Software Design Document

for

Distribured Hashtable

Version x.x approved

Prepared by Daniyal, Palak, Rachana, Andrew Frantsuzov

Instructor: Alfred Nehme

Course: CS481 Capstone

Date: 11/2/2019

Table of Contents

**Revisions**

**1 Introduction**

**1.1 Purpose**

**1.2 Scope**

**1.3 Document Overview**

**1.4 Definitions, Acronyms and Abbreviations**

**1.5 Document Conventions**

**1.6 References and Acknowledgments**

**2 System Overview**

**2.1 Architectural Design**

**2.2 Design Decomposition**

**2.3 Design Rationale**

**3 Data Design**

**3.1** **Data Description**

**3.2 Data Dictionary**

**4 Component Design**

**5 Human Interface Design**

**5.1 Overview of User Interface**

**5.2 Screen Images**

**6 Programming Tools**

Revision History

|  |  |  |  |
| --- | --- | --- | --- |
| **Primary Authors** | **Date** | **Reason For Changes** | **Version** |
| Andrew Frantsuzov | 11/2/2019 | Filled out sections 1.1-1.6, Architectural Design, Design Rationale, Overview of User Interface, Screen Images | First Draft 0.1 |
| Daniyal Adzhiyev | 11/4/2019 | Wrote Sections 2.2.1 |  |
| Palak Sharma | 11/6/2019 | Helped with sections 2.3, 3.1 and 6 | First Draft 0.1 |
| Rachana Mandal | 11/10/2019 | Helped with sections 1.1, 1.2, 1.3 and 6 | First Draft  0.1 |
| Daniyal Adzhiyev | 11/11/2019 | Wrote 2.2.2, Created the Sequence Diagram for 2.2.3, Helped with Section 6, added 3.1.2 |  |

1. INTRODUCTION

1.1 Purpose

The purpose of this software design document is to describe how to architect the Distributed Hashtable (DHT) as well as describing the system design for this DHT. This document is also intended to be a reference for our team when implementing the DHT, allowing us to make updates, refer to the architect and so on.

1.2 Scope

The distributed hashtable will provide a way to store data across several machines. The store will be accessible using an API client. It will be deployed in the cloud as an in-memory data service. The benefit of our project is that it allows data to scale along with the service.

1.3 Overview

This software design document for our DHT is to provide the reader with knowledge on the progress design and architecture plans for this DHT, APIs, and DH Browser. The document is organized in the following manner that follow the Table of Contents which is mentioned in previous page of this document and provide each section including their corresponding pages.

1.4 Reference Material

# References

|  |  |
| --- | --- |
| [1] | J. P. Carzolio, "A Guide to Consistent Hashing," TopTal, 2919. [Online]. Available: https://www.toptal.com/big-data/consistent-hashing. [Accessed 2 11 2019]. |

1.5 Definitions and Acronyms

DHT - Distributed Hashtable

LB – Load Balancer

SDK - Software Development Kits

2. SYSTEM OVERVIEW

A general description of the functionality of our project is that DHT will address the issue of storing massive amounts of data in a hash table. By redistributing the data among the nodes internally it will solve the problem of massive growth. Whenever a node goes down, the data will remain accessible.

2.1 SYSTEM ARCHITECTURE

2.1.1 Architectural Design

The following is the architectural layout of the distributed hashtable. One of the broader goals of this design is its decentralized nature, that is all VMs are equivalent to each other.

Furthermore, on the right-hand side we see a client or a microservice talking to the DHT cluster. We see that the cluster is behind an HLB which monitors health of either node and automatically routes requests only to the healthy nodes. The VMs within the cluster talk to each other and respond to client requests.



If we look at an individual VM, it will have the following layered architecture composed of a frontend, adapter, and backend. The layered architecture will process incoming requests at the front-end layer, as well as provide a probe for the Cluster Health Monitoring. The frontend will also receive, deserialize and authenticate the incoming request. After doing this the frontend will pass the request to the adapter layer, which will check the semantics of the request. The adapter layer, will also run the distributed hash to determine the correct owner VM of the key being requested. If the key owner machine is the current machine, then the Adapter will pass the request and the computed hash value to the backend for final processing. The backend will access the internal DHT (now HT) and finally execute the clients’ request. If there is a response body, the backend will pass it back to the frontend to send it back to the caller.



2.2 Decomposition Description

Provide a decomposition of the subsystems in the architectural design. Supplement with text as needed. You need to give an object-oriented description. For an OO description, put subsystem model, interface specifications, class diagrams, generalization hierarchy diagram(s) (if any), and sequence diagrams here.

