Fall 12-2013

An Analysis of Peer-to-Peer Distributed Hash Algorithms in Improving Fault Tolerance in the Hadoop Running Environment

Benjamin R. Knaus
University of Southern Mississippi

The University of Southern Mississippi

An Analysis of Peer-to-Peer Distributed Hash Algorithms in Improving Fault Tolerance in the Hadoop Running Environment

by

Benjamin Knaus

A Thesis

Submitted to the Honors College of

The University of Southern Mississippi

in Partial Fulfillment

of the Requirements for the Degree of

Bachelor of Science

in the Department of Information Technology

December 2013

Approved by

_____

Shaoen Wu

Department of Information Technology

_____

Joe Zhang, Director

School of Computing

_____

David R. Davies, Dean

Honors College

Abstract

Cloud computing is a “new frontier” in the world of computing. One of the cloud architectures widely used is the Hadoop running environment. Hadoop consists of many parts—including MapReduce, TaskTrackers, and JobTrackers. Right now, there is no fault-tolerance for JobTrackers in Hadoop. This paper analyzes four different distributed hash algorithms (Pastry, Tapestry, CAN, and Chord) that could be implemented inside Hadoop to improve JobTracker fault-tolerance. We recommend Chord as the best suited for integration and improvement of Hadoop.

Key Terms

Cloud computing, Hadoop, peer-to-peer, distributed hash algorithms, fault tolerance, Chord

Table Of Contents

Chapter 1—Introduction

Cloud computing is a model for enabling ubiquitous, convenient, and on-demand network access to a shared pool of configurable, computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction. One way that cloud computing is able to scale up and down easily is through a programming model called MapReduce. MapReduce has a master computer that receives the task which then breaks the task up and delegates (maps) the task to other computers. The master also assigns other computers to merge the pieces of completed computations of the task from the mappers back into bigger pieces (and eventually into the completed task). These computers are called reducers.

MapReduce is highly scalable and fault-tolerant. This means that it can work on large tasks that require large amounts of mappers and reducers just as easily as it can work on small tasks that only require a few computers. It also means that if a mapper or reducer fails, then the master can still complete the task by re-delegating the task of the failed computer to another computer. However, there still remains one major point of failure in the MapReduce architecture—the master. If the master fails, the task will not be able to be completed because there is currently no redundancy for the master, or JobTracker.

What this paper will do is analyze peer-to-peer distributed hash algorithms that can be used to create fault-tolerance for the JobTracker in MapReduce, and recommend one that would be best suited for the Hadoop environment. This will, hopefully, allow for

multiple JobTrackers to run on one MapReduce task, or at least be aware that it exists. So, in the event that one JobTracker goes down, there will be at least one more JobTracker still available and running the same task that can take over for the failed JobTracker.

The outline is as follows: First, a discussion of the background to the project—specifically, cloud computing, the Hadoop running environment, MapReduce, TaskTrackers, and JobTrackers. Second, an analysis of the four main distributed hash algorithms—Pastry, Tapestry, CAN, and Chord. Finally the conclusion as to which distributed hash algorithm would be best suited for integrating with Hadoop.

Chapter 2—Literature Review

According to the National Institute of Science and Technology, cloud computing can be defined as "a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction." [6] A cloud is typically a large group of commodity computers networked together that can be used to complete tasks, or jobs, that can be controlled remotely [1]. Most clouds are running virtualization software where the computation actually occurs [3].

Hadoop is essentially a fault tolerant "massively scalable queryable store and archive" [1] that includes a file system, queryable databases, archival store, and flexible schema [1]. Hadoop is also open source [4]. Hadoop can, and normally does, utilize the Hadoop Distributed File System (HDFS) that uses the "write once, read many" philosophy—meaning that instead of the hard disk constantly having to seek to modify data throughout the set, any new data is appended to the end of the current dataset [3].

MapReduce is the programming model that Hadoop uses to process data [10]. The datasets processed in Hadoop can be, and often are, much larger than any one computer can ever process [4]. MapReduce organizes how that data is processed over many computers (anywhere from a few to a few thousand) [4]. MapReduce also hides system-level details from the programmer—allowing the programmer to focus on what needs to happen instead of how it's going to happen [4].

MapReduce brings the programming code to the data instead of moving the data to the code. This decreases network traffic both in and to the cloud [4]. MapReduce then breaks the data down into value pairs [4]. The mapper inputs the data from the distributed file system, and computes the data to intermediate value pairs. The reducer then computes and aggregates the intermediate value pairs to create output value pairs.

