CS162
Operating Systems and Systems Programming
Lecture 15
Chord, Network Protocols

March 14, 2012

nthony D. Joseph and Ion Stoica

http://inst.eecs.berkeley.edu/~cs162

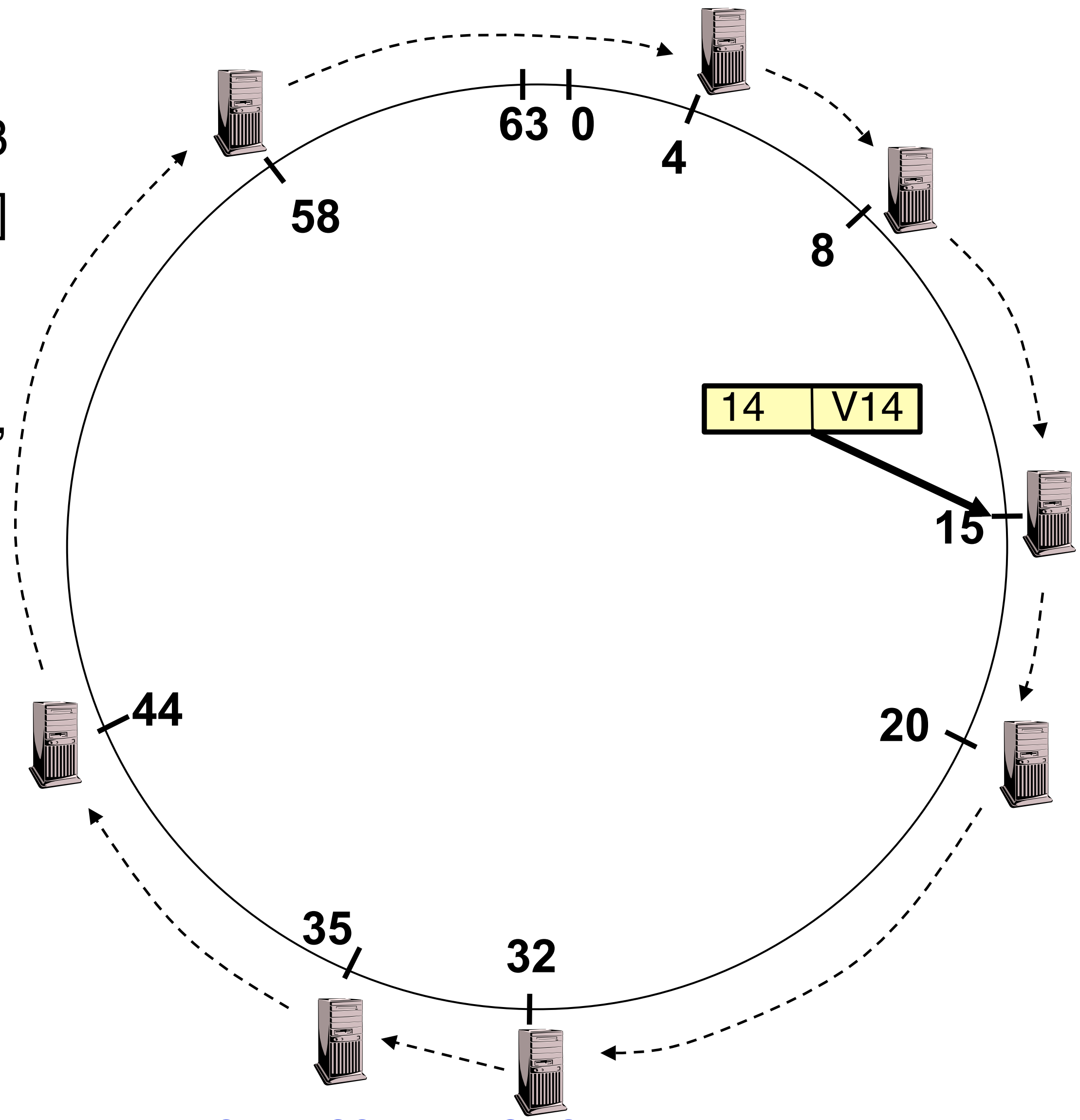
Recap: Scaling Up Directory

- Challenge:
  – Directory contains a number of entries equal to number of (key, value) tuples in the system
  – Can be tens or hundreds of billions of entries in the system!

- Solution: **consistent hashing**
- Associate to each node a unique *id* in an *uni-*dimensional space $0..2^m-1$
  – Partition this space across *M* machines
  – Assume keys are in same uni-dimensional space
  – Each (Key, Value) is stored at the node with the smallest ID larger than Key

Recap: Key to Node Mapping Example

- m = 8 → ID space: 0..63
- Node 8 maps keys [5,8]
- Node 15 maps keys [9,15]
- Node 20 maps keys [16, 20]
- …
- Node 4 maps keys [59, 4]



Recap: Scaling Up Directory

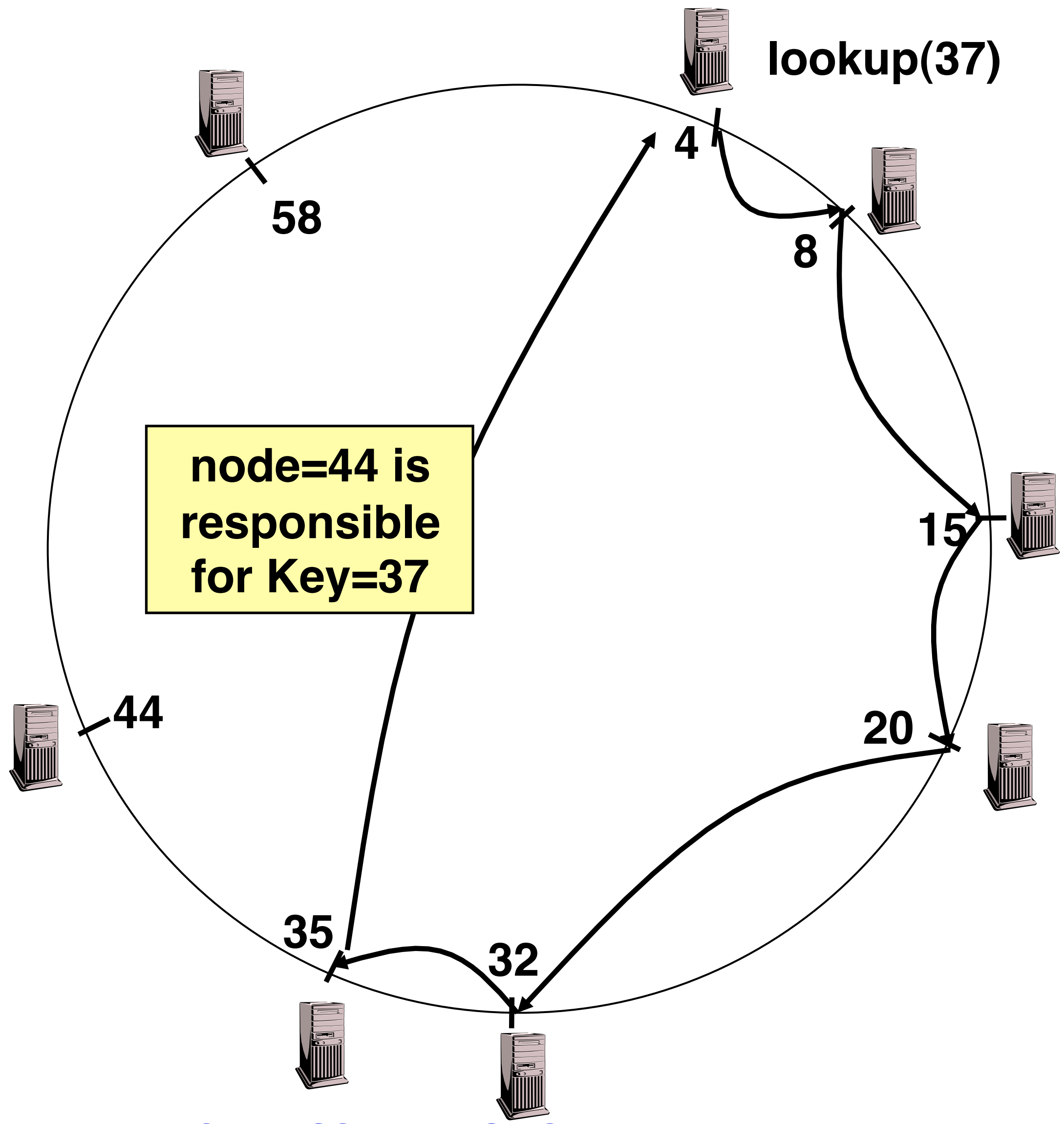
- With consistent hashing, directory contains only a number of entries equal to number of nodes
  - Much smaller than number of tuples
- Next challenge: every query still needs to contact the directory
- Solution: distributed directory (a.k.a. lookup) service:
  - Given a **key**, find the **node** storing that key
- Key idea: route request from node to node until reaching the node storing the request's key
- Key advantage: totally distributed
  - No point of failure; no hot spot

