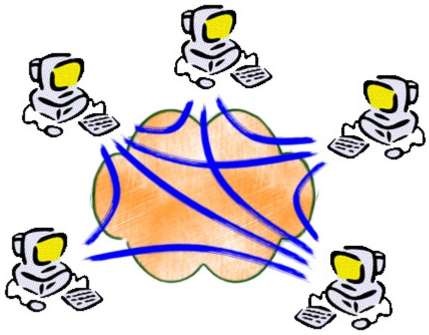
Chapter 6

Peer-to-Peer systems

1. **Introduction**
2. **Overlays**
3. **Distributed Hash Tables**



1

Chapter 7

Peer-to-Peer systems

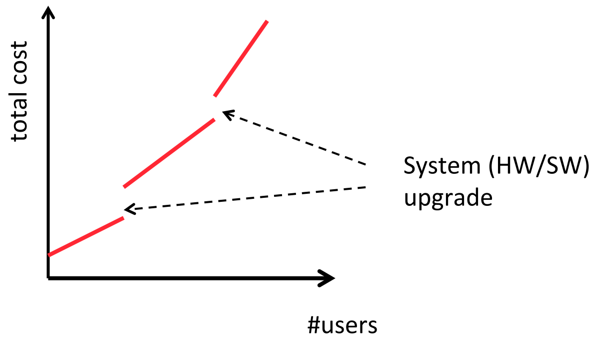
1. **Introduction**
    1. Why P2P systems ?
    2. P2P generations
2. **Overlays**
3. **Distributed Hash Tables**

2

The rationale for P2P

Scaling and cost of client server systems

popular services need large server infrastructures
-> high investment and operational cost



Observation

- performance of edge devices grows (gap client – server closes)
- performance only occasionally fully used

3

The rationale for P2P

“How to unleash the power of Internet’s dark matter ?”

P2P philosophy

make heavily use of edge resources
(CPU, storage, I/O devices, Information, ...)
#users ~ #edge devices
-> infrastructure grows together
(automatically) with user base

Need for software systems that can survive without central servers.

4

Some numbers

Edge resource estimation

In total

- #hosts : 10^9
- processor performance : 2 GFLOPs
- disk space : 500 GB

Available

- 1% connected hosts
- 50% CPU power
- 10% disk space

-> "P2P supercomputer"

  - 10^4 PFLOPs CPU power
  - 5 10^5 PB disk space

A "real" supercomputer

Nov 2012 :

1. Sequoia IBM (20 PFLOPs, 8 MW) @DoE
2. K Computer Fujitsu (11 PFLOPs, 12 MW) @Riken

Why is P2P difficult ?

Nodes

- large number of nodes
- unstable
- under control of end users

Interconnection infrastructure

- slow
- unreliable

No central control

- more complex managment
  (control infrastructure can fail)

=>

- decentralized control
- self-organization
  - handling failing nodes
  - spreading load dynamically

Characteristics of P2P systems

Shared characteristics
- Users contribute to the total pool of available resources (avoid free-riding)
- All nodes in principle equal
- System operation independent of centralized control

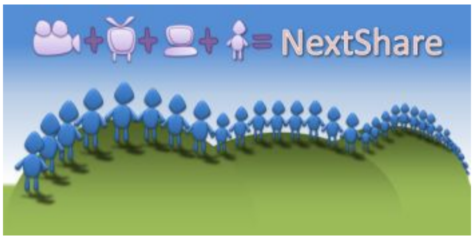
Application areas
- file sharing
- collaboration tools (Groove)
- communication (VoIP P2P [Skype], chatting [Jabber])
- CPU scavenging (SETI@Home, Folding@Home, ...)

Important problems
- data placement and lookup
- routing (use the network bandwidth efficiently)
- provide anonymity
- self-organization (self-management)

The history of P2P

Generation I : File sharing with centralized control
- index maintained in a centralized infrastructure
- download purely P2P
  - -> poor scalability
  - -> vulnerable to attacks

Generation II : File sharing with decentralized control
- self-organization through overlay construction
- two variants:
  - pure P2P (all nodes identical)
  - hybrid P2P (hierarchical structuring)
- scalable and robust



Generation III : P2P middleware
- middleware services offered:
  - data placement
  - data lookup
  - automatic replication/caching
  - authentication/security
- applications built on top of P2P middleware

P2P-architectures

	mediated	pure	hybrid
data traffic	P2P	P2P	P2P
control traffic	client-server	P2P	local : client-server on distance : P2P
efficiency	+ efficient search + efficient control	- inefficient search - BW consuming	+/-
scalability	- control hot spot (mirrors needed ?)	- BW needed grows rapidly	good compromise
robustness	- single point of failure - easy to attack	+ graceful degradation + difficult to attack	?
accountability	easy	difficult	difficult

Generation I (mediated) — Generation II (pure, hybrid)

Generation I : Rise and fall of Napster

napster

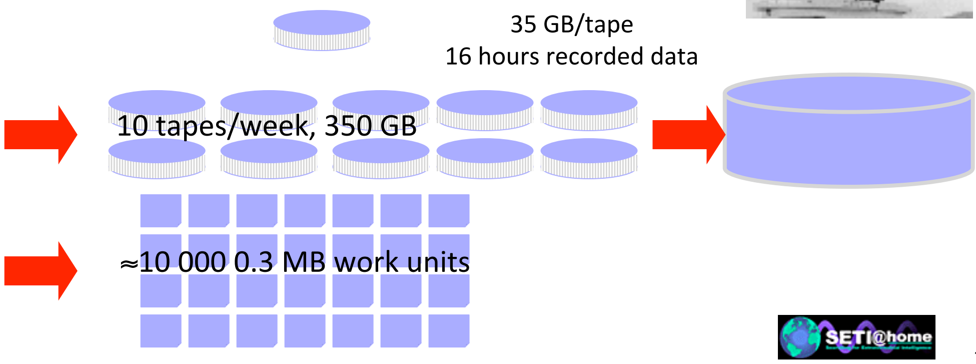
- Aug '99 — Shawn Fanning releases Napster on the Internet
- Oct '99
- Dec '99 — Recording Industry Association of America sues Napster for copyright infringement; College campuses block Napster
- Feb '00 — 5 million Napster users; $15mio investment Hummer Winblad $
- Apr '00 — Metallica sues Napster; 300,000 users blocked
- Jun '00 — Senate hearing on digital music and copyright ©
- Aug '00 — 20 million Napster users; BERTELSMANN
- Oct '00 — 32 million Napster users
- Dec '00 — Bertelsmann drops lawsuit, provides loan to develop subscription service
- Feb '01 — 2.7 billion songs downloaded in Feb01; Napster offers $1billion to record companies $
- Apr '01 — On court order, Napster adds filters to block copyrighted content
- Jun '01 — 90% of songs are blocked
- Aug '01 — Napster goes offline

