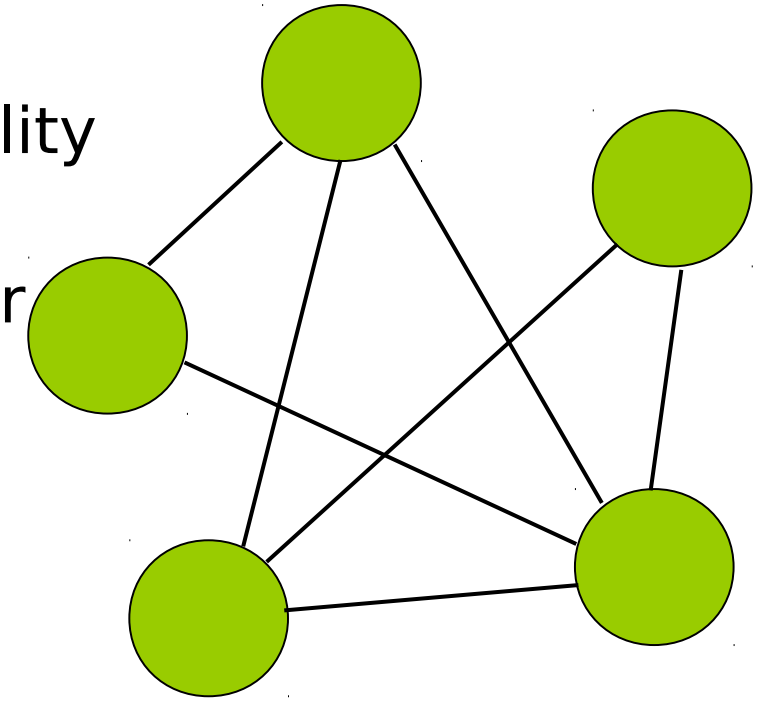
CS 347: Distributed Databases and Transaction Processing

Notes 05: P2P Systems

Hector Garcia-Molina

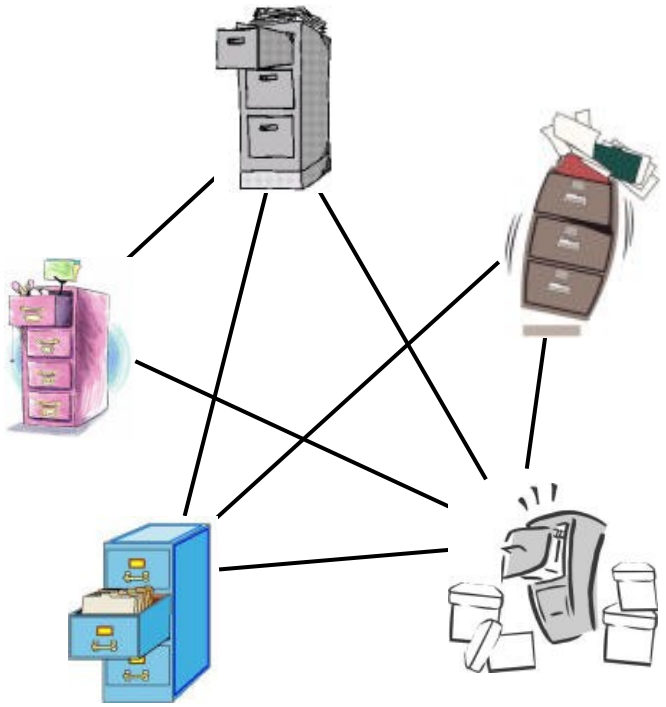
Peer To Peer Systems

- Distributed application where nodes are:
  - Autonomous
  - Very loosely coupled
  - Equal in role or functionality
  - Share & exchange resources with each other

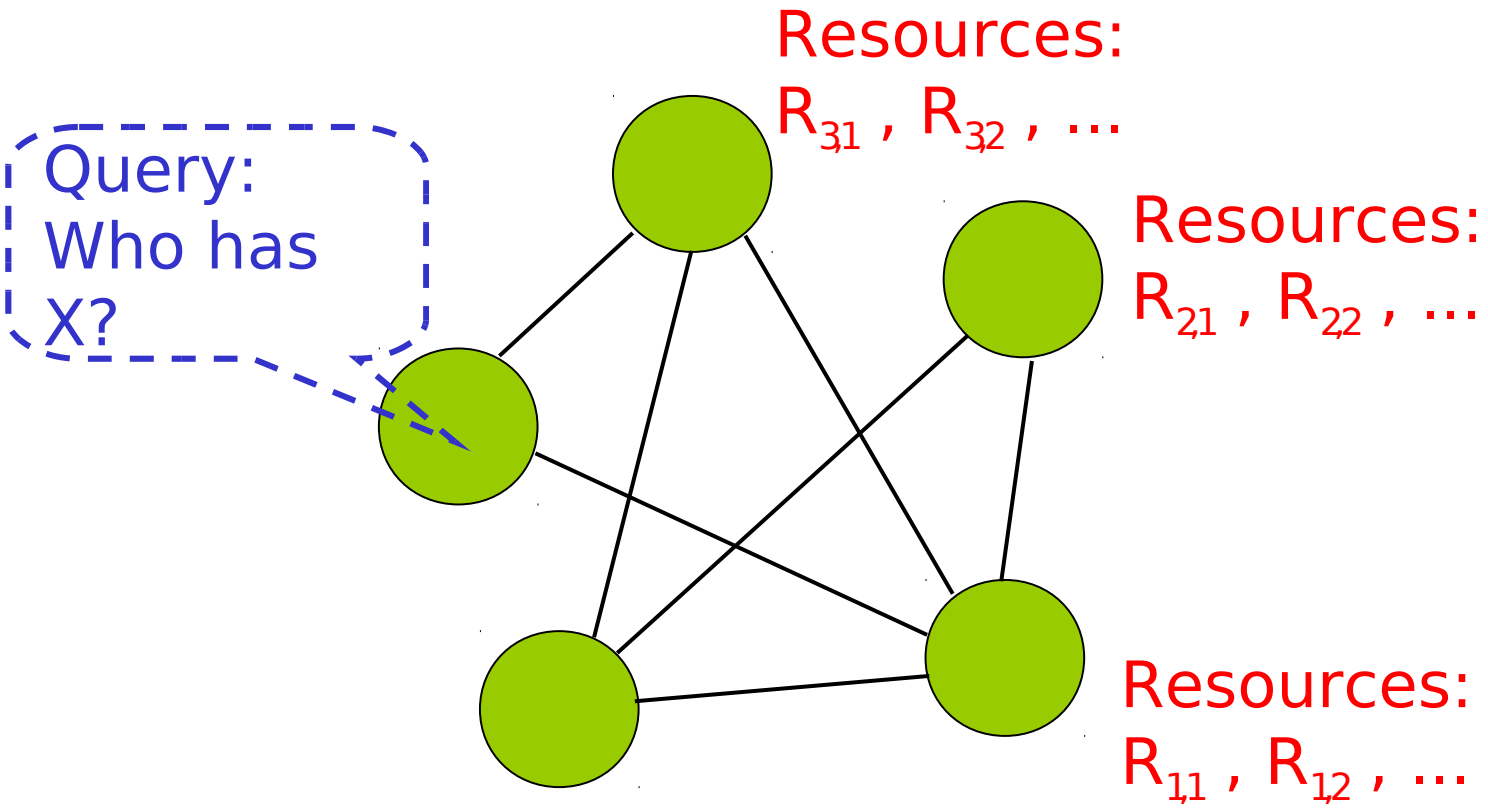


Related Terms

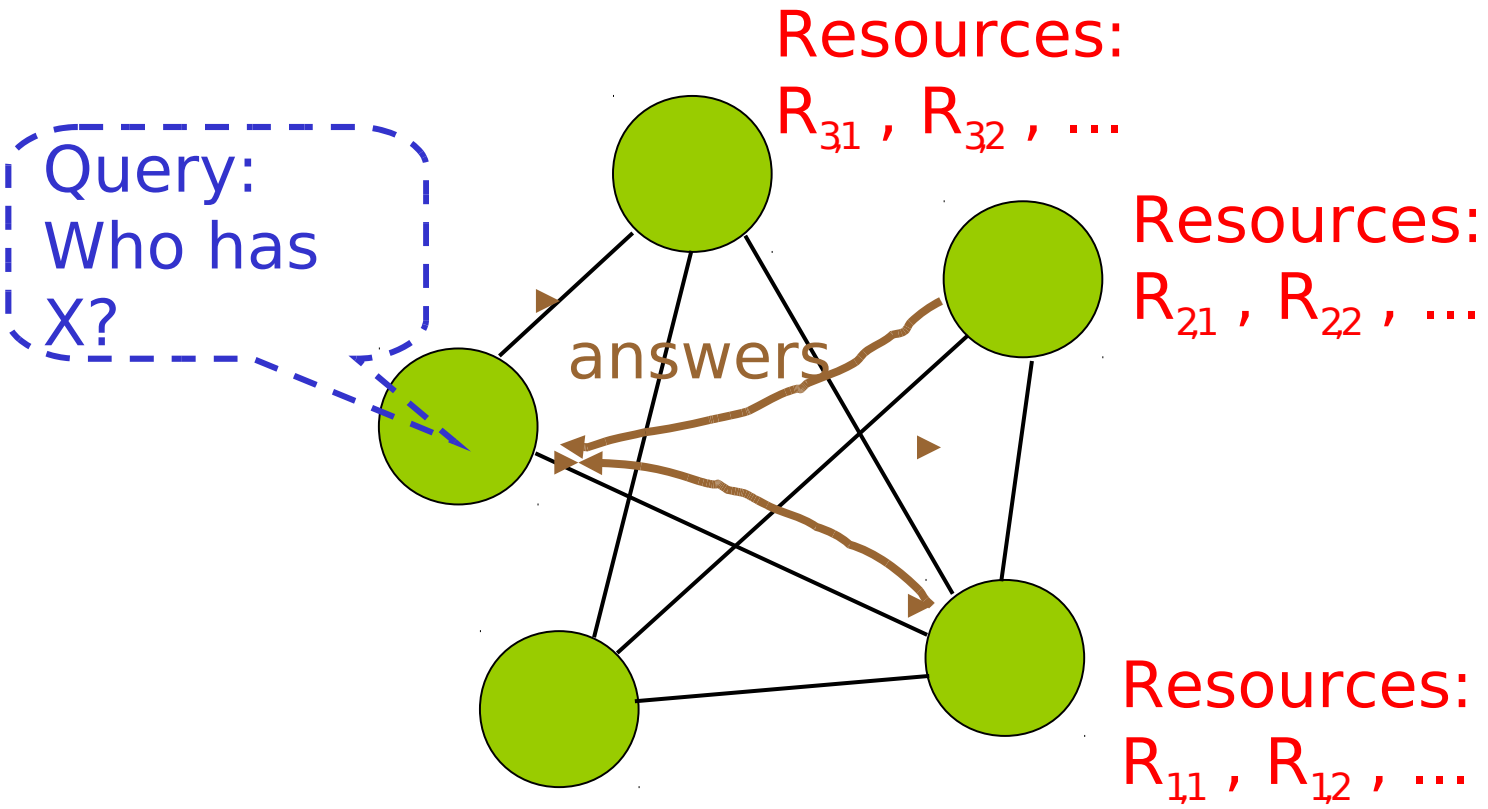
- File Sharing
  - Ex: Napster, Gnutella, Kaaza, E-Donkey, BitTorrent, FreeNet, LimeWire, Morpheus
- Grid Computing
- Autonomic Computing



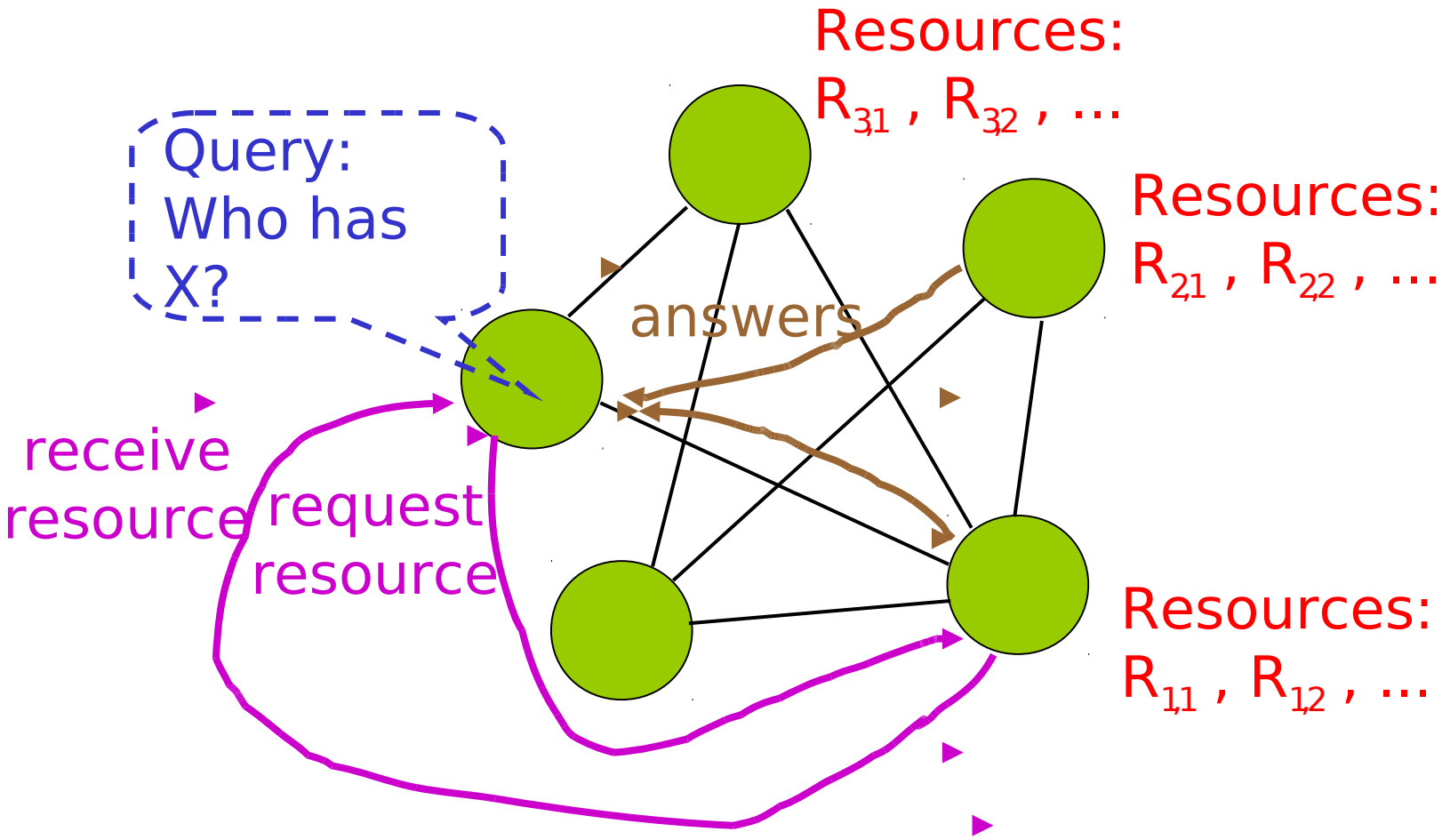
Search in a P2P System



Search in a P2P System



Search in a P2P System



Distributed Lookup

- Have <k, v> pairs
- Given k, find matching values

k	v
1	a
1	b
4	a
7	c
3	a
1	a
4	d

lookup(4) = {a, d}

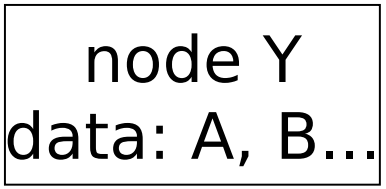
Data Distributed Over Nodes

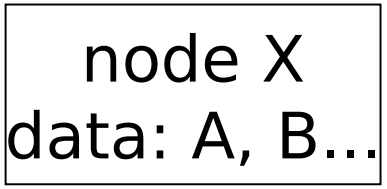
- N nodes
- Each holds some <k,v> data

Notation

- X.func(params) means RPC of procedure func(params) at node X
- X.A means send message to X to get value of A
- If X omitted, we refer to local procedure or data structures

```
...
B := X.A
A := X.f(B)
...
```





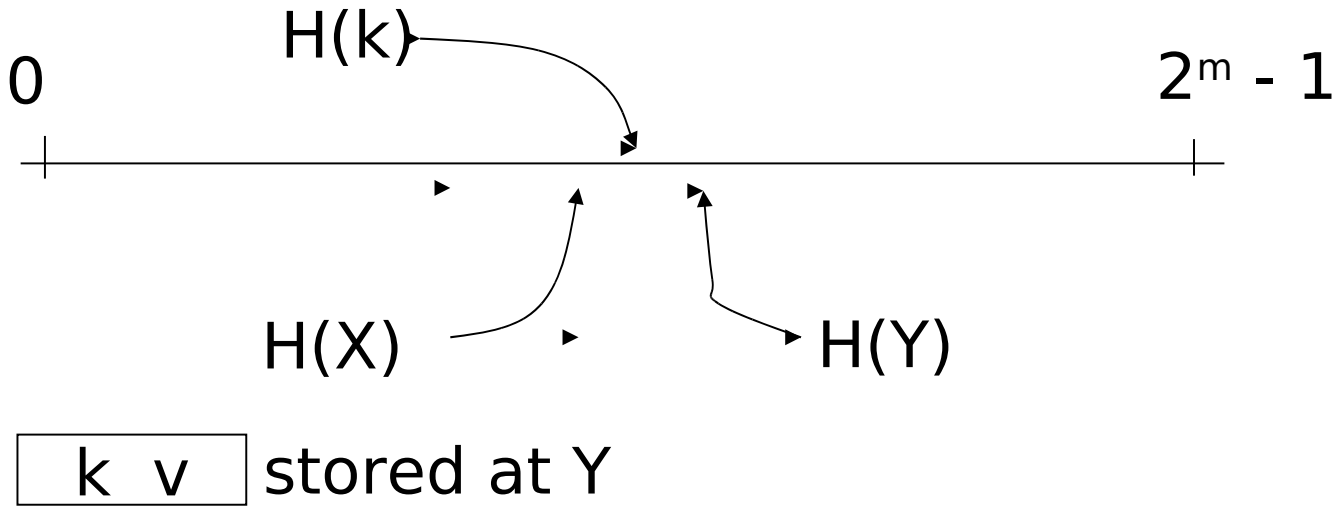
Distributed Hashing

- Chord
- Replicated HT

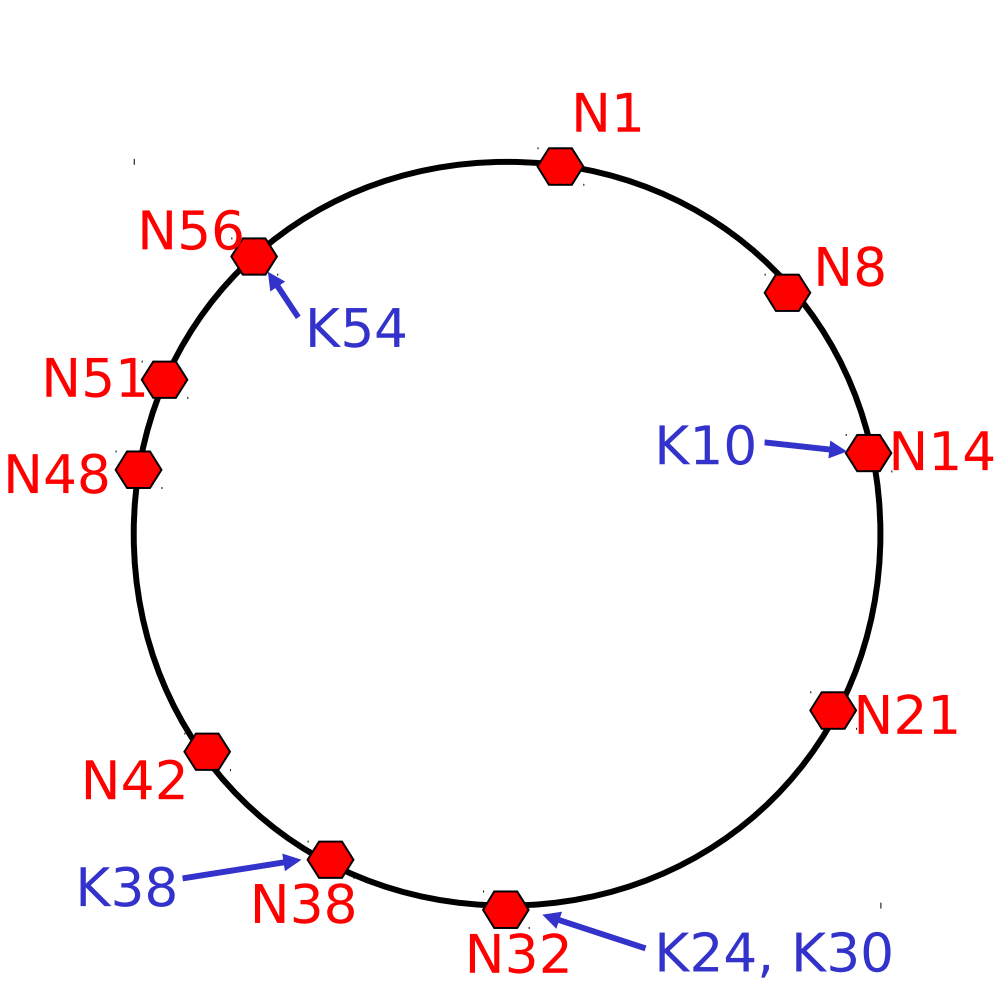
Chord Paper:
Ion Stoica, Robert Morris, David Liben-Nowell, David R. Karger,
M. Frans Kaashoek, Frank Dabek, Hari Balakrishnan,
Chord: A Scalable Peer-to-peer Lookup Protocol for Internet Applications.
IEEE/ACM Transactions on Networking
http://pdos.csail.mit.edu/chord/papers/paper-ton.pdf

Hashing Values and Nodes

- H(v) is m-bit number (v is value)
- H(X) is m-bit number (X is node id)
- Hash function is “good”



Chord Circle

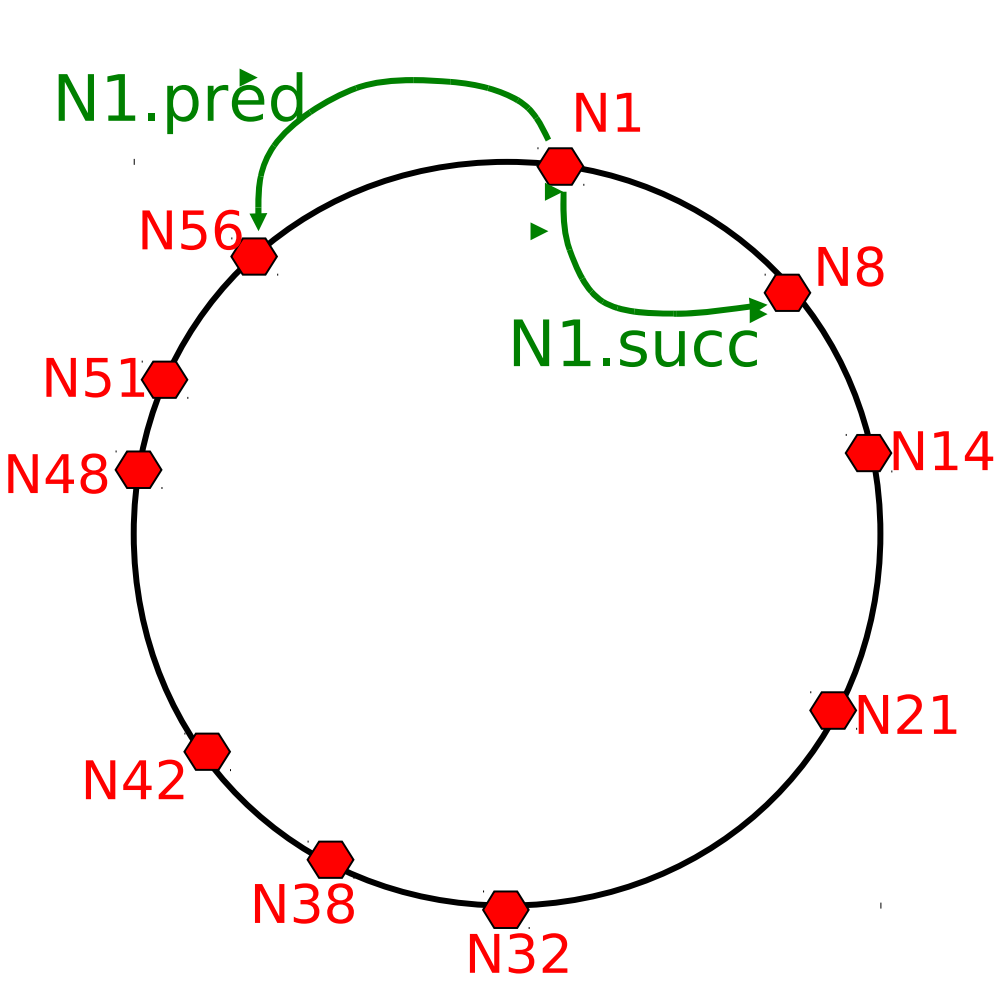


Rule

- Consider nodes X, Y such that Y follows X clockwise
- Node Y is responsible for keys k such that H(k) in (H(X), H(Y)]

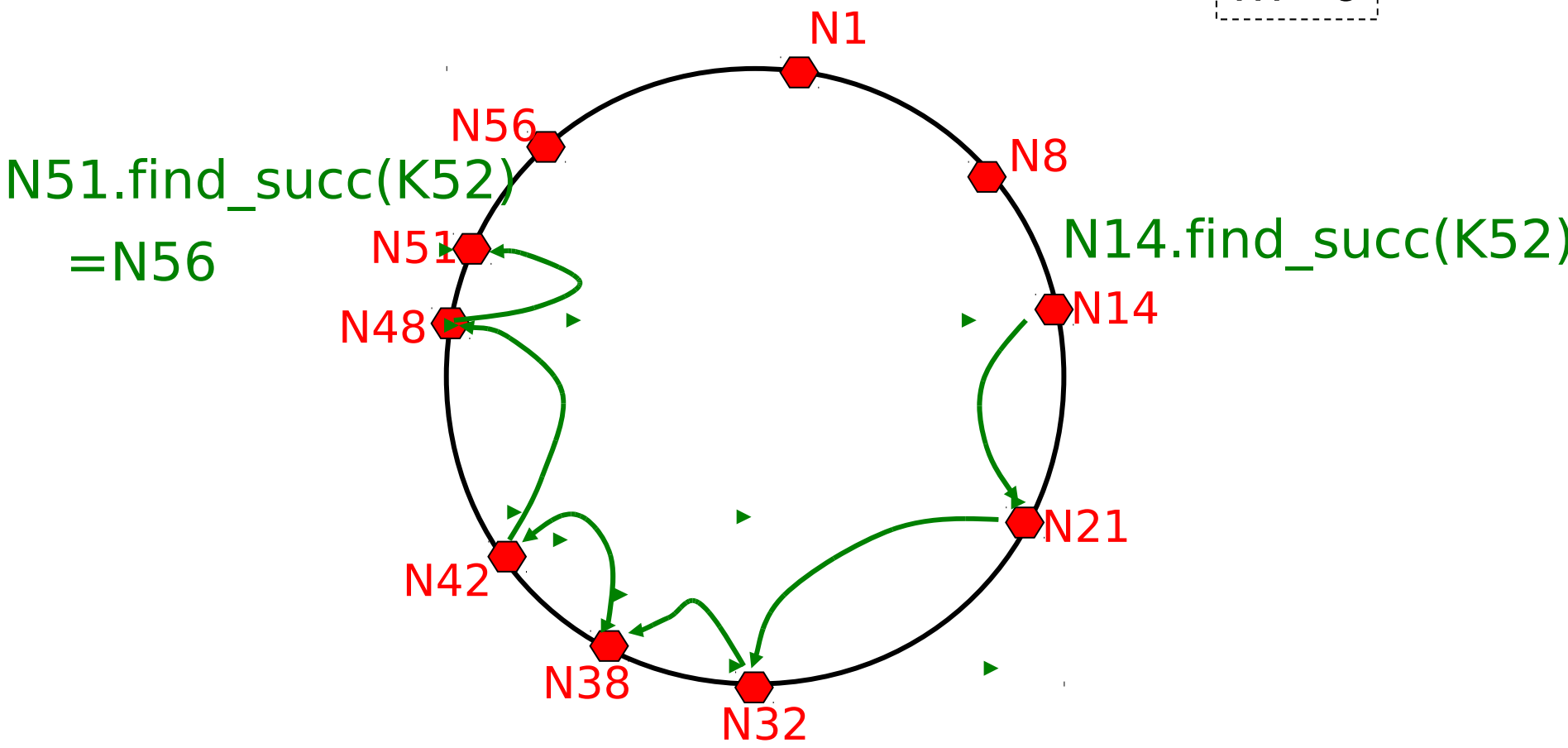
use hashed values...
e.g., N54 is node whose id hashes to 54.