Chord: Distributed Lookup (Directory) Service

- Key design decision
  - Decouple correctness from efficiency
- Properties
  - Each node needs to know about O(log(*M*)), where *M* is the total number of nodes
  - Guarantees that a tuple is found in O(log(*M*)) steps
- Many other lookup services: CAN, Tapestry, Pastry, Kademlia, …

Lookup

- Each node maintains pointer to its successor
- Route packet (Key, Value) to the node responsible for ID using successor pointers
- E.g., node=4 lookups for node responsible for Key=37



Stabilization Procedure

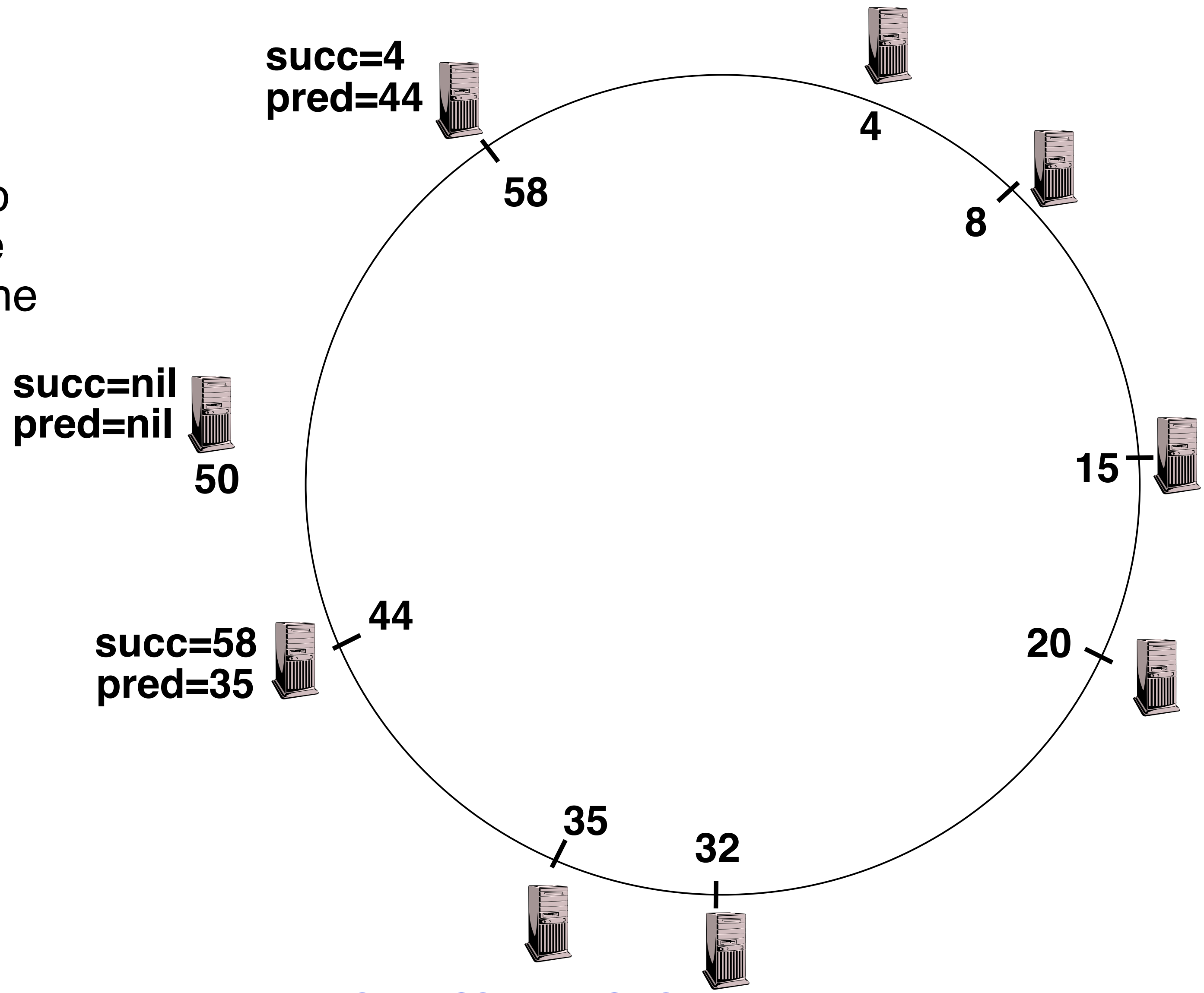
- Periodic operation performed by each node n to maintain its successor when new nodes join the system

```
n.stabilize()
  x = succ.pred;
  if (x ∈ (n, succ))
     succ = x;       // if x better successor, update
  succ.notify(n); // n tells successor about itself

n.notify(n')
  if (pred = nil or n' ∈ (pred, n))
     pred = n';       // if n' is better predecessor, update
```

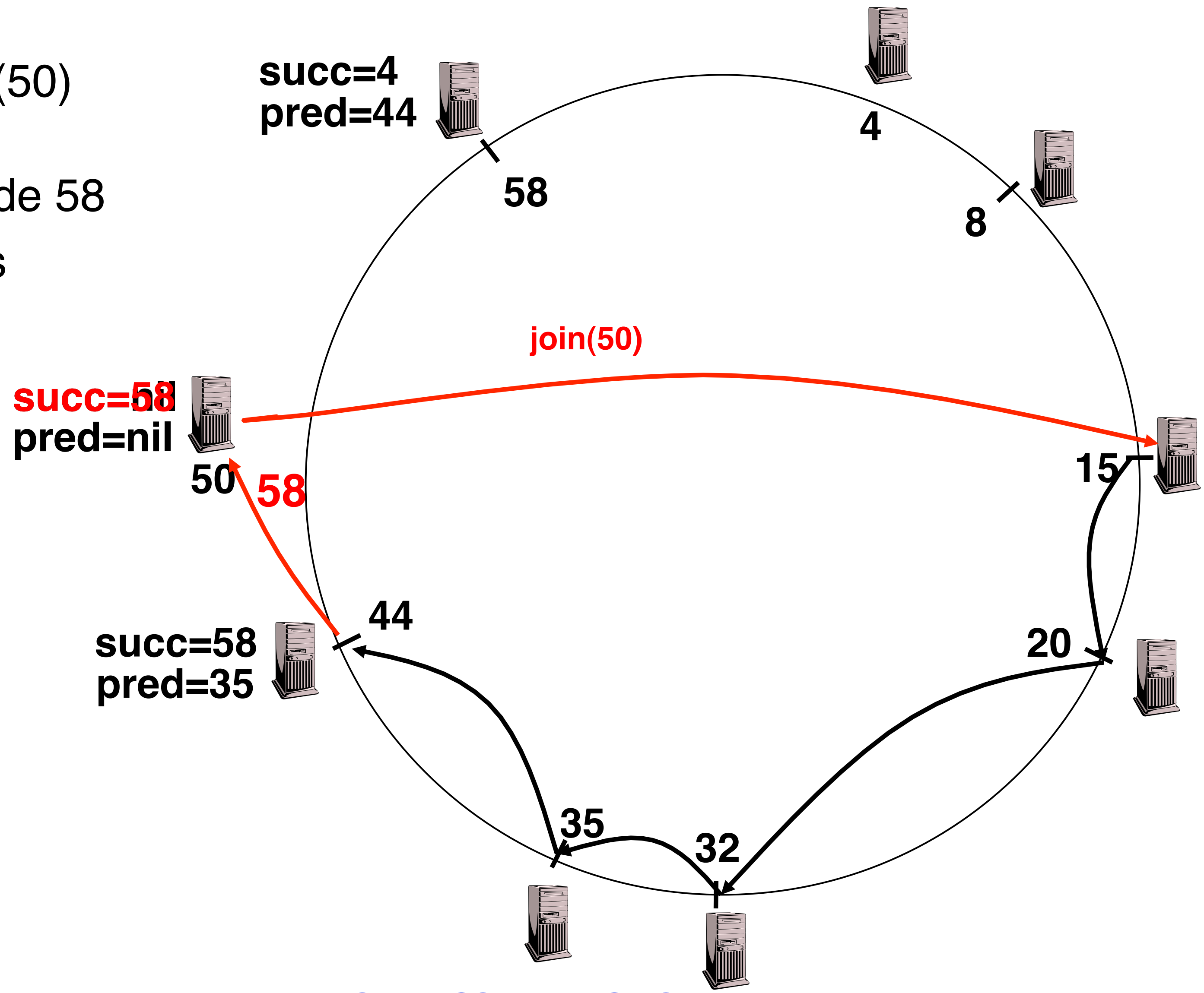
Joining Operation

- Node with id=50 joins the ring
- Node 50 needs to know at least one node already in the system
  - Assume known node is 15



