Structured P2P Networks

Magnus Jensen

1. Structured P2P Networks

- Structured P2P Networks
    - Motivation
    - Structured networks
    - Chord
    - Pastry
        - Routing table
        - Locality
    - Kademlia
        - K-Buckets
        - Updating K-Buckets
        - Parallelism
        - Joining

1. Motivation

- Routing information must be distributed – no central index
- How is the routing information created and maintained?
- How are peers inserted into the network? How do they leave? How are resources added?

2. Structured networks

Structured P2P networks maintains a Distributed Hash Table and allows each peer to be responsible for a specific part of the content in the network. These networks use hash functions and assign values to every content and every peer in the network and then follow a global protocol in determining which peer is responsible for which content. This way, whenever a peer wants to search for some data, it uses the global protocol to determine the peer(s) responsible for the data and then directs the search towards the responsible peer(s).

3. Chord

Lets start by talking about *Chord*. *Chord* is a protocol to manage distributed hash tables in a peer-to-peer network.

> Chord: Distributed hash table system in P2P

A hash table is a datastruktur, consisting of key/value pairs, where a value is found by looking up a key.

The protocol manages this hash table distributed, meaning that several nodes, which all belong to the system, will contain part of the combined table. *Chord* specifies how keys are given to nodes, and how a node can find a value for a given key, by first locating the node where it belong.

Chord seeks to do this, given the following goals:

- **Goals**
- `Load balance` The keys should be balanced evenly among the nodes
- `Scaleable` A lookup flows as the log of the number of nodes grow; so even large system are feasible.
- `Availability` Chord automatically adjusts its internal tables to reflect newly joined nodes as well as node failures.

Great, so lets start talking about how *chord* really does this. Every node in the system gets an ID. Such id is created by hashing the IP address and truncating the result to _m_-bits. Originally, SHA-1 was used.

> ID: SHA-1(ip)[m:]

Then, the nodes are placed on a circle with 2^m spots.

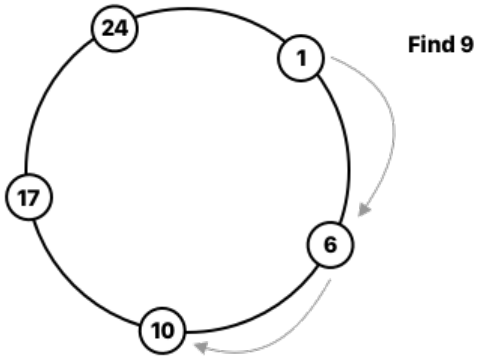
> mapped to circle of 2^m spots

Here it is clear, that we wanna use an m that is sufficient large enough to handle the expected amount of nodes.

When ever a file needs to be stored in the network, it gets given an ID using the same method:

> ID file: SHA-1(file)[m:]

And is then placed at the first node with an id equal or larger then the files.



We have now created a simple and linear network. When a node *n* joins the network, it gets allocated keys equal to or smaller than itself, from the node which is clockwise to it.

Again, a node *n* is clockwise to a node *m* if it is the first which satisfies:

> Clockwise: $id_n \geq id_m$

Which is also called its *successor*. And if a node leaves, all of it keyes are given to its successor.

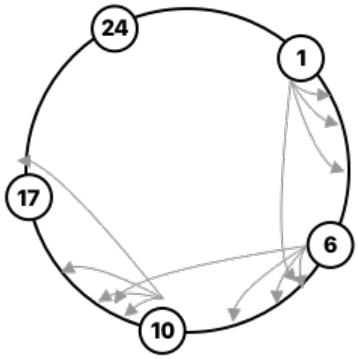
Since every node only knows about its successor, a lookup of key *k* will simply be to ask the successor, and wait for response.

This means, that lookups are O(n), but we can do better in chord! You see, each node contains what is called a *finger table*. It is a table with *m* identifiers following the node.

In the range `1 <= i <= m`, the table of node *n* will reference the successors of the identifiers given:

> $n + 2^{i-1}$

Well it is simply, the sequence: 1, 2, 4, 8, 16, 32 ... and so on.