Succ, pred links



Search using succ links

m=6



Code

- X.find_succ(k)

```
    if k in (X, succ]
        return succ
    else
        return succ.find_succ(k);
```

Notation: Use of Hashed Values

- X.find_succ(k)

```
if k in (X, succ]
        return succ
else
        return succ.find_succ(k);
```

should be

- X.find_succ(k)

```
if H(k) in (H(X), H(succ)]
        return succ
else
        return succ.find_succ(k);
```

Looking Up Data

- X.DHTlookup(k)

```
Y := find_succ(k);
S := Y.lookup(k);
return S;
```

- Y.lookup(k)

  returns local values associated with k

Another Version

- X.DHTlookup(k)
  X.find_succ(k, X);
  wait[ans(k, S)]: return S;
- X.find_succ(k, Y)
  if k in (X, succ]
    Y.ans(k, succ.lookup(k))
  else
    succ.find_succ(k, Y);

combines find and lookup; avoids chain of returns

Inserting Data

- X.DHTinsert(k, v)
  Y := find_succ(k);
  Y.insert(k, v);
- Y.insert(k, v)
  inserts [k,v] in local storage

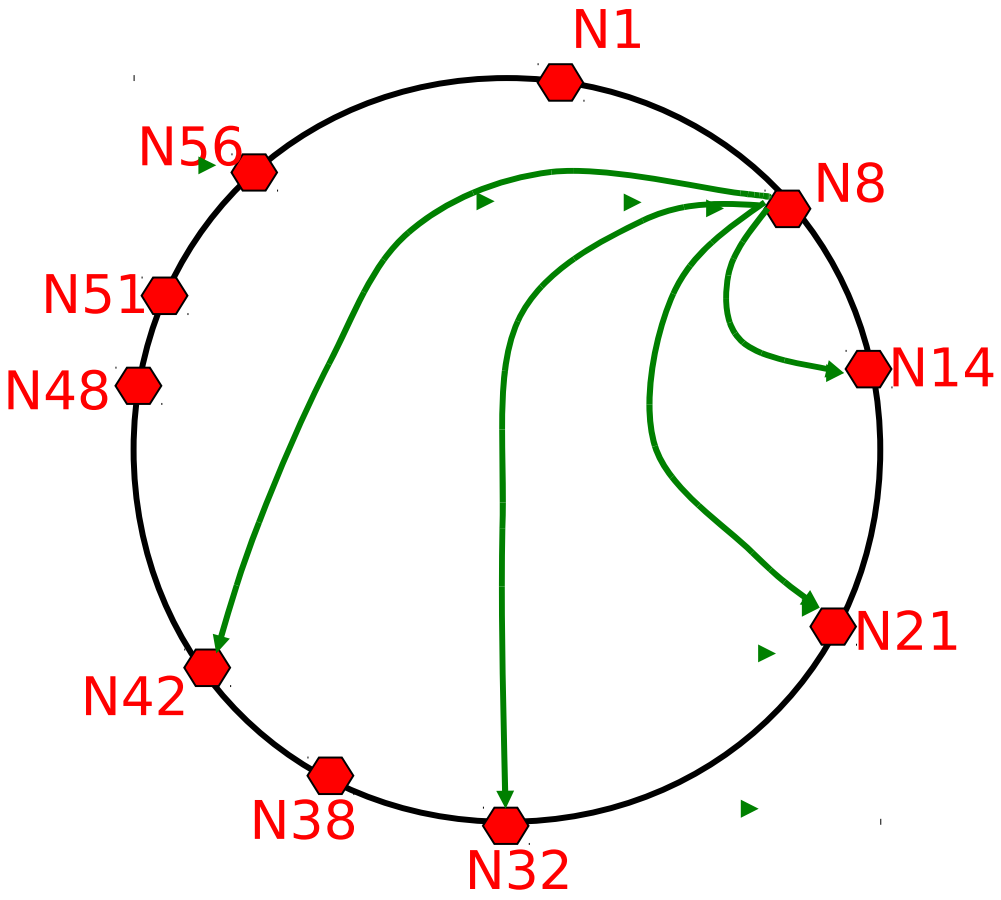
Another Version

- X.DHTinsert(k, v)

```
    if k in (X, succ]
        succ.insert(k,v)
    else
        succ.DHTinsert(k, v);
```

Finger Table

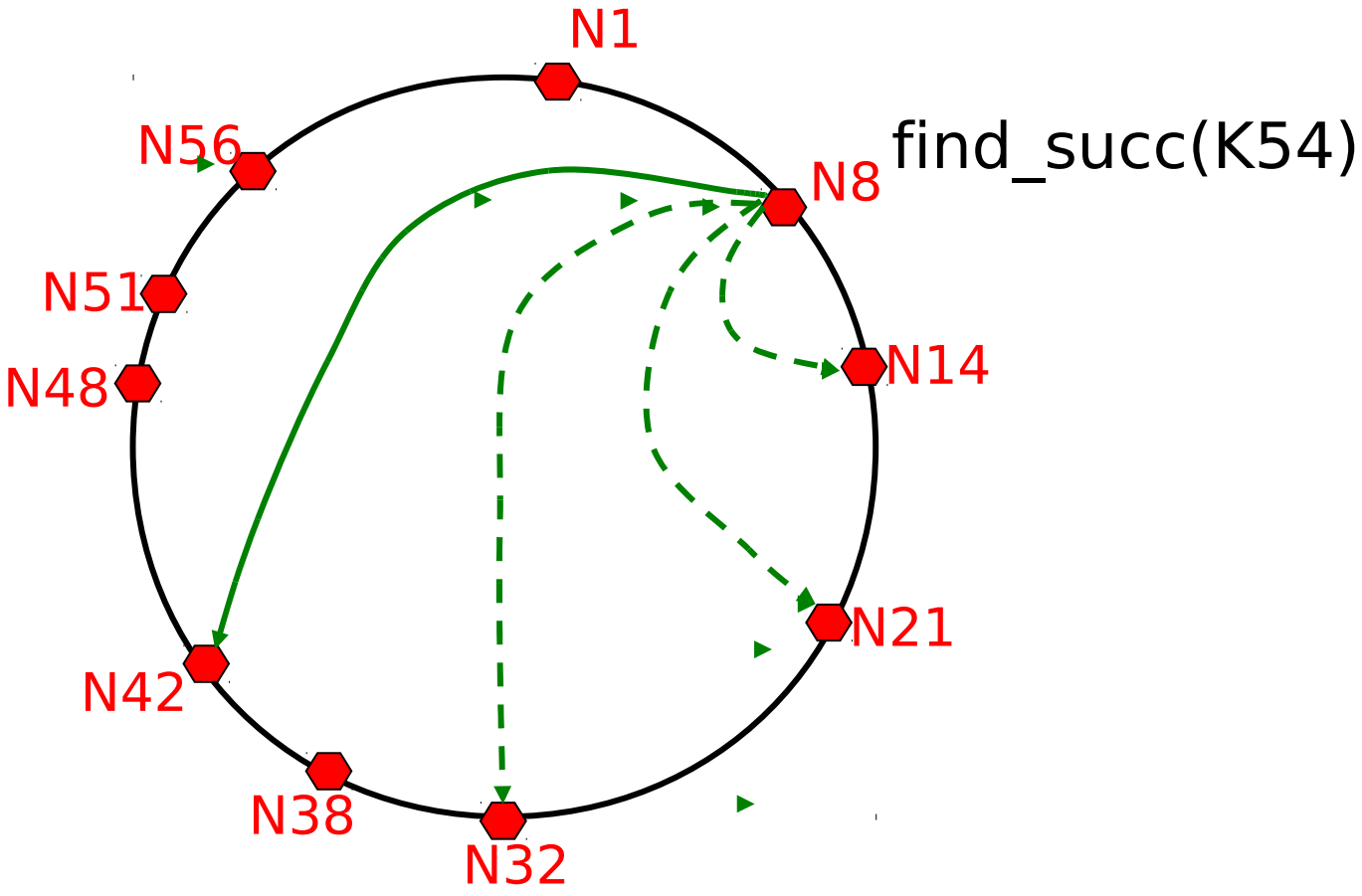
m=6



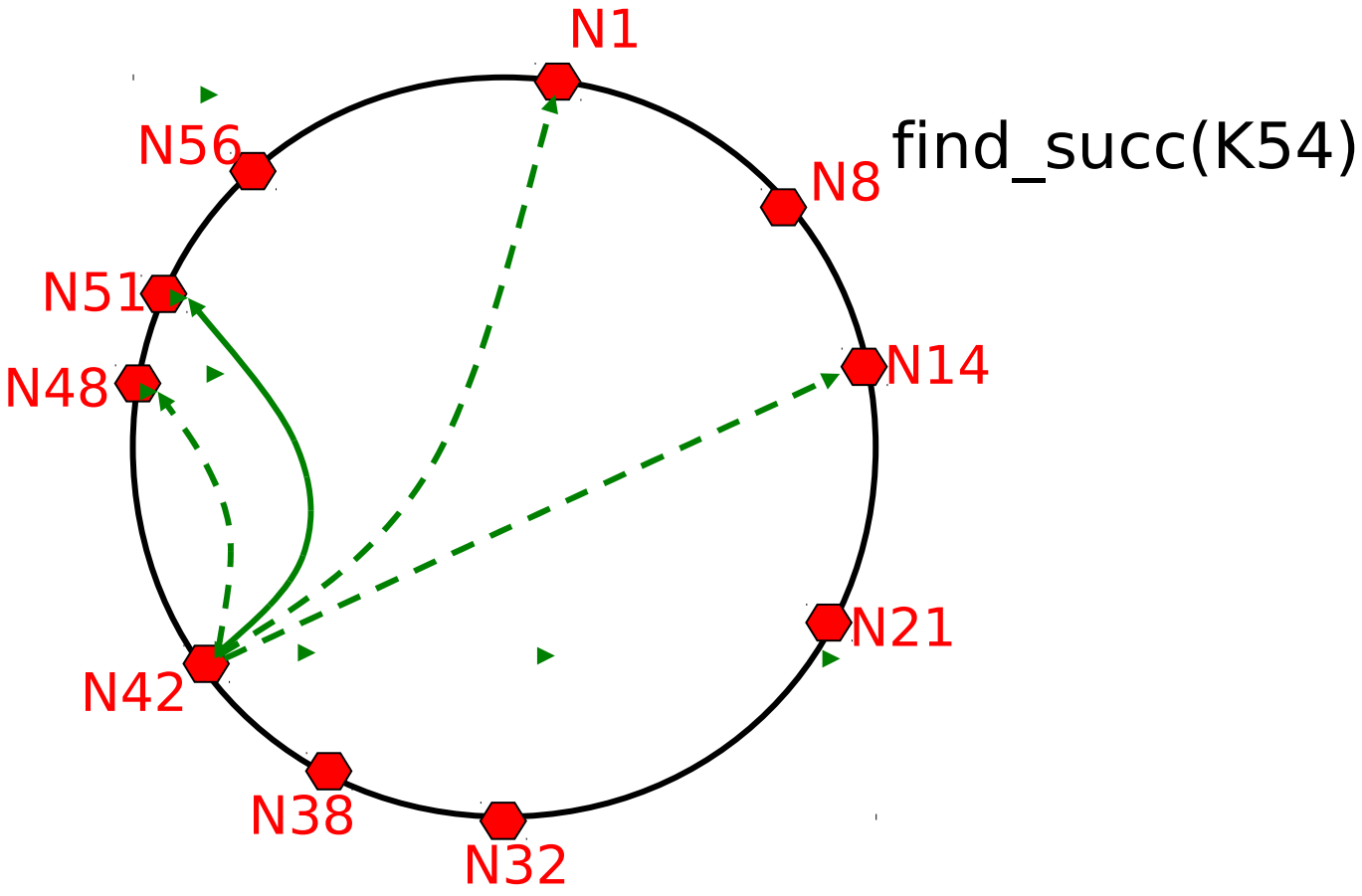
finger table for N8

N8+1	N14
N8+2	N14
N8+4	N14
N8+8	N21
N8+16	N32
N8+32	N42

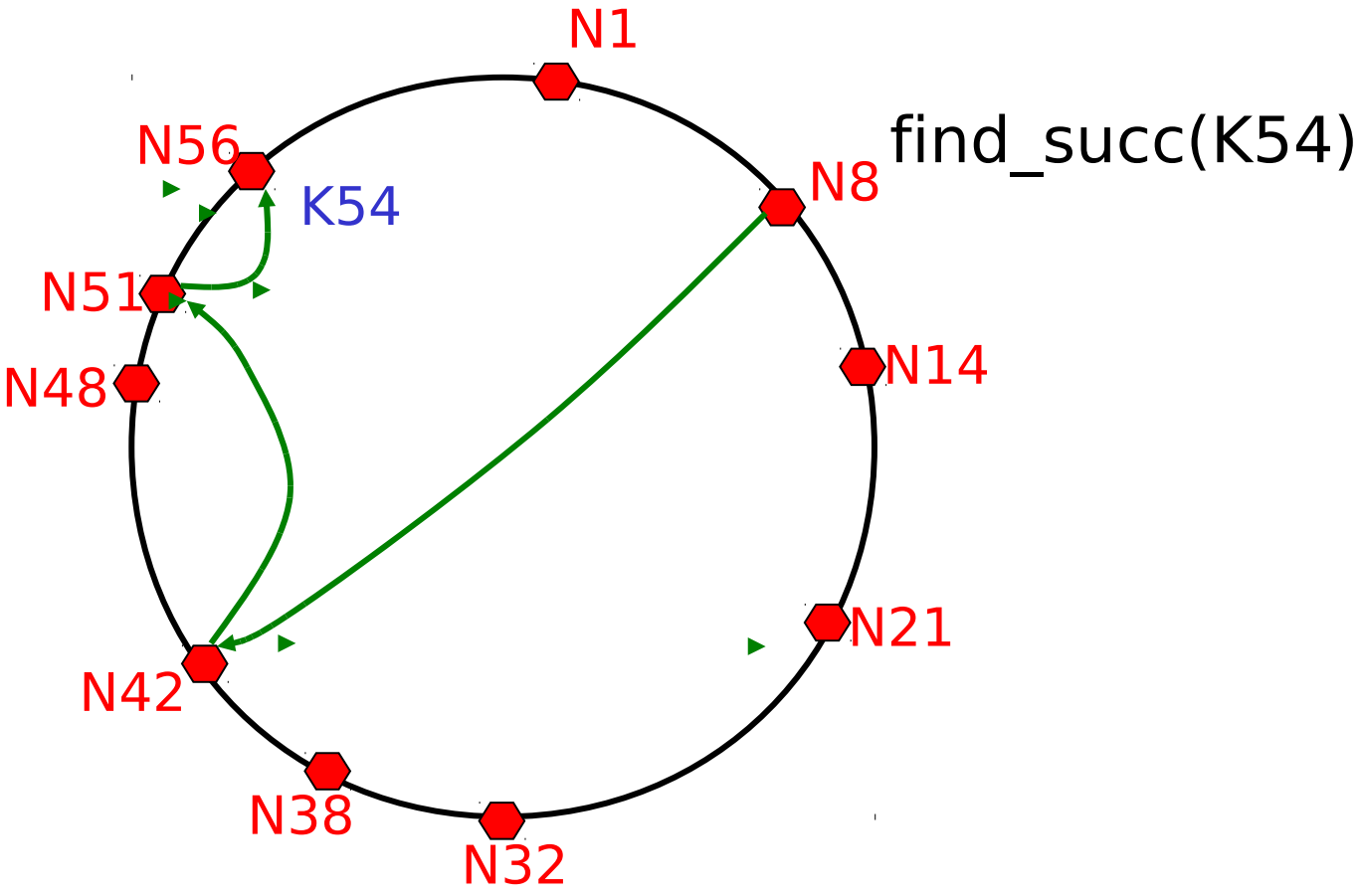
Example



Example



Example



Code

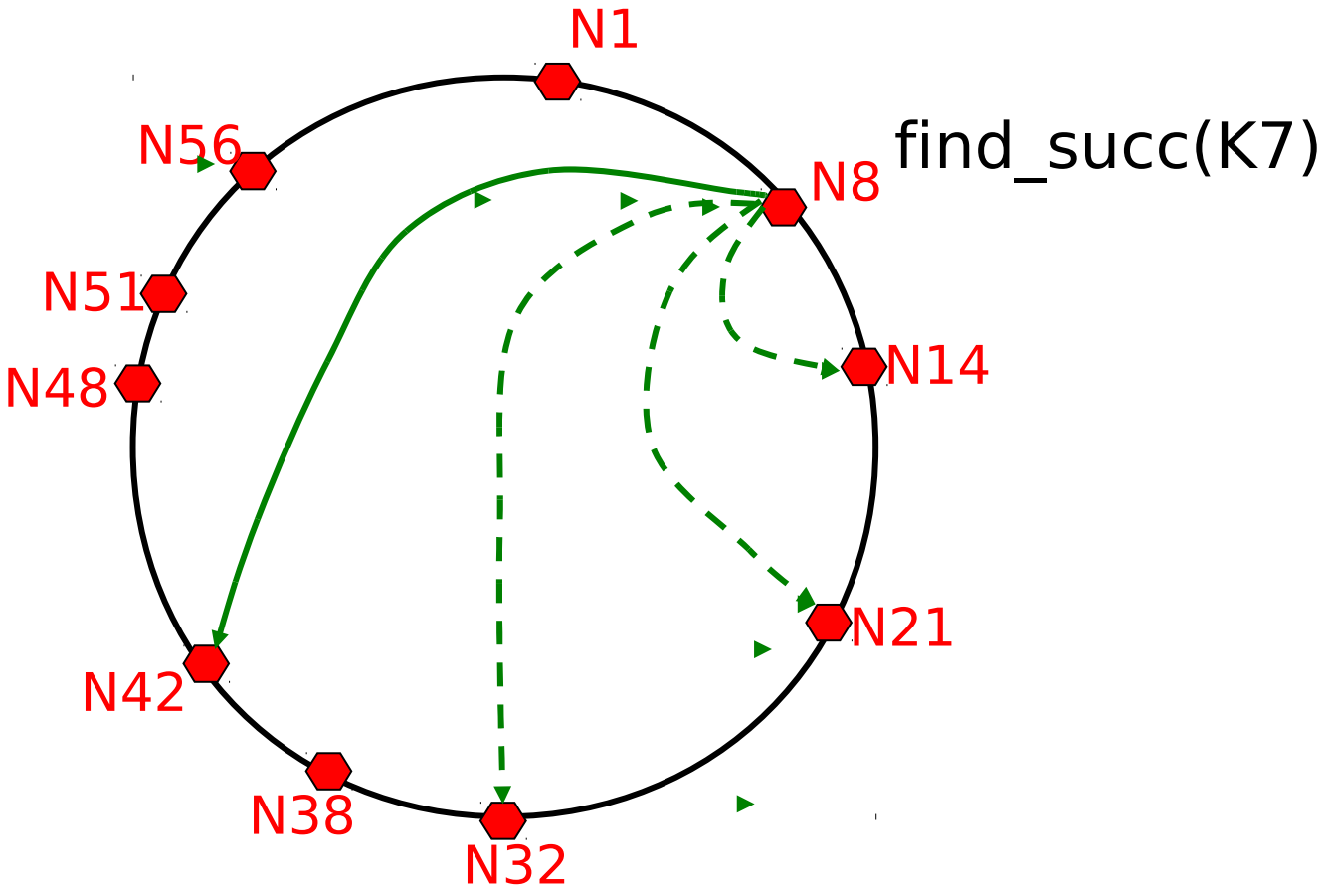
- X.find_succ(id)

```
if id in (X, succ]  return succ
else
Y := closest_preceed(id);
return Y.find_succ(id);
```

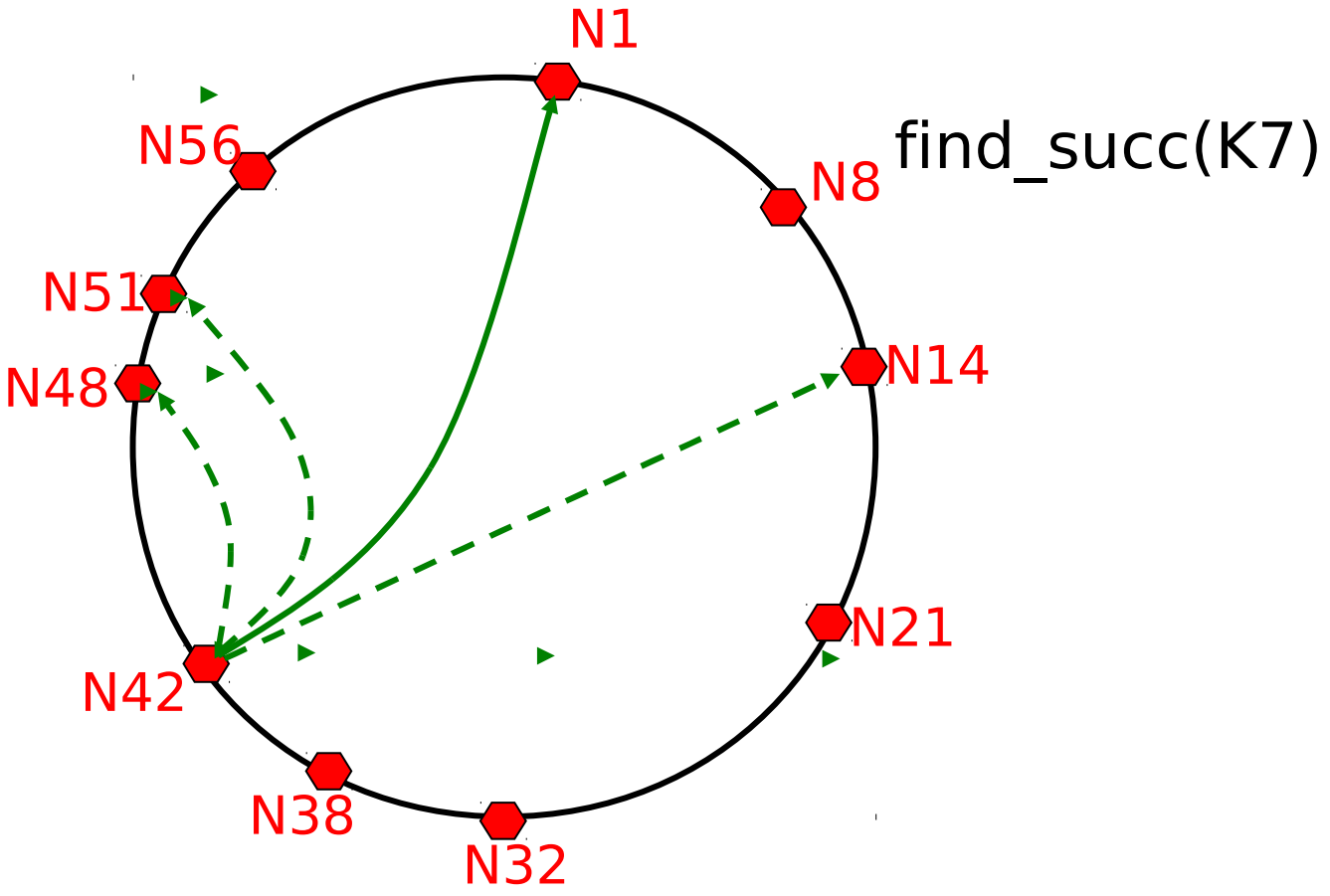
- X.closest_preceed(id)

```
for i := m downto 1
if finger[i] in (X,id) return finger[i];
return X;
```

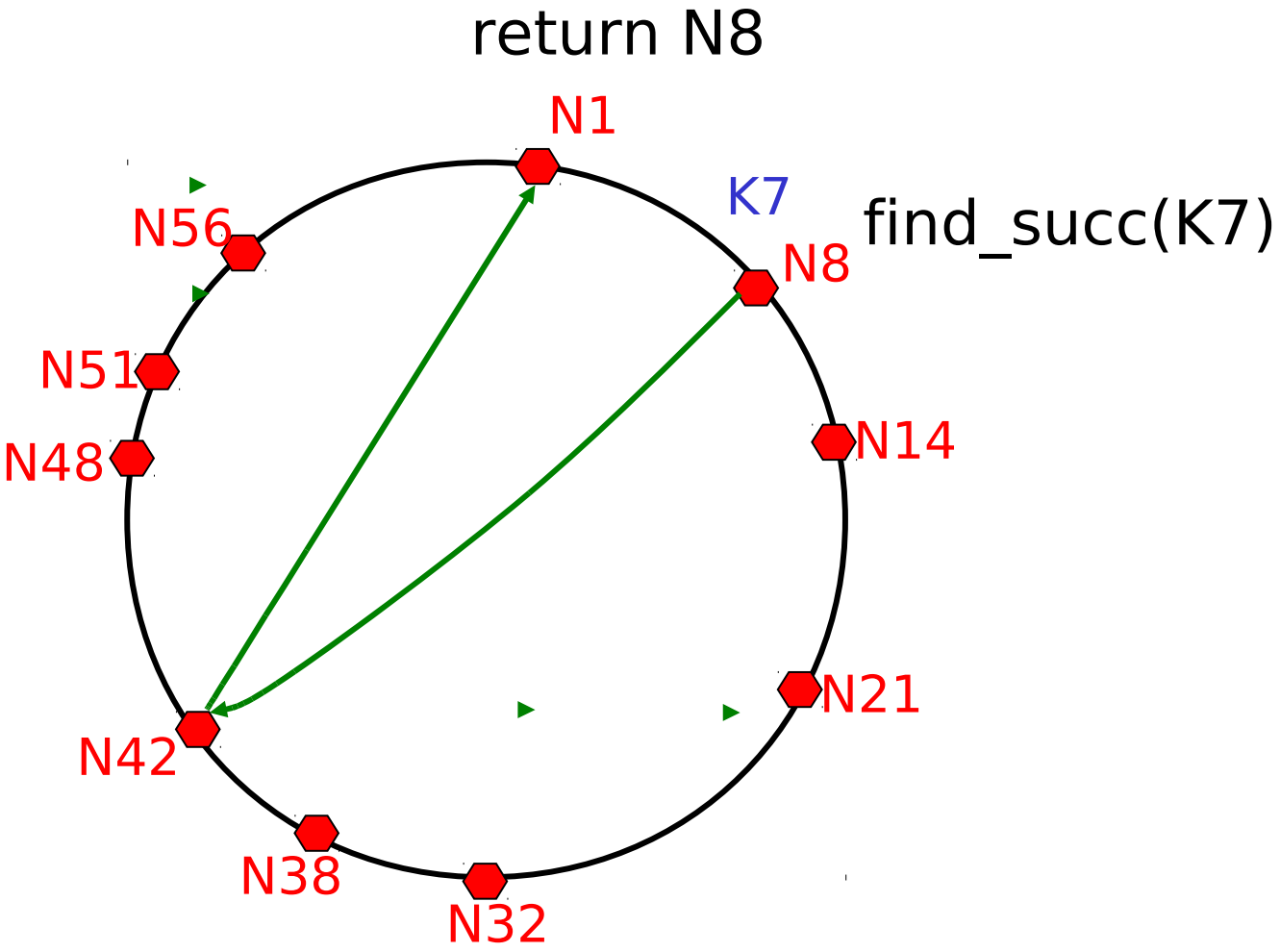
Another Example



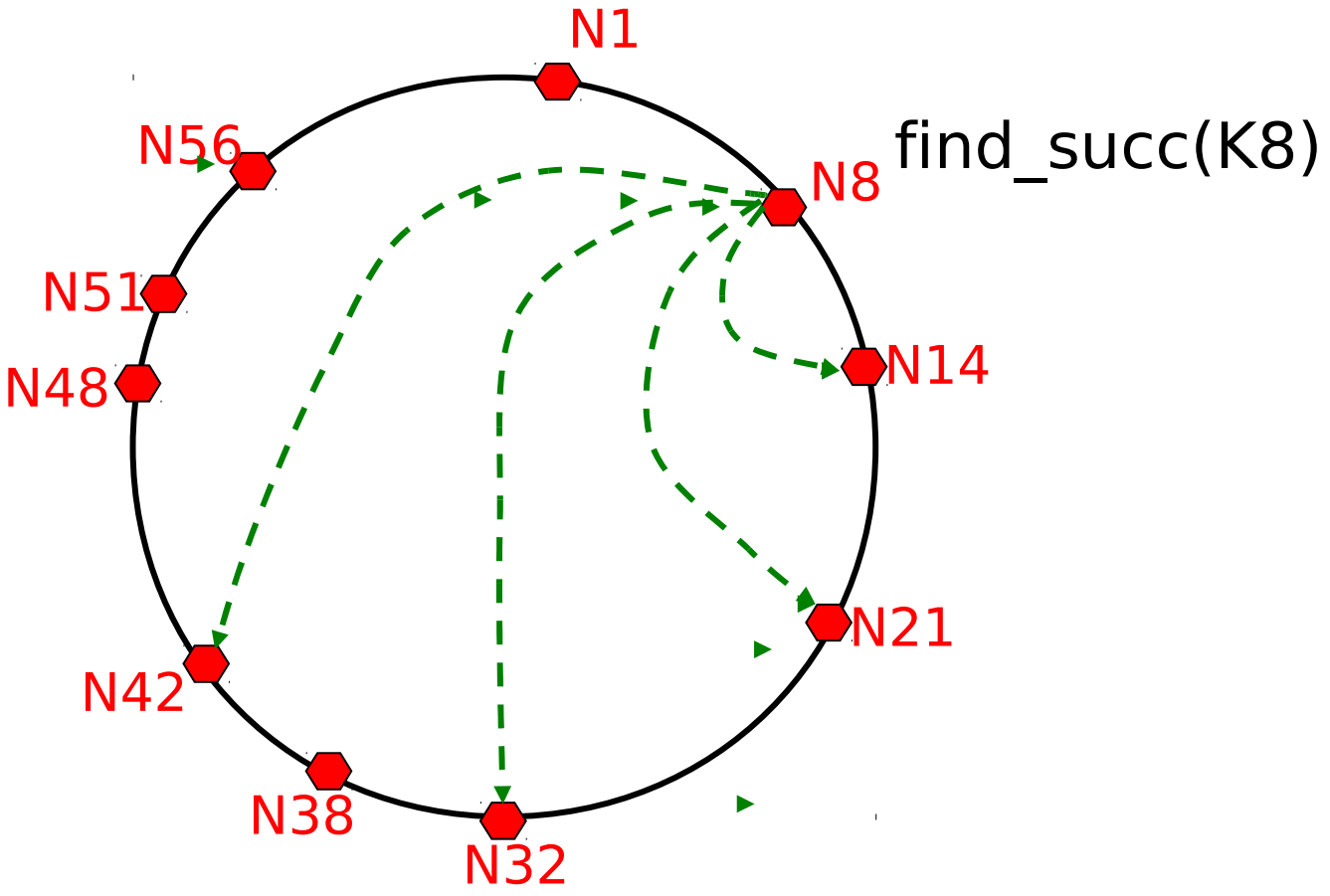
Example



Example

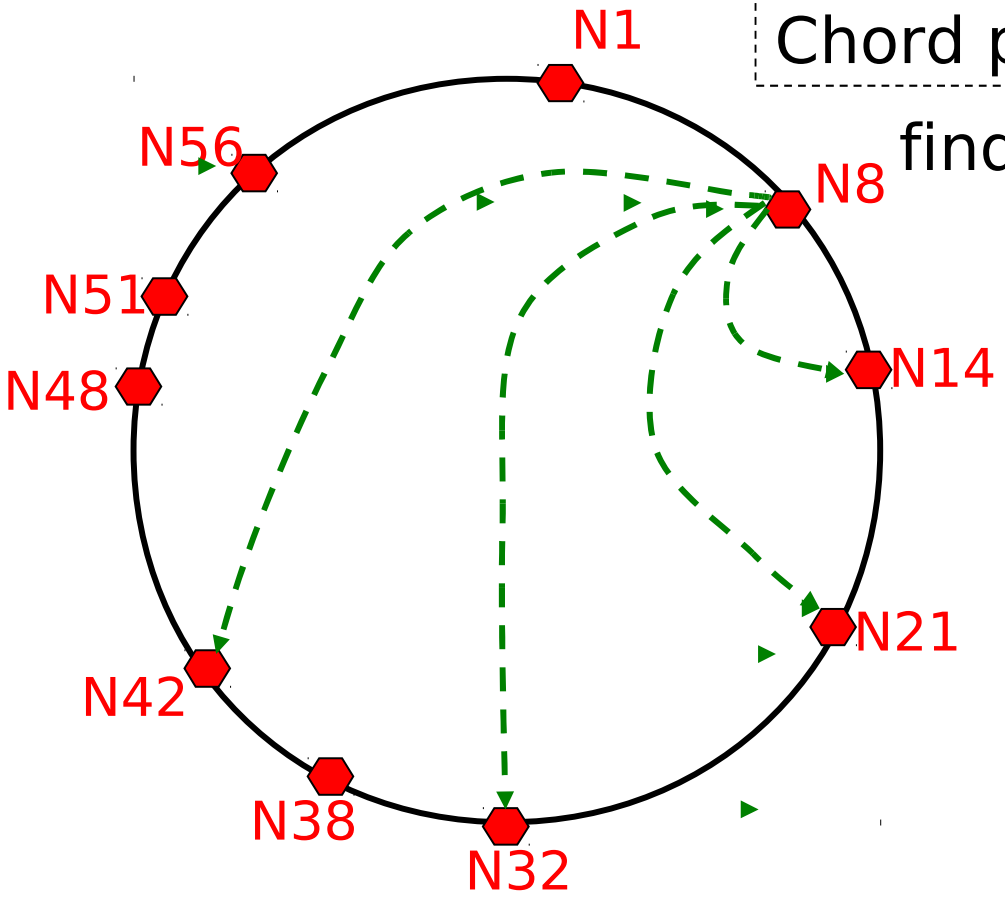


Yet Another Example



Yet Another Example

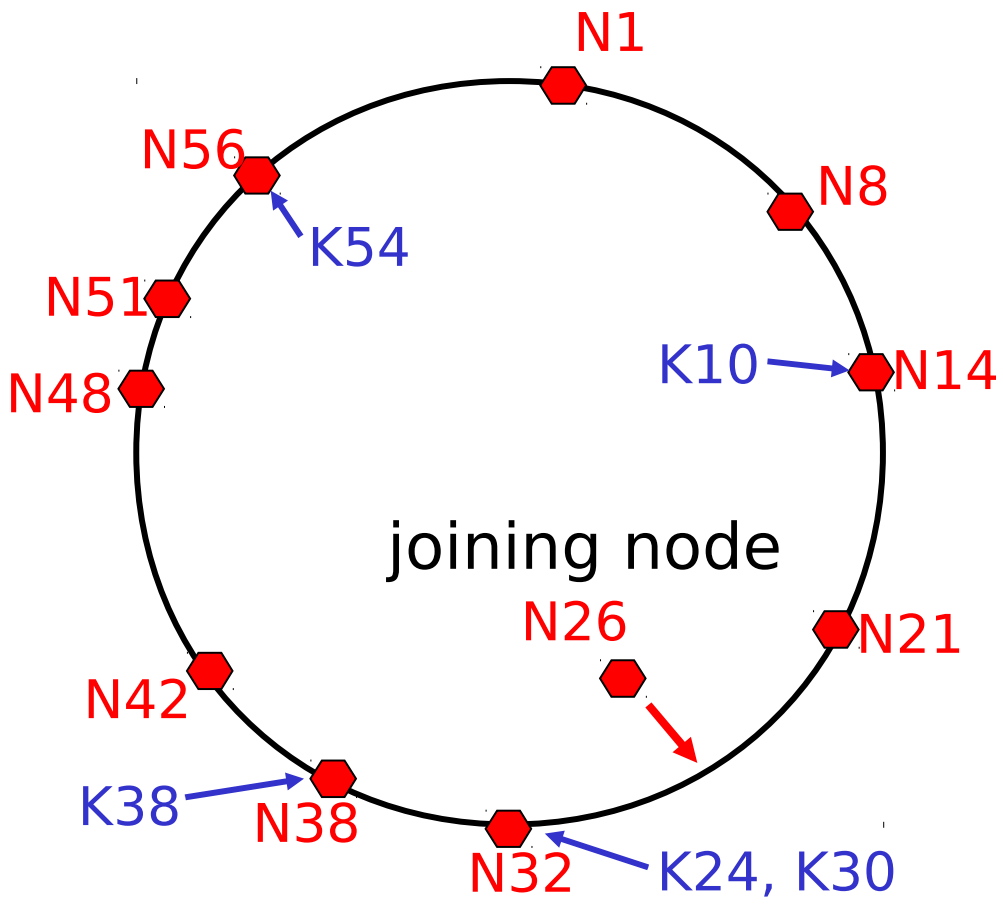
there seems to be a bug in code of Slide 26 (which is the same as Figure 5 of Chord paper).