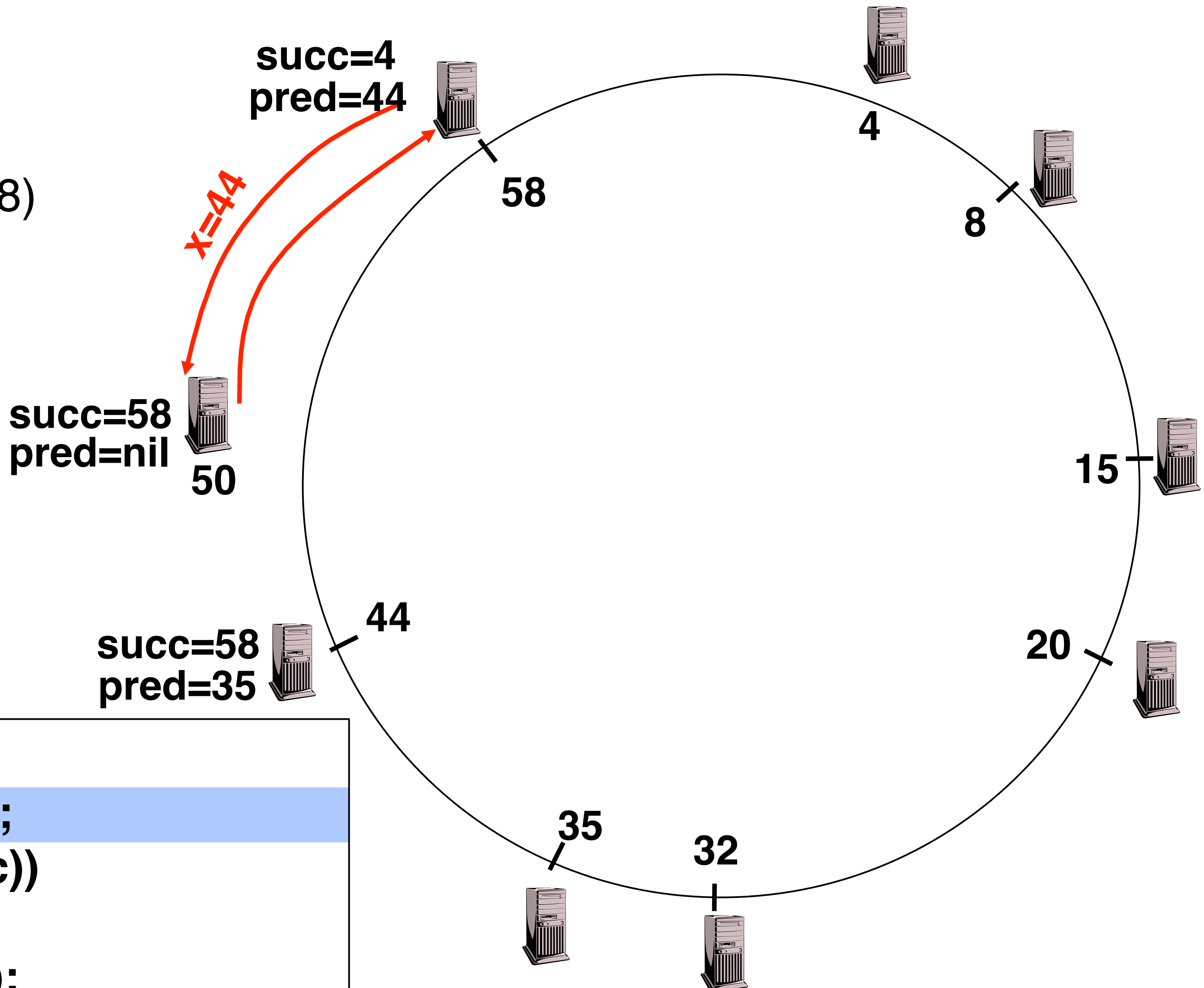
Joining Operation

- n=50 sends join(50) to node 15
- n=44 returns node 58
- n=50 updates its successor to 58



Joining Operation

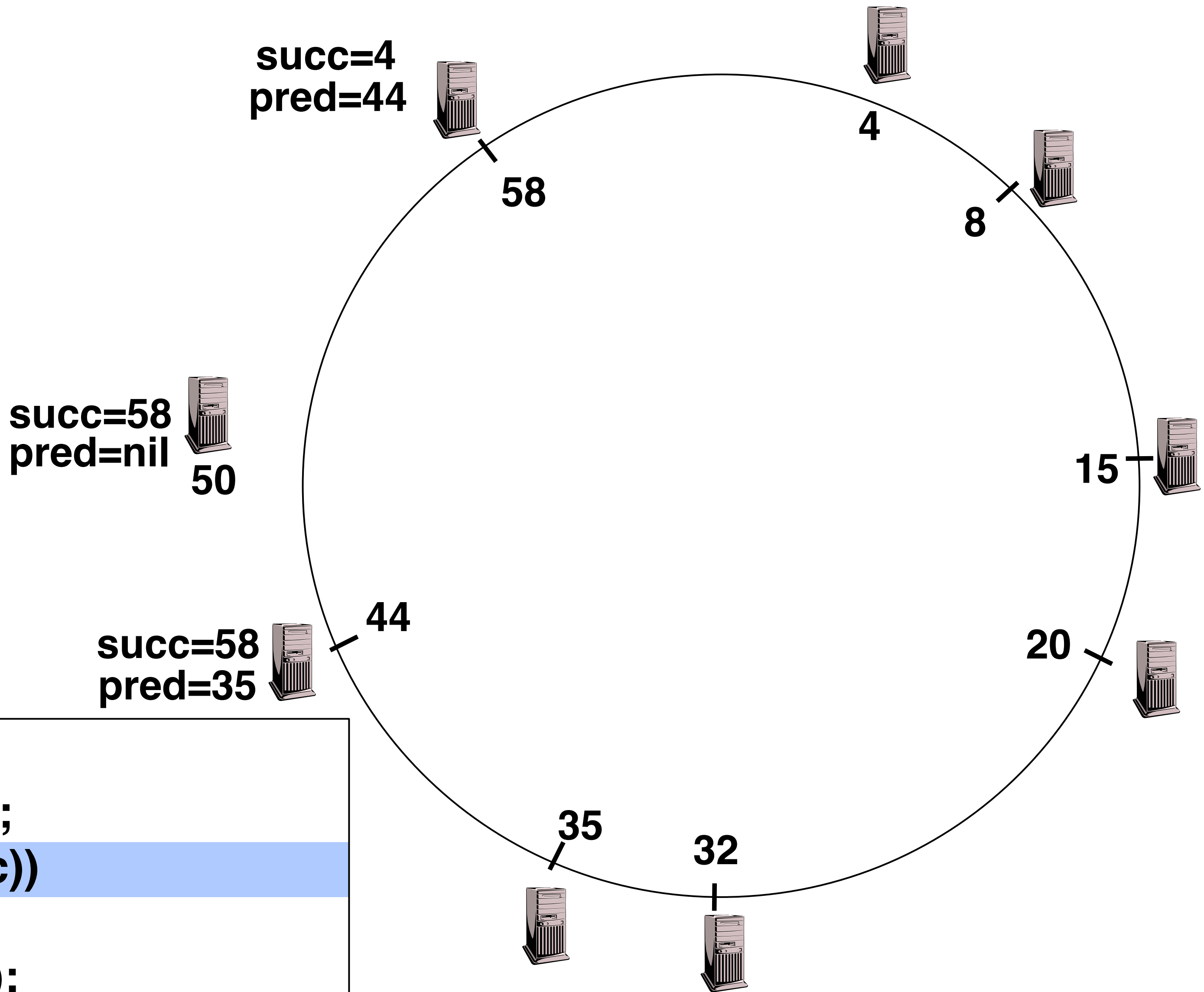
- n=50 executes stabilize()
- n's successor (58) returns x = 44




```
n.stabilize()
  x = succ.pred;
  if (x∈(n, succ))
    succ = x;
  succ.notify(n);
```

