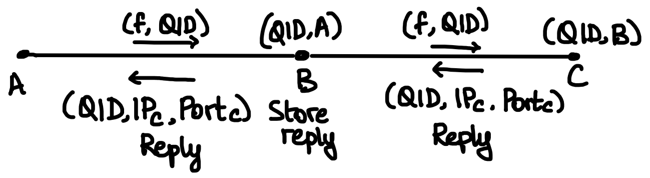
Application Layer

Lecture 28 Peer-to-Peer Networks

In the client-server model, the client gets a service from the server.
In a P2P network, there are many equal peers, who can provide services to each other.
→ How do we start using it? What services? How do we provide services?
It arose from the desire to share digital content, so not everyone needs to pay.
Napster had a central server, and logged in people could upload what files they have and freely share them. Others can search the server for some song, and download it from someone else if there is a match.
It finally shut down (it was illegal, after all).
- Centralised, so not fault-tolerant
- Legally easy to take down.

After Napster came GNUTELLA. It tried to get rid of the problem of being centralized.
To join a pre-existing network (bootstrapping),
→ The application may have the IPs of some peers.
or
→ We may be able to look up IPs from one or more websites.
To search for a file,
We do a limited broadcast — a broadcast but limited to a certain number of hops away. We can do so by setting a TTL at the application layer. Suppose the query has f (file name) and the query ID (but not the requester's IP. Each host who receives this query caches the query ID and the peer they heard it from. When it reaches someone who has a file, they reply with their own IP and Port number (with the query ID). This goes backwards until the requester gets to know who has the file, so we can then share.
Further, the intermediate peers store the reply for the particular file.



At its peak, there were millions of people connected to GNUTELLA.

The QID is to ensure that we do not redundantly send the same message multiple times (if received multiple times).

In an improved version, we can also store the requester's IP to allow a direct reply.

In v0.4, TTL = 7

v0.6, TTL = 4

+ There is no centralized node.

− We frequently broadcast.

How do we fix this? Is there a way to

- map files to IP addresses such that
- the IP stores who has a particular file.

(centralized but distributed)

centralized ↓ a single person knows who has the particular file

distributed ↓ this person is different for different files.

We would like to do this pseudo-randomly ensuring that everyone has the mapping.

To do this, we use a hash function (which is deterministic).

$$(\text{arbitrary length bitstring} \mapsto \text{fixed length bitstring})$$

The hash values will be more or less evenly distributed over the space.

To map to IP addresses, we further hash the IPs.

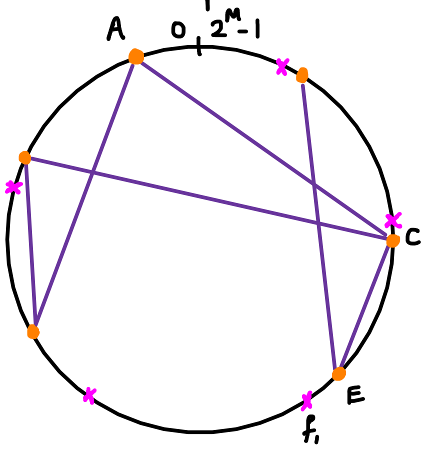
A hash (of a file) is then mapped to the hash (of an IP) that is closest to it.

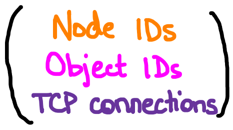
Lecture 29 PASTRY

Let us refer to the files as objects and peers as nodes. The object ID and node ID are the corresponding hashes. How do we find the node ID in the network closest to an object ID? Also, what do we do if someone leaves the network?

someone ↓ (who is mapped to)

The node IDs and object IDs can be represented on a circle:





We route messages to the $\underbrace{\text{neighbour}}$ closest to the object ID until we reach the desired node.

this set of neighbour peers is called the leaf set.

In the above example, if A has f_1, $A \rightarrow C \rightarrow E$.

One such protocol is PASTRY. It is such that

→ in the leaf set of node X, we include the $L/2$ closest nodes on either side of X in the virtual space (L is some parameter)

It is ensured that the leaf set of any node is of size at most $L + O(\log n)$.

This property ensures that the routing protocol gets to the node closest (to a hash).

Usually, $L > 2$ Now, how do we include these nodes in the Leaf set?

Suppose the P2P network already has the property that the leaf set of all nodes contain the $L/2$ closest nodes on either side.

→ When a new node X enters, it first routes a message to find the node whose ID is closest to its own ID. From this node, it gets to know the $L/2$ closest on each side (asking for $\frac{L}{2}-1$ on one side and $\frac{L}{2}$ on the other).

