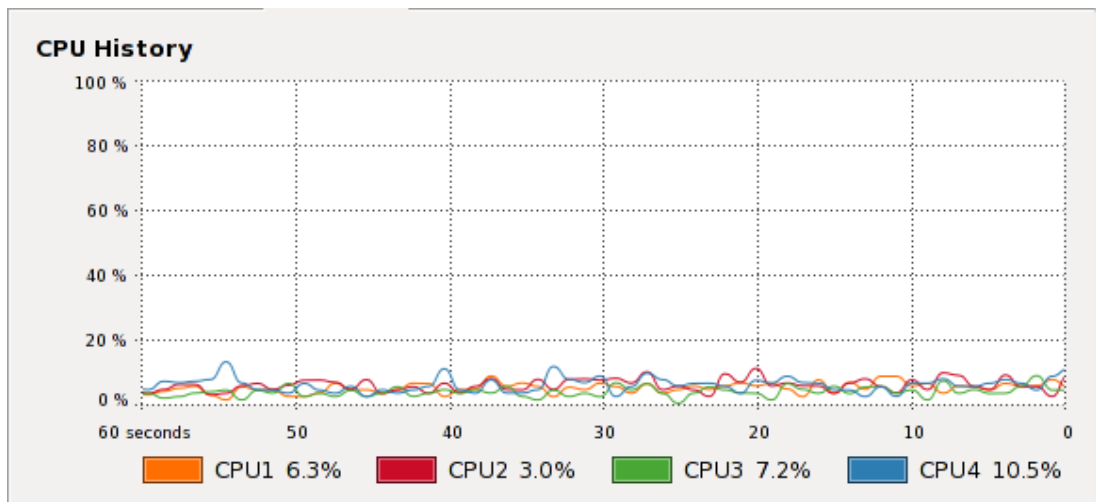


Figure 5.9: CPU Load on System Idling

Idling CPU load is around 5% and it is stable before the cluster setup happens as shown in the figure 5.9.

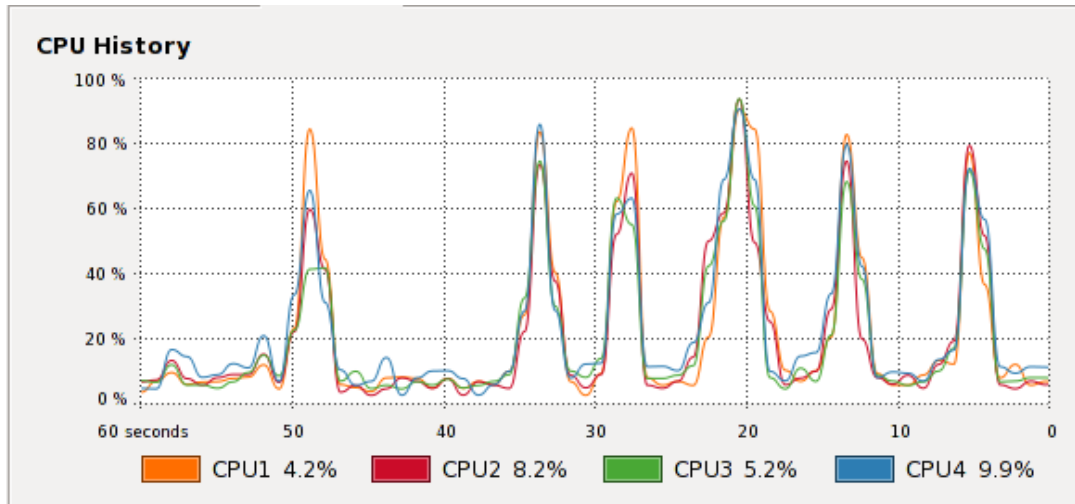


Figure 5.10: CPU Load on Cluster Setup

Here initially the server is started and then the clients are connected to the server node. In server start as well as while individual client connected, the CPU shows sudden spikes and returned to the idling state.

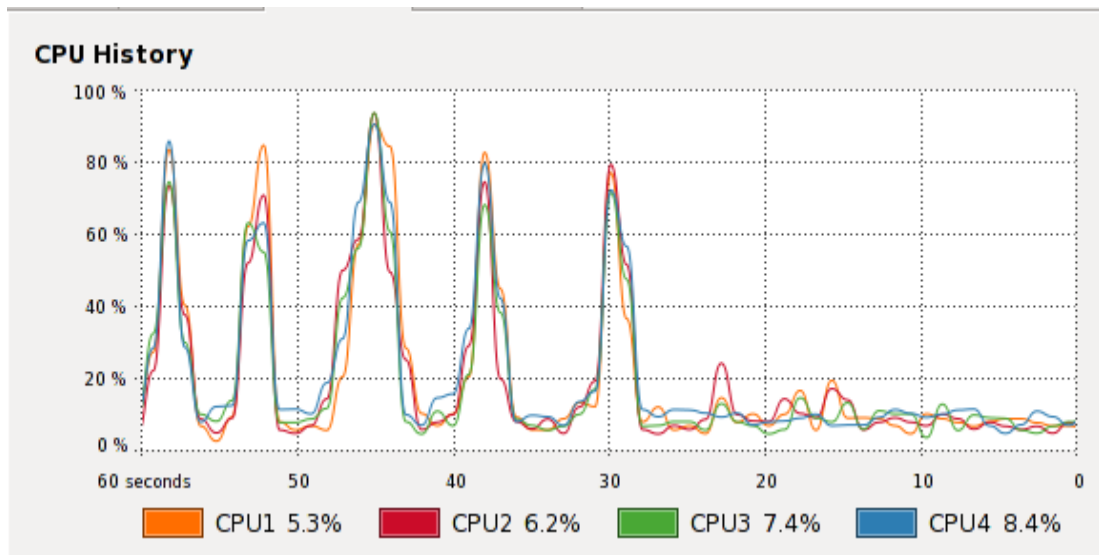


Figure 5.11: CPU Load after Cluster Setup

After completion of the cluster setup, the CPU load is returned back to a stable value. We can notice that the post cluster setup CPU load now is around 8% -10%, which is a slight increase than idling CPU load of 5%.

Low Network Overhead:

Our research work demonstrates a very low network usage clearly with the below statistics shown in the diagrams.

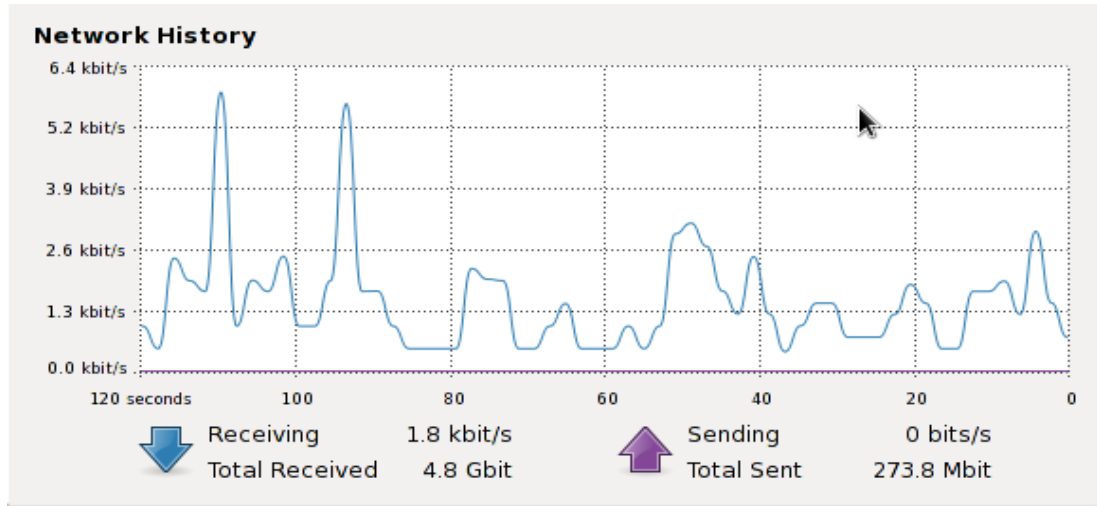


Figure 5.12: Network Utilization on System Idling

Idling network usage has several random spikes due to the background network access of the applications. However no major network utilization was noticed in the above diagram.

