Algorithmic Aspects of Distributed k-ary System

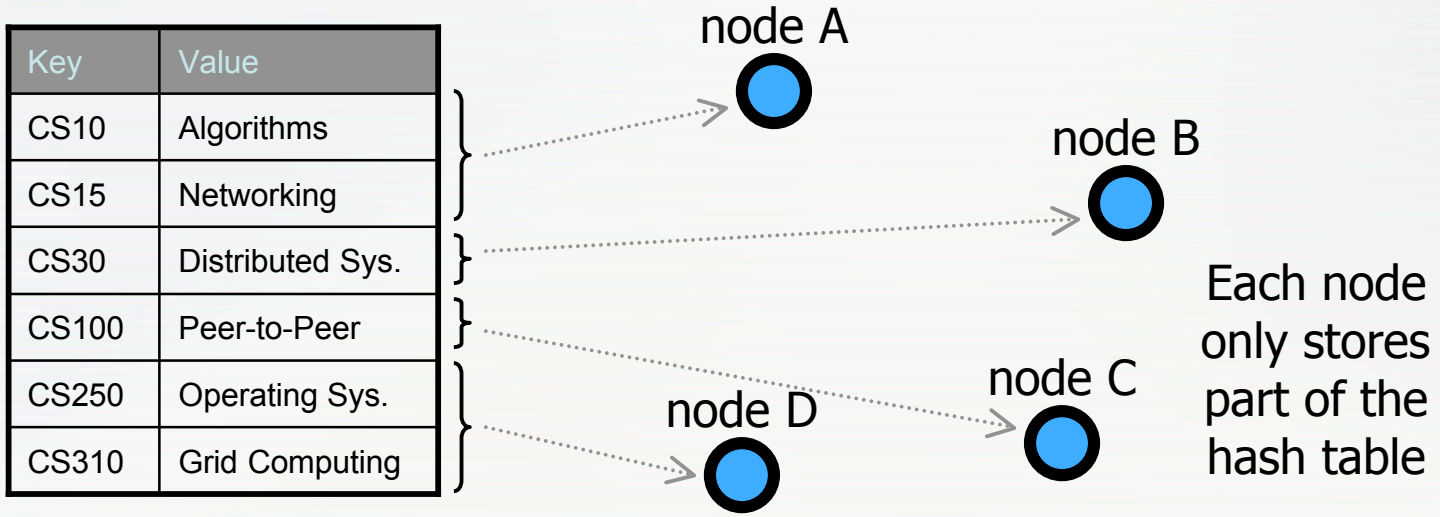
Seif Haridi
Ali Ghodsi
KTH/Royal Institute of Technology
Swedish Institute of Computer Science (SICS)

Outlook

- DHT Introduction
  - What is a DHT
  - Chord: How to partition the data
  - Chord: How to interconnect nodes
  - Chord: How to speed up search
  - Chord: How to maintain pointers
- DKS Algorithms
  - K-ary Search
  - Atomic Join and Leaves
  - Broadcast Algorithms
  - Multicasting
  - Bulk Operations
  - Replication
  - Topology Maintenance

What is a Distributed Hash Table (DHT)? 1/2

- **An infrastructure that enables the distribution of an ordinary *hash table* onto a set of cooperating nodes**



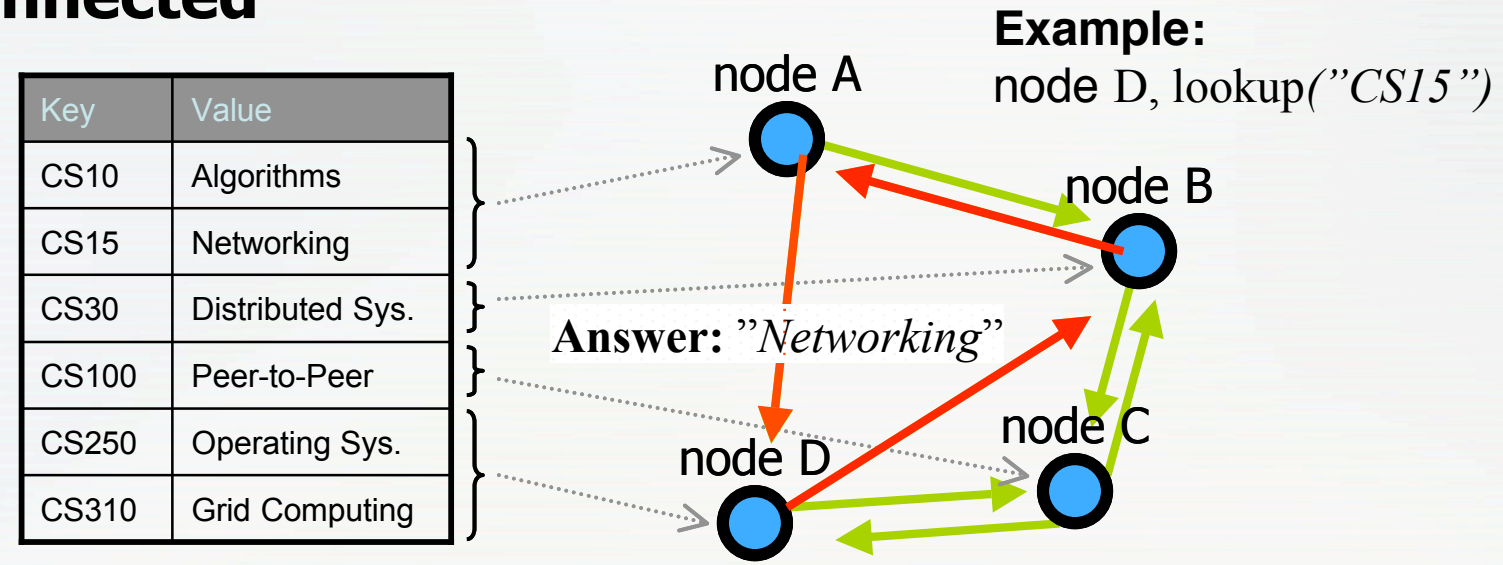
- **The DHT provides a basic *lookup service*, which allows *any* node to find the value associated with a given key**

- **Example:**

  lookup("*CS30*"), at any node should return: *"Distributed Sys."*

What is a Distributed Hash Table (DHT)? 2/2

- **To provide the lookup service, the nodes must be interconnected**



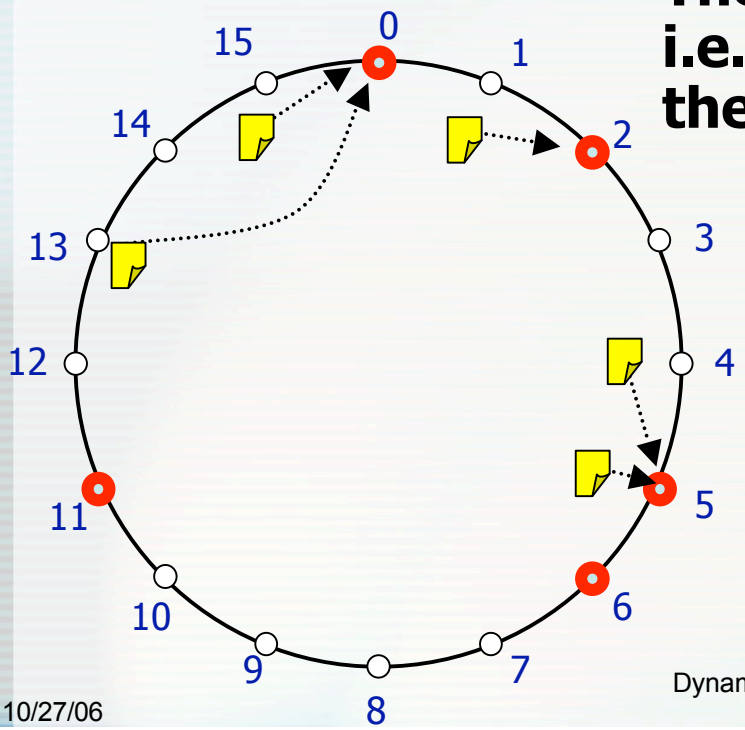
- **Each node maintains a *routing table* with pointers to some other nodes such that lookup requests can be routed to the node storing the requested key/value-pair (a.k.a. item)**

Outlook

- DHT Introduction
  - What is a DHT
  - Chord: How to partition the data
  - Chord: How to interconnect nodes
  - Chord: How to speed up search
  - Chord: How to maintain pointers
- DKS Algorithms
  - K-ary Search
  - Atomic Join and Leaves
  - Broadcast Algorithms
  - Multicasting
  - Bulk Operations
  - Replication

How do we distribute the hash table?

- **Use a logical name space called the *identifier space*, consisting of identifiers {0,1,2,…, N-1}**
- **The identifier space can be perceived as a logical ring modulo N**
- **Every node is assigned an identifier using a function H.**
- **Items are mapped to the identifier space using a function H, every node knows H**
- **The items are stored at their *successor*, i.e. the first node encountered moving in the clockwise direction**



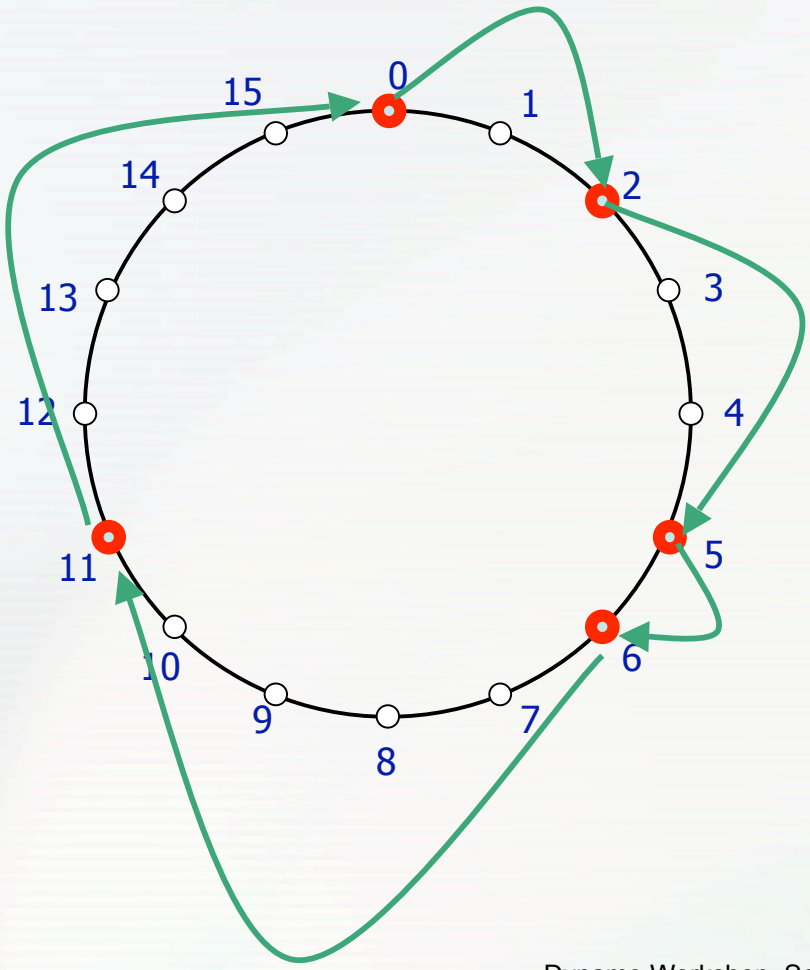
- Example: N=16, nodes {**a,b,c,d,e**}, and 5 items
- Node **a** gets identifier 0 since H_1(**a**)=0, the other nodes **b, c, d, e,** get identifiers 2, 5, 6, 11 the same way
- Item (”**cs15**”, ”**networking**”) is mapped to identifier 13 since H_2 (”**cs15**”) = 13, other items are similarly to 15, 2, 4, 5

Outlook

- DHT Introduction
  - What is a DHT
  - Chord: How to partition the data
  - Chord: How to interconnect nodes
  - Chord: How to speed up search
  - Chord: How to maintain pointers
- DKS Algorithms
  - K-ary Search
  - Atomic Join and Leaves
  - Broadcast Algorithms
  - Multicasting
  - Bulk Operations
  - Replication
  - Topology Maintenance

How do we interconnect the nodes?