find_succ(K8)

return N8

Adding Nodes to Circle



need to
1. update links
2. move data

for now, assume nodes never die

New Node X Joins

- Node Y is known to be in ring
- X.join(Y)

```
pred := nil;
succ := Y.find_succ(X);
```

Periodic Stabilization

- X.stabilize()

```
Y := succ.pred;
if Y in (X, succ)  succ := Y;
succ.notify(X);
```

- X.notify(Z)

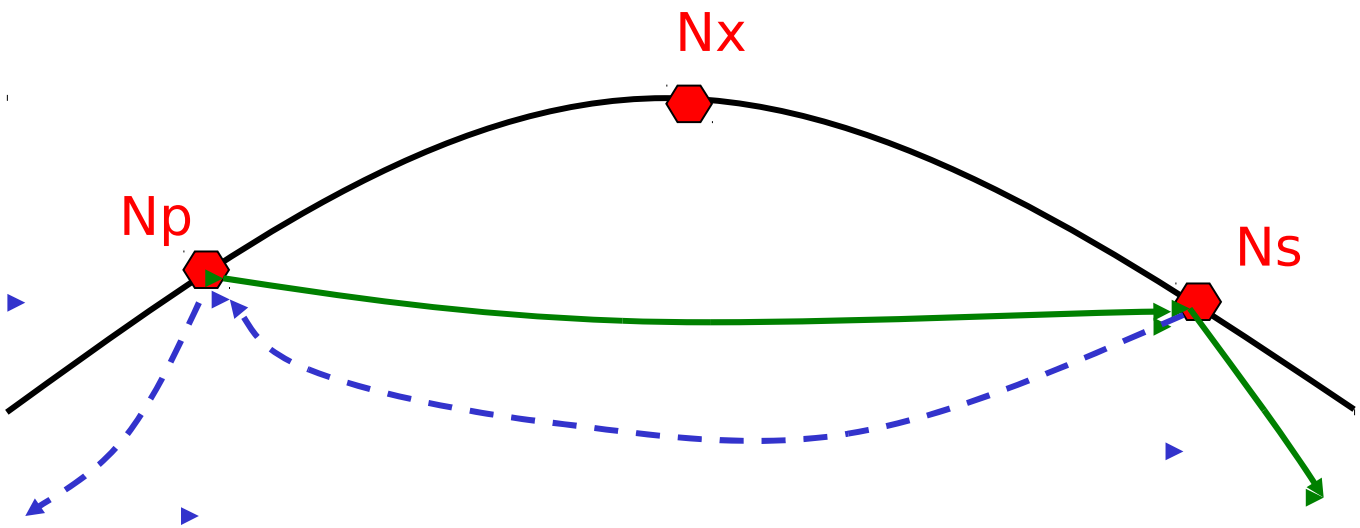
```
if pred = nil OR Z in (pred, X)
        pred := Z;
```

Periodic Finger Check

- X.fix_fingers()

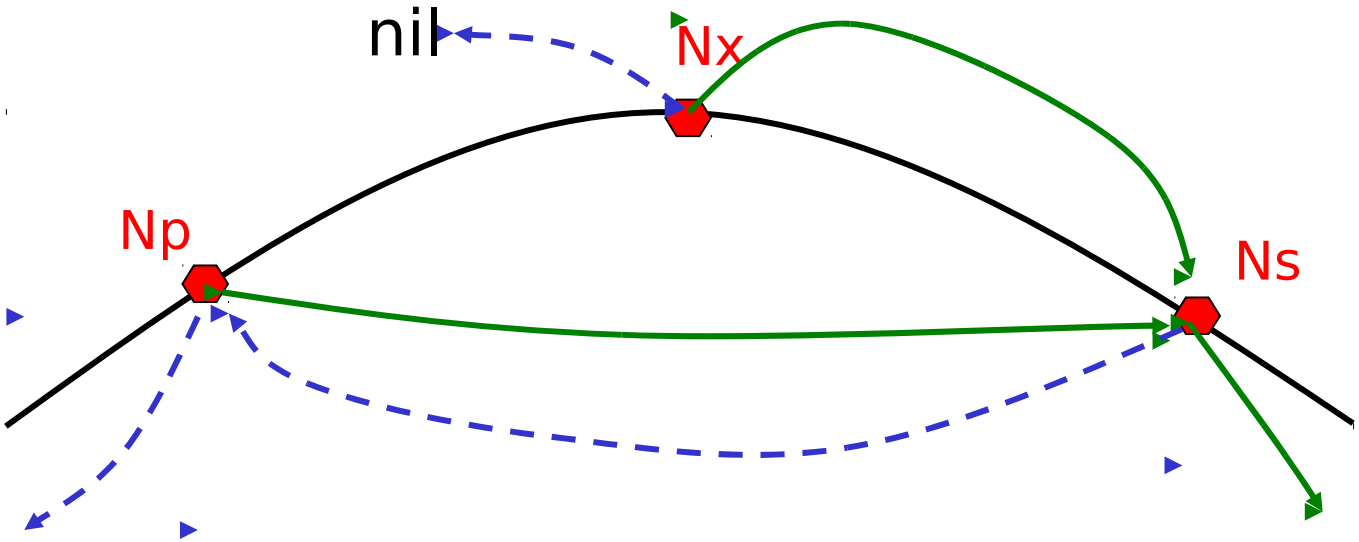
```
next := next + 1;
if next > m
        next := 1;
finger[next] := find_succ(X+2^(next-1))
```

Join Example



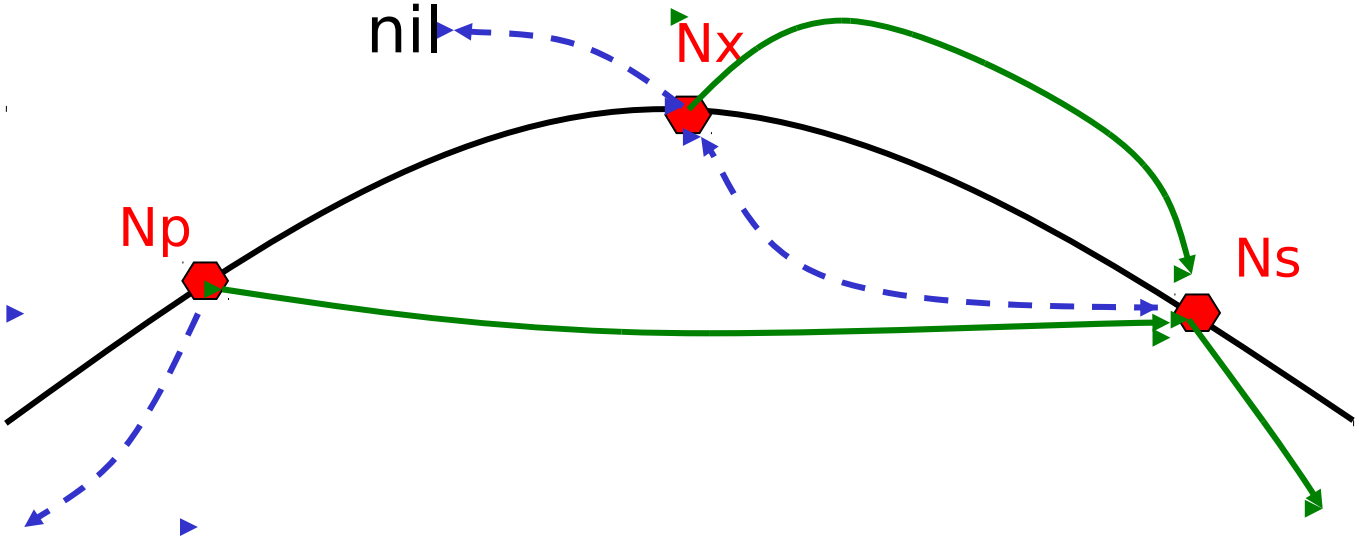
Join Example

after join:



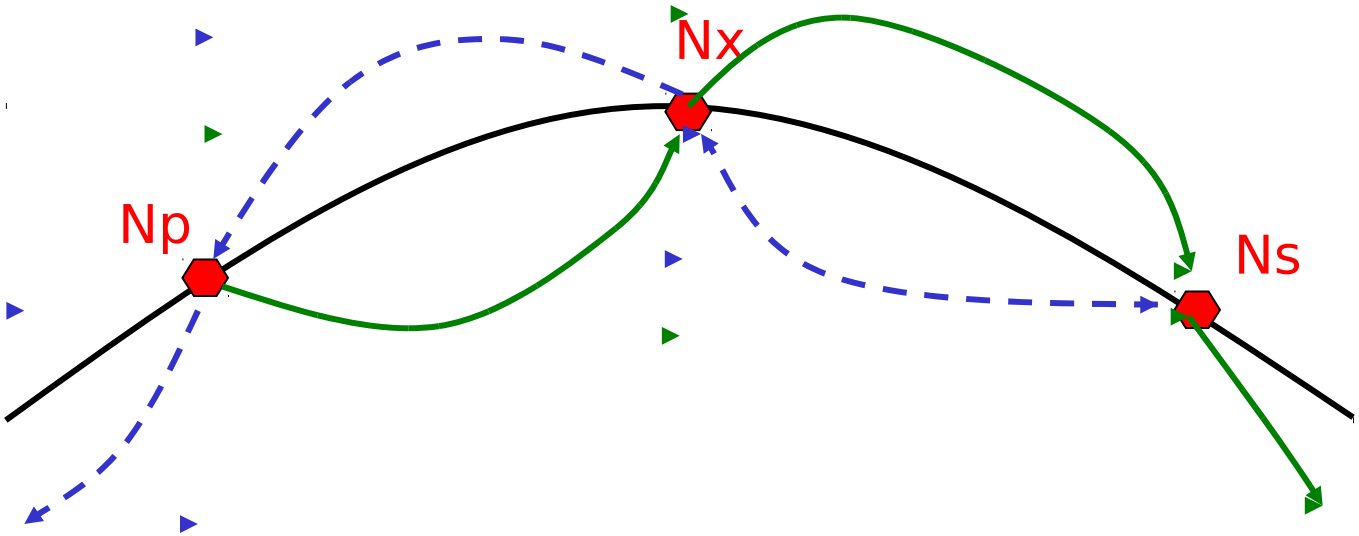
Join Example

after Nx.stabilize:
Y= Np

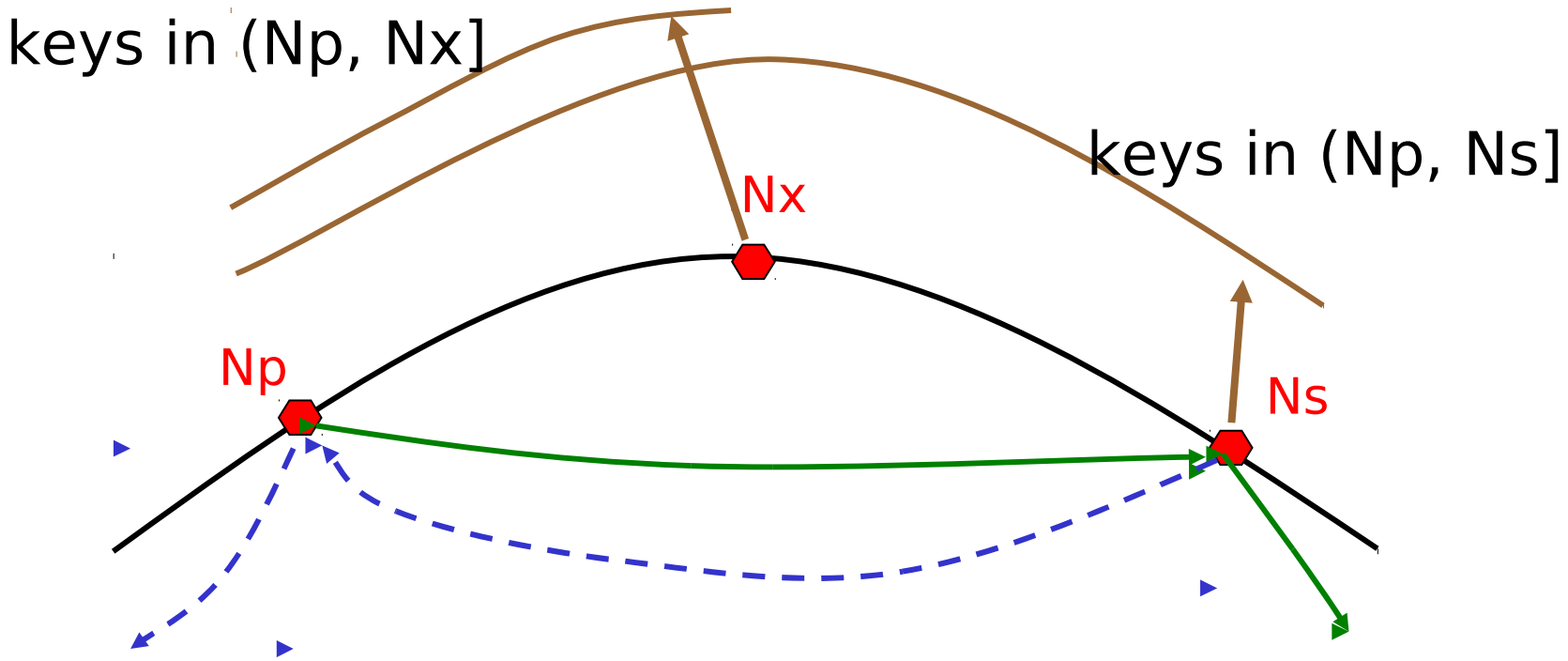


Join Example

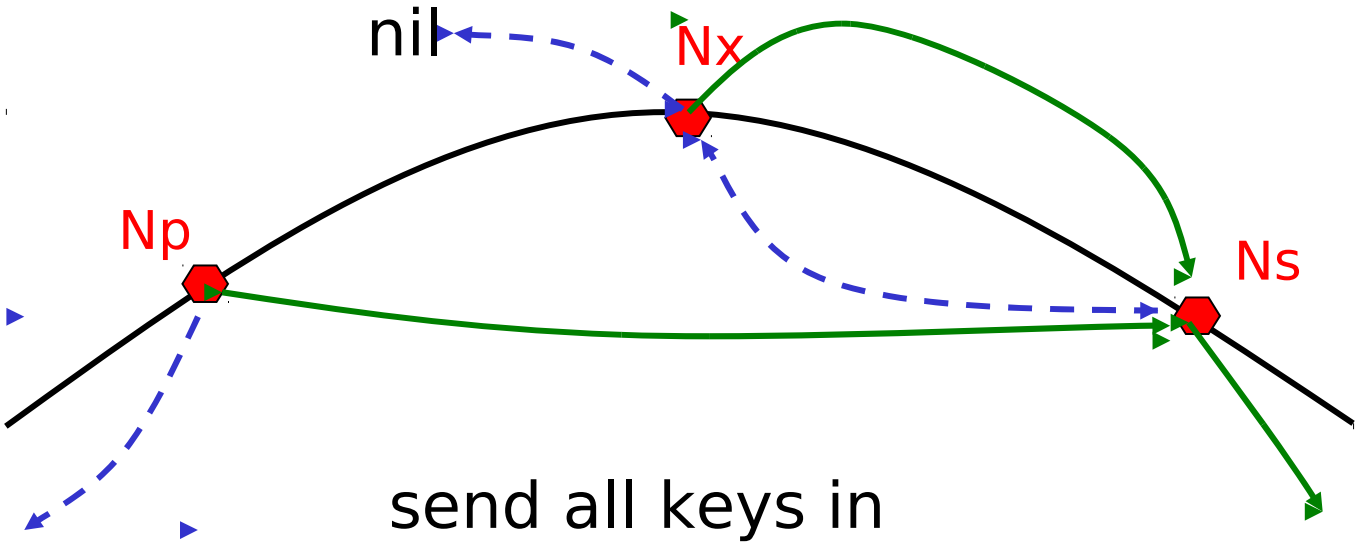
after Np.stabilize:
Y= Nx



Moving Data: When?



Move Data After Ns.notify(Nx)



send all keys in
range (Np, Nx]
when Ns.prev is updated...

Periodic Stabilization

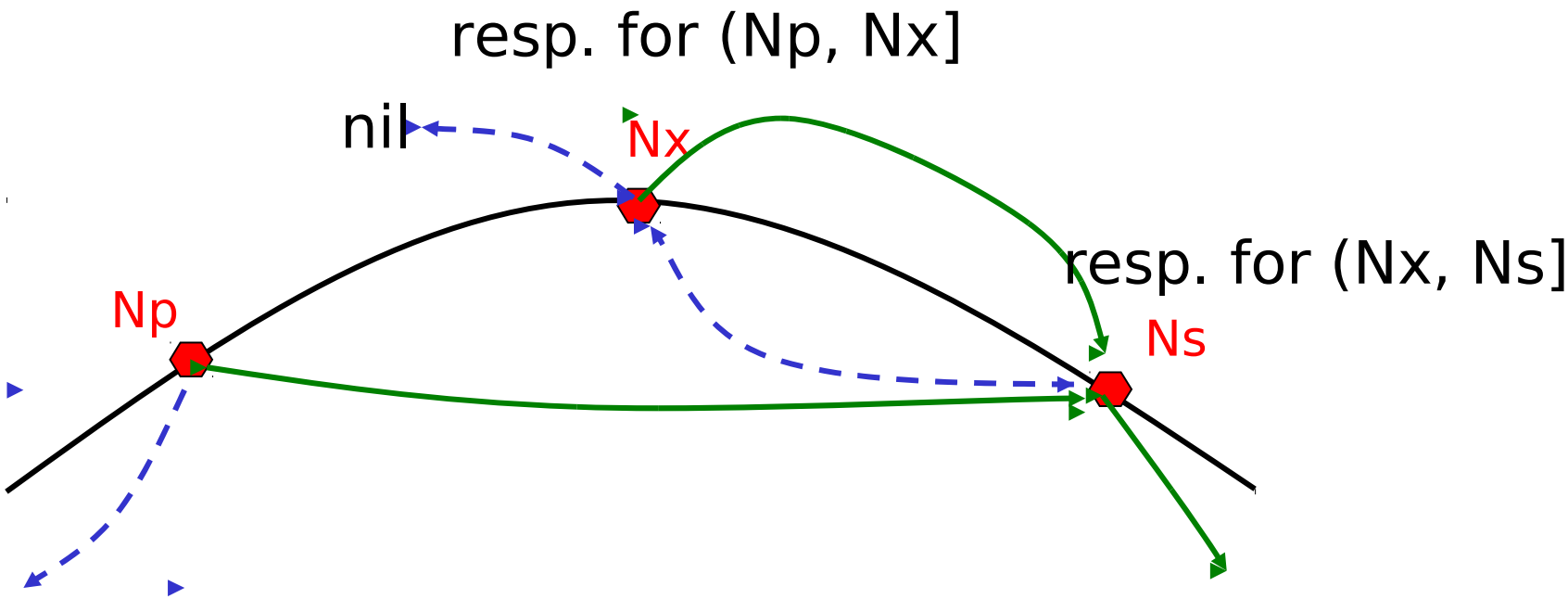
- X.notify(Z)

```
if pred = nil OR Z in (pred, X)
        Z.give(data in range (pred ,Z] )
        pred := Z
        X.remove(data in range
(pred ,Z] )
```

Question: What data do we give when pred=nil?

Note: We are glossing over concurrency issues,
e.g., what happens to lookups while we are moving data?

Lookup May be at Wrong Node!



lookup for k in (Np, Nx]
directed to Ns!

Looking Up Data (Revised)

- X.DHTlookup(k)

```
ok := false
while not ok do
Y := find_succ(k)
[ok, S] := Y.lookup(k)
return S
```

- Y.lookup(k)

```
if k in (pred, Y]
return [true, local values for k]
else return [false, {}]
```

Inserting Data (Revised)

- X.DHTinsert(k, v)

```
ok := false
while not ok do
Y := find_succ(k)
ok := Y.insert(k, v)
```

- Y.insert(k, v)

```
if k in (pred, Y]
insert [k,v] in local storage
return true
else return false
```

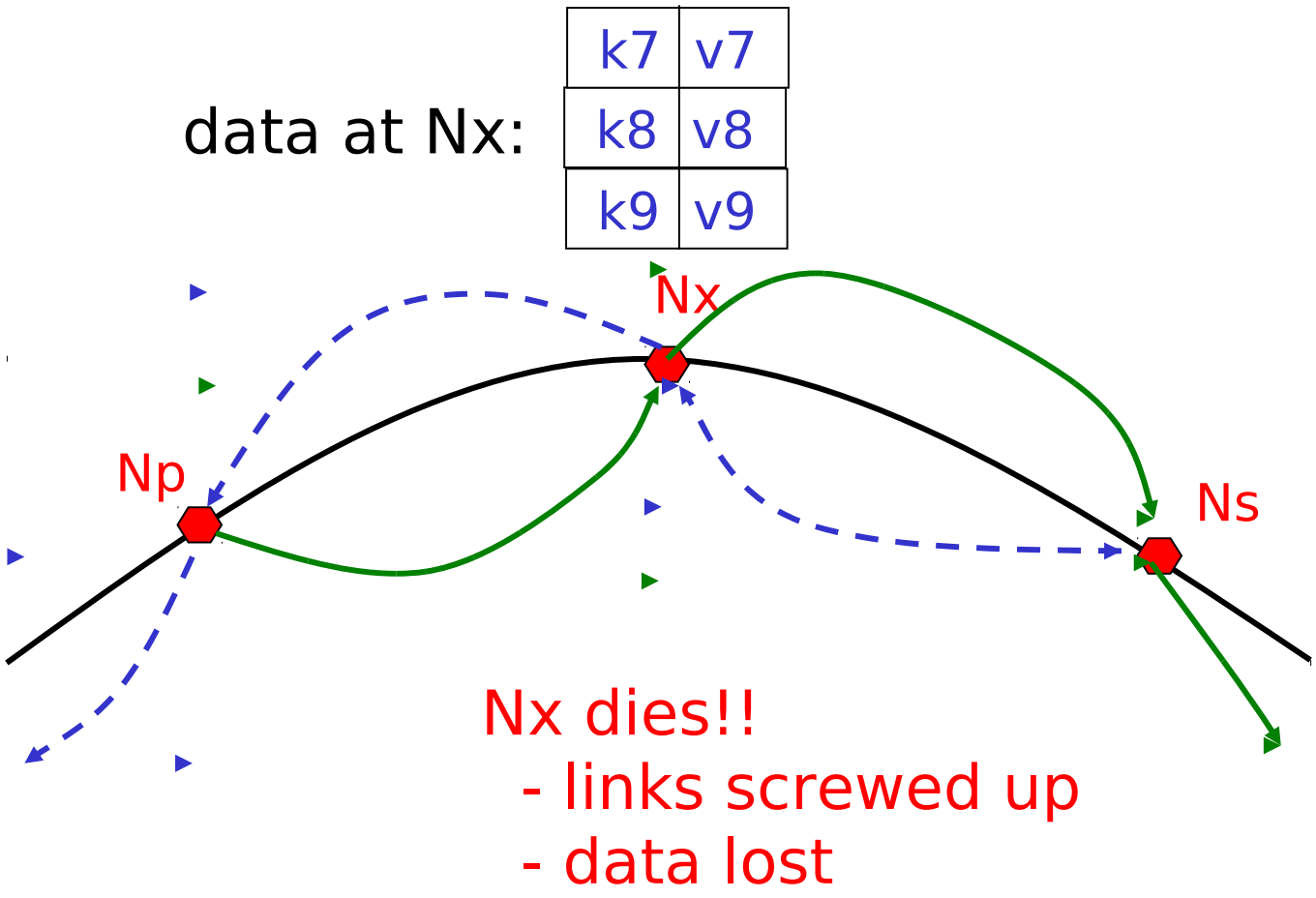
Why Does This Work?

- pred, succ links eventually correct
- data ends up at correct node
  key k is at node X where k in (X.prev, X]
- finger pointers speed up searches but do not cause problems

Results for N Node System

- With high probability, the number of nodes that must be contacted to find a successor is O(log N)
- Although finger table contains room for m entries, only O(log N) need to be stored
- Experimental results show average lookup time is (log N)/2

Node Failures

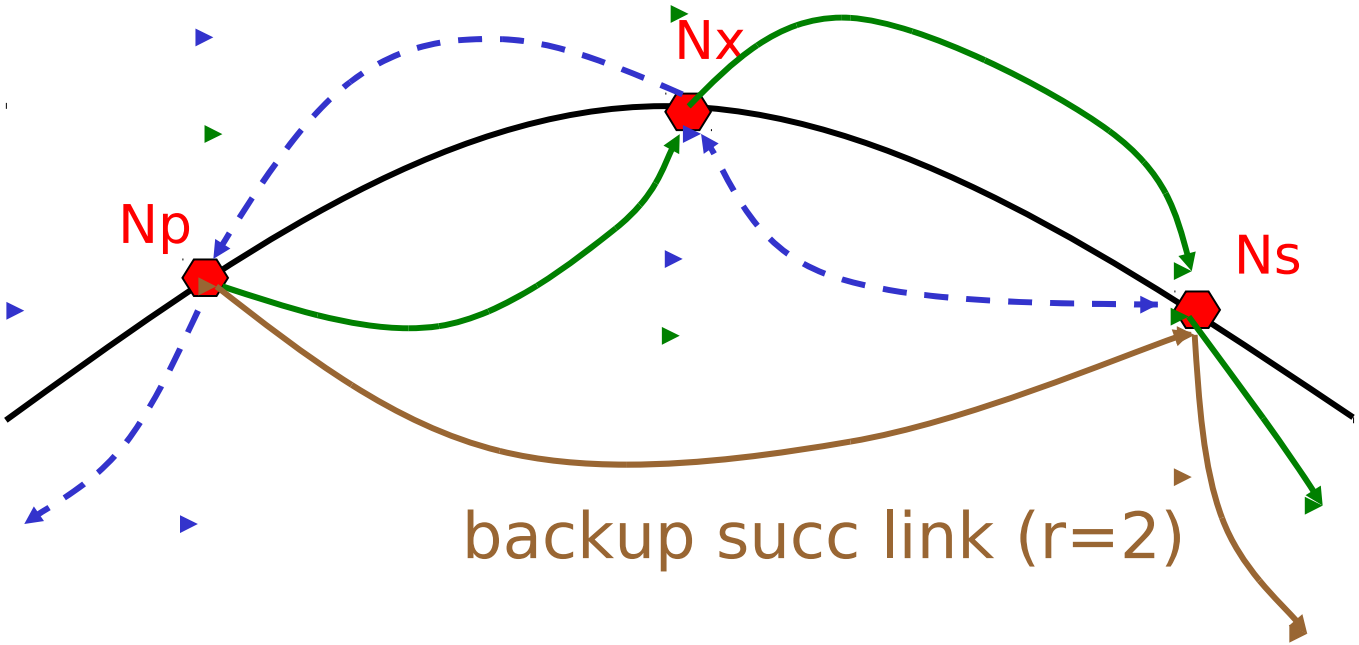


To Fix Links

- X.check_pred()
  ```
  if (pred has failed)
      pred := nil;
  ```
- Also, keep links to r successors in ring

Failure Example

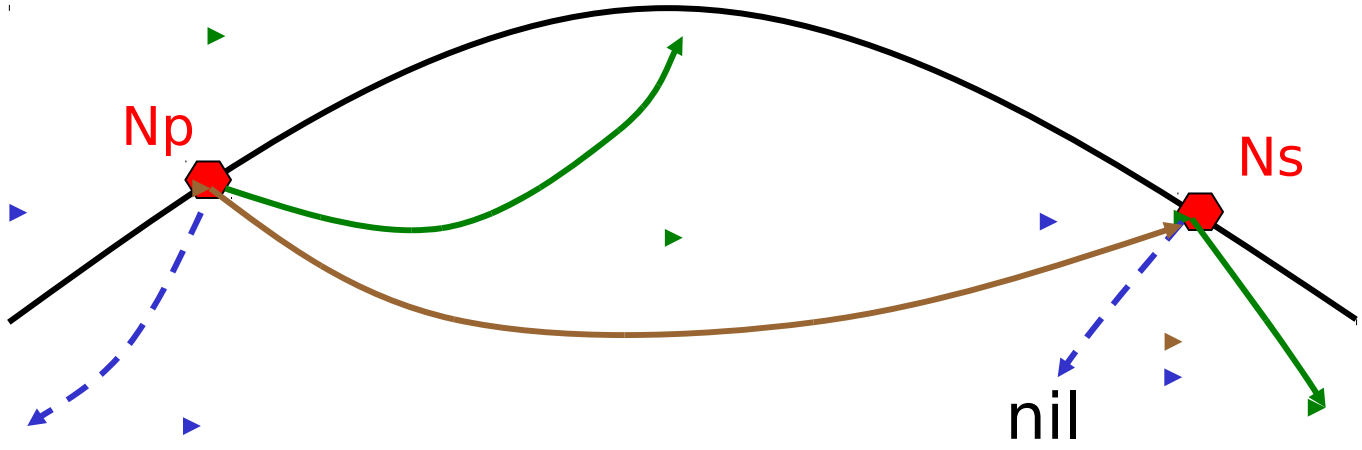
Initially...



Nx dies...

