Distributed Cache System for Node/Angular JS

Final Project Report

Qingjun Wu, Benjamin Cordova
University of Colorado, Boulder

Abstract

As of now, there are no Distributed Cache Systems available in the Node.js community. Our proposal is such a system based on new web technologies including Node.js, AngularJS, REST, and Bootstrap. Our system is implemented using the CHORD strategy. It consists of three, independent modules: multiple backend Cache Servers, a single Load Balancer and a front-end AngularJS Client Application.

1. Introduction

NodeJs and AngularJs are very popular in web client/server development. No distributed cache libraries for these tools exist in the open source community. The goal of our project is to implement a distributed system solution to resolve this problem. Our application provides a key/value store service and is implemented using NodeJs and AngularJs technologies. Our system consists of three modules: A Cache Node, a Load Balancer and a Client Application.

2. System Design

The multiple Cache Server nodes provide the underlying key/value store service and serve the Client Application. The Load Balancer tracks the Cache Servers and forwards proxy requests from the Client Application instances to Cache Server nodes. The Cache Nodes and Load Balancer are implemented in Node.js and provide various RESTful/JSON services. The system is depicted in Figure 1.

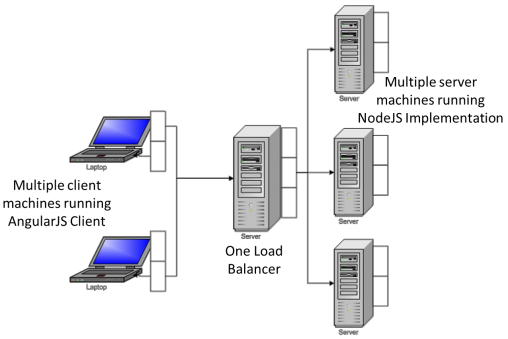

Figure 1. NodeAngularDistributedCache System Diagram

2.1. Load Balancer

The Load Balancer (or Proxy) is a NodeJS HTTP server which serves as a single access point into the system. The write requests are forwarded to a Cache Server based on a hash of the incoming key. Details on the hashing implementation will follow in §6. Distributed Hashing. Reads from the system cause the Load Balancer to query the proper server for the key. The result is then forwarded back to the AngularJS client application.

URI	GET	PUT	POST	DELETE
/servers	List details about the Cache Servers.	Not Supported	Not Supported	Not Supported
/data/<key>	Request forwarded to Cache Server based on <key>			

Figure 2. Load Balancer API Matrix

The Load Balancer also hosts an IPC (Inter Process Communication) Server in addition to the HTTP server. This interface is intended for system-level communication between the Load Balancer and the various Cache Servers. IPC Server provides two functions to the Cache Servers: registration and heartbeat. The server registration callback allows a quick add of a new server to the known servers data structure. This registration function calculates the hash address of a new server in the key ring and stores forwarding connection information of the server in a **`cachedServers`** array. The heartbeat function allows cache servers to check if the Load Balancer is still up and running. The current heartbeat is set to 30 seconds. The timestamp of the most recent heartbeat is stored on the Load Balancer for each Cache Server.

2.2. Cache Server

The Cache Server is setup as a simple GET/PUT/DELETE datastore. It takes the address and port of a Load Balancer as an initialization input. First thing it does is register with the Load Balancer.

URI	GET	PUT	POST	DELETE
/	Serve the AngularJS Client Application	Not Supported	Not Supported	Not Supported
/data/<key>	Retrieve <value> for <key> from local system. If <key> does not exist in the system, return a 404.	Store or replace <key> and its <value> in the local system.	Not Supported	Delete <key> and its <value> from the system.

Figure 3. Cache Server API Matrix

All RESTful responses are returned in a JSON format.

2.3. Client Application

The Client Application is served out of the "/" URI (Uniform Resource Identifier) on each of the Cache Servers. This in-browser application provides a graphical user interface to the caching system. It consists of two simple forms: one which allows the user to store key/value pairs into the system and one which retrieves data from the system. There is also a table of all key/value pairs in the system at page load time. The final panel contains a real time status of the last heartbeat contact from each Cache Server to the Load Balancer.

