CHORD

A Look-Up Service

P2P Computing

Md. Sahil, Priyank Lohariwal, Devesh Jalan
001710501029, 001710501055, 001710501071

BCSE IV – 2nd Semester
Distributed Computing

Introduction

In today's services, large amounts of data, e.g., video and audio content, subscriber management and accounting data, have to be stored and made available to applications. One example area for this are mobile networks, where user data is stored and accessed very often during the time a user is booked in. The number of database accesses is potentially very large and therefore poses a challenge to the system, which has to show a high availability and performance while being at the same time resource-efficient and scalable. These very large and still rapidly growing applications attest to a new era for the design and deployment of distributed systems. In particular, they reflect what the major challenges are today for designing and implementing distributed systems: scalability, flexibility and instant deployment.

In a more appropriate manner, this lookup problem can be defined as follows:
Some node A wants to store a data item D in the distributed system. D may be some data item, some location of some bigger content, or coordination data, e.g., the current state of A, or its current IP address, etc. Then, we assume some node B wants to retrieve data item D later. The problem then simplifies to following questions:

- Where should node A store data item D?
- How do other nodes, e.g., node b, discover the location of D?
- How can the distributed system be organized to ensure scalability and efficiency?

Conventionally, a number of lookup architectures have been introduced to address the lookup problem. The goal of all of them was to establish a Peer-to-Peer paradigm that relies on the design and implementation of distributed systems where each system has almost the same functionality and responsibility. By definition, these systems have to coordinate themselves in a distributed manner without centralized control and without the use of centralized services. Thus, scalability must be an inherent property of Peer-to-Peer systems.

The first generation of P2P distributed systems had a central server based approach. These central servers were used to store the location of each peer of the system. A node connects with the central server and queries for a data item. The server responds with the list of all the possible locations (IP Addresses) that have the data and are online. The node then queries the other nodes provided by the server and retrieves the data. The most popular system based on this approach was ***Napster.***

The second generation involved a flooding search technique. A full distributed system was developed requiring no central server (which could be a single point of failure) for indexing and searching. An overlay network was used for searching. Every node had a few peers named *neighbours.* The choice of neighbours were completely random and completely unrelated to their individual underlying network. A node floods the search query to every node in its underlying network which in turn floods in their own respective networks. This process continued until valid locations were found or nodes got exhausted. The querying node then retrieves the data directly from the peer holding the data. The most popular system based on this approach was ***Gnutella.***

Distributed Hash Tables

Both central servers and flooding-based searching exhibit crucial bottlenecks that contradict the targeted scalability of P2P systems. Indeed, central servers are disqualified with a linear time complexity for storage. Flooding-based approaches avoid the management of references on other nodes and, therefore, they require a costly breadth first search which leads to scalability problems in terms of the communication overhead.

The main goal of P2P systems are as follows:

- Harnessing computational and storage power of millions of nodes spread across the globe.
- Scalability - an architecture that can be scaled to the global level.
- Fault tolerance
- System stability: Nodes can arrive and depart the system on their own.

In order to address these requirements, Distributed Hash tables were introduced in the early 2000's. Distributed hash tables use a more structured key-based routing in order to attain both the decentralization of Gnutella, and the efficiency and guaranteed results of Napster. DHT is a distributed system that provides a lookup service similar to a hash table: key-value pairs are stored in a DHT, and any participating node can efficiently retrieve the value associated with a given key. The main advantage of a DHT is that nodes can be added or removed with minimum work around re-distributing keys. Keys are unique identifiers which map to particular values, which in turn can be anything from addresses, to documents, to arbitrary data. Responsibility for maintaining the mapping from keys to values is distributed among the nodes, in such a way that a change in the set of participants causes a minimal amount of disruption. This allows a DHT to scale to extremely large numbers of nodes and to handle continual node arrivals, departures, and failures. The nodes in a DHT are connected together through an overlay network in which neighbouring nodes are connected. This network allows the nodes to find any given key in the key-space.

The major challenge of DHT was load-balancing. When a node arrives or departs from the system, all of the data needs to be rehashed accordingly to accommodate this change in node population. To solve this issue, consistent hashing was introduced.

Consistent Hashing

Consistent hashing is a distributed hashing scheme that operates independently of the number of nodes in a distributed hash table by assigning them a position on an abstract circle or hash ring. This allows nodes to scale without affecting the overall system.

