2011-01-01

Decentralized Fault Tolerant Caching With Memcached

Bivas Das
University of Texas at El Paso, bdas@miners.utep.edu

DECENTRALIZED FAULT TOLERANT CACHING

WITH MEMCACHED

BIVAS DAS

Department of Computer Science

APPROVED:

Eric A. Freudenthal, Ph.D.

Luc Longpré, Ph.D.

Virgilio Gonzalez, Ph.D.

Patricia D. Witherspoon, Ph.D.
Dean of the Graduate School

to

MOTHER, FATHER

and

SHUBHRA

...

with love

DECENTRALIZED FAULT TOLERANT CACHING WITH MEMCACHED

by

BIVAS DAS, B.Tech.

THESIS

Presented to the Faculty of the Graduate School of

The University of Texas at El Paso

in Partial Fulfillment

of the Requirements

for the Degree of

MASTER OF SCIENCE

Department of Computer Science

THE UNIVERSITY OF TEXAS AT EL PASO

May 2011

Acknowledgement

First and foremost I offer my sincere gratitude to my supervisor, Dr Eric Freudenthal, who has supported me throughout my thesis with his patience while allowing me the room to work in my own way. I am greatly indebted to him for my Masters degree without whose encouragement and effort this theses would be incomplete.

I also thank the members of my graduate committee for their guidance and suggestions. Special thanks go to Dr. Luc Longpré, without his assistance this study would not have been successful. Also, thanks to Dr. Virgilio Gonzalez for helping me with this theses.

I am heartily thankful to my dear friends Manali Chakraborty and Amritam Sarcar. Without their persuasion and guidance, I would not have come to study abroad in the first place. They helped me in countless ways everytime I needed them. I greatly appreciate my good friends, Avranil Tah for all of his help and friendly support when I needed it the most.

I also want to thank my fiancé Shubhra Datta for her constant efforts and never ending patience for inspiring me to work harder.

Last but not the least, I thank my parents for supporting me throughout all my studies at UTEP, without their motivation and thoughtful decisions, I would not have been even what I am now.

And finally, I offer my thanks and regards to all of those who supported me in any respect during the completion of the thesis.

Bivas Das

NOTE: This thesis was submitted to my Supervising Committee on May 11, 2011.

Abstract

Recent changes in web trends not only have increased popularity of web services, but also have vastly increased usage. Popular techniques, such as web-caching, used by content and service providers to provide faster content delivery at user end, is not enough to keep up, due to diversity and dynamic nature of web-contents.

Emerging ideas to cache on the server side, for faster content generation and delivery, is currently in use by popular web-service providers. One example of such caching system is Memcached. However, memcached, a distributed high performance caching system, is ineffective in dynamic organization of itself and scaling when required, such as failure or adding more systems on demand.

In this theses I have designed and implemented a group membership protocol within a set of Memcached servers, used as a caching pool, that can make the pool aware of its participants. The protocol also helps to dynamically resize itself in case of change of members in the pool. This way in case of both failure and adding more nodes, the pool automatically adjusts itself and publishes the information to the pool clients when adapted to the protocol. I have also shown effectiveness of the protocol by running several test cases.

Table of Contents

List of Figures

Chapter 1

Introduction

1.1 Caching and Web Services

World Wide Web has an exponential growth and increasing number of users, resulting in network congestion and server overloading. As a solution researches from early 90s suggested to cache popular objects closer to client location [44]. Such as a proxy server on corporate network (see Fig. 1.1) can be used to cache external resources. With time, this web-caching became a popular trend to serve users with faster content delivery [34].

1.1.1 Advantages and Disadvantages of Caching

Caching reduces the used bandwidth and latency by fetching a majority of the contents from local location. Therefore, the un-cached documents are fetched faster using the bandwidth gained in caching mechanism [28], [20]. Furthermore, it not only reduces the workload on the remote server, but also assures some availability when remote server is not available. Also, the service providers can organize the contents and their availability by studying the usage pattern of clients.

Cache miss can significantly increase access latency based the availability and locality of the contents. A naive cache-update procedure may present the client with stale and irrelevant data. A single proxy-cache-host can be a bottleneck, may degrade performance, and can be single point of failure. Also proxy-cache must be set to maximum allowed to clients for stable operation. Furthermore, content providers' fail to find true access of contents, as caching reduces access to their (remote) server, consequently failing to rank

them based on access rate.

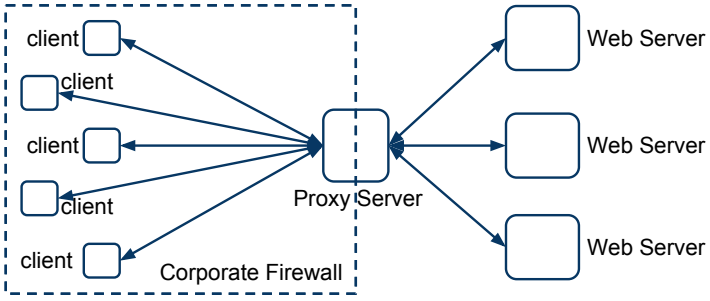

Figure 1.1: Proxy Caching in Corporate (Firewalled) Network

1.1.2 Evolution of Caching

Cache architecture (see Fig, 1.2), can be hierarchical [15] (contents are cached at multiple levels and queried in order until it is found) or can be distributed [39] (contents are placed at the same level and is shared without replication). It might also be a hybrid system between these two [18], optimizing replication and access latency. Cache usability determines which type of content will be cached by the system. Examples of cached contents are document, image, streaming content, results of computation, etc.

In prolonged operation, better availability can be assured by co-operation among caches, where neighbor responds to cache misses with appropriate content if available. Cache resolution and routing of the data efficiently assures smooth operation. Also, studying usage pattern and using heuristics to prefetch contents reduces latency significantly [38], [29], [19]. Caches placement and management also influences throughput and access latency [10].

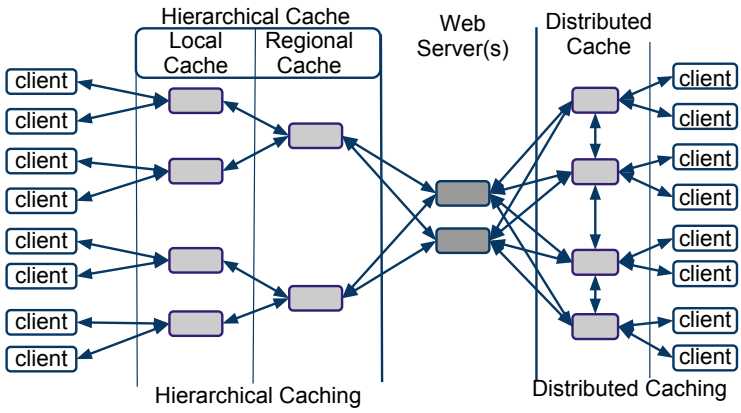

Figure 1.2: Hierarchical Caching and Distributed Caching

1.2 Emergence of WEB 2.0 as a platform

In the early days of Internet, sometimes referred as WEB 1.0, the contents were mostly static and organized in a distinctive bow-tie [12] or hierarchical structure. In this design, the site's main page showed summaries and guides to respective contents, sub-contents and more using hyperlinks. Furthermore, these sites contents were not user centric. Also, in WEB 1.0, the contents were mostly static with texts and images, thus this worked like producer/consumer model [16], with few producer creating contents and many consumer viewing/using those contents simultaneously.

While there was a notion of user centric web, as seen in online shopping services providers, they were still classified as Web 1.0. Amazon, launched in mid 90s evolved steadily as added new features for a refined online-shopping experience. With emerging popular sites like MySpace, Facebook, Twitter, LiveJournal, Youtube, LinkedIn, Digg, SlashDot, eBay and many more, the model started changing over time. However, it was not until in 2005 a definition, by O'Really [37], emphasized WEB 2.0 as a platform. One of the most important features that standardize a site as WEB 2.0 is its user centric design, where users are focused over contents. They have personal profiles; can connect with other users and post contents, comments and updates.

This is the key factor that changed the producer consumer model of WEB 1.0 to a publisher/subscriber model in WEB 2.0, resulting in a vast increase of dynamically generated contents by the users themselves. Furthermore, this not only increased creators and contributors, but also introduced a new way of generating content. Such as gathering contents from various connected users and culturing through the users' interaction with the content such as reviews and/or comments. Another interesting property in WEB 2.0 is mash-up of relevant data for users from various sources to one place to create a rich and distinct web-experience.

WEB 2.0 uses new technologies such as AJAX (Asynchronous JavaScript and XML) and embeds alien objects, such as applets and flash videos, through browser plugins, proving robust web-experience. Furthermore, WEB 2.0 optionally (e.g. Facebook) provides API's for developers to create Web based apps for users, improving user experience and to generating revenue for developers.

1.2.1 Impact of WEB 2.0

WEB 1.0 is simpler to design, therefore performance characteristics were easier to measure, mostly due to the static contents. However, that is not the case for WEB 2.0. While in WEB 1.0 a simple "click" fetches a page, in WEB 2.0 there are no clicks but activities referring to various tasks that can be accomplished based on the situation. Also, in addition to the static contents, users can provide rating, post comment on existing content, do casual and/or group communication and can create new content for others.

Furthermore, in WEB 2.0, the sites encourage the users to spend as much time as possible to increase the pportunity of higher advertising revenue. Thus they provide access from not only a standard web-browser but from various other platforms and mobile devices. This, accumulated with vast user base of millions, require those services to handle millions of requests with minimum failures for keeping the users interested in their service. Also, certain increased access in particular content, i.e. a video getting popularity in YouTube or a celebrity joining Twitter, creates huge demand for that particular content, known as

flash crowd, needs to be dealt with properly without any significant performance loss at the user end. Moreover, the contents generated in WEB 2.0 are so dynamic and user centric in nature, that caching them with the semantics of WEB 1.0 would be useless.

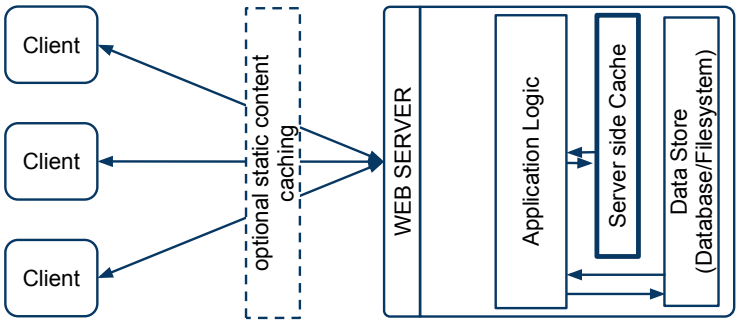

Figure 1.3: Caching on the Server Side

This introduced new strategies to improve performance, by caching popular contents in the server side (see Fig. 1.3) for quicker content delivery. In this way, the server deploys and uses its own cache at the server side, that provides faster access to contents than if it accessed from original storage. Furthermore, these caches are often used to store results of an computationally expensive calculation, for sharing and later use. The web application layer, when requested for a particular content, checks the cache first. If it is found, it is delivered reducing the response time. Otherwise it fetches/generates the content, stores it in the cache and then delivers it.