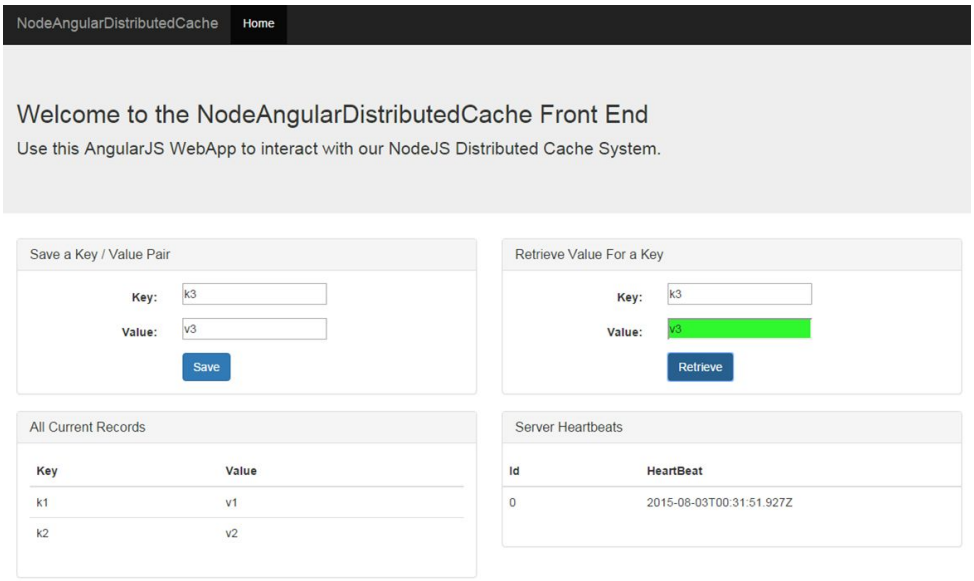

Figure 4. The AngularJS Client Application GUI

We developed this client application using the AngularJS framework plus the Bootstrap styling infrastructure published by Twitter.

2.3.1. AngularJS

AngularJS is an open javascript library which joins placeholder html templating with on-page javascript logic. This combination allows the developer to rapidly create rich, dynamic and functional web pages. Inline "ng" declarations are processed in real time on the page even without javascript driving the logic. The developer can add an "ng-model" attribute to any html tag and then use that to reference the tag's contents somewhere else on the page via a double-curly-brace placeholder, e.g., "{{placeholder}}." In this example the developer could set the "ng-model" attribute of a input-text tag equal to "placeholder" and AngularJS would connect the two. Any updates to the value of the input-text tag are instantly reflected at the placeholder location. The previous example was purely html. The other half of AngularJS is the dynamic control it provides via javascript. This javascript integration allows placeholder manipulation via an event driven framework.

Our implementation of the Client Application utilizes three main features of the AngularJS library: the "ng-model" attribute, event handlers and the "ngAnimate" library for animation. Both forms on the page contain "ng-model" attributes on their various input-text tags. Each button click drives an event in our AngularJS javascript. Clicking the "Save" or "Retrieve" buttons in the GUI causes a HTTP request to the Load Balancer based on the content AngularJS pulled out of the input fields. This is not a standard html form transaction. Our app then receives the JSON response from the Cache Server and populates proper AngularJS DOM items. The "ngAnimate" library is used to temporarily color the "Value" input-text tag of the "Retrieve" form upon successful JSON interaction. The box is colored green on successful key retrieve and red if the key doesn't exist in the cache system.