- **Each node maintains a *routing pointer* to the successor in the ring**



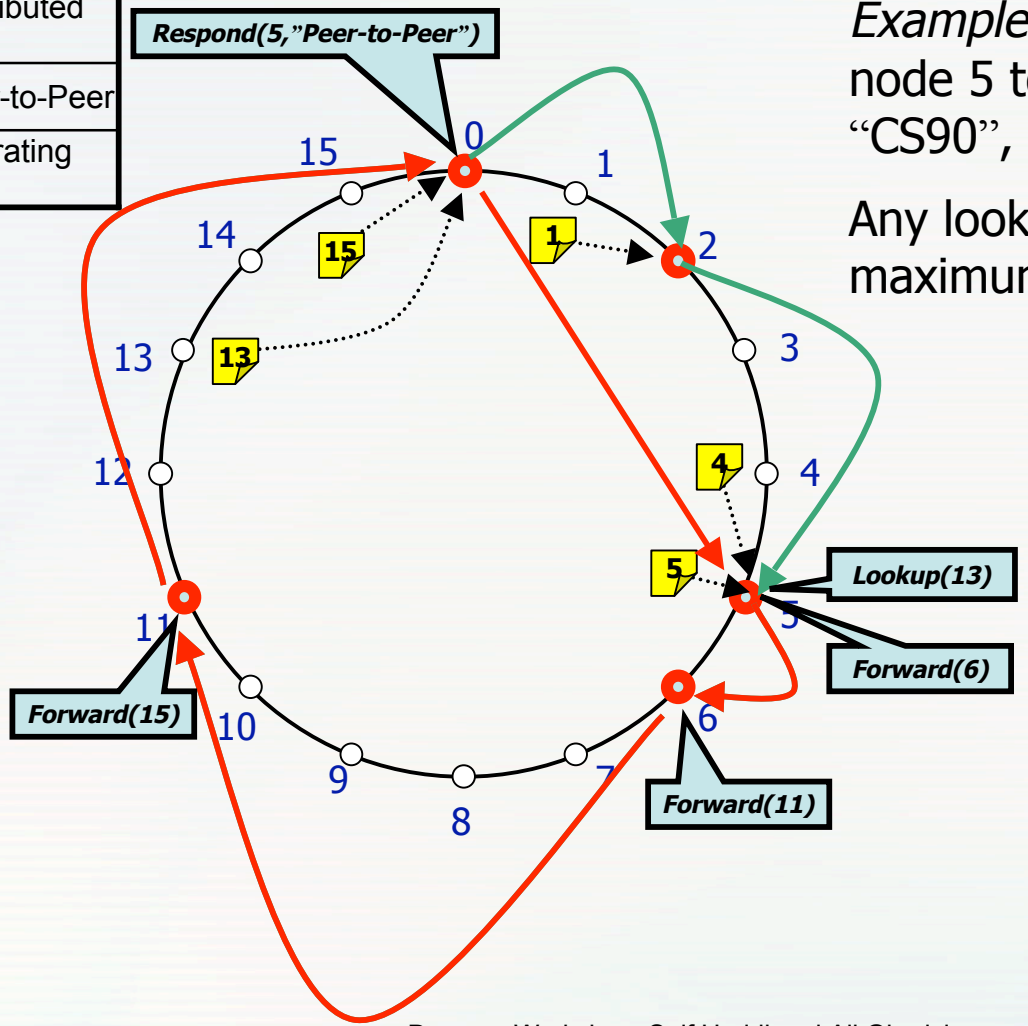
- **The successor of a node n is the first node met going in clockwise direction starting at $n+1$**

Successor of Node 0 → Node 2
Successor of Node 2 → Node 5
Successor of Node 5 → Node 6
Successor of Node 6 → Node 11
Successor of Node 11 → Node 0

Simple lookup Example

Key	Value
1 (CS10)	Algorithms
4 (CS15)	Networking
5 (CS30)	Distributed S.
13 (CS90)	Peer-to-Peer
15 (CS95)	Operating Sys.

- **Lookups can be resolved in by following the successors sequentially**



Example: A lookup is made at node 5 to get the value of key "CS90", identifier H("CS90")=13

Any lookup can be resolved in maximum O(*N*) hops. ***Too slow!***

Outlook

- DHT Introduction
  - What is a DHT
  - Chord: How to partition the data
  - Chord: How to interconnect nodes
  - Chord: How to speed up search
  - Chord: How to maintain pointers
- DKS Algorithms
  - K-ary Search
  - Atomic Join and Leaves
  - Broadcast Algorithms
  - Multicasting
  - Bulk Operations
  - Replication
  - Topology Maintenance

Speeding up lookups

- Each node, n, not only points to its successor but to the successors of

  $n+2^1, n+2^2, n+2^3, \ldots, n+2^L$ (all arithmetic operations modulo N)

- At each step in the routing, the distance between the currrent node and destination is halved (worst case).
- Yields $O(\log_2 N)$ hops at worst
  - N being the number of nodes

Example: node n=15

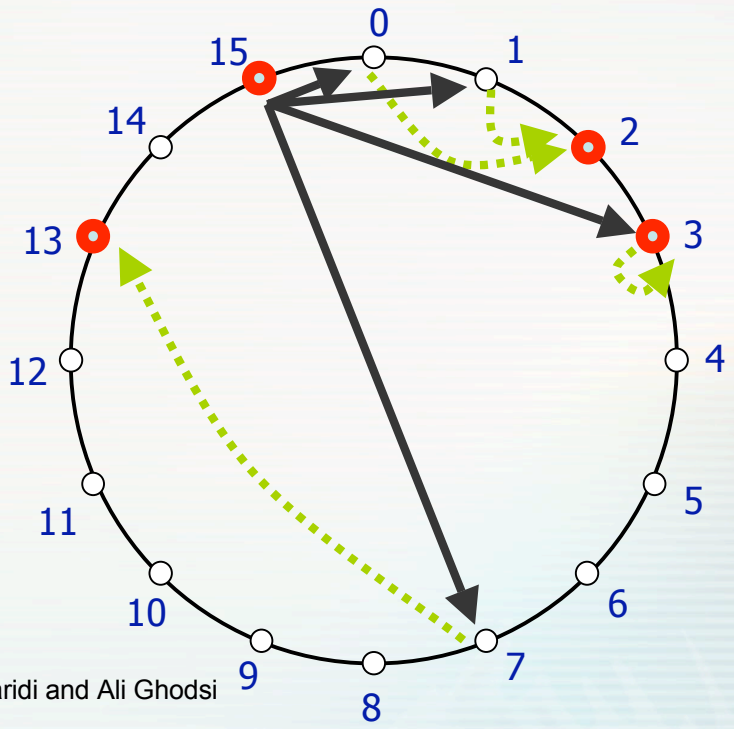
Start of the Intervals

Successor of $15+2^0=0$ → Node 2

Successor of $15+2^1=1$ → Node 2

Successor of $15+2^2=3$ → Node 3

Successor of $15+2^3=7$ → Node 13



Outlook

- DHT Introduction
  - What is a DHT
  - Chord: How to partition the data
  - Chord: How to interconnect nodes
  - Chord: How to speed up search
  - Chord: How to maintain pointers
- DKS Algorithms
  - K-ary Search
  - Atomic Join and Leaves
  - Broadcast Algorithms
  - Multicasting
  - Bulk Operations
  - Replication
  - Topology Maintenance

Checking for new joins and departures

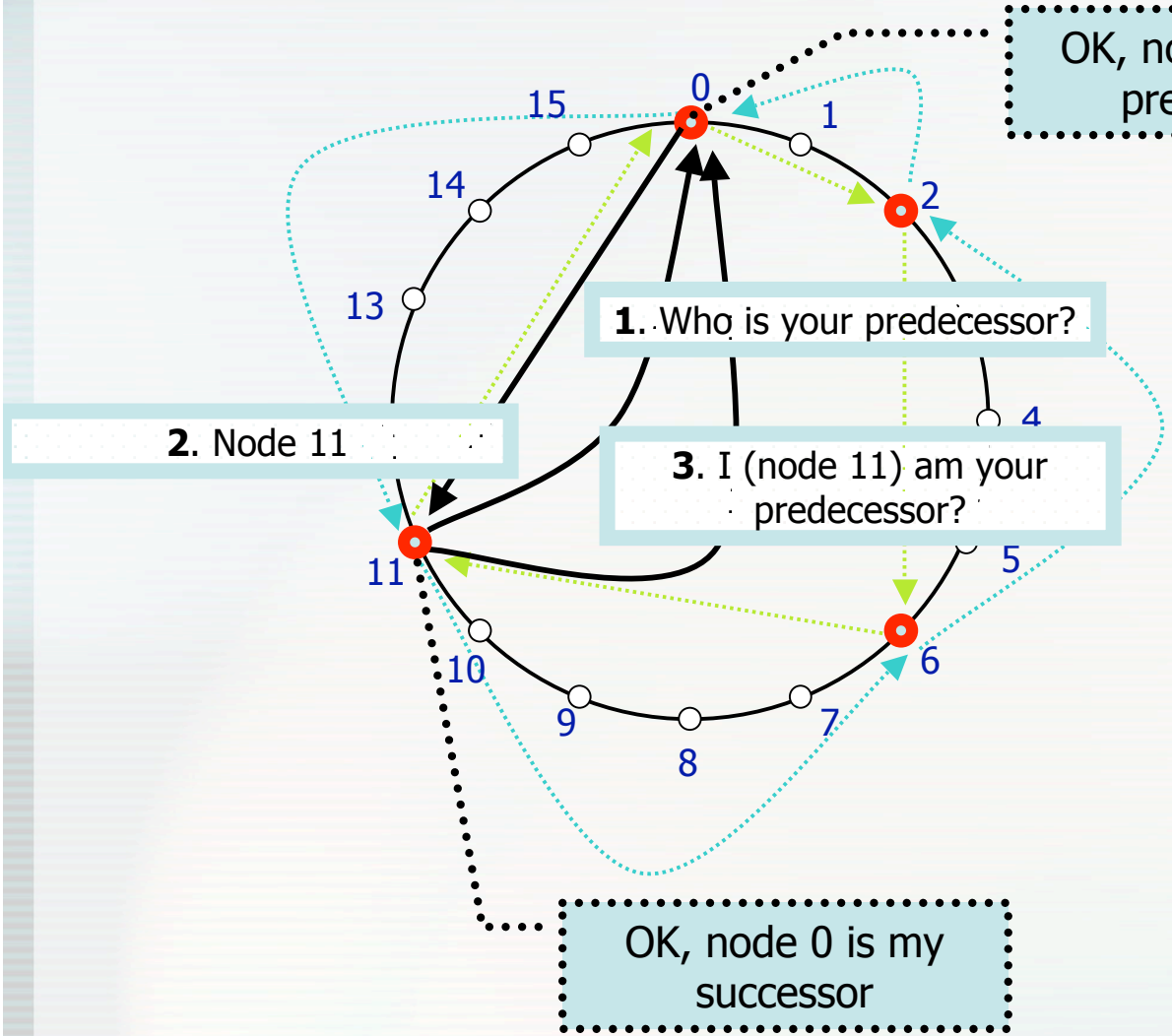
- **Each node periodically *stabilizes***

Algorithm 1 Chord's periodic stabilization protocol

```
1: procedure n.STABILIZE()
2:     p := succ.GetPredecessor()
3:     if p ∈ (n, succ) then
4:         succ := p
5:     end if
6:     succ.Notify(n)
7: end procedure

8: procedure n.GETPREDECESSOR()
9:     return pred
10: end procedure

11: procedure n.NOTIFY(p)
12:     if p ∈ (pred, n] then
13:         pred := p
14:     end if
15: end procedure
```