1.2.2 Caching in WEB 2.0

There are various strategies for server side caching. Caching frequently used queries [27], such as in MySQL server, delivers quicker response when multiple clients are looking for the same data. Also, centralized persistent key-value storage, such as Berkeley-DB [36], features hash-mapped access method. Distributed keyvalue storage systems, such as Dynamo [17], keySpace [43], etc. provide reliable and fault tolerant persistent key-value storage. Widely

used and popular memory based object caching, such as Memcached [21], provides high throughput due to its "pure in memory storage of contents.

1.3 Memory Based Object Caching With Memcached

Caching in the server side for faster content generation is a popular trend among web-services [2]. In this section I will describe how the popular solution, memcached, works.

Memcached starts as a service in the background listening to a specified port (or default 11211) for incoming requests. It provides the developers with an API to make queries, such as get (to retrieve value), set (to store a value), etc. Also, upon initialization, memcached allocates a specific amount of memory for storing contents. If a web server has, say, 1GB of memory to spare, it can run memcached in the background to store frequently accessed content, e.g. profile pictures, so that they can be delivered very fast from memcached without going to the disk to retrieve them, resulting in faster response.

Another interesting feature of memcached is, having a very simple programmatic interface for setting and getting contents, it can be scaled. For example, a simple setup can constitute of two memcached servers, while one caches just images from disk and another caches texts from database. This way a set of memcached servers within a network can create a pool of pure in memory state cache.

As more memcached server instances are added, for even distribution, clients utilize consistent hashing [25], like used in distributed hash table (DHT), to select a particular server from the pool as a storage destination for a particular content, in the form of key-value pair.

1.3.1 Managing Memcached with Membership Protocol

Though it is scalable, due to simplicity in design, this approach requires the clients of the memcached pool to use a static list of pool participants (the memcached server nodes) to use the pool. Therefore, dynamic changes to the pool go unnoticed by the clients. Also,

the pool participants remain unaware of fellow participants, the partitioning and function merely as a caching destination.

Furthermore, in case of a partial disruption of service, for some participants, not only the remaining active members are unaware of this, but also the clients notice this only after an error occurs when connecting to a pool-member. However, being bind to a static list and unaware of the actual status of the pool, the clients face a high response time while interacting to the poll as well a part of their requests go the inactive members, and fails. In this scenario manual intervention is needed to diagnose and re-organize the cache.

This not only harms the purpose of the cache, by failing to provide fast response, but also reduces the reliability. Membership protocols in distributed systems, such as Paxos group of membership protocols (discussed in Chapter 2, *Related Works*), introduces stability and reliability among arbitrary participants adhering to the membership protocol. This assures fault tolerance in the group, making them self-aware.

1.4 Scope of this Thesis

Since, a memcached pool is identified and used as a distributed cache, it is expected that it will act as a distributed system, by adhering to a membership protocol and responding to outage by self adjusting and reporting to the clients as soon as possible. Furthermore, the system needs to be easy to scale, making the dynamic changes available to the clients in real time.

Therefore, I propose Fault-Tolerant Memcached, a modification to memcached to elevate the need for manual intervention when partial failure takes down the network. I introduce a membership protocol to make the pool aware of the participants and select a Master. The protocol performs 6 tasks, as shown below.

- The pool forms a ring, organized by a Master
- Every node keeps track of their right member

- Every node notifies their aliveness to their left member
- Master adds new nodes when they comes up
- Master publishes the current status of the pool
- Upon partial failure the protocol re-organizes itself

This thesis is organized in 5 chapters. I have already discussed the emerging popularity of web services, the need of server side caching and the problem with memcached in this chapter. In the next chapter I have laid down a view of related works in this area. A design view of the protocol is given in chapter 3, followed by implementation details. Finally results are discussed in chapter 4. I have ended this with conclusion and future work in chapter 5.

Chapter 2

Related Works

2.1 Paxos Group of Membership Protocols

Paxos [33, 32] is a seminal work in consensus protocol by Leslie Lamport. It uses a state machine approach to achieve fault tolerant effectuation in services running in a distributed system. A paxos group typically rely on a leader to manage and control the group. Leader first sends a proposal number to its members and the members reply with an acknowledgment, which means they promise to accept the message of the proposal. Then the leader sends the actual message to the members to learn the message. Members, who already accepted the proposal, reply with an acknowledgment. The round fails when majority of the members fail to learn the message or multiple leaders send different proposal numbers, eventually confusing the members. At any case, the protocol ensures that after a successful round, majority of members will know the message consistently.

Paxos in Distributed Systems

There has been many uses of Paxos in different distributed systems' problems. One such instance is Chubby [13], a distributed lock service to keep the data store replicas in consistent state with high reliability and availability. Also, KeySpace [43], a distributed and replicated key-value storage with strong consistency and fault tolerance, uses a variant of Paxos. It automatically replicates the data to multiple nodes, avoiding a single point of failure. A live implementation of Paxos [14] makes noticeable modifications in the core algorithm to suit real-life failure situations un-handled in core paxos, such as disk-failure. Additionally, they reduce the message delays by reducing the need of sending the proposal

number followed by the acknowledgment, in case of a stale leader.

Furthermore, there has been variants of Paxos. Fast-Paxos [30], that reduces message delays by allowing the clients to send requests to the members directly. Cheap-Paxos [31] utilizes auxiliary leaders, which do not take part in the protocol, but takes control of the group in case of failure. However in case of too many failures, system halts until the auxiliary leaders can reconfigure the system.

Why Paxos

Coherency is a critical problem in distributed system. When N numbers of independent machines are contributing towards a common goal, in this case, better availability of caching system, it is important that they behave predictably. Such as when a new member starts and requests to joins the group, a reasonable behavior is it will be allowed to join and will be added to the active list. Also, when a member stops responding due to failure, it should be deleted from the active list and the group should be re-adjusted. However, since every machine is independent, the only way to produce such predictable behavior is by ensuring that every machine is following the same finite state machine. This way for every event noticed, the machines will go through the same procedure as others, resulting in a consistent behavior in the group.

2.2 Distributed Hash Table and Consistent Hashing

An efficient way to uniquely identify a data item is to use hash function to generate an unique identifier. These hashed values, when associated with the content, create an associative array known as hash table. This hash table can be used to scale the storage of the data to multiple location, where the hashed value can uniquely identify a particular location for the data. Using look up service [42, 24], this notion is used heavily in Distributed Hash Table, or DHT, where multiple nodes collaborate in a connected environment, and can handle extremely large number of participants.

DHT's use consistent hashing scheme to partition the data among participant nodes. The advantage of this is, adding and removing nodes require less reshuffling of data items compared to traditional hash function.

Distributed Hash Table in Application

Applications of DHT are diverse in peer-to-peer system [11]. File sharing services [41, 35] and distributed search engine [8] uses DHT for decentralization of the system. It is also used in web-caching [26] and content distribution network such as Coral [22]. Distributed file systems [1, 40, 23] and Distributed key-value storage [17, 43] also uses DHT to achieve high throughput and fault tolerance.

Advantages of DHT

One of the key benefits of DHT is the ability to form the system without a static central coordinator removing the chances of single point of failure. Also, DHT's can achieve high fault tolerance with data redundancy and can be flexible to allow joining and leaving nodes in the system. Furthermore they are highly scalable, and can function properly even with large number of systems.

2.3 Caching Trend in Popular Web Services

Popular web sites, like Facebook, Twitter, LinkedIn, Digg and others, of user-base in millions have a reasonable bound of expected loads. This makes them well aware of their requirements to handle the user-load without compromising performance at the user end. Millions of users create millions of contents and they are stored in disks, which is tedious, in terms of performance, to retrieve. Server side caching of frequently used content in memory, as mentioned in the previous section, significantly improves content generation time and thus improve content delivery latency.

Memcached, for this purpose, is widely used in various large scale sites with little to heavy modification as needed. However, the key mechanism behind this optimization is always the same, if fetching the data from storage is expensive, cache it in memory for faster access.

Facebook

With increasing popularity of Facebook, it needs to handle ever increasing requests (per/second) and massive amount of user data in the system. Currently it has 500 million active users with 200 million active mobile users, and it also gets roughly 100 billion hits everyday. It is also likely to be the largest user of memcached [5], deploying more than 800 servers with 28 Terabytes of memory for this purpose.

Twitter

Twitter's user base expanded very fast, from 0 to millions of hits happened within months. It also heavily uses memcached in their backend to support 350,000+ users with average 600 connections/per second. They provide fast response for fetching a friend's status updates, as the update resides in pure memory state [6].

LinkedIn

With more than 22 million users and heavy usage statistics, LinkedIn has deployed "The Cloud" [3], a server to cache the entire LinkedIn network map in memory with 12 GB of RAM and 40 server instances. The network size in question consists of 22 million nodes and 120 million edges. One interesting attribute of the system is it, while operating on a pure memory state, is persistent to disk on shutdown.

Digg

Based on voting among the users, Digg publishes the best information available in the web. An average statistics with an average user with 100 followers, results in 300 million

diggs per day, that's 3,000 writes per second, 7GB of storage per day, and 5TB of data spread across 50 to 60 servers. To handle this it uses a modified version of memcached [4], supporting persistent key-value storage, and stores de-normalized data from large number of normalized tables.

Others

Many popular and user centric web-sites like LiveJournal, Wikipedia, Flickr, Bebo, Typepad, Yellowbot, Youtube, WordPress, Craigslist, Mixi, etc. uses Memcached as well, tweaked and customized based on their load and requirement.

Chapter 3

Design and Implementation

3.1 Decentralized Leader Approach

Memcached, though extensively used in various web-services, does not adapt to membership changes. Therefore, it cannot be scaled up or down automatically, such as in an event of partial failure or adding more machines. Here I will discuss the design of the protocol I have integrated with memcached service.

When integrated, the protocol identifies each memcached server as members of the pool. Its main task is to keep track of the current members and make the clients, who are using the pool, aware of the changes dynamically. The protocol also selects a master node from the members to perform group maintenance tasks, while other nodes take part as slave.