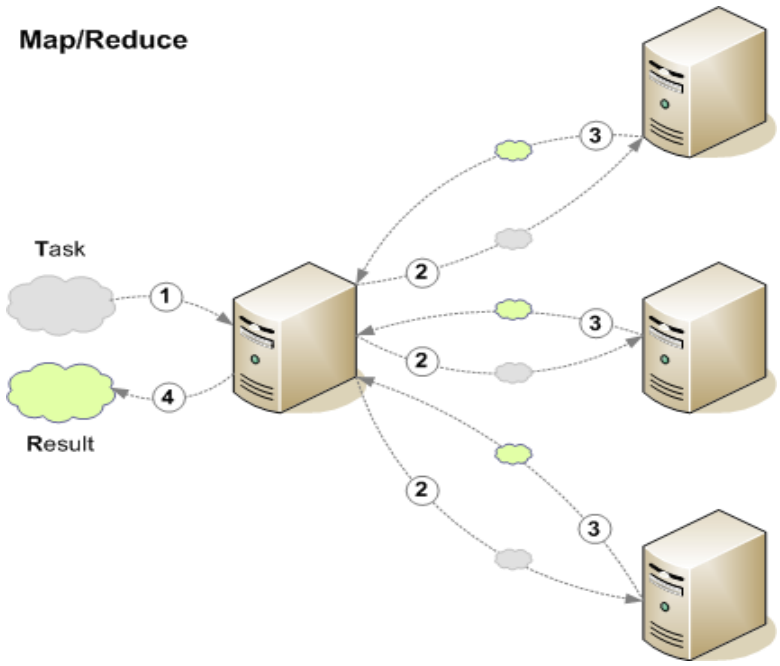

Figure 1: A simple version of MapReduce. Step 1 indicates the initialization of the JobTracker. Step 2 indicates the distribution of tasks to mappers. Step 3 indicates the intermediate results getting returned to the JobTracker. Step 4 indicates the finished result being delivered to the user. [5]

As seen in Figure 1, the MapReduce job begins when the job is submitted and the JobTracker (the central node in Figure 1) is initialized. The JobTracker then breaks down the job into several tasks, and gives those tasks to other machines called mappers. Once the mappers get a task, they then send regular updates about their progress, called heartbeats, back to the JobTracker. The process of heartbeating can be seen in Figure 2.

After the mapper is finished with the task, it then sends it back to the JobTracker which then initializes a reducer machine to aggregate and process the intermediate results provided by the mappers. The reducer also sends heartbeats back to the JobTracker until it is finished with the task. Then, the reduced task is sent back to the JobTracker which will either send it to other reducers to aggregate more data into the task or send it back to the user as the finished result. Figure 3 provides a more detailed view of how Hadoop runs a MapReduce job.

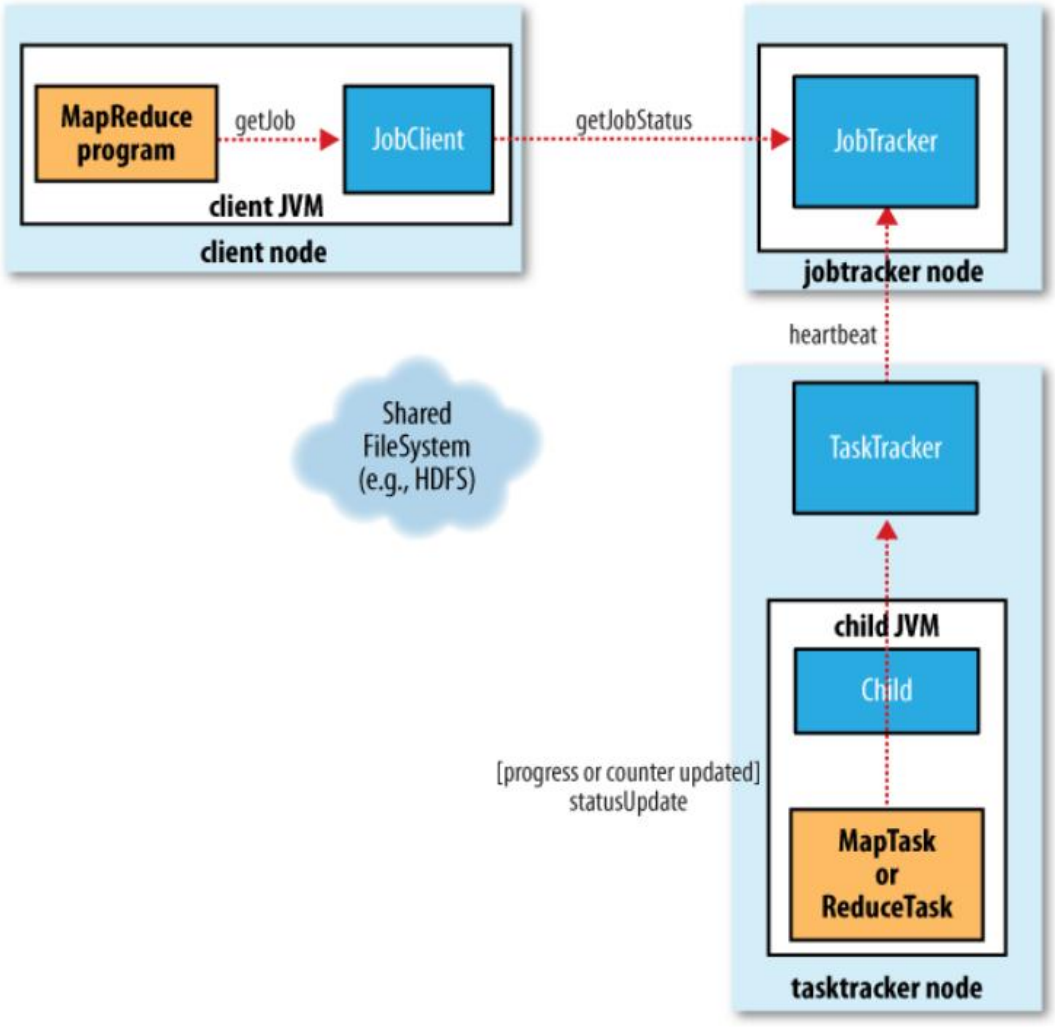

Figure 2: How status updates, or heartbeats, are propagated though the MapReduce system [10]

As seen here in Figure 2, the node (computer instance) that is running a mapper or reducer is also running a TaskTracker. The TaskTracker's job essentially is to supervise the operation of the local mapper or reducer, and to give the JobTracker status updates or heartbeats [10]. If a JobTracker stops receiving heartbeats from a TaskTracker, or if the heartbeats are slow or irregular, then the JobTracker will schedule the same task on another JobTracker in case the first fails [9].

