A Survey of three DHT Systems: CAN, Pastry, and Chord

Hoang Thai Duong

1. Overview

One of the key problems in P2P is efficient lookup. Given a key, we want to locate the nodes that store the data associated with the key. A system that provides that hash-table-like service is called a distributed hash table (DHT). In this report, we study three early, notable DHT systems out of the many that have been proposed in recent years, namely CAN [1], Pastry [2], and Chord [3]. We look at how each of them solves the basic lookup problem, compare their performances and scalabilities, and assess them with regards to important criteria such as load balancing, data availability, and robustness. Along the way, we also comment on their shortcomings and propose possible research directions to overcome those drawbacks.

2. Identifier Space

Usually, a DHT system uses some hash function to map keys to a space called “identifier space” (ID space). An ID space is divided among the nodes by assigning a different “zone” to each node. A node stores the data/values associated with the keys within its zone.

CAN uses a uniform hash function to map a key to a point in a wrapped d-dimensional Cartesian space. The entire space is partitioned into distinct zones, each own by a node. The mapping of a zone to a node is done randomly when the node joins the system.

The ID space in Pastry is a 128-bit field. Both the keys and the nodes’ IDs (e.g. IP addresses) are put through a hash function, typically SHA-1. In the ID space, the ID of a node or a key is a sequence of digits written in base 2^b. A node is responsible for the keys that are numerically closest to it.

Similar to Pastry, Chord also uses m-bit IDs for both keys and nodes, and uses SHA-1 as the hash function. The ID place consists of all possible IDs put on a circle of size 2^m (basically the IDs wrapped around, and all arithmetic operations are done in modulo 2^m). A key is assigned to the first node whose ID is equal or bigger than itself.

3. Routing

In this section, we look at how the DHT systems route a key lookup from a requesting node to the node(s) responsible for the key. DHT routing is often about the trade-off between per-node state size and routing path length. The more data a node store, the more efficiently it can route.

In CAN, each node maintains a list of neighbors. Two nodes are neighbors if their zones span the same ranges in $d-1$ dimensions and are adjacent in the last dimension. Upon receiving a key lookup, if a node doesn't own the key, it forwards the lookup message to one of its neighbors, the one whose zone is closest to the key. Basically, the message follows a straight line in the ID space from the source node to the destination node. A CAN node in a d-dimensional space manages a list of $O(2d)$ neighbors. Since d is fixed, as the number of nodes N increases, the storage on each node stays the same. However, a small d implies longer routing path lengths (number of nodes on the path). On the other hand, letting d grows dynamically as N grows means big overheads when N is large (since the topology of the ID space needs to be rebuilt again each time d changes), so that's not an option. With d fixed, the routing path length grows as fast as $O(N^{1/d})$, which is not as good as Pastry's and Chord's performances ($O(\log N)$).

A Pastry node has a list of leaf nodes, a routing table, and a neighborhood set. The leaf set consists of L nodes numerically closest to the current node. $L/2$ nodes out of which have smaller IDs than the current node, and $L/2$ nodes have larger IDs. For a system consisting of N nodes, a routing table has $\log_{2^b} N$ rows and 2^b columns. An entry on row n, column c ($c \leq 2^b$) points to a node whose ID shares the same first n digits as the current node's, and has c as its $(n+1)$th digit. In principle, there are many nodes satisfying the condition. The stored node is chosen so that it's close to the current node "geometrically" (e.g. in round-trip-time, or IP hops, not numerical distance in ID space). If c is the $(n+1)$th digit of the current node's ID, the entry is actually empty (it's supposed to be the current node itself). A neighborhood set contains M geometrically closest nodes to the current node; it is not used in routing but helps the system achieve better locality. Given a key lookup, a node first finds in its leaf set the node that covers the key. If such a node exists, it forwards the lookup message to that node. If not, it finds in the routing table a node whose ID have a longer common prefix with the key than its own and forward the message to that node. If that is still not possible, the current node looks (in its leaf set, routing table, and neighborhood set) for a node whose ID is numerically closer to the key than itself (both have the same common prefix length with the key) and forward to that node. The message will eventually reach its destination since each step shortens the distance towards the destination. In a Pastry system with N nodes, a routing table has approximately $\log_{2^b} N * (2^b - 1)$ entries. Typical values for L and M are 2^b or $2 * 2^b$. Typical value for b is 4, a small constant not depending on N. So the typical total amount of storage per node is $O(\log_{2^b} N)$. Path length is $O(\log_{2^b} N)$ as well since each routing step comes one digit closer to the key.