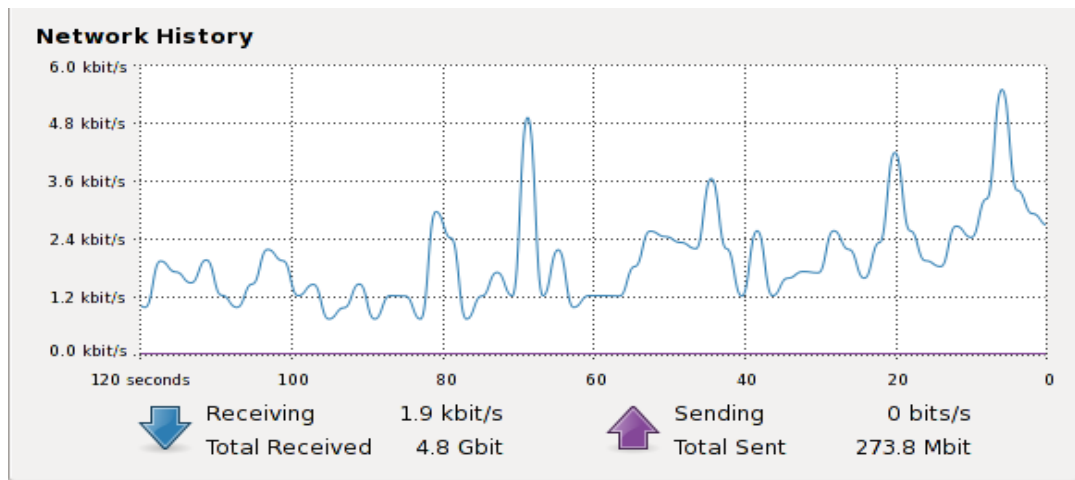


Figure 5.13: Network Utilization on Cluster Setup

However, when the cluster setup is started, ordered set of spikes started analogs to the server start client connections.

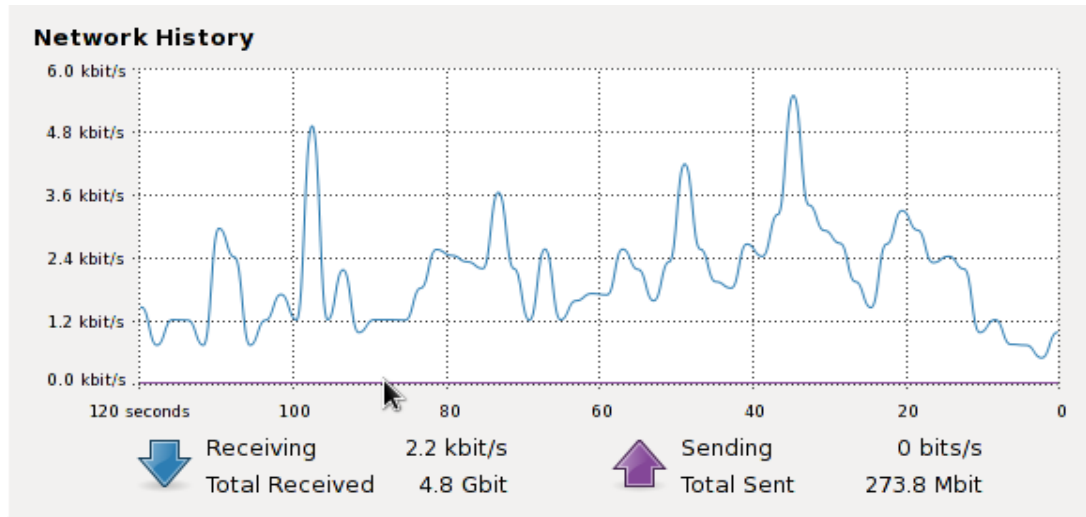


Figure 5.14: Network Utilization after Cluster Setup

When the cluster setup is completed, network usage is returned to usual value as no network interaction happens.

The usage of Web Sockets protocol heavily contributes to the lightweight -ness of the application which is demonstrated using the above figures.

5.3.3 Scalability & Stability

Initially the cluster is started with the default configuration mentioned in the section 5.1. Then the node count in the cluster is increased to 12 nodes. Cache cluster framework was capable enough to support the expansion without any issue as per the below log indications (log is configured to contain only the minimum required data set avoiding the detailed and non-related logs).

```

21:48:29.795 c.s.m.r.l.w.s.WebSocketServer Property file loaded successfully.
21:48:31.107 c.s.m.r.l.w.s.WebSocketServer [ Node Count : 1] [ Active Node List : [b52dce17]]
21:48:37.211 c.s.m.r.l.w.s.WebSocketFrameHandler [9595c2d3 {127.0.0.1:51208} ] is ONLINE.
21:48:37.215 c.s.m.r.l.w.s.WebSocketFrameHandler [ Node Count : 2] [ Active Node List : [b52dce17, 9595c2d3]]
21:48:40.751 c.s.m.r.l.w.s.WebSocketFrameHandler [845e11c4 {127.0.0.1:51231} ] is ONLINE.
21:48:40.751 c.s.m.r.l.w.s.WebSocketFrameHandler [ Node Count : 3] [ Active Node List : [845e11c4, b52dce17,
21:48:44.802 c.s.m.r.l.w.s.WebSocketFrameHandler [cc85ce28 {127.0.0.1:51269} ] is ONLINE.
21:48:44.803 c.s.m.r.l.w.s.WebSocketFrameHandler [ Node Count : 4] [ Active Node List : [845e11c4, b52dce17,
21:48:46.011 c.s.m.r.l.w.s.WebSocketFrameHandler [ebf21536 {127.0.0.1:51271} ] is ONLINE.
21:48:46.012 c.s.m.r.l.w.s.WebSocketFrameHandler [ Node Count : 5] [ Active Node List : [845e11c4, b52dce17,
21:48:47.849 c.s.m.r.l.w.s.WebSocketFrameHandler [e4cf07c9 {127.0.0.1:51292} ] is ONLINE.
21:48:47.850 c.s.m.r.l.w.s.WebSocketFrameHandler [ Node Count : 6] [ Active Node List : [845e11c4, e4cf07c9,
21:48:55.500 c.s.m.r.l.w.s.WebSocketFrameHandler [4eaf37a0 {127.0.0.1:51316} ] is ONLINE.
21:48:55.500 c.s.m.r.l.w.s.WebSocketFrameHandler [ Node Count : 7] [ Active Node List : [845e11c4, e4cf07c9,
21:48:57.907 c.s.m.r.l.w.s.WebSocketFrameHandler [8f318708 {127.0.0.1:51336} ] is ONLINE.
21:48:57.907 c.s.m.r.l.w.s.WebSocketFrameHandler [ Node Count : 8] [ Active Node List : [845e11c4, e4cf07c9,
21:48:59.712 c.s.m.r.l.w.s.WebSocketFrameHandler [32bb937a {127.0.0.1:51356} ] is ONLINE.
21:48:59.713 c.s.m.r.l.w.s.WebSocketFrameHandler [ Node Count : 9] [ Active Node List : [845e11c4, e4cf07c9,
21:49:06.165 c.s.m.r.l.w.s.WebSocketFrameHandler [60e8626c {127.0.0.1:51378} ] is ONLINE.
21:49:06.165 c.s.m.r.l.w.s.WebSocketFrameHandler [ Node Count : 10] [ Active Node List : [845e11c4, e4cf07c9,
21:49:08.590 c.s.m.r.l.w.s.WebSocketFrameHandler [f5697062 {127.0.0.1:51399} ] is ONLINE.
21:49:08.591 c.s.m.r.l.w.s.WebSocketFrameHandler [ Node Count : 11] [ Active Node List : [845e11c4, e4cf07c9,
21:49:10.442 c.s.m.r.l.w.s.WebSocketFrameHandler [b1cecc25 {127.0.0.1:51419} ] is ONLINE.
21:49:10.442 c.s.m.r.l.w.s.WebSocketFrameHandler [ Node Count : 12] [ Active Node List : [845e11c4, 8f318708,

