CS87 Project Proposal: Distributed Hash Table

Joon Sung Park and Jacob Carstenson
Professor Tia Newhall
Friday March 25, 2016

For our course project, we propose a novel implementation of a distributed hash table. The hash table algorithm that will serve as the backbone of our proposed implementation takes a similar approach in organizing its elements as that of extensible hash table algorithm, but maintains a greater degree of orderliness through its usage of binary search tree data structure. We believe our implementation will be scalable and efficient, and through our study, we hope to bring some interesting perspectives to the topic of designing a distributed hash table.

1. Background

One problem with a conventional hash table is in its process of resizing; when resizing the hash table, all the elements in the hash table has to be rehashed to fit the increased number of buckets, making the resizing computationally expensive. This makes the hash table's performance less predictable as the end user cannot predict when the expensive rehashing will occur, and ultimately makes the hash table less suitable for dynamic tasks. To answer this problem, a class of dynamic hash tables, such as linear hashing or extensible hash table, has been proposed and implemented.

The hash table algorithm that we will be focusing on in this project, which was designed by one of the authors last year, also tackles the same challenge as the dynamic hash table. Its approach is similar to that of the extensible hash table; all elements are hashed into a unique integer, and the resulting keys are put into a special type of binary search tree with a set upper and lower bounds whose child nodes compare not to the parent nodes, but to an abstract dividing

rod that divides all possible inputs into half at each level[4] (for more detailed description of the algorithm, please refer to the bibliography). This algorithm, like some of the dynamic hash tables, avoids the costly rehashing process during resizing, and grants a much more predictable and dynamic performance than a conventional hash table.

We believe this hash table algorithm's clean and efficient organization of data makes the hash table potentially very distributable. As mentioned, when an element is hashed in our current algorithm, the resulting key is a globally unique integer. The key and value pair is then placed into a binary tree whose upper and lower bounds contain the hashed key. These trees can easily be treated as a packet and planted into different nodes. For example, if a node contains a binary tree representing the values from 100 to 200, and the hashed key is an integer value 130, the key value pair should be placed in the node's tree that represents the values from 100 to 200.

Because the keys are globally unique across the distributed hash table, there is no further communication required between the participating nodes to figure out where the key value pair should be placed, as long as we can locate the tree that is representing the right range of values. This of course means that we need an efficient way to locate the right tree, but as long as we can locate the tree, the distributed hash table can function efficiently.

2. Related Work and Our Approach

Locating the right node is doubtlessly a main point of study in any distributed hash table and much work has been done in this regard. For example, one of the original distributed hash table implementation, Chord, makes a clever use of an identifier circle to find the successor and predecessor node in which to locate the newly inputted element[3]. On the other hand, there are algorithms that maintain a sort of routing table to keep track of the participating nodes; Tapestry,

for example, assigns a unique nodeID to each node that is uniformly distributed in a larger identifier space[1][2].

We are interested in, and are looking towards employing, the approach of maintaining a routing table for our project. Using a routing table to find the appropriate node in a distributed system is a common approach that has been successful in many distributed hash tables, and we believe it will provide a robust enough structure to our current implementation. While we are still trying to layout the details of the implementation, here are some of the directions we are trying to take:

Each participating node maintains its own routing table. The routing table should contain the IP addresses of the nodes that are participating, and the integer range of the key values each node is contributing to the distributed hash table. This much information will be enough for each node to direct the input values to the right location in the distributed hash table. In case the routing table needs to be updated, we need to find a way to propagate the updated table efficiently. For this, we are currently considering the usage of a supernode that first updates the table, and sends the updated version to other nodes. This point is still in discussion.

Finally, in this project, we will not be covering node failures. While node failures and ways to deal with them is an important topic for a robust distributed system, we believe this point may be beyond what is manageable in a month long project.