A node in Chord has a "finger table" with m entries. The ith entry in the table of node n refers to the first node that succeeds n by at least 2^{i-1} in the circle (e.g. when $i = 1$, it is the immediate successor of n). The ith entry also associates that node with a range, $[n + 2^{i-1}, n + 2^i)$ (remember all arithmetic operations are done in modulo 2^m). Finally, a node stores links to its r successors and the immediate predecessor in the circle. The role of the successor list is somewhat similar to that of the leaf set in Pastry (to route in "slow" mode when there are no other choices). When a node n receives a lookup message, there are three cases. In the first case, the key does not belong to n itself, and also does not belong to its immediate successor (i.e. the key falls between n and its successor), n uses its finger table to find the range in which the key falls into, and forwards the message to the node associated with that range. In the second case, the key belongs to n's successor, n forwards the request to the successor. In the trivial last case, the key belongs to n itself, no more routing needs to be done. The lookup message will eventually reach the destined node since each step makes progress towards the key. Each node in a Chord system with N nodes stores a finger table with m entries, so the per-node storage is fixed at $O(m)$ (in the sense that m does not change during runtime). Routing path length is $O(\log N)$ with high probability (experiments show the expected value is $\frac{1}{2}\log N$). Intuitively, that can be explained by observing that a lookup message for a faraway key travels very fast in the beginning, roughly halving the distance in each step. Since m is fixed, we can assume m is chosen to be big enough to "cover" any potentially big N, which implies a lot of redundant storage is wasted when N is actually small.

4. Self-Organization

Self-organization refers to the ability of a DHT system to maintain correct and efficient routing in the presence of multiple node arrivals and departures. A good DHT system should have small overheads when nodes joining and leaving the system.

4.1. Node Arrival

Generally, when a new node wants to join a system, it must somehow know about an existing node and contact that node to get information about other nodes so that it can initialize its own routing table. Usually one existing node will split its zone in the ID space and transfer a portion to the new node. Some other existing nodes must also be notified of the new node so that they can update their routing tables to reflect the new system state.

When a new node A wants to join a CAN system, it looks up the CAN domain name using DNS to find a bootstrap node's IP. The bootstrap node sends A the IPs of random nodes in the system. A then chooses a random point in the ID space and sends a join request for that point. The request is routed normally to a node B responsible for the point. B then splits its zone in two and gives A one half. A initializes its neighbor set using a subset of B's neighbor set. B also removes some nodes which are no longer its neighbor from the set. For other nodes in the system to know about the new situation, an update message is sent periodically by each node to its neighbor. All of A's neighbors will know about A when they receive update messages from A, and all of B's remaining neighbors will know that B now only owns half of its old zone once B updates them. Even though each node has only $O(d)$ neighbors, the addition of a CAN node has a cost of $O(N^{1/d})$, since it involves routing the new node to its chosen zone.

In Pastry, a new node X knows about an existing node A using "expanding ring" IP multicast or by outside means. A is assumed to be close to X in some proximity metric (such as distance message travel or IP routing hops). X sends A a join message with the key X. Pastry routes the join message to an existing node Z whose range covers X. X initializes its leaf set using Z's leaf set since these two nodes are very close numerically. With regards to the routing table, the first row in X's routing table is taken from the first row in A's table (assume A and X share no common prefixes). The second row in X's table is taken from the second row in B's table, where B is the second node (after A) on the path that the join message travels from A to Z. This makes sense since X and B share a common prefix of one digit. The process continues similarly for the rest of the rows. For the neighborhood set, X copies that from A since they are assumed to be close "geometrically". Finally, X sends its leaf set and routing table to every node it has found so far so that those nodes can update their state if necessary. There are no mentions of actual keys/values transfer in the paper, but we can assume this can be done easily by making the new node contacting its two immediate neighbors to get its share of keys/values. The cost of node arrival in Pastry is $O(\log_{2^b} N)$, which is roughly the number of nodes on the path from A to Z. $O(\log_{2^b} N)$ is the best node arrival performance among the three systems, as far as the $O()$ notation is concerned.