The search part of PASTRY ($L + O(\log n)$) starts by adding some nodes using the leaf sets of nodes involved in the search message — these are fairly well distributed. These add the additional $O(\log N)$ nodes to the leaf set (of a new node).

This sort of structure, wherein different nodes store where different objects are stored, is called a distributed hash table (DHT).

CHORD and TAPESTRY are other protocols similar to PASTRY.

What if a node fails? Suppose node X knows where f_1 is stored but it suddenly shuts down. We need some degree of redundancy to ensure that this does not cause problems. For this, we also store the information present in two nodes on either side. Each node also sends Keep Alive messages to the nodes they are connected to (the leaf set). If a node ≤ 2 away fails, it gets the information of another file on that side.

Lecture 30 DNS

DNS stands for Domain Name System. It essentially maps IP addresses to human-readable addresses. For example, cse.iitb.ac.in $\rightarrow$ DNS $\rightarrow$ IP address. We would like it to be robust.

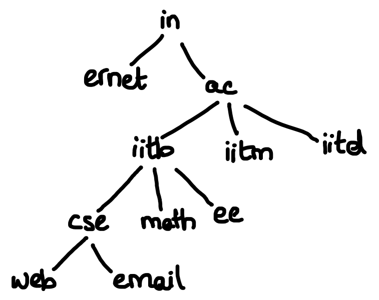
Idea.

Maintain a server with a well-known IP address. Use this server to perform the DNS lookup.

- Centralized, so not fault-tolerant.
- Can be overloaded if too many requests.

How do we fix this?

Let us look at the URL www.cse.iitb.ac.in
↳ India



$\longrightarrow$ There is a neat hierarchical structure. Can we use this?

We can get the information of the next level using the current level, beginning at the root server. So the server for in would give you the DNS for ac, which would give you the DNS of iitb. At the lowest level, the server gives you the IP. This uses the hierarchical structure to its advantage in a recursive method.

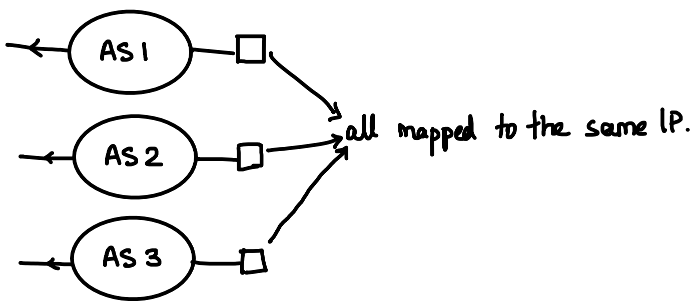
If we have cached the DNS of something lower in the hierarchy, we can use it directly.

Let us begin with the root server. Its IP should be well-known, and it should be robust to attacks.

→ It must be very well-provisioned with bandwidth and CPU, to prevent DoS (Denial of Service) attacks.

→ It must have some degree of redundancy. We have many root servers — currently 13. Their names are letter.root-servers.net, where letter goes from a to m. The reason for 13 is that the DNS packet size of 512 bytes has space for 13 servers (it is a legacy issue).

→ For another layer of redundancy, we use anycast. In this, a sent packet goes to any of the servers. To do this, all the servers have the same IP. We do not really care who gets the packet, so this is fine. Using this, each of the root-servers.net is mapped to many geographically distributed physical machines. This is done as



It might go to a different machine based on what BGP chooses.

A resource record has a

< Name , Value , Type , Class , TTL >

Name, Value → Mapping
Type → How value should be interpreted
Class → IN (for Internet)
TTL → Time before record expires. If we cache the IP, lets us know how long it will be good. (measured in seconds)

Type	Value
A	IP Address
NS	Name Server (Host running DNS service in domain corresponding to Name)
CNAME	Alias of Name (Canonical name of host in Name)
MX	Name of host running mail server in domain specified by name

For example, the root server can have

<edu, a3.nstld.com, NS>
↳ DNS server of edu domain

<a3.nstld.com, 192.5, 6.32, A>
↳ IP of "Name"

The server a3.nstld.com itself could have

< college.edu , dns.college.edu , NS>

<dns.college.edu , 128.112.129.5 , A >

At the college.edu server,

<www.college.edu , coreweb.college.edu , CNAME>

< college.edu , mail.college.edu , MX >

Let us look at an example of a lookup. Either we can manually configure the DNS or use DHCP to get to local DNS server.

DNS runs on UDP — port 53 for the server.

The local DNS server queries the root server to get the relevant resource records, and caches them (until the TTL ends). We use this to know who to get to next, and recurse. The query is the same at each step. Finally, we get the IP.

If useful cached resource records are present, we can use them to avoid going through the above every time.

—— X ——