2.3.2. Bootstrap

We choose to use Twitter's Bootstrap library to handle the look and feel of the client application because AngularJS doesn't directly offer complex CSS styling. The Bootstrap library adds a much needed translation layer which separates logical objects from the browser's rendering of html and css. The ideology is based on dividing a page up into pancake style, stacked rows. Each row has twelve logical columns. Every component therecontained is some twelfth fraction of a row. In our implementation, the tiling starts with two rows, each containing two columns of width-6 units. Each of the width-6 columns is a Bootstrap "Panel" which appears as a visually self contained box with a title. From there, in each width-6 column, there is a single row which again provides a logical-twelve-column space into which we can place objects. We placed the two forms, which are also Bootstrap visual, logical constructs, in the first local row on the page. The two data tables are in the two Panels in the second row. While this may seem complicated for a simple webpage, the strongest feature of Bootstrap is that each tile (a row-column logical object) on the page flows appropriately as the browser window is resized and is rendered correctly by an internal algorithm on mobile devices. We also took advantage of Bootstrap's built in "navbar" and "jumbotron" constructs instead of coding our own navigation and header sections.

2.3.3. NodeJS

Node.js is open-source, cross-platform framework which uses javascript to build web servers. It runs on top of Google javascript engine V8. Node.js supports many different network protocols, including TCP, UDP, and TLS/SSL, plus various data stream and I/O operations. Node.js is single threaded, its architecture is event-driven and provides non-block I/O operations. Node.js is a relatively new technology. It was originally created in 2009, but it has since grown a large community following. Companies like Microsoft have created and published their own versions of Node.js.

In our implementation, Node.js was used to create both the Load Balancer and Cache Server HTTP servers plus the TCP/IP server. One application can host both HTTP server and TCP/IP server on different ports. The role of HTTP server is used for client request handling; while TCP/IP server is used for Inter-Process Communication.

3. The LRU Cache

The LRU (Least-Recently-Used) Cache feature is the second iteration of our "Global Cache." In the first iteration, key/value pairs were stored internally in a simple javascript array on the Cache Servers.
Our LRU is implemented using a hybrid of doubly linked list and hash table. The elements in the linked list are each key/value pair. The most recently active element is always moved to the tail of the list. If number of elements in the list has reached the capacity of the linked list, the element at the head of the linked list will be removed. The hash table is used to locate a searched item easily. In the <key, value> table, the key is the key that is being searched, the value is the address (reference) to the node in the linked list. The hybrid of doubly linked list plus hash table allows both push() and get() access to the structure in constant time O(1). In terms of memory space, the structure is is linear O(n) where n is the number of keys in the list/hash table.

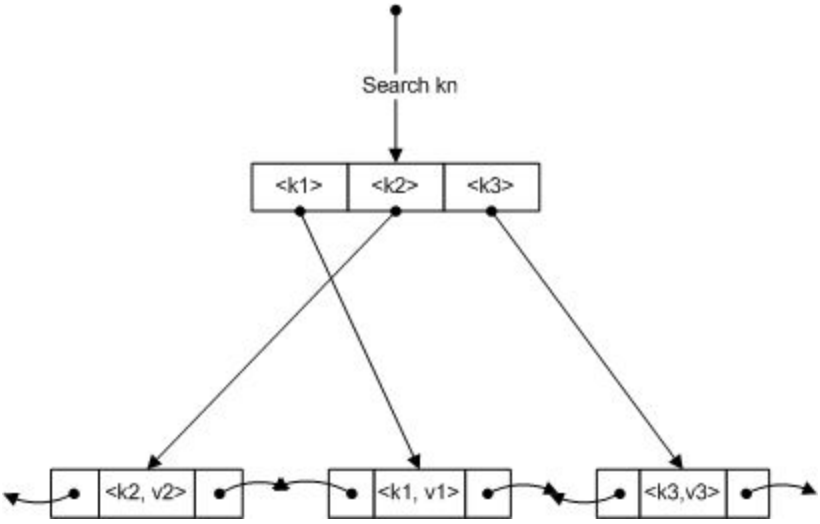

Figure 5. The Least Recently Used Cache

Concerning thread safety, the LRU global cache object maintains an "`Executing`" attribute. Before acting, the first thing any accessor method does is check this boolean.

```
var LRUCache = function(capacity) {
    this.Objectes = new Array();
    this.Size = capacity;
    this.Count = 0;
    this.Executing = false;
    this.dll = new Dll();
};
```

Figure 6. The LRU Object

4. Inter-Process Communication

TCP/IP sockets are used for inter-process communication. Messages are structured into JSON objects to send over the network. The structure of the messages is shown in Figure 7.

```
message: {
      command: 'list',
      server: 'localhost',
      ipcport: 8225
      }
```

Figure 7: JSON object structure

Two javascript classes IpcClient and IpcServer are implemented to server Inter-Process communication. Either cache server or proxy they could be both Ipc client or server, and each of them are listening to TCP/IP port when they work as Ipc server.