Generation I : SETI@Home

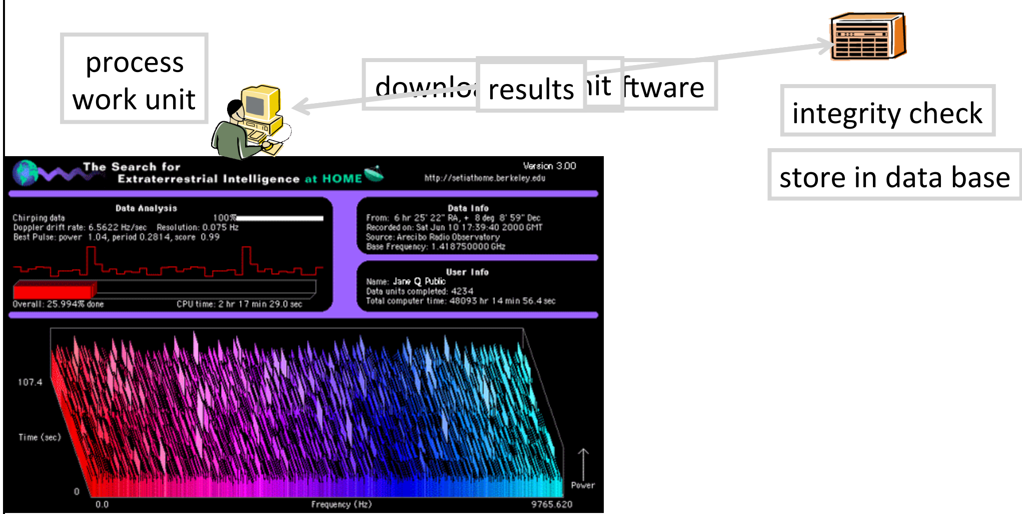
SETI

="Search for extraterrestrial Intelligence"

- started in 1998 as a 2 year project
- 4 M users signed up
- Radio telescope data sent to clients for digital signal analysis
- Nodes process data when cycles are available
  (works as screen saver)
- Using resources to allow better signal analysis



SETI : How it works



SETI : Some numbers

computations per work unit — 3.1×10^{12} FP-operations
work unit throughput — 700 000/day

→ 22×10^{17} FLOP/day

→ >25 TFLOPS

	SETI@home	ASCI White@DoE
		
Processing	25 TFLOPS	12.3 TFLOPS
Cost	1 M USD	110 M USD

Generation III : P2P middleware

Requirements
- add/find/remove distributed resources transparently
- globally scalable
- load balancing
- exploitation of locality
- dynamic adaptations (dynamic hosts/resources)
- security of data
- anonymity

Important platforms
- Pastry, Tapestry, CAN, Chord, Kademlia

Chapter 7

Peer-to-Peer systems

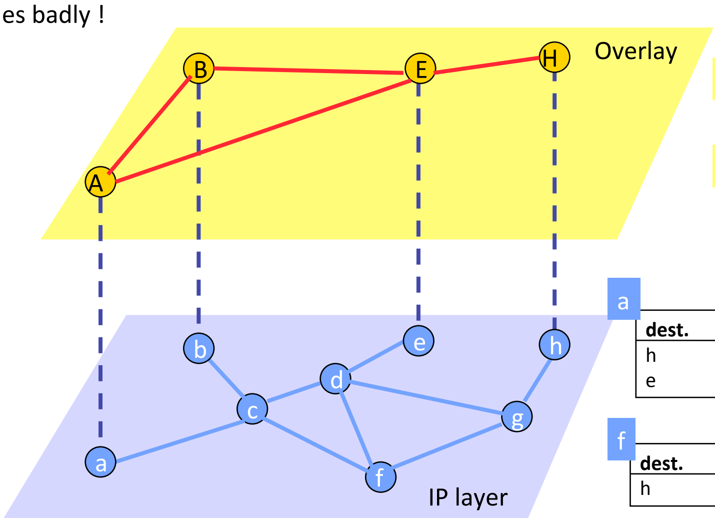
1. Introduction
2. **Overlays**
3. Distributed Hash Tables

15

2. Overlays

Overlay routing

Full mesh overlay topology scales badly !



A

dest.	next hop
H	B
E	E

B

dest.	next hop
H	E

E

dest.	next hop
H	H

a

dest.	next
h	c
e	c

c

dest.	next
h	f
e	d

f

dest.	next
h	g

g

dest.	next
h	h

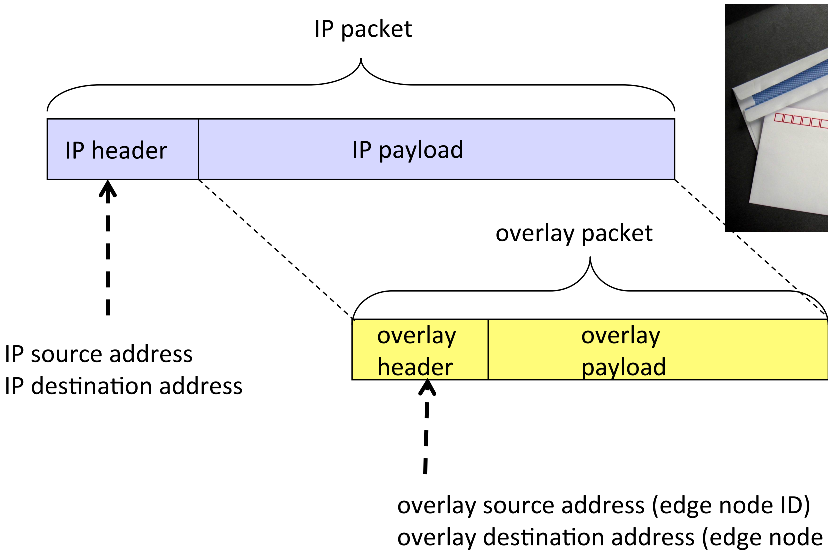
d

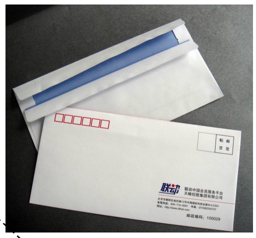
dest.	next
e	e

b

dest.	next
e	c

Overlay routing





IP layer and overlay layer have own
- addressing scheme
- routing protocol

Content based routing

Use ID of object to manipulate instead of node ID
-> overlay can redirect to closest replica
-> overlay can optimize placement and #replica's

Globally Unique ID (GUID)
- constructed through hash of object state
    - content itself
    - object description
- overlay = distributed hash table (DHT)

```
ff478f2d-e398-449a-93d4-62b0727
adffed61-73f9-42a1-8d54-90f3537
ec359ec3-99be-4c44-8449-be7a1ab
9349e44d-5367-4b20-b6b1-d421c5a
54fbb946-f9e3-498f-b04c-e91584d
c8d1308c-76d2-452e-8d80-6d08483
f1a841a3-b456-4e92-99b5-04f3d86
ba1a4110-01cb-4a4a-967e-b8adbf5
63d06b7a-8cf5-46d5-bcdd-ffd3be7
4ffa8171-78e0-455a-9cbe-e9fee93
```