```

Figure 5.15: Scaling the Cluster to 12 Nodes

Then then scaling is taken to another step by extending the cluster framework to 24 nodes, adding one node at a time as before.

```

21:49:06.165 c.s.m.r.l.w.s.WebSocketFrameHandler [ Node Count : 10] [ Active Node List : [845e11c4, e4cf07c9,
21:49:08.590 c.s.m.r.l.w.s.WebSocketFrameHandler [f5697062 {127.0.0.1:51399} ] is ONLINE.
21:49:08.591 c.s.m.r.l.w.s.WebSocketFrameHandler [ Node Count : 11] [ Active Node List : [845e11c4, e4cf07c9,
21:49:10.442 c.s.m.r.l.w.s.WebSocketFrameHandler [b1cecc25 {127.0.0.1:51419} ] is ONLINE.
21:49:10.442 c.s.m.r.l.w.s.WebSocketFrameHandler [ Node Count : 12] [ Active Node List : [845e11c4, 8f318708,
21:59:16.377 c.s.m.r.l.w.s.WebSocketFrameHandler [475f55be {127.0.0.1:51499} ] is ONLINE.
21:59:16.378 c.s.m.r.l.w.s.WebSocketFrameHandler [ Node Count : 13] [ Active Node List : [845e11c4, 8f318708,
21:59:18.410 c.s.m.r.l.w.s.WebSocketFrameHandler [21d60497 {127.0.0.1:51520} ] is ONLINE.
21:59:18.410 c.s.m.r.l.w.s.WebSocketFrameHandler [ Node Count : 14] [ Active Node List : [845e11c4, 8f318708,
21:59:20.832 c.s.m.r.l.w.s.WebSocketFrameHandler [1be6296e {127.0.0.1:51559} ] is ONLINE.
21:59:20.833 c.s.m.r.l.w.s.WebSocketFrameHandler [ Node Count : 15] [ Active Node List : [845e11c4, 8f318708,
21:59:21.817 c.s.m.r.l.w.s.WebSocketFrameHandler [47c46167 {127.0.0.1:51560} ] is ONLINE.
21:59:21.818 c.s.m.r.l.w.s.WebSocketFrameHandler [ Node Count : 16] [ Active Node List : [845e11c4, 8f318708,
21:59:23.778 c.s.m.r.l.w.s.WebSocketFrameHandler [8a083e1b {127.0.0.1:51580} ] is ONLINE.
21:59:23.778 c.s.m.r.l.w.s.WebSocketFrameHandler [ Node Count : 17] [ Active Node List : [845e11c4, 8a083e1b,
21:59:25.813 c.s.m.r.l.w.s.WebSocketFrameHandler [5c1d2da5 {127.0.0.1:51600} ] is ONLINE.
21:59:25.814 c.s.m.r.l.w.s.WebSocketFrameHandler [ Node Count : 18] [ Active Node List : [845e11c4, 8a083e1b,
21:59:27.744 c.s.m.r.l.w.s.WebSocketFrameHandler [017754f5 {127.0.0.1:51620} ] is ONLINE.
21:59:27.744 c.s.m.r.l.w.s.WebSocketFrameHandler [ Node Count : 19] [ Active Node List : [845e11c4, 8a083e1b,
21:59:29.468 c.s.m.r.l.w.s.WebSocketFrameHandler [86142948 {127.0.0.1:51640} ] is ONLINE.
21:59:29.469 c.s.m.r.l.w.s.WebSocketFrameHandler [ Node Count : 20] [ Active Node List : [845e11c4, 8a083e1b,
21:59:31.230 c.s.m.r.l.w.s.WebSocketFrameHandler [dcb759ee {127.0.0.1:51660} ] is ONLINE.
21:59:31.231 c.s.m.r.l.w.s.WebSocketFrameHandler [ Node Count : 21] [ Active Node List : [845e11c4, 8a083e1b,
21:59:37.806 c.s.m.r.l.w.s.WebSocketFrameHandler [356cb68a {127.0.0.1:51680} ] is ONLINE.
21:59:37.807 c.s.m.r.l.w.s.WebSocketFrameHandler [ Node Count : 22] [ Active Node List : [845e11c4, 8a083e1b,
21:59:39.653 c.s.m.r.l.w.s.WebSocketFrameHandler [32bae5f9 {127.0.0.1:51700} ] is ONLINE.
21:59:39.653 c.s.m.r.l.w.s.WebSocketFrameHandler [ Node Count : 23] [ Active Node List : [845e11c4, 8a083e1b,
21:59:41.550 c.s.m.r.l.w.s.WebSocketFrameHandler [9deba5d0 {127.0.0.1:51720} ] is ONLINE.
21:59:41.551 c.s.m.r.l.w.s.WebSocketFrameHandler [ Node Count : 24] [ Active Node List : [845e11c4, 8a083e1b,

```

Figure 5.16: Extending to 24 Nodes

No unusual behaviors or concerns were noticed while executing this experiment as well. As a result, we can prove that our cluster framework can be scaled smoothly without concerns and after the cluster setup is completed, the system is stable and performing well to the client request as expected.