3.1.1 Master Selection Within The Pool

The Master in the pool keeps track of existing members, publishes the list of active members to the clients, and performs other maintenance tasks as required. Other approaches such as Centrifuge [9], uses two different sets of nodes, where one set exclusively runs Paxos protocol to select an active master, and the other set is responsible for caching the contents. In this approach, however there are always a set of nodes who do not contribute to caching.

The protocol described here takes a different approach. It runs within the memcached nodes themselves. This design approach was chosen to reduce the number of overhead nodes in the group as well as to make the group decentralized. Instead of relying on an external set of servers to control the memcached pool, in this approach the group itself

does the necessary communication to keep itself organized, agrees on a master to represent the group and keeps itself aware of failures.

3.1.2 Cache Reliability and Availability

Paxos is an efficient way to ensure information can be consistently distributed among participants. However, systems deploying paxos ensure reliability by relying on replication of the data while paxos ensures participants are always aware of the latest data and are updated to that. The situation is different in this case. Memcached's "in memory state" cache resides in pure soft state without replication and relies on the clients to populate the data. Therefore reliability in this case does not concern the data, but concerns stability so that clients can reliably use the pool and be updated about the latest participants. Furthermore, as the master of the pool resides within the pool itself and not an outside entity, losing a member and a master has the same impact on the pools structure, compared to failing a master from a masters group running outside the pool. With this protocol, every member keeps track of their position in the pool consistently and responds to changes it notices.

In this protocol, the participants form a unidirectional ring and each keeps track of their either left or right node's availability. From start, one of the members takes the responsibility of master of the pool and it publishes the current state periodically. When a member or even the master stops responding due to failure, the remaining members adjust themselves and one of them is selected to replace the failed master. This way the clients are always informed about the current members of the pool.

Paxos ensures consistency by ensuring that majority of the participants will always agree on the same outcome for a particular event, such as electing a group leader, through communication within the participants. The same is achieved in this case by ensuring every participant will produce the same result for such events. In case of an election, due to current master being unresponsive, the member gets selected to represent the group is consistent among all the participants outcome. Using the unidirectional ring of participants,

everybody mutually selects the member that comes after the missing master in the ring to represent the group. This procedure although happens without any communication, ensures the pool will always have a master after the last master fails to respond. Furthermore, when duplicate masters come into existence, due to lack of communication in the election phase, they soon discover each others presence. In that case they mutually agree upon an unique master from the duplicates who can continue and others marge in. They use a consistent winner selection routine, based on who is contributing most to the pool or, in case of equal contribution, who has lower address.

3.1.3 Ring Formation and Keeping Track of Members

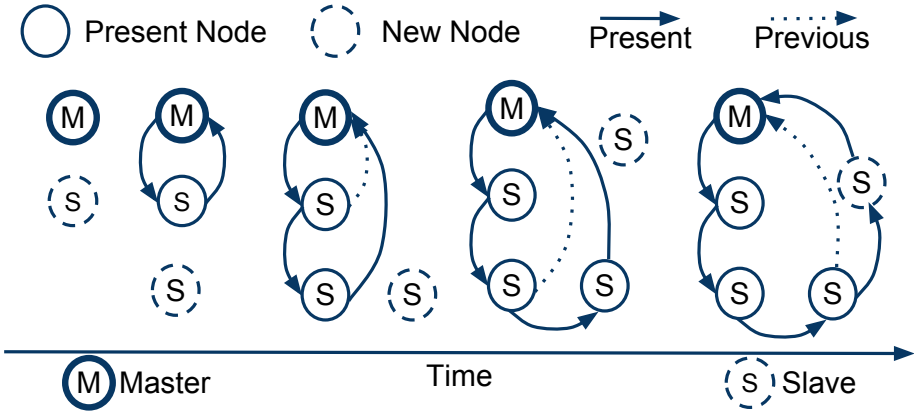

Figure 3.1: Ring Formation Within The Pool

The protocol forms a unidirectional ring within the pool with its participant nodes, as described in the previous section. Therefore in the pool every node has two adjacent nodes, node before it and the node after it, but the pool itself is represented as a list starting from master, going to the node after it, to all the way to the node before the master.

A node, when starts the protocol as master, creates a list with single entry of itself only. In future when new node joins, the master appends the node to the list and updates itself and the new node about their before and after nodes (see Fig. 3.1).

Every node periodically sends a message to the node after it in the ring. By receiving that message every node keeps track of the node before them in the list, referred as the neighbor, about the nodes aliveness. In case a node does not receive the message in some time, it assumes the node before it to be missing and takes necessary steps to reorganize the pool.

3.2 Message Passing Architecture

The protocol depends on communication and uses various types of messages for the same purpose. Such as, publish messages detail current pool members, join message requests joining of new members, missing message identifies a missing member, etc. Details of the messages are described in this section.

3.2.1 Communication Messages

Every message in this protocol has a message type and a message body. The message types tell which event has occurred and subsequently determine which action to take and the message body details the event and acts as the information needed to perform that action. The messages are described below.

PUBLISH This message contains the current active list to the client as well as to the members. Clients use it to (re)initialize themselves to use the pool, while the members use it check consistency within the group.

JOIN This message is sent by a new member, receiving a publish message, to request the current master to join the group.

JOINOK The current master sends this message as a response to the `JOIN` message acknowledging it to join. Furthermore, this message tells the new member about its adjacent members in the group.

ALIVE This message is sent by every member to their adjacent member in the group, whether they are in master or slave state. This tells their aliveness to the adjacent member.

MISSING If the `ALIVE` message is not received by a member for some time; it reports the current master about the missing member with this message. A member expects a response (`ADJUST`) for this message and repeats until that is received.

ADJUST Upon receiving a `MISSING` message, the master removed the missing member and responds with this message to let the reporting node adjust to the modified pool.

ADJUSTOK An `ADJUST` message is responded with this message.

REGROUP In case of various inconsistencies, this message is used by old, new or duplicate master to notify its members to rejoin a new master. This ensures that members join fresh while maintaining the consistency of the pool.

3.2.2 Fault Tolerance due to Message Loss

The protocol uses UDP [7] to communicate messages. This was chosen to reduce message overhead in the network due to this protocol. However, since UDP is not reliable, the protocol uses a pre-configured value for fault tolerance, say N. When a member (memcached node) expects a message but never receives it, it tries to receive the same for N times before giving up. This provides an easy way to deal with message loses with N-fault-tolerance. When a node is up and running, but the expecting message is not delivered, it is not immediately considered as not-alive, and can still function as a member in the group upto N message loses.

Furthermore, the protocol listens for incoming messages for a pre-configured time, say T seconds, and stores them in a buffer for later processing. This also limits continuous flow of messages in the network for maintaining the group, as all the communication is performed every T seconds.

These two configurations introduce a delay in the failure discovery process. In an event of real failure, due to T and N, it will not be actually discovered for (TxN) seconds. Actual values of T and N need to be adjusted based on network capacity, load and message loss probability.

3.3 Server Side Protocol Description

The protocol consists of two distinct parts, a client part that uses the pool and a server part that adheres to the group membership protocol. At the server side the protocol runs parallel with all communicating participants, while a mutually agreed upon group master tries to maintain consistency inside the group. The clients however run independently listening for the published group members by the current group master about the current state of the pool.

The server side protocol uses a finite state machine where a server can be at any one of the defined states and it changes state based on the messages received and outcome of the actions performed. This simplifies the behavioral model of the protocol.

3.3.1 High-Level Protocol Description

The server protocol consists of 6 distinct high level states' cloud, henceforth referred as stages, namely `Initialization`, `Pre Grouping`, `Pre-Master`, `Master`, `Slave` and `Recovery`. Each of these stages have distinct enter and exit criterion and they help to describe the high level operation of the protocol (see Fig. 3.2) clearly.

At `Initialization`, the protocol tries to find the status of the local memcached server. If no local memcached server is presnt it exits, otherwise upon success it starts listening for publish message from the current group master (see Fig. 3.3). If such message is received, it moves to `Pre-Grouping` stage. Alternatively if it does not find any existing group to join after a predefined time, it moves to `Pre-Master` stage.

In `Pre-Grouping` (see Fig. 3.4), the protocol sends a join request to current group

