Lecture 3 — February 4, 2002

Lecturer: John Byers *BOSTON UNIVERSITY* *Scribe: Manish Sharma*

Today's session was divided into two halves. During the first half we discussed the **CHORD** paper [3] and in the second half the building blocks used in the **Consistent Hashing** [1] paper like **Occupancy** and **Chernoff Bounds**. The notes are also divided into two sections. The first section discusses the important aspects of the CHORD paper and the second section discusses the Consistent Hashing related concepts.

3.1 CHORD

The motivation for CHORD comes from a Napster like application whereby there is a large set of documents that are shared and the network is comprised of nodes which both request and serve these documents. The challenge is that such peer-to-peer applications should efficiently locate the node that stores a particular data item in a scalable manner. *Chord*, a distributed lookup protocol, addresses this problem. it provided support for just one operation: given a key, it maps the key onto a node. The key could be an identifier for a document. *Chord* indexes by routing requests through one or more peers and responds by returning a key-value pair.

3.1.1 Approaches

There are 3 approaches which are used for indexing.

1. **Centralized Mapping:** The key value pairs are stored in a central server. For example, in Napster a central server stores the index of all the files available within the Napster user community. To retrieve a file, a user queries this central server using the desired file's well known name and obtains the IP address of a user machine storing the requested file. The drawback here is that is both expensive (to scale the central directory, central server has to maintain $O(N)$ state) and vulnerable (since there is a central point of failure).

2. **Flooding Mapping:** Requests for a file are flooded with a certain scope. Flooding on every request is not scalable and, because the flooding has to be curtailed at some point, may fail to find content that is actually in the system. Gnutella is one such example. Though it is robust, in the worst case it can have $O(N)$ messages per lookup.

3. **Routing/Gradient Ascent Method:** Nodes have some notion of distance measurei, they know how far away they are from the content and how to go "forward" looking for the content, (e.g.: Freenet, Globe).

3.1.2 Challenges

Ideally, there should be no infrastructure beyond the nodes and the nodes should be completely self organizing. Such a system poses the following challenges:

- Avoid centralized point of failure.
- Define a key "nearness" metric.
- Keep a small amount of state per node. If there are N servers then it should maintain an $O(\log N)$ state or less.
- While routing messages in order to retrieve information about a key from any other node, it should keep a small maximum hop count. This bound is also $O(\log N)$.

- It should stay robust primarily to node failures.

Chord is based on Routing/Gradient Ascent method. If there are N servers, Chord maintains an $O(\log N)$ state and the messaging complexity is also $O(\log N)$.

3.1.3 Hashing

To meet the challenges posed above, the idea comes from Consistent Hashing paper [1]. The basic idea is to hash keys into a certain space and hash peer-id's also into the same space. The consistent hash function assigns each node and key an m-bit identifier using a base hash function such as SHA-1. SHA-1 is used because it makes the protocol deterministic. It is also believed that it is hard to invert the SHA-1 function. A node's identifier is chosen by hashing the node's IP address, while a key identifer is produced by hashing the key.

$hash(string)$ for the key
$hash(IP)$ for the nodes.

However, they use a "variant" of Consistent Hashing. Identifiers are ordered in an *identifier circle* modulo 2^m. Keys and nodes are roughly uniformly distributed around the circle. Keys are bound to the nodes by a notion of successor. Key k is assigned to the first node whose identifer is equal to or follows (the identifier of) k in the identifier space. This node is called the *successor node* of key k, denoted by *successor(k)*. If identifiers are represented as a circle of numbers from 0 to 2^m - 1, then *successor(k)* is the first node clockwise from k. Figure 3.1 shown below illustrates the concept of successors and a chord ring.

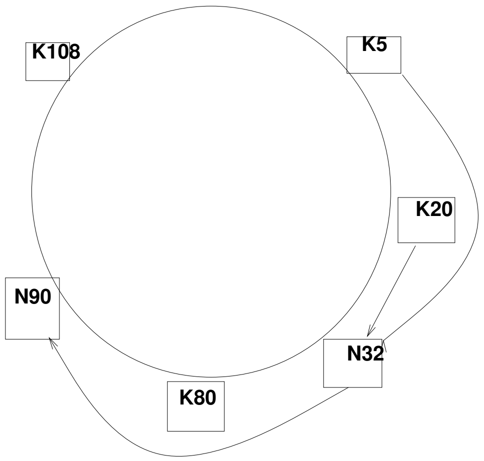

Figure 3.1. A Chord Ring

3.1.4 Key Lookup

Queries for a given identifier can be passed around the circle via these successive pointers until they first encounter a node which succeeds the identifier; this is the node that the query maps to. A portion of the Chord protocol maintains these successor pointers, thus ensuring that all lookups are resolved correctly. However this does not provide low messaging complexity as it may require traversal of all N nodes to find the appropriate mapping. A basic lookup algorithms is shown in Figure 3.3 and a basic key lookup is demonstrated in Figure 3.2.