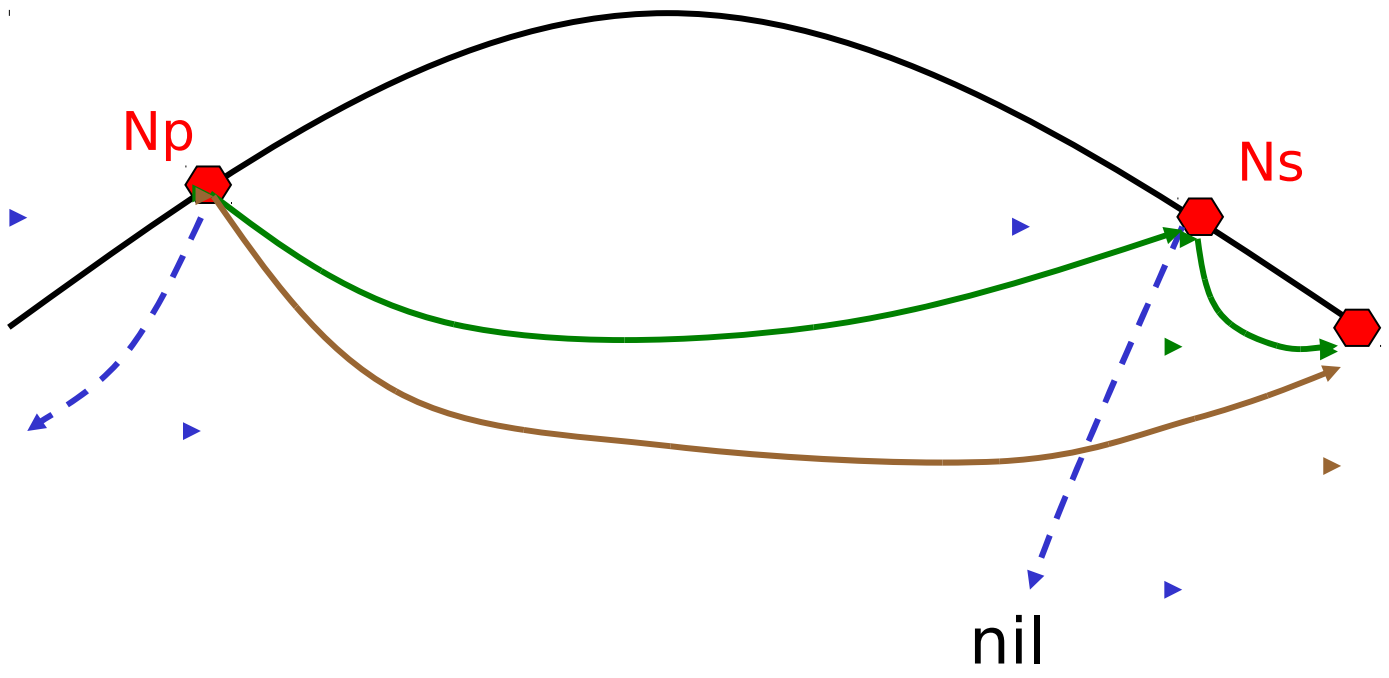
Failure Example

After Ns.check_pred



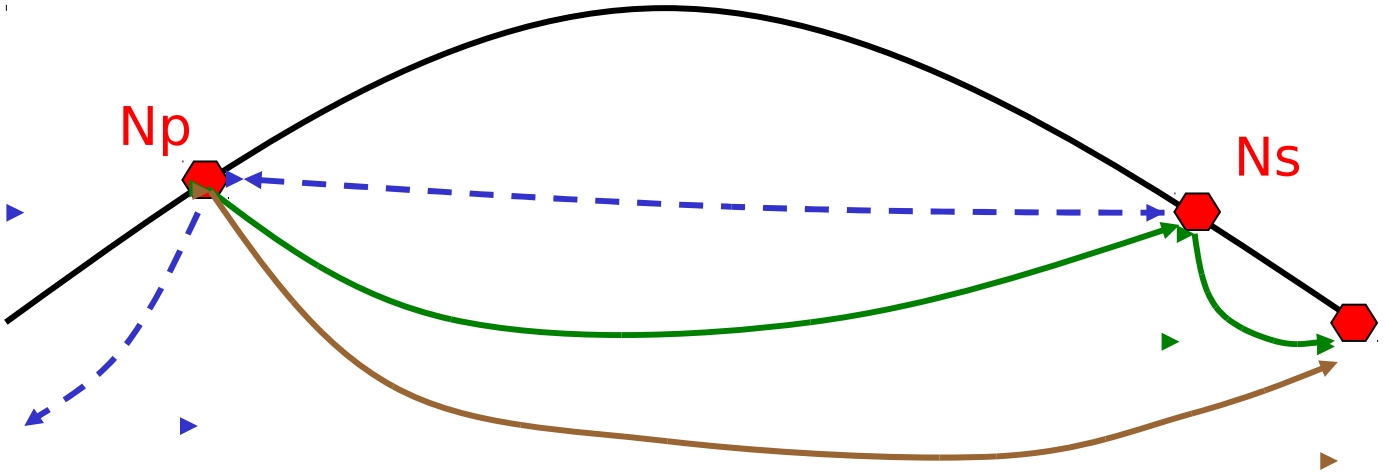
Failure Example

After Np discovers Nx down...



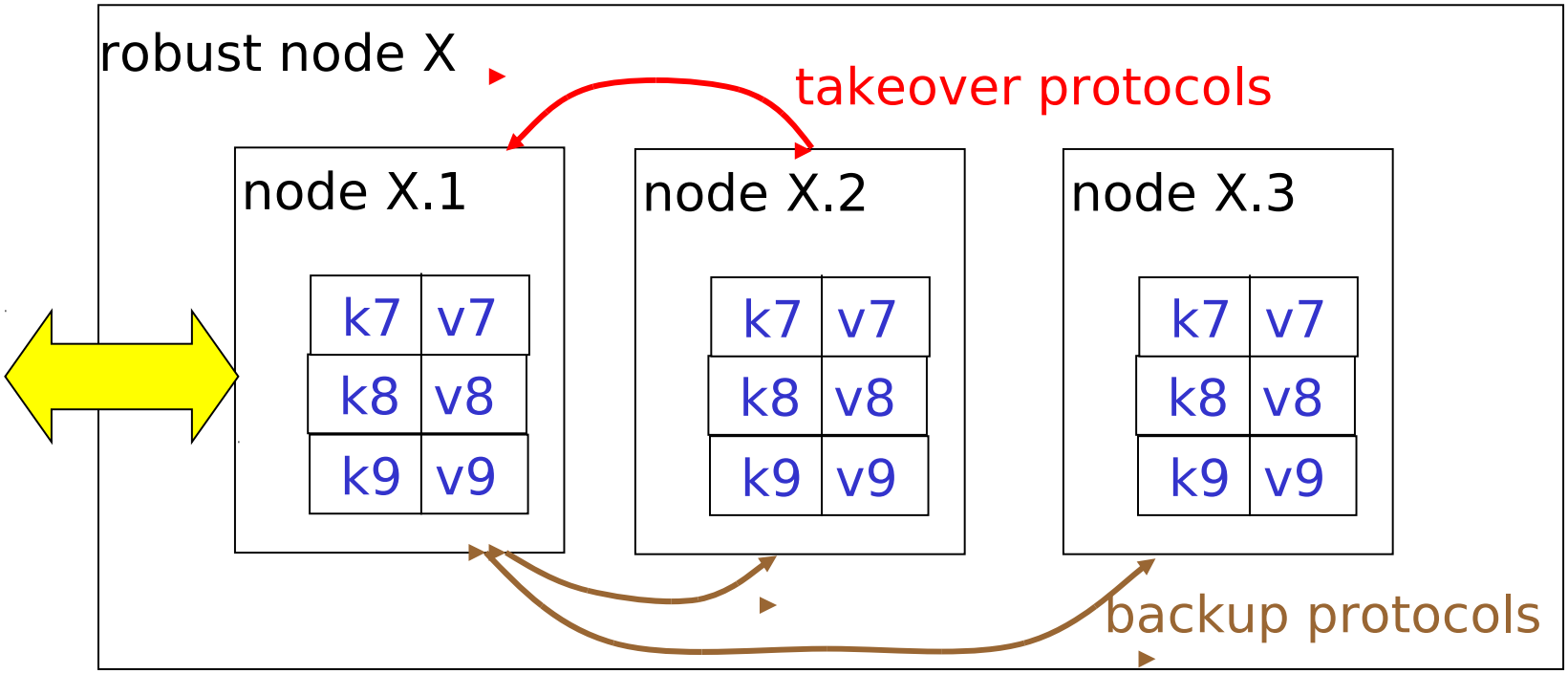
Failure Example

After stabilization...

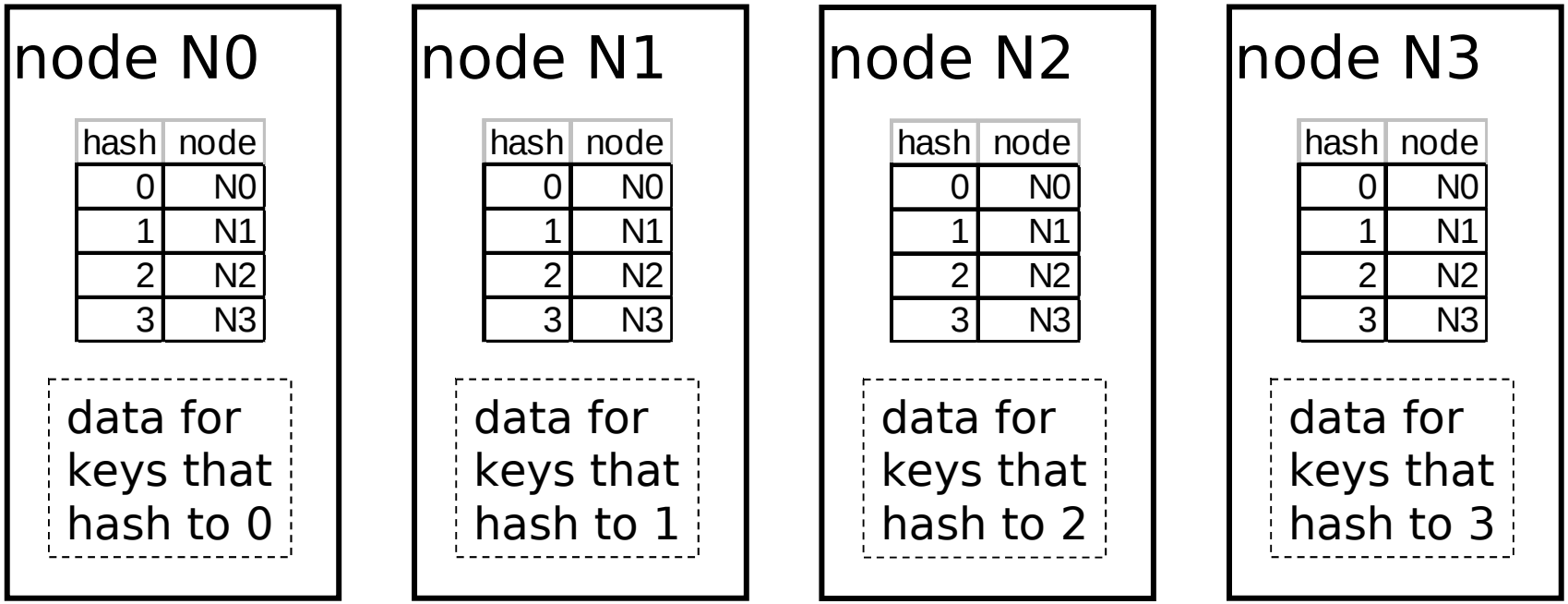


Protecting Against Data Loss

- One idea: robust node (see notes on replicated data)



Replicated Hash Table



Looking Up Data

- X.DHTlookup(k)

```
id := H(k);
if X = X.node(id)
        return local values for k
else
        Y := X.node(id);
        return Y.DHTlookup(k)
```

Concurrency Control

- X.DHTlookup(k)

```
id := H(k);
lock(table);
if X = X.node(id)
        temp := local values for k;
        unlock(table);  return temp
else
        unlock(table);
        Y := X.node(id);
        return Y.DHTlookup(k)
```

control for bucket id could move to another node

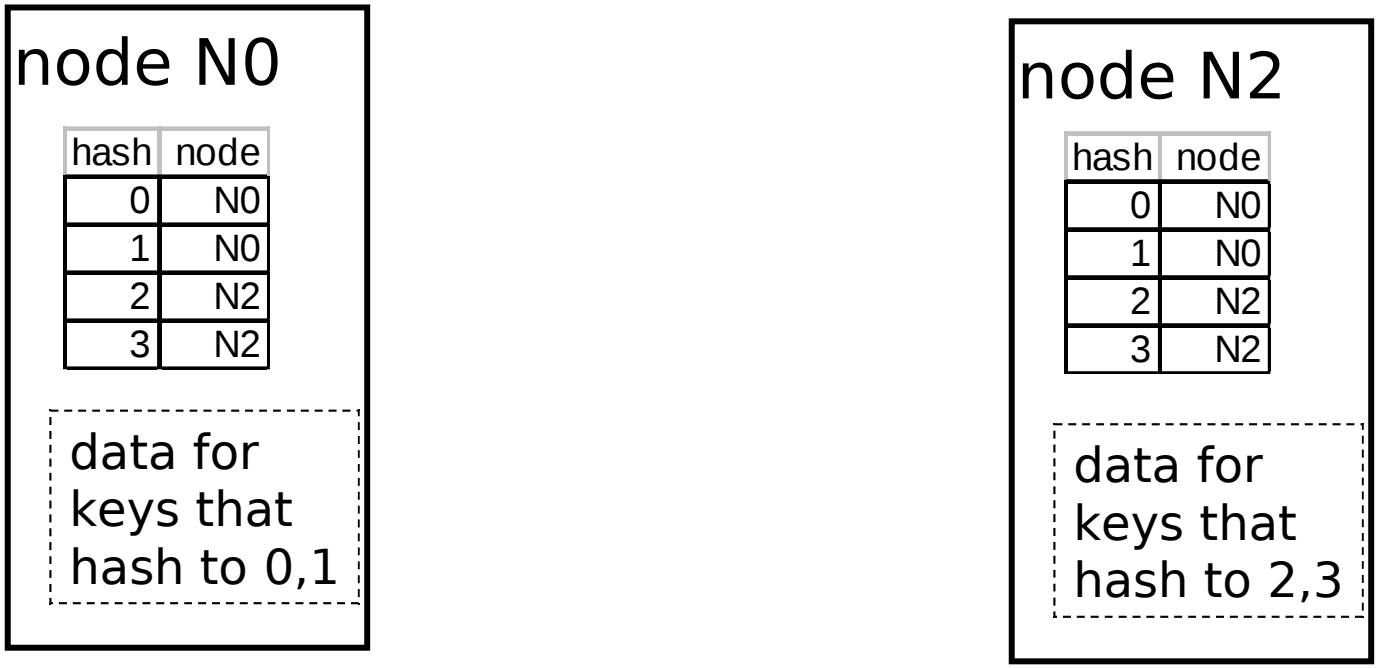
assume single statements are atomic

Shorthand

- X.DHTlookup(k)

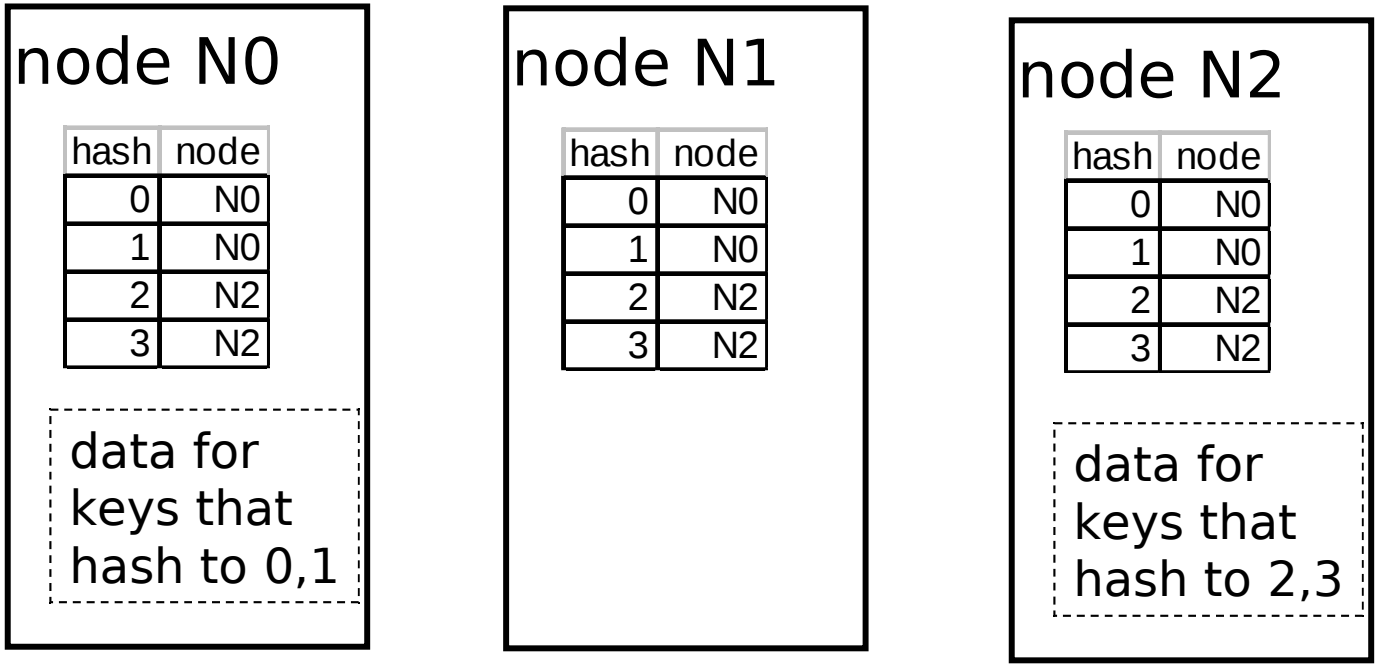
```
id := H(k);
[[ if X = X.node(id)
        return local values for k
else  ]]
        Y := X.node(id);
        return Y.DHTlookup(k)
```

Node Joins

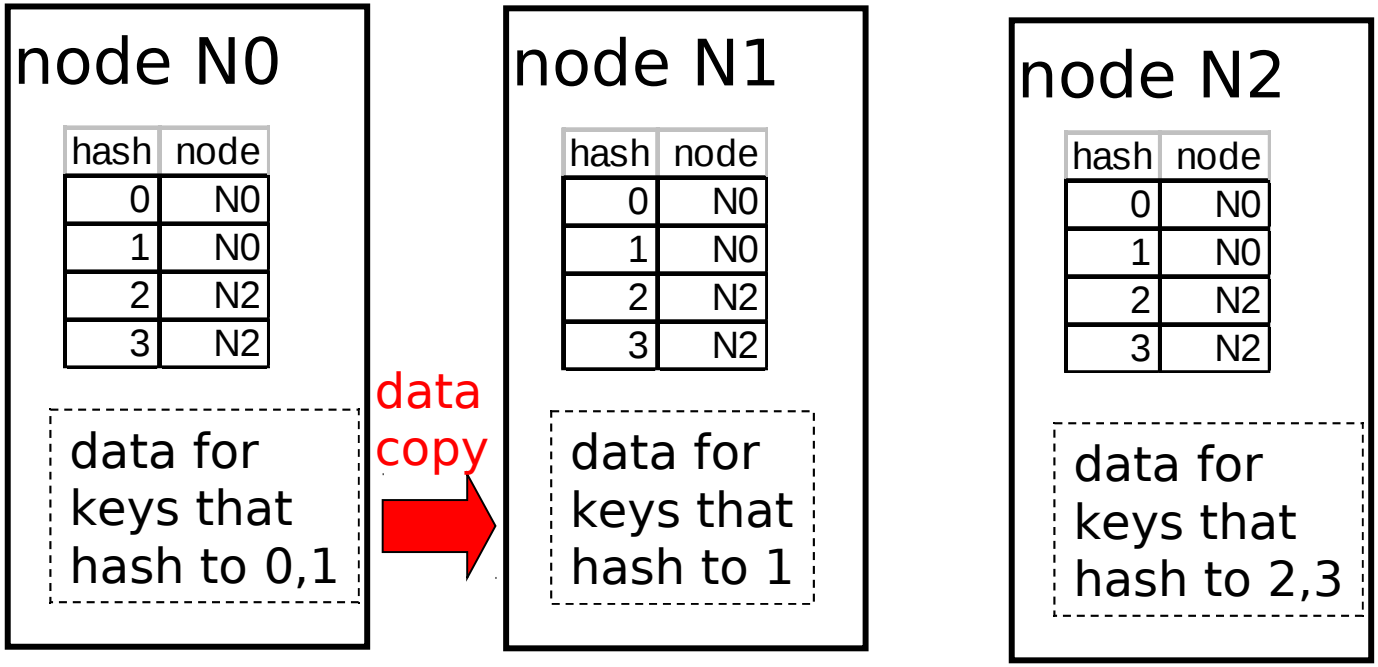


N0 overloaded, asks N1 for help...

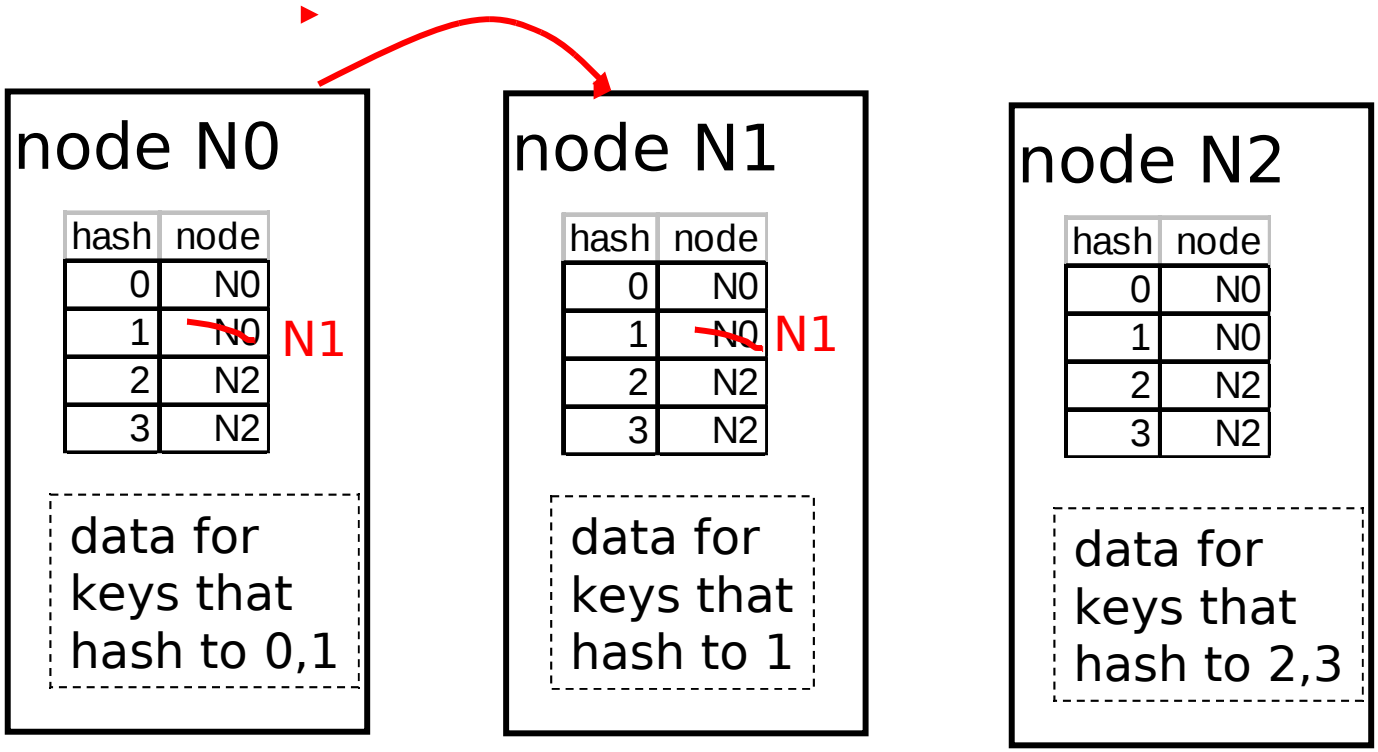
Node Joins First, set up N1...



Node Joins Copy data to N1...

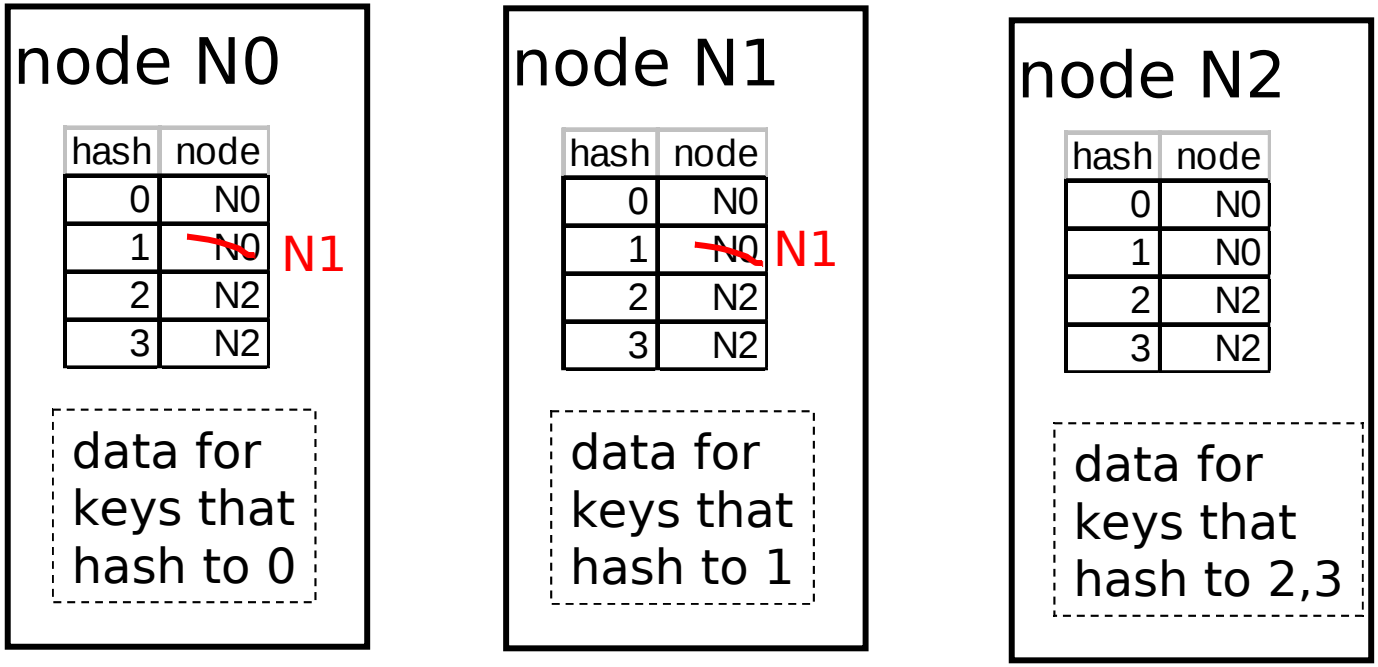


Node Joins Change control...

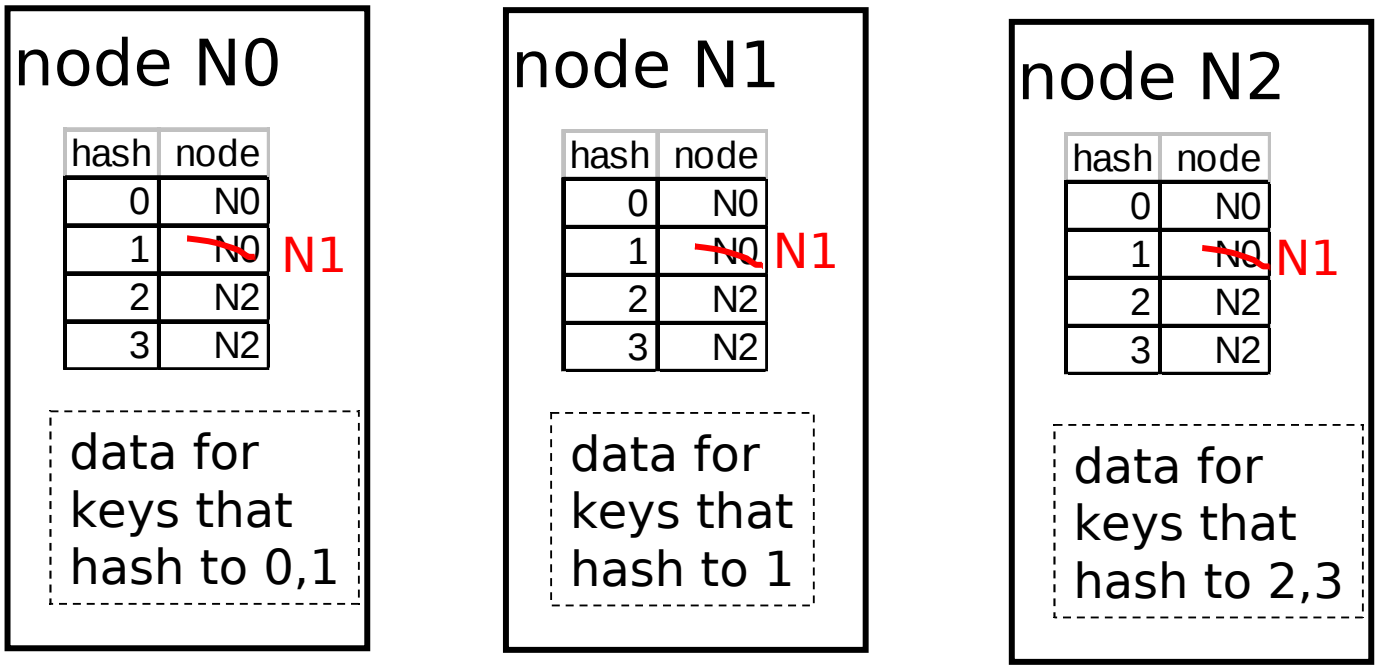


Which do we update first, N0 or N1??

Node Joins Drop data at N0...