Joining Operation

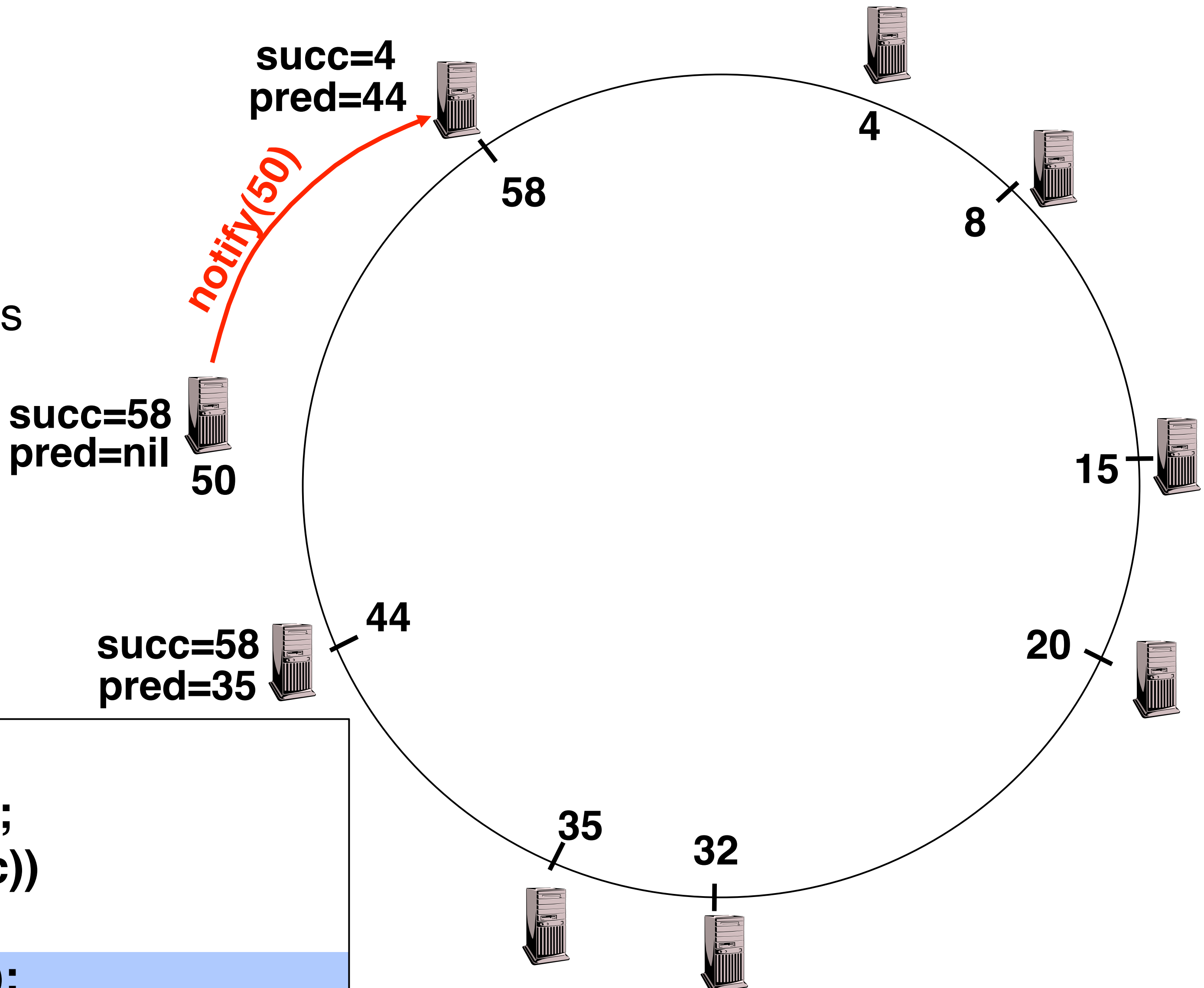
- n=50 executes stabilize()
  - x = 44
  - succ = 58




```
n.stabilize()
  x = succ.pred;
  if (x∈(n, succ))
     succ = x;
  succ.notify(n);
```

Joining Operation

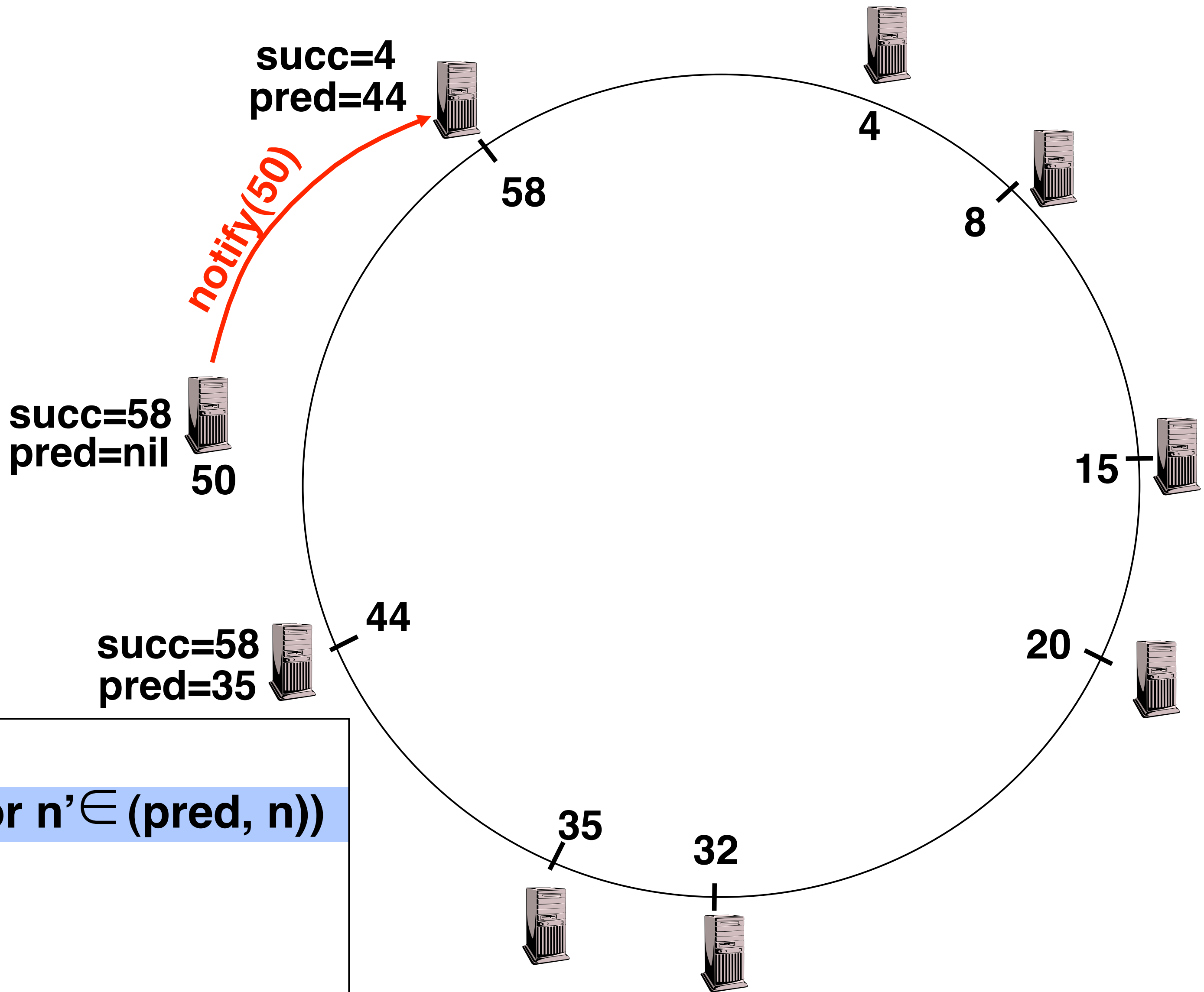
- n=50 executes stabilize()
  - x = 44
  - succ = 58
- n=50 sends to it's successor (58) notify(50)




```
n.stabilize()
  x = succ.pred;
  if (x∈(n, succ))
     succ = x;
  succ.notify(n);
```