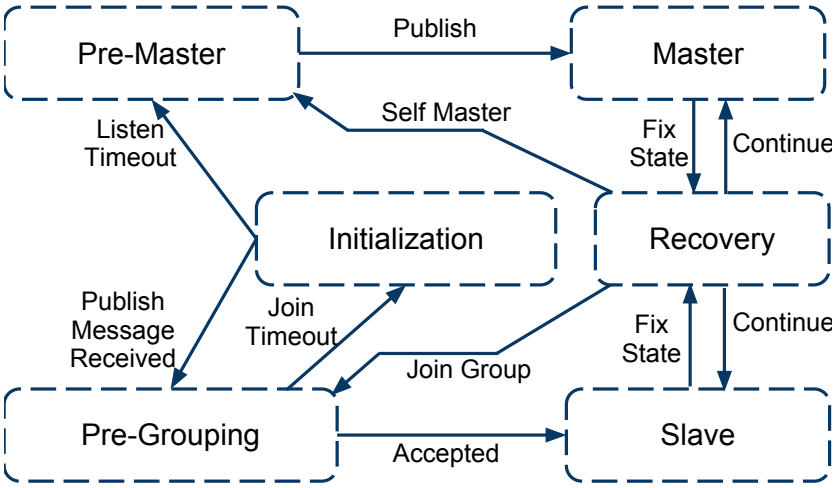

Figure 3.2: High Level Protocol Diagram

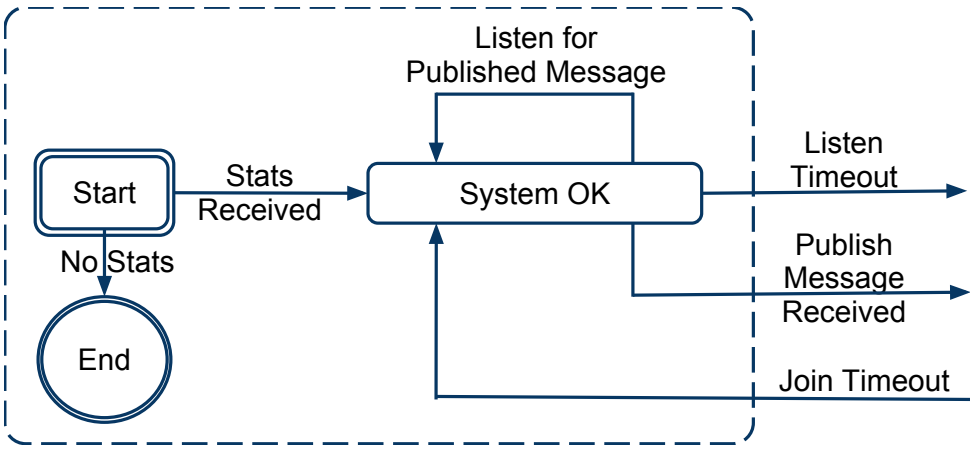

Figure 3.3: Initialization Stage

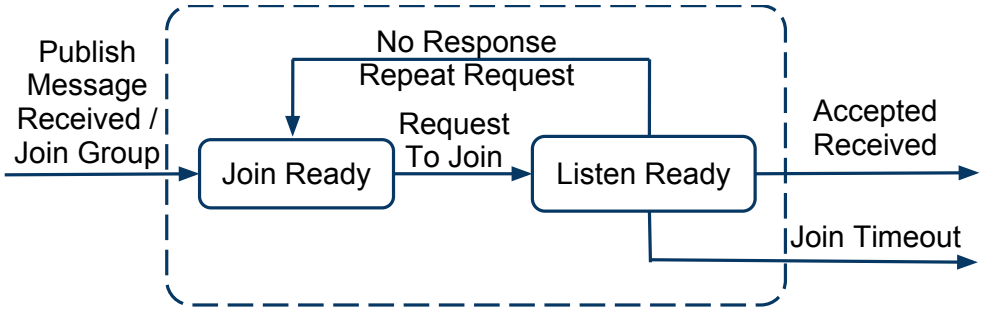

Figure 3.4: Pre-Grouping Stage

master and waits to receives an acknowledgment to join the group. Upon receiving, it moved to `Slave` stage and joins the group.

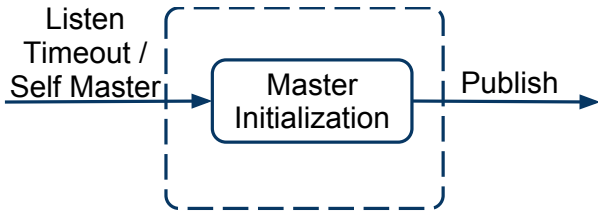

Figure 3.5: Pre-Master Stage

In `Pre-Master`, the protocol proclaims itself as master and publishes itself as current group. It then moved to `Master` stage for future participants to join (see Fig. 3.5).

The `Master`, `Slave` and `Recovery` stages (detailed in the next section) responsible for maintaining the group consistency and scaling the group dynamically. Where `Master` and `Slave` stages tries to maintain consistency in normal operation, in case of any inconsistency, such as partial failure, the protocol moves to `Recovery` stage to fix the problem.

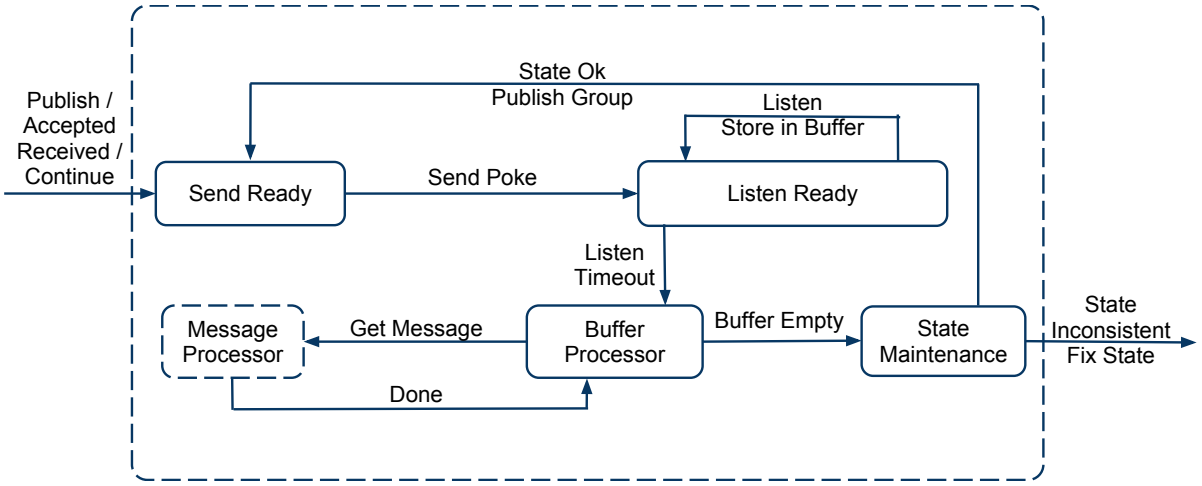

Figure 3.6: Master and Slave Protocol Diagram

3.3.2 Detail Description of The protocol

Master and Slave Stages

The `Master`, `Slave` stages are identical in nature but differs by the type of messages they accepts to process (see Fig. 3.6). They consists of `Send Ready`, `Listen Ready`, `Buffer Processor`, `Message Processor`, and `State Maintenance`. Moreover, `Message Processor` is another stage that consists of various states, which is described later.

In this stage, the protocol first sends aliveness message to its neighbor and starts listening for incoming messages for a predefined time, while saving the arrived messages in buffer for future processing. After, the predefined time is over, it starts processing the messages in first in first out manner until the buffer is empty. In `Message Processor`, it also updates the its current status as per the messages received. Finally, when the buffer is empty, the protocol checks consistency of it's current status. If it is ok, the protocol returns to start of their respective `Master` or `Slave` stage to repeat itself. However in case the state is inconsistent, it tries to fix it by moving to `Recovery` stage. Also, for the `Master`, while returning, it publishes the current status for the clients.

Message Processor in Master and Slave

`Message Processor`, of simply processor, differs in `Master` and `Slave` by the messages it accepts to process (see Fig. 3.7). In both `Master` and `Slave` stages, processor accepts `alive` message from neighbor which eventually assures aliveness of the group.

Additionally, in `Master` stage, processor accepts and processes the following messages.

- `Join Request` Adds the member, updates group and sends adjust to affected members.
- Member `Missing Report` Removes the member, updates group, and sends adjust to affected members.
- `Adjust OK Report` Updates that member is up to date.
- `Publish Group` If not No self published, updates status with master duplicate.

Also, in `Slave` stage, processor accepts and processes the following messages.

- `Adjust Received` Adjusts self and sends Adjust OK message.
- `ReGroup Requested` Sends regroup message to the new master.
- `Published Group` If inconsistent, updates group status.

A list of messages used in this protocol is listed with their description in Chapter 4, *Implementation*.

Recovery State

Every node can detect three types of failure Fig. 3.8). First when a member node is missing and the group needs to be adjusted, second when the master itself is missing and a new master needs to be elected, and finally when more then one nodes, missing each others' communication messages, becomes master creating multiple pools. In this