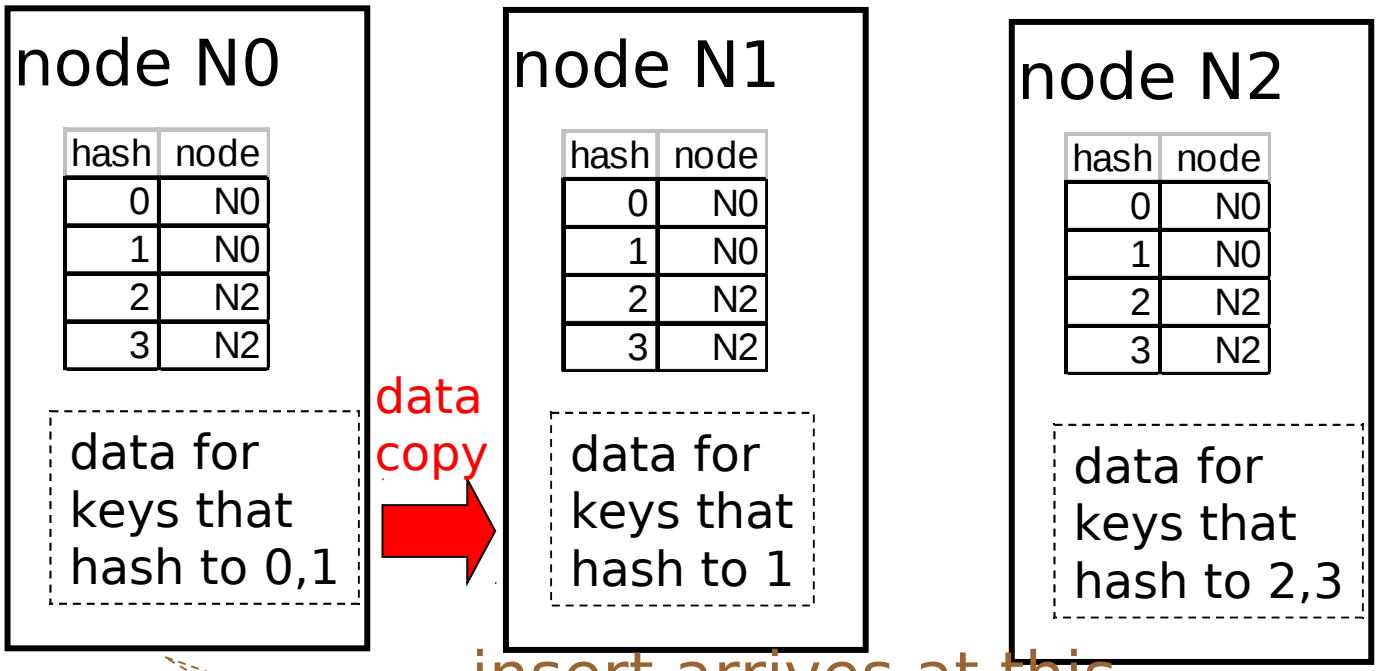
Update Other Nodes



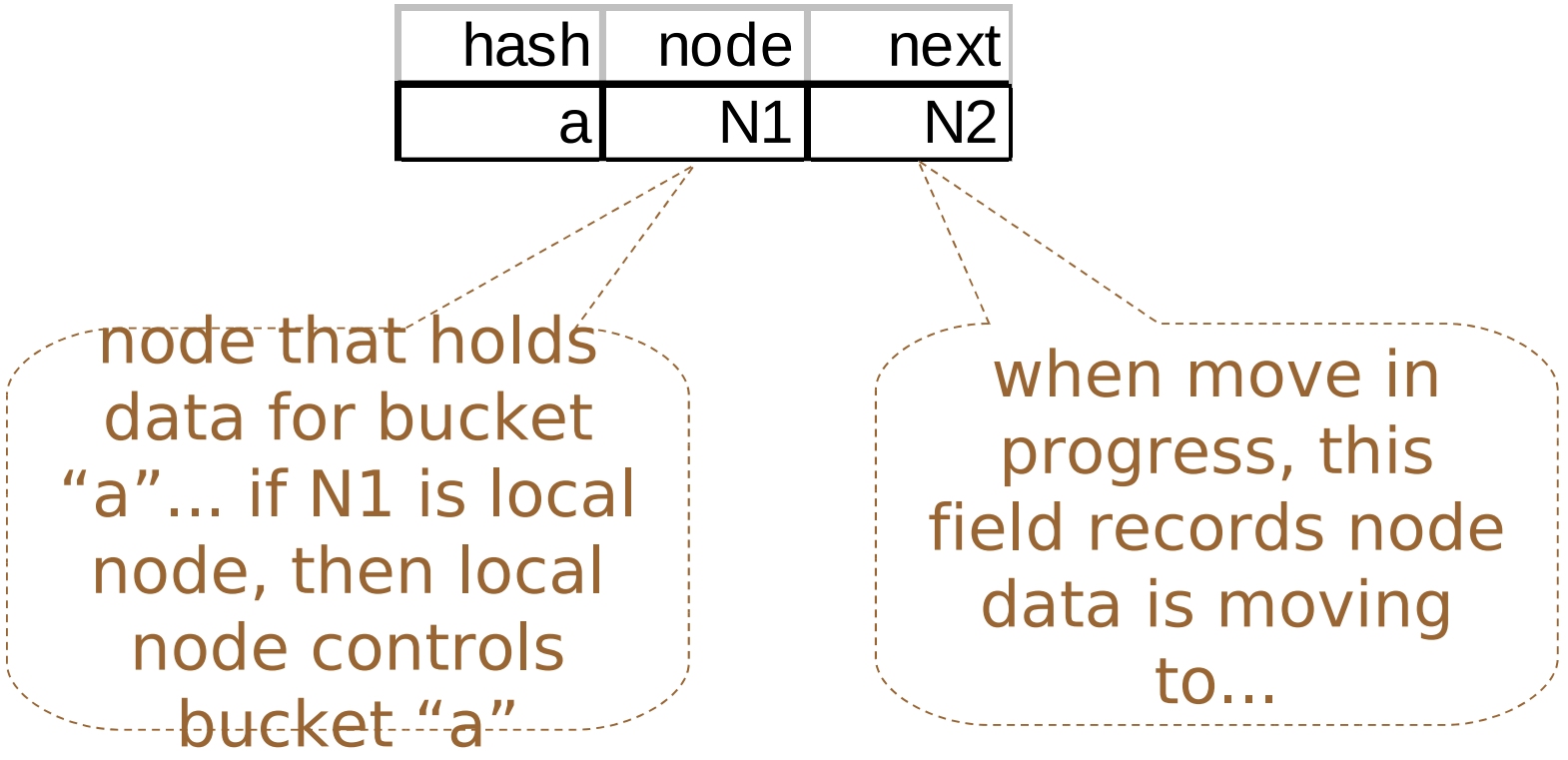
How do other nodes get updated?

- Eagerly by N0?
- Lazily by future lookups?

What about inserts?



Control of Hash Table



Different Scenarios

X:

hash	node	next
a	X	nil (X)

Node X controls a,
lookups local, inserts local

X:

hash	node	next
a	Y	nil (Y)

Forward lookups and inserts to Y

X:

hash	node	next
a	X	Y

Node X controls a,
lookups local, inserts to Y (& X?

Y:

hash	node	next
a	X	Y

Inserts local, forward lookups to X

New Node Y Joins

- X.join(Y)
  [[select HT entry [j, X] willing to give up
  (note, next[j] should be nil)
  next[j] := Y;]]
  copy HT making Y.next[j] := Y;
  copy data for key j to Y
  Y.node[j] := Y; Y.next[j] := nil;
  node[j] := Y; next[j] = nil;
  remove data for key j

Looking Up Data (Revised)

- X.DHTlookup(k)

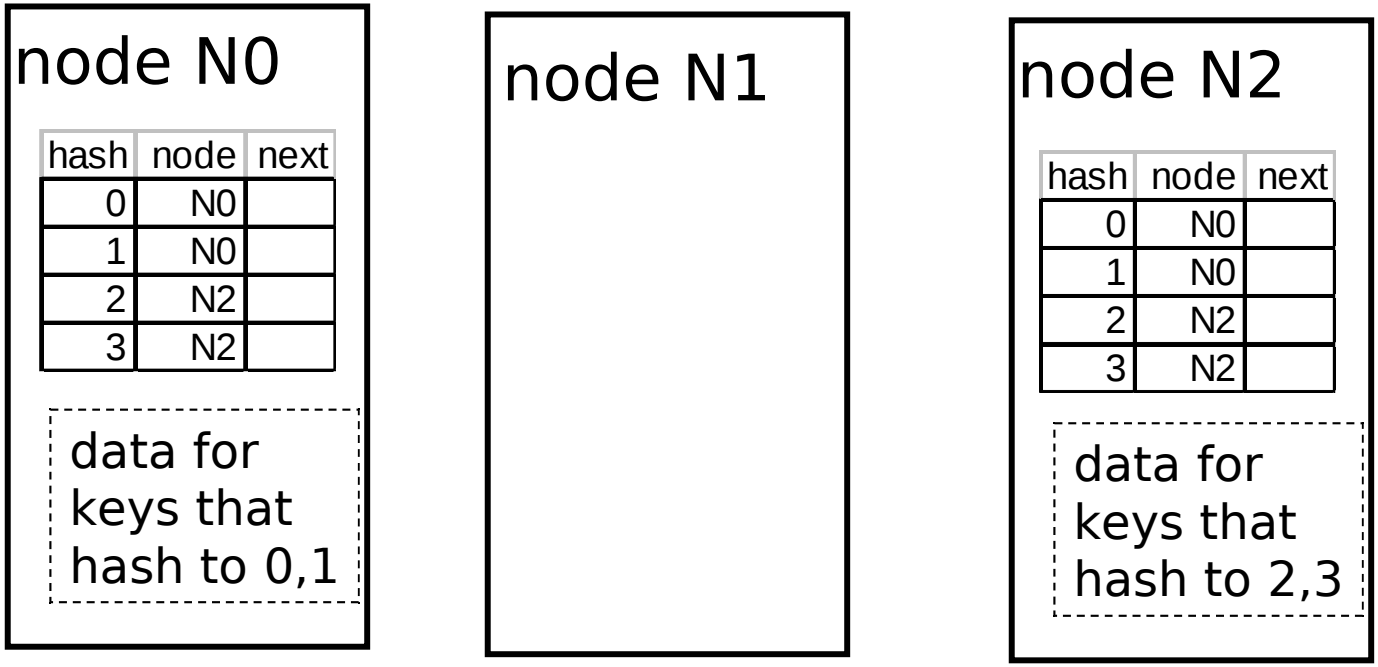
```
id := H(k)
[[ if X = X.node(id) then
return local values for k
else ]]
Y := X.node(id); tt := Y.node(id);
[[ if (tt neq Y) and (X.node(id)=Y)
X.node(id) := tt ]]
return( Y.DHTlookup(k) )
```

Storing Data

- X.DHTinsert(k, v)

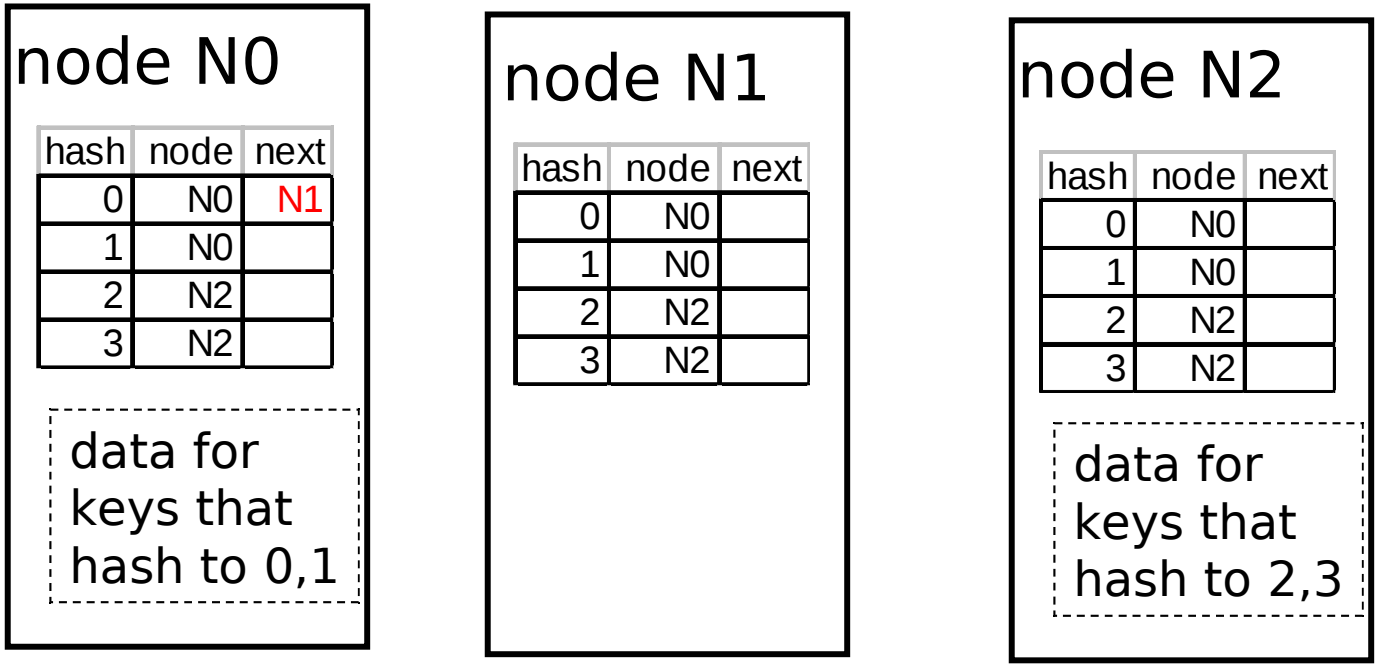
```
id := H(k)
[[ tt := node(id); nn := next(id);
if (X = tt and nn=nil) or (nn=X)
        insert (k,v) locally
else  ]]
        if nn = nil  Y := tt else Y := nn;
        Y.DHTinsert(k,v)
```

Example



N0.join(N1) is executed...

Example: Step 1

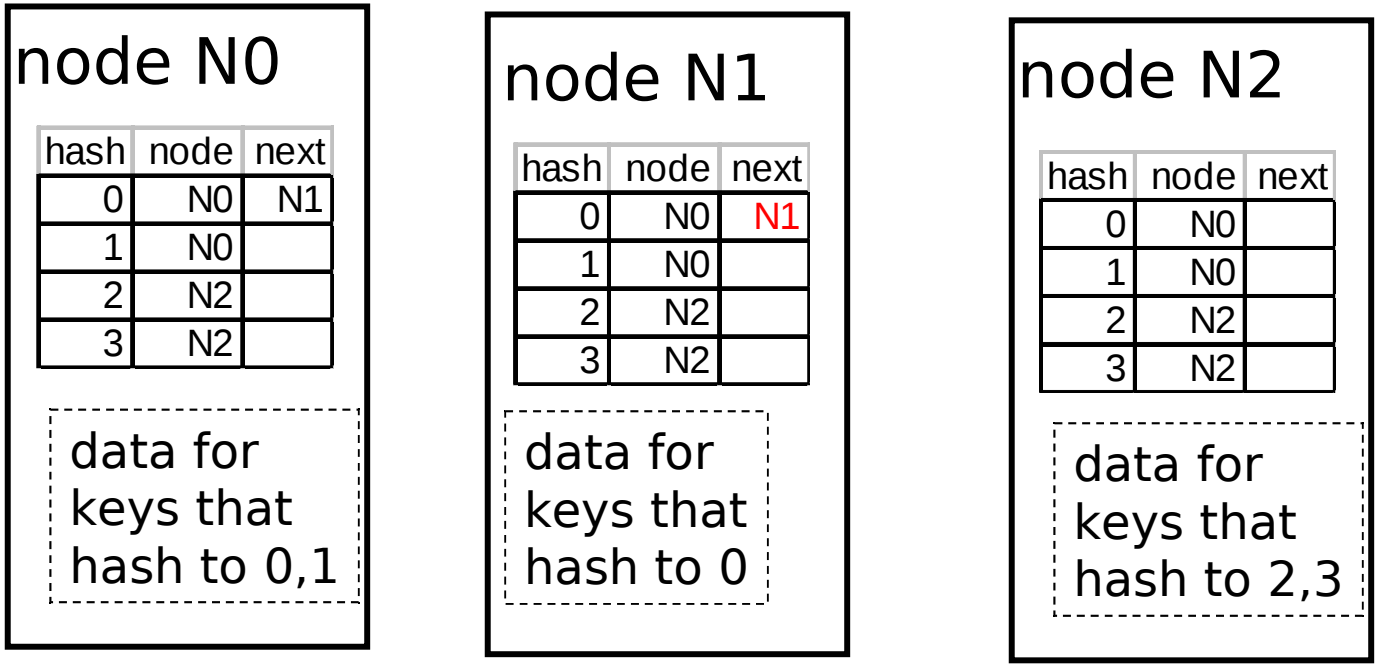


ucket 0 is reserved for N1...

lote: What happens to lookups & inserts at this point in time

(Inserts can bounce back and forth from N0 to N1... Problem?)

Example: Step 2

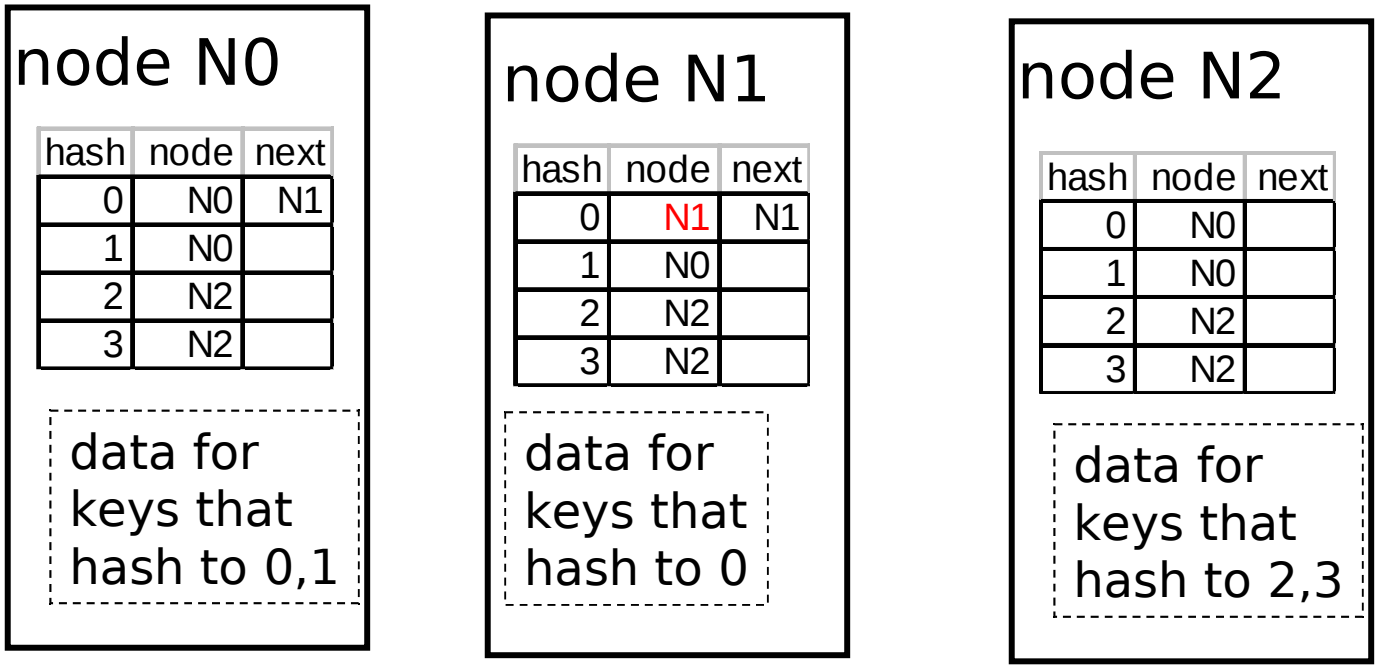


N1 starts accepting inserts and data from N0...

Note: What happens to lookups & inserts at this point in time

(Lookups will miss recently inserted data...)

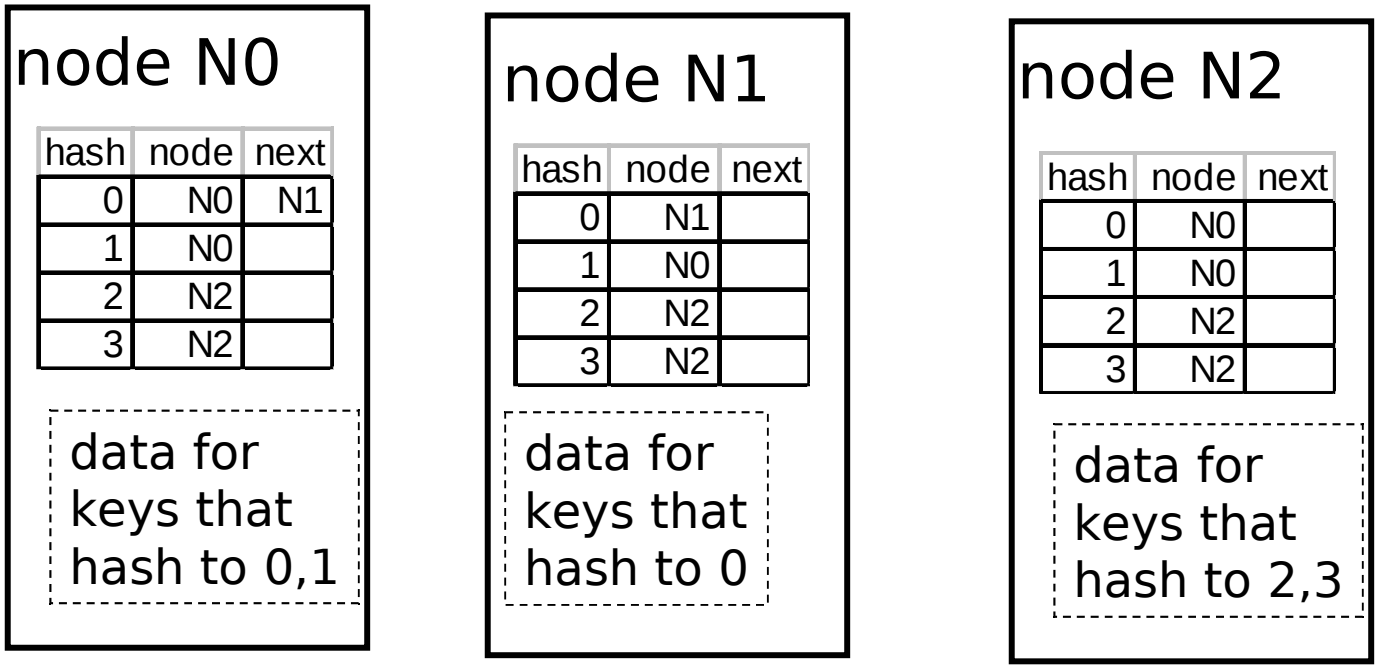
Example: Step 3



Copy complete; Node N1 activated (part 1)

Note: What happens to lookups & inserts at this point in time

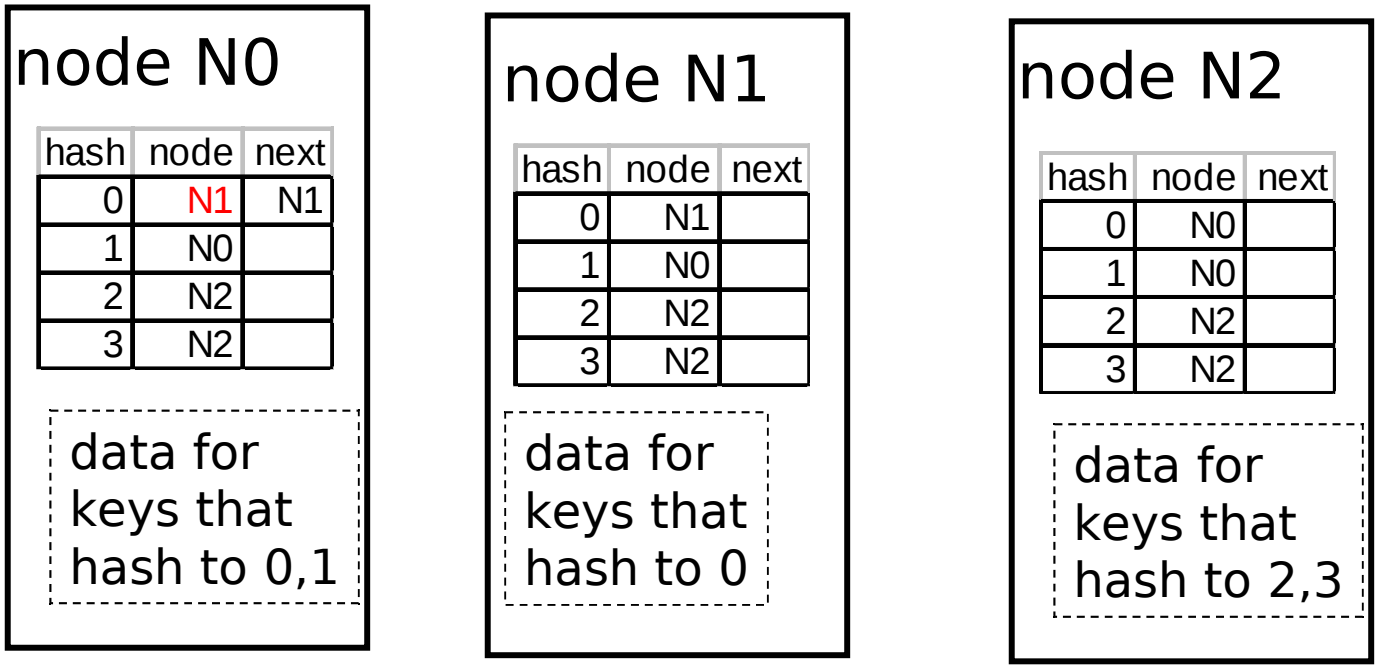
Example: Step 4



Copy complete; Node N1 activated (part 2)

Note: What happens to lookups & inserts at this point in time

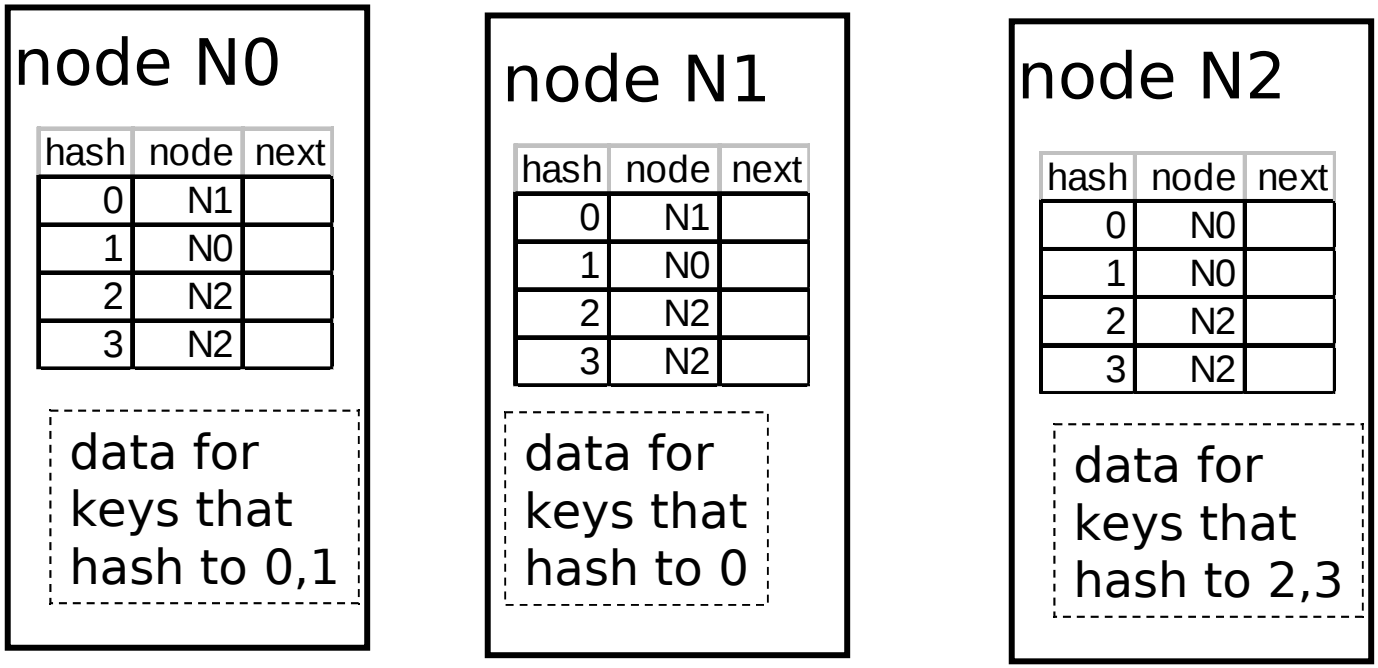
Example: Step 5



Node N0 records new master (part 1)

Note: What happens to lookups & inserts at this point in time

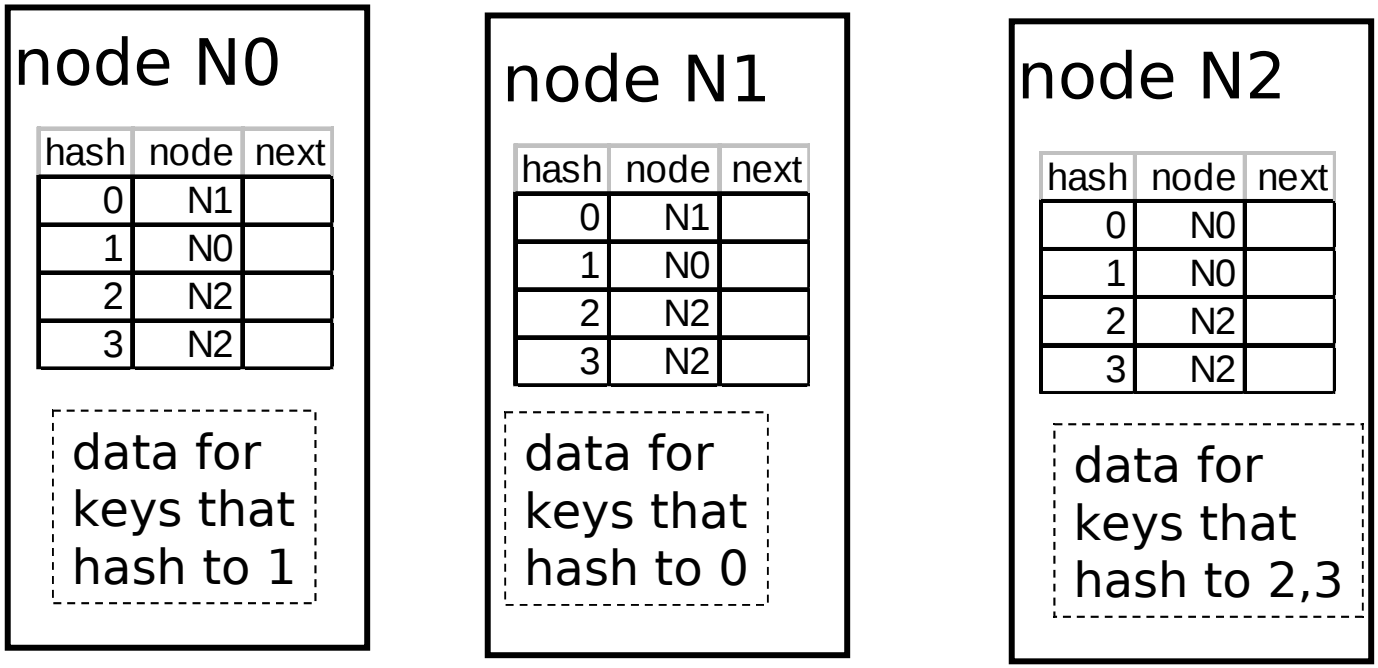
Example: Step 6



Node N0 records new master (part 2)

Note: What happens to lookups & inserts at this point in time

Example: Step 7

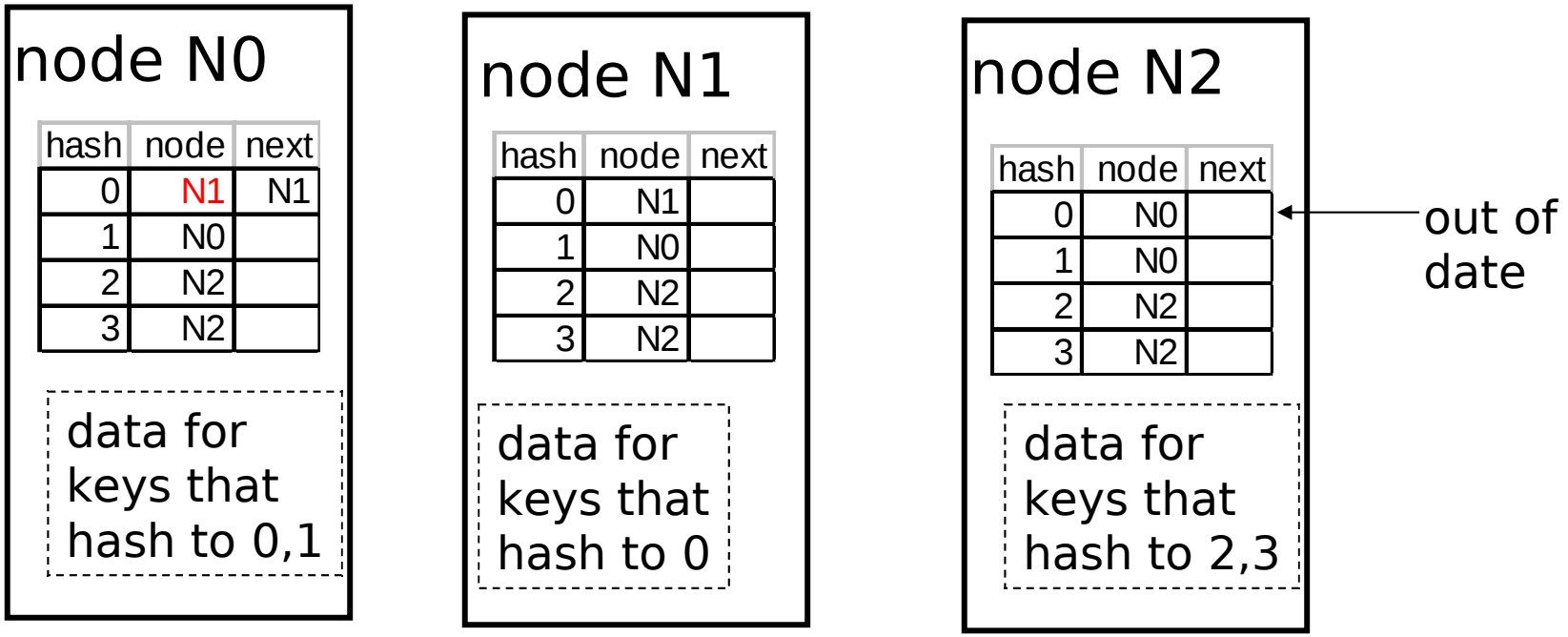


Node N0 removes data for bucket 0...