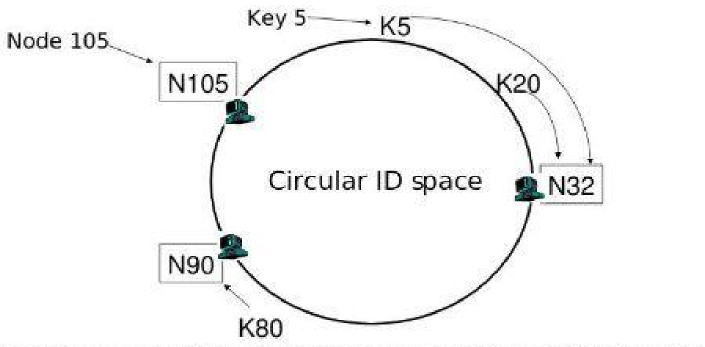
Nodes are individual peers in the network. Each node is provided with an Id. The Ids are mapped to a large circular space. All the keys are also mapped to the same circular space. The basic idea is to assign each key to the nearest node available. This provides a mechanism of maintaining load in the network by distributing keys among every node available on the network. When a node joins or exits the network, the keys are remapped to the available nodes. This mapping is done only for the keys that were associated with the arriving or departed node. Hence, the number of servers impacted by this transition in the number of nodes is minimized, preventing possible downtime and performance issues.

Chord

Chord is a protocol and algorithm for a peer-to-peer distributed hash table. It is one of the four original distributed hash table protocols, along with CAN, Tapestry, and Pastry. It was introduced in 2001 by Ion Stoica, Robert Morris, David Karger, Frans Kaashoek, and Hari Balakrishnan, and was developed at MIT.

Terminology

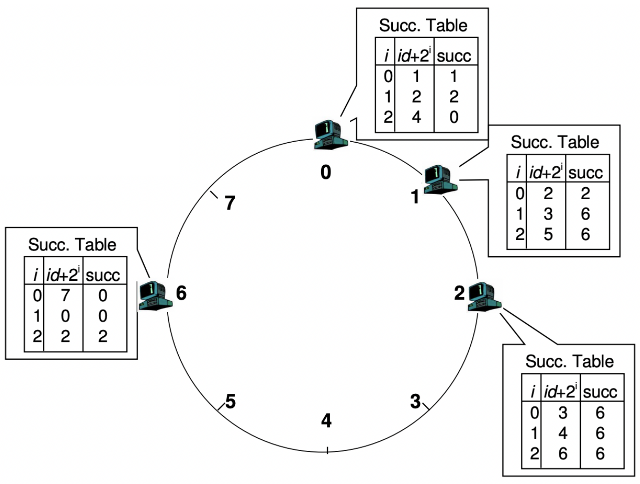
Chord uses the concept of consistent hashing for maintaining load balancing features. The consistent hash function assigns each node and key an m-bit identifier using SHA-1 as a base hash function. A node's identifier is chosen by hashing the node's IP address, while a key identifier is produced by hashing the key. Identifiers are ordered on an identifier circle modulo 2^m. The key ***k*** is assigned to the first node that is equal or follows ***k*** in the identifier space. This node is formally called the **successor node** of key *k* and denoted by **successor(k).**



A key is stored at its successor: node with next higher ID

Each node **n** maintains a routing table called the **Finger table.** The **i**th entry in the finger table contains the identity of the first node **s** that succeeds **n** by at least $\mathbf{2^{(i-1)}}$ on the identifier circle. This **ith** entry of the finger table is called the **i^th^ finger** of **n.** This table contains the chord identifier and the IP address of the relevant nodes.

This scheme has two important characteristics. First, each node stores information about only a small number of other nodes, and knows more about nodes closely following it on the identifier circle than about nodes farther away. Second, a node's finger table generally does not contain enough information to directly determine the successor of an arbitrary key k.