Now at looktime time, if the successor does not contain the file, the lookup will be propagated to the largest node in the finger table, which has an id smaller than that of the file.

Thereby, now we have lookup as an `O(log(n))` operation, and to prevent failures, each node will keep its successors table, to be able to rebuild the system.

But does it contain any problems? Well sure, it is still a bit too simple and does not consider locality or strength of peers.

4. Pastry

Lets look at another p2p system, Pastry, and I'll explain how routing is done here. As with Chord, it assign ids to node, but every node now also knows its predecessor.

> - **Pastry**
> - Assigns 128bit hash IDs in a ring
> - Each node has Leaf Set
>   - Nummerical Successors and predecessors ~~plural!~~

~~I only talk about routing in pastry, hence I do not speak about a neighbour table, which I did not really understand, but is used to keep the network alive by periodically check the table for liveness.~~

4.1. Routing table

Now, I chord every node ended up with a Finger Table. That is not the case here, but every node do have a Routing Table, which is based on Prefix Match

> - routing table -> prefix matching -> O(log(n))!

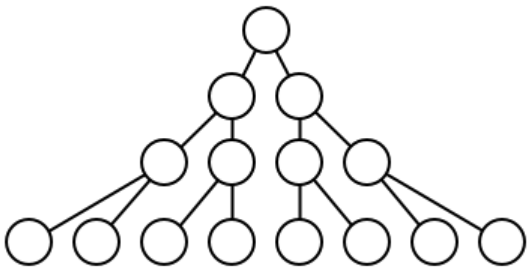
Say that we have a peer with ID *1010*, it will then maintain neighbor peer with an id matching the following sequence

- *
- 1*
- 10*
- 101*

As we can see, Chord and Pastry are much alike, also functionality wise. At lookup time, if the id falls within the leafset, the job is done, otherwise, the routing table will be used, and lookup will be propagated to the entry with the largest matching prefix.

4.2. Locality

So, does Pastry somehow take into account the locality of nodes? Well if we assume that the 128 bit-space is uniformly distributed, I would like to draw references to a typical tree structure.



As we see in the tree, the further down we go, the more nodes are there in a row. Lets imagine these as neighbours, then we have the potential need for larger and larger hops as we go down.

Therefor, in the routing table, the smaller a prefix for a routing is, the smaller jump. Therefor, in Pastry, we expect in the beginning of a query to start with smaller jumps, and as we locate the destination, we should be making larger jumps.

5. Kademlia

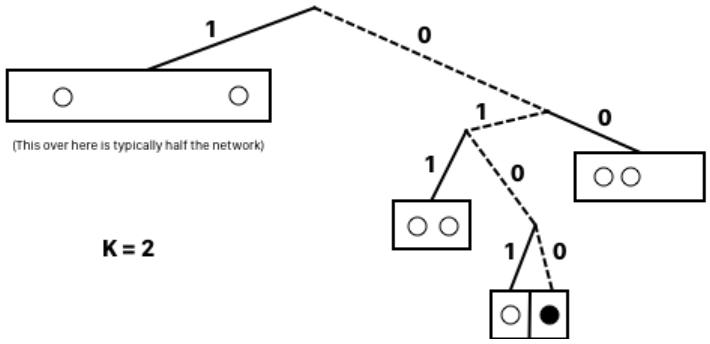
Great, we have briefly discussed two different systems now, which both have their problems, the latter, *Pastry* is seen to have a rather complicated routing algorithm.

Next we will be looking a *Kademlia*, which had a breakthrough I networking, by using the XOR operation to minimize the internode messaging. Another advantage is also, the the Kademlia system prefers long living nodes, since it expects those to also be alive next time they are needed.

Kademlia is built like a binary tree structure - which in itself is the routing table. All identifiers er 160 bit long, found randomly or by SHA1. Each node is then a leaf on the tree, positioned as the shortest unique prefix of its id.

