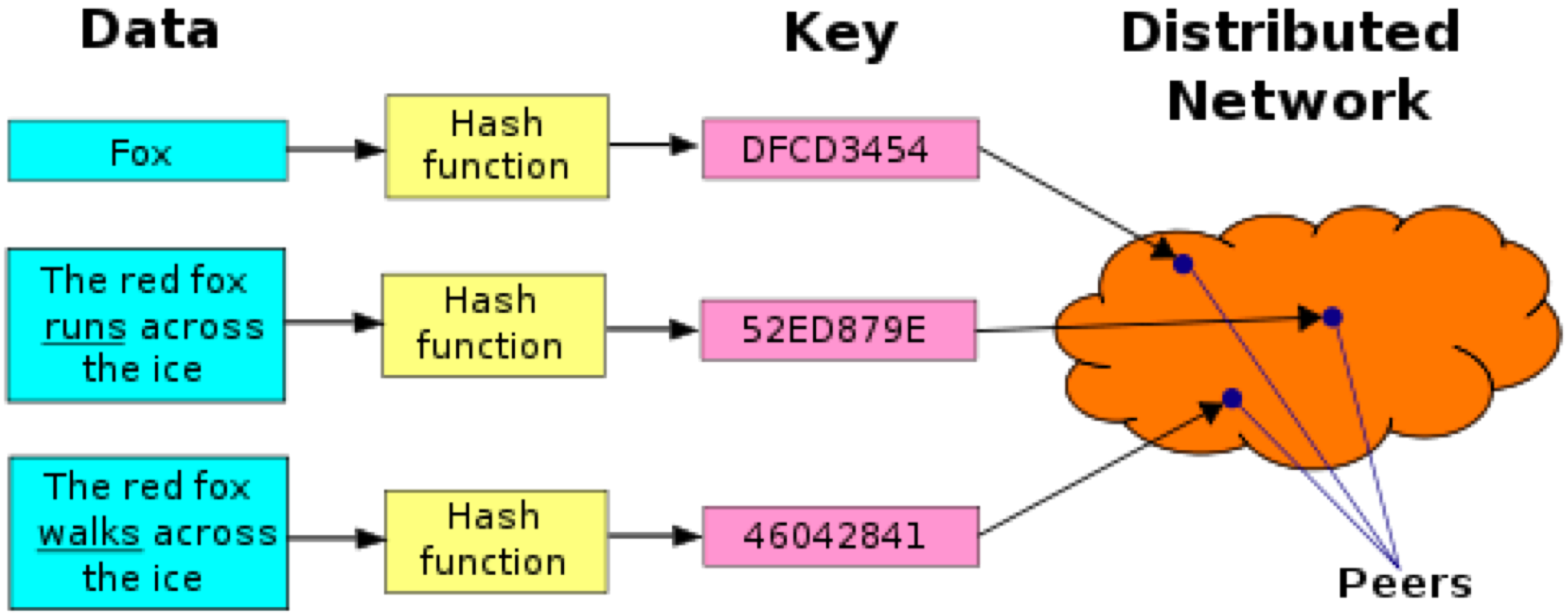
Distributed Hash Table

-In Algorithm Perspective-

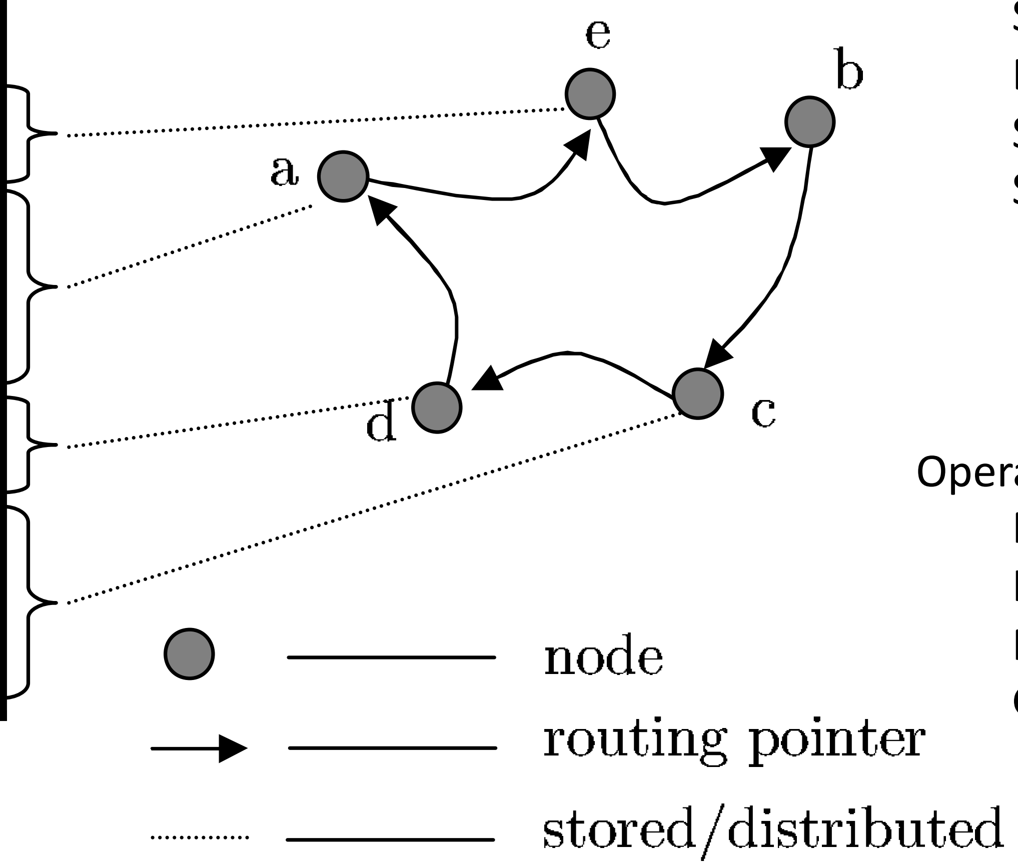
Wenguang Liu

Distributed Hash Table



Distributed Hash Table

Key	Value
"abc.txt"	<url$_1$>
"me.jpg"	<url$_2$>