5.3.4 Fault Tolerance

To demonstrate the fault tolerance behavior, we have used the below methodology. In the initial step, we have added nodes to the cluster framework and let it stable with 18 nodes.

```
FrameHandler [ Node Count : 17] [ Active Node List : [NodeData{remoteAddress='127.0.0.1:37474'}
FrameHandler [ Node Count : 17] [ Active Node List : [571502f0, 993b1b9b, a6f9ce94, 25c10411,
FrameHandler [ Node Count : 18] [ Global Node List : [571502f0, 993b1b9b, a6f9ce94, 25c10411,
```

Figure 5.17: Stable Cluster with 18 nodes

In the next step, three nodes are forcefully removed from the cluster one by one. However as expected, our cluster framework could handle the node failures as expected and immediately update the Node Registry accordingly. Due to that, the client application using the framework can achieve fault tolerance by handling the user requests with the online nodes filtering out the failed node without any failures on the application.

```
FrameHandler [25c10411 {127.0.0.1:37449} ] is INACTIVE.
FrameHandler [ Node Count : 16] [ Active Node List : [NodeData{remoteAddress='127.0.0.1:37474'}
FrameHandler [ Node Count : 16] [ Active Node List : [571502f0, 993b1b9b, a6f9ce94, c576286a,
FrameHandler [25c10411 {127.0.0.1:37449} ] is OFFLINE.
FrameHandler [ Node Count : 18] [ Global Node List : [571502f0, 993b1b9b, a6f9ce94, 25c10411,
FrameHandler [2e166c1a {127.0.0.1:37453} ] is INACTIVE.
FrameHandler [ Node Count : 15] [ Active Node List : [NodeData{remoteAddress='127.0.0.1:37474'}
FrameHandler [ Node Count : 15] [ Active Node List : [571502f0, 993b1b9b, a6f9ce94, c576286a,
FrameHandler [2e166c1a {127.0.0.1:37453} ] is OFFLINE.
FrameHandler [ Node Count : 18] [ Global Node List : [571502f0, 993b1b9b, a6f9ce94, 25c10411,

FrameHandler [3329f385 {127.0.0.1:37443} ] is INACTIVE.
FrameHandler [ Node Count : 14] [ Active Node List : [NodeData{remoteAddress='127.0.0.1:37474'}
FrameHandler [ Node Count : 14] [ Active Node List : [571502f0, 993b1b9b, a6f9ce94, c576286a,
FrameHandler [3329f385 {127.0.0.1:37443} ] is OFFLINE.
FrameHandler [ Node Count : 18] [ Global Node List : [571502f0, 993b1b9b, a6f9ce94, 25c10411,
```

Figure 5.18: Stable Cluster after Node Failures

5.4 Performance Evaluation and Analysis

We performed a simulated experimental use-case on our cluster framework and extended it to a simulated experimental evaluation, making a comparison against a reference framework which used TCP protocol for node to node communication (clustering).

5.4.1 Application Use case

In our application we used an incremental approach to assess our application. It is started with a baseline measurement having matrices such as CPU load, Memory Usage and Network Utilization.

The environment used was the exact runtime environment defined in section 5.1 with default Linux resource monitoring application. The monitoring window was 30 seconds on each matrix. The matrices used in the simulated experiment are as follows,

- **Node Count** – Number of nodes in the stable cluster
- **CPU Load** – CPU utilization as a percentage (%)
- **Memory Usage** – Memory usage in Gigabytes (GB)
- **Network Utilization** – Network Utilization in bits per second (bits/s)

Methodology

As mentioned above, an experimental use-case was used with incremental approach. Starting from a baseline with no application running, in each step, the number of nodes is increased one by one in a set of 06 nodes.

This experiment is carried out to a maximum of 42 nodes, measuring the above mentioned other metrics and recording them. Then those measurements are analyzed to identify the application behavior on cluster scaling by node addition. Following are the steps of the application experiment.

Step 0

Measure the baseline CPU load, Memory Usage and the Network Utilization on stable system idling.

- **CPU Load**

The average CPU Load (as system is with 4 CPU cores) in system idle on baseline was around 3 – 4 % with basic system functions. And, the CPU contains recurring spikes on several intervals.

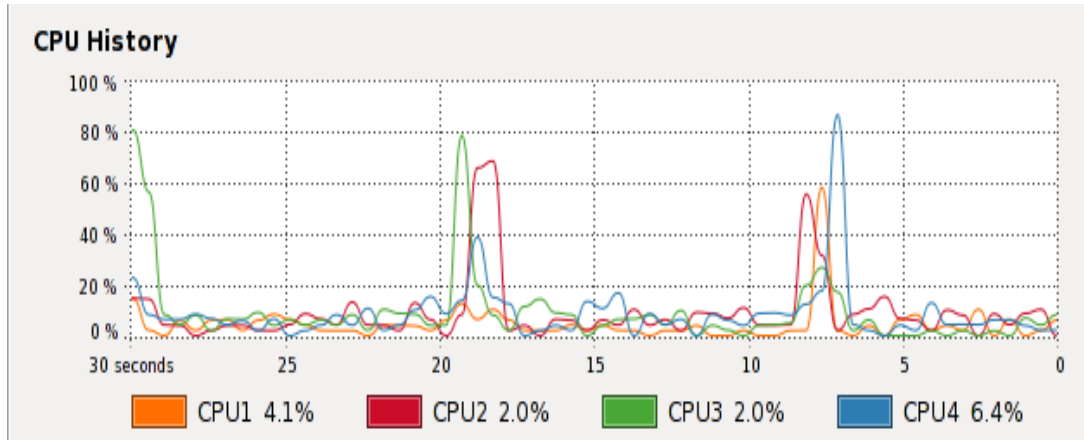


Figure 5.19: CPU Load on System Idling

- **Memory Usage**

The baseline memory usage on the system was around 2.1- 2.2 GB. The above baseline was verified by loading several applications and terminating them to make sure the system is returning to the baseline footprint eventually.

However, swapping is not used and considered on the experiment.

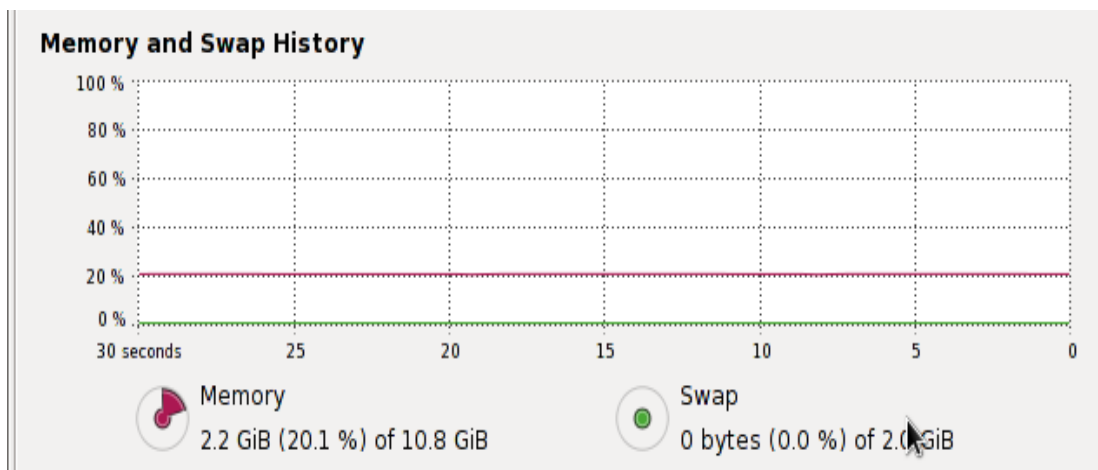


Figure 5.20: Memory Usage on System Idling

- **Network Utilization**

Network utilization was almost 0 % on the idling mode as there no significant network interacting applications running on the system. To measure the network utilization changes accurately and notice the tiny changes, bits per second (bits/s) was used as the unit.