Responsibilities of DHT
- find object given GUID (route requests)
    -> map GUID to node ID
- announce GUIDs of new objects
- remove GUIDs of old objects
- manage with nodes becoming unavailable

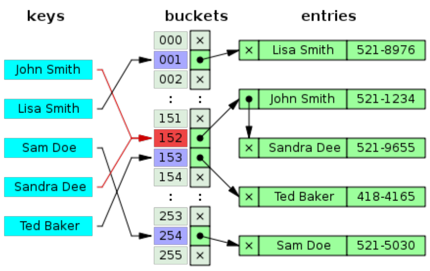
DHT routing vs. IP routing

	IP routing	DHT routing
Size	IPv4 : 2^{32} nodes IPv6 : 2^{128} nodes but: hierarchical structured	> 2^{128} GUIDs flat
Load balancing	routes determined by topology (e.g. OSPF routes)	any algorithm can be used objects can be relocated to optimize routing
Dynamics	static (time constant 1h) asynchronous w.r.t. application	can be dynamic synchronous or asynchronous
Resilience	built in	ensured through replication of objects
Target	IP destination maps to 1 node	GUID can map to different replicas

Chapter 7

Peer-to-Peer systems

1. Introduction
2. Overlays
3. **Distributed Hash Tables**
    1. Introduction
    2. Circular routing (1D)
    3. Pastry (1D)
    4. Chord (1D)
    5. CAN (nD)



Data management APIs

Distributed Hash Table [DHT] API

put(GUID,data)
remove(GUID)
value = get(GUID)

DHT takes care of:

- finding good location
- number of replicas needed

Distributed Object Location and Routing [DOLR] API

publish(GUID)
unpublish(GUID)
sendToObj(message,GUID,[#])

node made explicitely responsible for GUID

Bootstrapping

The bootstrapping problem
how to find a network node ?

Approaches
- pre-configured static addresses of stable nodes
  - should have fixed IP address
  - should always be on
- DNS-service
  domain name resolves to nodes

Configuration info made available
- routing table bootstrap info
- GUID space the node is responsible for
- protocol information

DHT : GUID routing

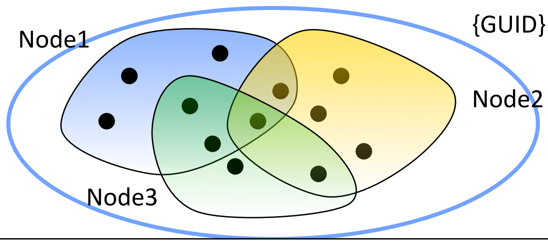
Concept

Content Item → hash function → GUID

e.g. Node Public key → GUID

Node is responsible for range of GUIDs it belongs to

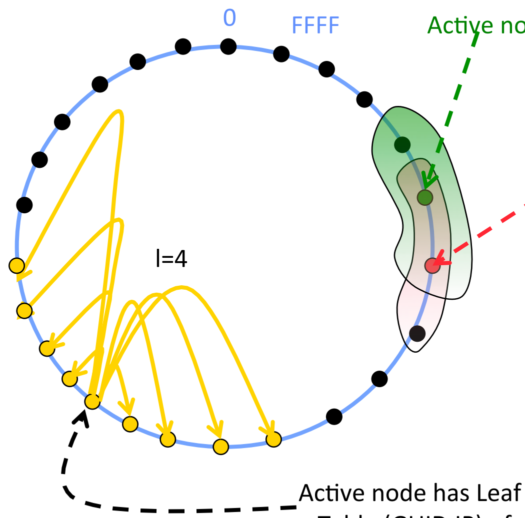
Nodes are responsible for part of GUID space



GUIDs in intersection are replicated

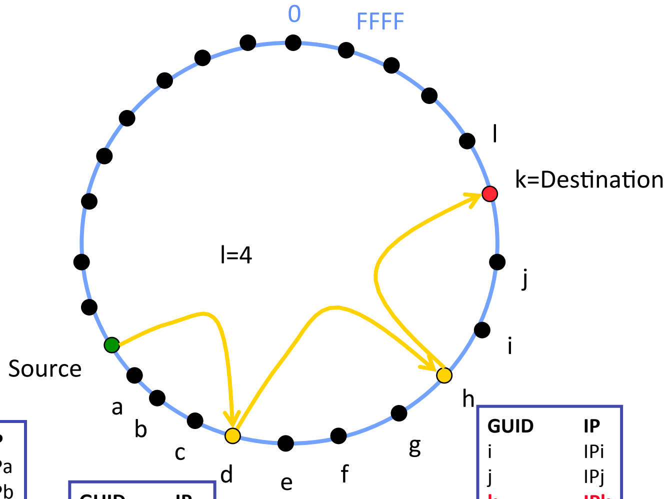
Circular routing

Suppose : GUID consists of 4 hex symbols



Active node has Leaf set (size 2l)
= Table (GUID,IP) of neighbours

Circular routing



GUID	IP
a	IPa
b	IPb
c	IPc
d	**IPd**

GUID	IP
e	IPe
f	IPf
g	IPg
h	**IPh**

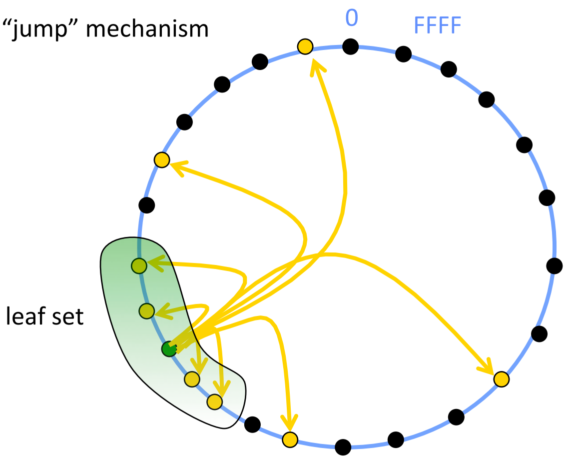
GUID	IP
i	IPi
j	IPj
k	**IPk**
l	IPl

N active nodes leaf set size = 2l	=>	worst case N/2l hops

Prefix routing

Optimize routing efficiency
- allow for longer jumps
- without increasing leaf set