Note: What happens to lookups & inserts at this point in time

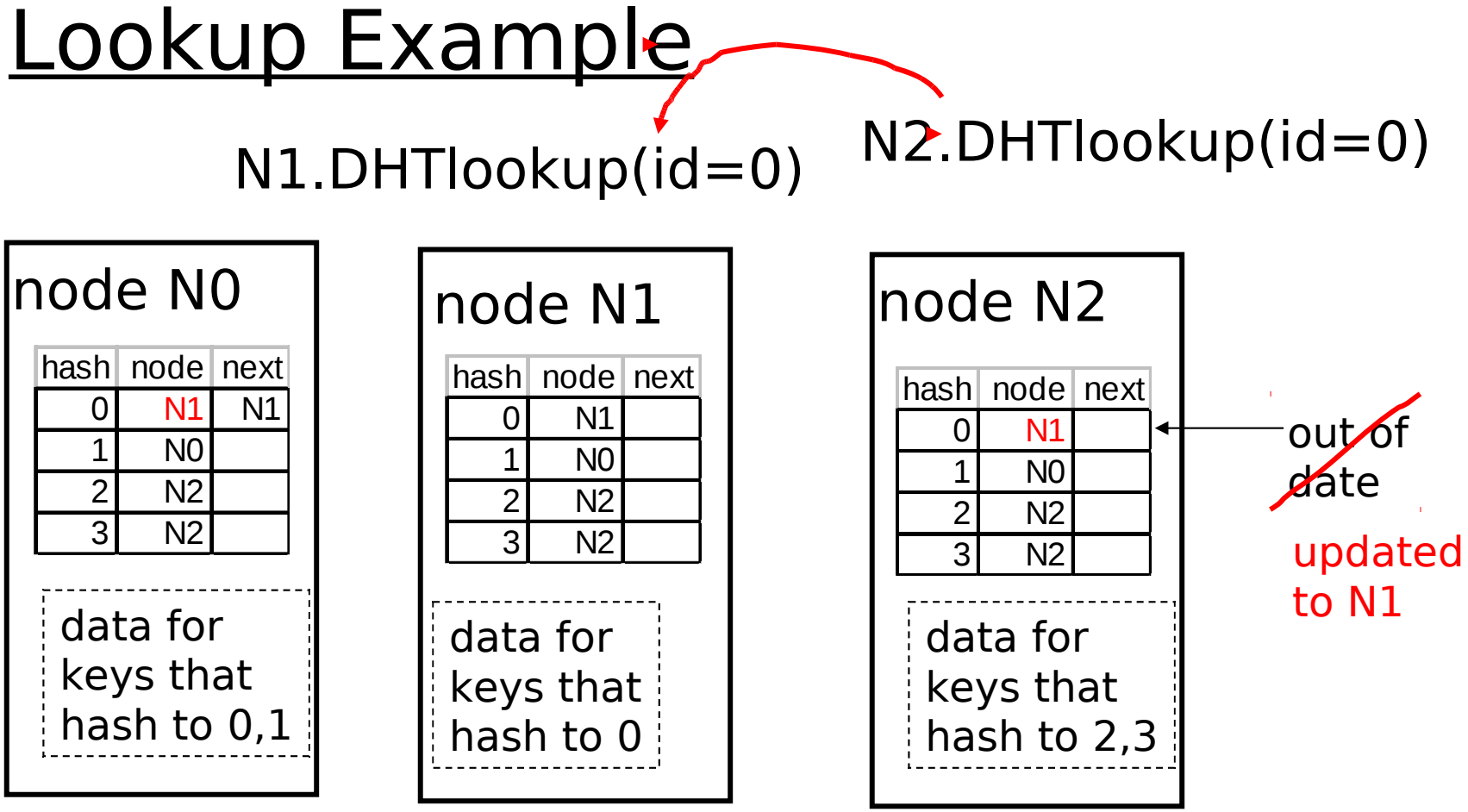
Lookup Example

N2.DHTlookup(id=0)



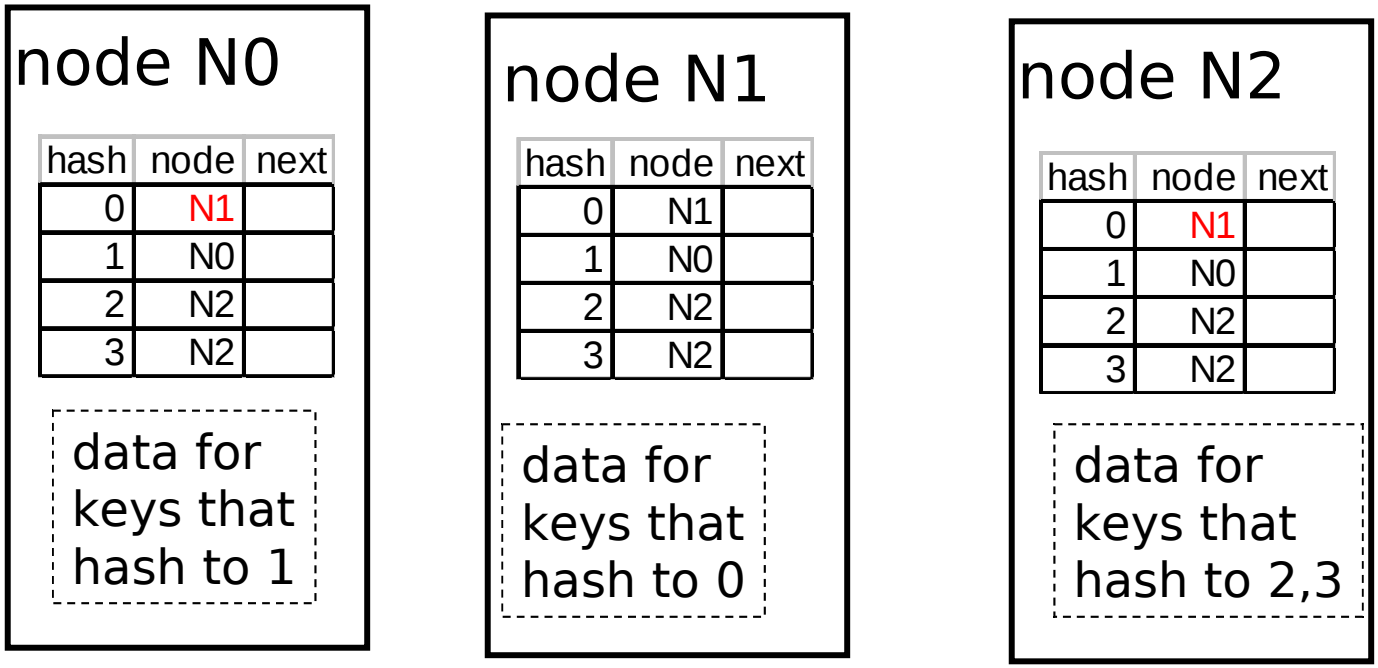
Updates to hash table in progress...

Lookup Example



Updates to hash table in progress...

Lookup Example



Final state...

Can we prove scheme works?

- Assume only joins, no failures
- Remember: OK to miss recent inserts
  - if not, need 2PC to handoff control...

Exercise

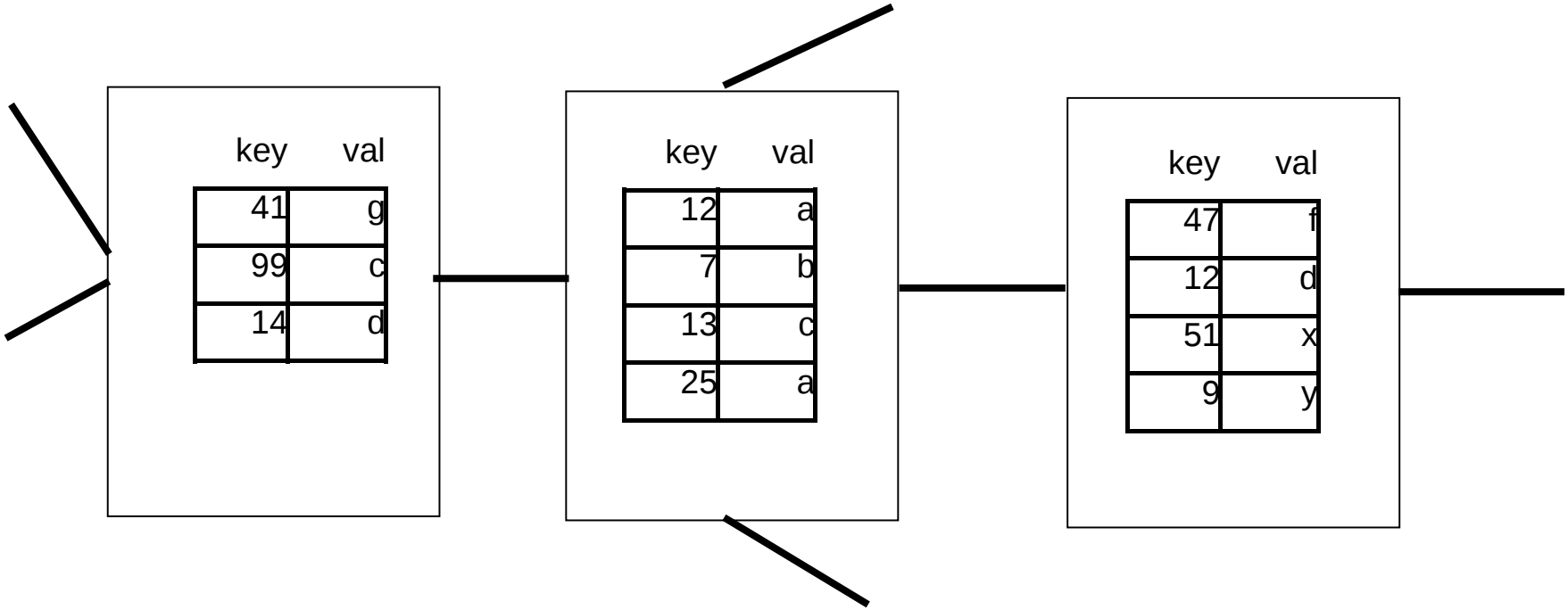
- Make HT extensible
- Start with m entries in HT, then dynamically double size
- Make sure we don’t get confused when HTs of different sizes exist

Chord vs Replicated HT

- Which code is simpler?
  - note that Chord code does not show data migration!
- Lookups O(log N) vs O(1)
- Impact of caching
- Routing table: Log N vs N
- Anonymity?
- Bootstrapping

Neighborhood Search (Gnutella)

- Each node stores its own data searches nearby nodes



Storing Data

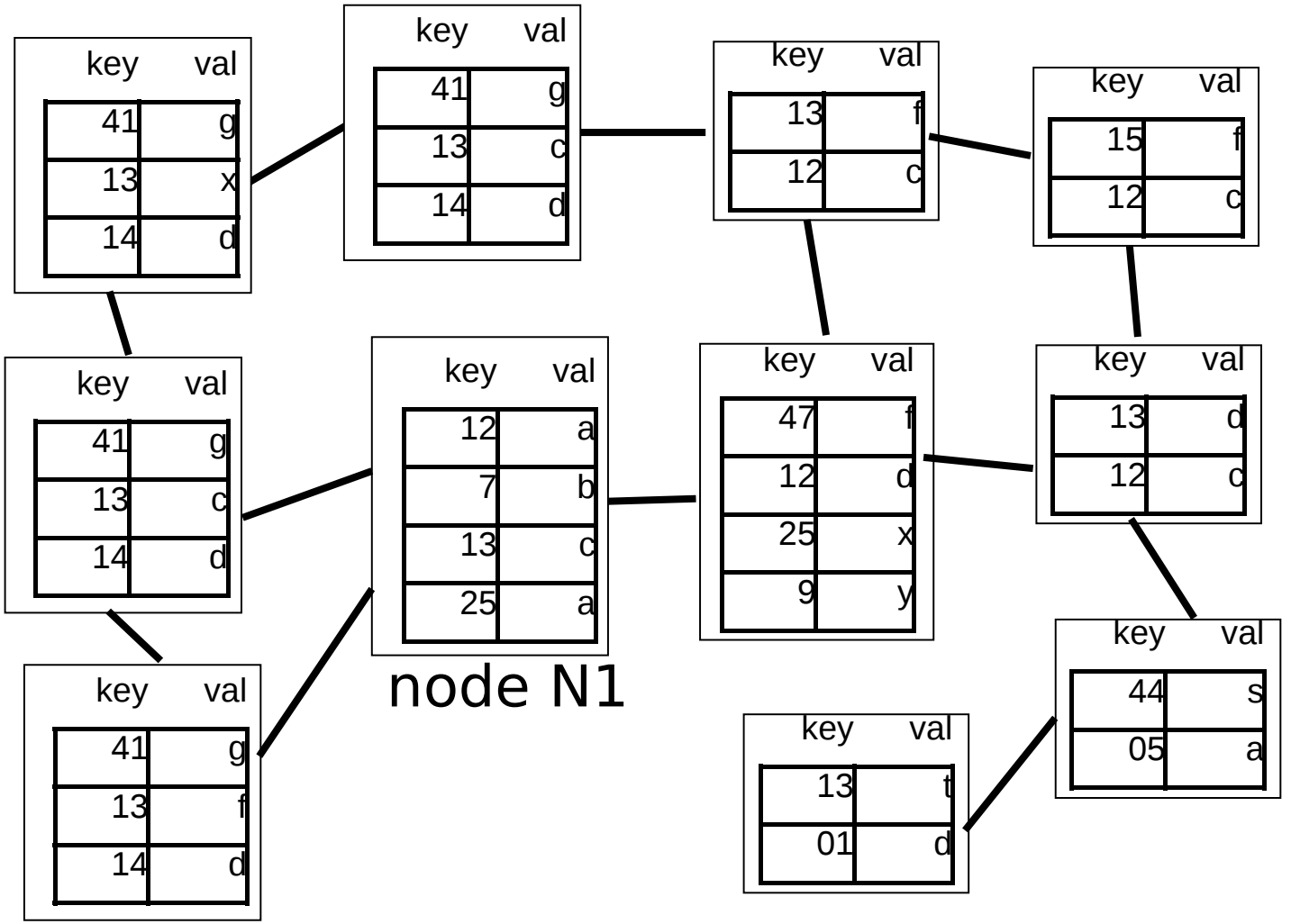
- X.DTinsert(k, v)
  insert (k,v) locally at X

Lookup

- X.DTlookup(k)
  TTL := desired value
  return(X.find(k, TTL, X))
- X.find(k, TTL, Z)
  TTL := TTL – 1
  S := local data pairs with key k
  if TTL > 0 then
  for all Y in X.neighbors (Y neq Z) do
  S:=S union Y.find(k, TTL, X)
  return(S)

Example

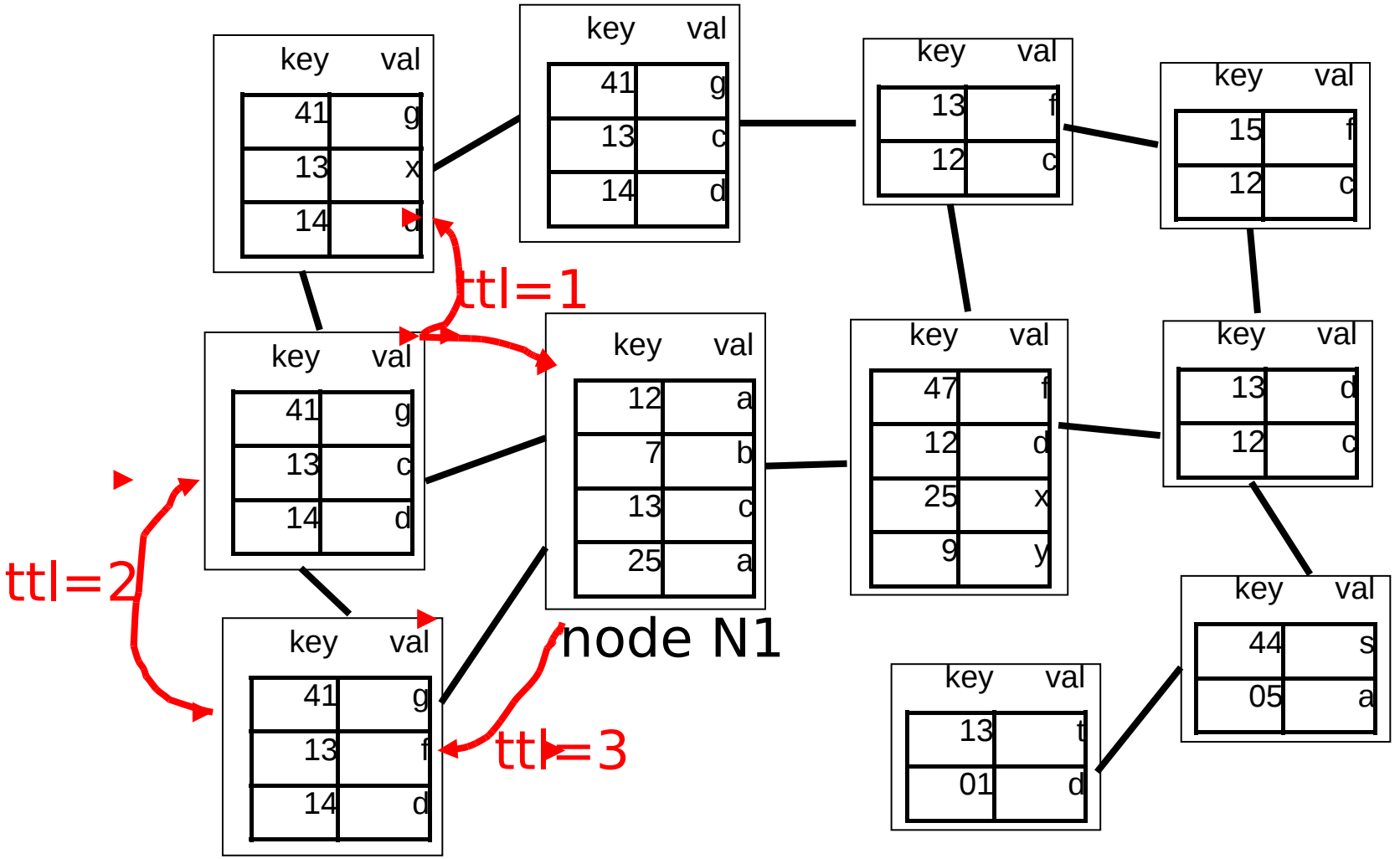
N1.DTlookup(13), TTL = 4



Example

N1.DTlookup(13), TTL = 4

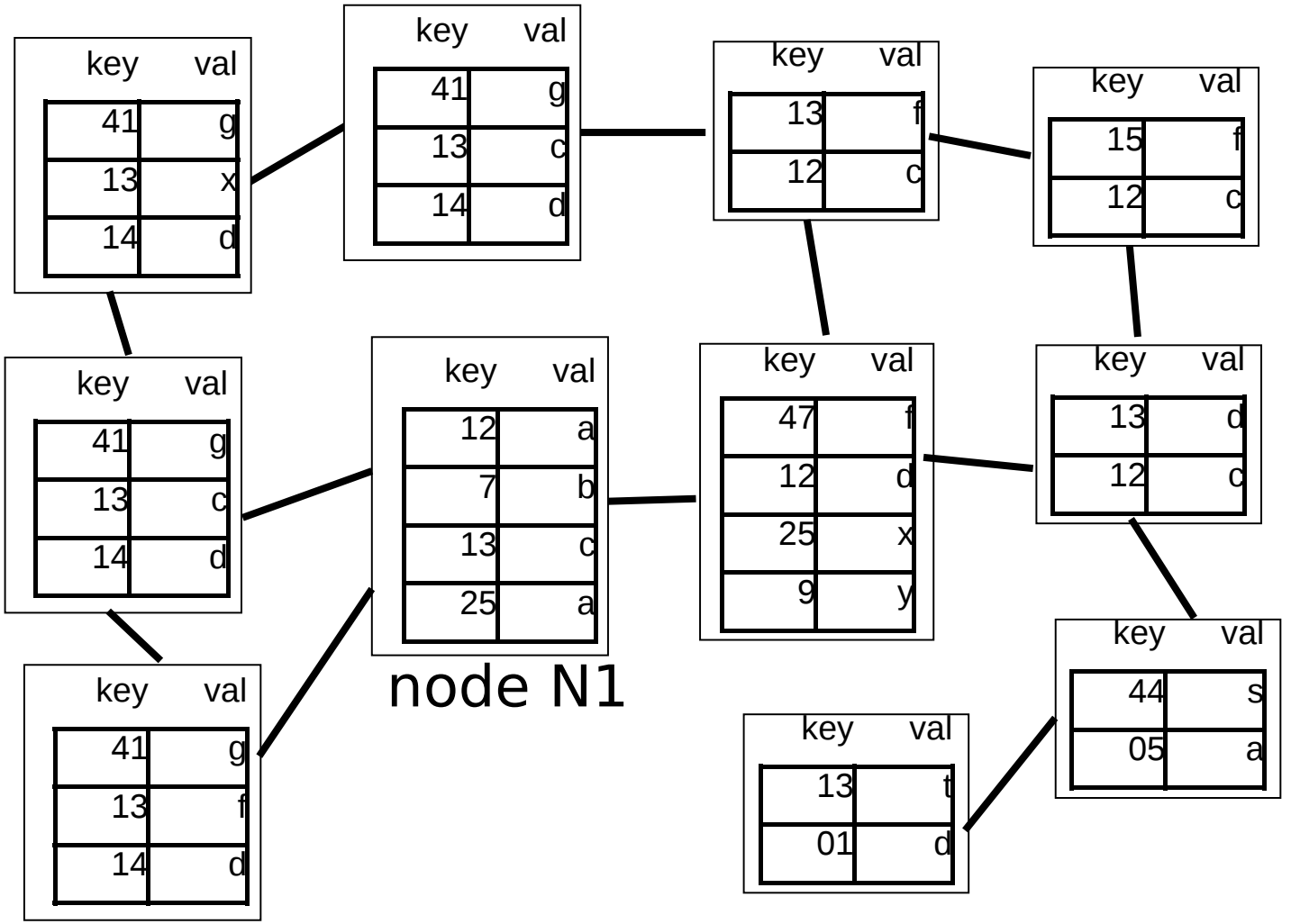
answer so far = {c, f, x}



Example

N1.DTlookup(13), TTL = 4

answer = {c, f, x, d} (no t!)

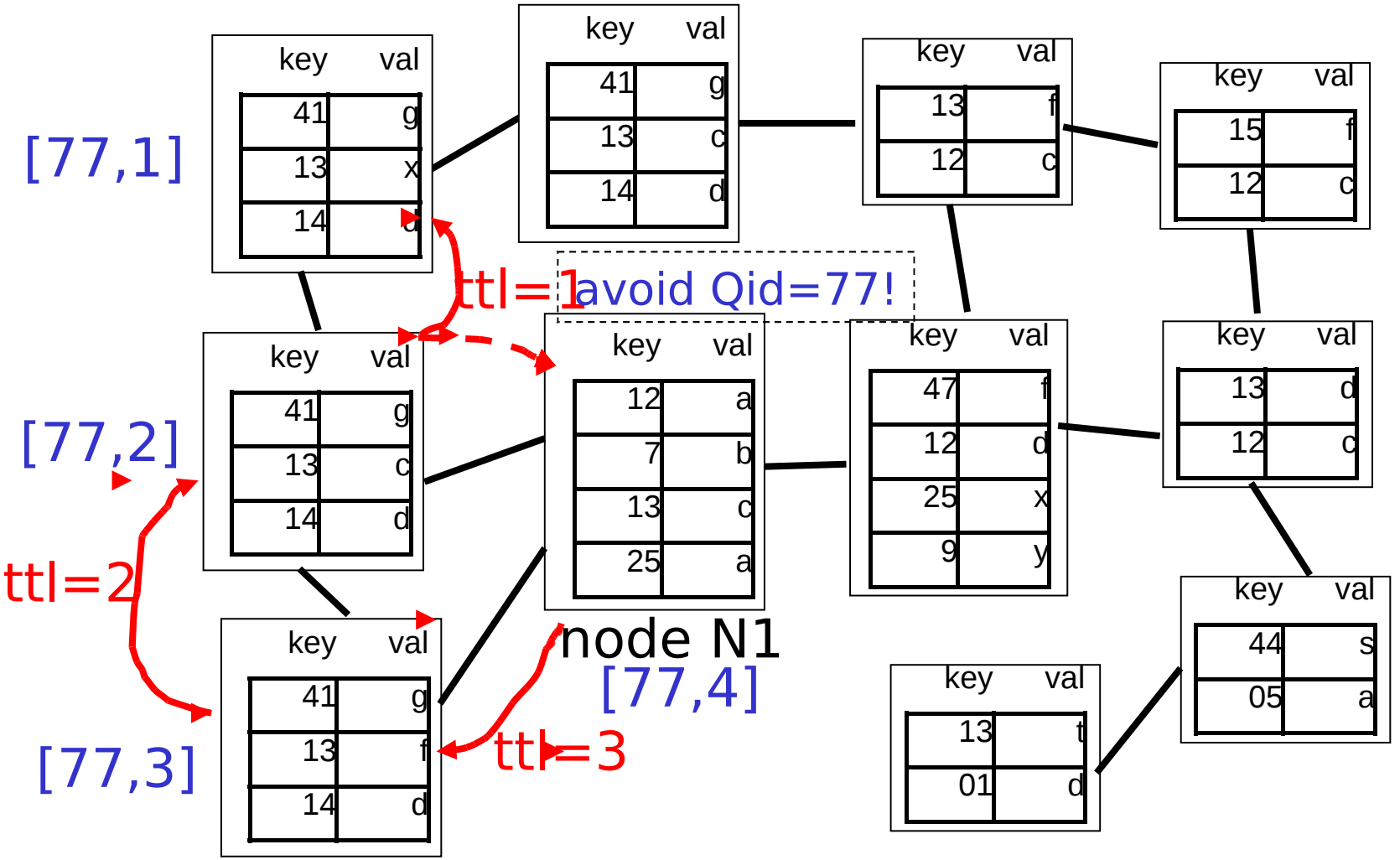


Optimization

- Queries have unique identifier
- Nodes keep cache of recent queries (query id plus TTL)

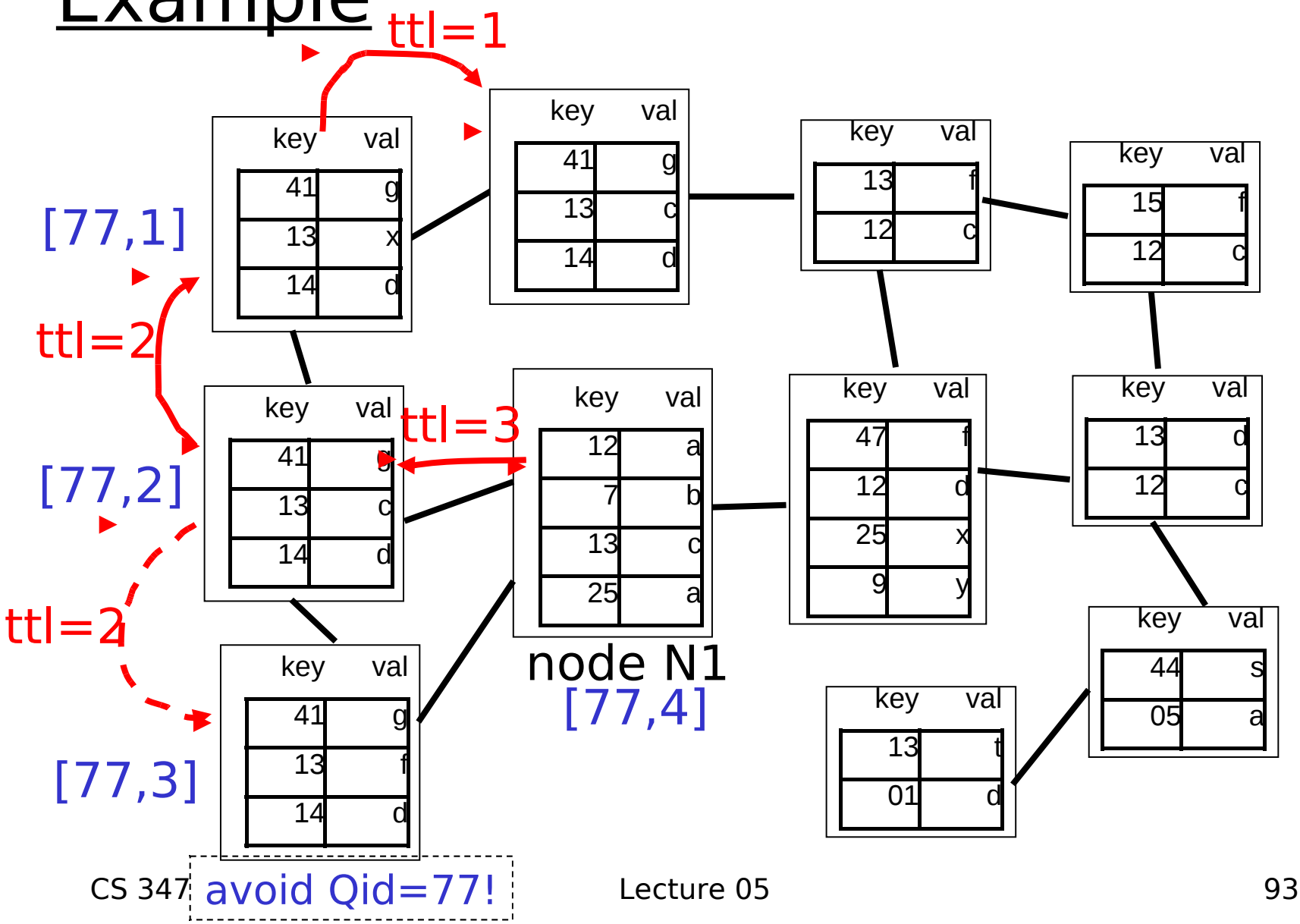
Example

N1.DTlookup(13), TTL = 4, Qid=77



Example

N1.DTlookup(13), TTL = 4, Qid=77



Joins

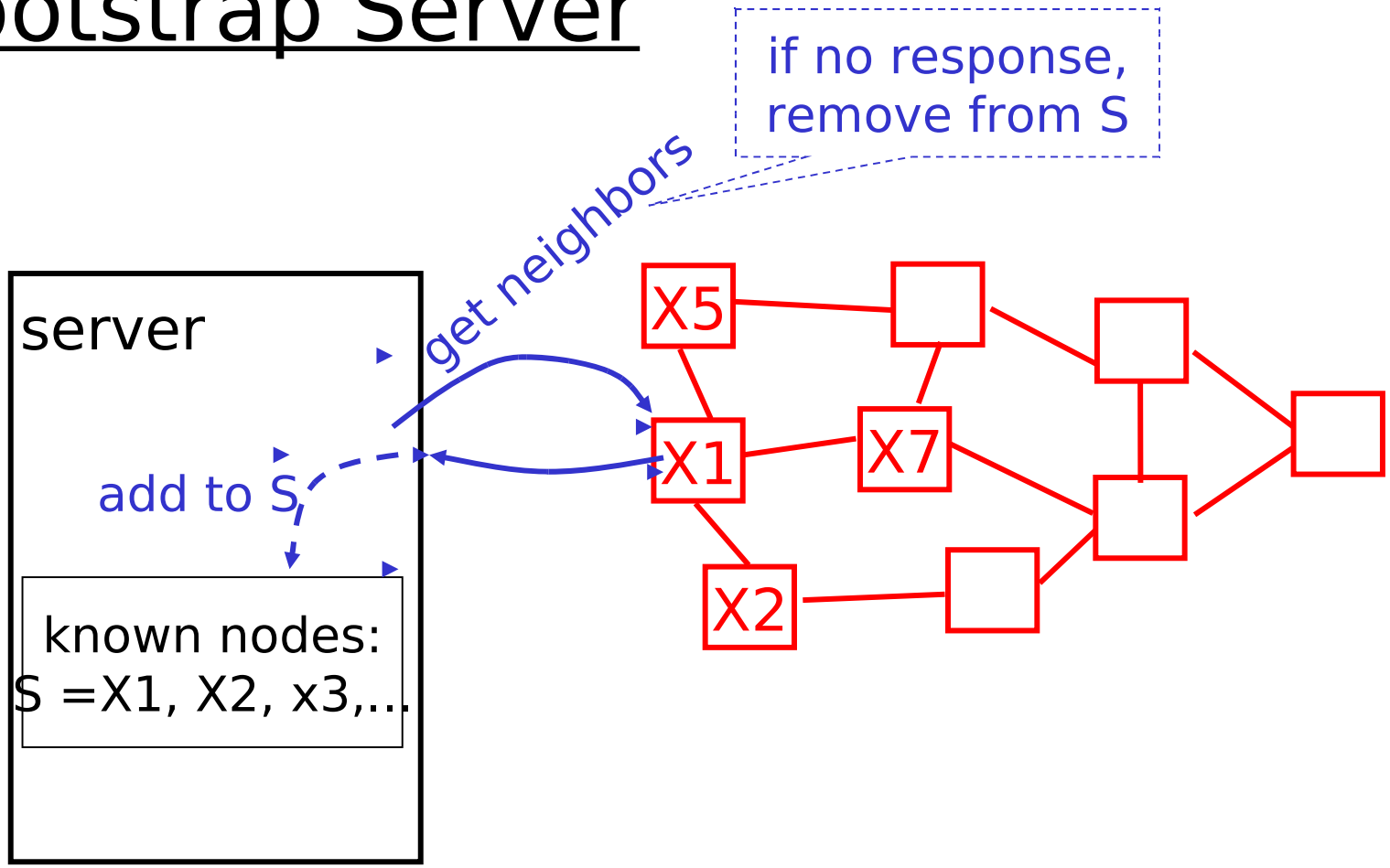
- X.join

  neighbors := {}
  cand:= nodes a “friend” recommends
  Z:= bootstrap server we hear about
  cand:= cand union Z.getNodes
  for Y in cand do
  ok := Y.wantMe(X)
  if ok then
  neighbors :=neighbors $\cup$ {Y}
  if |neighbors| > limit, return