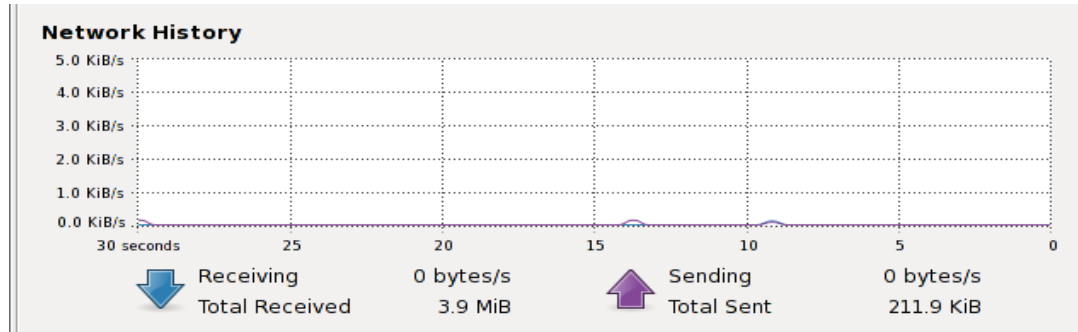


Figure 5.21: Network Utilization on System Idling

Step 1 - 7

Measure the baseline CPU load, Memory Usage and the Network Utilization on a stable system adding six nodes to the cluster in each step.

Results

- **CPU Load**

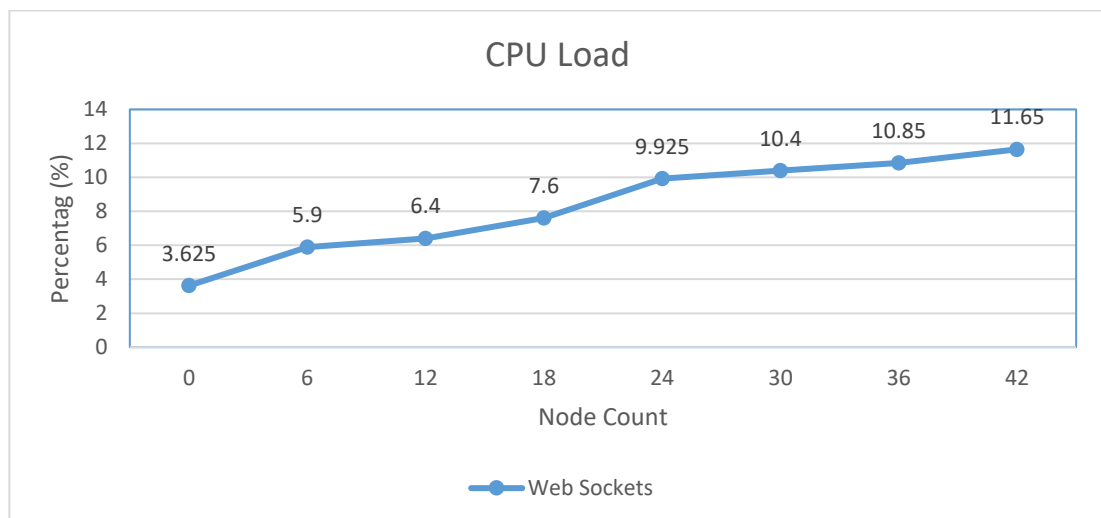


Figure 5.22: CPU Load with Node Count

Figure 5.22 shows a very significant and important characteristic of our cluster framework. When the node count is increased from 0 to 42 nodes, the CPU load is increased from 3.6 % to 11.6 % which is an 8 % increment.

However, in such system of 42 nodes, it is an outstanding character to maintain the CPU Load to a minimum. This is mainly due to the NIO behavior of the Netty platform with Web Sockets protocol usage.

Here the client connect/ disconnects are minimal due to web sockets and CPU Load is required only for message transfer related tasks. Even they are also handled smoothly due to the non-blocking I/O behavior of the framework.

- **Memory Usage**

While in the cluster node increment, the memory usage of the system is increased from 2.5 GB to 5.2 GB, which is a 2.7 GB increment. As an average each node consumes 0.06 GB memory to maintain the cluster membership between nodes.

This also seems a fine for a cluster with 42 node members. However, this is due to the connected and duplex behavior of the web sockets protocol in the cluster.