Checking for new joins and departures

- **Each node periodically *stabilizes***

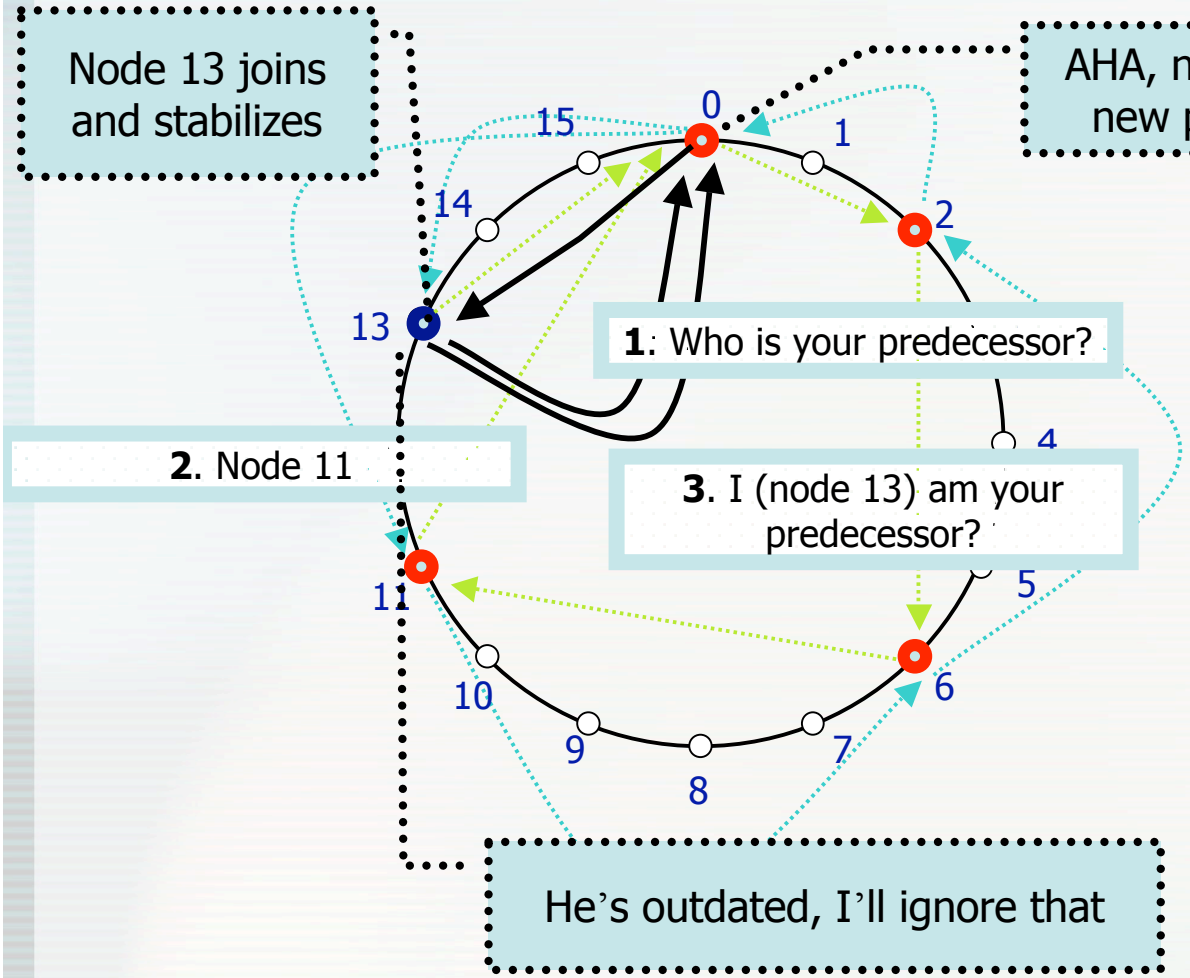


Stabilization Algorithm:

1. Ask successor "who is your predecessor"
2. Successor answers, if the answer is more "suitable" change my successor pointer
3. Tell the current successor about my identifier

Stabilization: join

- **A joining node, only notifies its successor about its existence**

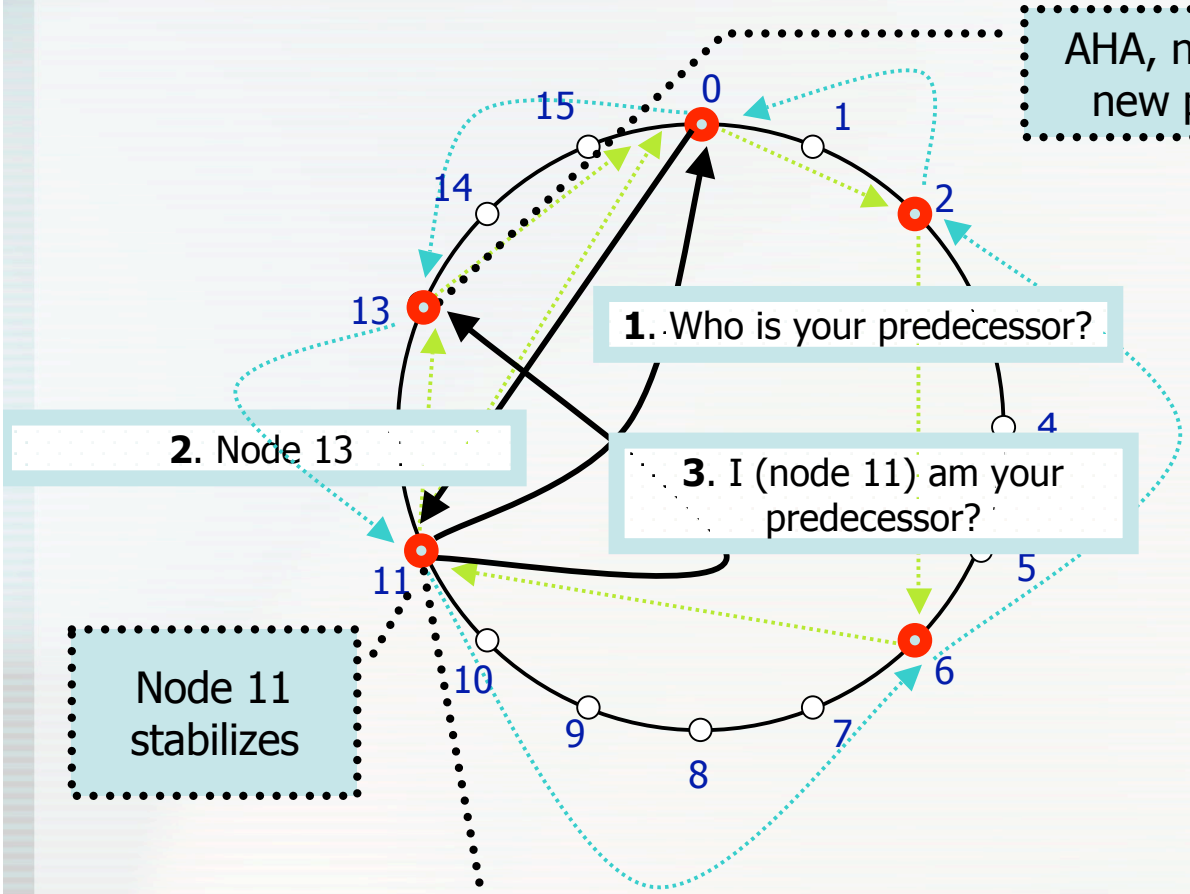


Stabilization Algorithm:

1. Ask successor "who is your predecessor"
2. Successor answers, if the answer is more "suitable" change my successor pointer
3. Tell the current successor about my identifier

Stabilization: join

- **A joining node, only notifies its successor about its existence**

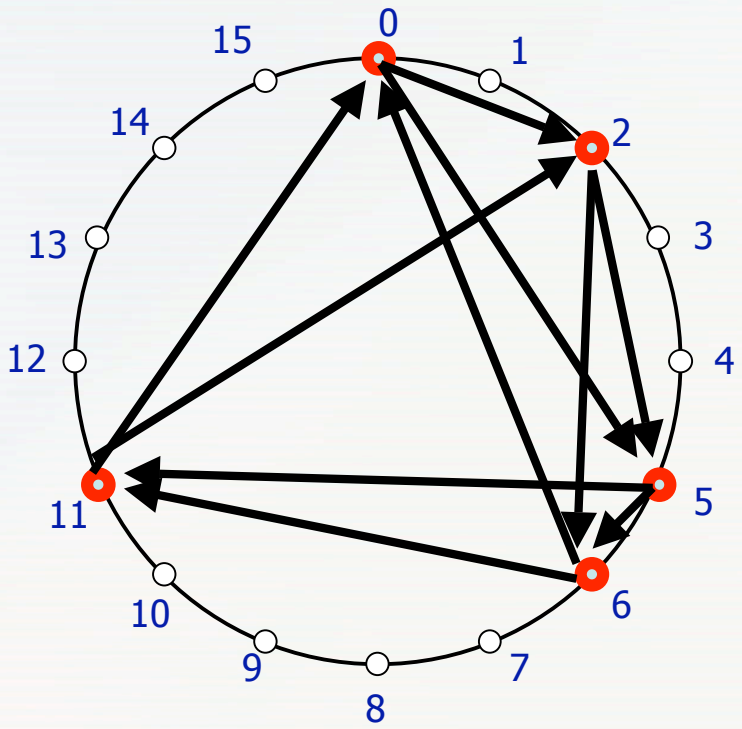


Stabilization Algorithm:

1. Ask successor "who is your predecessor"
2. Successor answers, if the answer is more "suitable" change my successor pointer
3. Tell the current successor about my identifier

Handling dynamism: successor-lists

- **Every node in the system maintains additional routing pointers to its *f* successors**



Example: *f*=2

Every node knows its *f*=2 successors

Node **0** : Node 2 and Node 5

Node **2** : Node 5 and Node 6

Node **5** : Node 6 and Node 11

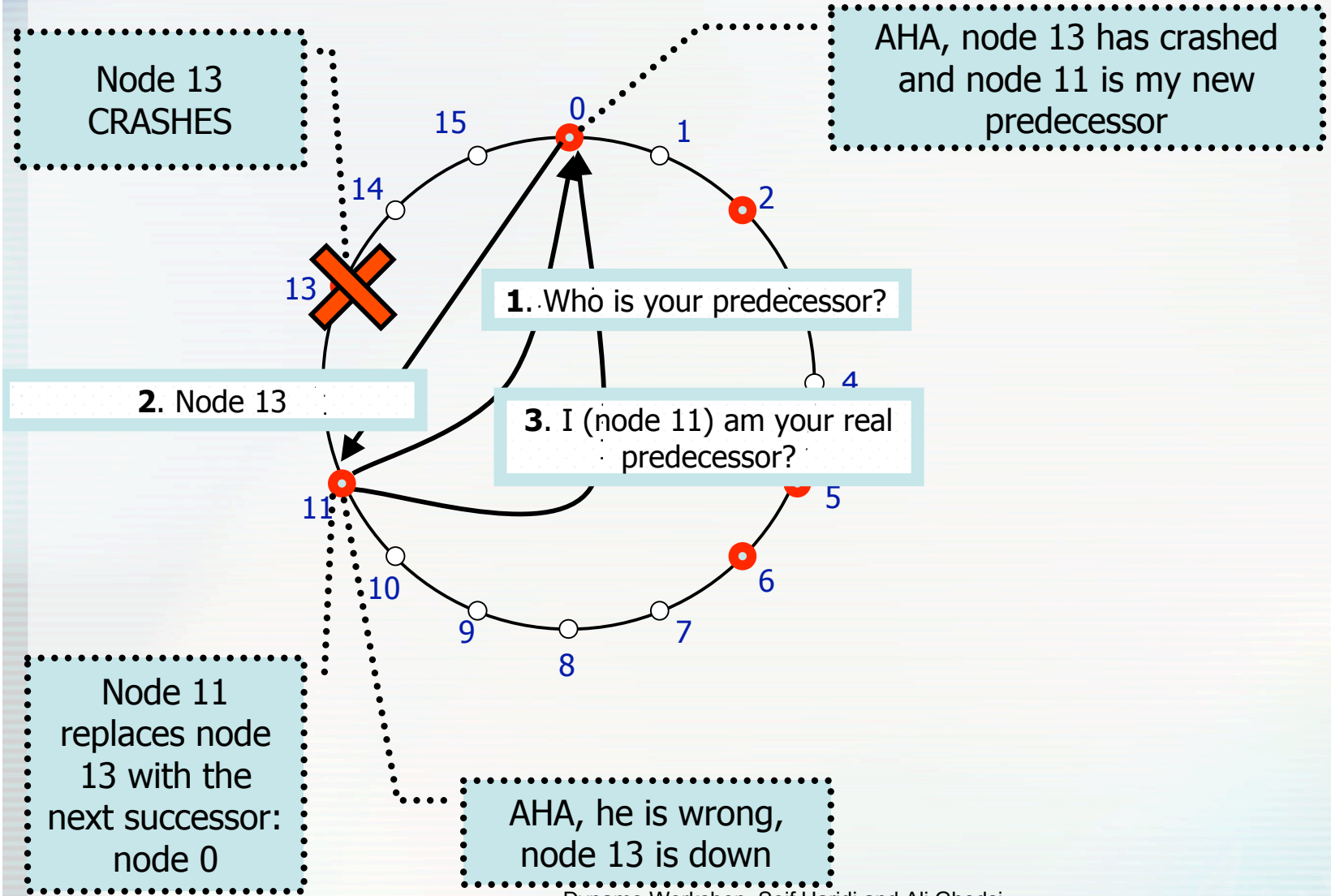
Node **6** : Node 11 and Node 0

Node **11** : Node 0 and Node 2

- **If node *n* detect that its successor has failed, it replaces it with the first alive successor node it knows**