3. Experiments

We will be comparing our hash table implementation to a more conventional hash table (much like the one we developed in CS35). We will test the performance frequently, at each step of the project. As for the specifics of the experiments, we expect them to resemble what we did

in CS35. That is, we are planning to build an application interface that would be using our hash tables, performing large number of insert, delete, lookup, and other general data operations. When the data inputted into the hash table is sufficiently large, we expect to see boosts in performance as we distribute the hash table over multiple nodes.

4. Equipment Needed

Given the short time we have, we are planning to use Python to program our hash table as well as the simple hash table we will use for testing. We are considering using the Twisted library to make the networking part of this project easier. For the distributed aspect of the hash table, we will be using MPI for communications between the nodes. For testing, we will first use one of the lab machines in the Swarthmore computer science department for testing our hash table design on one node, then we will expand the testing to include multiple nodes, using several of the lab machines.

5. Schedule

A rough schedule we have decided on takes place over a five week period. It involves first making a simple sequential hash table used for testing purposes, which will take less than a week. After this we plan on designing our new, smarter, hash table that is not distributed yet. This will take roughly one week. Then we will spend one to two weeks designing the distributed aspect of the hash table. By week four, we will have spent a week testing this using one node, then gradually growing to test with multiple nodes. For each of these steps, testing will be done with the sequential hash table as a base standard in order to make sure the hash table we design is correct and ideally performs better than the rudimentary sequential version.

6. Conclusions

The field of distributed hash table is a rich one; many studies have gone through, and there is no shortage of existing algorithms that are well exploited. However, it is also a tremendously interesting and important topic of study. We hope that our perspective and effort may bring some new insights into the growing body of distributed hash table studies.

Annotated Bibliography

[1] Rowstron, Antony, and Peter Druschel. “Pastry: Scalable, Decentralized Object Location, and Routing for Large-Scale Peer-to-Peer Systems.”Middleware 2001 Lecture Notes in Computer Science, 2001, 329-50. Doi:10.1007/3-540-45518-3_18.

There are four main distributed hash table algorithms that we came across in our preliminary research: Chord, CAN, Tapestry, and Pastry. After a huge spike of interest in peer-to-peer distributed hash table in the aftermath of Napster, these four algorithms are some of the early results of the field of distributed hash table. These algorithms are still very influential today, and became important guide for us in understanding the popular makings of distributed hash tables. Pastry, which is introduced in this article, utilizes routing table to direct the key value pairs to their rightful location. Pastry and Tapestry that is similar in design as Pastry were particularly helpful to us in understanding how routing tables are generally used as this is the direction we are considering as well.

[2] Zhao, B.y., L. Huang, J. Stribling, S.c. Rhea, A.d. Joseph, and J.d. Kubiatowicz. “Tapestry: A Resilient Global-Scale Overlay for Service Deployment.” *IEEE J. Select. Areas Commun. IEEE Journal on Selected Areas in Communications* 22, no. 1 (2004): 41-53. Doi:10.1109/jsac.2003.818784

This article introduces Tapestry, the distributed hash table algorithm that was mentioned right above.

[3] Stoica, Ion, Robert Morris, David Krager, M. Frans Kaashoek, and Hari Balakrishnan. “Chord.” *Proceedings of the 2001 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications - SIGCOMM ‘01,* 2001. Doi:10.1145/383059.383071

This article introduces Chord. Chord takes an interesting approach to direct the key value pairs to the right node, utilizing an identifier circle to efficiently distribute the inputs to all nodes in the system. While not as closely related to our current direction as Pastry and Tapestry, this article provided some general insights into measuring efficiency of distributed hash table algorithms.

(Only for additional information) [4] Park, Joon Sung. “A New Algorithm for a Chained Hash Table with a Partition Driven Binary Search Tree.” http://joonsungpark.com/Joon_Sung_Park_HashTable_Design.pdf 2015.

This is the study one of the authors of this proposal did during late 2015. The hash table algorithm developed in this study is the basis of our current project. Please use the link if further description of our basis hash table algorithm is needed.