Joining Operation

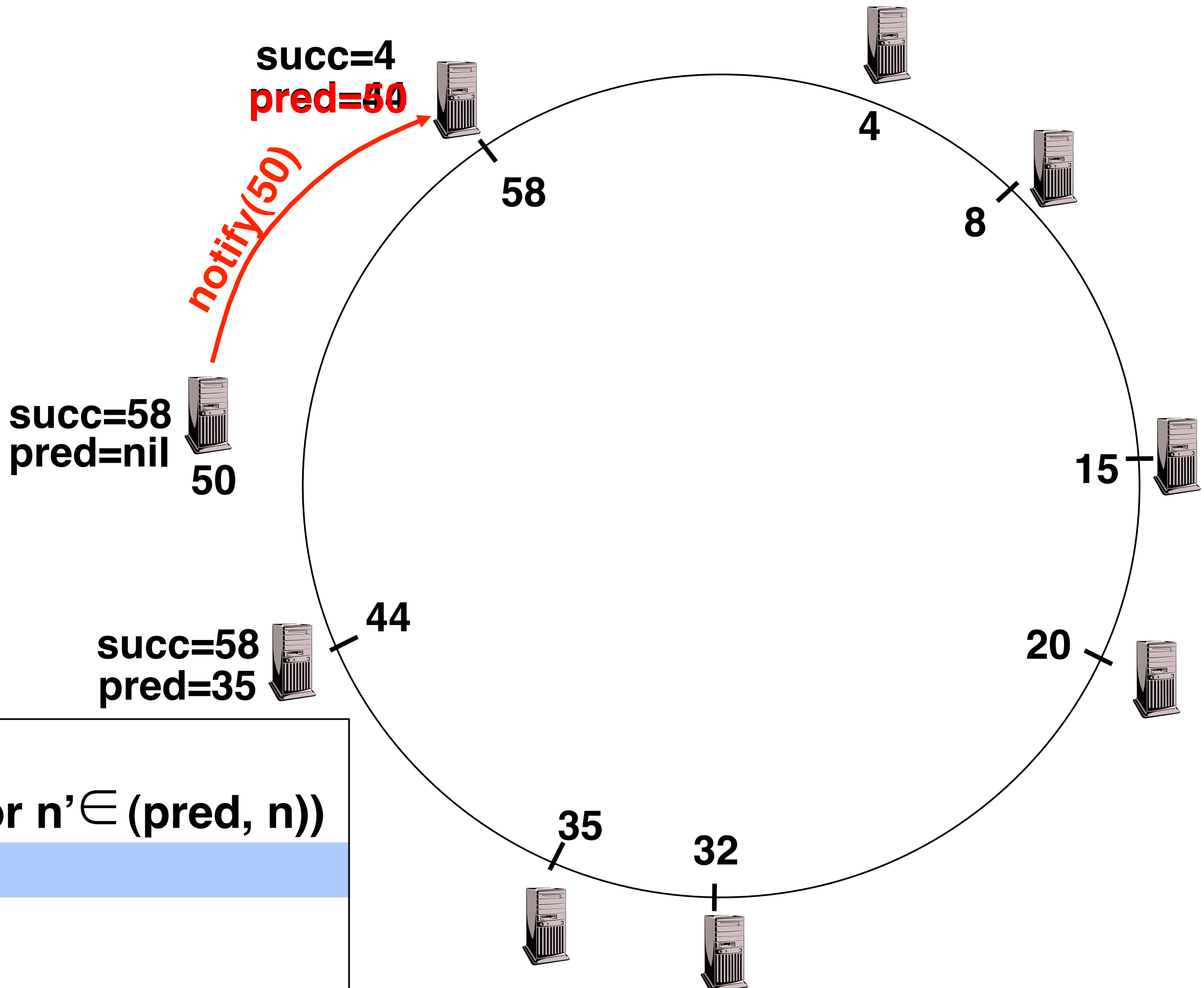
- n=58 processes notify(50)
  - pred = 44
  - n' = 50



n.notify(n')
if (pred = nil or n'∈(pred, n))
pred = n'

Joining Operation

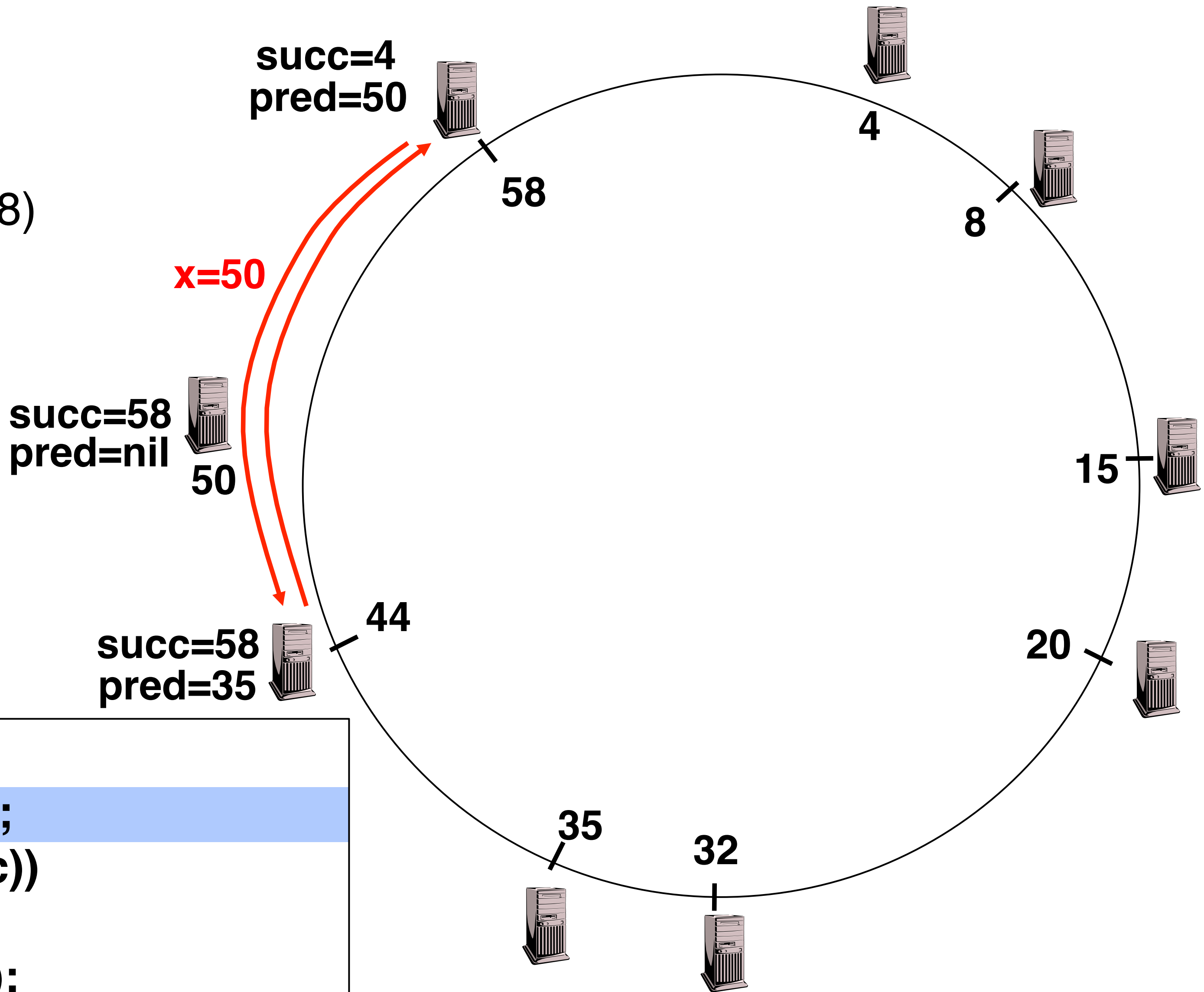
- n=58 processes notify(50)
  - pred = 44
  - n' = 50
- set pred = 50




```
n.notify(n')
  if (pred = nil or n'∈(pred, n))
    pred = n'
```