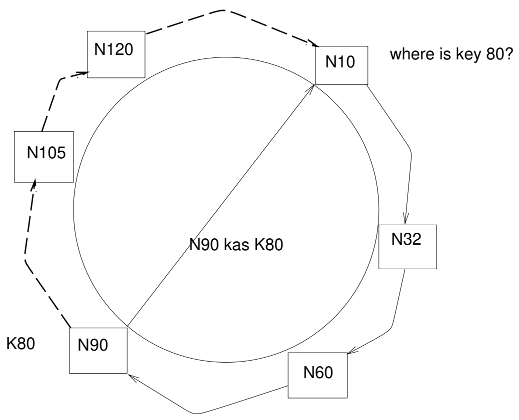

Figure 3.2. A Basic key Lookup

To accelerate the lookup process, Chord maintains finger tables. Each node, n, maintains a routing table with (at most) m entries, called the finger table. 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, i.e., s = successor(n + 2^{i-1}), where $1 \leq i \leq m$ (and all arithmetic is modulo 2^m). Node s is called the i^{th} finger of node n. A finger table entry includes both the Chord identifier and the IP address (and port number) of the relevant node. Finger tables allows $\log_2 N$ lookups. Figure 3.3 shows a finger take lookup algorithm.

```
Basic Lookup of (key k, node N)
n = my_successor
if (my_id < n < key_id)
 lookup of k at n
else  /* if my_id < key_id < n */
 return my_successor
```

(a)Basic Key Lookup

```
Lookup with Fingers
Lookup(my_id, key_id)
 look in local finger table for
          highest node n s.t. my_id < n < key_id
 if n exists
  call Lookup(id) on node n //next hop
 else
  return my_successor  //done
```

(b) Finger table Key Lookup

Figure 3.3. key Lookups

A finger table lookup, shown in Figure 3.4 shows lookup of key 19 made at node 32. In this case node 20 is the owner of key 19. The lookup takes $O(\log N)$ hops.

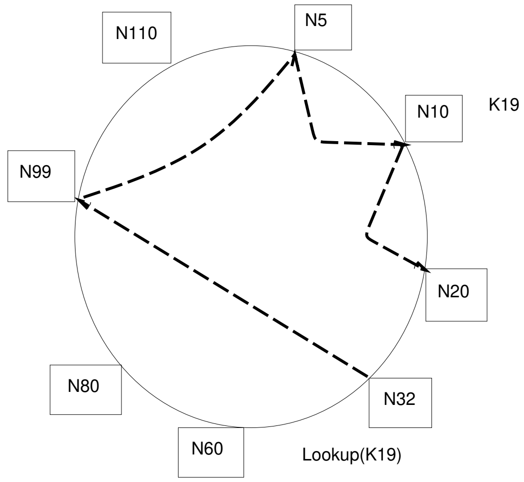

Figure 3.4. A key Lookup with finger tables

3.1.5 Insertion

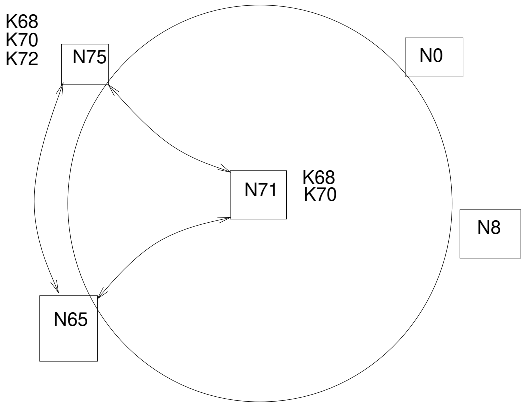

Figure 3.5. A Node Insertion

In a dynamic network, nodes can join and eave at any time. The main challenge in implementing these operations is preserving the ability to locate every key in the network. To do a successful insertion into the ring, the following steps have to be performed by a new node.

