U18ISE0006- Cloud Computing

Chord, Pastry and Kelips P2P system Case study

Name : Monika M

Roll no: 20BIS025

Assignment no : 01

Title : Chord, Pastry and Kelips Case study

Course Code: U18ISE0006

Date : 20/09/2022

1.Chord P2P system:

Chord is a Peer-to-Peer (P2P) protocol introduced in 2001 by Stoica, Morris, Karger, Kaashoek, and Balakrishnan. An ideal Peer-to-Peer system has no centralized control and no imbalance in responsibilities between nodes. This allows for better distribution of workload throughout the system, and does not introduce a single point of failure. Chord, specifically, is a distributed hash table (DHT): an application of P2P systems that allows for fast lookup across a distributed system. Chord's main contribution is the ability to achieve lookups in O(log N) steps, with each node only keeping track of O(log N) other nodes in the system.

	Memory	Lookup Latency	#Messages for a lookup
Napster	*O(1)* (*O(N)*@server)	*O(1)*	*O(1)*
Gnutella	*O(N)*	*O(N)*	*O(N)*
Chord	*O(log(N))*	*O(log(N))*	*O(log(N))*

Why CHORD:

P2P systems were often used for file-sharing, with applications such as Napster and Gnutella. However, Napster had a centralized look-up system, offering a single point of failure, and Gnutella had an inefficient flooding mechanism, which sent out O(N) queries per look-up. These design choices negatively impacted the scalability and security of these underlying systems; Chord resolved these shortcomings through alternative designs and routing schemes.

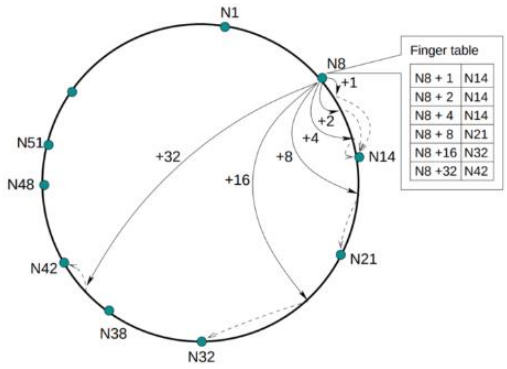
Purpose Of CHORD:

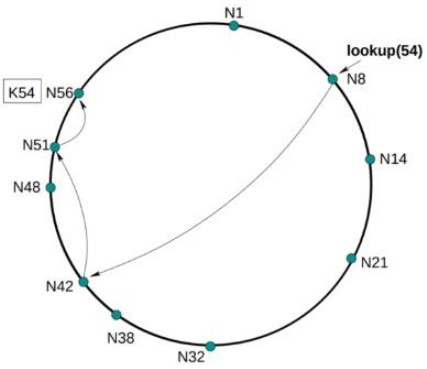
Chord supports the lookup operation of mapping a key onto a node. With this operation, one can create a DHT: the key can be associated with a value, which is stored on the node that the key maps to. To achieve scalability, the mapping occurs in sublinear O(log N) number of look-ups, and each node also only stores routes to sublinear O(log N) other nodes.

Chord's structure can be described by a number line that contains 2^m integer valued addresses between [0, 2^m), where the line wraps around via the modulo operation. Thus, from here on, we will refer to the structure as the Chord Ring, with values increasing clockwise around the ring. Each Node is assigned an ID/position on this Chord Ring through the SHA-1 hash of its IP address. Similarly, each key is assigned an ID through its hash. The key is stored on its "successor": the Node with the ID that directly succeeds it; in the case where the key has the same ID as a Node, the key is stored on that node.

A trivial method of locating the Node that stores a key is to move clockwise among the nodes until we find the Node ID that succeeds the key ID. However, this takes O(N) lookups, even though each Node only needs to keep track of it's successor.

To achieve faster look-up, Chord uses the concept of a finger table. Each Node keeps track of up to m other Nodes in its finger table, where the i^{th} entry of the table is the Node that directly succeeds the ID of the Node + 2^i. This allows for faster routing with O(log N) lookups, since rather than going to the next Node clockwise on the Chord Ring, a query can skip to a Node at least half the remaining distance to the true ID of the key.





How CHORD maintains Correctness:

In order for this routing to work under a dynamic environment where Nodes are constantly joining and leaving, Nodes in Chord need to have accurate successor pointers and finger tables. Therefore, Nodes will occasionally stabilize by checking if a) their successor is still alive, and b) their successor is correct: its predecessor index points back to them. If either of these fail, Nodes will update their successors accordingly. Nodes will also occasionally fix their finger tables through the find_successor method to keep them up to date. To ensure a lookup doesn't fail, each Node will also maintain a successor list that keeps track of its m direct successors nodes. If the direct successor is no longer alive, the Node will query subsequent successors until it finds one that is alive. Therefore, the lookup can only fail if no nodes in this list are alive, which is improbable. The successor list is maintained in the stabilize step.

Load Balance:

Chord's load balancing guarantees are presented through the following three Theorems :

Theorem 1: For any set of N nodes and K keys, each node is responsible for at most(1 + log N) K / N keys with high probability.

Theorem 2: With high probability, the number of nodes that must be contacted to find a successor in an N-node network is O(log N).

Theorem 3: With high probability, any node joining or leaving an N-node Chord network will use $O(\log^2 N)$ messages to re-establish the Chord routing invariants and finger tables.