2.2.1 Cluster Nodes

The nodes will follow a peer-to-peer communication protocol using the Chord algorithm. Chord will allow for a balance of quick CRUD operations while also causing minimal impact to other nodes when additions and departures of nodes occur. Each node will only be responsible for communicating with log(n) number of other nodes. A query sent to any node in the system will be able to route through to the correct node by contacting the closest peer in the node’s routing table, until it is able to ‘jump’ to the correct node responsible for the key, the look-up cost is also O(logn). The node values and keys will be determined using a consistent hash function, known as SHA-1, where the node’s value will be a hash of its String IP address, and the key will be generated by hashing the String title of the file.

Node’s will need to keep:

* + 1. A Hash Table of <key, value> pairs.
    2. An integer list of all keys that it is responsible for.
    3. A routing table to communicate with other nodes in the network.
    4. The nodeID of its successor, the node directly in front of it (clockwise) on the network.
    5. The nodeID of its predecessor, the node directly behind it (counter clockwise) on the network.

PSUEDO-CODE FOR getValue(key) method within a node:

Hash the Value of the Key

IF the value is in the keyList for the Node

Search in the node’s HashTable to see if the key is there

IF The key, value pair exists

RETURN the Value for the Key

ELSE

Search through the routing table to find the correct node to send the query

Forward the getValue(key) query to the next Node.

REPEAT

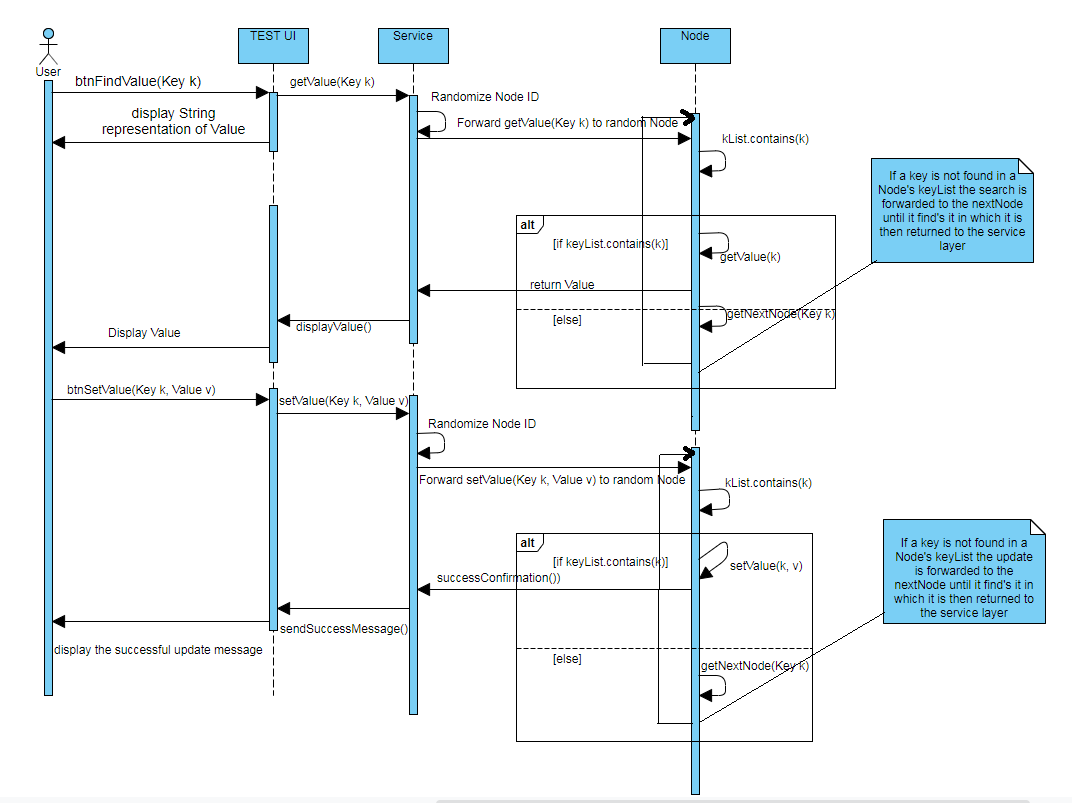
END IF

END IF

2.2.2 Service Layer

The service Layer will act as an intermediate layer between the VM’s Nodes and the front-End UI. It will maintain a connection between all the node’s and upon receiving a CRUD operation through the UI, will randomize which node it will send the query to. It will be responsible for returning the results of the query from the Node’s back to the UI.