1. Hash itself and find successor.
2. Attach to its successor.
3. Take over key-ownership.

4. Find predecessor and splice in.

This operation is also explained in Figure 3.5, where nodes 65 and 75 already exist. Node 75 is the successor of node 65 and also the owner of the keys 68, 70 and 72. Node 71 is the newly inserted node. It has taken over the ownership of the keys 68 and 70 from node 75. node 75 now becomes the successor of node 71 and node 65 becomes the predecessor of node 71. Hence it can be clearly seen that the nodes need to maintain a doubly-linked list, one end pointing to the predecessor and the other link pointing to the successor.

3.1.6 Failures and Replication

When a node n fails, nodes whose finger tables include n must find n's successor. In addition, the failure of n must also not be allowed to disrupt queries that are in progress as the system is re-stabilizing. To help achieve this, each Chord node maintains a "successor list" of its r nearest successors on the Chord ring.
If every node fails with a probability $\frac{1}{2}$ then,

$$\Pr[\text{all } r \text{ successors fail if 50\% of nodes fail}] = \frac{1}{2^r}$$

$$\Pr[\text{none of the nodes are disconnected}] = (1 - \frac{1}{2^r})^n$$

Now, $(1 - \frac{1}{2^r})^n \approx \exp^{\frac{-n}{2^r}}$

And, $\exp^{\frac{-n}{2^r}} < 1 - \frac{n}{2^r}$ using the inequality $\exp^{-y} < (1 - y)$

Therefore when $r = 2\log n$,

$$\Pr[\text{none of the nodes are disconnected}] = 1 - \frac{1}{n}$$

Some relevant plots relevant to the performance of CHORD with respect to failures and replication can be found at [3]. This concludes our discussion of Chord. Now we study the basic problems underlying the Consistent Hashing [2] analysis.

3.2 Occupancy Problem

Occupancy problem [2] deals with pairing of objects. The basic occupancy problem is about placing m balls into n bins. This problem has a vast number of applications. The balls could be viewed as jobs and the bins could be viewed as machines. The jobs have to be ditributed amongst the machines in way such that none of the machines gets more than its fair share. From the Consistent hashing point of view, the balls can be viewed as objects or files and the bins can be viewed as caches. In this problem, there are m indistinguishable objects ("balls") being randomly assigned to one of n distinct classes ("bins"). Each ball is placed in a bin chosen independently and uniformly at random. We are interested in questions such as: "What is the maximum number of balls in any bin? and what is the expected number of bins with k balls in them?"

One key tool that is used for this analysis is: *the probability of the union of events is no more than the sum of their probabilities.*
If $E_j(k)$ is the event that bin j has at least k balls then,

$$Pr[\bigcup_{j=1}^{n} E_j(k)] \leq \sum_{j=1}^{n} Pr[E_j(k)]$$

This simplifies the hard problem of computing the union of events to computing the sum of events (with some loss of precision)

Now let is try to calculate the probability that no bin receives more than k balls for a suitably chosen k. Let $E_j(k)$ denote the event that bin j has k or more balls in it.
Since,

$$Pr[E_1(k)] = Pr[E_2(k)] = ... \leq n.Pr[E_1(k)]$$

we concentrate on calculating $Pr[E_1(k)]$.
For simplification purpose, we assume $m = n$.

Now, probability that bin 1 gets exactly i balls is

$$\binom{n}{i}(\tfrac{1}{n})^i(1-\tfrac{1}{n})^{n-i} \leq \binom{n}{i}(\tfrac{1}{n})^i \leq (\tfrac{ne}{i})^i(\tfrac{1}{n})^i = (\tfrac{e}{i})^i$$

the second inequality comes from an upper bound of binomial coefficients. Therefore,

$$Pr[E_1(k)] \leq \sum_{j=k}^{m} (\tfrac{e}{j})^j$$

Therefore,

$$\begin{aligned} Pr[E_1(k)] &\leq \sum_{j=k}^{m} (\tfrac{e}{j})^j \\ &< (\tfrac{e}{k})^k(1 + (\tfrac{e}{k}) + (\tfrac{e}{k})^2 + ...) \\ &\leq (\tfrac{e}{k})^k(\tfrac{1}{1-\frac{e}{k}}) \end{aligned}$$