In Chord, a new node n needs to know about an existing node n' before it can join a system. To fill its finger table, n asks n' to look for the successors of $n + 1$, $n + 2^1$, $n + 2^2$, etc. It optimizes a bit by checking if the ith finger is also the $(i + 1)$th finger without querying n' every time. n's predecessor and successor pointers can also be found easily using its neighbor's corresponding pointers. To notice other nodes to update their finger tables, for each $i \leq m$, n find the immediate predecessor of $n - 2^{i-1}$, called p. If the ith finger of p succeeds n, then p needs to update its ith finger to n. Finally, n contacts its immediate successor to ask that node to transfer the keys/values that n is now responsible for. It is shown in the paper that with the optimization mentioned above (when filling the finger table), the cost of node addition in Chord is $O(log^2 N)$. This is better than CAN but worse than Pastry. As an improvement, a new node will not right away construct its finger table. It will only ask for the immediate successor and predecessor to be able to route in the most basic way. The successor and predecessor pointers are also checked periodically by each node. Also, each node will periodically "update" a random entry in its finger table (by sending a find-predecessor request, similar to a key lookup) and attempt to fix the entry if it's incorrect. The cost of this stabilizing process is quantified in [4].

4.2. Node Departure/Failure

Usually when a node leaves, its zone/range should be transferred to some other node, and nearby nodes must update their routing tables. When a node fails, some other node must detect the failure and can take over the failed node's zone, other nodes also need to update their routing states. As we will see, some DHT systems consider leaving the same as failing and no zone transfer is done, only routing information is corrected.

A CAN node before leaving will transfer its zone to one of its neighbors, the one with the smallest zone. If the transferred zone and its neighbor's zone can be merged into one valid zone, it's done. Otherwise, the neighbor will temporarily handle both zones. A background process to reassign zones will prevent further fragmentations of the ID space. On the other hand, node failure is detected by the absence of periodic updating messages. Using different timers, among the neighbors, the one with the smallest zone will take over the failed node's zone. Zone transferring and take over is unique to CAN since it the mapping of zones to nodes is less "deterministic" than in the other two systems. That is to say, when a node in Pastry or Chord leaves, its zone automatically belongs to another node. However, Pastry and Chord may still benefit from zone transferring since not only the keys but also the associated values are transferred; it should improve data availability. Unlike in the case of node arrival, node departure in CAN does not involving routing, so the complexity is only $O(d)$, which is better than the other two systems.

In Pastry, a node departure/failure is treated as the same thing and is detected by its neighbors when they try to contact it unsuccessfully. If A detects that some node B in its leaf set is gone, it contacts the furthest node Z in the leaf set in the direction of the failed node, and asks for Z's leaf set. A then chooses one appropriate node C in the received leaf set to replace B in its own leaf set. To replace a failed node on row i column j in the routing table, a node first asks other nodes on the same row for the entry at (i,j) in their tables. Should that fails, it tries the next row, $i + 1$, and so on. Finally, if a node finds that some node in its neighborhood set fails, it queries other nodes in the set for their neighborhood sets and use that to repair its own neighborhood set. Because each node has $O(\log_{2^b} N)$ neighbors, if we assume the expected amount of operations a neighbor must do when it detects a node failure in its routing table is constant, the cost of node departure/failure in Pastry is $O(\log_{2^b} N)$.

A Chord node keeps a list of r successors and updates this list every time it detects node failure/departure (same thing in Chord). A stabilization process is run periodically in the background to correct wrong finger table and successor entries. The successor list helps when all routing paths found using the (outdated) finger table fail, in which case a message is routed using only successors (slow but correct). The cost of node failure/departure in Chord is $O(log^2 N)$ since there are expectedly $O(\log N)$ nodes that need to repair their finger table index entry that contains the fail node, which in turn takes $O(\log N)$ steps.

5. Additional Properties

Beside the basic operations described above, we can assess a DHT system based on other important criteria such as load balancing, locality, data availability, and routing fault tolerance.

5.1. Load Balancing

In a P2P system, it's very important that every node has roughly the same amount of computation and traffic load. In the context of DHTs, there are two different kinds of load balancing: static load balancing and dynamic load balancing. Static load balancing has to do with the hash function and the way the identifier space is partitioned: the more uniform it is, the more balanced the load. Dynamic load balancing attempts to make "hot" keys more available by, for example, replicate them on multiple nodes (so that the few nodes responsible for those keys are not too busy with lots of queries).