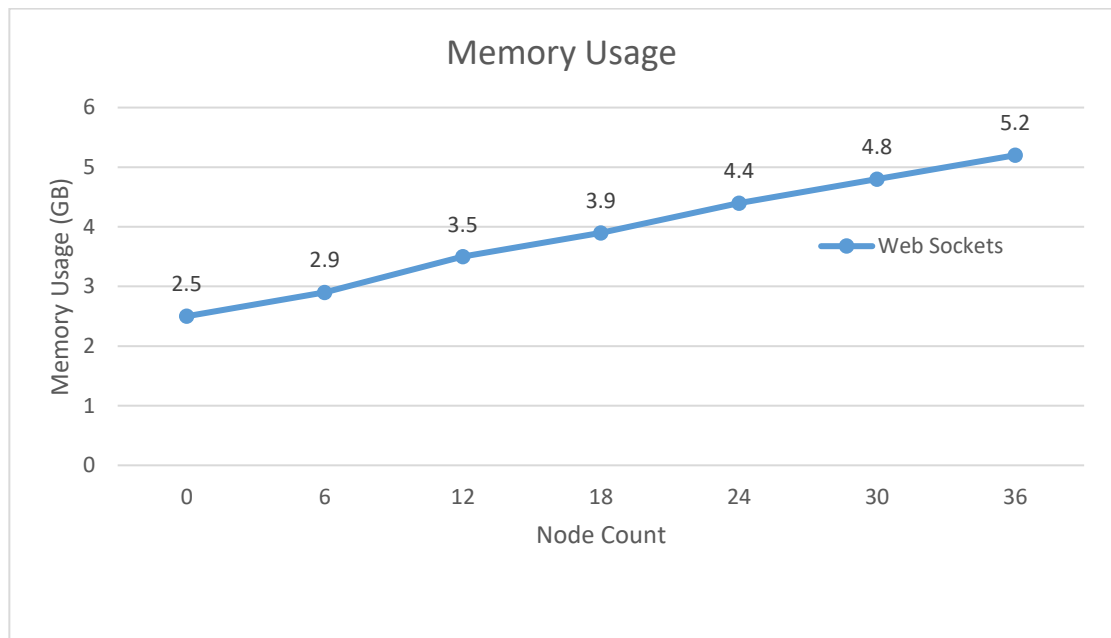


Figure 5.23: Memory Usage with Node Count

- **Network Utilization**

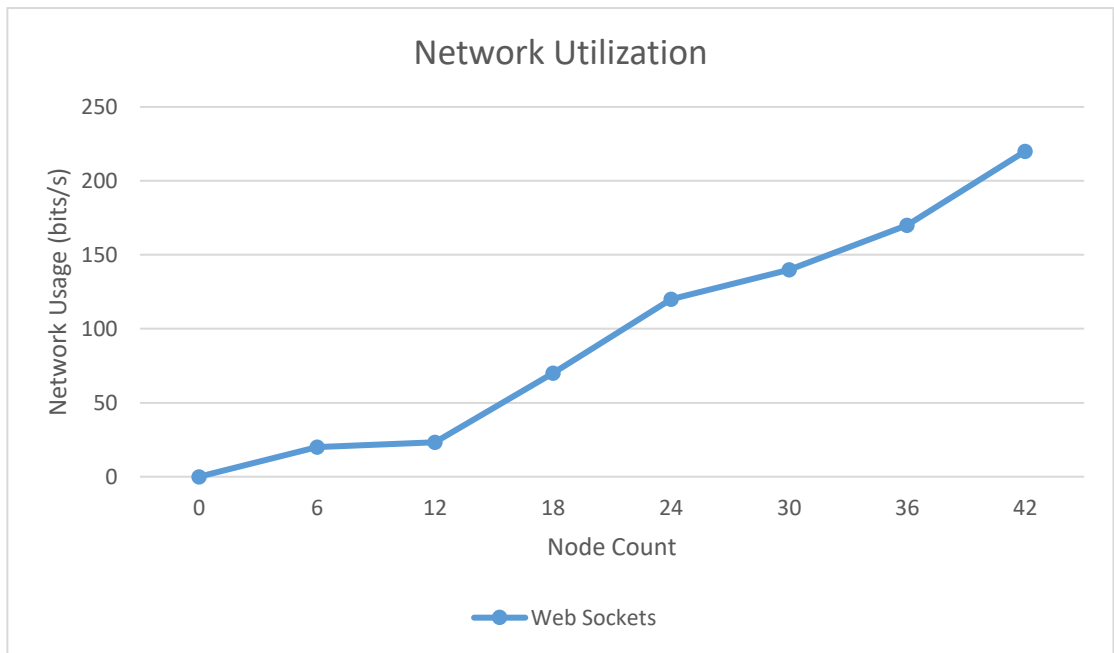


Figure 5.24: Network Utilization with Node Count

We noticed a very small network utilization out framework. This is mainly due to the characteristics of the Web Sockets protocol. As the nature of web sockets, its remains connected and reduces the header repetition overheads in a polling like system. Here, the network bandwidth is used only for the data transfers among nodes which is used for node sync and cache sync messages.

Summary

In the above use case, we have noticed that our cluster framework is capable to be scaled seamlessly well with minimum CPU and network overhead to the underlying system. However, the memory usage has a tradeoff by nature of the protocol used, but it is also within the acceptable limits for a cluster framework of such a higher number of nodes.

5.4.2 Experimental Evaluation

In the next phase, we extended our use case in the section 5.4.1 of the cluster framework to a simulated experimental evaluation with a comparison to a TCP polling-based reference framework. To make the two-system equal in core concepts and components, we developed a reference TCP polling application on top of the same Netty framework TCP components.

The environment used in this simulated experiment was exactly same environment defined in the section 5.1. Also, the same baseline was aligned to carry out the experiment with reference framework.

The polling configuration was set to have following configurations,

- Reconnect Delay – 0 seconds
- Read Timeout – 1 second

CPU Load

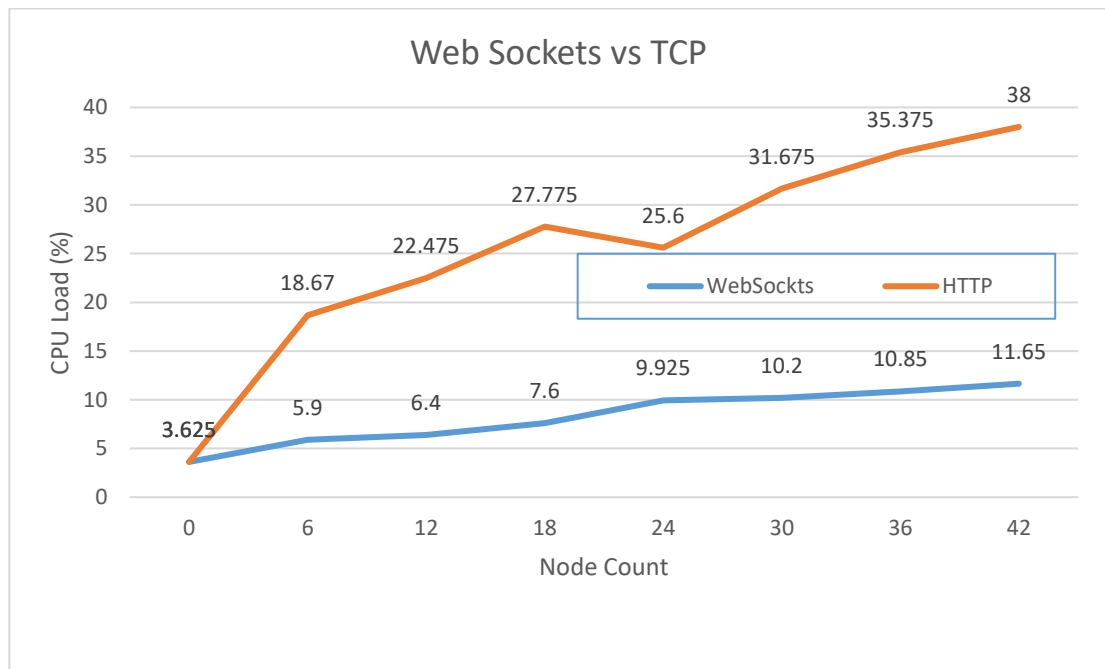


Figure 5.25: CPU Load, Web Sockets vs. TCP

When evaluating our work with the reference framework, it can be clearly noticed the performance advantages of our cluster framework using web sockets. In the reference framework, the CPU load is increased to an unexpected level.

In contrast our work can manage the cluster with minimal CPU load. In a TCP based framework, the CPU needs to perform considerable amount of work on client connection/ disconnections, so to cluster management. As Web Sockets maintain connected nodes, this overhead is minimized.

Memory Usage

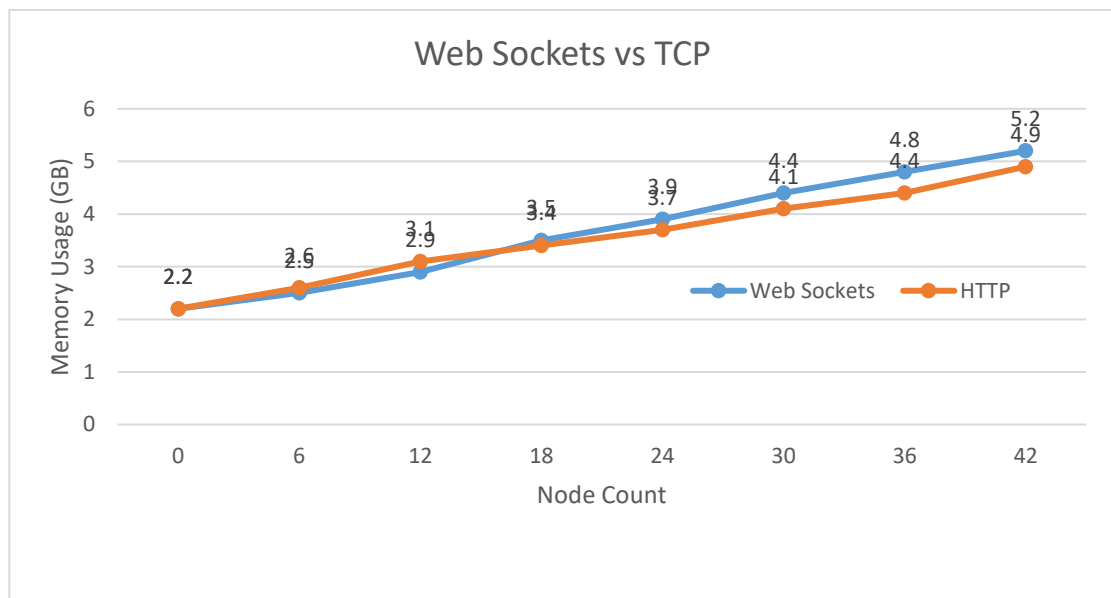


Figure 5.26: Memory Usage, Web Sockets vs. TCP

Our work clearly aligned with the expectations of the protocols used. In TCP, the connections are not maintained for a long period, they are connected and disconnect after a “read time out”. Hence, the memory usage on resource allocation is often released.