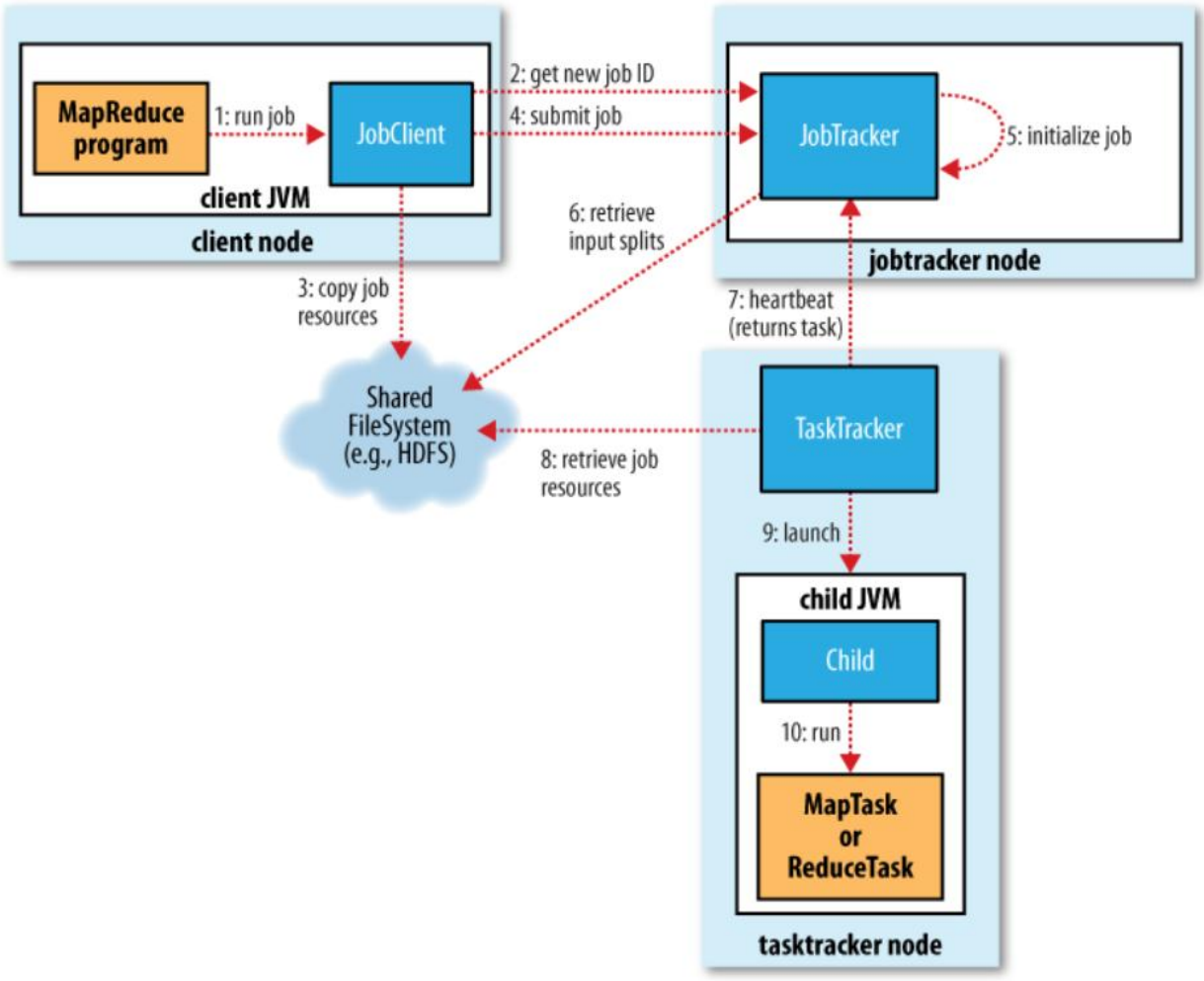

Figure 3: A more detailed view of how Hadoop runs a MapReduce job [10]

As seen in Figure 3, the MapReduce program issues a command to run a job to the JobClient. Then the JobClient issues a "get new job ID" command to the JobTracker, copies the job resources into the cloud distributed file system, and submits the job to the JobTracker. At this point the JobTracker initializes the job, retrieves the input splits of the job resources, and assigns various TaskTrackers to map or reduce the job. The TaskTrackers then begin to send heartbeats back to the JobTracker, retrieve the job resources needed, and launch a child locally. The child then runs the map or reduce task until it is completed. [10] The data is not deleted once the local map or reduce tasks are complete. Instead, they are deleted when the finished job is given back to the JobClient. [10]

There is fault-tolerance in almost all areas of the Hadoop cloud environment. If a mapper fails or runs slowly, the JobTracker will activate another mapper instance to complete the same task. If a reducer fails, then the completed mapper tasks are sent to another reducer that will then process the mapper results. However, if a JobTracker fails, there is no redundancy. The entire job is lost if the JobTracker fails. [9] Since most cloud platforms are running on commodity hardware, a JobTracker failure, although not common, is certainly a possibility [3]. To gain this fault-tolerance, a peer-to-peer algorithm will be used to make other JobTrackers aware of each other's existence, and the jobs that each is running. That way, if a JobTracker fails, another JobTracker can easily step in and continue the tasks of the failed JobTracker until the tasks are complete.

