Pastry and its Applications

Magnus Jensen

2. Pastry and its Applications

- Pastry and its Applications
    - Motivation
    - Pastry
        - Routing table
        - Locality
        - Joining
    - PAST
        - Insert
        - Lookup
        - Reclaim
        - Storage management
    - SCRIBE
        - Multicast tree
        - Groups
        - Repair
    - SplitStream

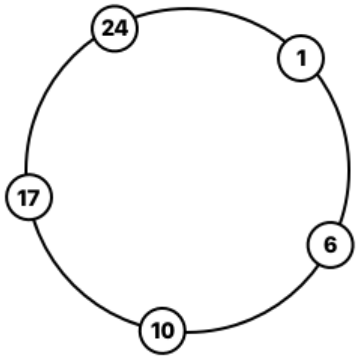
1. Motivation

2. Pastry

Great, so how about starting out, by exploring just what is Pastry. Shall we?

Pastry is a structured P2P network. Such a network 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).

Pastry does this, by creating an ID for every node, as a hash of their IP. Then the nodes are placed in a circle with 2^m seats.



Here, every node has a leaf-set, of their closets numerical neighbours. They have a routing table of how they are supposed to find a given ID in the ring, and lastly they have a neighbours set of their closest nodes given a distance metric.

> - **Pastry**
> - 128bit hash IDs
> - Placed in a ring
> - Leaf Set L ~~tættest på I ID space~~
> - Routing table
> - Neighbours set ~~tættest på I IP space~~

2.1. Routing table

In the routing table located at every node, there is a link to a successor node in the ring. The routing table is created based on a *prefix matching* metric.

> - **routing table**
> - prefix matching
> - `O(log(n))` routing can be done in O(logn) time!

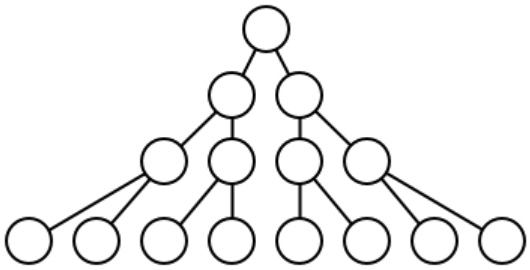
Say that we have a peer with ID *1010*, it will then maintain routing table with the some peers with an id matching the following sequence

- *
- 1*
- 10*
- 101*

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.

2.2. 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 the 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.

2.3. Joining

Locality is of course something that needs to be painted, especially doing joining of new nodes.

- `X near A joins` Lets imagine a node X, near A, joins the network
- `A routes JOIN msg with ID_x` X asks A to route a “join” message with key equal to X
- `A...->...Z` Pastry routes this message to node Z with nodeId numerically closest to X ~~Andre kender jo ikke til X, så siden Z er den tættest på, går de mod den~~
- `All returns state` All nodes en-route to Z returns their state to X
- `neighbourhood set of A`
- `leaf set of Z` leaf set is based on leaf set of Z (since Z has nodeId closest to nodeId of X)
- `routing table from others` rows of routing table are initialised

based on rows of routing tables of nodes visited en-route to Z (since these share increasing common prefixes with X)
- `X sends its state` to all the nodes mentioned in its leaf set, routing table, and neighbour list
- Then a second stage, is done - where X tunes its routing table

3. PAST

Great, so having described what Pastry is, lets begin to look at a few applications of it.

PAST in one of such system. It is a distributed file system, build on top of Pastry.

> - **PAST**
> - File system on Pastry
> - Used Private/Public keypairs

By creating a filesystem on Pastry and a multitude of internet nodes, PAST achieve strong persistence and high availability. PAST can be used as a global storage for backup, mirroring and such, and can share storage and bandwith among groups of nodes, to creates larger capacities of any individual node. All for one and one for all!

Being built on top of Pastry, PAST is in itself self-organising, though, a centralised component to hand out private and public keys could be used.