A series of commands are defined for communication purposes:

- list: list all the key-value pairs in the buffers of all the server. This is used to test purpose to validate the system.
- registerserver: register the server into the proxy- which is also server manager. This event happens when a cache server is just created and expecting to register itself to the proxy.
- heartbeat: heartbeat message. For every fixed period of time, the cache server will send a heartbeat message to the proxy, the latter will update last heartbeat of the server. Based on the heartbeat, the proxy will be able to detect if a server is failed.

Originally we applied RESTful API to communicate the proxy/servers with each other, but since these servers (including proxy) are located in the same domain, a IPC communication is more efficient, and more importantly, it is more secure since RESTful API will be exposed to public.

6. Distributed Hashing

Distributed hash tables are used to map the key and servers into a virtual ring, as shown Figure 8. Each server owns the fraction of keys in the counter-clock direction. As Cache servers register with the Load Balancer, it gets assigned an address by calculating a SHA1 hash on the connection parameters of the Cache server. This hash is then used to generate an index into the list of registered servers.

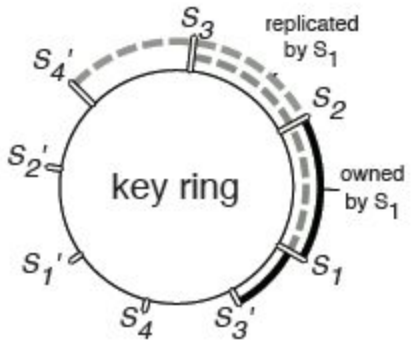

Figure 8: Key Server Virtual Ring

When the user needs to access a key, the AngularJS Client Application creates a REST HTTP GET request, in the form of /data/:key, to the Load Balancer. The Load Balancer then calculates a SHA1 hash on the incoming key to figure out which Cache server owns that key. Once the Load Balancer determines the owner of the key, it forwards the HTTP GET request onto the owner via HTTP Redirect. From there, the Cache server is responsible for handling the remainder of the transaction with the Client Application. This implementation is based on the CHORD design [5,7].

7. Heartbeat

In this system, the Cache Servers send heartbeat messages to the Load Balancer server. The Load Balancer server keeps a local table of the last update from each Cache server. By doing this, it knows if any of the servers is down. Upon receiving a heartbeat message, the Load Balancer updates the Cache Server table with the new timestamp for this Cache Server. The Load Balancer also checks this table every other N seconds. If it detects that any server has not been heard from for a predefined interval of time, it assumes the server is down.

8. Evaluation

All three modules in the system have been successfully prototyped. They interact with each other as intended. The Cache Servers appropriately save and retrieve keys. The Load Balancer is successfully serving the AngularJS Client Application and forwarding requests. We have test scripts which test the API at both the Cache Node level and at the Load Balancer level. We've prototyped the AngularJS Client Application web framework. Figures 9 and 10 depict an example execution of the system.