Chapter 3—Analysis

There are really four main Distributed Hash Table (DHT) algorithms: Pastry, Tapestry, CAN, and Chord.

Pastry is a "generic peer-to-peer object location and routing scheme, based on a self-organizing overlay network of nodes connected to the Internet" [2]. This algorithm makes a special point to take network locality into account. "Each Pastry node keeps track of its immediate neighbors in the nodeId space, and notifies applications of new node arrivals, node failures, and recoveries. Because nodeIds are randomly assigned, with high probability, the set of nodes with adjacent nodeId is diverse in geography, ownership, jurisdiction, etc." [2]. It does this by assigning each node on the network a 128-bit node identifier, or nodeId. "The nodeId is used to indicate a node's position in a circular nodeId space, which ranges from 0 to $2^{128} - 1$. The nodeId is assigned randomly when a node joins the system. It is assumed that nodeIds are generated such that the resulting set of nodeIds is uniformly distributed in the 128-bit nodeId space" [2]. nodeIds are cryptographic hashes that can be generated from anything unique about each node. Examples include a hash of the node's public key, IP address, or MAC address [2]. Each file stored on a node is given a fileId. This provides a unique identifier for the file in the network and allows for nodes to query for the specific fileId of the desired file [2]. The fileId is generated by making a hash out of a unique characteristic of the file (ex. the filename and owner of the file), similar to how nodeIds are created [2].

One way that Pastry keeps itself organized is by giving each node its own routing table. "A node's routing table, *R*, is organized into $[log_{2^b} N]$ rows with $2^b - 1$ entries each. The $2^b - 1$ entries at row *n* of the routing table each refer to a node whose nodeId

shares the present node's nodeId in the first *n* digits, but whose *n* + 1th digit has one of the $2^b - 1$ possible values other than the *n* + 1th digit in the present node's id" [2]. This gives Pastry the desired distributiveness that its developers were looking for.

"Tapestry is an overlay location and routing infrastructure that provides location-independent routing of messages directly to the closest copy of an object or service using only point-to-point links and without centralized resources" [11]. Like Pastry, Tapestry combines location and routing algorithms. The idea was for Tapestry to have the "ability to address messages with location-independent names and to request those messages be routed directly to the closest copy of an object or service that is addressed by that name" [11].

Both Pastry and Tapestry implement variations of the Plaxton location and routing system. The way Plaxton works is that it uses local routing maps at each node to route overlay messages to the destination ID digit by digit [11]. It operates in a similar manner to the longest prefix routing in the CIDR IP address allocation architecture [11]. Each node will have a neighbor map with multiple levels. Each level matches a suffix up to the digit position in the ID and contains a number of entries equal to the base of the ID, "where the *i*th entry in the *j*th level is the ID and location of the closest node which ends in "*i*"+suffix(*node, j* -1)" [11]. For every *n*th node a message reaches, it shares a suffix of at least length *n* with the destination ID [11]. "Assuming consistent neighbor maps, this routing method guarantees that any existing unique node in the system will be found within at most *Logb N* logical hops, in a system with an *N* size namespace using IDs of base *b"* [11]. It is also important to note that in the Plaxton algorithm, every destination

node is the root node of its own tree. The Plaxton algorithm creates a mesh of embedded trees in the network with one tree being rooted at every node [11].

The Plaxton algorithm has many benefits—including simple fault handling, scalability, exploiting locality, and proportional route distance [11]. One of the limitations of the Plaxton algorithm, however, includes the requirement that all nodes have to be present and known at the creation of the mesh. This makes it very hard to add and remove nodes from the network once the mesh is established. Another drawback of the Plaxton algorithm is the vulnerability of the root nodes. "The root node for an object is a single point of failure because it is the node that every client relies on to provide an object's location information. A corrupted or unreachable root node would make objects invisible to distant clients" [11]. A third drawback of the Plaxton algorithm is its lack of ability to adapt to dynamic query patterns. Every time a node would be added or removed, the entire tree or potentially the entire mesh would have to be remapped [11]. This is simply not a viable option in an environment like Hadoop where the TaskTrackers and JobTrackers are only around as long as it takes to run a job and are, therefore, constantly going on and off line.

CAN, or Content-Addressable Network, is a scalable peer-to-peer indexing system that is composed of many individual nodes. "Each CAN node stores a chunk, or zone, of the entire hash table. In addition, a node holds information about a small number of "adjacent" zones in the table. Requests (insert, lookup, or delete) for a particular key are routed by intermediate CAN nodes towards the CAN node whose zone contains that key" [8]. The CAN design centers around virtual multi-dimensional Cartesian coordinate spaces that are completely logical. This means that the logical space has little to no

relation to the physical location of the devices. “The entire logical coordinate is dynamically partitioned among all the nodes in the system such that every node “owns” its individual, distinct zone within the overall space”[8]. The (key, value) pairs are stored in this space. “To store a pair (K_1, V_1), key K_1 is deterministically mapped onto a point P in the coordinate space using a uniform hash function. The corresponding (key, value) pair is then stored at the node that owns the zone within which the point P lies.” [8]

Retrieving data from CAN is in a similar fashion. “To retrieve an entry corresponding to key K_1, any node can apply the same deterministic hash function to map K_1 onto point P and then retrieve the corresponding value from the point P.” [8] When the data is not owned by the node that is making the request, the request is then sent through the CAN infrastructure until it reaches the node that does own the data. This means that CAN *has* to be implemented in an environment that allows for efficient routing.

