POCS: Naming, Recitation

Katerina Argyraki
School of Computer & Communication Sciences

Chord

Problem

Design a scalable decentralized system that maps keys to nodes, under significant node churn.

So, Chord is not itself a naming service, but it can be a component of a decentralised naming service.

App/Chord Interface

- downcall (app to Chord): lookup(key) —> IP address of node resp. for key
- upcall (Chord to app): notify(changes)

Downcall may time out. App tries again later.

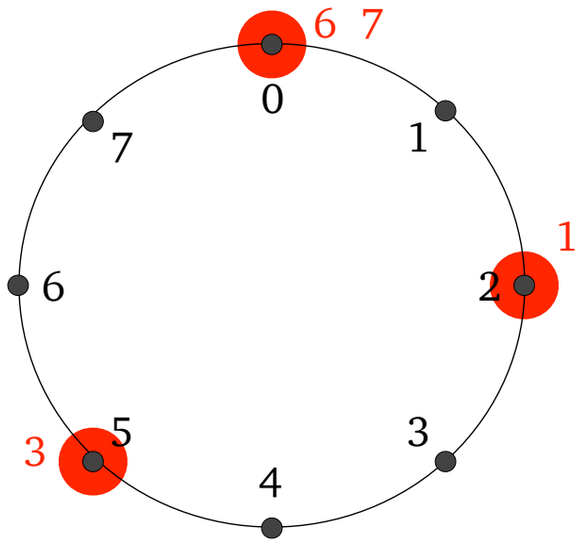
Solution

- Consistent hashing assigns to each node and each key a flat m-bit ID
  - *IDs organized in a circle from 0 to 2^m-1*
- Key k assigned to the first node n>=k
  - *successor(k)*
- Correctness: each node knows the successor of its own ID
- Performance: each node knows the successors of up to m IDs
  - *fingers*

Separate parts of the protocol for correctness and performance.

—> Was there a case in GNS where we saw a similar separation of concerns?
- Sweep for eventual consistency.
- Direct messages for performance.

Lookup



lookup(key 3)
lookup(key 7)

key (start)	node (succ)
3	5
4	5
6	0

Failures

- For correctness: each node maintains list of r nearest successors
- For performance: if a contacted node does not respond,
  contact the nodes preceding it in the finger table (or one of the successors)

Scalability

- Number of keys: arbitrary…
  - *as long as the number of keys per node does not exceed per-node memory*
- Number of organisations: 2^m (as many as the number of nodes)

Assumption: each nodes is managed by one organisation

Flat, fixed-length names vs. hierarchical, variable-length names.

- controls the number of names.

Scalability

- Per-node state (memory): up to m entries
- Nodes contacted during lookup (latency): O(logN) [Theorem 2]

Interesting points

- Scalability is quantified through formal bounds
- Separation of correctness from performance concerns

Practice

Characterise Scalability

- Compare the scalability of GNS and Chord.

Chord Scalability

- Number of keys: arbitrary…
  - *as long as the number of keys per node does not exceed per-node memory*
- Number of organisations: 2^m (as many as the number of nodes)

Assumption: each nodes is managed by one organisation

Flat, fixed-length names vs. hierarchical, variable-length names.

Comparison to GNS:
- Number of names per directory should not exceed per-server memory.
- A bit more practical: server/directory admin controls the number of names.

Chord Scalability

- Per-node state (memory): up to m entries
- Nodes contacted during lookup (latency): O(logN) [Theorem 2]

Comparison to GNS:

Hard to say. Depends on the form of the GNS tree.

Assuming a balanced tree

GNS latency depends on
- the depth of the directory tree, which is related to
- how many directories the tree is accommodating.

Assuming a perfectly balanced directory tree:
dirs = (children^tree-depth - 1) / (children - 1) <=>
tree-depth = log_children (dirs*children - dirs + 1) <
log_children (dirs (children-1)) =
log_children(dirs) + log_children(children-1) —>
log_children(dirs).

So, the nature of the trade-off is similar: Given the same number of “routing pointers,” lookup time is a logarithmic function of the number of nodes/directories.

However, if we plug in numbers, GNS offers better lookup time. E.g., for m=32, N=1000:
- Chord’s lookup time is on average 0.5 * log_2(1000) = 5.
- GNS’s lookup time is log_32(1000*31+1) = log_32(31001) < 3.

The form of the tree evolves “organically,” depends on admins and how they create directories.

Chord relies on a hash function.

Define an interface

- Write down a simplified version of the interface that the current DNS system exposes to DNS clients. Compare it to the interface exposed by GNS to its clients: How are they similar? How do they differ? Is one a “superset” of the other (i.e., does one provide a superset of the functionality provided by the other)?

A simplified version of the interface exposed by DNS to its clients is:
DNS_GetValue (EntityType entity, PropertyType property) --> (value, cachingDuration)
Entities and properties must be of pre-defined types. E.g., an entity can be a domain name or an IP address, while a property can be a domain name’s IP address, or a domain name’s mail server. The cachingDuration specifies for how long the client should cache the mapping; however, there is no guarantee that the mapping won’t change in the meantime. This is similar to one of the procedures exposed by GNS to its clients:
GNS_GetValue ValueDesignator —> tree
Both procedures take as input a name and return the value of that name. The two procedures differ in that:
In DNS_GetValue, entities and properties must be of specific, pre-defined types. In GNS_GetValue, the input (resp. output) is a generic hierarchical name (resp. tree of names).
DNS_GetValue returns a cachingDuration, which, however, does not guarantee that the value won’t change in the meantime. In GNS_GetValue, each name encoded in the returned tree is associated with an expiration time, and the name is guaranteed not to change until that time.
However, GNS exposes more procedures to its clients, which enable them to update values. The interface exposed by GNS is a “superset” of the one exposed by DNS: Any mapping that is stored in DNS could be stored in GNS. E.g., a mapping from domain name “www.epf.ch” to IP address “1.2.3.4” could be stored as “/ch/epfl/www/IPv4-address/1.2.3.4.” Also, GNS provides stronger guarantees about the consistency between the cached version of a mapping and its authoritative version.

Explain a statement and/or a solution

- The conclusion of the GNS paper states that one of its most interesting aspects is “the ability to name servers using the service itself.” What does that mean? What is the challenge with providing this ability? How does GNS solve this challenge?

In order to look up a name in a directory, a client must first learn the name of a server that stores a copy of that directory, and then map the server’s name to an address. E.g., to look up /DEC/SRC/Lampson/Password, a client must first learn the name of a server that stores a copy of /DEC/SRC. To store and access the mappings from server names to server addresses, GNS uses itself as a service (it stores them and accesses them the same way a GNS client stores and accesses mappings). The challenge is avoiding loops. E.g., consider a server with name /DEC/SRC/alpha that stores a copy of directory /DEC/SRC. To lookup a name in /DEC/SRC on this server, a client must first lookup the server’s name /DEC/SRC/alpha, which is itself a name in /DEC/SRC. GNS solves this challenge through invariants S1 and S2. The former says that every directory that is part of a server name has the variable “inSN” (“in server name”) set to “true.” The latter says that every directory d with inSN = true either is the root or is stored in a server with a name shorter than any direct name of d. E.g., /DEC/SRC has inSN = true, because it is part of server name /DEC/SRC/alpha; hence, /DEC/SRC/alpha is stored in a directory with a name shorter than /DEC/SRC/xxx, e.g., /DEC/alpha. The two invariants together guarantee that a directory is always stored in at least one server whose name does not involve that directory.