"music.mp3"	<url$_3$>
"piano.mp3"	<url$_4$>
"cv.doc"	<url$_5$>
"source.zip"	<url$_6$>



Properties:

- Scalable
- Dispersed items
- Scales with dynamism
- Self-manage
  - Node JOIN,
  - Node LEAVE,
  - Node FAIL

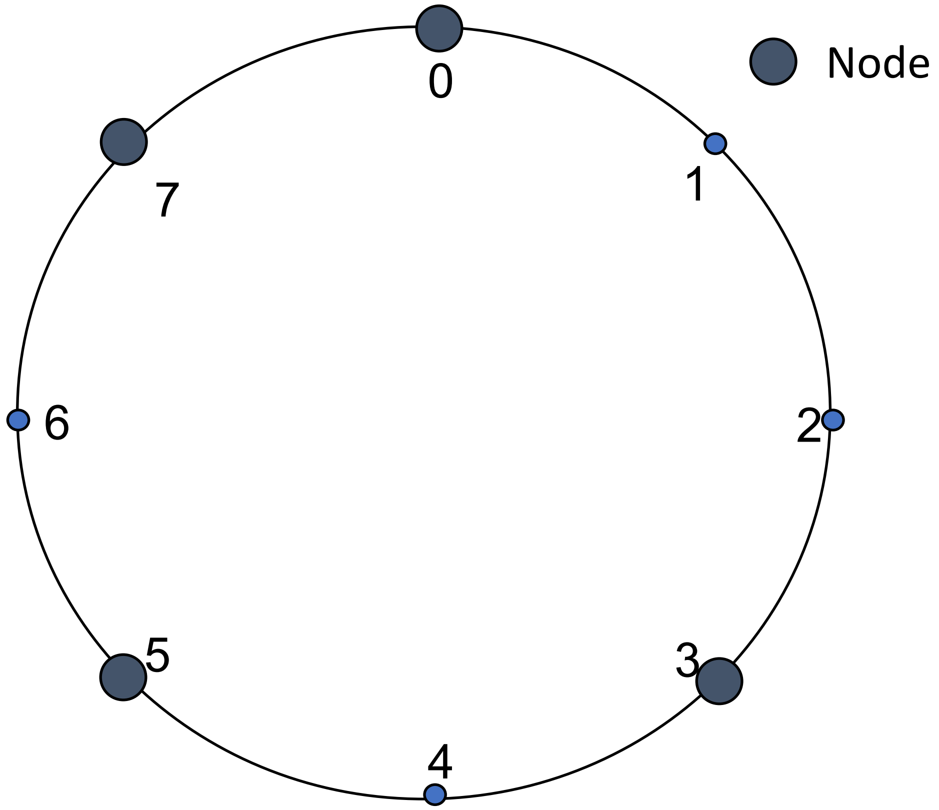
Operations:

- Node management
- K/V management
- Range query
- Group communication

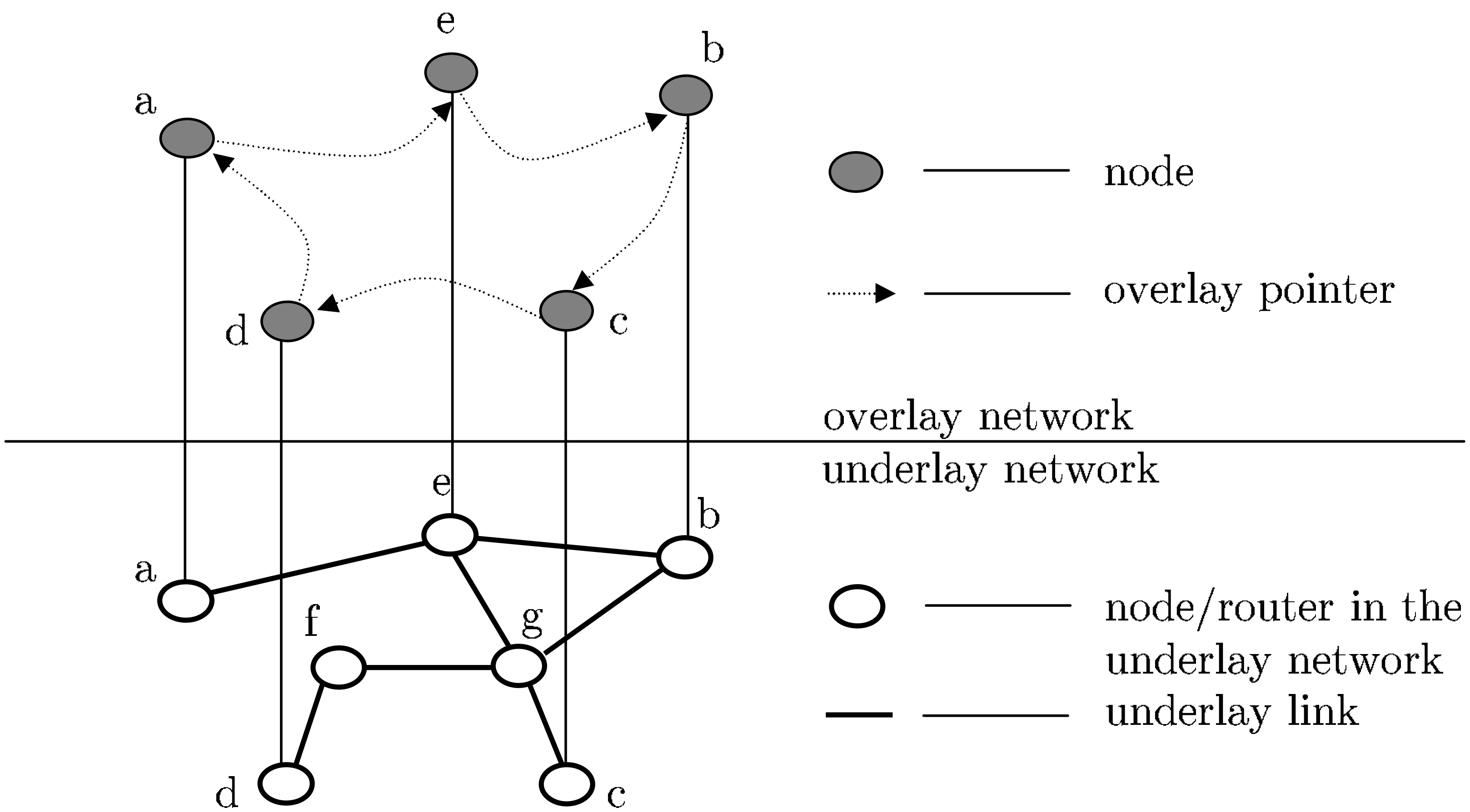
Distributed Hash Table

- Structure
  - Centralized
  - Decentralized/P2P

- Concept
  - Keyspace, e.g. $[0, 2^{\mathrm{N}})$
  - Keyspace Partition, e.g. Consistent Hashing



Overlay Networks



Topic

- Atomic Ring Maintenance
- Routing Maintenance
- Group Communication
- Replication
- Applications

Atomic Ring Maintenance

Atomic Ring Maintenance

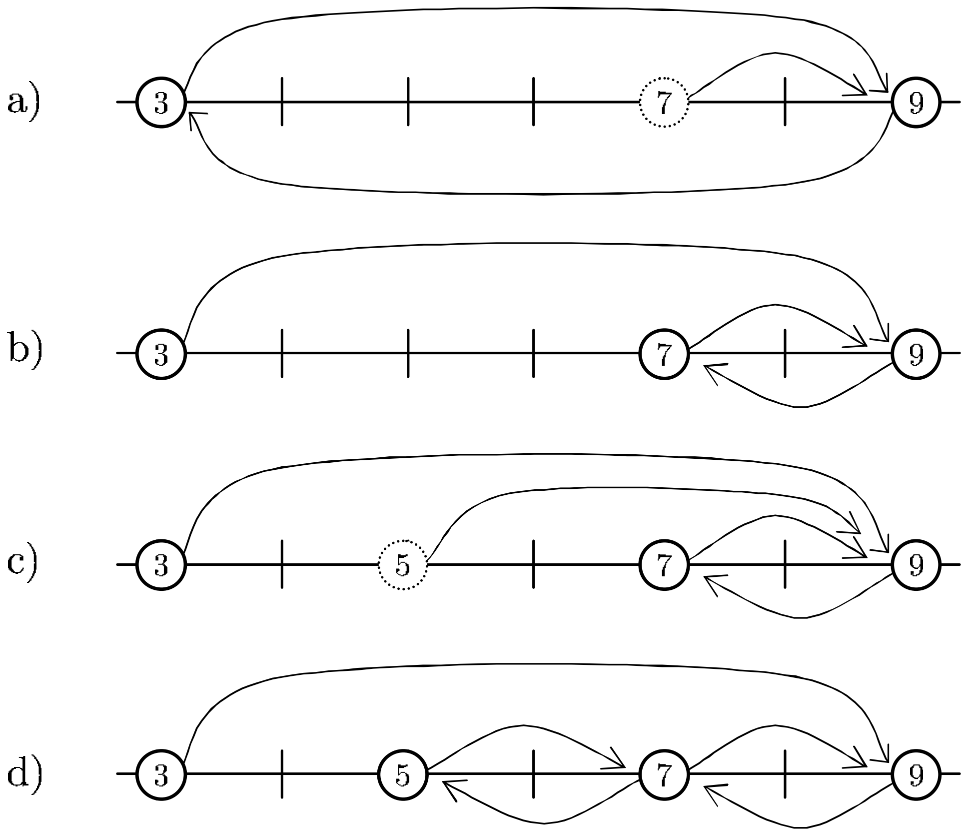


Figure 3.1: Example of inconsistent stabilization.