2.2.3 Sequence Diagram for Distributed Hash Table



2.3 Design Rationale

One of the major design decisions for this system is to make the system de-centralized, this means that there is no single point of failure. This helps the DHT achieve the goal of being a robust system, as well as simplifying the design. Additionally, this design allows the DHT to scale up, simply by adding more nodes. This also enables the DHT to be reliable. Because the system is decentralized, even if a node goes down, there will be minimal impact on the entire system. Using the chord structure makes querying faster as the time complexity is log(n) where n is number of nodes.

The VM internals are designed to be structured as a layered architecture. This is the best construct for us because we can break the system down into layers. Layers can be organized and tested into smaller chunks of the system. Also, layers can be modified and split amongst multiple developers for work.

3. DATA DESIGN  
3.1 Data Description

3.1.1 Node Data

Data will be stored on the nodes’ Hash Tables as key and value pairs. Then, it is hashed and sent to an appropriate server based on the hashed value. After it can be retrieved by the client from any node.

3.1.2 Backup Data on Amazon DynamoDB

When the Network is offline and not active the project requires stored data in a permanent secured location. We will use noSQL and Amazon DynamoDB to store all data. Upon the network coming online data, as key, value pairs will be populated and stored on the nodes from Amazon DynamoDB.

4. COMPONENT DESIGN

In this section, we take a closer look at what each component does in a more systematic way. As gave an OO description in section 3.2, summarize each object member function for all the objects listed in 3.2 in (Procedural description Language) PDL or pseudocode. Describe any local data when necessary.

5. HUMAN INTERFACE DESIGN

5.1 Overview of User Interface

There will be no primary UI for the DHT since it is an API. However, we will provide a UI app to browse and manipulate the DHT for testing, demonstration, and debugging purposes. In this section we will provide an overview of the DHT UI Client.

**REQ-1: Hashtable to support Get operations**

The get operation will retrieve an existing key from the hashtable. The API should be like

String Get the (string key)

The user passes a key to the API and the return is the value from the hash-table. If the KEY is null, an InvalidKeyExceptionIt is possible for a value to be NULL. Therefore, if the key does not exist in the distributed hashtabke, a KeyNotFoundException should be raised.

Get will work immediately after AddorUpdate was called. Get will return the most recent value that was added or updated, without any delay.

**REQ-2: Hashtable Add and Update operations**

The API for AddorUpdate is:

string AddorUpdate(string key, string value, lambda updateOnConflict)

It will accept a string key and value. The AddOrUpdate function should handle concurrency. In these cases, the same key may be added by two different machines at the same time. When this happens, the API should invoke the updateOnConflict lambda to let the caller update the final value.

The function will return the new value was effectively added.

**REQ-3: Hashtable to support Remove function**

The API for remove is

Bool Remove (string key)

Remove will accept a key as a string. Because Distributed Hashtable is designed for concurrency, will not throw an exception if a key doesn’t exist. Instead, if a key doesn’t exist, Remove will return false if no change was made. If a key was removed, then this method will return true.

REQ-1:

REQ-2:

5.2 Screen Images

The front window of the UI will connect to the “Default” server and download (GET) all the key and values. The UI will allow all the API operations (New/Update, Delete). The user will be able to view all the keys in a specific node, or view all of the nodes simultaneously. Getting all the keys should not freeze the UI as it may take some time to complete. As keys are received from each node the UI should update.  
  
Once all of the keys are downloaded, only the selected values will be downloaded.

The UI will refresh every 10 seconds idle to update any keys, and/or nodes that were added/removed.

A screenshot of a social media post

Description automatically generated

In the UI will need to be able to select a server to connect to. This will be done by the Connection Tab. The connection Tab will allow the user to view already configured connections, and add/remove server connections, as well as setup the default startup connection. The connect button will initiate the GET operation from the currently selected server. The user may switch back to the main tab to view the data inside the selected server.

The UI will intentionally supper the local server, as one of the main scenarios will be to test the local service. In this case the UI will simulate a load balancer to connect to a random instance of the service.

A screenshot of a social media post

Description automatically generated

6. Programming Tools

The programming tools that we will use are as follows:

1. Java: This is a programming language that is commonly used, better security and many development tools are available for Java. Java is well known industry language and it is simpler if you know object-oriented programming principle.
2. AWS for the cloud: It is a cloud computing services that we will be using as developer tools. We will use AWS command line tools and Software Development Kits (SDK) to deploy and manage applications and services.
3. Eclipse: We will use Eclipse to write our Java Programs such as the UI Test Application, Service Application, and Backend for the node’s.
4. Eclipse:
5. Eclipse WindowBuilder: We will use this GUI Builder to help create the visual design for the Test UI.
6. Amazon DynamoDB: We will use DynamoDB to save all data that we will use to populate the HashTables with Data.