The latter inequality holding because $[1 + x + x^2 + ... = \frac{1}{1-x}]$

Lemma 1. *When* $k = \frac{e \ln n}{\ln \ln n}$, $Pr[E_1(k)] \leq \frac{1}{n^2}$

To ensure that all bins have less than k balls, we use the union bound discussed earlier. Therefore,

$$Pr[\bigcup_{j=1}^{n} E_j(k)] \leq \sum_{j=1}^{n} Pr[E_j(k)] \leq \frac{1}{n}$$

Proof: One possible way to prove Lemma 1 could be the following

$$k = \frac{e \ln n}{\ln \ln n}$$

We have to prove that,

$$(\tfrac{e}{k})^k(\tfrac{1}{1-\frac{e}{k}}) \leq n^{-2}$$

Taking ln of both sides,

We need to show

$$k \ln(\tfrac{e}{k}) - \ln(1 - \tfrac{e}{k}) \leq -2 \ln n$$

Ignoring the second term on the left hand side of the inequality since it will always be positive and substituting the value of k in the inequality, the left hand side becomes

$$\frac{e \ln n}{\ln \ln n} \ln(\frac{\ln \ln n}{\ln n})$$

$$= \frac{e \ln n}{\ln \ln n}(\ln \ln \ln n - \ln \ln n)$$

$$\leq \frac{e \ln n}{\ln \ln n}((\tfrac{e-2}{e}) \ln \ln n - \ln \ln n)$$

$$= \frac{e \ln n}{\ln \ln n}\left(\frac{-2}{e} \ln \ln n\right)$$

$$= -2 \ln n$$

□

3.3 Chernoff Bounds

Let $X_1, ..., X_n$ be independent and identically distributed Bernoulli trials such that, for $1 \leq i \leq n$, $Pr[X_i = 1] = p$ and $Pr[X_i = 0] = 1 - p$. Let X = $\sum_{i=1}^{n} X_i$; then X is said to have the binomial distribution. More generally, let $X_1, ..., X_n$ be independent coin trials such that, for $1 \leq i \leq n$, $Pr[X_i = 1] = p_i$ and $Pr[X_i = 0] = 1 - p_i$. Such coin tosses are referred to as Poisson trials. In this discussion, the random variable X = $\sum_{i=1}^{n} X_i$, where X_i are Poisson trials.
The expectation of X is $\mu = \sum_{i=1}^{n} p_i$. We want to know what are the bounds by which X deviates from its expected value. Close approximate answers to such questions come from a technique known as Chernoff bound [2].

Theorem 1. *As defined earlier, for* $X = \sum_{i=1}^{n} X_i, \mu = E[X] = \sum_{i=1}^{n} p_i$, *and* $0 < \delta \leq 1$,

$$Pr[X < (1-\delta)\mu] < \exp^{\left(\frac{-\mu\delta^2}{2}\right)}$$

Proof:

$$Pr[X < (1-\delta)\mu] = Pr[-X < -(1-\delta)\mu]$$
$$= Pr[exp(-tX) > exp(-t((1-\delta)\mu)]$$

for any positive real t.
According to Markov Inequality: If Y is a random variable assuming only non-negative values, the for all positive real t's

$$Pr[Y \geq t] \leq \frac{E[Y]}{t}$$

Applying the Markov Inequality, we obtain that

$$Pr[X < (1-\delta)\mu] < \frac{\prod_{i=1}^{n} E[\exp[-tX_i)]}{\exp(-t(1-\delta)\mu)}$$

Computing $E[exp(-tX_i)]$,

$$Pr[X < (1-\delta)\mu] < \frac{\exp(\mu(e^{-t}-1))}{\exp(-t(1-\delta)\mu)}$$

If $t = \ln(\frac{1}{(1-\delta)})$,

$$Pr[X < (1-\delta)\mu] < \left[\frac{e^{-\delta}}{(1-\delta)^{1-\delta}}\right]$$

This is simplified by noting that for $\delta \in (0, 1]$,

$$(1-\delta)^{(1-\delta)} > \exp(-\delta + \frac{\delta^2}{2})$$

using the Mclaurin expansion for $\ln(1-\delta)$, it yields the desired result. □

Bibliography

[1] David Karger, Eric Lehman, Tom Leighton, Matthew Levine, Daniel Lewin, and Rina Panigrahy. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. In *Proceedings of the 29th Annual ACM Symposium on Theory of Computing*, May 1997.

[2] Rajeev Motwani and Prabhakar Raghavan. *Randomized Algorithms*, pages 43–44, 68, 70. Cambridge University Press, 1995.

[3] Ion Stoica, Robert Morris, David karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of SIGCOMM 2001*, March 2001. http://www.pdos.lcs.mit.edu/ rtm/sigcomm01.ppt.