Stabilization: failure

- **A joining node, only notifies its successor about its existence**



DHT Summary

- **Completely decentralized**
- **Self-organizes as nodes join, leave, and fail**
- **Maximum $\log_2(N)$ number of hops to find items, where N is the number of hops**

- **Each node only stores a small amount of items, on average D/N (D is the number of items)**
- **Each node only maintains a small amount of routing information $\log_2(N)$**
- **Each join/leave/failure event requires D/N items to be reshuffled**
- **Each join/leave/failure event requires $(\log_2(N))^2$ messages to restore the routing state**

Outlook

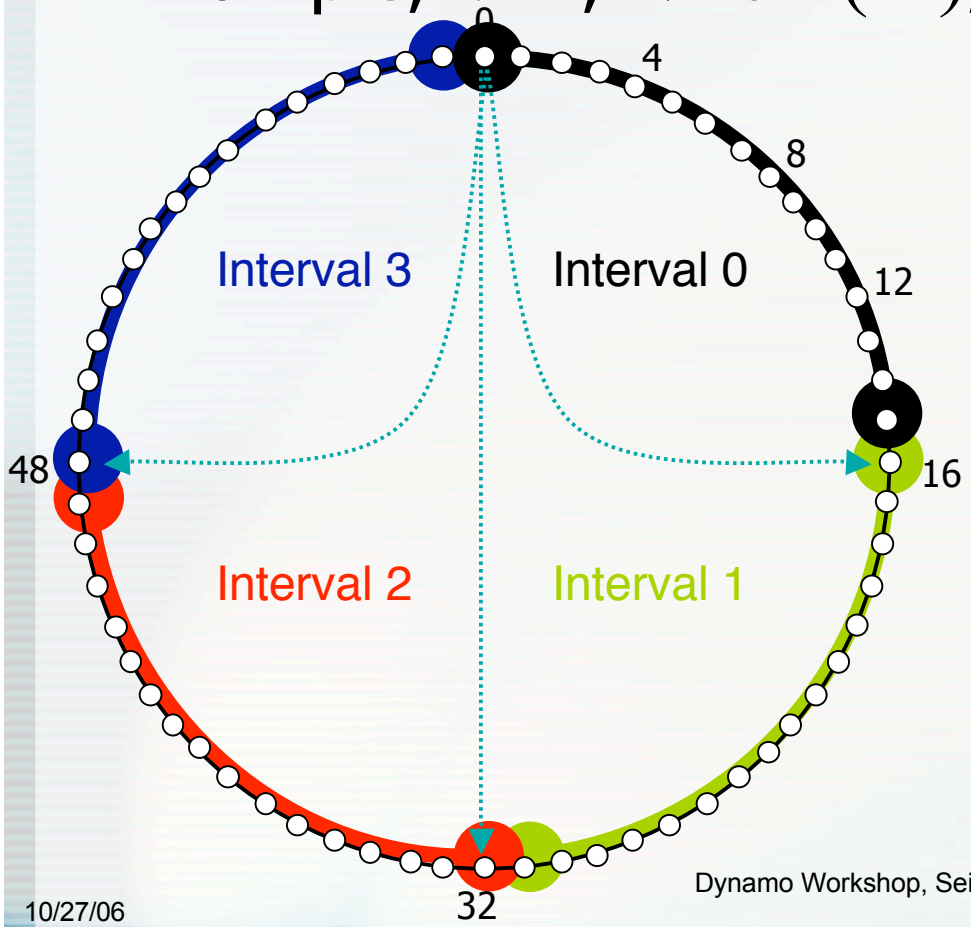
- DHT Introduction
  - What is a DHT
  - Chord: Distributing a hash table
  - Chord: How to partition the data
  - Chord: How to interconnect nodes
  - Chord: How to speed up search
  - Chord: How to maintain pointers
- DKS Algorithms
  - K-ary Search
  - Atomic Join and Leaves
  - Broadcast Algorithms
  - Multicasting
  - Bulk Operations
  - Replication
  - Topology Maintenance

K-ary Search: intuition

- Goal:
  - At most $\log_k(N)$ hops per lookup
    - k being a configurable parameter
    - N being the number of nodes
  - Instead of only $\log_2(N)$

Achieving log_kN lookup effiency

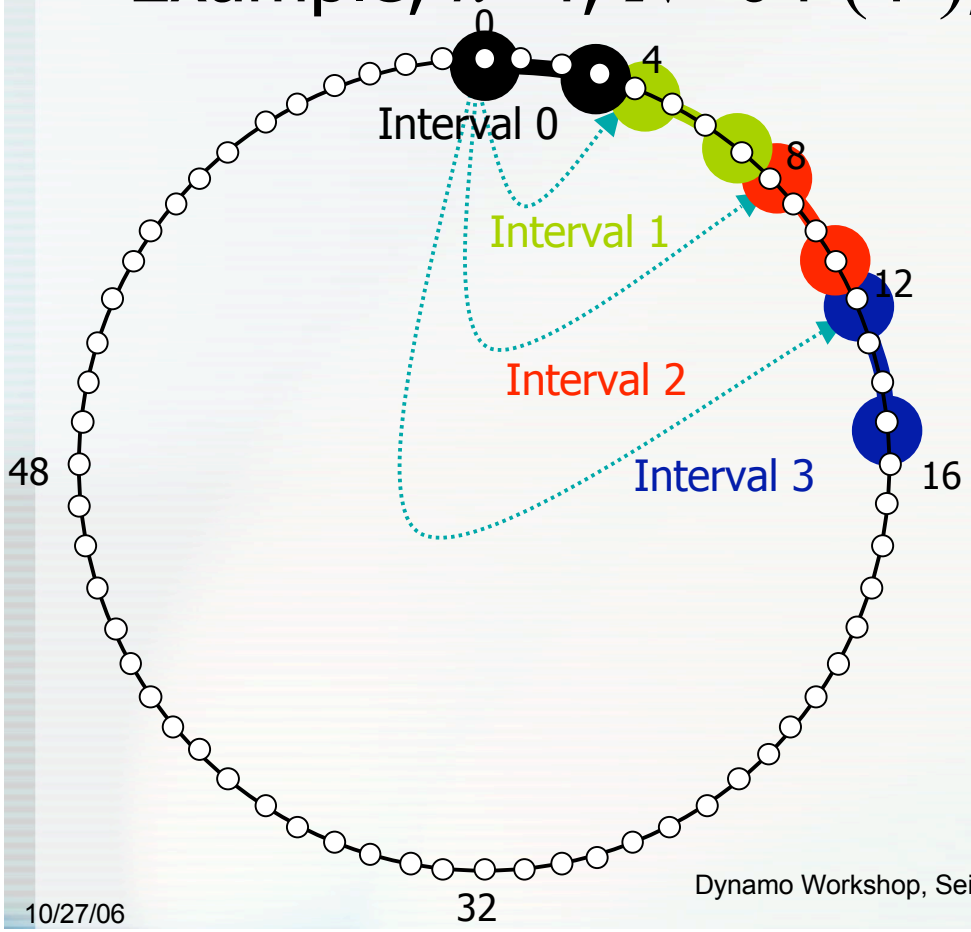
- Each DKS node maintains $\log_k N$ *levels*, and each level contains k *intervals* with pointers to the first node encountered in the interval
- Example, k=4, N=64 (4^3), node 0



- level 1, 4 intervals

Achieving log_kN lookup effiency

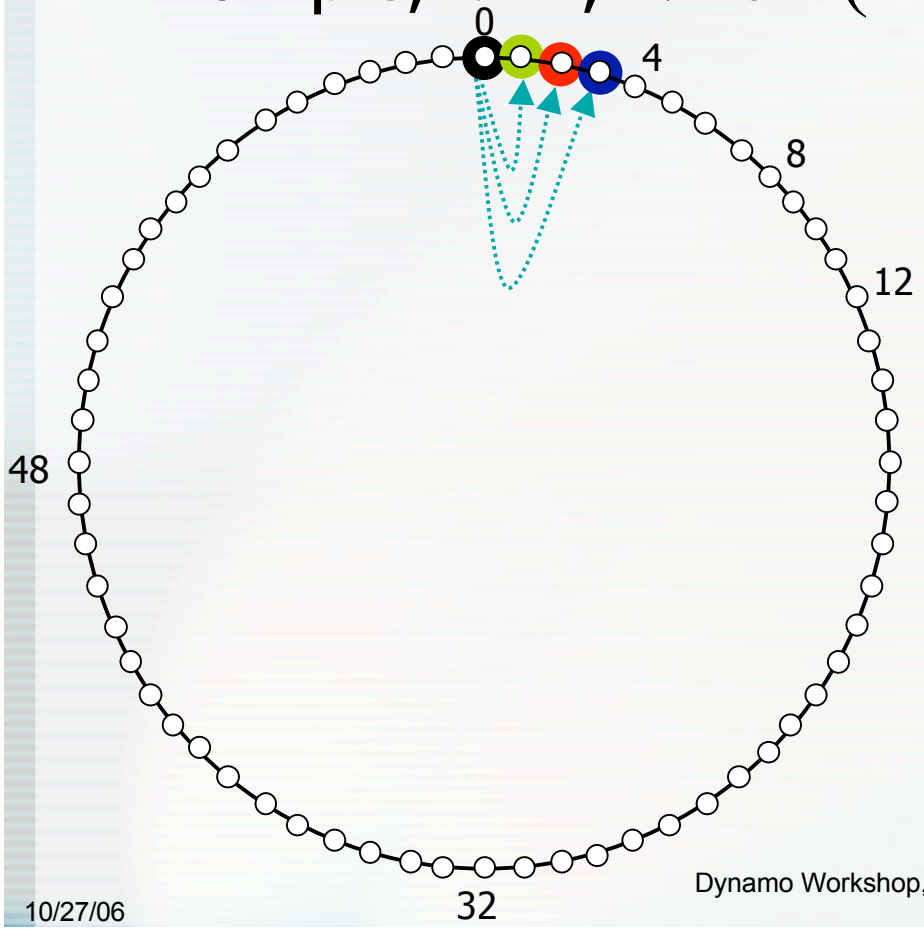
- Each DKS node maintains $\log_k N$ *levels*, and each level contains k *intervals* with pointers to the first node encountered in the interval
- Example, k=4, N=64 (4^3), node 0



- level 1, 4 intervals
- level 2, 4 intervals

Achieving log_kN lookup effiency

- Each DKS node maintains $\log_k N$ *levels*, and each level contains k *intervals* with pointers to the first node encountered in the interval
- Example, k=4, N=64 (4^3), node 0