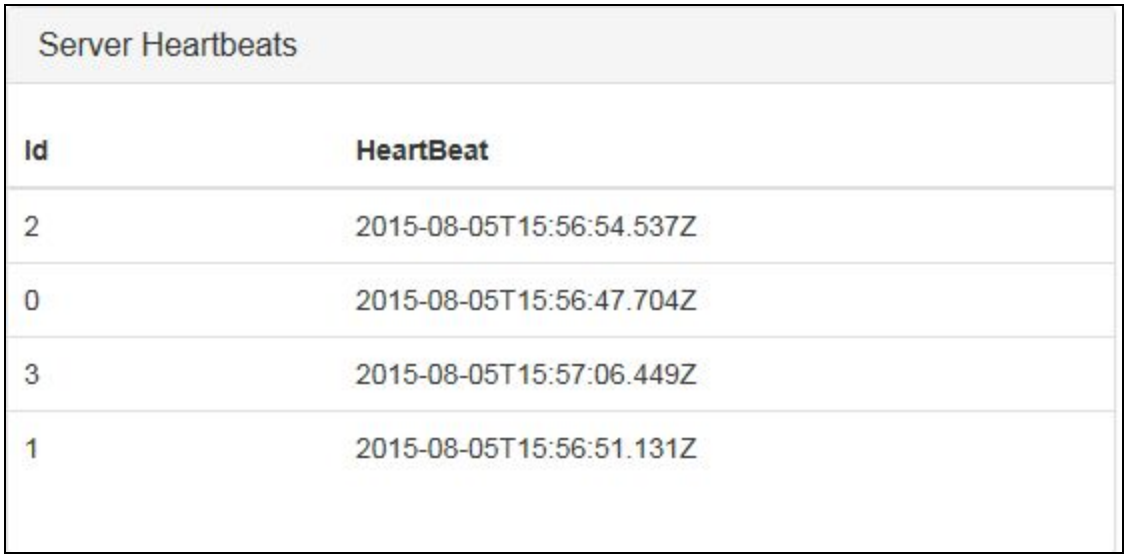

Figure 9. Launch 1 proxy + 4 cache servers, heartbeating from the servers:

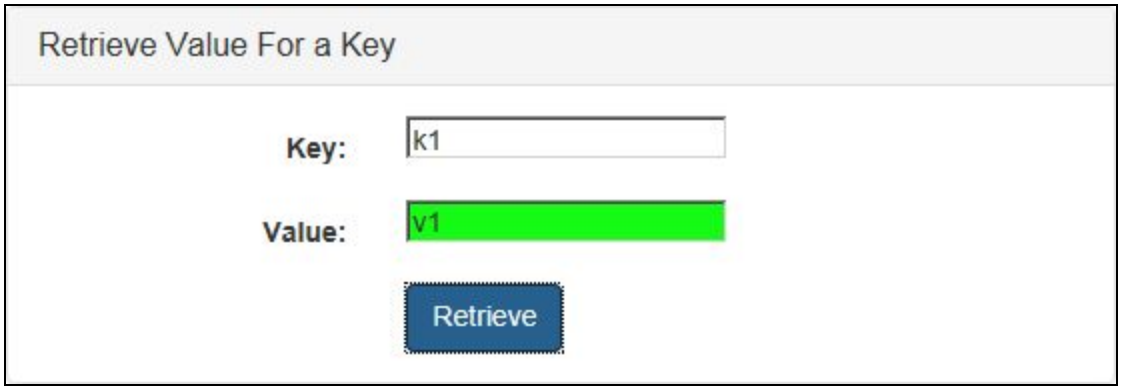

Figure 10. Save a datum <k1,v1>, and click retrieve

9. GitHub

We have created a public GitHub repository located at http://github.com/wuqingjun/NodeAngularDistributedCache. All team members worked on code in the master branch at this location. A transcript of the commit history can be found in Reference [6].

10. Conclusion and Future Work

Our NodeAngularDistributedCache solution has the potential to serve a niche need in the fast growing Node.js community. Specifically, there are no modules we could find that yet serve the purpose of providing a distributed, CHORD-like cache system. Our system is mature enough that it can support multiple, geographically separated Cache nodes with a single Load Balancer. Another asset of this system is that it is already divided along natural Model-View-Controller lines. Because the AngularJS client application communicates to the Load Balancer via simple RESTful API calls, any equivalent front end

application could easily replace it. There's nothing about our AngularJS Client specifically provides to work with the backend parts of the system. This implies that one could write their own GUI or tool interface into our system.

The limitations of the system appropriately scale with the hardware capabilities of the member systems. A large, production-level implementation of this system would most likely consist of a highly network capable Load Balancer connected to many Cache Systems; "many" in this case means tens of thousands of systems at the high end. The Cache Systems themselves need to each have large memory available to store the data. The AngularJS Client application is lightweight enough that it's not going break any modern browsers. It also does not use any components that need to vary by browser. Likewise, the Bootstrap CSS styling engine has good cross browser support.