The CAN nodes organize themselves into a network on the virtual coordinate space discussed earlier. Each node is assigned a zone, and then begins the process of discovering and maintaining the IP addresses of its immediate neighbors. This list of IP addresses essentially serves as the routing table for the node to forward information. [8] Each CAN message that traverses the network contains the destination coordinates—which allows a node to use the greedy algorithm to forward the message on to the neighbor with the closest match to the destination coordinates. This means that “for a d dimensional space partitioned into n equal zones, the average routing path length is $(d/4)(n^{1/d})$ hops and individual nodes maintain $2d$ neighbors” [8].

As seen in Figure 4, when a new node joins the CAN, that node is assigned its own coordinate space that has been "carved" out of its neighbor's coordinate spaces. It is important to note that the coordinate space wraps around the ends. [8] Usually the existing node splits its space in half, gives a half to the new node, and retains a half for itself. This process is usually done in three steps:

"1. First the new node must find a node already in the CAN.

2. Next, using the CAN routing mechanisms, it must find a node whose zone will be split.

3. Finally, the neighbors of the split zone must be notified so that routing can include the new node." [8]

When a join happens, every node in the system sends an immediate update message. This is followed by periodic refreshes with the node's currently assigned zone to all of its neighbors. So, a node addition only affects a small number of nodes that are already on the network in a small region of the coordinate space. In fact, only *O(number of dimensions)* nodes are affected. [8]

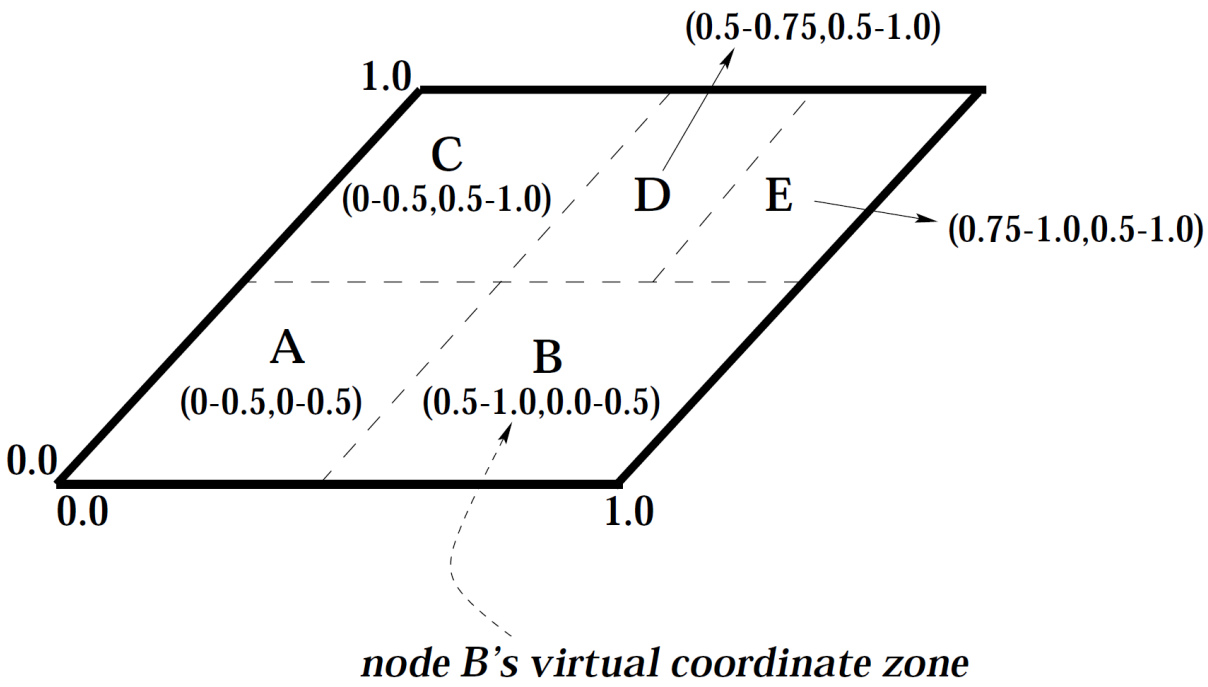

Figure 4: An illustration of an example 2-d space in CAN with five nodes. [8]

When a node departs the network, it explicitly hands the responsibilities of its zone over to a neighboring node. However, when a node fails and unexpectedly leaves the network, the (key, value) pairs that are held by the node are lost until the state is refreshed by the data holders. Usually, a node sends update messages periodically to its neighbors. These updates both prove that the node is still operational, and also provide the neighbors with a list of the node's neighbors and their zone coordinates. When a node does not receive updates from its neighbor for a prolonged amount of time, then the node assumes that its neighbor has failed and left the network. When a node is assumed to have failed, its neighbors start a takeover process to re-allocate the contents of the failed node's zone to its neighbors.

Chord is a distributed lookup protocol that provides the efficient location of data items [9]. In fact, the only major operation that Chord supports is that, given a key, it maps the key onto a node. The node may or may not be responsible for the value associated with the key—that is up to the applications above Chord. Chord just uses distributed hashing to assign keys to Chord nodes. Distributed hashing tends to balance load, and it requires relatively small amounts of key movement when nodes join and depart—hence its use in Chord [9].

When Chord is operating in the steady state, in a system with N nodes, each node maintains information about only $O(\log N)$ other nodes. Each node also resolves lookups with $O(\log N)$ messages to other nodes. Chord only needs one piece of correct information per node in order for guaranteed correct routing [9].