- level 1, 4 intervals
- level 2, 4 intervals
- level 3, 4 intervals

Arity can matter

- **Maximum number of hops can be configured**

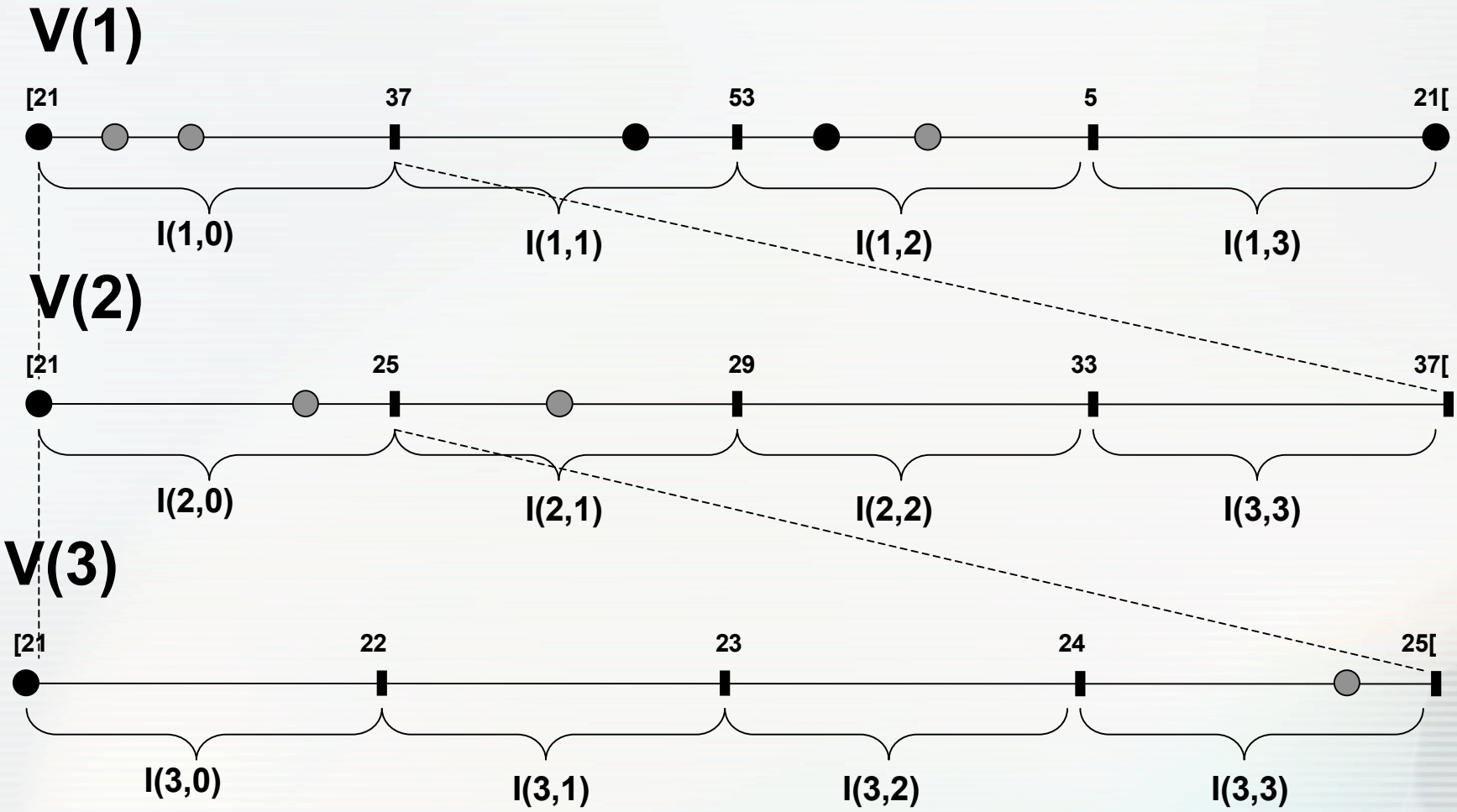
$$k = N^{\frac{1}{r}}$$

$$\log_k(N) = \log_{N^{\frac{1}{r}}}(N) = \log_{N^{\frac{1}{r}}}\left(\left(N^{\frac{1}{r}}\right)^r\right) = r$$

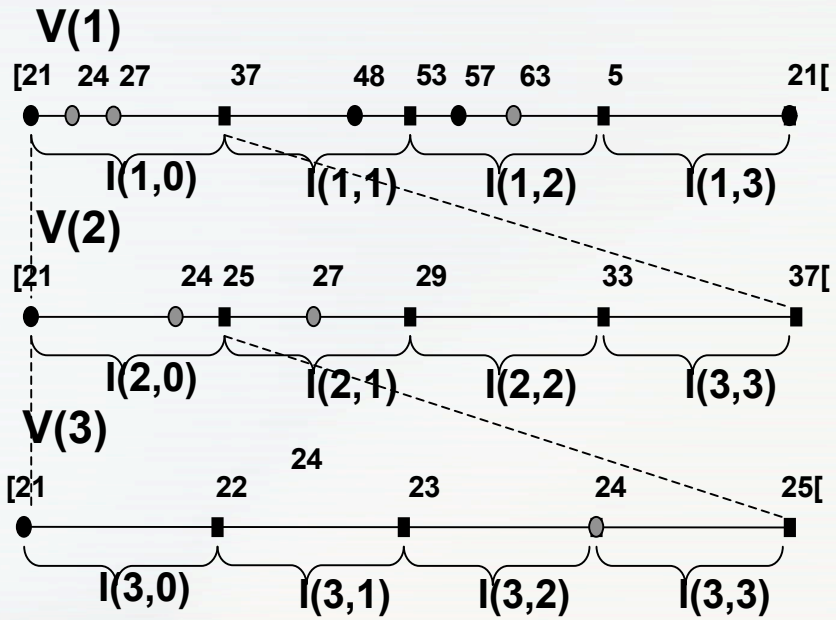
Routing table

- For the desired based k, the identifier space N is a power of k
  - $N = k^L$
- $L = \log_k N$ is the number of levels
- RT is of size (k-1)L
- Views
- Level 1: V(1) = [n, n+N)
- Level 2: V(2) = [n, (n+N)/k)
- Level ℓ : $V(l) = [n, (n+N)/k^{\ell})$
- Intervals: at any level $\ell \in$ _, $V(\ell)$ is partitioned into k intervals $I(\ell, i)$, for $0 \le i \le k-1$

Routing Table for node 21
Views, Levels (k = 4)



Routing Table for node 21 Views, Levels (k = 4)



(21,21)	(37,48)	(53,57)	(5,21)
(21,21)	(25,27)	(29,48)	(33,48)
(21,21)	(22,24)	(23,24)	(24,24)

$R(\_,i) = \langle n + i(N/k_{\_}), succ(n + i(N/k_{\_})) \rangle$

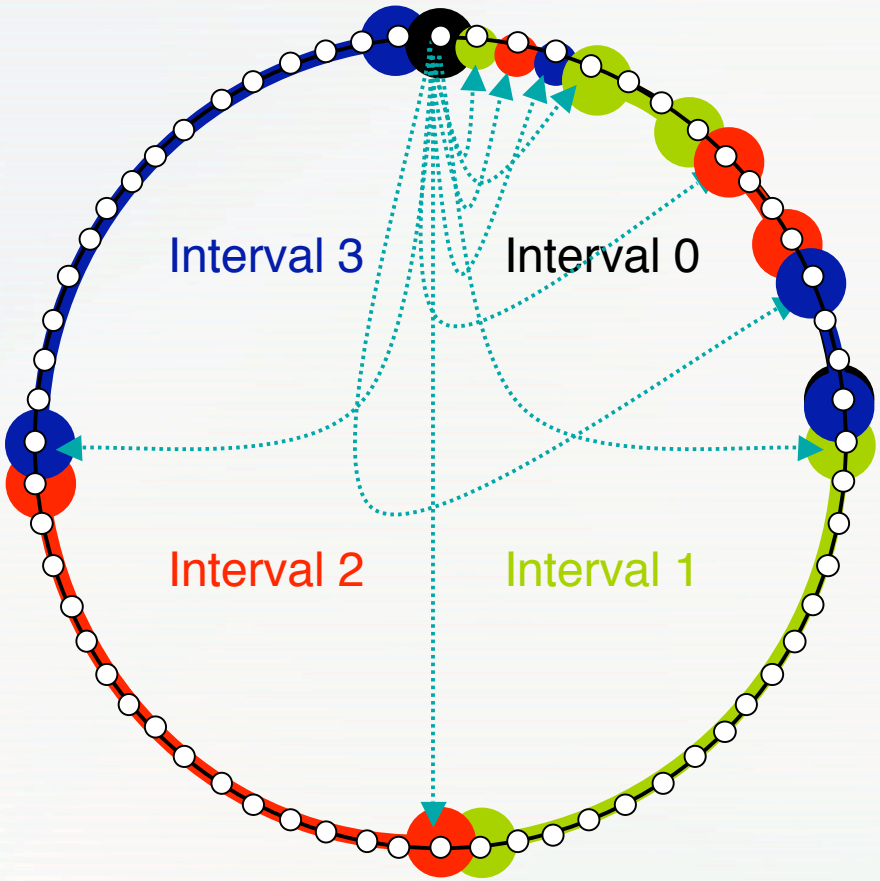
$R(\_,i).start = n + i(N/k^{\ell})$

$R(\_,i).node = succ(n + i(N/k_{\_}))$ or any other node in the interval

Routing

- **The routing table can be organized as a monotonically increasing set of pointers**
- **Each pointer refers to a node in corresponding interval [f(i), f(i+1))**
- **Start of each interval i, $1 \le i \le (k-1)\log_k N$:**
  - **$f(i) = n + (1 + (i-1) \bmod (k-1)\ )\ k^{\lfloor (i-1)/(k-1) \rfloor}$**

Monotonically Increasing Pointers



Simple Recursive Routing

Algorithm 11 Recursive Lookup Algorithm

```
1: procedure n.LOOKUP(i, OP)
2:     if TERMINATE(i) then
3:         p :=NEXT_HOP(i)
4:         res := p.OP()                    ▷ OP could carry parameters
5:         return res
6:     else
7:         m :=NEXT_HOP(i)
8:         return m.LOOKUP(i)
9:     end if
10: end procedure
```

Greedy Routing

Algorithm 14 Greedy Lookup

```
1: procedure n.TERMINATE(i)
2:     return i ∈ (n, succ]
3: end procedure

1: procedure n.NEXT_HOP(i)
2:     if TERMINATE(i) then
3:         return succ
4:     else
5:         r := succ
6:         for j := 1 to K do                ▷ Node has K pointers
7:             if rt(j) ∈ (n, i) then
8:                 r := rt(j)
9:             end if
10:        end for
11:        return r
12:    end if
13: end procedure
```

Topology Maintenance

- **Pointers in the routing table are monitored by an inaccurate failure detectors, and reset (nil) if a node is suspected**
- **Failed pointers leads to a lookup to the start of the corresponding entry**
- **The result is the successor and its successor list**
- **A suitable node is installed**