CAN supports static load balancing (uniform partition) by changing the way a new node joining the system acquires its zone. When A tries to join, it decides on a random position P, which is in the zone of B. B finds among its neighbors a node C that has the largest zone, and let C splits its zone to give to A (it was B who did the splitting in the original design). Similarly, when a node leaves the system, the neighbor with the smallest zone takes over, as mentioned before. With regards to dynamic load balancing (hotspot control), CAN does both caching and replication. Values are cached along routing paths so that when the same key lookup is encountered, the corresponding value can be accessed from the cache. Also, a node can replicate certain keys/values on its neighbors if it finds the keys being looked up by many nodes. The design of CAN is elegant since static load balancing is achieved almost automatically (unlike Pastry and Chord which must worry about uniform distribution). However, caching and replication are not integrated parts in the design and can be adopted in other systems as well.

Pastry makes no efforts to support dynamic load balancing. For static load balancing, other than using a uniform hash function (e.g. SHA-1), Pastry does not guarantee anything. This is partly because the mapping of keys to nodes in Pastry is more deterministic than in CAN. In Pastry the nodes are also hashed into the ID space, and a node is responsible for the keys closest to it, so techniques used in CAN do not work here. The base hash function (SHA-1) and the mapping of keys to nodes in Pastry, however, is quite similar to Chord. So it may benefit from the techniques Chord uses to achieve more load balancing, described next.

Chord also uses SHA-1 as the base hash function to map nodes and keys to identifier space, and consistent hashing to map keys to nodes. However, the authors find that nodes are not distributed evenly in the identifier space under the base hash function. They propose distributing $r * N$ $(r \sim \log N)$ nodes uniformly in the identifier space and assign r virtual nodes to each real node. That means the mapping of keys to nodes described before is still correct, but with regards to virtual nodes instead. This way Chord achieves better load balancing but at the cost of increased per-node storage since each real node now must store a finger table for each of the r virtual node. Also, the worst-case query path length is longer, though this is not reflected well by the big O analysis ($O(\log N \log N)$ is still $O(\log N)$). Similar to Pastry, no explicit support for dynamic load balancing is mentioned in the paper.

5.2. Locality

(Logical) path length is not the only metric used to measure the efficiency of a routing algorithm. Another useful metric concerns the actual number of IP hops or routing latencies between nodes. Good locality means on average the travelling time of messages between nodes are short.

The authors of CAN propose two improvements to shorten the routing latency. The first improvement is to use the ratio of logical distance and actual round-trip-time as the routing metric and not the logical distance alone. This is actually a greedy algorithm and may not work well in all cases. This scheme works better as the number of dimension increases, however, because then the number of a node's neighbors increases, the average path length decreases, and the greedy choice becomes more "certain". The second improvement is that when a node joins the system, it is assigned a zone according to its distances to a well-known set of machines on the net (e.g. the root DNS servers) called landmarks. The reason is that nodes that are close to each other "geometrically" are likely to have similar distances to those landmarks and thus be close in the ID space. This scheme, however, depends a lot on the distribution of the landmarks, and in some cases can conflict with the load balancing requirement, since some parts of the ID space can be more populated than the others.

Pastry achieves locality by storing on each node a neighborhood set M. When a node X joins the system, it builds its routing table based on the nodes on the routing path from A to Z (see the routing section). X also retrieves the neighborhood sets of those nodes to augment its own routing table and neighborhood set. The paper argues that if the nodes in the routing table of any node B_i on the path from A to Z are close to Bi, then the nodes in X's routing table are also close to X. This argument is rather vague and weak, drawing conclusion mostly from experimental results. The routing based on a routing table constructed in this way is also a greedy approach, which is not guaranteed to work well in all cases, similar to CAN.

Chord is different from the other two systems in that it makes no explicit efforts to optimize for locality in its design. One possible way to make Chord aware of locality is to borrow CAN's idea and attach a round-trip-time to each entry in the finger table, and use it as a weight to estimate the message travel time when messages are routed to each finger node (e.g. some finger may result in longer path length but shorter latency to make up for that). However, that's still a greedy, local approach which requires experiments to verify.