There are three features that set Chord apart from other peer-to-peer protocols: 1) Chord's simplicity—additionally, it handles joins and failures quite well, 2) Chord's provable correctness—even when the routing information it has is partially incorrect, & 3) Chord's provable performance. Chord's stabilization algorithms have been proven to maintain good lookup performance even through continuous joining and failure of nodes. [9]

The Chord software is a library that is linked with the applications that use it. "The application using Chord is responsible for providing any desired authentication, caching, replication, and user-friendly naming of data." [9] Chord's flat key-space makes implementation of these features quite easy.

Chord's routing is similar to that of the Grid Location System (GLS). In fact, Chord is a rendition of a one-dimensional analogue of GLS. "Chord maps its nodes to an artificial one-dimensional space within which routing is carried out by an algorithm similar to Grid's." [9]

"Chord acts as a distributed hash function, spreading keys evenly over the nodes"—thus providing a degree of natural load balancing. [9] "Chord is fully distributed. No node is more important than any other." [9] Thus robustness is improved, and makes Chord a great candidate for loosely organized peer-to-peer applications. "The cost of a Chord lookup grows as the log of the number of nodes, so even very large systems are feasible." [9] These systems require no parameter tuning to achieve this scalability. "Chord automatically adjusts its internal tables to reflect newly joined nodes as well as node failures." [9] This virtually guarantees that the node responsible for a key will always be found—unless there is a major network failure. This is even the case when the system is in a continuous state of change. "Chord places no constraints on the structure of the keys it looks up."[9] Chord uses a completely flat key space—which means that higher level applications have great leeway as to how they associate their own names to the Chord keys. [9]

Chord uses the SHA-1 as its base hash function because SHA-1 has good distributional properties. [9] Each node is assigned an *m* bit identifier that comes from the SHA-1 hash. The length of the *m* bit identifier only needs to be large enough to make the probability of two keys or nodes hashing the same identifier a negligible risk. "Identifiers are ordered on an *identifier circle* modulo 2^m. Key *k* is assigned to the first node whose identifier is equal to or follows the identifier of *k* in the identifier space. This node is

called the *successor node* of key *k*. If identifiers are represented as a circle of numbers from 0 to 2^m-1, then the *successor node* of key *k* is the first node clockwise from *k*." [9] This identifier circle is also known as the Chord ring—essentially the roadmap that Chord uses to do its routing, and can be seen in Figure 5. [9]

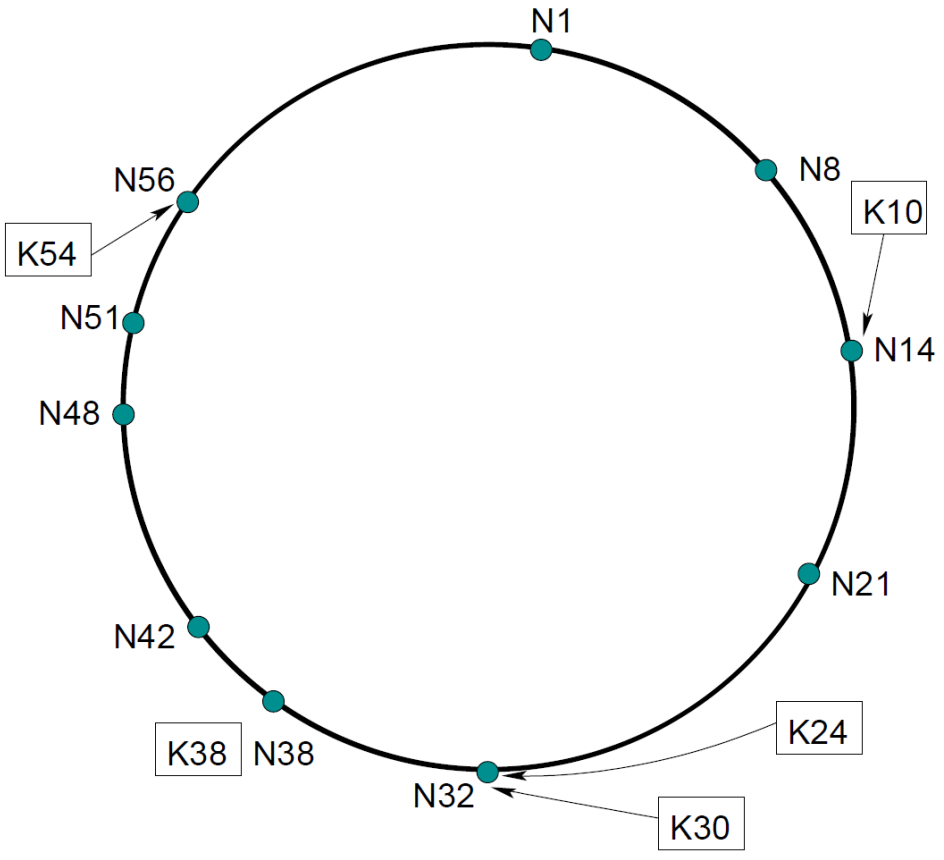

Figure 5: An example Chord ring with 10 nodes (*n*) and five keys (*k*). [9]

In the Chord ring, in order to ensure consistency in hash mapping when a new node joins the network, certain keys that were assigned to the new node's successor become assigned to the new node. [9] When the new node leaves the network, all the

keys assigned to the new node return to the new node's successor. There is no need for any other changes in assignment of keys to nodes. [9]