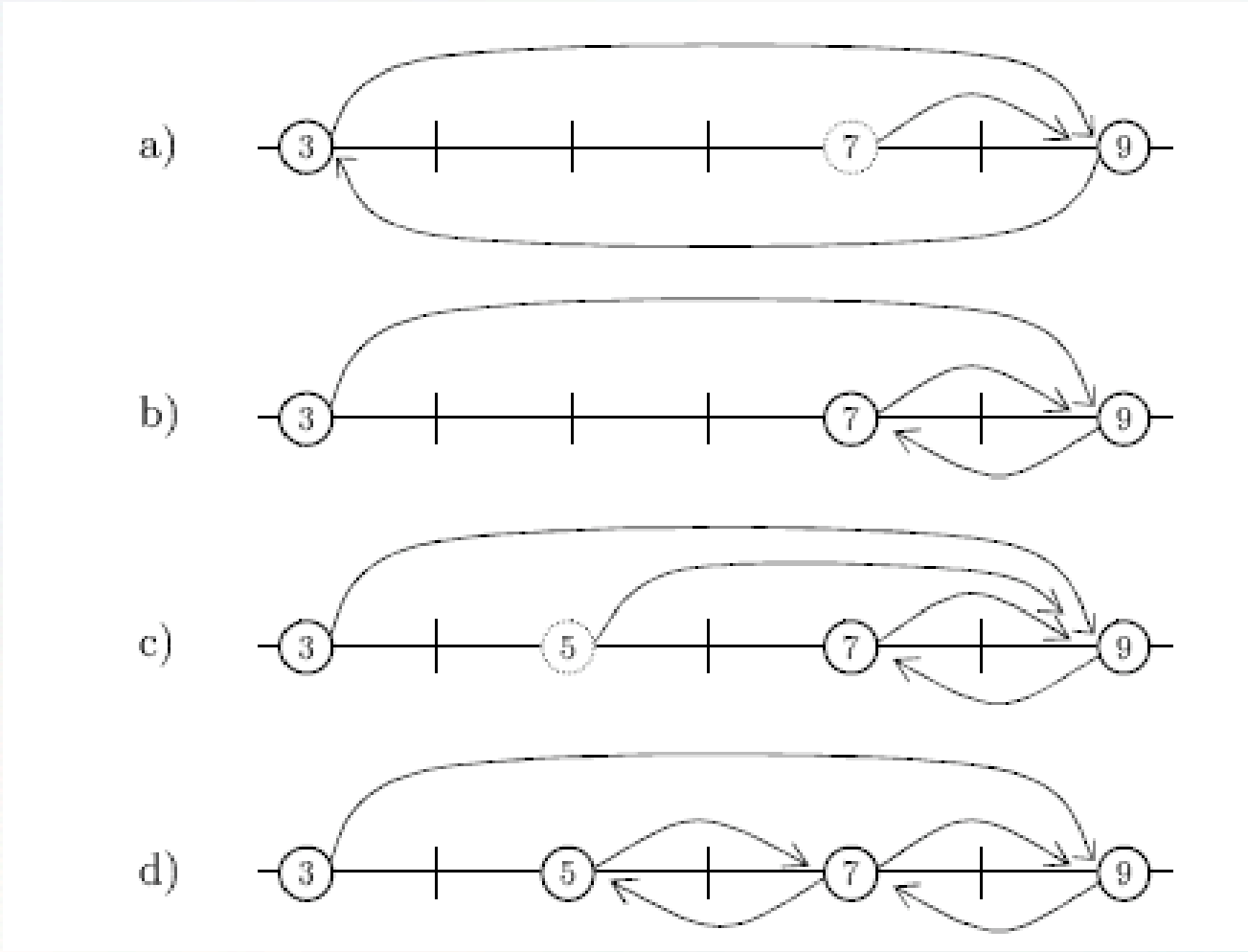
Outlook

- DHT Introduction
  - What is a DHT
  - Chord: How to partition the data
  - Chord: How to interconnect nodes
  - Chord: How to speed up search
  - Chord: How to maintain pointers
- DKS Algorithms
  - K-ary Search
  - Atomic Join and Leaves
  - Broadcast Algorithms
  - Multicasting
  - Bulk Operations
  - Replication
  - Topology Maintenance

Atomic Join and Leave

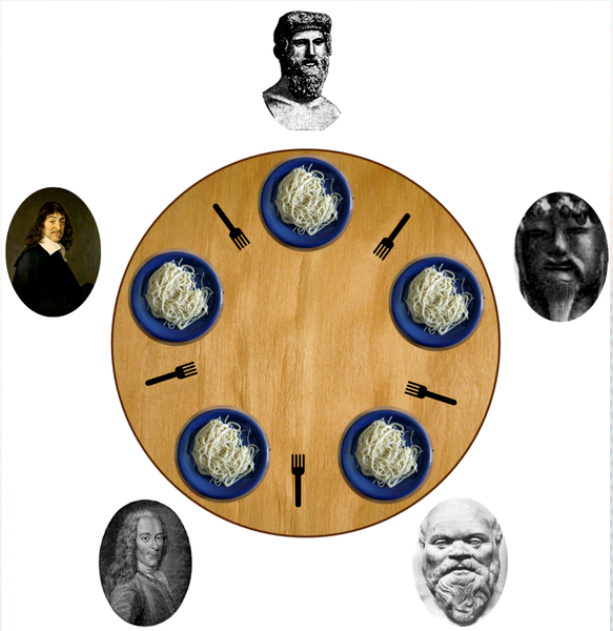
- **Goal**
  - Key or Lookup Consistency
- **Informally**
  - At any time, at most one node responsible for any key
  - Joins/leaves should “not affect” functionality of lookups/inserts/updates/deletes

Lookup consistency is not guaranteed in traditional DHTs: lookup(4)



Locking to achieve atomicity

- Problem partly reduced to dining philosophers
  - Each node has a *fork* and a *lock queue*
  - Current node's fork and successor's fork acquired before modifying the ring
  - Avoiding deadlocks
    - Asymmetric Locking: one node acquires locks in reverse order
    - Probabilistic Locking: preempt locks



Locking to achieve atomicity

- Pitfalls:
  - Join adds a "philosopher"
    - Solution: some requests in the lock queue forwarded to to new node
  - Leave removes a "philosopher"
    - Problem:

      if leaving node gives its lock queue to its successor, some nodes will get a worse position in queue

      starvation!

Correctness Properties

- Safety
  - Deadlock freedom
  - Termination (trivial)
- Liveness
  - Livelock freedom (always some progress)
  - Starvation freedom (every node makes progress)

Looking Consistency with Locking

- Lookup Consistency with Joins
  - Successor forwards requests to new node
- Lookup Consistency with Leaves
  - Leaving node forwards requests to successor
- **Proving Lookup Consistency**
  - Configuration is the pointers of all nodes
  - Show: any reachable configuration, only one node responsible for a key, starting lookup anywhere

Outlook

- DHT Introduction
  - What is a DHT
  - Chord: How to partition the data
  - Chord: How to interconnect nodes
  - Chord: How to speed up search
  - Chord: How to maintain pointers
- DKS Algorithms
  - K-ary Search
  - Atomic Join and Leaves
  - Broadcast Algorithms
  - Multicasting
  - Bulk Operations
  - Replication
  - Topology Maintenance

Broadcasting

- Broadcast to all nodes in the DHT
- Fast dissemination
  - Broadcasting proceeds in parallel
  - Time complexity *O(log N)* , N is number of nodes
- Efficient
  - No redundancy, full coverage
  - *O(N) message complexity*, N is number of nodes

Correctness of Broadcast

- Safety
  - **Non-redundancy:** A node never receives the same message more than once
- Liveness
  - **Termination:** every broadcast eventually terminates
  - **Coverage:** every node eventually gets the message

Simple Best-Effort Broadcast

Algorithm 17 Simple Broadcast Algorithm

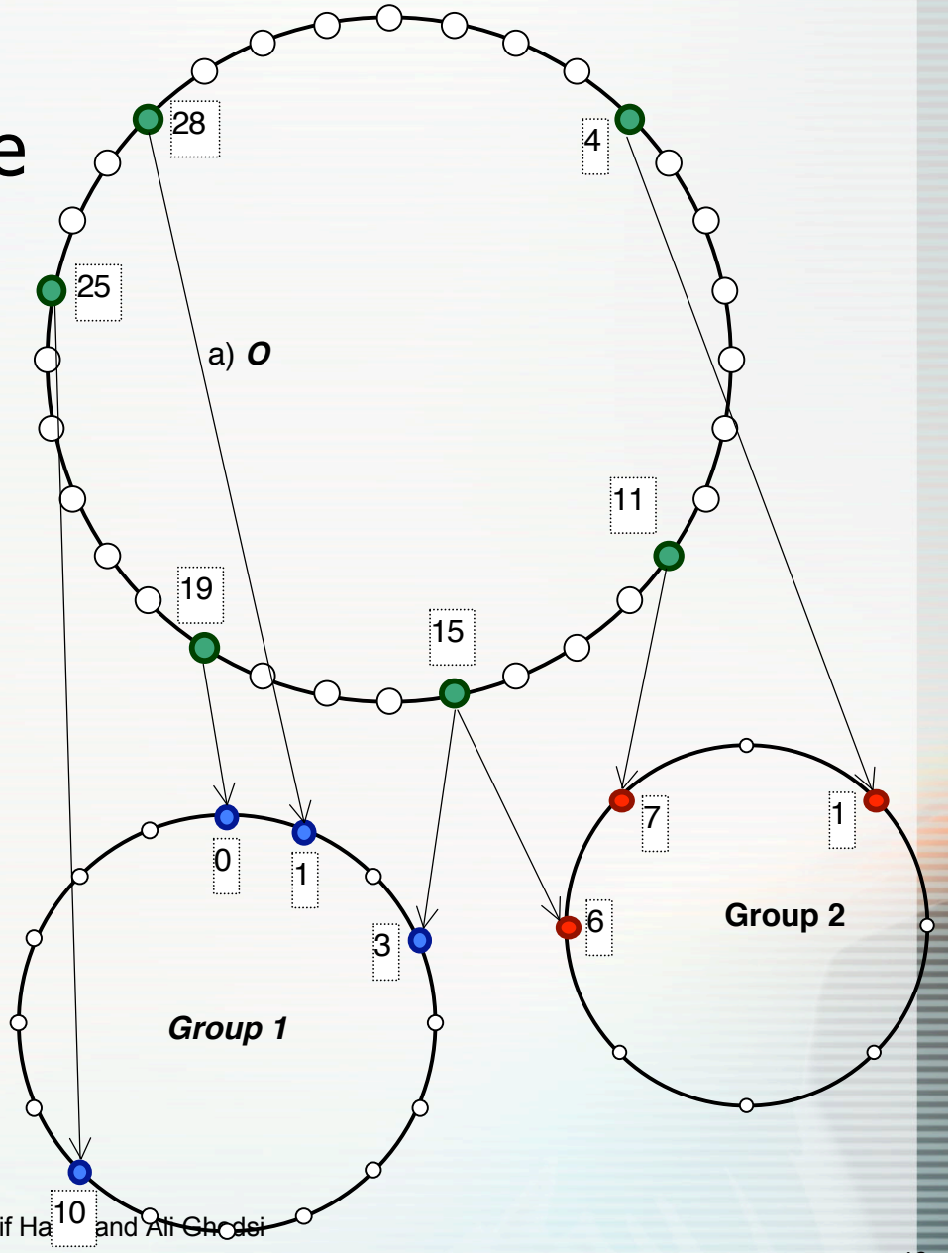
```
1: event n.STARTSIMPLEBCAST(msg) from m
2:     sendto n.SIMPLEBCAST(msg, n)              ▷ Local message to itself
3: end event

1: event n.SIMPLEBCAST(msg, limit) from m
2:     Deliver(msg)                              ▷ Deliver msg to application
3:     for i := M downto 1 do                    ▷ Node has M unique pointers
4:         if u(i) ∈ (n, limit) then
5:             sendto u(i).SIMPLEBCAST(msg, limit)
6:             limit := u(i)
7:         end if
8:     end for
9: end event
```

Outlook

- DHT Introduction
  - What is a DHT
  - Chord: How to partition the data
  - Chord: How to interconnect nodes
  - Chord: How to speed up search
  - Chord: How to maintain pointers
- DKS Algorithms
  - K-ary Search
  - Atomic Join and Leaves
  - Broadcast Algorithms
  - Multicasting
  - Bulk Operations
  - Replication
  - Topology Maintenance

Multicasting

- All nodes in the system are members of an instance of
- Group creation:
  - Create a *DKS* instance
  - Make it available in O
  - Joins the group
- Multicasting
  - Broadcasting within the group



Outlook

- DHT Introduction
  - What is a DHT
  - Chord: How to partition the data
  - Chord: How to interconnect nodes
  - Chord: How to speed up search
  - Chord: How to maintain pointers
- DKS Algorithms
  - K-ary Search
  - Atomic Join and Leaves
  - Broadcast Algorithms
  - Multicasting
  - Bulk Operations
  - Replication
  - Topology Maintenance

Bulk Operations: motivation

- Background
  - Assume building filesystem on-top of DHT
  - 4mb file of 4kb block -> 4000 blocks
- Making 4000 lookups expensive
  - Marshaling/unmarshaling (XML?)