However, in WebSockets, as the connections are kept alive, the memory usage on resource allocations are fixed. Even though it has fixed assignments, the memory usages are admirable with the NIO capabilities of the Netty framework.

Network Utilization

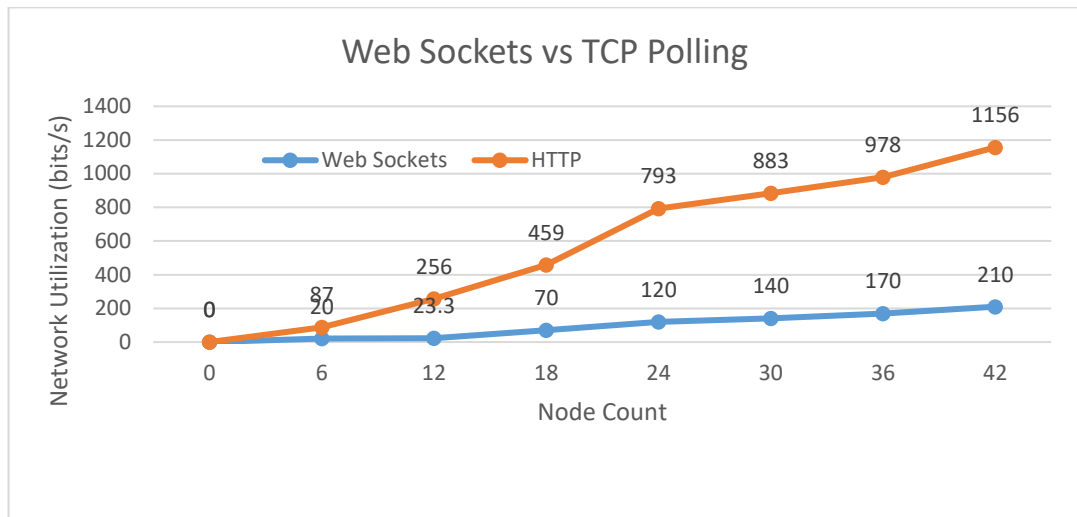


Figure 5.27: Network Utilization, Web Sockets vs. TCP Polling

Web sockets contribute heavily to maintain a very low network utilization level from its design. As the connections are maintained and kept alive, the network overhead is very low than in TCP. In TCP, the headers and the connection attempts are repeated frequently (by polling interval), it appears a huge network overhead with same bulky data.

Summary

In the above simulated experimental evaluation, we have clearly demonstrated that our work, cluster framework using WebSockets is outstanding beyond the reference framework with TCP in CPU load and network utilization heavily. More importantly, it is going hand in hand with the TCP framework even in its weakest point, memory usage due to its connected behavior.

CHAPTER 6

6 CONCLUSION & FUTURE WORK

In this project we used an efficient and sophisticated communication model for the network communication, WebSockets. Once the cluster is setup, the connection is continued to make the data transfers while cluster status directly depends on the current connectivity. This makes the network utilization more efficient on node management, saving more bandwidth for the application level service traffic.

Fault tolerance usually comes in to the play after node failure happens. For an example, if a node left from the cluster due to the failures, it is always required to provide fault tolerance mechanism for the efficiency and the availability of the distributed application. Here, handling a failed node is simplified by ignoring the legacy data the node contained when it was live.

As a limitation, manual addition and removal of the live nodes (existing server node without relevant configuration setup) is not supported, and it is out of the scope of this project. Also, the presented cluster framework does not support node authentication. It means an authentication mechanism is not implemented to validate for allowed nodes to be connected. It is a security threat to the system as any node can be configured to join and work for the cluster without any verification. Basically cluster needs to have a `cluster.secret.key/ cluster.secret.password` as configurations to set in node configuration and they needs to be validated while joining a node to the cluster. This can be considered as a future work of the cache cluster framework project.

Caching Framework application has been heavily benefiting from the Web Sockets protocol to achieve the above lightweight-ness on both system load and the network as expected. That is because of the main feature of Web sockets to remain connected avoiding resource/ network consuming header info and handshake repetitions which are a common behavior in Polling, P2P or gossip protocols.

We believe that this effort would be contributing to develop distributed applications adding efficient framework for cluster-awareness as a building block. Also, the application will be able to benefit from the replicated-ness of the cache and the configurations implemented in the framework, which making a significant value addition to the system.

REFERENCES

- [1] Fekete, A. D., & Ramamritham, K. (2010). Consistency models for replicated data. In *Replication* (pp. 1-17). Springer, Berlin, Heidelberg.
- [2] Factor, M., Schuster, A., & Shagin, K. (2004, April). A distributed runtime for Java: yesterday and today. In *Parallel and Distributed Processing Symposium, 2004. Proceedings. 18th International* (p. 159). IEEE.
- [3] Estublier, J. (2000, May). Software configuration management: a roadmap. In *Proceedings of the Conference on the Future of Software Engineering* (pp. 279-289). ACM.
- [4] Randell, B. (1975). System structure for software fault tolerance. *IEEE Transactions on Software Engineering*, (2), 220-232.
- [5] Hayashibara, N., Cherif, A., & Katayama, T. (2002). Failure detectors for large-scale distributed systems. In *Reliable Distributed Systems, 2002. Proceedings. 21st IEEE Symposium on* (pp. 404-409). IEEE.
- [6] Wolfe, A. (1993, September). Software-based cache partitioning for real-time applications. In *Third International Workshop on Responsive Computer Systems*.
- [7] Terry, D. B. (1987). Caching hints in distributed systems. *IEEE Transactions on Software Engineering*, (1), 48-54.
- [8] Bennett, J. K., Carter, J. B., & Zwaenepoel, W. (1990). Adaptive software cache management for distributed shared memory architectures (Vol. 18, No. 2SI, pp. 125-134). ACM.
- [9] Voulgaris, S., Gavidia, D., & Van Steen, M. (2005). Cyclon: Inexpensive membership management for unstructured p2p overlays. *Journal of Network and Systems Management*, 13(2), 197-217.
- [10] Ferreira JF, Sobral JL, Proença AJ. JaSkel: A Java skeleton-based framework for structured cluster and grid computing. In *Cluster Computing and the Grid, 2006. CCGRID 06. Sixth IEEE International Symposium on 2006 May 16* (Vol. 1, pp. 4-pp). IEEE.
- [11] Berman F, Fox G, Hey AJ, editors. *Grid computing: making the global infrastructure a reality*. John Wiley and sons; 2003.