"Chord lookups could be implemented on a Chord ring with little per-node state. Each node needs to only know how to contact its current successor node on the identifier circle. Queries for a given identifier could be passed around the circle via these successor pointers until they encounter a pair of nodes that straddle the desired identifier." [9]

"Each node *n* maintains a routing table with up to *m* entries, called a *finger table*. An example of which can be seen in Figure 6. The i^{th} entry in the table at node *n* contains the identity of the *first* node *s* that succeeds *n* by at least 2^{i-1} on the identifier circle." [9]

Usually a finger table entry includes both the Chord identifier and the IP address (and port number) of the relevant node. [9]

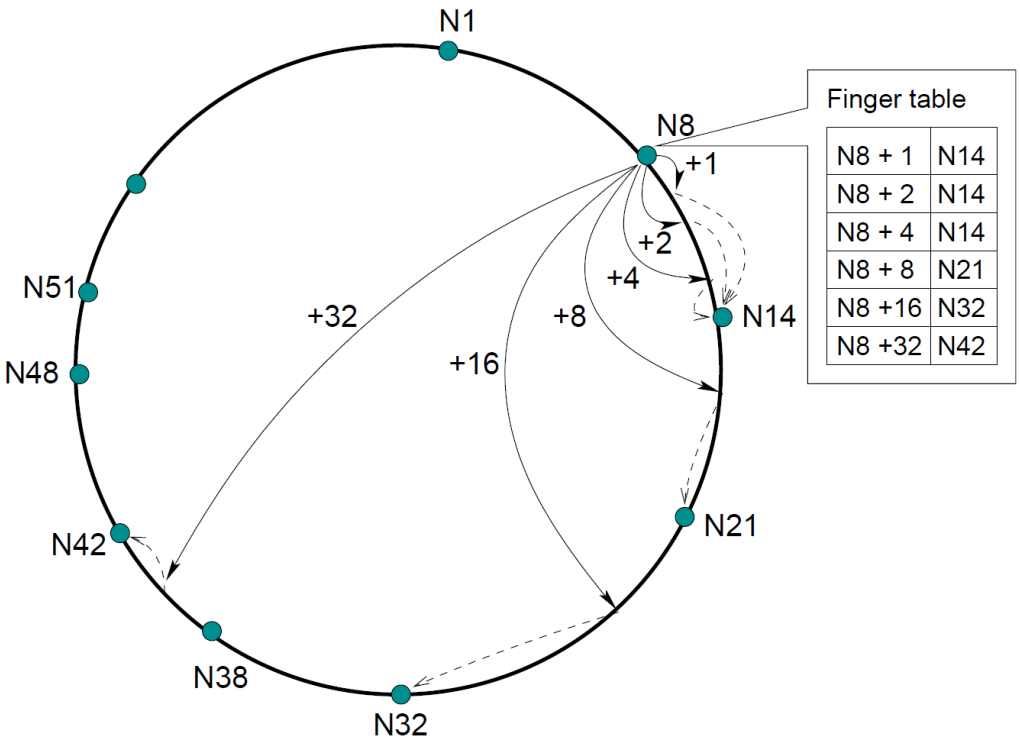

Figure 6: The finger table for Node 8 in an example network. [9]

Each node only knows about a few other nodes, and knows the most about the nodes closely following it on the identifier circle. [9] Each node has finger entries around the ring at power of two intervals. This means that each node can forward a query at least halfway along the remaining distance between the node and the destination. [9] "If the distance between the node handling the query and the predecessor *p* halves in each step, and is at most 2^m initially, then within *m* steps the distance will be one, meaning we have arrived at *p*." [9] "After 2log*N* forwardings, the distance between the current query node and the key will be reduced to at most $2^m/N^2$." [9] The average lookup time is ½ log *N*. [9]

For the lookups to execute correctly, as the nodes come on and off line, Chord must know that each node's successor pointer is accurate with the current state of the nodes. [9] Each node runs *stabilize*() function periodically in order to learn about newly joined nodes. Each time *stabilize*() runs on a node, "it asks its successor for the successor's predecessor *p*, and decides whether *p* should be *n*'s successor instead." [9] The *stabilize*() function also notifies the node's successor of the node's existence, giving the successor the chance to change its predecessor to the newly joined node—assuming there's no other predecessor than the node. [9]

"At each step in the process, n_s is reachable from n_p using successor pointers; this means that lookups concurrent with the join are not disrupted." [9] The stabilization scheme guarantees nodes will be added to a Chord ring in a way that preserves reachability of existing nodes—even with concurrent joins, lost messages, and reordered messages. [9]

"To increase robustness, each Chord node maintains a *successor list* of size *r*, containing the node's first *r* successors." [9] If a node's immediate successor is non-responsive, the node can substitute the second entry in its successor list for the non-responsive node. [9] If the node's successor has failed, it replaces the successor with the first live entry in its successor list and changes its successor list to reflect the new successor. "At that point, *n* can direct ordinary lookups for keys for which the failed node was the successor to the new successor." [9]

"The fact that a Chord node keeps track of its *r* successors means that it can inform the higher layer software when successors come and go, and thus when the software should propagate data to new replicas." [9] This feature would prove very helpful in achieving fault-tolerance inside Hadoop. Usually, when a node departs the Chord network, it happens in a manner as follows: "First, a node *n* that is about to leave may transfer its keys to its successor before it departs. Second, *n* may notify its predecessor *p* and successor *s* before leaving. In turn, node *p* will remove *n* from its successor list, and add the last node in *n*'s successor list to its own list. Similarly, node *s* will replace its predecessor with *n*'s predecessor." [9]