Bulk Operations

- **Bulk(I)**
  - I is a set of identifiers
  - Reach every node with identifier in I
- **Bulk_Owner(I)**
  - I is a set of identifiers
  - Reach node responsible for every id in I
- **Bulk_Feedback(I)**
  - Same as BULK, but gets feedback through the virtual dissemination tree
- **Bulk_Owner_Feedback(I)**
  - Same as BULK, but gets feedback through the virtual dissemination tree
- No redundant messages sent
- Max log(n) messages per node

Bulk Properties

- Extreme Case 1
  - I is all identifiers
  - N messages to reach N nodes
  - Completed in log(N) time
  - Identical to broadcast

- Extreme Case 2
  - I is a singleton with one identifier
  - log(n) messages to perform lookup
  - Completed in log(N) time
  - Identical to lookup

Bulk Applications

- Bulk lookup
  - Fetch all values associated with keys {x1, x2,…}
- Pseudo reliable broadcast
  - Broadcast with feedback, use time outs to avoid hanging, retransmit message with bulk to those not covered
- Range queries
  - Cover all nodes in range]i, j]
- Replication
  - Replicate item x to following responsible nodes {x1,x2,…}
  - Symmetric Replication
- Topology maintenance
  - Update all nodes with identifiers in range {x1,x2,…}
  - Correction-on-change

Outlook

- DHT Introduction
  - What is a DHT
  - Chord: How to partition the data
  - Chord: How to interconnect nodes
  - Chord: How to speed up search
  - Chord: How to maintain pointers
- DKS Algorithms
  - K-ary Search
  - Atomic Join and Leaves
  - Broadcast Algorithms
  - Multicasting
  - Bulk Operations
  - Replication
  - Topology Maintenance

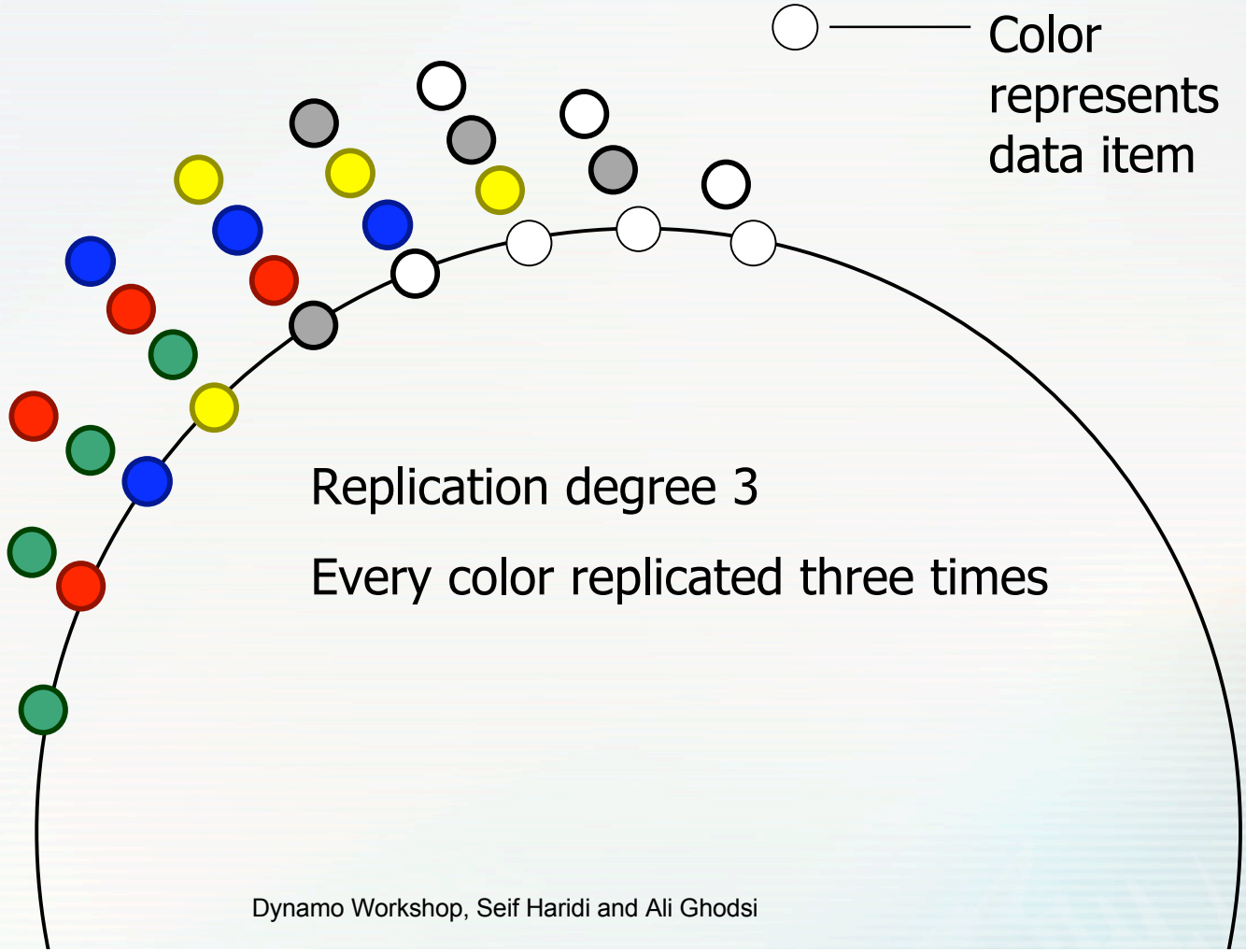
Background of Symmetric Replication

- DKS used to use Successor-lists, like Chord, Pastry, Koorde etcetera.

- This was abandoned in favor of Symmetric Replication because …

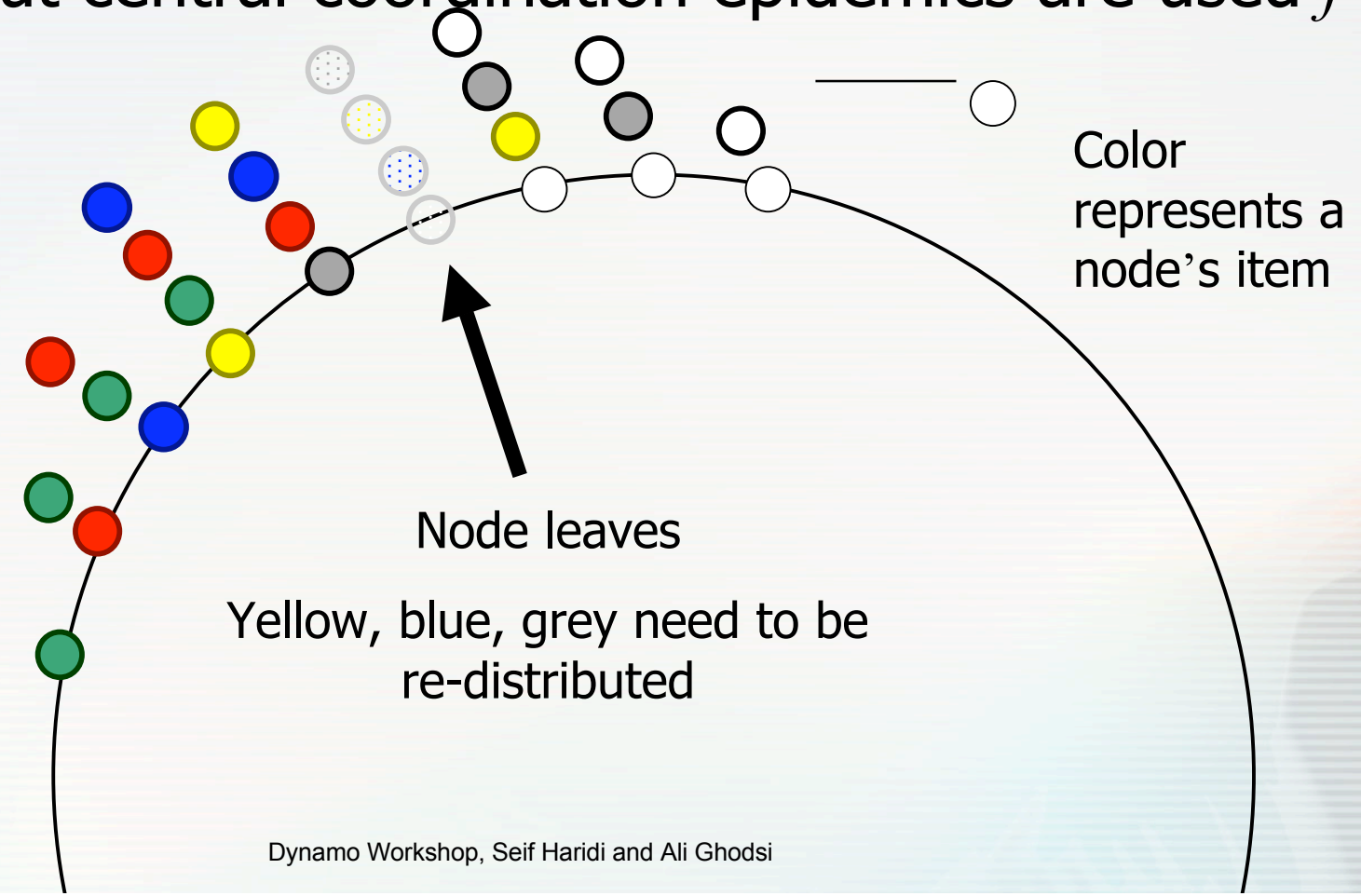
Successor-lists and Leaf sets

- If a node joins or leaves
  - f replicas need to be updated



Successor-lists and Leaf sets

- If a node joins or leaves
  - f replicas need to be updated
  - Without central coordination epidemics are used f^2



Symmetric Replication

Goal of Symmetric Replication:

- Simplicity!
- Enable concurrent requests
  - Do load-balancing
  - Increase robustness (high failure rates)
- Use with erasure codes
  - Given a k/n erasure used
  - Replicate n times
  - Fetch k replicas with bulk operations

Symmetric Replication

Idea:

- Partition the identifier space into m equivalence classes such that
- The cardinality of each class is f
  - m=N/f
- Each node replicates the equivalence class of every identifier it is responsible for