Algorithm 1 Chord's periodic stabilization protocol

```
procedure n.Stabilize()
    p := succ.GetPredecessor()
    if p ∈ (n, succ) then
        succ := p
    end if
    succ.Notify(n)
end procedure

procedure n.GetPredecessor()
    return pred
end procedure

procedure n.Notify(p)
    if p ∈ (pred, n] then
        pred := p
    end if
end procedure
```

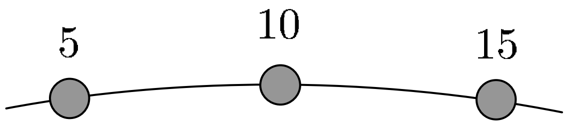


Figure 3.2: Perfect system state before a leave operation.

10 and 15 leave concurrently

Atomic Ring Maintenance

- Concurrency Control strategies:
  - Lock the whole ring;
  - Three locks: predecessor’s lock, successor’s lock, joining/leaving node’s lock;
  - Two locks: Joining/leaving node’s own lock, and its’ successor’s lock;
    - Used in this paper
    - Suffer with Dining philosophers
- Safety
- Liveness
  - Asymmetric locking
  - Randomized locking: release all locks when timeout.

Atomic Ring Maintenance

- Asymmetric Locking: *LockQueue*

Algorithm 2 Asymmetric locking with forwarding

```
procedure n.Join(succ)                          ▷ Join the ring with succ as successor
    Leaving :=false                             ▷ Initialize variable
    LockQueue.Enqueue(n)                        ▷ Enqueue request to local lock
    slock :=GetSuccLock()
    pred := succ.pred
    pred.succ := n
    succ.pred := n
    LockQueue := succ.LockQueue                 ▷ Copy successor's queue
    LockQueue.Filter((pred, n])                 ▷ Keep requests in the range
    succ.LockQueue.Filter((n, pred])            ▷ Keep requests in the range
    LockQueue.Dequeue()                         ▷ Remove local request
    ReleaseLock(slock)
end procedure
procedure n.Leave()                             ▷ Leave the ring
    if n > succ then                            ▷ Asymmetric Locking
        slock :=GetSuccLock()
        Leaving := true                         ▷ Enable forwarding
        LockQueue.Enqueue(n)                    ▷ Enqueue request to local lock
    else
        Leaving := true                         ▷ Enable forwarding
        LockQueue.Enqueue(n)                    ▷ Enqueue request to local lock
        slock :=GetSuccLock()
    end if
    pred.succ := succ
    succ.pred := pred
    LockQueue.Dequeue()                         ▷ Remove local requst
    ReleaseLock(slock)
end procedure
```

Atomic Ring Maintenance

- Lookup consistency in the presence of JOINs: *JoinForward*

Algorithm 4 Pointer updates during joins

```
event n.UpdateJoin() from n                ▷ Assuming succ is correct
    sendto succ.UpdatePred()
end event

event n.UpdatePred() from m
    JoinForward := true                    ▷ Forwarding Enabled
    sendto m.JoinPoint(pred)               ▷ Join Point
    oldpred := pred
    pred := m
end event

event n.JoinPoint(p) from m
    pred := p
    succ := m
    sendto pred.UpdateSucc()
end event

event n.UpdateSucc() from m
    sendto succ.StopForwarding()
    succ := m
end event

event n.StopForwarding() from m
    JoinForward := false
    sendto pred.Finish()
end event
```

Atomic Ring Maintenance

- Lookup consistency in the presence of Leaves: *LeaveForward*

Algorithm 5 Pointer updates during leaves

```
event n.UpdateLeave() from n
    LeaveForward := true                          ▷ Forwarding Enabled
    sendto succ.LeavePoint(pred)
end event

event n.LeavePoint(p) from m
    pred := p
    sendto pred.UpdateSucc()
end event

event n.UpdateSucc() from m
    sendto succ.StopForwarding()
    succ := m
end event

event n.StopForwarding() from m
    LeaveForward :=false                          ▷ Forwarding Disabled
end event
```