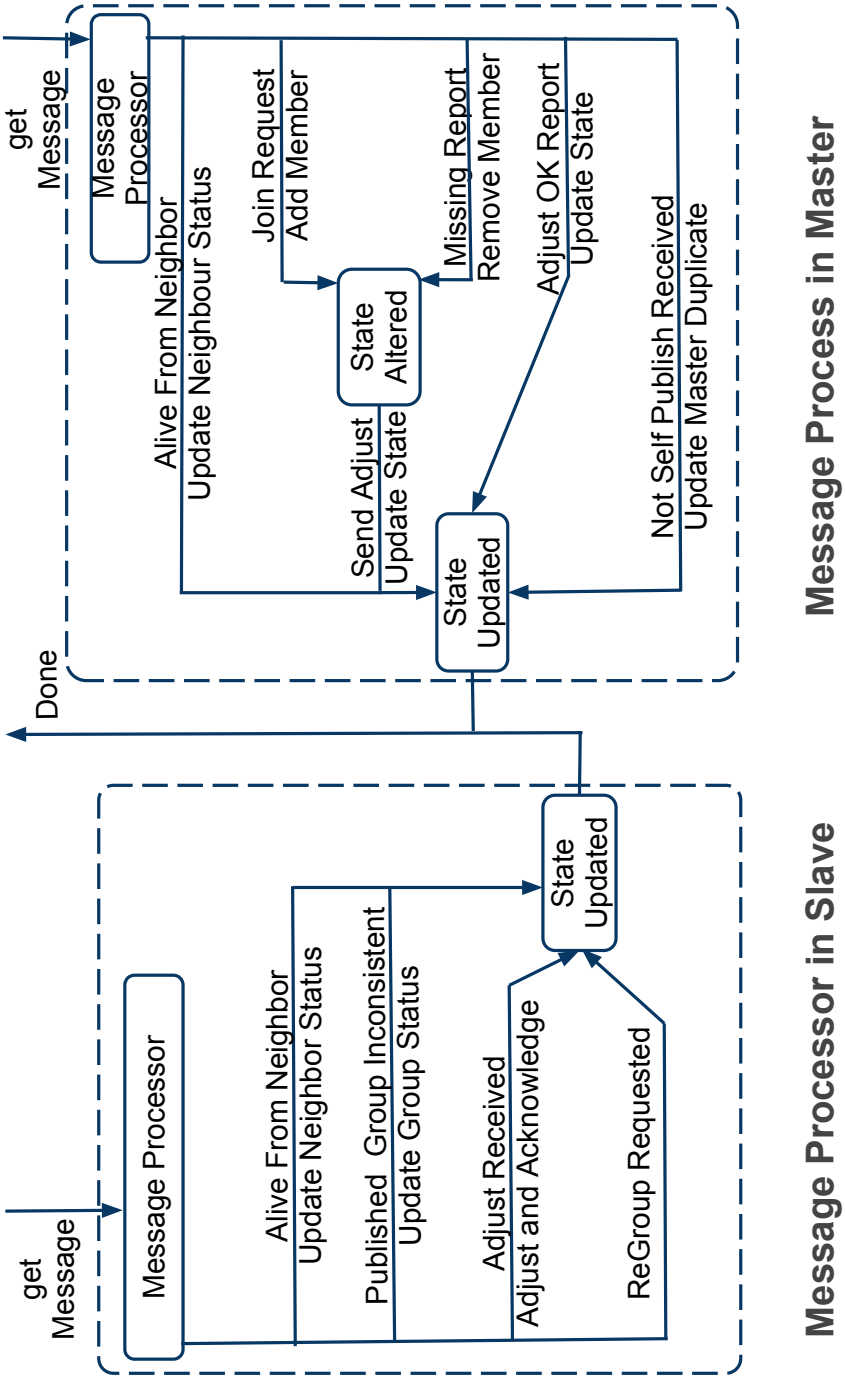

Figure 3.7: Process State in Master and Slave

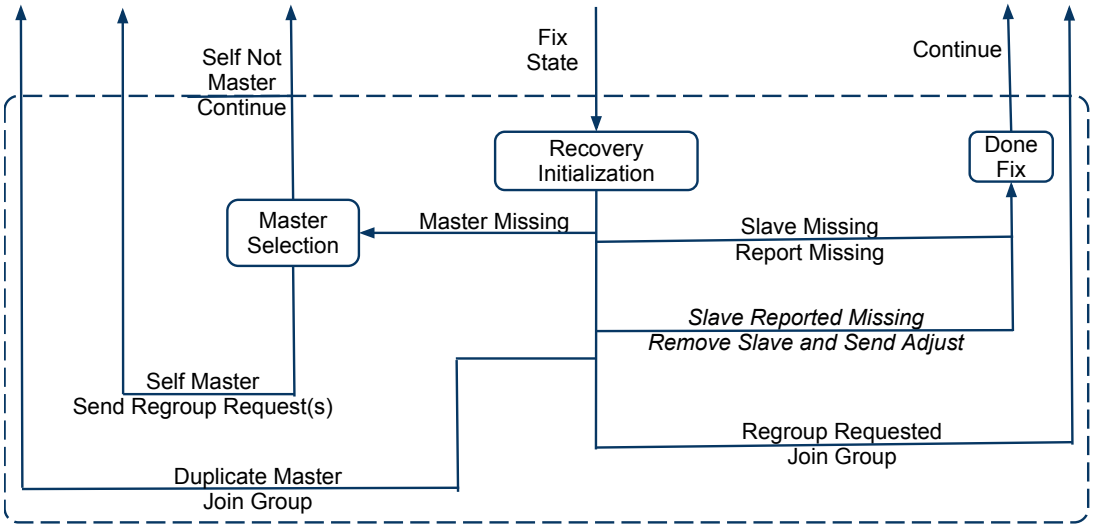

Figure 3.8: Recovery State Protocol Diagram.

duplicate masters' case, upon discovering conflicting broadcast messages, both masters' apply a consistent master selection algorithm. The selected one remains as master, while another master notifies its members to re-join the winning group.

Recovery stage consists of 3 states, namely `Recovery Initialization`, `Master Selection` and `Done Fix`. Furthermore, from the `Recovery` stage the protocol returns to respective `Master` or `Slave` stage in case of minor failure, such as a slave fails to respond. In case the master itself fails to respond, the protocol returns to `Pre-Grouping` or `Pre-Master` based on if the node is selected as master.

Recovery from missing Member When a neighbor stops responding, when noticed by a slave member, it sends a message to the master node and expects a confirmation message to adjust itself. In such case, the master re-organizes the group and sends the confirmation message. Failure of more than one consecutive node is handled in the same manner, removing and sending confirmation one at a time. However, if the master notices a missing slave, it removes the node itself and sends adjust to affected members.

Recovery from missing Master When the master itself becomes unresponsive, the protocol explicitly invokes a master selection procedure. To maintain the consistency at the client end, it tries to maintain the pool structure while reorganizing the pool. Therefore, the node next to the current master in the list of members becomes responsible to lead the group. It then removes the unresponsive master, and sends a regroup request to all the remaining nodes. The other nodes, upon receiving the regroup request, join the group. Here also, failure at multiple nodes are handled sequentially. If the node very next to the master also fails with the master node, the node next to it will automatically select itself in the recovery process, and recovery process will not stop until one node becomes master.

Recovery from Duplicate Master When two or more nodes start simultaneously, they all might start as master missing each others' communication. This race condition is handled in the recovery state as well. The protocol determines which pool should continue and which should not based on their participants' size. The bigger one can live on while the smaller ones need to join the bigger one. When a pool detects such master conflict, its' members decides which pool to join as well as the master sends a regroup request to all of its members to do the same. Eventually all the members ends up joining in a bigger pool, resolving all the conflicts and duplicates.

3.4 Client Side API Modification

Due to lack of organization mechanism in memcached servers, the memcached API uses a static list of servers to initialize itself. It also uses it for consistent hashing and selecting which node where a particular key will be stored. This increases complexity when more memcached instances need to be added for more capacity. Also, clients become aware of a failed/removed node when storing data on that node fails, increasing the response time in case of partial failures.

Using this protocol the organization is simplified and automated. Instead of a static

list now the client listens for broadcasts from the current group leader, and initializes itself based on the message received. This way, In case of a failure, the clients can automatically adjust the pool, without having to face long timeout delay before realizing an instance is unresponsive. Furthermore, this approach simplifies scalability, adding more nodes automatically makes the clients aware.

3.4.1 Client Side API

Clients of the memcahced pool listens for broadcasts from the current master and initializes itself using the active participants' list in the broadcast message. Clients run a thread in the background for this purpose, and if it notices any change in the current pool, it adjusts itself based on the current active list in the broadcast message. The client API provides methods like *get(key)* and *set(key, value)*. When there is no active pool, the clients fail to receive message from the master, and consequently fail to initialize. Then any call to the above-said (*get/set*) functions return *False* and *0* respectively. In this case caching is disabled. However, when an active pool becomes ready, the API automatically initializes a distributed hash table, and the key-value pair is stored in appropriate node. Here also, a *get(key)* operation for a non-existent key returns *False*.

The *get(key)* and *set(key, value)* methods provide an easy access to the memcached pool for storing and retrieving data. Furthermore, the API removes the need of a static list to initialize the client and provides dynamic organization of the pool members. This also improves availability in case of failure, then clients automatically adjusts their copy of distributed hash table to reflect the changes in the pool. Same advantage is gained when more nodes need to be added to handle extra load, the clients, upon receiving an updated list, automatically reorganize themselves.

3.5 Implementaion Details

3.5.1 Server Side and Client Side

There are two distinct sections in the *Implementation*, a *Server* side that creates the pool and the *Client* side that uses the published pool. Furthermore, for simplicity the server side is implemented with Master and Slave separately in different function.

Server Side

At the server side, the program uses event-driven approach. Messages act as events and different actions are performed based on the messages processed from the stored messages, resulting in a change in the current state.

The program follows the protocol, described in the design section. It listens and stores incoming messages for a predefined time, then it processes them one at a time and changes its state accordingly. Then it performs a maintenance on its current state to check if everything is consistent. If not, it calls appropriate function to fix the issues.

Client Side

At the client side, a threaded approach is chosen. One thread constantly listens for the published group-list. If it receives a list that differs from the one currently in use, it updates itself. Furthermore, there are API function such as `set(key, value)` and `get(key)` to interact with the pool. Also, if it does not receive any published group for a defined time, it disables caching all together, assuming the pool has fully failed.

3.5.2 Implementation

The protocol is implemented in Python, but can be implemented in any language. Python was chosen for its simplicity and inbuilt features that match the requirements of the protocol. Some of these features are discussed below.

Tuple Python features tuple objects where two or more data items can be clubbed together and can be used without declaring a data structure first. It is used extensively due to this feature.

Dictionary Python supports dictionary objects that stores key-value pairs. This is very useful and can be used as a message buffer. Keys for the dictionary are tuples of sender's address and a sequences of indices, while value is the message itself.

List Python's List is like a standard array but it features inbuilt methods for managing the list. This is used to manage the pool members.

Select The protocol's listen state is implemented with asynchronous i/o. The motivation was to reduce processing requirement or idling the protocol while listening for messages. Python's select module provides required support for this.

Threading Python's threading module features easy to use synchronization methods for locking shared objects. This was useful in Client side for running a thread in background to update itself to latest status of the pool.

Chapter 4

Experiments and Results

4.1 Experiment Setup

4.1.1 Designing The Pool (of Servers)

A distributed system, such as one to deploy this protocol, can be assembled using some computers with network interface and a router to connect them all in a local network. However, that setup is not portable, clumsy and debugging is harder with such setup. Due to this, I decided to setup my own virtual and portable system pool under Windows 7, using available open-source solutions. I have used a popular open-source virtualization solution from Sun/Oracle, called VirtualBox (Version 4.0.4) and a bare minimum linux distribution called Micro-Core-Linux, a command line variation of Tiny-Core-Linux for machine instances because it.s a pure in memory operation, requiring 36MB for Micro-Core and 48MB for Tiny-Core, of RAM per instance. Also, Cygwin is used for various scripting and interfacing with the pool.

Pseudo Hardware Setup (VirtualBox VMs)

In VirtualBox, each machine is given 128MB of RAM, 1GB of Virtual Disk space and a Virtual CDRom. Furthermore, since the goal was to create a bare minimum setup, I have removed 3D/2D video acceleration, sound and USB support and only networking was enabled with host-only mode. I have chosen Micro-Core-Linux over Tiny-Core-Linux, mainly because of its even lighter requirement, no X-server and very small size (6.7MB ISO). It uses a specific target folder from the persistent disk, named “tce“ to save and load

application, known as Tiny-Core-Extensions (tce). Also, by design the core system is not persistent, so any configuration change made for specific application is lost upon reboot. However, it comes with a tool that can backup specific files in a compressed tar archive (tgz) upon proper shutdown and restores those file when booting up. With these tools configured in order, I setup the first system to have a memcached instance and an openssh server for remote login and running programs. Then I added the list of configuration files I created/modified to the backup tool to have a persistent setup upon reboots. Then I made duplicates (clone) of the virtual hard disks of this setup by using command line tool of VirtualBox, that creates a clone of the disk. Simple copy is not usable in this case as each disk is assigned with an unique "Universally Unique Identifier" (UUID) and it is not possible to use two disk of same UUID in the same VirtualBox installation. Finally for each cloned disk, an identical system was created and the disk was assigned.

Pseudo Software Setup (shell scripts)

I have created various scripts in cygwin for easy interaction with the created pool. Unfortunately the VM's do not startup when invoked from within Cygwin using native cygstart (used to invoke windows programs). I have used a batch script for that purpose. Upon invoking it prompts for a specific machine id to start or if nothing entered, it starts all VM's. This behavior is selected based on my needs. After they start up, it is easy to communicate with them with ssh client in Cygwin (as those machines already have an openssh running). Also, I have created several scripts such as vm_check, vm_cp, vm_connect and vm_run. Again these only work for the particular setup I have created but can be customized as needed. I have also used a file called "vmlist". containing the list of created local VMs. For simplicity, I have used only the last octave of the ip of each machine as identifier, since in host-only mode VirtualBox assigns IP addresses ranging from 192.168.56.101-254, only changing the last octave.