Conclusion

Both Pastry and Tapestry have the ability to correlate the physical location of the node and the spatial location of the node in the DHT. However, they both use the same underlying algorithm to accomplish this task, and that algorithm does not handle large changes in the network in short amounts of time very well [7]. In the Hadoop environment, with nodes coming on and offline based on the progress of their assigned jobs, the DHT that is implemented will need to handle large changes in the network in very short amounts of time quite well. CAN and Chord seem more suited for this. However, because both CAN and Chord do not use the Plaxton routing and location system (the algorithm that makes it possible to correlate the physical and spatial locations of nodes, and the one that doesn't handle change well), they both cannot correlate the physical and spatial locations of nodes. This means that a node that is spatially very close may indeed be many physical hops away from its neighbor (i.e. in another state or country).

There are several similarities and differences between CAN and Chord. CAN uses a bootstrap node to maintain a partial list of nodes believed to be currently in the system at a given point in time [8]. It is unclear what happens if the bootstrap node fails. This could be a single point of failure. CAN does do a good job of making sure the physical neighbors have a low chance of being logical neighbors by randomly assigning new nodes space in the network [8]. However, the joining process in CAN could lead to localized broadcast flooding [8]. This would not be an acceptable behavior inside Hadoop. Another drawback with CAN is in the way that CAN handles unexpected departures. This may lead to a temporary loss of (key, value) pairs that were in the space

of the departed node until that space can be remapped by neighboring nodes that are still in the network [8]. However, CAN does have a heartbeating mechanism built in that allows a node to know if its neighbors are still in the network [8]. Additionally, CAN has a sequence built in to takeover the space of any departed nodes [8]. Probably the most interesting part of CAN is that it has the ability to map the same (key, value) pairs on multiple dimensions. This allows for several interesting features: 1) it allows for a node to have "express routes" across a network, and 2) all JobTrackers used to ensure fault-tolerance on a task can be brought into the same dimension [8].

Chord, on the other hand, appears to be a little faster than CAN in getting to and maintaining stability [9]. Chord is also more often correct when given partially incorrect routing information [9]. Both CAN and Chord are self-organizing, and both algorithms will learn and maintain addresses of their neighbors [8] [9].

Chord is a distributed lookup protocol that specializes in mapping keys to nodes [9]. "Data location can be easily implemented on top of Chord by associating a key with each data item, and storing the key/data pair at the node to which the key maps. Chord adapts efficiently as nodes join and leave the system, and can answer queries even if the system is continuously changing. " [9]. This is especially a good fit to MapReduce because MapReduce already maps key-value pairs to process data, and mappers and reducers are constantly being brought up and shutdown [4]. Chord does very well with load balancing because it acts as a distributed hash function and spreads keys evenly over the participating nodes. Chord is also decentralized; no node is of greater importance than any other node. Chord scales automatically without need to do any tuning to achieve success at scale. "Chord automatically adjusts its internal tables to reflect newly joined

nodes as well as node failures, ensuring that, barring major failures in the underlying network, the node responsible for a key can always be found. This is true even if the system is in a continuous state of change." [9]. Chord also features flexibility in nomenclature because there is no hierarchy in the key-space. This allows Chord to easily adapt to other names or keys that applications have already assigned to the data, and map or use them as Chord keys.

In the end, Chord's simplicity, ability to handle joins and failures, correctness in "hostile" routing environments, performance, and stabilization algorithms lend to the conclusion that Chord would be the best fit for implementing fault-tolerance among JobTrackers inside the Hadoop running environment.

Literature Cited

[1] D. Borthakur, “Hadoop Architecture and its Usage at Facebook”, lecture, Oct., 2009.

[2] P. Druschel and A. Rowstron. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. Submission to ACM SIGCOMM, 2001.

[3] A. Joseph, “Cloud Computing Infrastructure: networking, storage, computation models”, lecture, RWTH Aachen, Mar. 19, 2010.

[4] J. Lin and C. Dyer, *Data-Intensive Text Processing with MapReduce*, Morgan & Claypool, 2010

[5] “Map/Reduce.”Internet: http://www.cbsolution.net/ontarget/mapreduce_vs_data_warehouse, May 24, 2011 [Dec. 12, 2011].

[6] P. Mell and T. Grance, “The NIST Definition of Cloud Computing,” Nat. Inst. of Standards and Tech., Gaithersburg, MD, Special Publication 800-145, Sep. 2011.

[7] C. Plaxton, Rajaraman, R., and Richa, A. “Accessing nearby copies of replaced objects in a distributed environment”. In *Proceedings of the ACM SPAA* (Newport, Rhode Island, June 1997), pp. 311-320.

[8] S. Ratnasamy, Francis, P., Handley, M., Karp, R., and Shenker, S. “A scalable content-addressable network”. In *Proc. ACM SIGCOMM* (San Diego, CA, August 2001), pp. 161-172.

[9] I. Stoica *et al.,* “Chord: a scalable peer-to-peer lookup protocol for Internet applications”, *IEEE/ACM Transactions on Networking*, vol. 11, issue 1, pp. 17-32, Feb. 2003.

[10] T. White, *Hadoop: The Definitive Guide, Second Edition*, Sebastopol, CA: O’Reilly, 2011.

[11] B. Y. Zhao, J. D. Kubiatowicz, and A. D. Joseph. Tapestry: An infrastructure for fault-resilient wide-area location and routing. Technical Report UCB//CSD-01-1141, U. C. Berkeley, April 2001.