Atomic Ring Maintenance

- Combine together

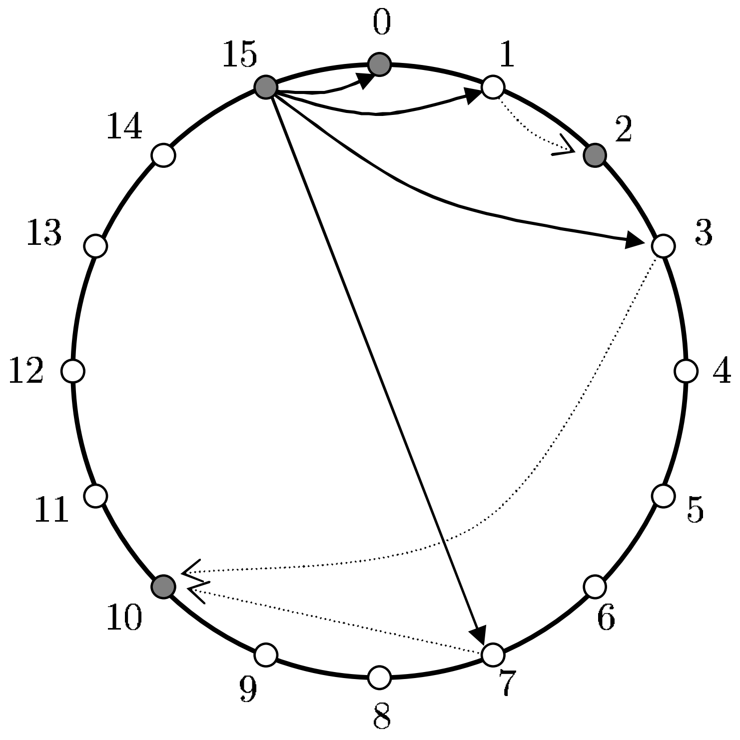
Algorithm 6 Lookup algorithm

```
event n.Lookup(id, src) from m
    if JoinForward = true and m = oldpred then
        sendto pred.Lookup(id, src)                ▷ Redirect Message
    else if LeaveForward = true then
        sendto succ.Lookup(id, src)                ▷ Redirect Message
    else if pred ≠ nil and id ∈ (pred, n] then
        sendto src.LookupDone(n)
    else
        sendto succ.Lookup(id, src)
    end if
end event
```

Routing Maintenance

Routing Maintenance

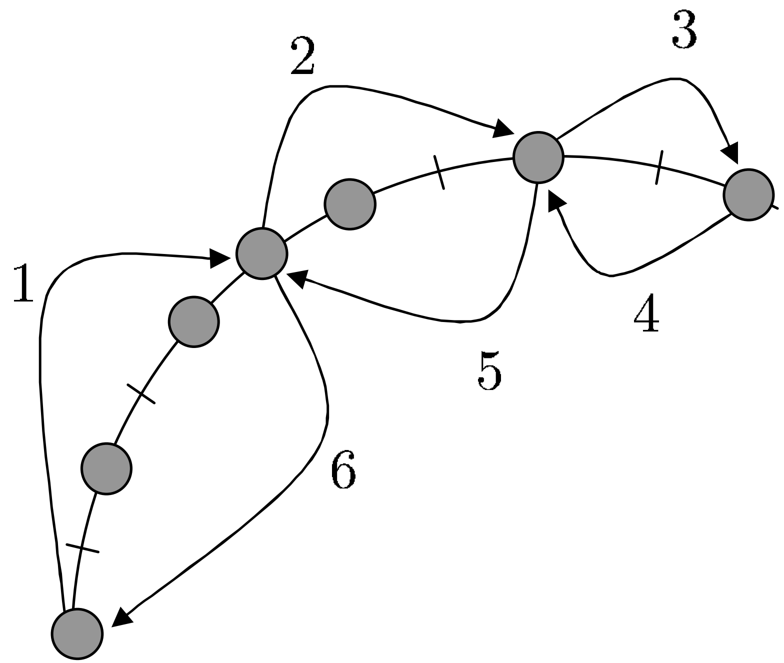
- Extension to the ring: pred, succ, successor-list