Joining Operation

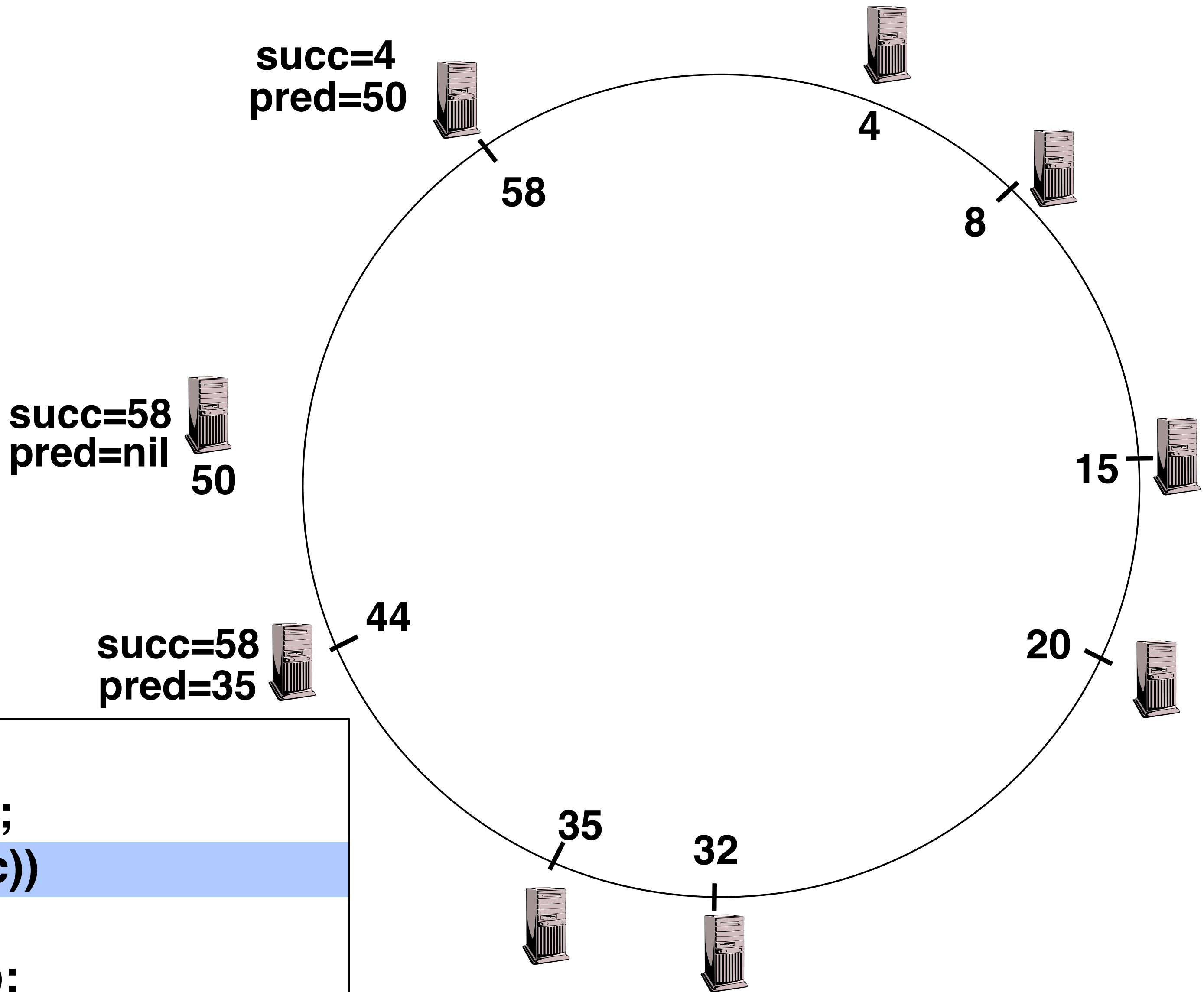
- n=44 runs stabilize()
- n's successor (58) returns x = 50




```
n.stabilize()
  x = succ.pred;
  if (x∈(n, succ))
     succ = x;
  succ.notify(n);
```

Joining Operation

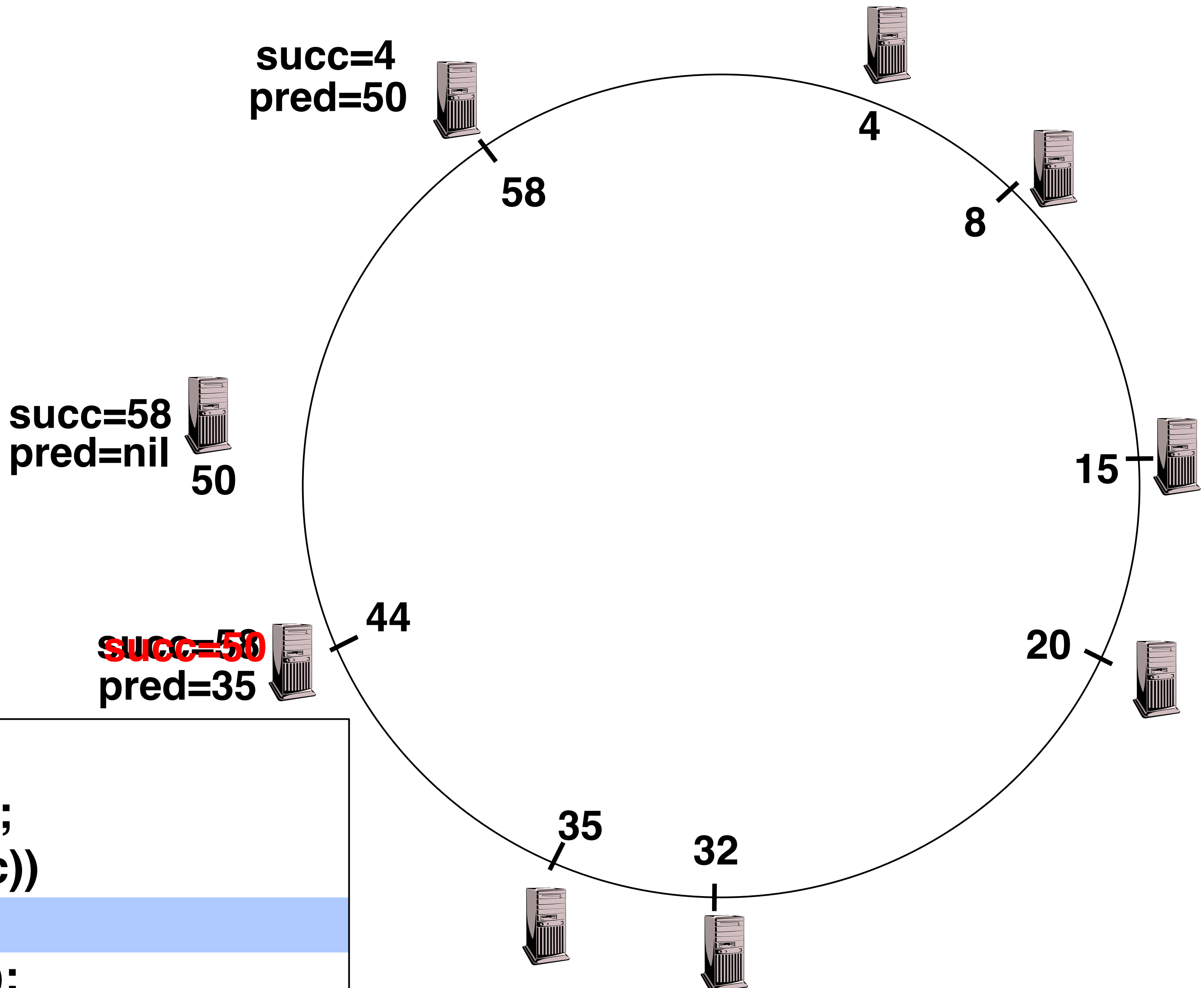
- n=44 runs stabilize()
  - x = 50
  - succ = 58




```
n.stabilize()
  x = succ.pred;
  if (x∈(n, succ))
     succ = x;
  succ.notify(n);
```

Joining Operation

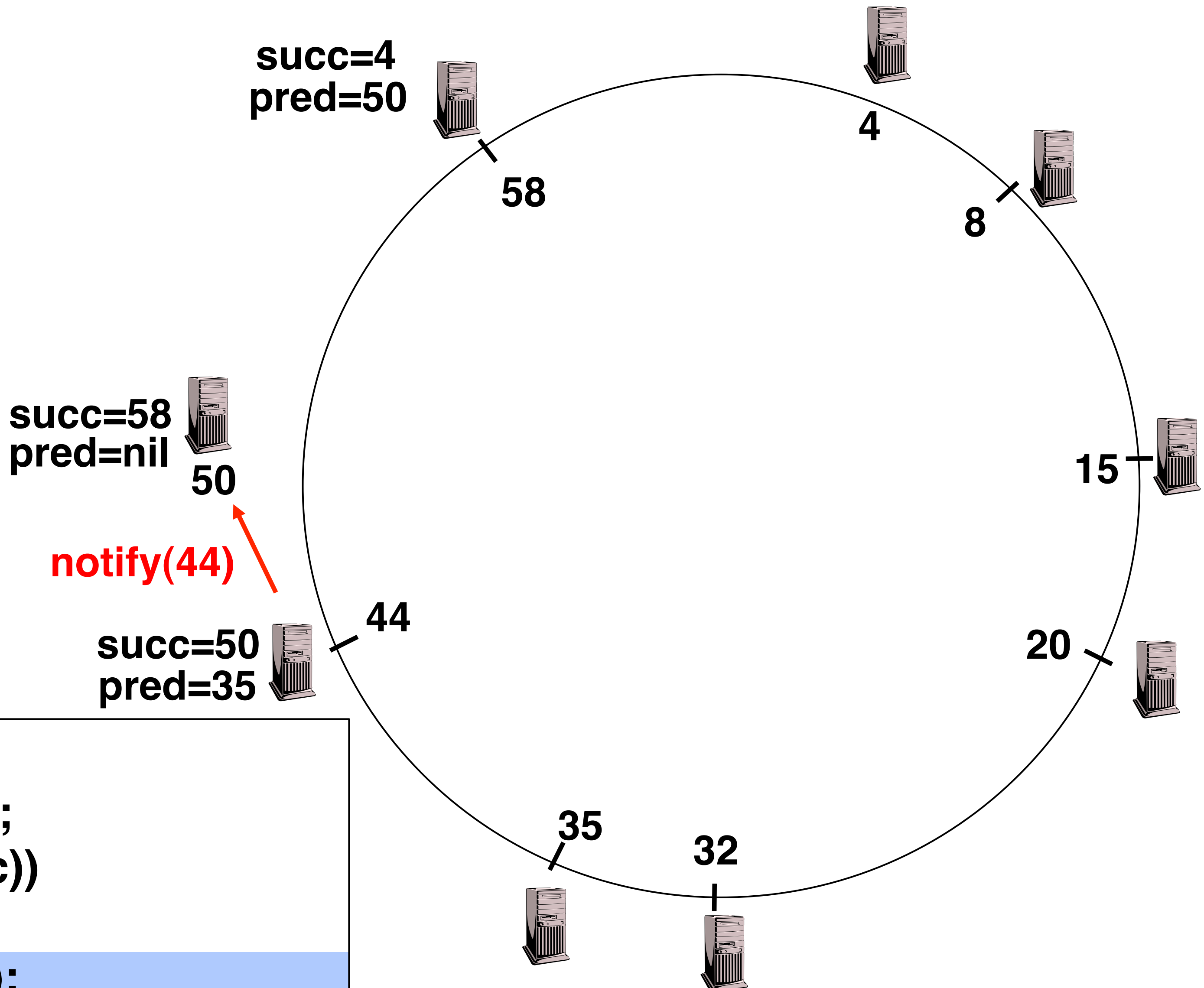
- n=44 runs stabilize()
  - x = 50
  - succ = 58
- n=44 sets succ=50




```
n.stabilize()
  x = succ.pred;
  if (x∈(n, succ))
    succ = x;
  succ.notify(n);
```