Below are the outputs from the scripts I created, showing their usage details.

vm_check vm_check <all|xxx> ...

specify to check ‘all’ or specific ‘xxx’ m/c

automatically assumed ip address 192.168.56.xxx

vm_connect vm_connect <xxx>

automatically assumed ip address 192.168.56.xxx

vm_cp vm_cp [-d] [-a <xxx>] <file> [<file> ...]

with ‘-d’ directories from specified file path will be created at /

with -a <xxx> copy only to address ‘xxx’

automatically assumed ip address 192.168.56.xxx

also works with simple regular expression, *.ext, etc.

vm_run vm_run [-a <xxx>] <remote-cmd> <arg1> <arg2> ...

with -a <xxx> run only at address ‘xxx’

automatically assumed ip address 192.168.56.xxx

automatically a pseudo terminal from remote m/c will be created

vm_stop vm_stop <all|xxx> ...

specify to stop ‘all’ or specific ‘xxx’ m/c

automatically assumed ip address 192.168.56.xxx

4.2 Results

The protocol is tested with two scenarios, with a huge difference in the length of the listen state or the interval. Where the first one is 5 seconds long, a higher one, makeing the protocol active only 12 times a minute and idling most of the time, the second one is rapid 100 milliseconds, a lower one, making it fast and responsive with 600 times active a minute.

The protocol is tested with 5 memcached instances, each initialized with 64MB ram and the protocol is configured with a fault tolerance of 2, meaning it will try maximum 2 times before giving up on a unresponsive node. A lower value is chosen for the fault-tolerance as the internal network behaved very stable.

The clients, adhering the protocol, were configured to listen for 10 seconds before deciding on a dead pool. This enabled them to try endure enough broadcast misses.

4.2.1 Responsiveness With Higher Interval

4.2.2 Awareness With Dynamic Changes

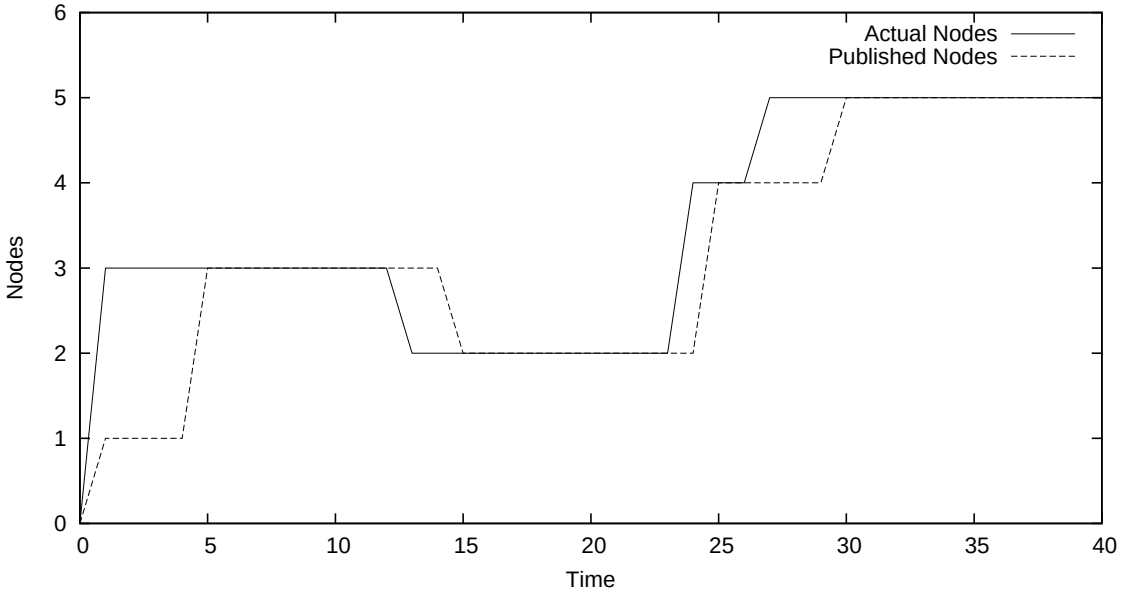

Figure 4.1: System Responsiveness at Higher Interval

The memcached instance and the protocol is configured to start with the machines themselves and an external listener is attached to the network to trace the published groups

with relative time-stamp. This was done with a higher interval (5 second). First 3 nodes are started followed by a removal and joining of two more nodes. Each such incident's relative time of occurrence is noted. Finally, trace from the listener is merged with the noted times and plotted to reflect the responsiveness of the system (see Fig. 4.1).

Furthermore, the similar experiment was conducted with a lower interval (100 milliseconds). In this case, starting of 5 machine, followed by removal of 2 machines were automated with an delay of 5 second in between. The actual nodes up with the published nodes are plotted to reflect the system responsiveness in lower interval (see Fig. 4.2).

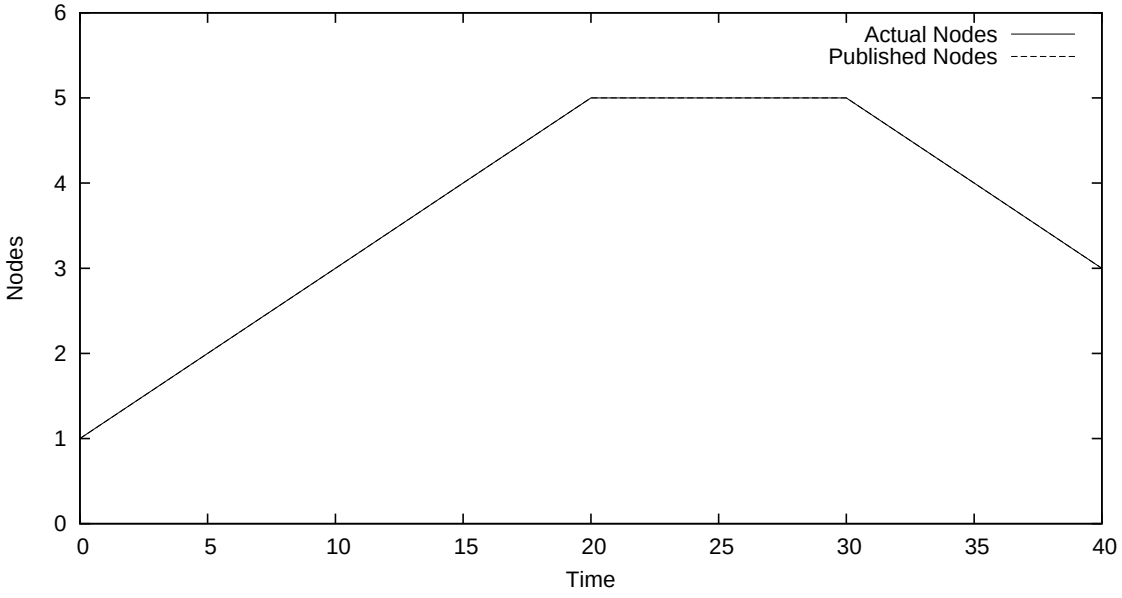

Figure 4.2: System Responsiveness at Lower Interval

4.2.3 Responsiveness In Simple Failure

Two clients are configured with and without adaptation of the protocol. One using a static list while another listening to the published ones. Both of them loops indefinitely, creates an unique key-value pair and stores them using the `set` method and then tries to retrieve it with `get` method. It is also configured to take an argument to decide how often it should report back with the statistics. Every such occasion it writes the statistics of time taken and no. of cache misses (`get/set` operation). Furthermore, any such failure results a penalty of 100 milliseconds, to reflect fetching of the real content from a slower source.

This setup was put under test by shutting down a single node abruptly while the test clients are running.

Client Without The Protocol

Client who did not adapt to the protocol, as expected, took longer to respond after the failure. This is due to part of the keys assigned to the removed machine, were facing penalty (see Fig. 4.3).

Client With The Protocol

The client which adapted the protocol, also showed longer response time, but for a limited time only (see Fig. 4.4). This is due to the same reason as previous client. However, the response time eventually reverted back to normal. This delay was caused by the time it took discover the removed node, as shown in Fig 4.1.

4.2.4 Handling Complex Failure

The same test is performed with removal of multiple node, one slave and the master node. the objective is to study the failure pattern for both client with and without adapting the protocol.