=> need for additional “jump” mechanism



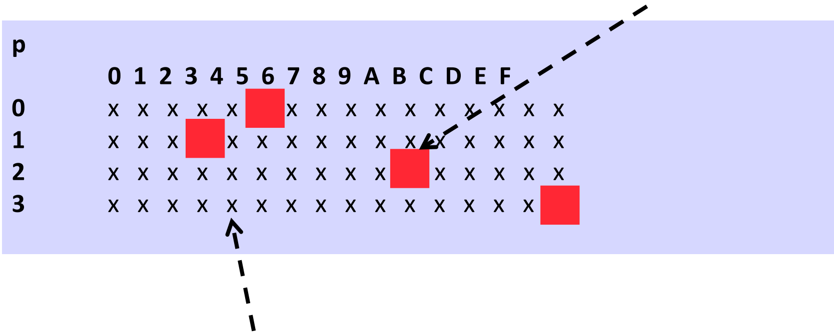
Prefix routing

Routing table structure @active node

n-bit GUID -> n/4 routing table rows
(16 bit -> 4 rows)

p = longest prefix match
Routing table of node with GUID 53AF

[GUID,IP] of next hop for message destinated for 5Cxx



[GUID,IP] of next hop for message destinated for 53A3 (=closest active node to 53A3)

Prefix Routing

Message m from S(ource) to D(estination) arrives in C(urrent) node
Leaf set : size 2l (L_{-l} -> L_l)
R : routing matrix

```
if (L_-l ≤ D ≤ L_l) {
        forward m
                - to GUID_i found in leaf set
                - to current node C
} else {
        find longest prefix match p of D and C
        c = symbol (p+1) of D
        if(R_pc ≠ null) forward m to R_pc
        else {
                find GUID_i in L or R with |GUID_i – D|<|GUID_i-C|
                forward m to GUID_i
        }

}
```

Observation:

correct routing possible with empty routing table

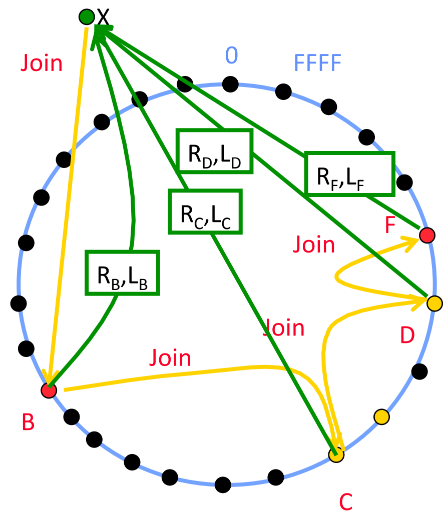
Node dynamics : Joining

X wants to join

- how to construct R_X, L_X ?
- how to adapt R of all other nodes ?

Special "nearest neighbour" discovery algorithm to find a nearby active node B

1. X sends join(X,$GUID_X$) to B
2. DHT routes join the usual way to node F
   min $|GUID_F - GUID_X|$
3. All nodes on routing path send info to X to construct R_X,L_X



Node dynamics : Joining

Constructing R_X

$\{B_0, B_1, B_2, \ldots, B_{N-1}\}$ = set of physical nodes visited by Join message
B_0=B
B_{N-1}=F

Row 0 :
- used to route GUID with no common prefix
- bootstrap node B_0 (very) close to X
=> $R_X[0]=R_{B0}[0]$

Row 1 :
- used to route GUID with common prefix 1
- prefix($GUID_{B1}$,$GUID_X$)≥1
=> $R_X[1]=R_{B1}[1]$

…
Row i : => $R_X[i]=R_{Bi}[i]$

Constructing L_X

X should be neighbour of F
initial choice : L_X=L_F

Node dynamics : Leaving

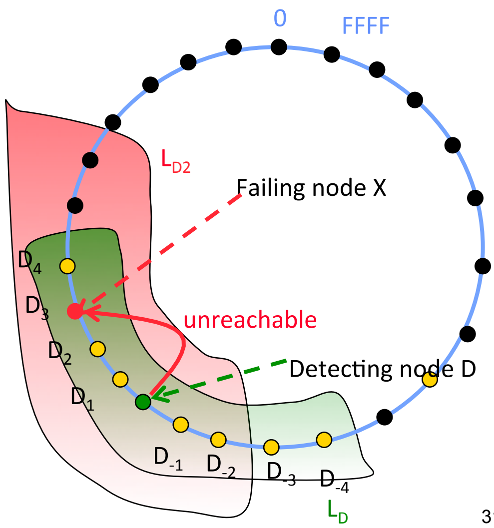
Only leaf sets are repaired

l=4

L_D

$GUID(D_{-4})$	$IP(D_{-4})$
$GUID(D_{-3})$	$IP(D_{-3})$
$GUID(D_{-2})$	$IP(D_{-2})$
$GUID(D_{-1})$	$IP(D_{-1})$
$GUID(D_{1})$	$IP(D_{1})$
$GUID(D_{2})$	$IP(D_{2})$
$GUID(D_{3})$	$IP(D_{3})$
$GUID(D_{4})$	$IP(D_{4})$

1. D finds node close to $GUID_X$
   -> D_2
2. Get L_{D2}
3. Adapt L_D based on
   -> Remove D_3
   -> Add D_5
4. Propagate to neighbours



31

Chord

Basic Chord API

lookup(key) : maps key -> IP address of node responsible for the key

Application built on top on Chord

- uses lookup(key)
- gets informed by Chord when key-set of current node changes
- is responsible for (if desired)
  - authentication
  - caching and replication

Sample applications

- cooperative mirroring (multiple servers cooperate to store content)
- time shared storage (ensuring data availability, even if server off-line)
- distributed indexes (content sharing)
- large-scale combinatorial search (e.g. code breaking)

32

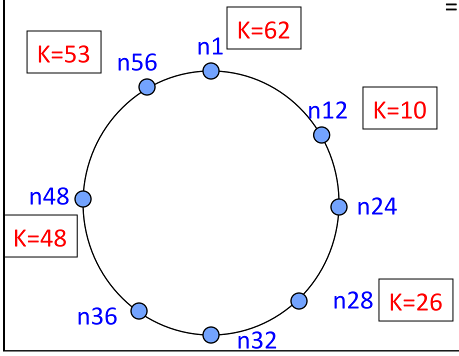
Consistent hashing

Hash function (SHA-1) produces m-bit identifiers

hash(nodeIP)
hash(key)

Identifiers mapped to identifier circle modulo 2^m

Key k -> hash(k) -> node(hash(k)) = successor(k)
= node with smallest identifier ≥ hash(k)

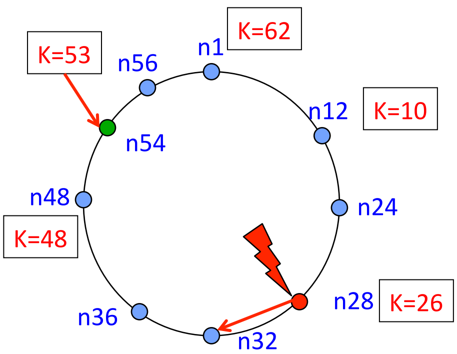


m = 6
8 nodes
5 keys stored

Joining and Leaving

X Leaving
keys associated with X -> reassigned to successor(X)