Joining Operation

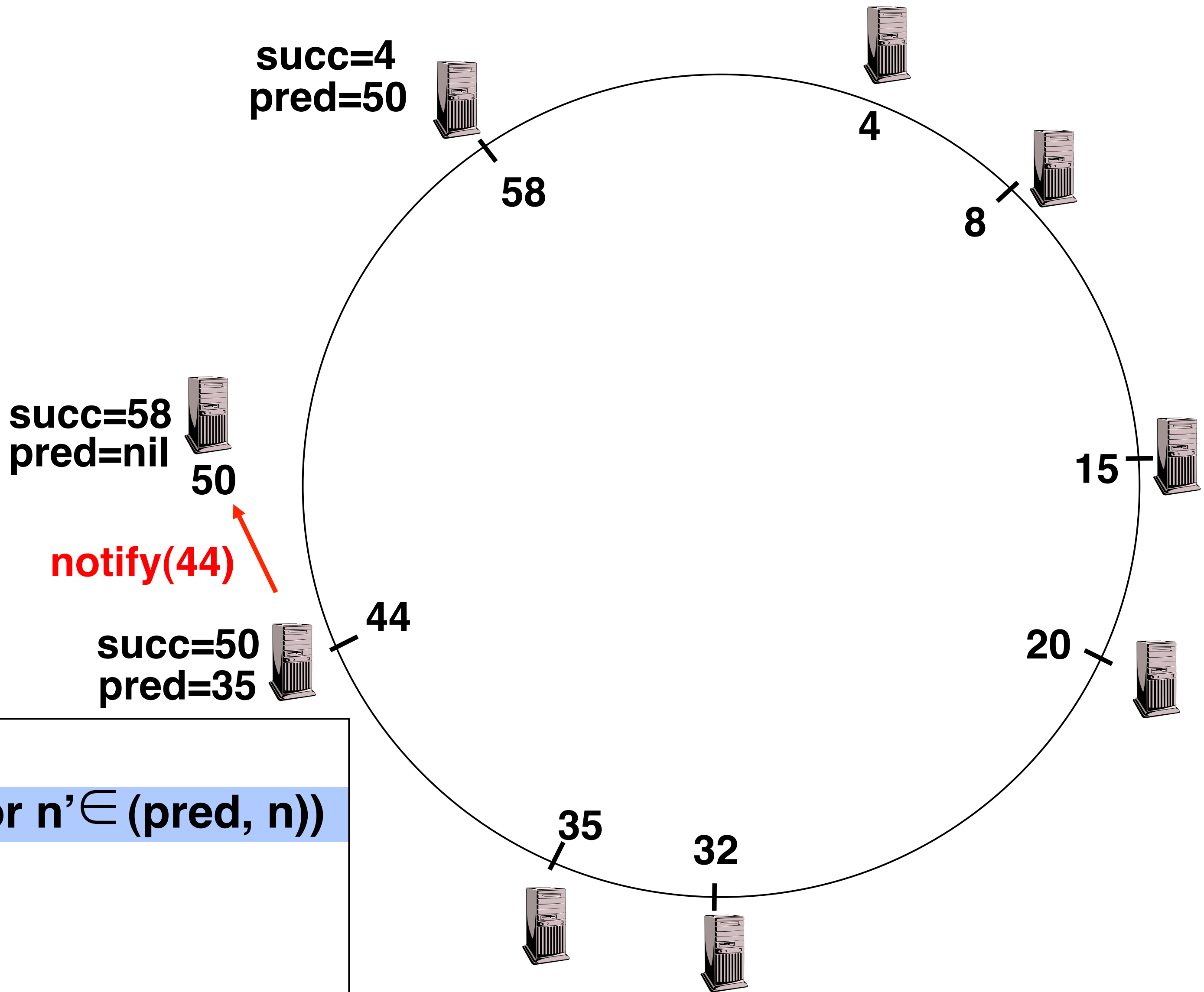
- n=44 runs stabilize()
- n=44 sends notify(44) to its successor




```
n.stabilize()
  x = succ.pred;
  if (x∈(n, succ))
     succ = x;
  succ.notify(n);
```

Joining Operation

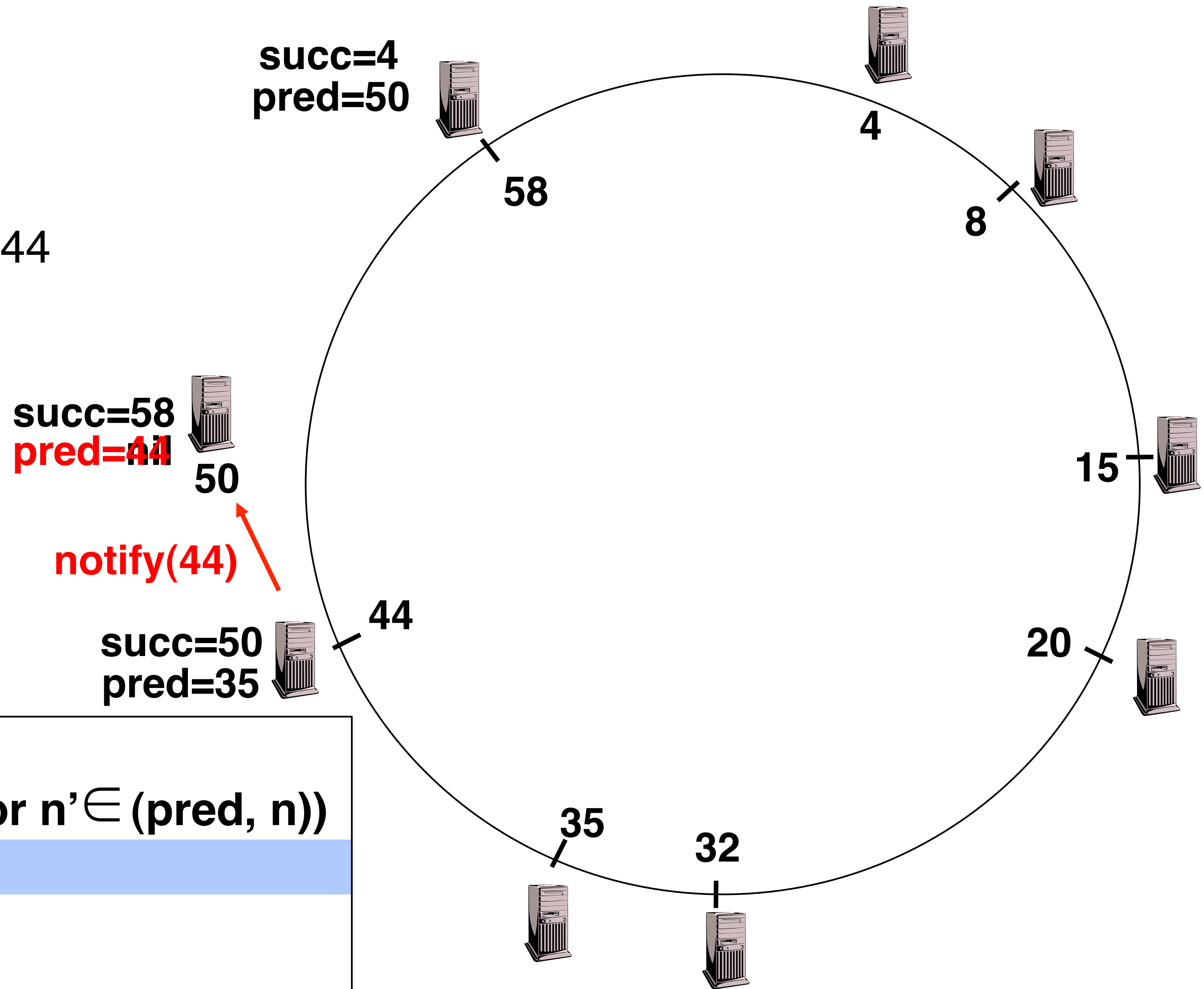
- n=50 processes notify(44)
  - pred = nil




```
n.notify(n')
  if (pred = nil or n'∈(pred, n))
    pred = n'
```