There are many components of the system that could be improved or expanded with new features. The highest priority "next logical step" would be to implement CHORD's Finger Table function to optimize finding keys within the ring. This would allow Cache Servers to more efficiently figure out the location of keys. From there, with this additional knowledge about their peers, the system could be optimized by improving the handling of data when Cache Servers are added or removed, intentionally or otherwise from the system. Currently, keys are only stored on one Cache Server in the system. Key redundancy would be a natural expansion.

An optimization of the LRU cache would be to subdivide it into "In Memory" and "Pushed to Disk" parts. The doubly linked list would need to store a finite amount of data in RAM and, only when the searching of that object has been exhausted, would the LRU attempt to pull a key's value from disk.

One the above features have been implemented, we recommend improving the Load Balancer to proxy the key requests to a geographically convenient server for the AngularJS Client application. That way, a user wouldn't have to traverse a long network distance if a key happened to be stored on a Cache Server that is far away. With this geographic expansion, one could modify the system to support multiple, local Load Balancer nodes as opposed to the single instance in the current implementation. This would resolve an outstanding issue with the current implementation that the Load Balancer is a single point of failure.

Another eventual, necessary expansion of the system would be writing multiple front-end APIs to co-exists with our AngularJS Client Application. We're looking for interfaces to support multiple programing languages and situations. One could imagine libraries for python, Java, PHP, Ruby, etc. Basically, anything that needs to store large amounts of data.

11. References

[1] "AngularJS — Superheroic JavaScript MVW Framework." Super-powered by Google ©2010-2015 Accessed June 24, 2015, from https://angularjs.org/.

[2] "API Guide | restify documentation." Mark Cavage. 2012. Accessed June 28, 2015 from http://mcavage.me/node-restify/.

[3] "Applications Programming in Smalltalk-80: How to use Model–View–Controller (MVC)" Burbeck, Steve. 1992.

[4] "Bootstrap." Designed and built with all the love in the world by @mdo and @fat. Code licensed under Apache License v2.0, documentation under CC BY 3.0. Glyphicons Free licensed under CC BY 3.0. Accessed July 6, 2015, from http://getbootstrap.com/2.3.2/.

[5] "Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications", SIGCOMM 2001, Stoica, I. and Morris, R. and Karger, D. and Kaashoek, M.F. and Balakrishnan, H. Accessed July, 15, 2015, from http://0-dl.acm.org.libraries.colorado.edu/citation.cfm?id=383059.383071&coll=DL&dl=ACM&CFID=534257868&CFTOKEN=15884273.

[6] "Commits · wuqingjun/NodeAngularDistributedCache." Q. Wu, B. Cordova. Accessed August 4, 2015, from https://github.com/wuqingjun/NodeAngularDistributedCache/commits/master.

[7] "CSCI 5673, Distributed Systems, Lecture Set Tselve (sic), Peer to Peer Systems, DHTs: Chord, CAN, Tapestry." Shivakant Mishra. September, 30, 2013. Accessed July 15, 2015, from https://moodle.cs.colorado.edu/pluginfile.php/12010/mod_folder/content/0/LectureSetTwelve%20-%20p2p-DHTs.pdf?forcedownload=1.

[8] "Principled design of the modern Web architecture." Fielding, R. T.; Taylor, R. N. 2000. pp. 407–416.

[9] "Node.js." © 2015 Node.js Foundation and the Linux Foundation. Accessed June 24, 2015 from https://nodejs.org/.

[10] "Node.js Is Big, And Still Getting Bigger." Michael Singer, for readwrite.com. © 2015 Wearable World Inc. Accessed August 4, 2015, from http://readwrite.com/2013/12/07/nodejs-os-infrastructure.

[11] "nodejitsu/node-http-proxy." nodejitsu. Jul 9, 2015. Accessed July 16, 2015 from https://github.com/nodejitsu/node-http-proxy.