Nodes in the system will all need to contribute as an access point for users, though from different policies of implementations, it could also be that a requirement was to contribute some storage or handle routing.

Lets begin to talk about how the system works.

3.1. Insert

```
fileId = Insert(name, owner-credentials, k, file)
```

The Inserts method, does just that, with replicas of file on the k nodes whose IDs are numerically closest to fileId. Meaning that the system must maintain k copies of the file. Of course, k has to be less than $|L|$.

> $k < |L|$

The *fileId* is a SHA-1 hash of the file-name, the public key and some salt.

> fileId: SHA(file-name, public key, salt)

A certificate of the file is created and signed with the private key. It contains.

> - **File cert.**
> - fileId
> - `SHA1(file-content)` a hash of the content
> - k
> - the salt ... And other meta data

Then the certificate and the file is routed to the *fileId* destination. From the owner-credentials, the public key can be used to validate the insertion, and if so - it is forwarded to $k - 1$ closets nodes in the leaf set. If all accepts, a *store receipt* is returned.

3.2. Lookup

Are we to find file, we will perform a *lookup* on the system.

> file = Lookup(fileId)

We will have to know the fileId somehow, but performing this operation, a lookup request is routed towards a node with an ID closets to the fileID.

Since the system is built, so k nodes will hold the given file, any node doing so may respond with the file, and the certificate of course.

Most likely, it will be one of the nodes containing a replica, do to the very nature of Pastry and its leaf sets.

3.3. Reclaim

Now.. what is a reclaim? Reclaim, lets imagine it as *reclaiming* the space; so it is a delete, actually it is a soft delete. Well.. what is a soft delete, you may ask?

After a reclaim, a lookup of a fileId is no longer guaranteed to return a result. Why is that? Imagine that such a reclaim I routed, and one of the k nodes which contains the file, is down for moment. It will never get the reclaim message! Hence, we only delete it softly.

```
reclaim(fileId, owner-credentials)
```

Given the field and the credentials of the owner, only the owner of the file is able to reclaim it. The operation itself is kind of just analogous to the Insert method, and in the end, the owner will retrieve a *reclaim-received*.

3.4. Storage management

In such networks, we would like the available space to be filled properly before any insertion operation are rejected.

On way to do this, is called *Replica diversion*.

> - **Replica diversion**:
> - `Balance space in leaf` the idea is to balance the free space among nodes in the leaf set
> - `No space? pointer` if a node cannot store a replicate, it ask one from its leaf set and stores a pointer to it

Reasons to reject a replica, could be many, but replica size divided available space, should always be lower than a threshold, to allow room for smaller files.

Another way to manage this is called *File diversion*.

> - **File Diversion**
> - No room in idSpace
> - use new salt for new fileId
> - only used if Replica fails

Using such storage managements strategies, turns out to be a really good idea. Rejections fall from ~51% to under 5%, and space used goes from 60% to up to ~93%

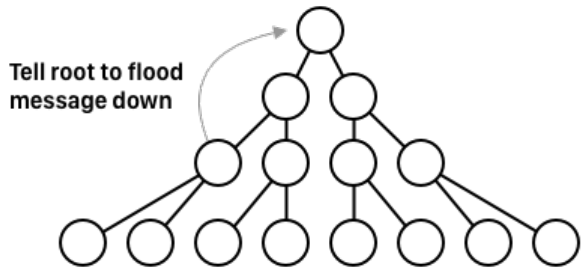
4. SCRIBE

Lets look at something totally different. Well it is still built on top of pastry, but the application is a whole other thing. SCRIBE is a way to handle multicast, well, in terms we have some messages and we want to share it with the world. Thereby, SCRIBE is a decentralized publish/subscribe system that uses Pastry for its underlying route management and host lookup. Users create topics to which other users can subscribe. Once the topic has been created, the owner of the topic can publish new entries under the topic which will be distributed in a multicast tree to all of the SCRIBE nodes that have subscribed to the topic.