Joins (continued)

- Y.wantMe(X)
  if I want X as neighbor then
  neighbors := neighbors $\cup$ {X}
  return (true)
  return (false)

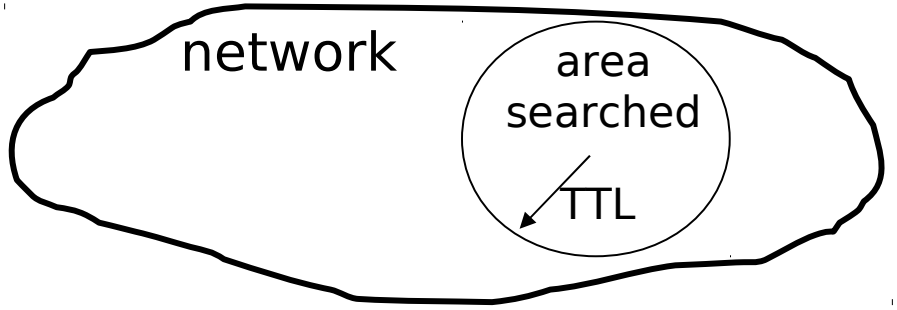
Bootstrap Server



sometimes called “pong server”

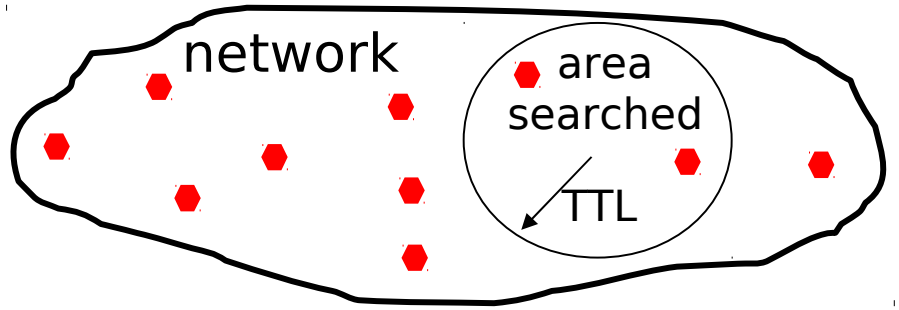
Problems with Neighborhood Search

- Unnecessary messages
- High load and traffic
  - Example: nodes have M new neighbors, number of messages is M^{TTL}
- Low capacity nodes are a bottleneck
- Do not find all answers



Why is Neighborhood Search Good?

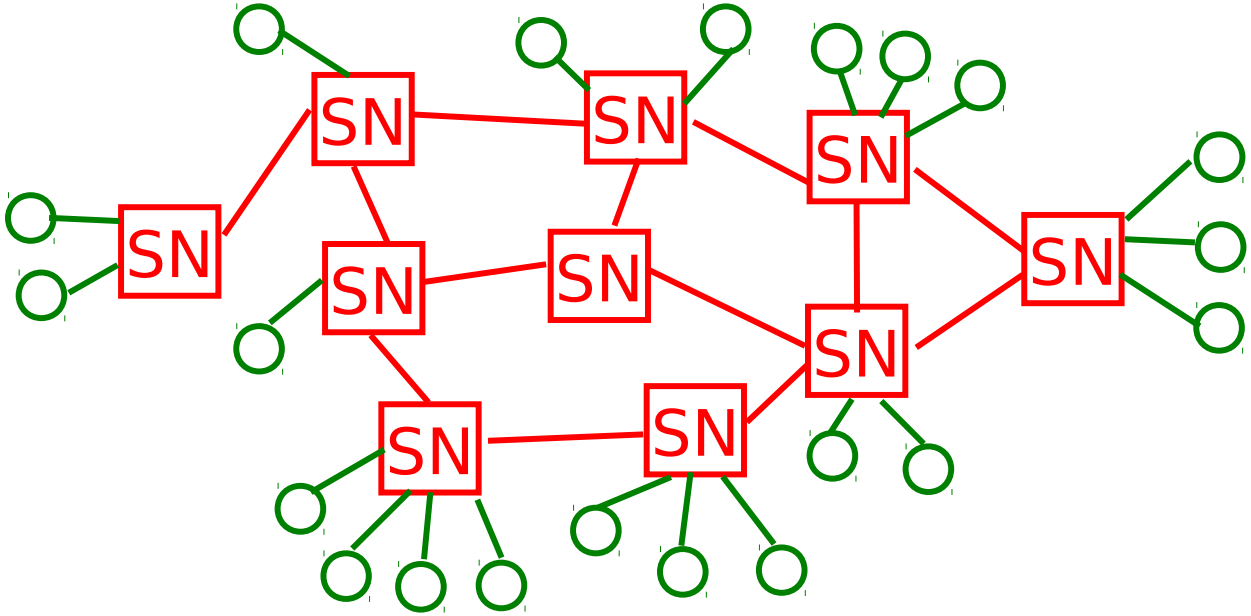
- Can pose complex queries
- Simple robust algorithm
- Works well if data is highly replicated



sites that have latest Madonna song

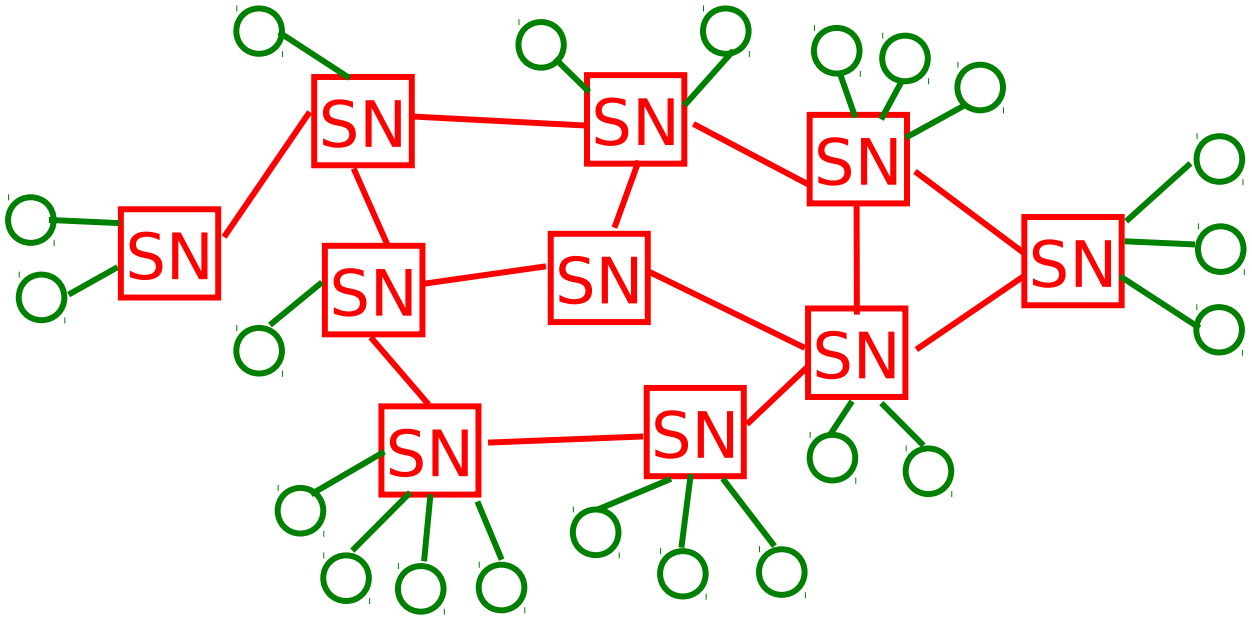
Super-Nodes

- Regular nodes index their content at super-nodes
- Super-nodes run neighborhood search



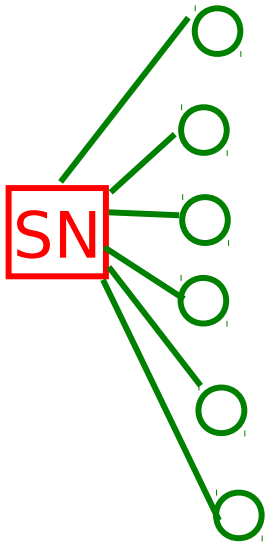
Motivation for Super-Nodes

- Take advantage of powerful nodes
- Searching larger index better than searching many smaller ones



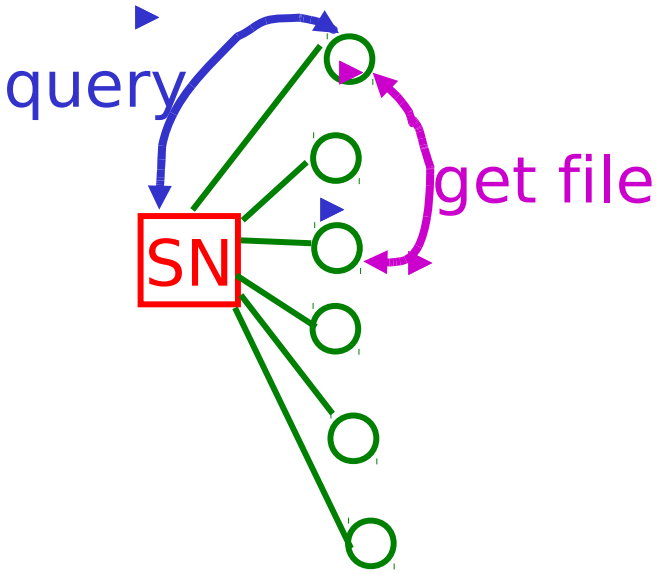
Napster (Original One)

- Single Super-Node



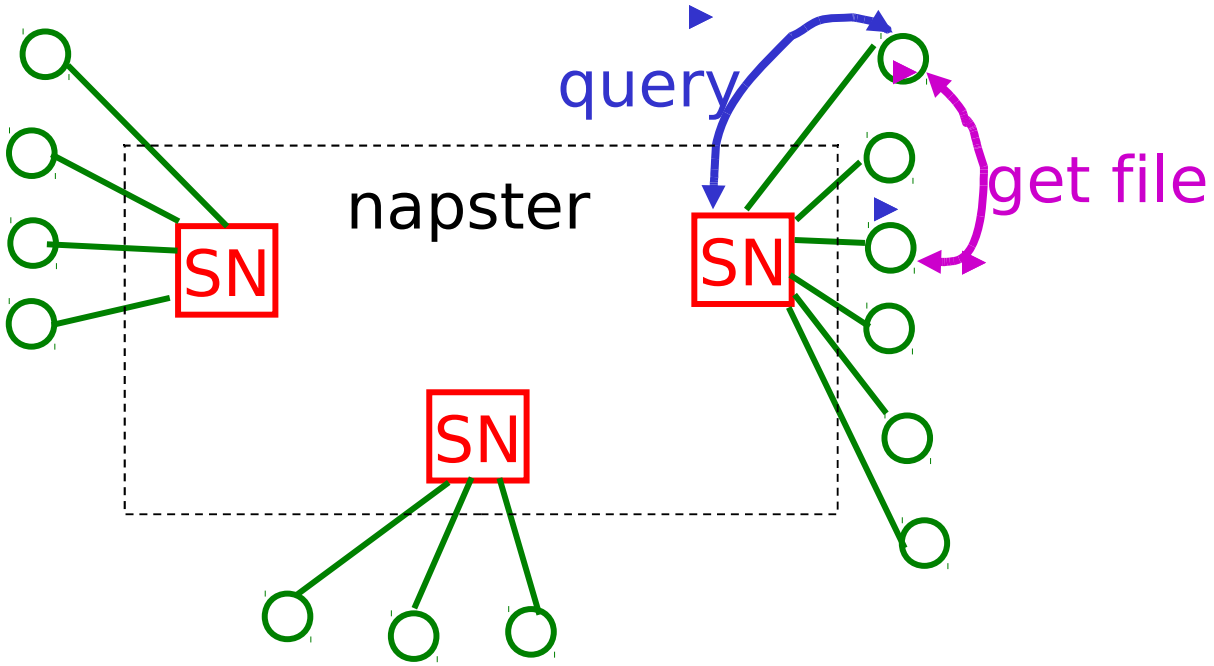
Napster (Original One)

- Single Super-Node



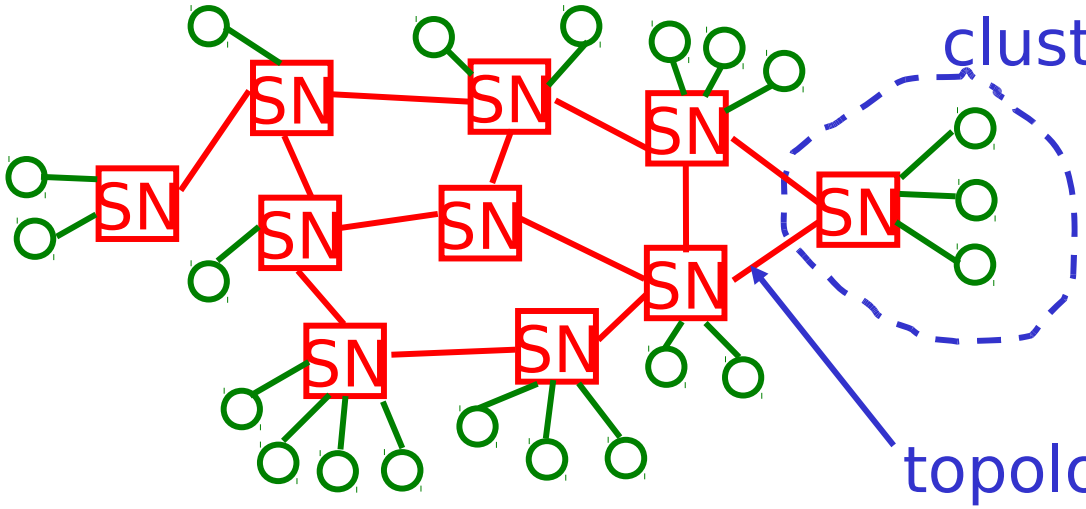
Napster (Original One)

- Actually, had several disconnected SNs



Performance Evaluation

- Yang, Beverly; Garcia-Molina, Hector. Designing a Super-peer Network, IEEE International Conference on Data Engineering, 2003.



number of nodes = 10,000

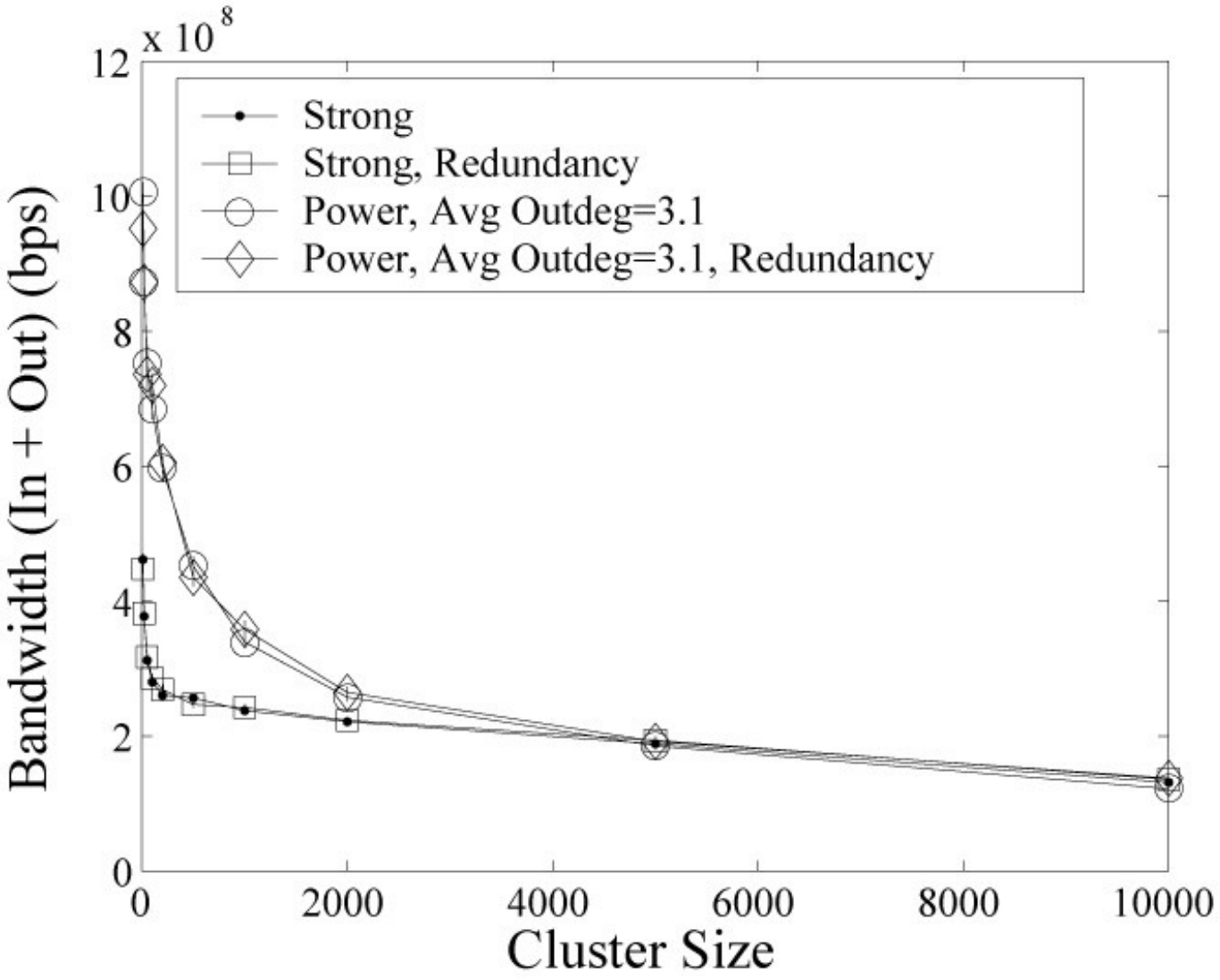
cluster size

topology

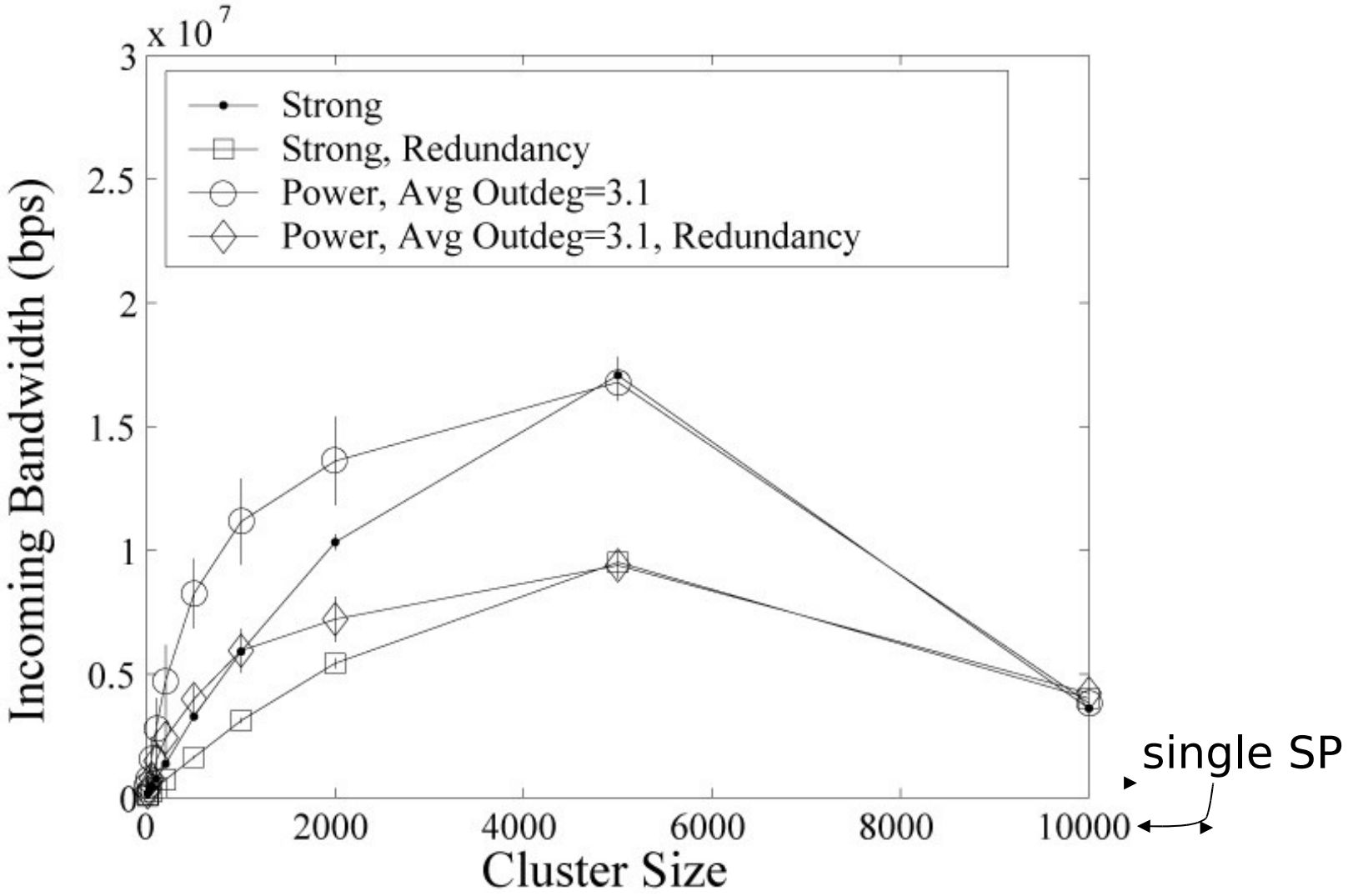
Metrics:

- Bandwidth
- Compute Load

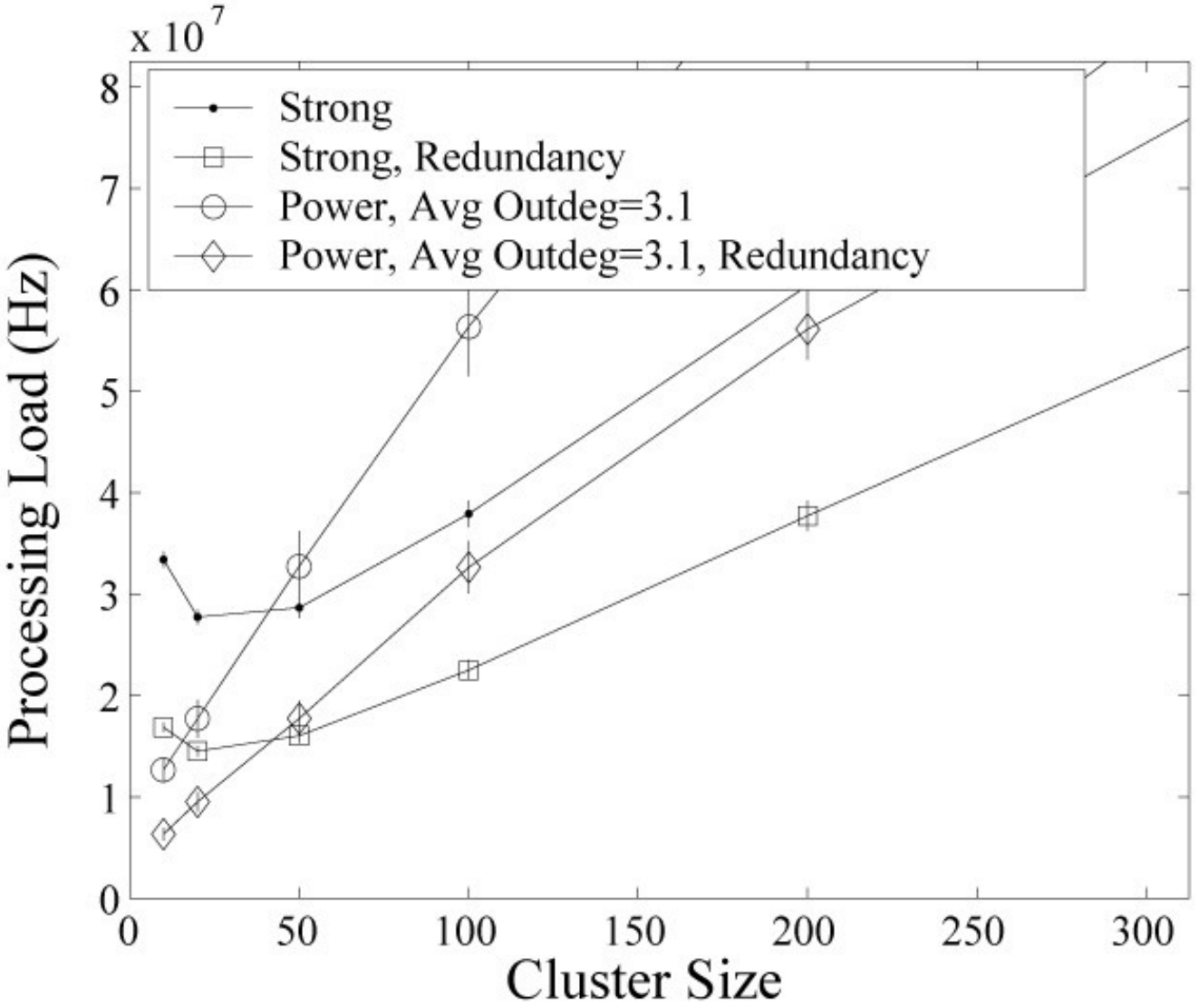
Aggregate Bandwidth



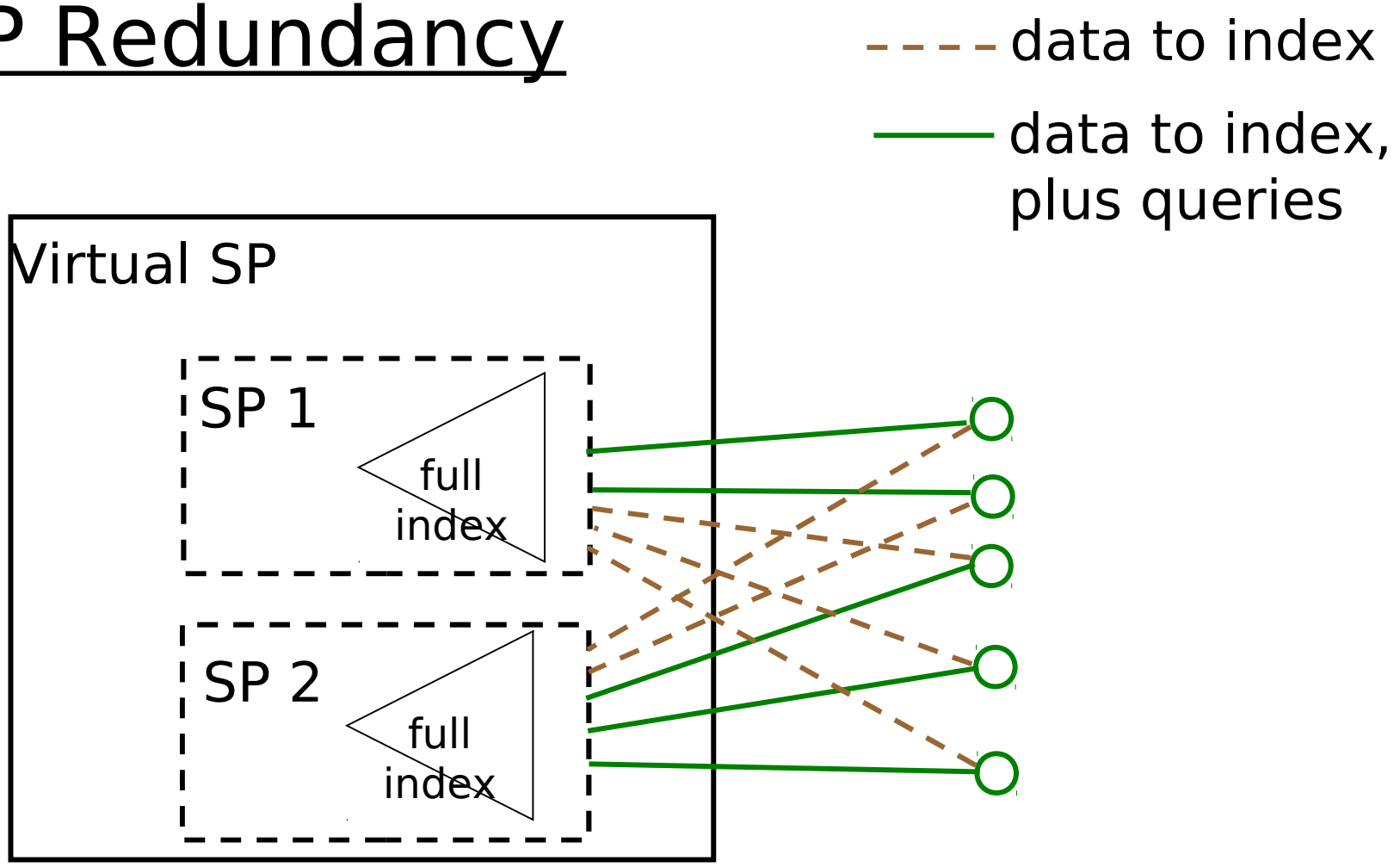
Individual Bandwidth Load



Individual Compute Load

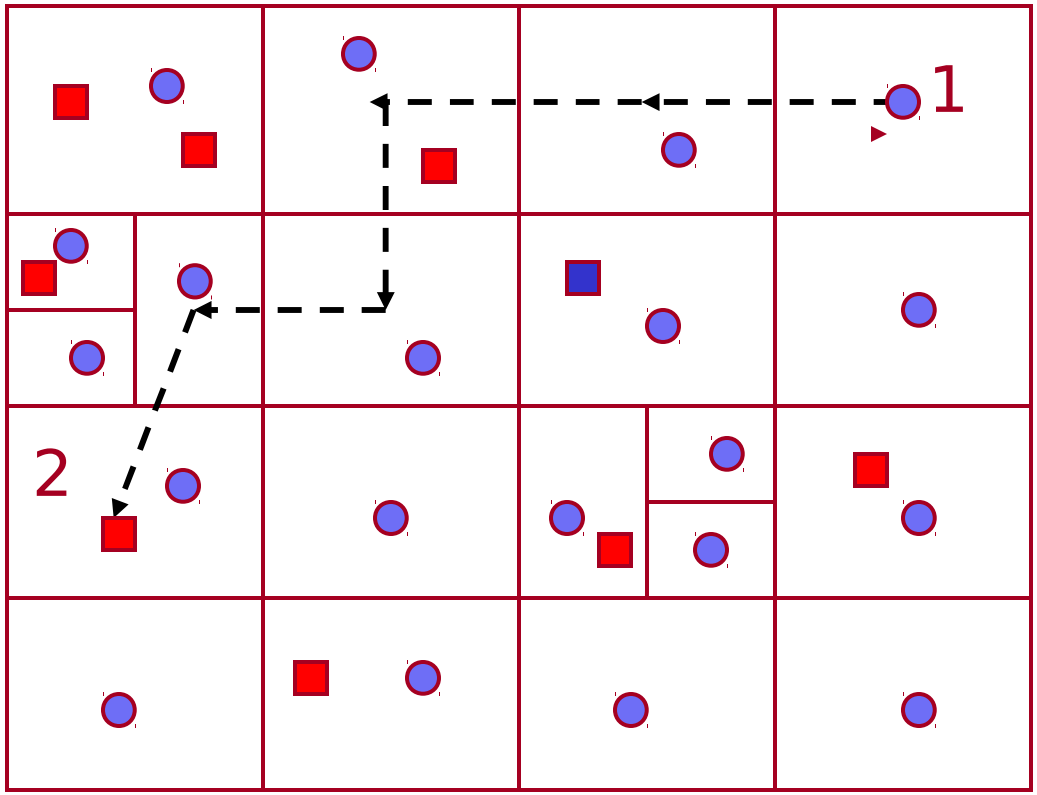


SP Redundancy



Content Addressable Network (CAN)

Nodes
Data



What is a P2P System?