$p \oplus 2^{L-1},$

Routing Maintenance

- Lookup: strategy-Recursive Lookup

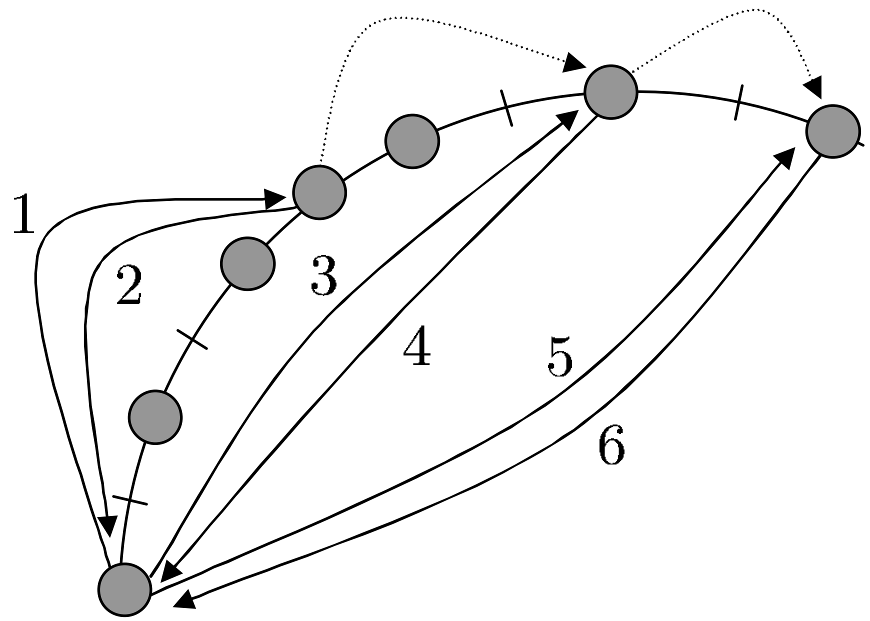


Algorithm 12 Recursive lookup algorithm

```
procedure n.LOOKUP(i, OP)
    if TERMINATE(i) then
        p :=NEXT_HOP(i)
        res := p.OP(i)                    ▷ OP could carry parameters
        return res
    else
        m :=NEXT_HOP(i)
        return m.LOOKUP(i, OP)
    end if
end procedure
```

Routing Maintenance

- Lookup: strategy-Iterative Lookup

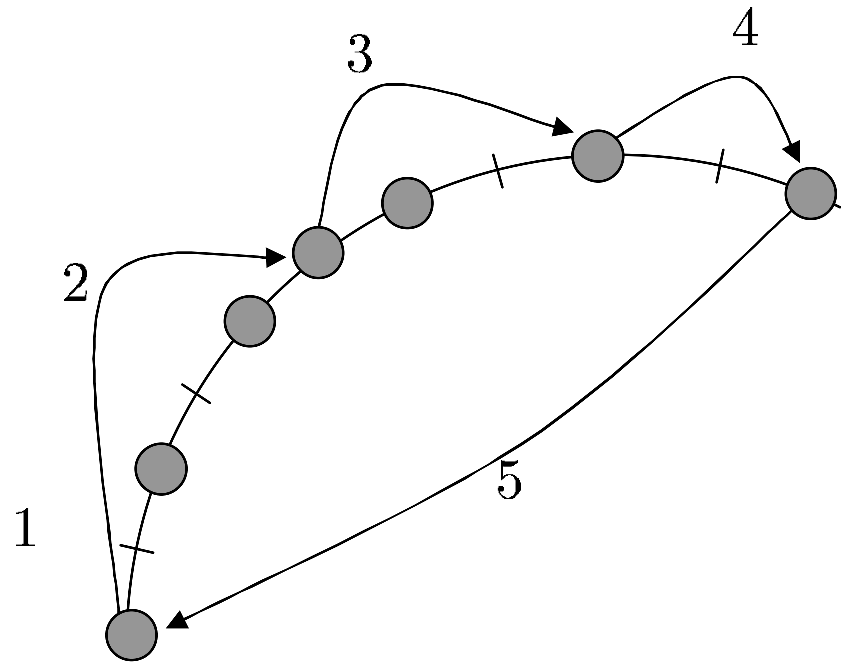


Algorithm 13 Iterative lookup algorithm

```
procedure n.LOOKUP(i, OP)
    m := n
    while not m.TERMINATE(i) do
        m := m.NEXT_HOP(i)
    end while
    p := m.NEXT_HOP(i)
    return p.OP(i)
end procedure
```

Routing Maintenance

- Lookup: strategy-Transitive Lookup



Algorithm 14 Transitive lookup algorithm

```
procedure n.LOOKUP(i, OP)
    sendto n.LOOKUP_AUX(n, i, OP)
    receive LOOKUP_RES(r) from q
    return r
end procedure

event n.LOOKUP_AUX(q, i, OP) from m
    if TERMINATE(i) then
        p := NEXT_HOP(i)
        sendto p.LOOKUP_FIN(q, i, OP)
    else
        p :=NEXT_HOP(i)
        sendto p.LOOKUP_AUX(q, i, OP)
    end if
end event

event n.LOOKUP_FIN(q, i, OP) from m
    r := OP(i)
    sendto q.LOOKUP_RES(r)
end event
```

Routing Maintenance

- Greedy Lookup Algorithm
  - rt(i): sucessors sorted by routing distance asc.

Algorithm 15 Greedy lookup

```
procedure n.TERMINATE(i)
    return  i ∈ (n, succ]
end procedure

procedure n.NEXT_HOP(i)
    if TERMINATE(i) then
        return succ
    else
        r := succ
        for j := 1  to  K do
            if rt(j) ∈ (n, i) then
                r := rt(j)
            end if
        end for
        return r
    end if
end procedure
```

Group Communication

Group Communication

- Motivation
  - Exactly match → Wildcard expression

- Desirable properties
  - Termination
  - Coverage,
    - All the *designated* nodes that are *reachable* should receive the message
  - Non-redundancy
    - Never receive a message more than once.