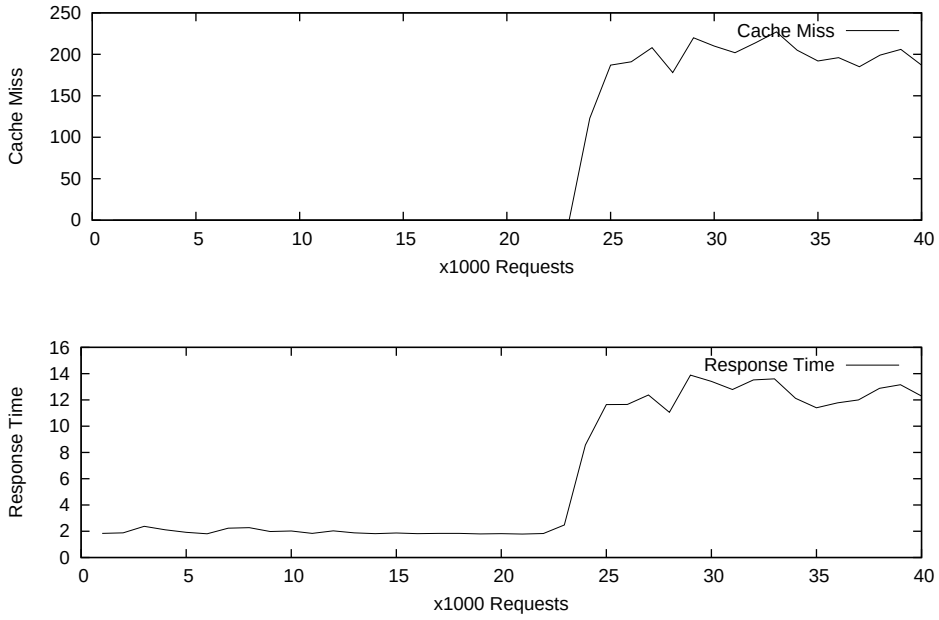

Figure 4.3: Client without Protocol in Simple Failure

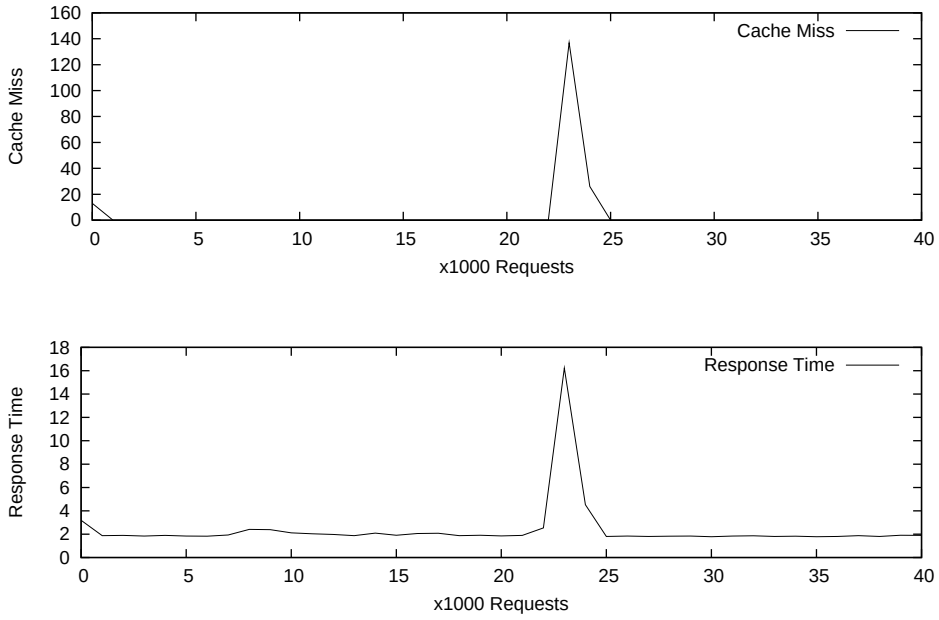

Figure 4.4: Client with Protocol in Simple Failure

Client Without The Protocol

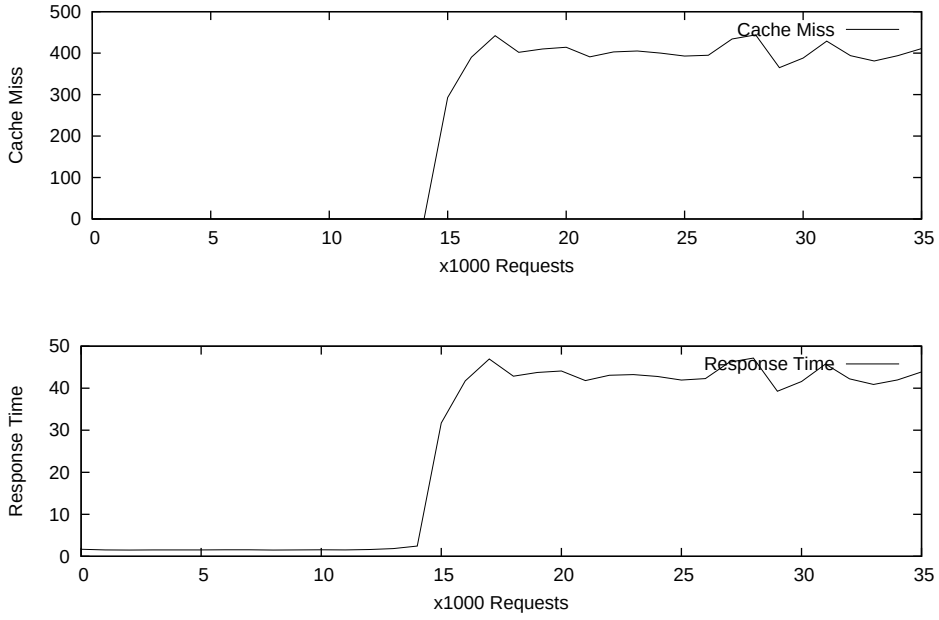

Figure 4.5: Client without Protocol in Complex Failure

In this case the response time increased significantly (see Fig. 4.5), due to the fact that 2/5 of the cache capacity is reduced, therefore 2/5 of the keys were facing penalty.

Client With The Protocol

Same as above the response time increased but eventually it reverted back to normal. Here again, the delay caused while discovering the modified pool (see Fig. 4.6).

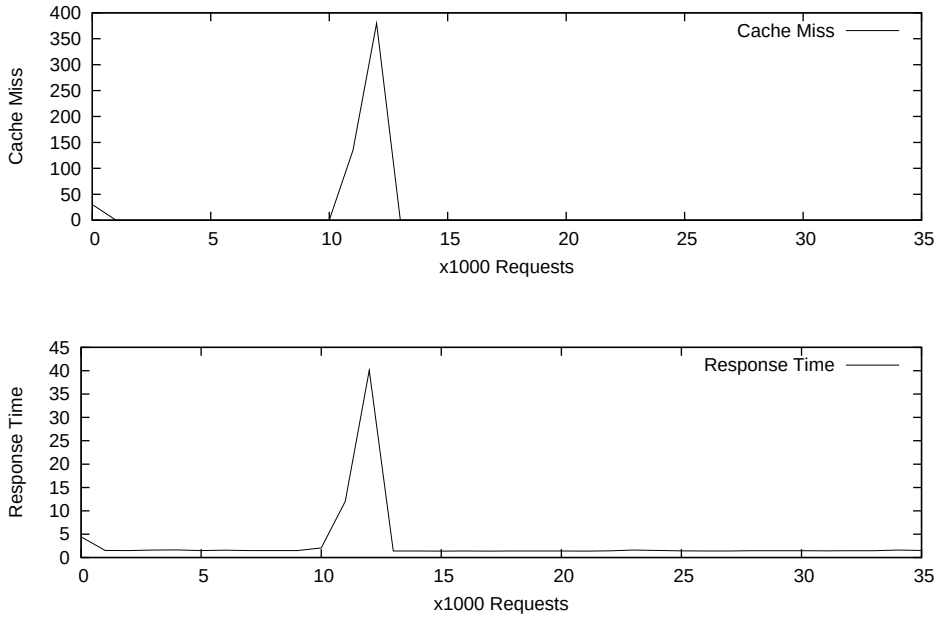

Figure 4.6: Client with Protocol in Complex Failure

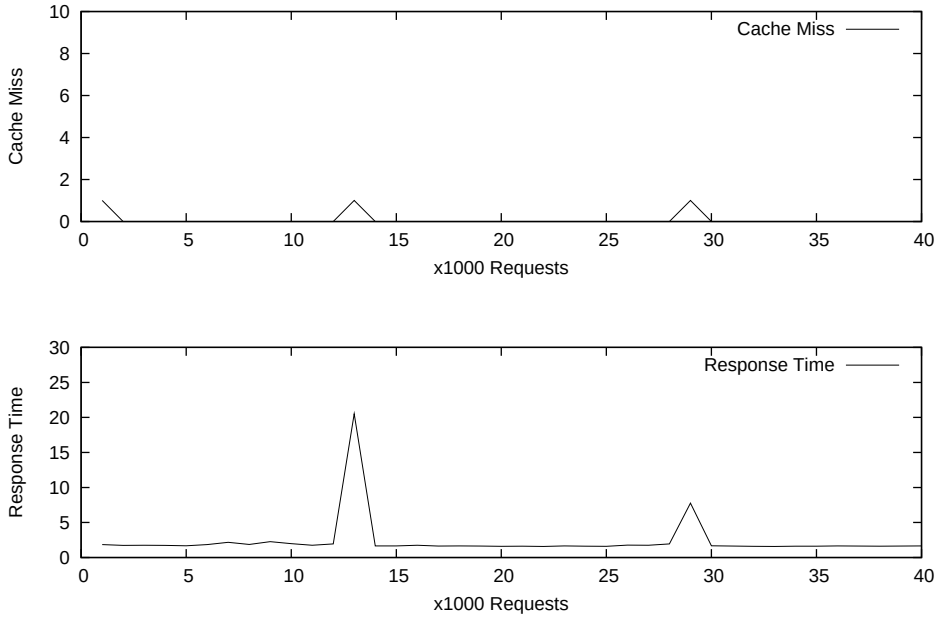

Figure 4.7: Client with Protocol With Lower Interval

Responsiveness With Lower Interval

The protocol is again tested with a faster interval (100 millisecond). The response of the clients without the protocol is irrelevent, as they are not affected by this, and is ignored.

Though the response time does not improve dramatically, one interesting thing to notice is the reduction of misses. Due to the fact that the group status is evaluated every 10th of a second it faces very minimal misses and re-organized almost instantaneously when the pool changes. An interesting application of this is discussed in the *Future Works* section.

4.2.5 Interpreting The Results

The protocol is aware of dynamic changes as it is showing in Fig. 4.1. The configured interval time (time spent listening) between broadcast publishes causes a delay in real discovery of added/removed nodes. However, as discussed (in section 4.1.2) previously, this was a design choice to reduce overhead messages in the network due to usage of this protocol. Furthermore, adding more nodes dynamically for increased cache size is an interesting feature of the protocol. This allows dynamic scaling without manual intervention.

Also, it smoothly with dynamic changes. As it shows in Fig. 4.3 a client without adapting the protocol takes more time to respond, compared to one taking advantage of the protocol as in Fig. 4.4. The delay is caused as it tries to use a unresponsive server consequently facing failures.

Furthermore, as shown in Fig. 4.6, the protocol handles complex failures and client resumes normal operation when adapted the protocol. With the same failure the client who did not adapt the protocol, faces significant delay in response, as shown in Fig. 4.6.

Runtime configuration can also be tweaked to achieve faster awareness. A smaller interval between broadcasts will allow quicker discovery of changes as shown in Fig. 4.4.

Chapter 5

Future Works and Summary

5.1 Future Works

There are several extensions to this work. Being dynamic in nature, the protocol can be extended to a "need-basis" caching. With this, any node with reduced load and spare memory in hand can start the protocol and join a pool. In future when its own load increases, it can quit safely without disrupting the service. With results shown in Fig. 4.4, a faster system response can alleviate delays due to frequent changes in the protocol. This will dynamically provide caching storage based on load . Furthermore, the protocol can be further enhanced to support dynamic participation sizes of nodes. Instead of completely removing itself, a node may change the amount of memory allocated for caching and still be part of the pool. To improve performance and tighter integration the protocol can be implemented within the Memcached process itself and faster processing due to native c code.

There is another possible extension, that with data duplication, some level of data availability can be assured in case of partial failure. With awareness in the group the protocol can be further modified to allow every node to use half of its original capacity for storing data it is responsible and logically partition it in equal halves. Each such partition will consume a quarter of its allocated memory. The other half, split in half again, can be used to store one such logical partition from its left and right node. In this case when a node goes missing, the data can be distributed upon adjusting and can be available to the clients. However, this will increase overhead significantly due to data replication and will

reduce the total capacity of the pool in half. Therefore this requires further investigation, that whether data availability can be a trade off for reduced storage capacity and maybe slower performance for a "soft state cache".