Advantages :

The chord is indeed has good load balance properties during normal operations

Chord addresses peer addressability and peer findability and message routability challenges by organizing all peers in the P2P network into a single virtual ring.

Each peer is assigned a unique GUID which is used for addressability, findability and message routability.

Disadvantages :

Even though having good balance properties but chord is tolerant to node failure(when nodes voluntarily leave the system).

A timeout occurs when a node tries to contact a failed node. This occurs during the find_successor operation when a node traverses its finger table and its successor list to find the closest living predecessor to a specific key.

A failed lookup occurs when a query is unable to find the closest living successor to a target key. It is important to note that a lookup succeeds even if the found successor has not claimed the key yet (i.e. the key is currently owned by a departed node).

2.Pastry P2P system:

Pastry is an overlay network and routing network for the implementation of a distributed hash table (DHT) like Chord. The key–value pairs are stored in a redundant peer-to-peer network of connected Internet hosts. The protocol is bootstrapped by supplying it with the IP address of a peer already in the network and from then on via the routing table which is dynamically built and repaired. It is claimed that because of its redundant and decentralized nature there is no single point of failure and any single node can leave the network at any time without warning and with little or no chance of data loss. The protocol is also capable of using a routing metric supplied by an outside program, such as ping or traceroute, to determine the best routes to store in its routing table.

Although the distributed hash table functionality of Pastry is almost identical to other DHTs, what sets it apart is the routing overlay network built on top of the DHT concept. This allows Pastry to realize the scalability and fault tolerance of other networks, while reducing the overall cost of routing a packet from one node to another by avoiding the need to flood packets.

Working of Pastry:

The hash table's key-space is taken to be circular, like the key-space in the Chord system, and node IDs are 128-bit unsigned integers representing position in the circular key-space. Node IDs are chosen randomly and uniformly so peers who are adjacent in node ID are geographically diverse. The routing overlay network is formed on top of the hash table by each peer discovering and exchanging state information consisting of a list of leaf nodes, a neighbourhood list, and a routing table. The leaf node list consists of the L/2 closest peers by node ID in each direction around the circle.

In addition to the leaf nodes there is also the neighbourhood list. This represents the M closest peers in terms of the routing metric. Although it is not used directly in the routing algorithm, the neighbourhood list is used for maintaining locality principles in the routing table.

Finally there is the routing table itself. It contains one entry for each address block assigned to it. To form the address blocks, the 128-bit key is divided up into digits with each digit being b bits long, yielding a numbering system with base 2b. This partitions the addresses into distinct levels from the viewpoint of the client, with level 0 representing a zero-digit common prefix between two addresses, level 1 a one-digit common prefix, and so on. The routing table contains the address of the closest known peer for each possible digit at each address level, except for the digit that belongs to the peer itself at that level. This results in the storage of 2 power b-1 contacts per level, with the number of levels scaling as (log N)/b.

Routing:

A packet can be routed to any address in the key space whether there is a peer with that node ID or not. The packet is routed toward its proper place on the circular ring and the peer whose node ID is closest to the desired destination will receive the packet. Whenever a peer receives a packet to route or wants to send a packet it first examines its leaf set and routes directly to the correct node if one is found. If this fails, the peer next consults its routing table with the goal of finding the address of a node which shares a longer prefix with the destination address than the peer itself. If the peer does not have any contacts with a longer prefix or the contact has died it will pick a peer from its contact list with the same length prefix whose node ID is numerically closer to the destination and send the packet to that peer. Since the number of correct digits in the address always either increases or stays the same — and if it stays the same the distance between the packet and its destination grows smaller — the routing protocol converges.

Chord and Pastry protocols:

1.More structured than Gnutella and Naspers

2.BlackBox lookup algorithm

3.Churn Handling can get complex

4.O(log N) memory and lookup cost

3.KELIPS P2P System:

constant lookup cost to DHT
* concept of affinity groups, k affinity groups where k = sqrt (N)
* Each node hashed to a group (hash mod k), where hash can be SHA-1
* Node's neighbours
- All other nodes in its own affinity group (on an average sqrt(N)
- One contact node per foreign affinity group (k-1 such foreign affinity groups)
- Total = 2 sqrt(N) - 1

Kelips Files and Metadata

* File can be stored at any nodes
* Decouple file replication/location (outside kelips) from file querying (inside kelips)
* Each filename hashed to a group
-- All nodes in the group replicate pointer information i.e. <filename, file location>
-- Affinity group doesn't store files

Kelips lookups

--Find file affinity group by hashing the filename
-- From the node's(querying node) neighbours list go to contact for the file affinity group (just one hop)
-- Failing that try another of your neighbours to find a contact (maximum of two hops)
-- Note lookups are one hope or fewer
-- Memory cost is O(sqrt(N) : 1.93 MB for 100K nodes, 10M files
-- Fits in RAM of most workstations/laptops today (COTS machine)

Kelips Soft State:

Membership lists

--Gossip-based membership, within each affinity group and also across affinity groups
--O(log N) dissemination time
--Unlike Chord and Pastry which have to look at other successors/predecessors in the even its immediate successor failure, Kelips lookup has lot more options in asking nodes within its own

affinity group for the required affinity group OR it can ask node in a different affinity group for the required affinity group

File metadata

--needs to be periodically refreshed from source node in gossip style, otherwise the file metadata information times out and eventually gets removed from the system. The advantage is when a file is deleted, it doesn't need to be explicitly from all the nodes in the affinity group

* Slightly higher memory and background bandwidth cost, but lookup cost is O(1)