Symmetric replication

Replication degree *f*=4, *Id*={0,…,15}

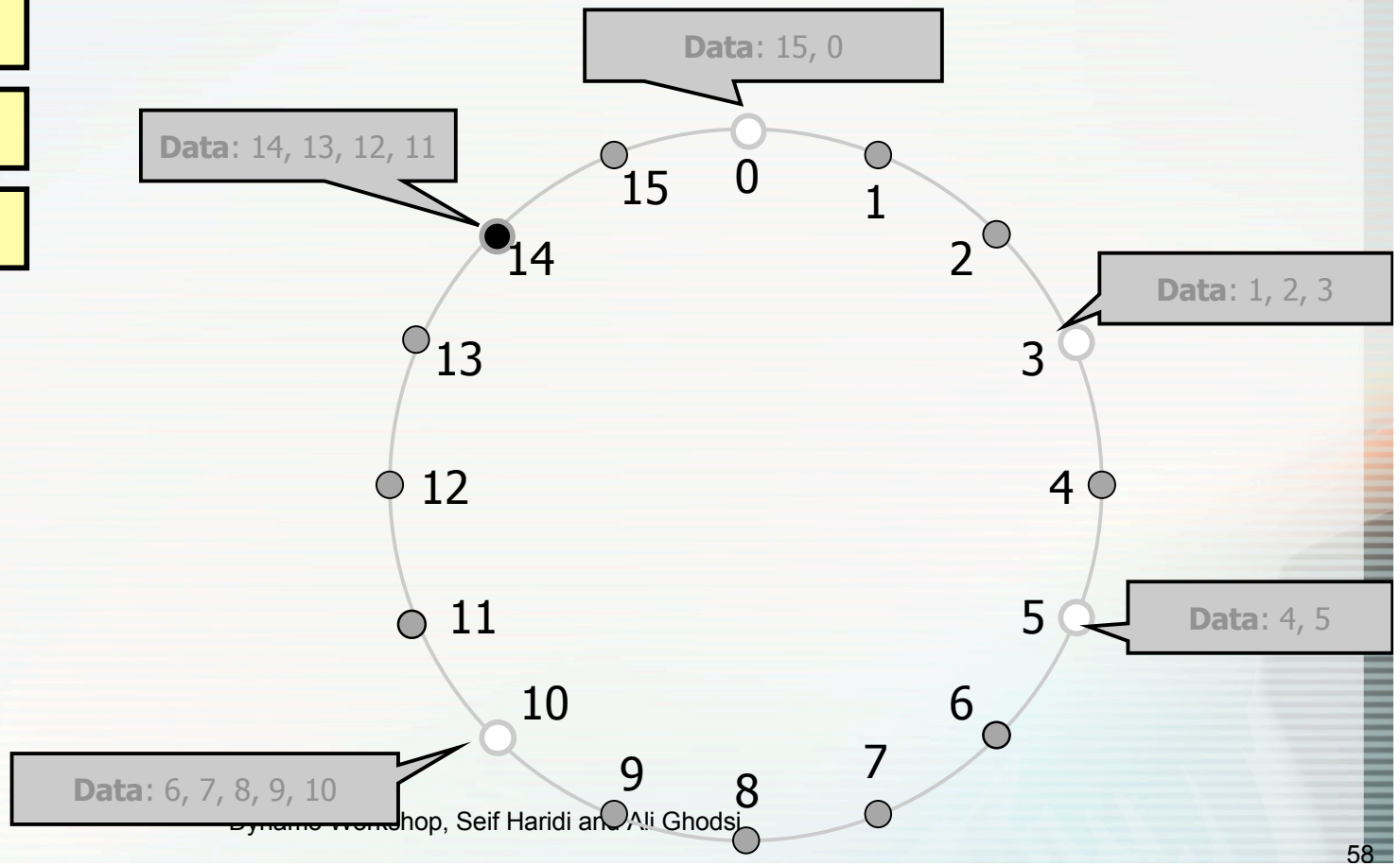
- **Congruence classes modulo 4:**

0, 4, 8, 12

1, 5, 9, 13

2, 6, 10, 14

3, 7, 11, 15



Symmetric replication

Replication degree *f*=4, *Id*={0,…,15}

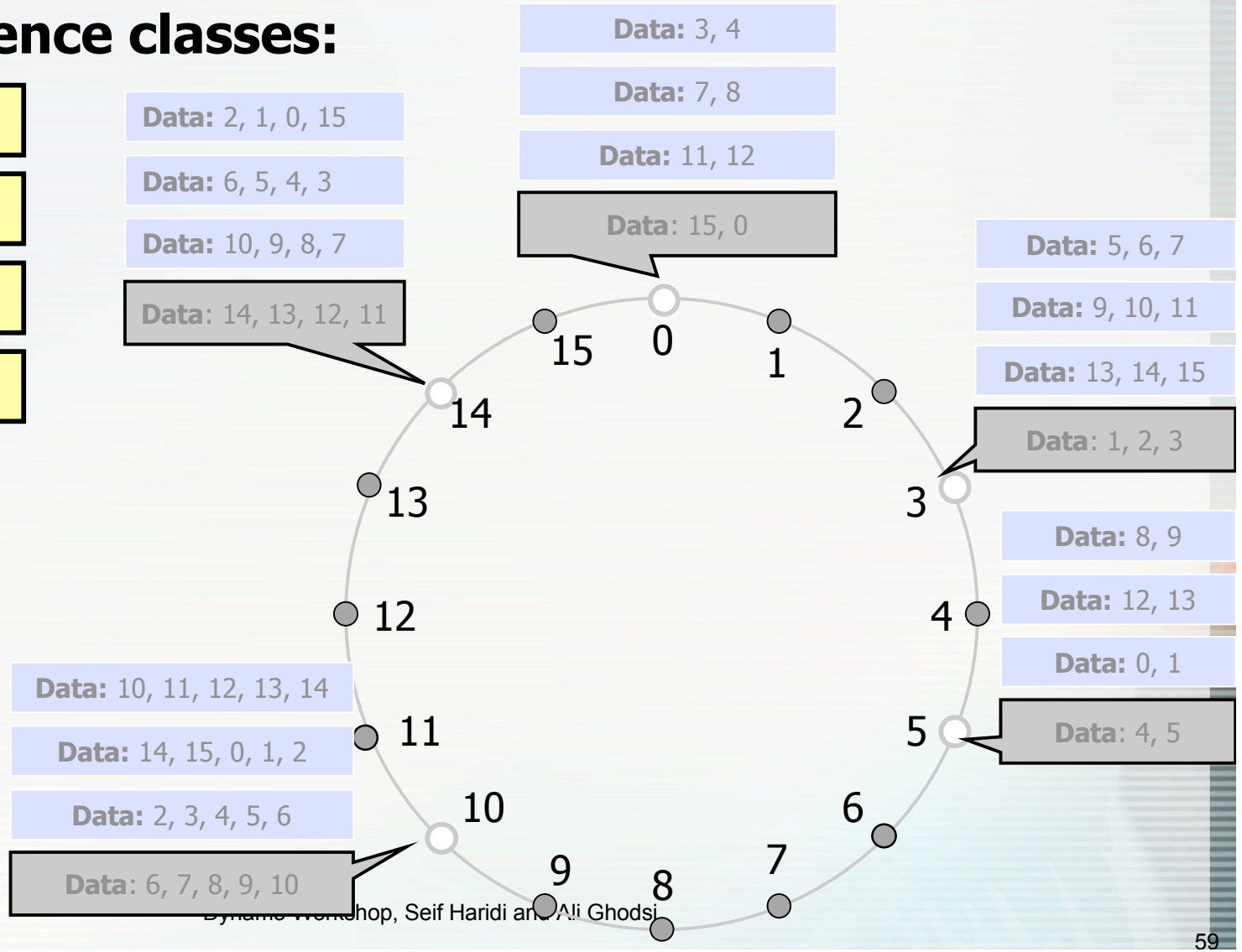
- **4 equivalence classes:**

0, 4, 8, 12

1, 5, 9, 13

2, 6, 10, 14

3, 7, 11, 15



Symmetric replication: joins

Replication degree *f*=4, *Id*={0,…,15}

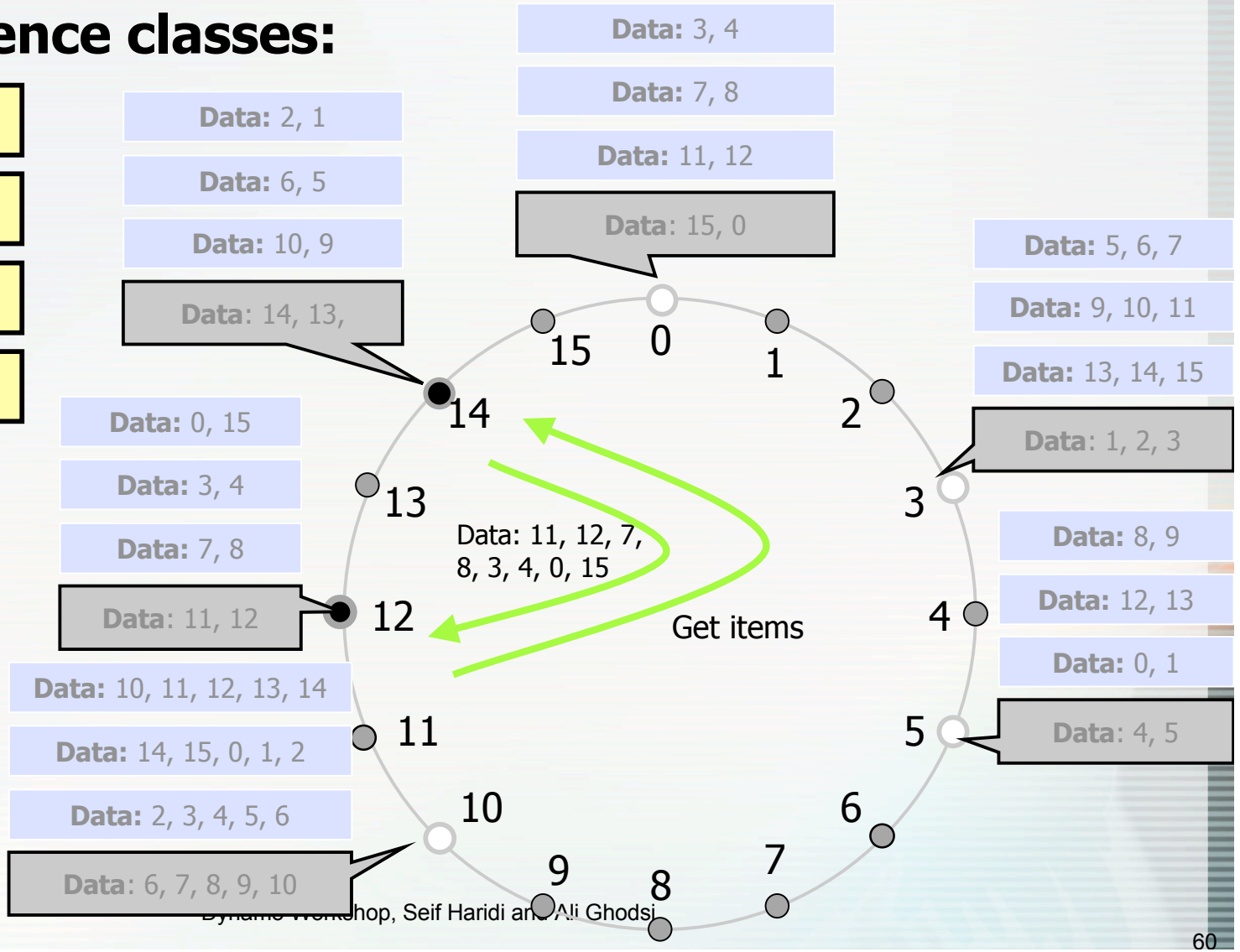
- **4 equivalence classes:**

0, 4, 8, 12

1, 5, 9, 13

2, 6, 10, 14

3, 7, 11, 15



Outlook

- DHT Introduction
  - What is a DHT
  - Chord: How to partition the data
  - Chord: How to interconnect nodes
  - Chord: How to speed up search
  - Chord: How to maintain pointers
- DKS Algorithms
  - K-ary Search
  - Atomic Join and Leaves
  - Broadcast Algorithms
  - Multicasting
  - Bulk Operations
  - Replication
  - Topology Maintenance

Applications ontop of DKS

- **MyriadStore**
  - **Distributed Backup**
  - **Use DKS for metadata storage**
- **Keso**
  - **Distributed File System**
- **DOH**
  - **Replicated Web Servers**
- **Delegent**
  - **Decentralized Trust Management Systems**

The End

- **Thank You!**

http://dks.sics.se

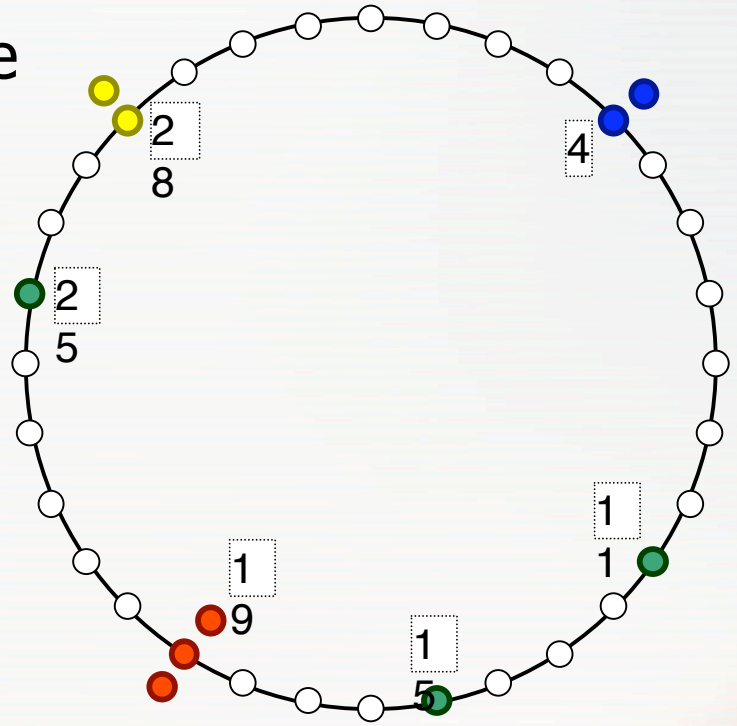
http://www.sics.se/~ali/dissert.pdf

Outlook

- DHT Introduction
  - What is a DHT
  - Chord: How to partition the data
  - Chord: How to interconnect nodes
  - Chord: How to speed up search
  - Chord: How to maintain pointers
- DKS Algorithms
  - K-ary Search
  - Atomic Join and Leaves
  - Broadcast Algorithms
  - Multicasting
  - Bulk Operations
  - Replication

IP multicast vs Overlay Multicast

- Idea
  - Utilize IP multicast where available
- Before joining
  - IP multicast to discover *local* nodes
  - Use same identifier as local nodes
  - Otherwise join as usual
- Whenever receive a message
  - IP multicast to all local nodes



Topology Maintenance

- Chord:
  - Extra pointers to *successor* of intervals

    $n+2^1, n+2^2, n+2^3, \ldots, n+2^L$

    (arithmetic modulo N)

- DKS:
  - Extra pointers to *predecessor* of intervals
  - Ring kept as Chord
  - Responsibility as Chord
    - Items stored on successor

- Consequence:
  - DKS: Cost of maintenance O(1) if no churn
  - Chord Cost of maintenance O(log N) if no churn