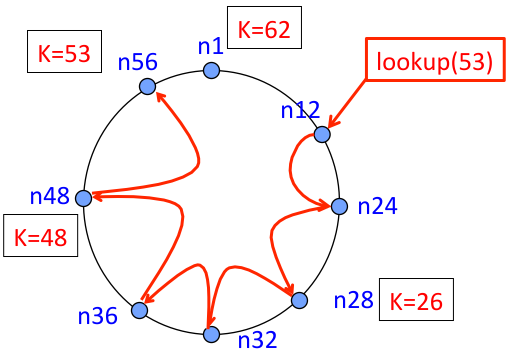
X Joining
X gets some of the keys assigned to successor(X)



Simple Routing

Request lookup(id) arrives at node n
Node n knows id of its successor

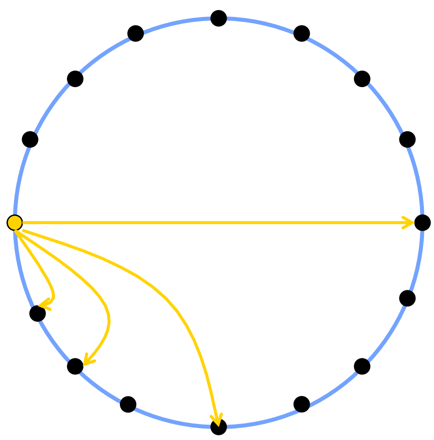
```
n.findSuccessor(id) {
        if id in (n,successor]  return successor
        else return successor.findSuccessor(id)
}
```



(a,b]
Clockwise interval
Excluding a, including b

Skiplist routing

Each node has **finger table**
≈ leaf set with non-equidistant node intervals



Finger table (m entries)

Distance	ID	IP
1		
2		
4		
8		
…		

Routing to GUID
- send message to successor GUID
- = highest ID in finger table with ID≤ GUID

O(log N) hops needed

Robustness
- node keeps track of n successors
- = circular routing in case finger table not valid

Skiplist routing

Finger table

DistanceID **IP**

1

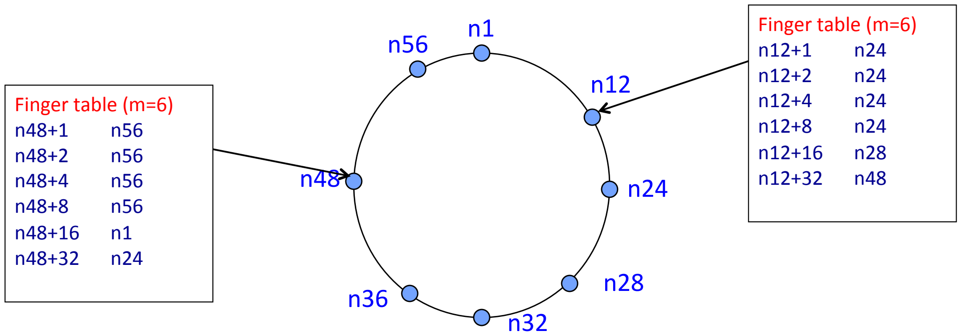
2

4

More formally

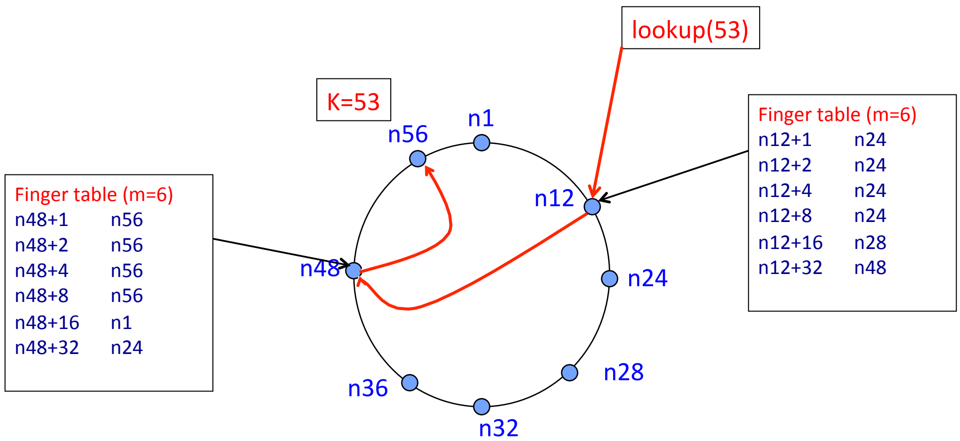
row i in finger table is node at distance larger than 2^{i-1}

finger[i] =successor($n+2^{i-1}$) ($i \geq 1$)



Skiplist routing

Routing : take largest jump possible !



Skiplist routing

Routing : take largest jump possible !

```
n.findSuccessor(id) {
        if id in (n,successor]  return successor
        else {
                n'=closestPreceedingNode(id)
                return n'.findSuccessor(id)
        }
}

n.closestPreceedingNode(id) {
        for i=m downto 1
                if finger[i] in (n,id] return finger[i]
        return n
}
```


Node dynamics : Joining

Bootstrapping (create a new ring)

```
n.create() {
        predecessor=null
        successor=n
}
```

Joining a ring (containing n')

```
n.join(n') {
        predecessor=null
        successor=n'.findSuccessor(n)
}
```

Node dynamics : Stability

Stabilizing the ring
-Runs periodically
-Informs nodes about newly joined nodes
-Fix finger tables
-Fix predecessors

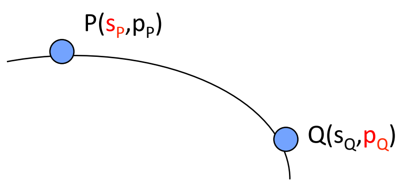
```
n.stabilize() {
        x=successor.predecessor
        if x in (n,successor]
                successor = x      // better successor found
        successor.notify(n)        // adapt predecessor of successor
}


n.notify(n') {             //n' thinks it is the predecessor of n
        if (predecessor==null) OR (n' in (predecessor,n[)
                predecessor = n'  // better predecessor found
}
```


3. Distributed Hash Tables
4. Chord

Node dynamics : Stability

```
n.stabilize() {
        x=successor.predecessor
        if x in (n,successor] successor = x    // better successor found
        successor.notify(n)                    // adapt predecessor of successor
}
n.notify(n') {             //n' thinks it is the predecessor of n
        if (predecessor==null) OR (n' in (predecessor,n[)
                predecessor = n'      // better predecessor found
}
```

Steady-State : no changes !