5.2 Summary

This thesis was motivated by poor performance of Memcached under partial failure, resulting in delays in response. Furthermore, scalability was an issue which the system does not address well. In this thesis I tried to explore a new direction of server side caching, with dynamic adaptation to changes. I have designed and implemented a group membership protocol that can dynamically adapt to failures and report clients about the changes due to the failure. I have also discussed test scenarios of clients with and without the protocol in the results.

References

[1] Amazon simple storage service. http://aws.amazon.com/s3/.

[2] Free & open source, high-performance, distributed memory object caching system. http://memcached.org/.

[3] Linkedin architecture. http://hurvitz.org/blog/2008/06/linkedin-architecture.

[4] Scaling digg. http://highscalability.com/blog/2009/2/14/scaling-digg-and-other-web-applications.html.

[5] Scaling memcached at facebook. http://www.facebook.com/note.php?note%20id=39391378919.

[6] Scaling twitter. http://highscalability.com/blog/2009/6/27/scaling-twitter-making-twitter-10000-percent-faster.html.

[7] User datagram protocol. http://tools.ietf.org/html/rfc768.

[8] Yacy. http://yacy.net.

[9] Atul Adya, John Dunagan, and Alec Wolman. Centrifuge: integrated lease management and partitioning for cloud services. In *7th USENIX conference on Networked systems design and implementation*, NSDI'10, pages 1–1, Berkeley, CA, USA, 2010. USENIX Association.

[10] Charu Aggarwal, Joel L. Wolf, and Philip S. Yu. Caching on the world wide web. *IEEE Trans. on Knowl. and Data Eng.*, 11:94–107, January 1999.

[11] S Androutsellis-Theotokis and D Spinellis. A survey of peer-to-peer content distribution technologies. *ACM Computing Surveys, Vol 36, Issue*, (4), 2004.

[12] Andrei Broder, Ravi Kumar, Farzin Maghoul, Prabhakar Raghavan, Sridhar Rajagopalan, Raymie Stata, Andrew Tomkins, and Janet Wiener. Graph structure in the web. *Computer Networks*, 33(1-6):309 – 320, 2000.

[13] Mike Burrows. Chubby distributed lock service. In *In Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, 2006.

[14] Tushar Chandra, Robert Griesemer, and Joshua Redstone. Paxos made live - an engineering perspective. In *In Proceedings of the 26th ACM Symposium on Principles of Distributed Computing (PODC 07*, 2007.

[15] Anawat Chankhunthod, Peter B. Danzig, Chuck Neerdaels, Michael F. Schwartz, and Kurt J. Worrell. A hierarchical internet object cache. In *Proceedings of the 1996 annual conference on USENIX Annual Technical Conference*, pages 13–13, Berkeley, CA, USA, 1996. USENIX Association.

[16] G Cormode and B Krishnamurthy. Key differences between web1.0 and web2.0. *First Monday*, 13(6):1–30, 2008.

[17] G DeCandia, D Hastorun, M Jampani, G Kakulapati, A Lakshman, A Pilchin, S Sivasubramanian, P Vosshall, and W Vogels. Dynamo: Amazon.s highly available key-value. *Store,. ACM SIGOPS Operating Systems Review*, (41):205–220, 2007.

[18] Li Fan, Pei Cao, Jussara Almeida, and Andrei Z. Broder. Summary cache: a scalable wide-area web cache sharing protocol. *SIGCOMM Comput. Commun. Rev.*, 28:254–265, October 1998.

[19] Li Fan, Pei Cao, Wei Lin, and Quinn Jacobson. Web prefetching between low-bandwidth clients and proxies: potential and performance. *SIGMETRICS Perform. Eval. Rev.*, 27:178–187, May 1999.

[20] Anja Feldmann, Ramn Cceres, Fred Douglis, Gideon Glass, and Michael Rabinovich. Performance of web proxy caching in heterogeneous bandwidth environments. In *IN PROCEEDINGS OF IEEE INFOCOM 99*, pages 107–116, 1999.

[21] Brad Fitzpatrick. Distributed caching with memcached. *Linux J.*, 2004(124):5, 2004.

[22] M J Freedman, E Freudenthal, and D Mazires. Democratizing content publication with coral. In *In Proc. of NSDI.04*, 2004.

[23] S Ghemawat, H Gobioff, and S-T Leung. The google file system. *SIGOPS Oper. Syst. Rev*, (37):2003.

[24] M F Kaashoek and D R Karger. Koorde: A simple degree-optimal distributed hash table. In *In Proceedings of the 2nd International Workshop on Peer-to-Peer Systems*, 2003.

[25] David Karger, Eric Lehman, Tom Leighton, Rina Panigrahy, Matthew Levine, and Daniel Lewin. Consistent hashing and random trees: distributed caching protocols for relieving hot spots on the world wide web. In *Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*, STOC '97, pages 654–663, New York, NY, USA, 1997. ACM.

[26] David Karger, Alex Sherman, Andy Berkheimer, Bill Bogstad, Rizwan Dhanidina, Ken Iwamoto, Brian Kim, Luke Matkins, and Yoav Yerushalmi. Web caching with consistent hashing. *Computer Networks*, 31(11-16):1203 – 1213, 1999.

[27] Arthur M. Keller and Julie Basu. A predicate-based caching scheme for client-server database architectures. *The VLDB Journal*, 5:35–47, 1996. 10.1007/s007780050014.

[28] Thomas M. Kroeger, Darrell D. E. Long, and Jeffrey C. Mogul. Exploring the bounds of web latency reduction from caching and prefetching. In *Proceedings of the USENIX Symposium on Internet Technologies and Systems on USENIX Symposium on Internet Technologies and Systems*, pages 2–2, Berkeley, CA, USA, 1997. USENIX Association.

[29] Thomas M. Kroeger, Darrell D. E. Long, and Jeffrey C. Mogul. Exploring the bounds of web latency reduction from caching and prefetching. In *Proceedings of the USENIX Symposium on Internet Technologies and Systems on USENIX Symposium on Internet Technologies and Systems*, pages 2–2, Berkeley, CA, USA, 1997. USENIX Association.

[30] L Lamport. Fast paxos. *Distributed Computing*, (19):103, 2006.

[31] L Lamport and M Massa. Cheap paxos. In *In Proceedings of the International Conference on Dependable Systems and Networks (DSN2004*, pages 307–314. IEEE Computer Society, 2004.

[32] Leslie Lamport. Paxos made simple. In *In SIGACT*, 2001.

[33] Leslie Lamport and Keith Marzullo. The part-time parliament. *ACM Transactions on Computer Systems*, 16:133–169, 1998.

[34] Ari Luotonen and Kevin Altis. World-wide web proxies. *Computer Networks and ISDN Systems*, 27(2):147 – 154, 1994. Selected Papers of the First World-Wide Web Conference.

[35] Petar Maymounkov and David Mazires. Kademlia: A peer-to-peer information system based on the xor metric. In *In IPTPS .01: Revised Papers from the First International Workshop on Peer-to-Peer Systems*, pages 53–65. Springer-Verlag, 2002.

[36] Michael A. Olson, Keith Bostic, and Margo Seltzer. Berkeley db. In *ATEC '99: Proceedings of the annual conference on USENIX Annual Technical Conference*, pages 43–43, Berkeley, CA, USA, 1999. USENIX Association.

[37] T O'Reilly. What is web 2.0. o'reilly network. In *September 30, 2005 http://www.oreillynet.com/pub/a/oreilly/tim/news/2005/09/3 0/what-is-web-20.html*.

[38] Venkata N. Padmanabhan and Jeffrey C. Mogul. Using predictive prefetching to improve world wide web latency. *COMPUTER COMMUNICATION REVIEW*, 26:22–36, 1996.

[39] Dean Povey and John Harrison. A distributed internet cache. 1997.

[40] S Shvachko, H Kuang, S Radia, and R Chansler. The hadoop distributed file system. In *Symposium on Mass Storage Systems and Technologies*, pages 1–10, 2010.

[41] Moritz Steiner, Taoufik En-Najjary, and Ernst W. Biersack. A global view of kad. In *Proceedings of the 7th ACM SIGCOMM conference on Internet measurement*, IMC '07, pages 117–122. ACM, 2007.

[42] I Stoica, R Morris, D Karger, M F Kaashoek, and HBalakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *In Proc. of ACM SIGCOMM*, 2001.

[43] Marton Trencseni and Attila Gazso. Keyspace: A consistently replicated, highly-available key-value store, 2009.

[44] Jia Wang. A survey of web caching schemes for the internet. *ACM Computer Communication Review*, 29(5):36–46, 1999.

Resources

Few useful tools used in this work are given below with information on where to get them.

Micro Core Linux It is a minimal Linux distribution, sized under 7 MB, with modified lightweight kernel and Busybox tools.
Availabale at http://distro.ibiblio.org/tinycorelinux/

VirtualBox A opensource virtualization solution, with both GUI and command line control availability.
Available at http://www.virtualbox.org/

Python An interpreted high level programming language which is versatile, easy to use and cross platform.
Available at http://www.python.org/

Cygwin A collection of tools and utilities under windows, proving features and usefulness of Linux. Few tools used in this project are Latex, GnuPlot, Shell Scripting, etc.
Available at http://www.cygwin.com/

ScribTex Online latex editor and compiler, useful because documents can be edited and compiled from *any* internet connected browser, hence any machine that's connected to the internet.
Available at http://www.scribtex.com

Curriculum Vitae

Bivas Das, son of Prodyot Kr. Das and Manju Das was born on 1st February 1986 in Kolkata, India. He completed his higher secondary education in July 2003 under West Bengal Board of Higher Secondary Education. He entered West Bengal University of Technology in the Fall of 2003 and he graduated with a bachelor's degree in Computer Science and Engineering in the Spring of 2007. In July, 2007 he joined a Cognizant, an IT Software Firm as a Programmer Analyst and continued to work there till Dec, 2008. In the spring of 2009 Bivas resumed his studies at The University of Texas at El Paso for his master's degree. While pursuing his master's degree in Computer Science he worked as a Research Assistant under Dr. Eric Freudenthal and Dr. Luc Longpré. Later on he also worked as a Teaching Assistant for the Computer Science department. He also worked in Cierp, Center for Research, Evaluation and Planning department of UTEP, during the last semester of his degree.

Email:
bivas.das@gmail.com
bdas@miners.utep.edu

Permanent Address:
24 Balaram Bose 1st Lane, Bhawanipore
Kolkata, West-Bengal, India. Zip - 700020