Group Communication

- Broadcast algorithms – simple broadcast

Algorithm 19 Simple broadcast algorithm

```
event n.StartSimpleBcast(msg) from app
    sendto n.SimpleBcast(msg, n)            ▷ Local message to itself
end event

event n.SimpleBcast(msg, limit) from m
    Deliver(msg)                            ▷ Deliver msg to application
    for i := M downto 1 do                  ▷ Node has M unique pointers
        if u(i) ∈ (n, limit) then
            sendto u(i).SimpleBcast(msg, limit)
            limit := u(i)
        end if
    end for
end event
```

Group Communication

- Broadcast algorithms – simple broadcast with feedback

Algorithm 20 Simple broadcast with feedback algorithm

```
event n.StartBcast(msg) from app
    sendto n.Bcast(msg, n)                          ▷ Local message to itself
end event

event n.Bcast(msg, limit) from m
    FB := Deliver(msg)              ▷ Deliver msg and get set of feedback
    par := n
    Ack := ∅
    for i := M  downto  1 do                  ▷ Node has M unique pointers
        if u(i) ∈ (n, limit) then
            sendto u(i).Bcast(msg, limit)
            Ack := Ack ∪ {u(i)}
            limit := u(i)
        end if
    end for
    if Ack = ∅ then
        sendto par.BcastResp(FB)
    end if
end event
```

```
event n.BcastResp(F) from m
    if m = n then
        sendto app.BcastTerm(FB)
    else
        Ack := Ack − {m}
        FB := FB ∪ F
        if Ack = ∅ then
            sendto par.BcastResp(FB)
        end if
    end if
end event
```

Group Communication

- Bulk Operations,
  - To designated nodes with identifier in **Bulk Set** *I*.

Algorithm 21 Bulk operation algorithm

```
event n.BULK(I, msg) from m
    if n ∈ I then
        Deliver(msg)                        ▷ Deliver msg to application
    end if
    limit := n
    for i := M downto 1 do                  ▷ Node has M unique pointers
        J := [u(i), limit)
        if I ∩ J ≠ ∅ then
            sendto u(i).BULK(I ∩ J, msg)
            I := I − J                      ▷ Same as I := I − (I ∩ J)
            limit := u(i)
        end if
    end for
end event
```

Group Communication

- Bulk Operations with feedback

Algorithm 22 Bulk operation with feedback algorithm

```
event n.BulkFeed(I, msg) from m
    if n ∈ I then
        FB := Deliver(msg)                    ▷ Deliver and get set of feedback
    else
        FB := ∅                               ▷ No feedback
    end if
    par := m
    Ack := ∅
    limit := n
    for i := M downto 1 do                    ▷ Node has M unique pointers
        J := [u(i), limit)
        if I ∩ J ≠ ∅ then
            sendto u(i).BulkFeed(I ∩ J, msg)
            I := I − J                        ▷ Same as I := I − (I ∩ J)
            Ack := Ack ∪ {u(i)}
            limit := u(i)
        end if
    end for
    if Ack = ∅ then
        sendto par.BulkResp(FB)
    end if
end event
```

```
event n.BulkResp(F) from m
    if m = n then
        sendto app.BulkFeedTerm(FB)
    else
        Ack := Ack − {m}
        FB := FB ∪ F
        if Ack = ∅ then
            sendto par.BulkResp(FB)
        end if
    end if
end event
```

Group Communication

- Bulk Owner Operations

Algorithm 24 Bulk owner operation algorithm

```
event n.StartBulkOwn(I, msg) from m
    sendto n.BulkOwn(I, I, n, msg)                 ▷ Local message to itself
end event
```

```
event n.BulkOwn(I, R, next, msg) from m
    MS := R ∩ (u(M), n]                            ▷ u(M) is same as pred
    if MS ≠ ∅ then
        Deliver(msg, MS)                           ▷ App is responsible for ids in MS
    end if
    limit := n
    lnext := next
    sentsucc :=false
    for i := M  downto  1 do                       ▷ Node has M unique pointers
        J := (u(i), limit]
        if I ∩ J ≠ ∅ then
            K := (u(i − 1), u(i)]
            sendto u(i).BulkOwn(I ∩ J, I ∩ K, lnext, msg)
            I := I − J                             ▷ Same as I := I − (I ∩ J)
            limit := u(i)
            lnext := u(i)
            if i = 1 then
                sentsucc :=true
            end if
        end if
    end for
    J := (n, u(1)]
    if I ∩ J ≠ ∅ and sentsucc = false and next ≠ u(1) then
        sendto u(1).BulkOwn(∅, I ∩ J, limit, msg)
    end if
end event
```

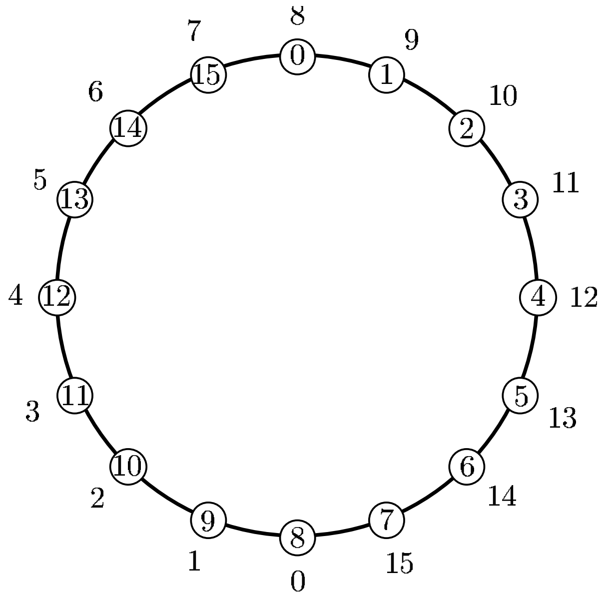
Group Communication

- Fault-tolerance
  - Use timeouts to detect node failure.
- Efficient overlay multicast
  - Multicast-Group: creating, joining, leaving.
  - IP multicast integration