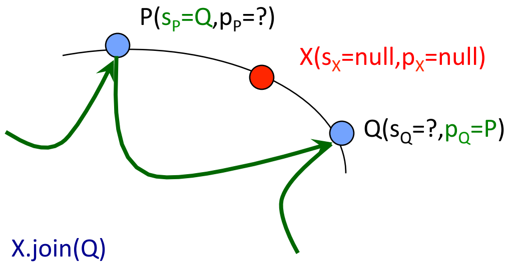
P.stabilize()
- adapts successor @P (s_P)
- adapts predecessor @Q (p_Q)

If successor and predecessor @P and Q correct,
i.e. s_P = Q, p_Q = P
x=successor.predecessor=(s_P).predecessor=$p_{(sP)}$=p_Q=P
-> x in (P,Q] ? -> FALSE
-> Q.notify(P)

P in (p_Q,Q[? -> P in (P,Q[? -> FALSE

NO UPDATES

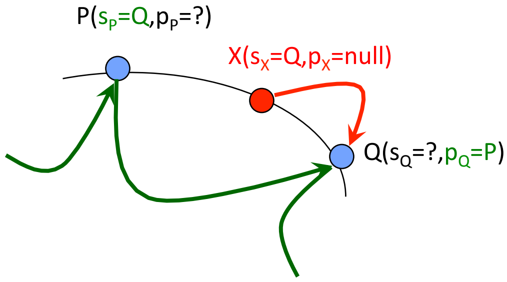
Node dynamics : Join



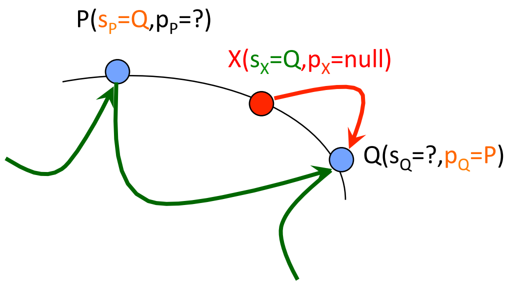
X joins, $P<X<Q$
Ring stable

X.join(Q)

-> p_X=null
-> s_X=Q.findSuccessor(X)=Q



Node dynamics : Join

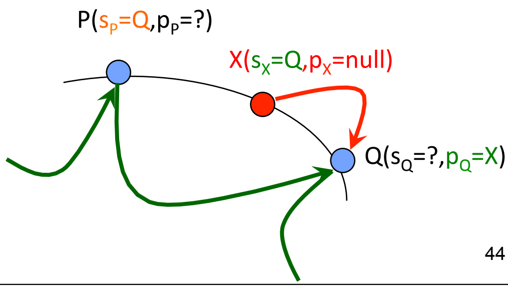


X.stabilize()

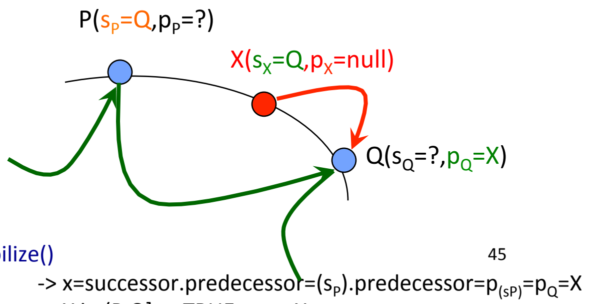
-> x=successor.predecessor=(s_X).predecessor=$p_{(sX)}=p_Q$=P
-> P in (X,Q] -> FALSE
-> Q.notify(X)

Q.notify(X)

-> X in (p_Q,Q[= (P,Q[-> TRUE -> p_Q=X



Node dynamics : Join

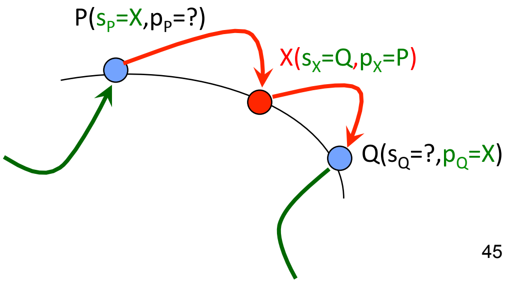


P.stabilize()

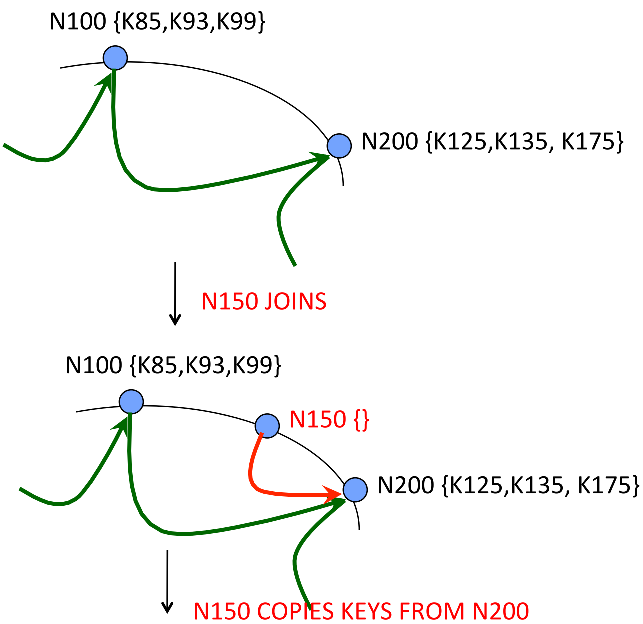
45

-> x=successor.predecessor=(s_P).predecessor=$p_{(sP)}=p_Q$=X
-> X in (P,Q] -> TRUE -> s_P=X
-> X.notify(P)

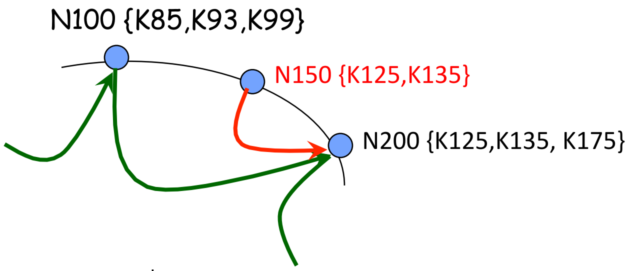
X.notify(P)
-> p_X == null -> p_X=P



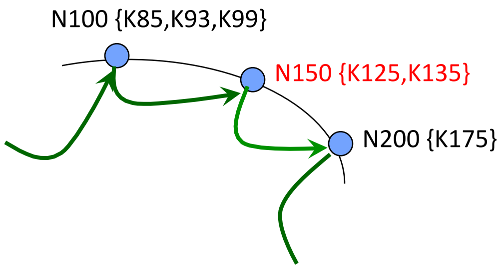
Node dynamics : Join example



Node dynamics : Join example



N100, N150 and N200 stabilize



Node dynamics : Stability


```
n.fixFingers() {
        next++                              // f[next] will be fixed
        if (next>m) next = 1
        finger[next]=findSuccessor(n+2^(next-1))
}

n.checkPredecessors() {                 // allows active predecessor to enter
        if (predecessor failed) predecessor = null
}
```

Node dynamics : Failures

As long as each node knows successor -> correct operation of lookup

Consider finger table @P

P+1	**Na**
P+2	**Nb**
P+4	**Nc**
P+8	**Nd**
P+16	**Ne**
P+32	**Nf**

Suppose

- lookup(id) is launched @P, Nd<id<Ne
- nodes Na, Nb, Nc, Nd and Ne fail simultaneously

-> successor(id)=Nf (instead of Ne)

-> ERROR

Robustness ?