- [12] Verma A, Pedrosa L, Korupolu M, Oppenheimer D, Tune E, Wilkes J. Large-scale cluster management at Google with Borg. In Proceedings of the Tenth European Conference on Computer Systems 2015 Apr 17 (p. 18). ACM.
- [13] Robbert Van Renesse, Yaron Minsky, and Mark Hayden. A gossip-style failure detection service. In Service, T Proc. Conf. Middleware, pages 55-70, 1996.
- [14] P. Eugster, S. Handurukande, R. Guerraoui, A.-M. Kermarrec, and P. Kouznetsov, “Lightweight probabilistic broadcast”, Proc. Int. Conf. Dependable Systems and Networks (DSN 2001), 2001
- [15] A. Lakshman and P. Malik. Cassandra: A decentralized structured storage system. ACM SIGOPS Operating Systems Review, 44(2):35–40, 2010.
- [16] Giuseppe de Candia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: amazon~Os highly available key-value store. In Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles, pages 205{220. ACM}, 2007.
- [17] P. Eugster, S. Handurukande, R. Guerraoui, A.-M. Kermarrec, and P. Kouznetsov, “Lightweight probabilistic broadcast”, Proc. Int. Conf. Dependable Systems and Networks (DSN 2001), 2001
- [18] Kenneth P. Birman, Mark Hayden, Ozgur Ozkasap , Zhen Xiao , Mihai Budiu , Yaron Minsky, Bimodal multicast, ACM Transactions on Computer Systems (TOCS), v.17 n.2, p.41-88, May 1999
- [19] M.-J. Lin and K. Marzullo. Directional gossip: Gossip in a wide area network. Technical Report CS1999-0622, University of California, San Diego, Computer Science and Engineering, June 1999.
- [20] J. Leitaó, J. Pereira, and L. Rodrigues. HyParView: A membership protocol for reliable gossip-based broadcast. In DSN '07: Proc. of the 37th Annual IEEE/IFIP Intl. Conf. on Dependable Systems and Networks, pages 419-429, Edinburgh, UK, 2007. IEEE Computer Society.
- [21] Robbert van Renesse, Dan Mihai Dumitriu, Valient Gough, and Chris Thomas. Efficient reconciliation and flow control for anti-entropy protocols. In Proceedings of the 2nd Large Scale Distributed Systems and Middleware Workshop (LADIS '08), New York, NY, USA, 2008. ACM.

- [22] Giuseppe de Candia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Alex Pilchin, Swaminathan Sivasubramanian, Peter Voshall, and Werner Vogels. Dynamo: amazon-Os highly available key-value store. In Proceedings of twenty-rst ACM SIGOPS symposium on Operating systems principles, pages 205{220. ACM, 2007.
- [23] Taylor, K. and Golding, R. Group Membership in the Epidemic Style. Dept. of Computer Science Rep. UCSC-CRL-92-1. University of California at Santa Cruz, 1992.
- [24] Alan Demers, Dan Greene, Carl Hauser, Wes Irish, John Larson, Scott Shenker, Howard Sturgis, Dan Swinehart, and Doug Terry. Epidemic algorithms for replicated database maintenance. *Operating Systems Review*, 22(1):8–32 (January 1988).
- [25] Alan Demers, Mark Gealy, Dan Greene, Carl Hauser, Wes Irish, John Larson, Sue Manning, Scott Shenker, Howard Sturgis, Dan Swinehart, Doug Terry, and Don Woods. Epidemic algorithms for replicated database maintenance. Technical report CSL–89–1 (January 1989). Xerox Palo Alto Research Center, CA.
- [26] Jelasity M, Babaoglu O. T-Man: Gossip-based overlay topology management. *Engineering Self-Organising Systems*. 2005 Jul 25; 3910:1-5.
- [27] Voulgaris, S., van Steen, M., & Iwanicki, K. (2007). Proactive gossip-based management of semantic overlay networks. *Concurrency and Computation: Practice and Experience*, 19(17), 2299-2311.
- [28] Karakaya M, Korpeoglu I, Ulusoy Ö. Free riding in peer-to-peer networks. *IEEE Internet computing*. 2009 Mar;13(2):92-8.
- [29] Xavier D´efago, P´eter Urb´an, Naohiro Hayashibara, and Takuya Katayama. The ϕ accrual failure detector. In RR IS-RR-2004-010, Japan Advanced Institute of Science and Technology, pages 66–78, 2004.
- [30] Lin S, Taiani F, Blair GS. Gossipkit: A framework of gossip protocol family, 2007.
- [31] I. Fette and A. Melnikov, “The WebSocket Protocol,” 2011.
- [32] Hanson J. What is HTTP Long Polling. Pubhub. December. 2014.
- [33] Pimentel, V., & Nickerson, B. G., Communicating and displaying real-time data with WebSocket. *IEEE Internet Computing*, 16(4), 45-53, 2016.

- [34] Lubbers P. Html5 web sockets: A quantum leap in scalability for the web. <http://www.websocket.org/quantum.html>. 2011.
- [35] JBoss.Netty project. [2012-4-1]. <http://netty.io>
- [36] Apache MINA. Apache MINA Project. 2012-07-07. <http://mina.apache.org>. 2009.
- [37] Bortvedt J. JCache-Java Temporary Caching API. Java Specification Request. 2001 Mar; 107:19.
- [38] Infinispan J. Infinispan Cache Mode.
- [39] I. Hickson, "The WebSocket API," W3C candidate recommendation, Dec. 2011; www.w3.org/TR/websockets.
- [40] Writing WebSocket servers, https://developer.mozilla.org/en-US/docs/Web/API/WebSockets_API/Writing_WebSocket_servers
- [41] Johns, M. (2013). Getting Started with Hazelcast. Packt Publishing Ltd.
- [42] Wind, D. (2013). Instant effective caching with ehcache. Packt Publishing Ltd.
- [43] Yang, J., Zhang, H., Han, L., Cui, B., & Dong, G., Design and implementation of software consistency detection system based on Netty framework. In International Conference on Broadband and Wireless Computing, Communication and Applications (pp. 343-351). Springer International Publishing, November 2016.
- [44] Montresor, A. (1999). The Jgroup distributed object model. In Distributed Applications and Interoperable Systems II (pp. 389-402). Springer, Boston, MA.
- [45] Birman, K. P. (1993). The process group approach to reliable distributed computing. Communications of the ACM, 36(12), 37-53.
- [46] Object management group. The Common Object Request Broker: Architecture and Specification, Rev. 2.2. OMG Inc., Framingham, Mass., March 1998.
- [47] Wollrath, A., Riggs, R., & Waldo, J. (1996). A Distributed Object Model for the Java[^] T[^] M System. Computing Systems, 9, 265-290.
- [48] Sun microsystems. Java remote method invocation specification, rev. 1.42. Sunmicrosystems, inc., mountain view, california, october 1997.
- [49] Kasera, S. K., Kurose, J., & Towsley, D. (1997). Scalable reliable multicast using multiple multicast groups. ACM Sigmetrics performance evaluation review, 25(1), 64-74.