Succ. Table

i	id+2^i	succ
0	1	1
1	2	2
2	4	0

Succ. Table

i	id+2^i	succ
0	2	2
1	3	6
2	5	6

Succ. Table

i	id+2^i	succ
0	7	0
1	0	0
2	2	2

Succ. Table

i	id+2^i	succ
0	3	6
1	4	6
2	6	6

Operations

- **Find Successor**
  Ask node **n** to find the successor of id **k**

```
// ask node n to find the successor of id
n.find_successor(id)
    if (id ∈ (n,successor])
        return successor;
    else
       n' = closest_preceding_node(id);
       return n'.find_successor(id);

// search the local table for the highest predecessor of id
n.closest_preceding_node(id)
for i = m downto 1
    if (finger[i] ∈ (n,id))
    return finger[i];
return n;
```

- **Join**
  When node **n** first starts, it calls **n.join(n')**, where **n'** is any known Chord node, or **n.create** to create a new Chord network. The **join** function asks **n'** to find the immediate successor of **n**. By itself, **join** does not make the rest of the network aware of **n**.

```
// create a new Chord ring.
n.create()
    predecessor = nil;
    successor = n;

// join a Chord ring containing node n'.
n.join(n')
    predecessor = nil;
    successor = n'.find_successor(n);
```

- **Stabilize**
  Every node performs **stabilize** periodically to learn about newly joined nodes. Each time node ***n*** performs stabilizations, it asks its successor for the successor's predecessor ***p***, and decides whether ***p*** should be ***n***'s successor instead. This would be the case if node ***p*** recently joined the system. In addition, **stabilize** notifies node **n**'s successor of **n**'s existence, giving the successor the chance to change its predecessor to **n**. The successor does this only if it knows of no closer predecessor than **n**.

```
// called periodically. verifies n's immediate
// successor, and tells the successor about n.
n.stabilize()
    x = successor.predecessor;
    if (x ∈ (n,successor))
        successor = x;
    successor.notify(n);

// n' thinks it might be our predecessor.
n.notify(n')
    if (predecessor is nil or n' ∈ (predecessor, n))
    predecessor = n';
```

- **Fix Fingers**
  Each node periodically calls f**ix_fingers** to make sure its finger table entries are correct. This is how new nodes initialize their finger tables, and it is how existing nodes incorporate new nodes into their finger tables. Each node also runs **check_predecessor** periodically, to clear the node's predecessor pointer if **n.predecessor** has failed; this allows it to accept a new predecessor in **notify**.

```
// called periodically. refreshes finger table entries.
// next stores the index of the next finger to fix.
n.fix_fingers()
    next = next + 1 ;
    if (next > m)
        next = 1 ;
    finger[next] = find_successor(n + $2^{next-1}$);


// called periodically. checks whether the predecessor has failed.
n.check_predecessor()
    if (predecessor has failed)
        predecessor = nil;
```

Node Exit

Consider there are 3 nodes in succession

A -> B -> C.

If B fails or exits voluntarily,

A -> _ -> C.

Now the successor of **A** and predecessor of **C** have changed. This is said to be a node failure and following steps helps in again stabilizing the chord network.

- The **fix_fingers** operation on **A** updates its finger table thereby updating the successor of **A** to **C**.
- The **check_predecessor** function runs on **C** that detects the failure of its predecessor and sets its **predecessor** to **nil**.
- The **stabilize** operation on **A** notifies **C** that **A** is its predecessor thereby updating the **predecessor** of **C**.

Properties of Chord

- **Autonomy and decentralization** - the nodes collectively form the system without any central coordination.

- **Fault Tolerance** - The system is able to maintain the correct state even when multiple nodes join or leave concurrently by the method of stabilization.

- **Load Balancing** - The data items are evenly distributed among all the nodes of the network via consistent hashing. Additionally, this ensures that only **O(1/N)** data items need to be remapped in every node arrival or departure where **N** is the number of nodes in the network.

- **Scalability** - The time required for each query done in the chord network depends on the number of forwardings needed to reach the node that stored the index. Since each node has finger entries at power of two intervals around the identifier circle, each node can forward a query at least halfway along the remaining distance between the node and the target identifier. With high probability, the number of nodes that must be contacted to find a successor in a **N**-node network is **O(logN)**.

Conclusion

Determining where the data is stored is a challenging problem for many distributed peer to peer applications. The Chord protocol solves this challenge in a decentralized manner. Given a key it can determine the node responsible for the key's value very efficiently. In an N-node Chord network, each node is responsible for storing the routing information of only O(logN) other nodes, and lookups are resolved through O(logN) inter-node communications.

Chord is simple, efficient, robust, and handles node failures, exists and joins efficiently. Chord also scales well with increase in number of nodes and can recover from large numbers of simultaneous node failures and joins.

All these features make Chord an attractive protocol for peer to peer applications.