- each node has list of r first successors
- to corrupt ring -> r nodes have to fail simultaneously
- changes needed to stabilization and lookup code

Node dynamics : Failures (pseudo code)


```
n.findSuccessor(id) {
        if id in (n,successor]  return successor
        else {
                try {
                        n'=closestPreceedingNode(id)
                        return n'.findSuccessor(id)
                } catch(TimeoutException e) {
                        invalidate n' from finger
                        and successor table
                        return n.findSuccessor(id)
                }
        }
}

n.closestPreceedingNode(id) {
        for i=m downto 1
                if finger[i] in (n,id] return finger[i]
        for i=r downto 1
                if successor[i] in (n,id] return successor[i]
        return n
}
```

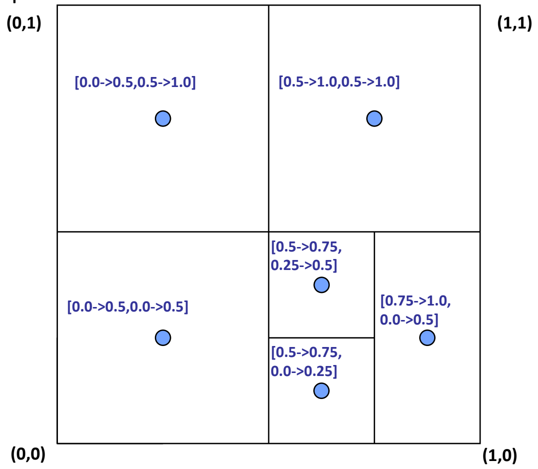
Multi-dimensional routing

CAN : Content addressable network

Basic idea

use d-dimensional space to map keys
- node is responsible for d-dimensional hypercube
- Cartesian coordinates used
- space is d-torus

Example 2D



Multi-dimensional routing

Mapping content to node

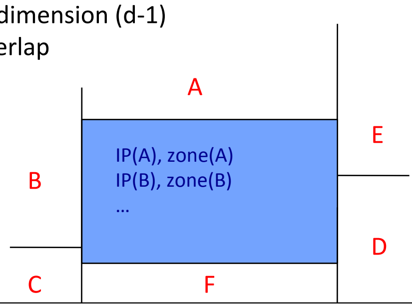
- Key-Value pair <K,V> mapped to point P in Cartesian space
  P=hash(K)
- P belongs to region owned by node N
- N stores <K,V>

Retrieving entry for key K

- compute P=hash(K)
- if P not owned by requester or neighbours
  -> route request in the CAN network

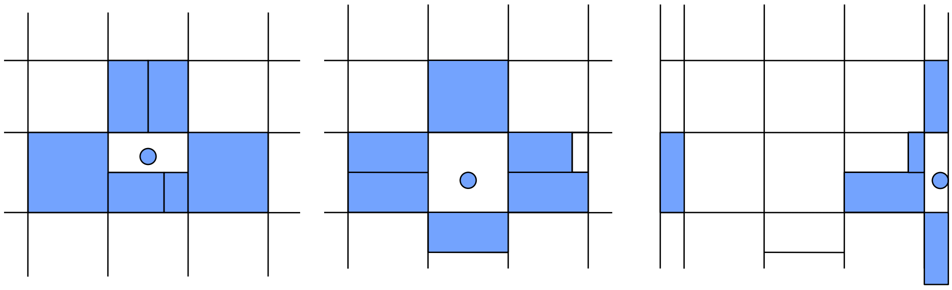
Self-organizing routing

node learns and stores set of neighbours
- neighbour : shares hyperplane of dimension (d-1)
- -> in (d-1) dimension, intervals overlap
- info stored per neighbour n
  - IP(n), zone(n)



Multi-dimensional routing

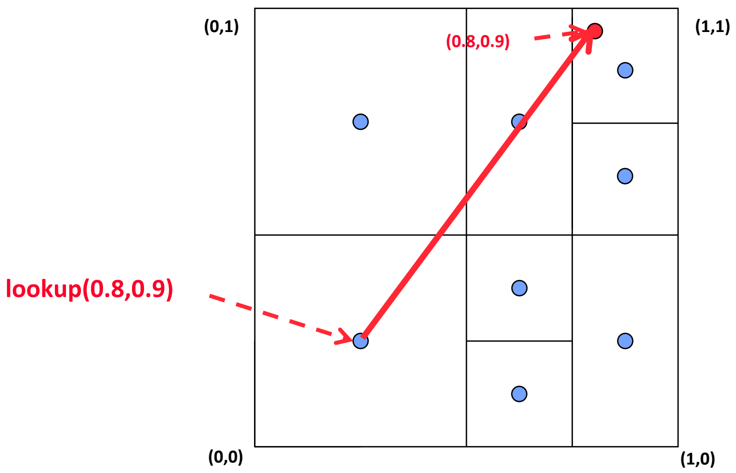
Neighbour nodes : examples



Suppose equally partitioned zones (regular d-dimensional grid)
-Number of neighbours : 2d
-Average routing path length = (d n $^{1/d}$)/4

Multi-dimensional routing

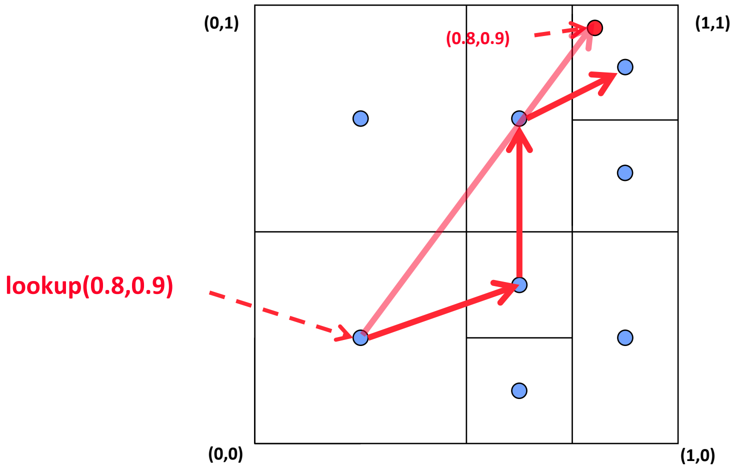
Routing in d dimensions
Follow straight line from source to destination