5.3. Data Availability

In a good P2P system, when some of the nodes fail, the data belong to them should still be available. This is not a strict requirement but is nice to have. Usually availability is achieved replicating data on multiple nodes. However, with replication, a system's complexity and traffic load will increase in general.

CAN improves data availability through the use of multiple "realities", overloaded zones, and multiple hash functions. Multiple realities mean each node is assigned a different zone in each reality, so the same zone actually belongs to multiple nodes in different realities. Overloading zones is to assign two or more nodes to a zone instead of one. Multiple hash functions imply hashing a key k times using k different hash functions so that the k images belong to different nodes. The three techniques sound very similar to one another and are all "orthogonal" to the main design of CAN, which means they can possibly be used by other systems as well.

A Pastry system makes data more available by replicating the value associated with a key on k closest nodes to the key. This is simple and sufficient.

Chord is different to the other two system in that it doesn't assume the responsibility of replication. Instead, replication is supposed to be done by the system in a higher "layer" which uses Chord's functionalities. Chord can use the successor list to inform the higher layer application to replicate a key to k nearest nodes whose IDs are equal or more than the key.

5.4. Routing Fault-Tolerance

Routing fault-tolerance refers to a system's ability to route correctly (if possible) even when there are multiple node failures. In general, it means there must be more than one path between any two nodes, and the system should try all those paths in case routing attempts between the two nodes repeatedly fail. All three systems satisfy this condition; when some nodes fail, the systems still route correctly, albeit maybe more slowly. Over time, a stabilization process will fix the broken links in routing tables. This section only concerns the efficiency of routing before fail links are fixed.

A CAN node stores $O(2d)$ neighbors and can route through any of them. When nodes fail, on average we expect the routing path length to be longer but is still $O(N^{1/d})$.

The design of Pastry is quite similar to Chord, so we expect the same results proven in the Chord paper are true to Pastry as well, which means the average routing path length when there are failed nodes is the same as in normal case – $O(\log N)$. The experiments also show confirmative results.

The authors of Chord prove that in case of multiple node failures, with the use of successor list, the expected routing path length is still $O(\log N)$.

6. Advanced Analysis of Chord

Liben-Nowell et al. [4] prove a lower-bound on the complexity of maintenance (self-organization) of P2P system in general and analyze the maintenance process of Chord more formally. In general, in the presence of node arrivals and departures, a P2P system needs $O(\log N)$ message exchanges at least to "stabilize" the system. In addition to the ideal Chord state used in the original paper, this paper defines 3 new loosen "ideal" states in the presence of pure node joins, pure node failures, and a combination of both. An ideal state must satisfy certain conditions on node distribution, connectivity and validity of successor list and finger table. It shows that in all three cases, the routing in Chord is still efficient ($O(\log N)$), and the stabilizing process runs in $O(log^2 N)$ time, which is a bit more than the theoretical lower bound proven above. One final contribution of this paper is an extension of the Chord protocol to allow it to recover from an arbitrary state, not just states that can be reached from node arrivals and departures. They show that their algorithm can stabilize the system from an arbitrary state in $O(N^2)$ time without node failures, and in $O(N^3)$ time with node failures. These are all interesting results and a possible future work would be doing the same (or possibly more) formal analysis to other DHT systems.

7. Conclusion

In this report we investigate and compare 3 early DHT systems that are CAN, Pastry, and Chord. The designs of CAN and Chord are quite simple and elegant in many aspects. The main drawback of CAN is the rather high bound on routing path length. Pastry's design, on the other hand, is somewhat more ad-hoc and not "integrated". As a result, it is difficult to assess Pastry formally in some criteria (such as locality). There are also quite a number of improvements proposed in one paper but can benefit other techniques as well, since those improvements are not specifically tied to that one technique. Finally, to better understand the three systems (and many others proposed later), formal as well as experimental/practical frameworks such as [4] need to be invented, so that we have common grounds based on which the techniques can be better assessed and compared.

8. References

[1] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A scalable content-addressable network. In ACM SIGCOMM, 2001.

[2] A. Rowstron and P. Druschel. Pastry: Scalable, distributed object location and routing for

large-scale peer-to-peer systems. In ACM ICDSP, 2001.

[3] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for Internet applications. In ACM SIGCOMM, 2001.

[4] D. Liben-Nowell, H. Balakrishnan, and D. Karger. Analysis of the Evolution of Peer-to-Peer Systems. In ACM PODC, 2002.