4.1. Multicast tree

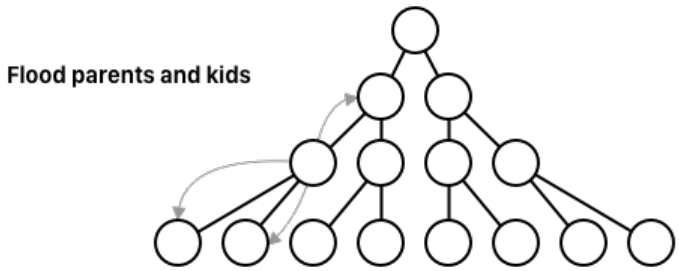
Lets briefly discuss multicast and how it is built on tree. There are two strategies of casting in a tree.

The first is a top-down approach, where a node tells the root if the tree to message its kids, which will message its kids and then the flooding happens.



The pro of doing it from a top down, is that the order is guaranteed, but the load on the root can be hard.

Another way is crawling around the tree, so a node will ask both its children and its parent to propagte the message.



Here, the root aint under the same load, but maintaining an order is hard.

4.2. Groups

To create a group,

> - **Create group:**
> - `groupId = SHA1(group name + owner)` a group id is created from the owners name and the groups name
> - `CREATE -> groupid` a CREATE message is then send towards the node in the pastry network which has an ID closest to that of the group
> - `receiver = tree root` the node which ends up receiving the CREATE message in the end, will now be the root of the tree.

To join the group,

> - **Join groupe:**
> - `send JOIN at groupeId` a node will send a JOIN message in the pastry network at the groupId
> - `forwarder` if an intermediate node that receives this JOIN message is a *forwarder* for the group, meaning that it is in the tree, it adds the new member as a child; yet if it aint in the tree, it will become so as a forwarder, by sending a “join” message to the group and add the member to its child.

An important note here, is - that being in the tree, you can either be a simple forwarder or a fully member; while the member also is a forwarder of messages.

To leave a group,

> - **Leave groupe**
> - `downgrade to forwarder` a node marks itself as downgraded to forwarder
> - `wait for 0 kids, LEAVE` then it waits until I have 0 kids, where it sends a LEAVE message to its parents which carries up the tree.

Lastly, to send a message:

> - **Send message**
> - `Send directly` messages are send outside of pastry, so its directly in the tree, using the top-down approach

4.3. Repair

In case of the tree somehow gets broken doing its lifetime, periodically, each non-leaf node sends a heartbeat message to its children and multicast messages are used as an implicit heartbeat. If children have not received a heartbeat for a set amount of time, it assumes something is wrong with its parent, and then it simply rejoins the group by sending a “join” message towards the tree node.

In case the root of the tree, somehow gets lost, there is also taken measures to fix this. Each node in the leaf set of the root, will at all time keep a copy of the state of the root. Because the nodes in the leaf set is numerical close to the root, one of these will be the new root.

So when the root disappears, the children will rejoin the group do to the missing heartbeat, and then one of the leaf nodes from the old root will to its surprise be the new root.

5. SplitStream

~~Du vil måske undlade denne del, jeg fatter ærligtalt ikke hvad det er og kan ikke stå inden for det. Desuden, er tiden måske brugt når du kommer her til.~~

~~I det følgende bruges ordet “interior node” som er en node i et træ det ikke er et leaf~~

Now we have discussed Pastry which is the foundation for SCRIBE, lets quickly get an overview for a system that is build on both! SplitStream!

The stream is split into k stripes, encoded such that only a subset is sufficient for playback. A joining peer specifies its in- and outbound capacity that is how many stripes it wishes to receive and how many it can transmit.

In SplitStream this is built based on a multi-tree architecture. Where every node is an interior node in one tree and a leaf node in every other tree. Doing such splits the data up, and creates a high bandwith content distribution system, which utilises tree from SCRIBE and routing from Pastry.