Multi-dimensional routing

Routing in d dimensions

Forward to closest neighbour (greedy forwarding)
-> each node keeps track of at least d neighbours
-> space complexity O(d)
-> lookup cost $O(dN^{1/d})$ (N number of nodes)



Node dynamics : Joining the CAN

1. Find an active node in the CAN (Bootstrapping)

- CAN has DNS name
- DNS resolves to IP address of bootstrapping node
- each bootstrap node has list of probably active nodes
  - nodes that joined before
  - nodes can notify before they leave
  - nodes supposed to reply to ping-messages from bootstrap
- bootstrap node replies to “join” by random selection from the active node list
- essential to minimize activity of the bootstrap node

2. Find a zone to care for

3. Publish new node to neighbours

Node dynamics : Joining the CAN

3. Distributed Hash Tables
5. CAN

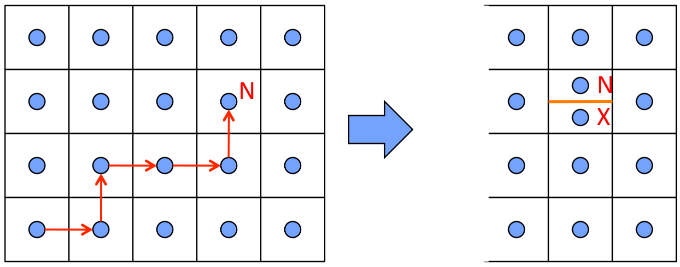
1. Find an active node in the CAN (Bootstrapping)

2. Find a zone to care for

- X selects random point P
- X sends JOIN(P) -> arrives at node N currently responsible for P
- N splits zone in 2
- N sends to X
    - range of keys X will be responsible for
    - <Key,Value> pairs X will handle

3. Publish new node to neighbours

X -> JOIN(P)

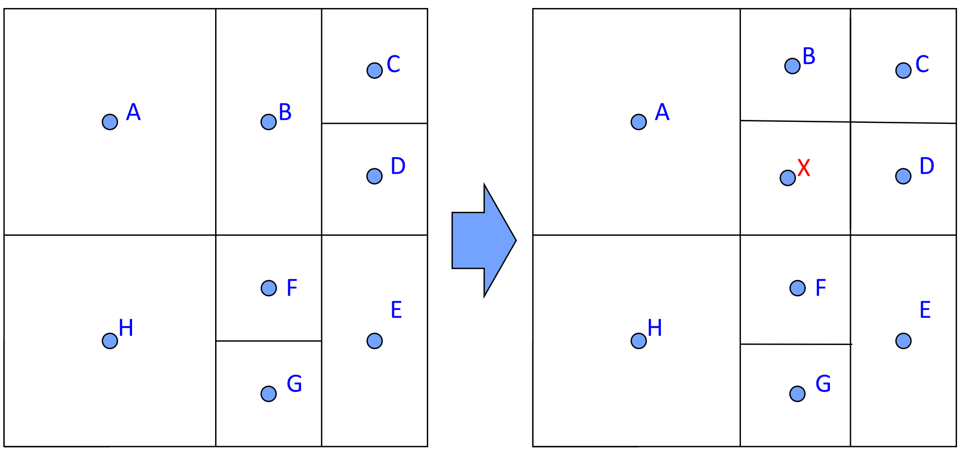


57

Node dynamics : Joining the CAN

3. Distributed Hash Tables
5. CAN

Joining : example



neighbour(B) = {A,C,D,F,G}

neighbour'(B) = {A,C,G,X}
neighbour'(X) = {A,D,F,B}

58

Node dynamics : Joining the CAN

1. Find an active node in the CAN (Bootstrapping)
2. Find a zone to care for
3. **Publish new node to neighbours**

- X and P update neighbour set

$$neighbour'(X) \subset \left(neighbour(P) \cup \{P\}\right)$$

$$neighbour'(P) \subset \left(neighbour(P) \cup \{X\}\right)$$

- X and P send immediate update on zone(X) and zone(P) to neighbour(P) and neighbour(P')

- every node sends periodic refreshes (soft-state updates)
  - n + zone(n)
  - for each neighbour : ID + zone

Joining is a LOCAL operation

-> only neighbours are affected

-> O(d)

Node dynamics : Leaving the CAN

Normal exit

- X checks if zone(X) can be merged with zone(n)
- if so, zone(X) is handed to n
- if not, a neighbour zone will take care of multiple zones
- which neighbour ?

  minimize maximum zone size of neighbours

Unexpected departure (failure)-> Immediate take-over

Failure detected through absence of update messages

When node N detects X has departed (probably multiple N's !)

- start take over timer

```
timeout = C zoneSize(N)
when time-out {
    N sends TAKEOVER(zoneSize(N)) to neighbour(X)
    update zone(N) info
}
```

- on receipt of TAKEOVER(zoneSize(Y)) at node N

```
if(zoneSize(Y)<zoneSize(N)) {
    cancel timer @ N
    update neighbour info
}
else send TAKEOVER(zoneSize(N)) to neighbour(X)
```

Node dynamics: Leaving the CAN

Unexpected departure (failure)

Immediate take-over algorithm OK to handle single node failures
If less than 50% of neighbour(X) nodes reachable
-> locate active neighbours through expanding ring search
prior to take-over

Expanded ring search @ node N

```
TTL = 1
do {
        broadcast request for info to neighbours(N)
        TTL++
} while neighbourInfo incomplete
```

Avoid too much fragmentation
background zone-reassignment process

Advanced CAN

1. Increase d
   - reduces hop count and latency
   - increases node state

2. Use multiple "realities"
   - as if separate instances of CAN are running on same node infrastructure
   - in each reality, node is responsible for different zone
     -> improved data availability, robustness
     -> reduced latency

3. RTT based routing metrics
   - use RTT weighted distances when forwarding
     -> favours low latency paths
     -> lower latency

4. Overloading coordinate zones
   - multiple nodes are responsible for same zone ("peers")
   - reduces number of zones
     -> reduced path length
     -> reduced latency
     -> improved fault tolerance

Advanced CAN

5. Multiple hash functions
   - k different hash functions map same value to k nodes
   - queries are sent to k nodes
     -> improved latency
     -> improved data availability
   - cost : larger node state, increased query traffic
   - instead of launching parallel queries: start with query to closest node

6. Organize coordinate space based on physical network layout

7. Uniform coordinate partitioning
   prior to splitting zone, check whether neighbour could be split
   (i.e. has larger zone)
     -> better load balancing

8. Caching and replication to manage hot spots
   - cache recently accessed data in node, and check cache before forwarding
   - replicate frequently accessed data in neighbouring nodes

Research challenges

1. Appropriate distance function
2. Keep system structure simple under frequent joins/leaves
3. Fault tolerance measures
4. Concurrent changes
5. Proximity routing (adapt logical routing to physical topology)
6. Cope with malicious nodes
7. Indexing and keyword search (instead of ID's)