- Multiple sites (at edge)
- Distributed resources
- Sites are autonomous (different owners)
- Sites are both clients and servers
- Sites have equal functionality

P2P Purity

P2P Benefits

- Pooling available (inexpensive) resources
- High availability and fault-tolerance
- Self-organization

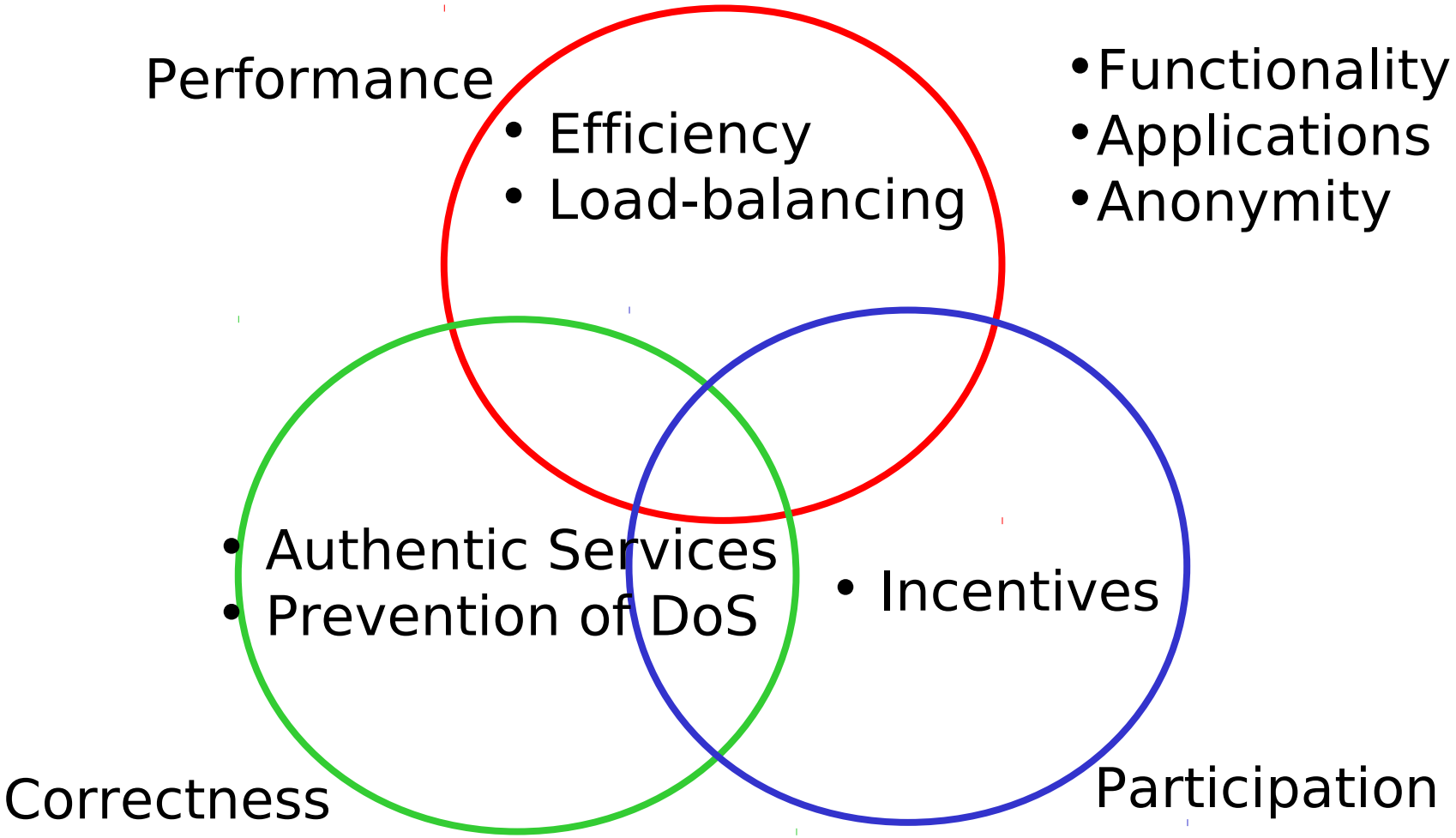
Comparison

	Gnutella	CAN	Others?
Expressivness	▯ ▯		
Comprehensivness	▪ ▪		
Autonomy	▪ ▪ ▪ ▪		
Efficiency	▪		
Robustness	▪ ▪ ▪		
Topology	pwr law		
Data Placement	arbitrary		
Message Routing	flooding		

Comparison

	Gnutella	CAN	Others?
Expressivness	■ ■ ■ ■	■	
Comprehensivness	■ ■	■ ■ ■ ■	
Autonomy	■ ■ ■ ■	■ ■	
Efficiency	■	■ ■ ■	
Robustness	■ ■ ■	■ ■	
Topology	pwr law	grid	
Data Placement	arbitrary	hashing	
Message Routing	flooding	directed	

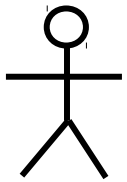
Open Problems



Open Problems: “Bad Guys”

- Availability (e.g., coping with DOS attacks)
- Authenticity
- Anonimity
- Access Control (e.g., IP protection, payments,...)

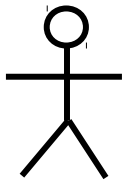
Authenticity



?

title: origin of species
author: charles darwin
date: 1859
body: In an island far, far away ...
...

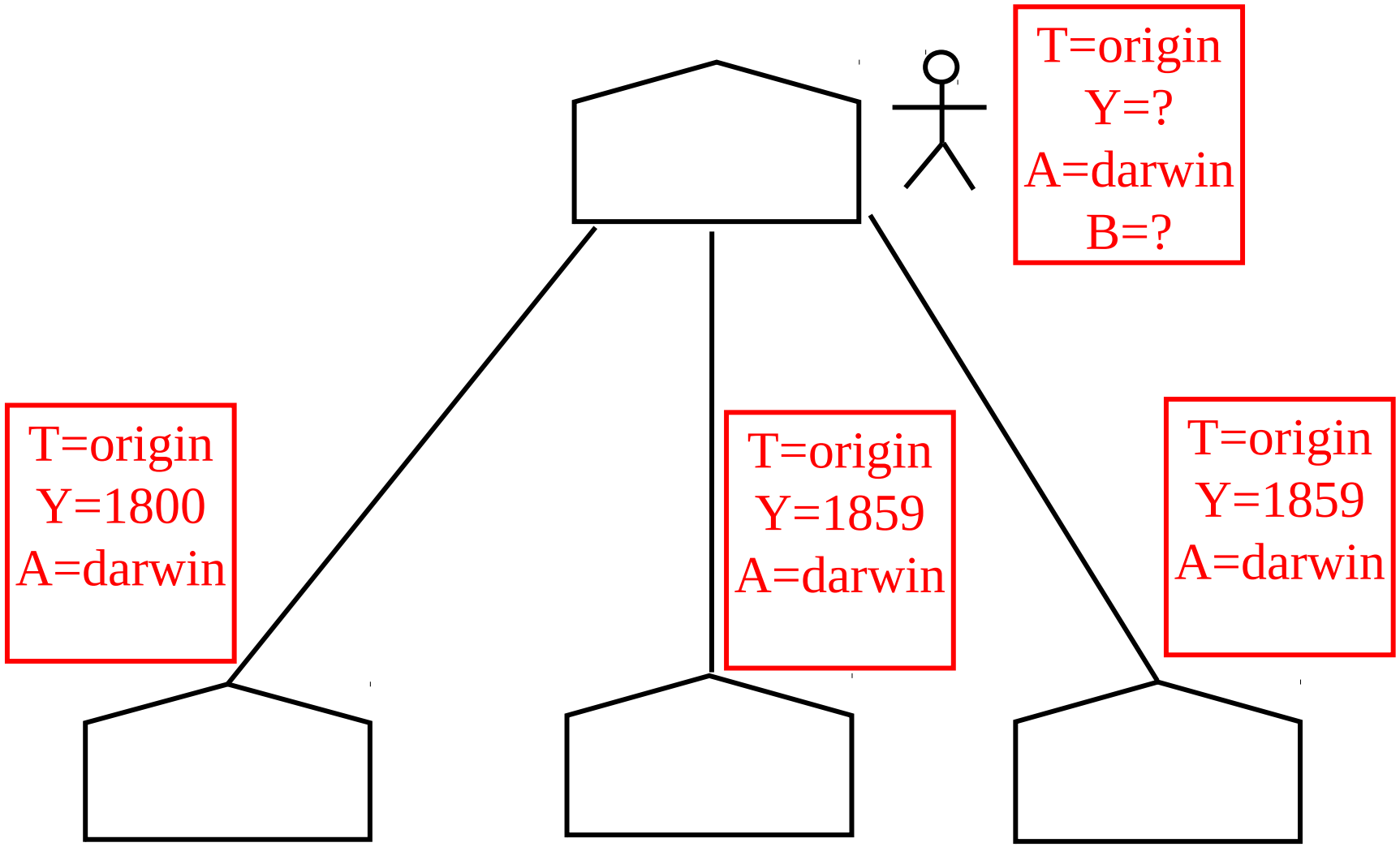
Authenticity



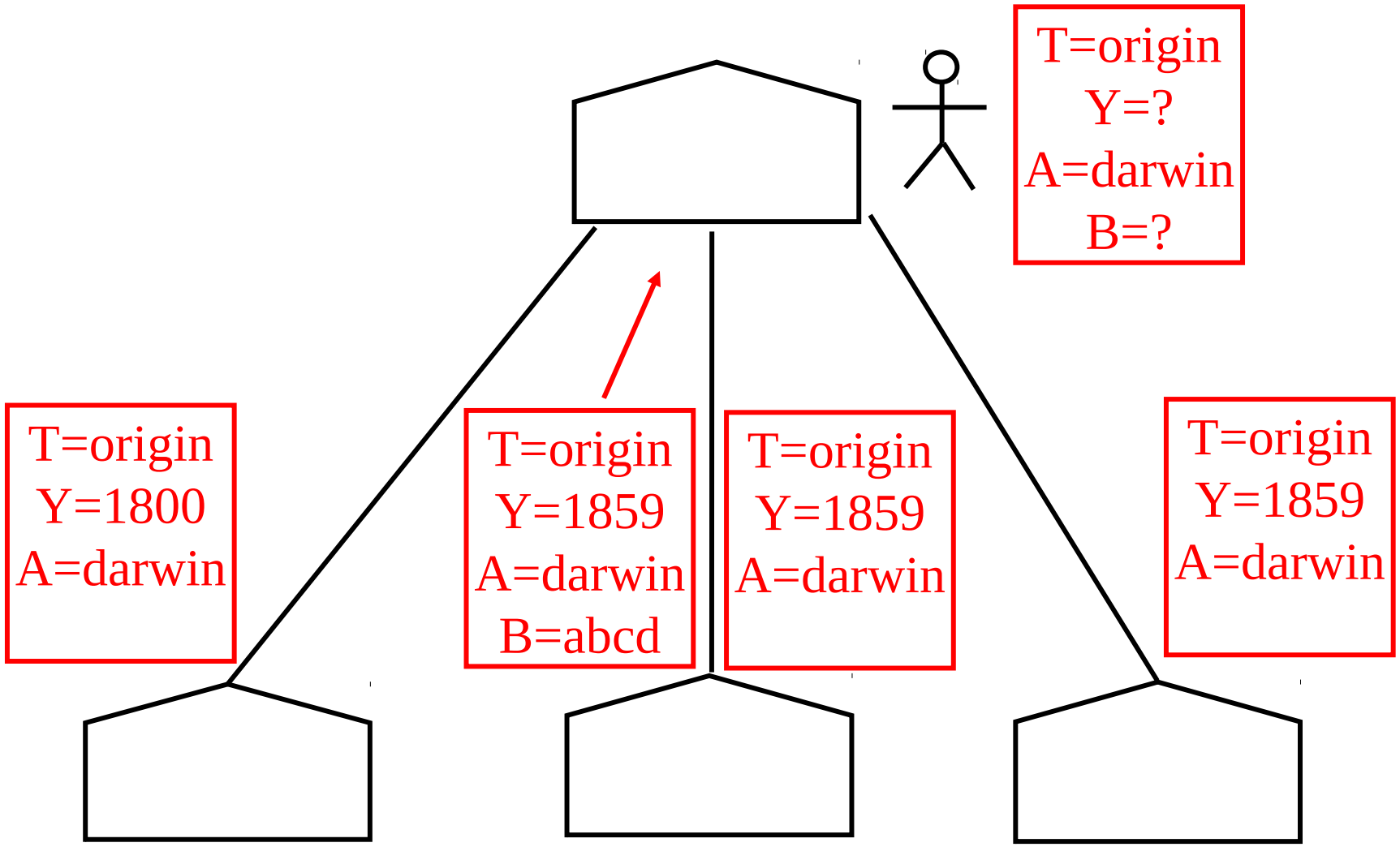
?

title: origin of species
author: charles darwin
date: 1859 00
body: In an island far, far away ...
...

More than Fetching One File



More than Fetching One File

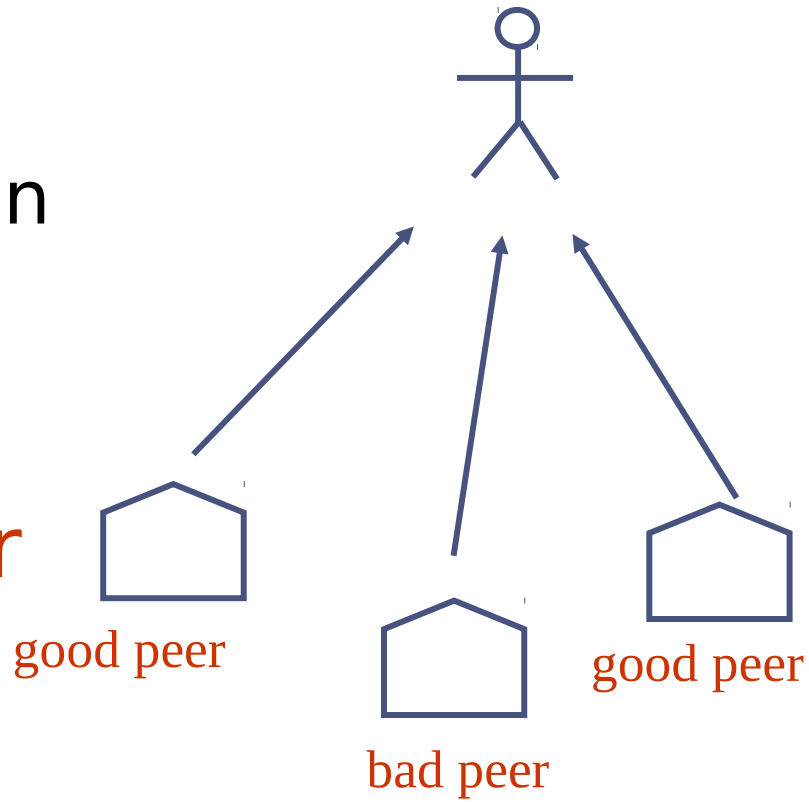


Solutions

- Authenticity Function A(doc): T or F
  - at expert sites, at all sites?
  - can use signature expert → sig(doc) → user
- Voting Based
  - authentic is what majority says
- Time Based
  - e.g., oldest version (available) is authentic

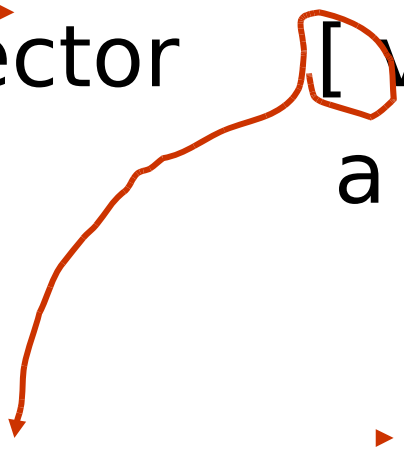
Added Challenge: Efficiency

- Example: Current music sharing
  - everyone has authenticity function
  - but downloading files is expensive
- Solution: Track peer behavior



How to Track Peer Behavior?

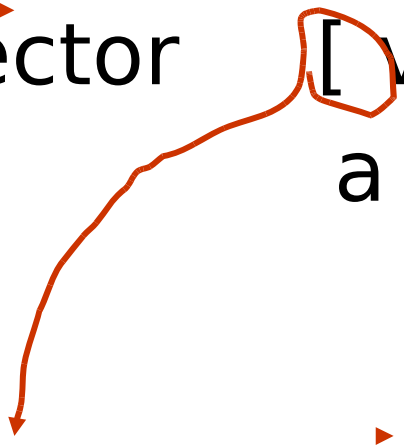
- Trust Vector [v_1, v_2, v_3, v_4]
  a b c d



- Single value between 0 and 1?

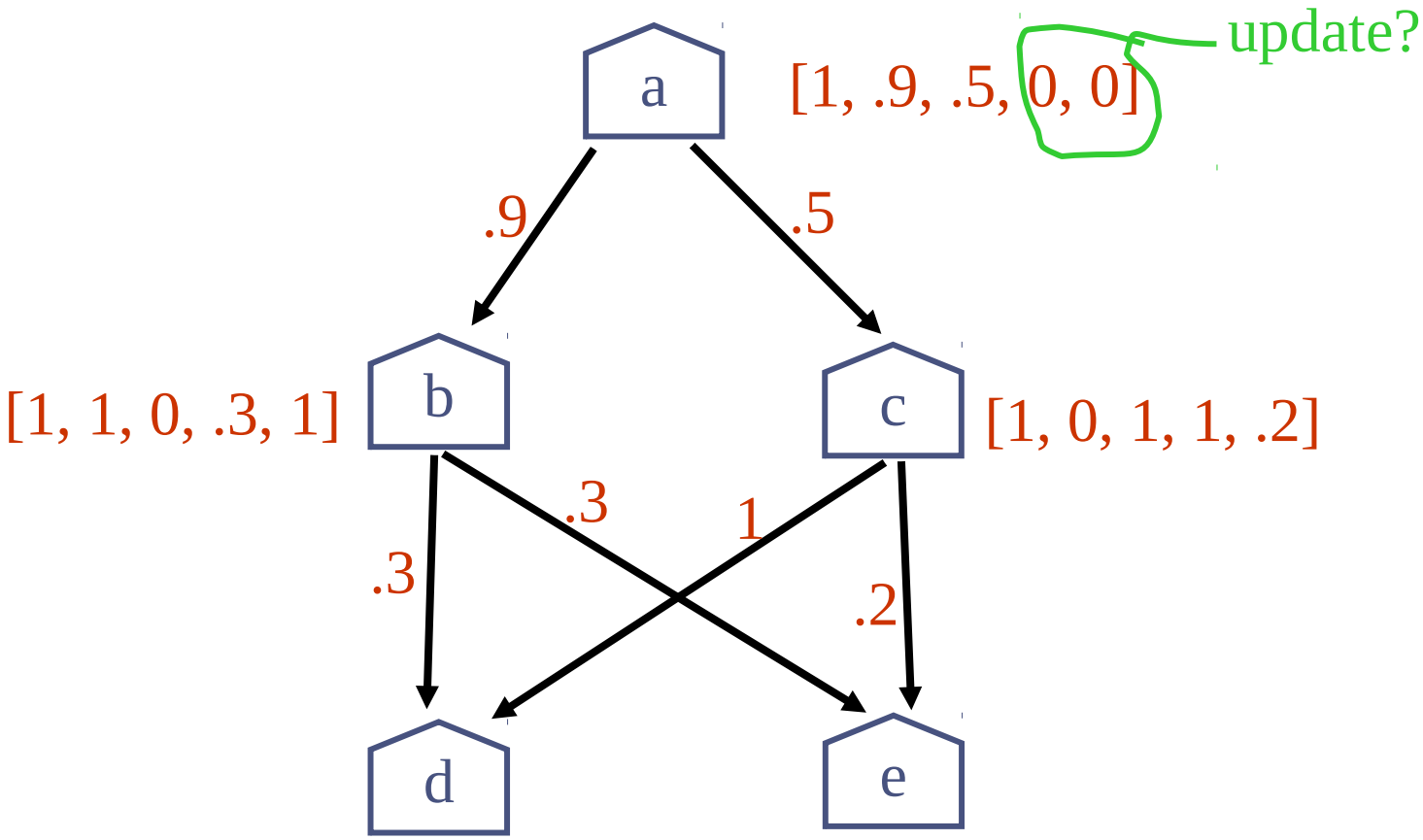
How to Track Peer Behavior?

- Trust Vector [v_1, v_2, v_3, v_4]
  a b c d



- Single value between 0 and 1?
- Pair of values
  [total downloads, good downloads] ?

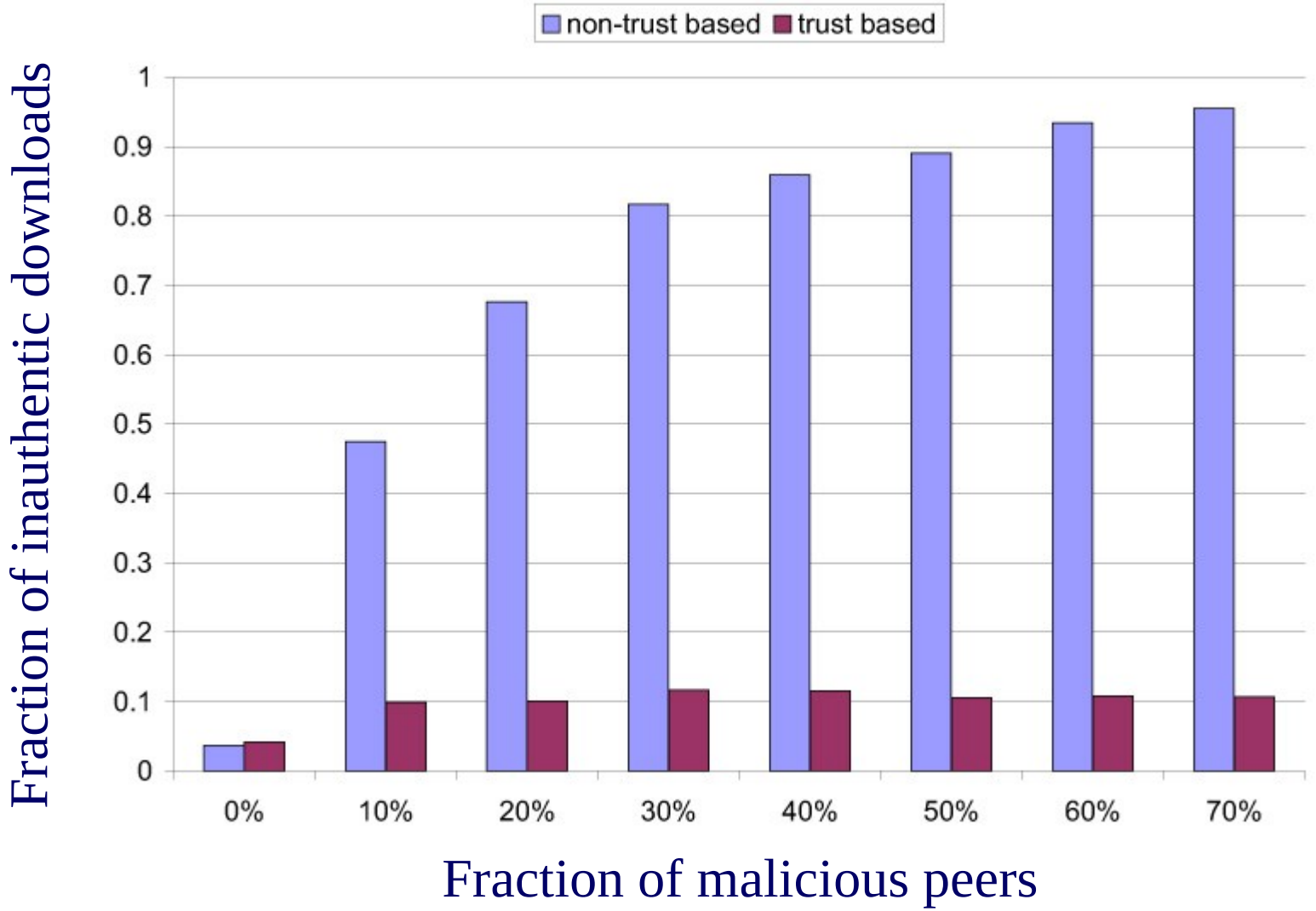
Trust Operations



Issues

- Trust computations in dynamic system
- Overloading good nodes
- Bad nodes can provide good content sometimes
- Bad nodes can build up reputation
- Bad nodes can form collectives
- …

Sample Results



Participation & Incentives

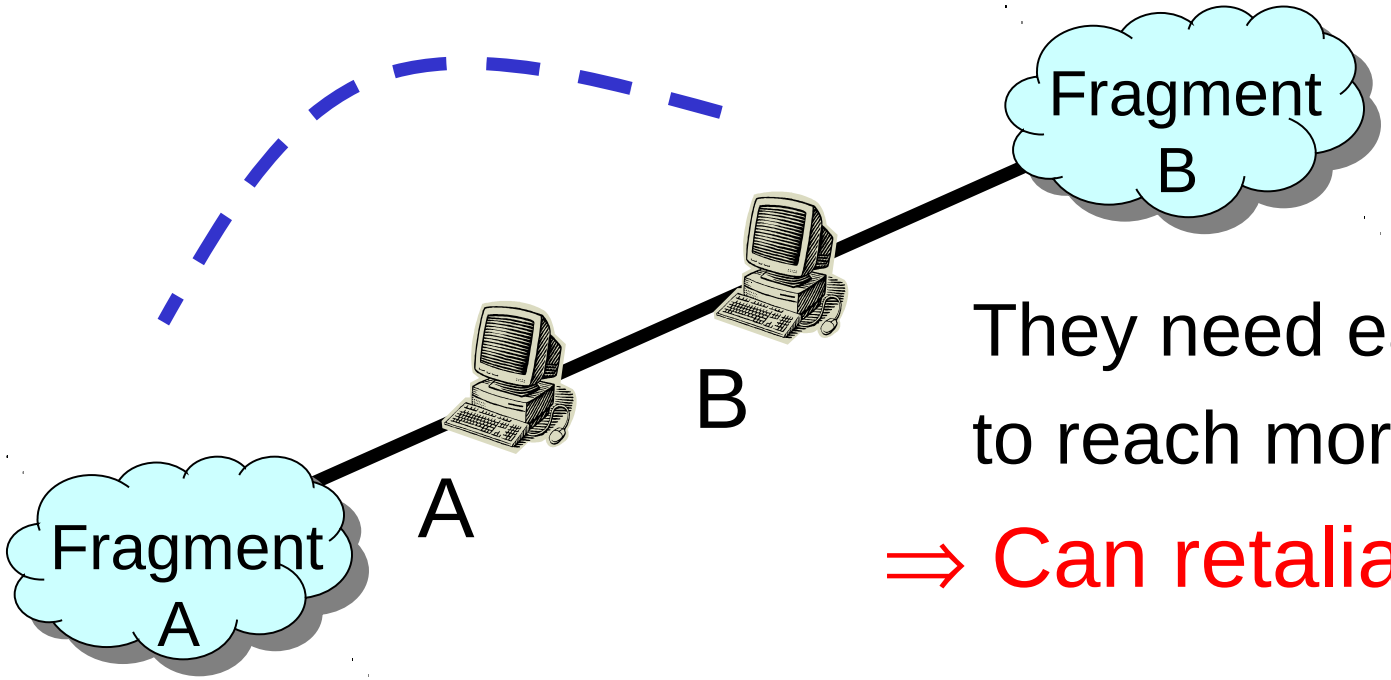
- Autonomous nodes need incentives to work together
  - Forward messages
  - Perform computations
  - Share/store files
  - Provide services
  - Etc.

Incentive types

- Three main kinds of incentives (thus far):
  - Tit for tat
  - Reputation
  - Money/Currency

Tit-for-Tat

- "I do to you what you do for me"
- Example

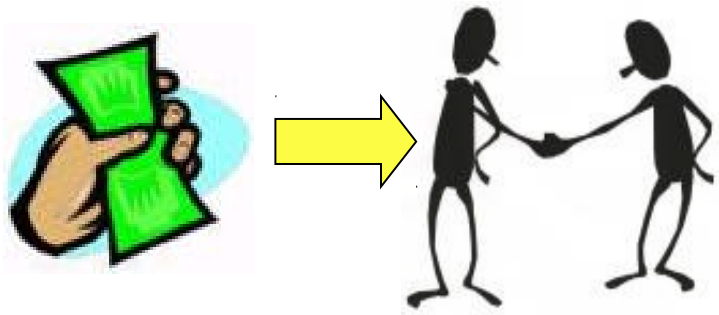


They need each other to reach more nodes.

⇒ Can retaliate

Reputation and Currency

- “If you do something for me, I will give you reputation/money”



Pros and Cons

- Tit-for-Tat
  - Pros: Requires minimal infrastructure and overhead, least prone to cheating
  - Cons: Requires symmetric relationships
- Currency
  - Pros: Everyone wants money! In some applications it is required
  - Cons: Requires the heaviest infrastructure
- Reputation
  - Applies to most situations, but has some overhead, as well as it's own incentive issues

Reputation and Currency

- For these techniques, there are 2 questions:
  - If we have money/reputation scores, how do we use it to give peers incentives?
  - How do we implement money/reputation score in a P2P fashion?

P2P Summary

- Search
  - Chord DHT
  - Replicated DHT
  - Gnutella
  - Super-Peers
- Dealing with Bad Guys
- Dealing with Lazy Guys