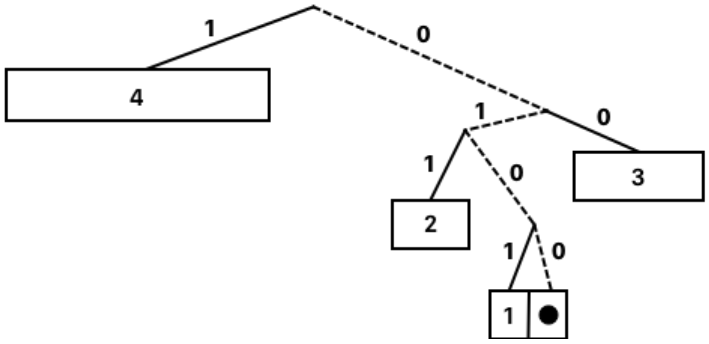
5.1. K-Buckets

Each node contains a list of what is called *k-buckets*. A node will have a *k-bucket* for every bit up from its unique prefix. That is, going up in the tree, from the node itself, each time it can go down along a different path, that tree will be called a bucket, and in that (sub)tree, the node will know *k* of the given nodes in it. If it is known that nodes that have been connected for a long time in a network will probably remain connected for a long time in the future. Because of this statistical distribution, Kademlia selects long connected nodes to remain stored in the k-buckets.



That means, that we know our neighbour hood very well, but in the other half of the system, we only know *k* nodes.

The further away a bucket is located from a node, the higher the number is given a bucket. That, goes up the tree to find buckets, that are given a name starting from 1 and increasing.



Why is that important you ask? Well, it is at lookup time. Say *node n* is searching for *file f*. Then it will XOR $ID_n \bigoplus ID_f$, and the location of the most significant bit, tells which *k-bucket* to contact. Contacting a node in that bucket, will get node *n* closer to the destination. The node in the bucket will return *k* closest nodes it knows of, which will then be contacted.

~~Det kan maksimalt være 160 k-buckets~~

And by means of these *k-buckets* and XOR operation, the network implements *find-value* and *find-node*. But this is an, relatively, awful lot of communication back and forth ... cant we utilise it somehow? Well, im glad you asked!

5.2. Updating K-Buckets

The essence of Kademlia and why it is better than both Pastry and Chord, is due, partly, to the updates of the *k-bucket*.

Upon receiving a new message from a node, the given *k-bucket* I belongs to is identified, of course by XORing the identifier with the identifier of itself. Then one of the following thins happen.

> - `If in bucket, move to tail` If the node is already in the bucket, move it to the tail.
> - `Bucket not full, insert at tail` If the bucket is not filled up, move the node the the tail of the bucket
> - `Bucket full, least recently unresponsive, replace at tail` but what if the bucket is full? Well, we contact the least seen node, and if it appears unresponsive, we replace it with the new node at the tail
> - `} else { bail out }` and if non of these things are possible, we simple bail out and ignore storing the node

In this way, the different buckets are populated, and maintained to prefer older nodes. An implementation detail, could be to keep new members in a cache if there is no room to spare in the buckets.

5.3. Parallelism

Okay great, so lets boil it down.

When a node *n* contacts some nodes from a bucket, it happens in parralism. That means, that when the node *n* receives another nodes *k-bucket* it can choose to contact those which it got first. To keep latency down!

Because of this, and those structures which we have been discussing, we are ensured locality and the strength of the nodes.

5.4. Joining

Lets briefly discuss joining the network. Well because that is a pretty important part of making it all work.

> - `Calc ID` so, you start by calculating your id
> - `locate peer in network` then you somehow locate a peer in the network
> - `add to bucket` and add that node, to the appropriate bucket
> - `FIND_NODE on own ID` doing FIND NODE on its own ID, ensures that the new now communicates with a lot of different nodes in the network and gains some buckets
> - `FIND_NODE random ID's in buckets` lastly we perform a FIND_NODE on some random ids in the buckets, as a way to publish the arrival of the new node in their buckets.

So with Kademlia, we have an network which is:

- Built on the experiences from earlier structured networks
- It ensures high performance through parallelism
- All traffic contributes to routing table upkeep
- In widest use of all structured networks