Replication

Replication

- Symmetric replication scheme
  - the identifier space is partitioned into *N/f* equivalence classes,
    - identifiers in an equivalence class are all associated with each other
  - if the identifier *i* is associated with the set of identifiers *r1, ...,rf*, then the identifier *rx*, for *1 ≤ x ≤ f* , is associated with the identifiers *r1, ...,rf* as well.



$$r(i,x) = i \oplus (x-1)\frac{N}{f}$$

$$1 \le x \le f, \quad f = 2.$$

$$1 \equiv 9 (\text{mod } 8).$$

Replication

- Join and Leave Algo

Algorithm 25 Symmetric replication for joins and leaves

```
event n.JoinReplication() from m
    sendto succ.RetrieveItems(pred, n, n)
end event

event n.LeaveReplication() from m
    sendto n.RetrieveItems(pred, n, succ)
end event

event n.RetrieveItems(start, end, p) from m
    for r := 1 to f do
        items[r] := ∅
        i := start
        while i ≠ end do
            i := i ⊕ 1
            items[r][i] := localHashTable[r][i]
        end while
    end for
    sendto p.Replicate(items, start, end)
end event

event n.Replicate(items, start, end) from m
    for r := 1 to f do
        i := start
        while i ≠ end do
            i := i ⊕ 1
            localHashTable[r][i] := items[r][i]
        end while
    end for
end event
```

Replication

- Lookup and Insertion

Algorithm 26 Lookup and item insertion for symmetric replication

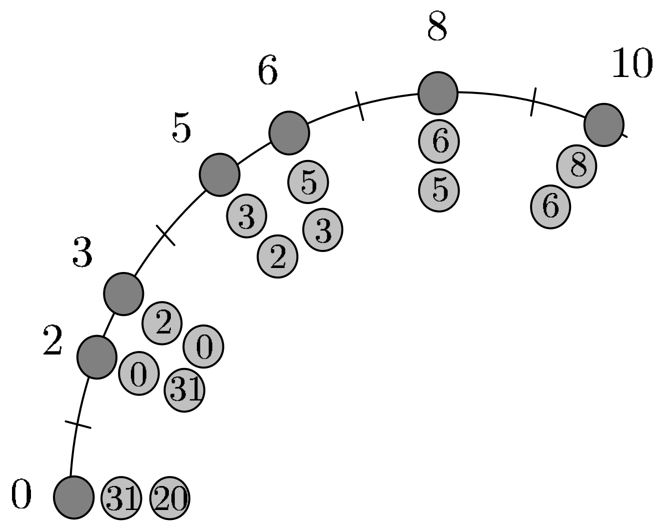
```
event n.InsertItem(key, value) from app
    for r := 1 to f do
        replicaKey := key ⊕ (r − 1)N/f
        n.Lookup(replicaKey, AddItem(replicaKey, value, r))
    end for
end event

procedure n.AddItem(key, value, r)
    localHashTable[key][r] := value
end procedure

event n.LookupItem(key, r) from app
    replicaKey := key ⊕ (r − 1)N/f
    Lookup(replicaKey, GetItem(replicaKey, r))
end event

procedure n.GetItem(key, r)
    return localHashTable[r][key]
end procedure
```

Replication

- Symmetric replication scheme
  - Symmetric replication enables an application to make parallel lookups to exactly k replicas of an item, where $k \leq f$ if the replication degree is f .
    - to speed up the lookup process
  - a join or a leave only requires the joining or leaving node to exchange data with its successor prior to joining or leaving.

Replication

- Other replica placement schemas
  - Multiple Hash Functions: hash key with f hash functions
  - Successor lists: store at the f closest successors
  - Leaf sets: store on $\lfloor f/2 \rfloor$ closest successors and $\lfloor f/2 \rfloor$ closest predecessors



successor-list replication

Applications

Applications

- Storage system
  - PAST[Storage management and caching in past, a large-scale, persistent peer-to-peer storage utility]
  - CFS[Wide-area cooperative storage with CFS]
- Host discovery and mobility
- Web caching and web servers
- Publish/subscribe systems, e.g. FeedTree
- P2P, e.g. BitTorrent

Take-home Message

- A Randomized locking mechanism with node's and succ's lock
  - to support atomic ring management (JOIN, LEAVE, LOOKUP)
- Routing maintainese
  - by additional routing pointer augment.
  - With Recursive/Iterative/Transitive/Greedy lookup algo.
- Provide the Algo for Group communication
  - Broadcast, Bulk, Bulk own
- Provide the symmetric replication mechanism
  - to augment the robustness

References

- ALI GHODSI, Distributed k-ary System: Algorithms for Distributed Hash Tables.
- https://en.wikipedia.org/wiki/Distributed_hash_table