Joining Operation

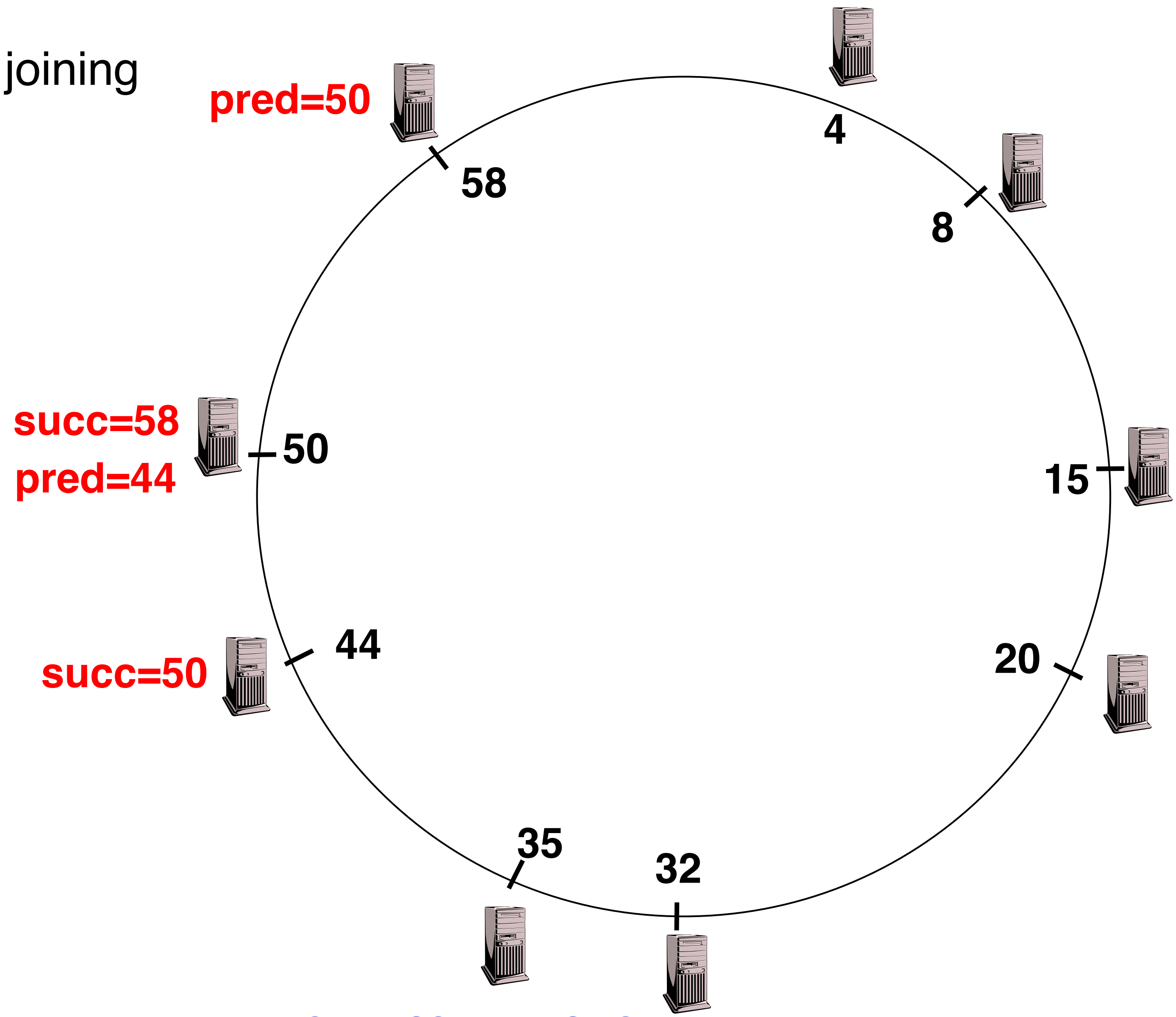
- n=50 processes notify(44)
  - pred = nil
- n=50 sets pred=44




```
n.notify(n')
  if (pred = nil or n'∈(pred, n))
     pred = n'
```

Joining Operation (cont'd)

- This completes the joining operation!

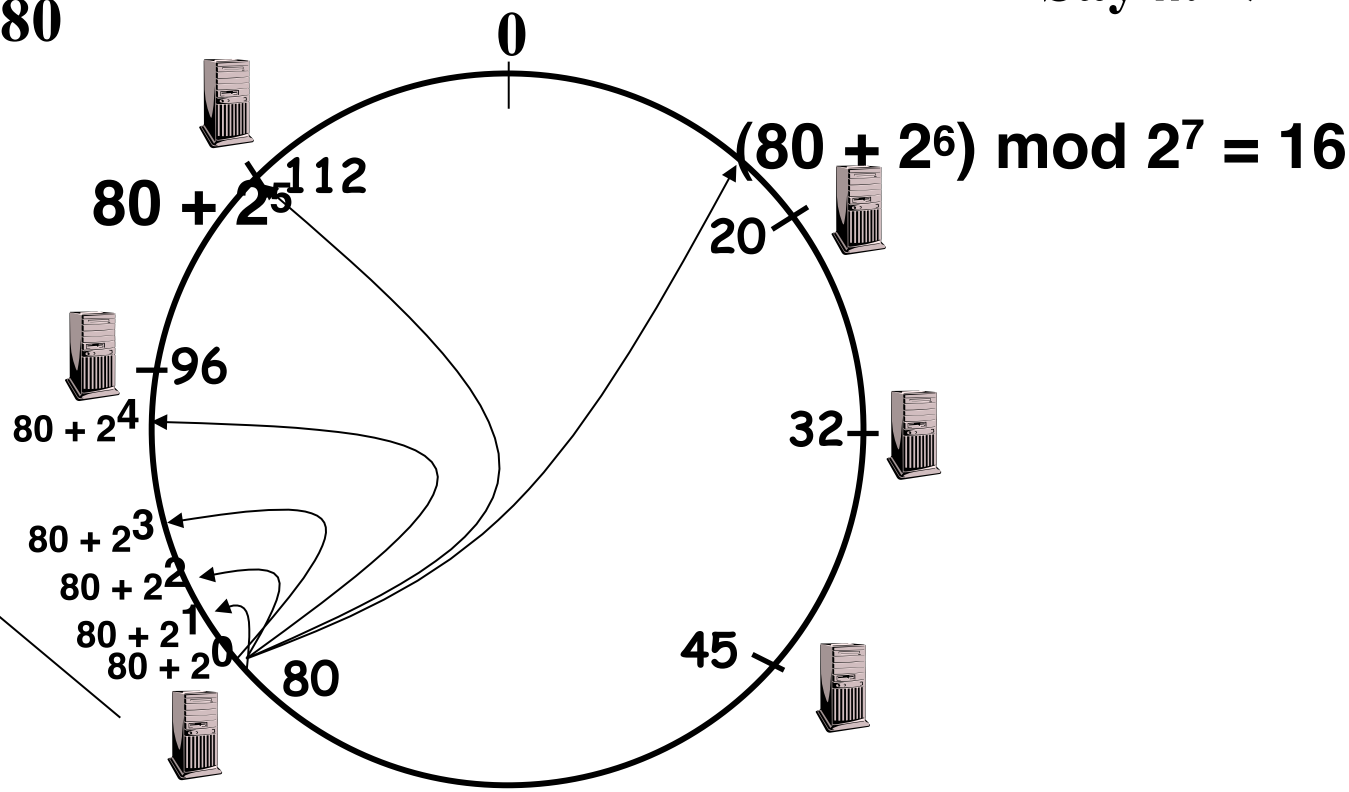


Achieving Efficiency: *finger tables*

Say $m=7$

Finger Table at 80

i	$ft[i]$
0	96
1	96
2	96
3	96
4	96
5	112
6	20



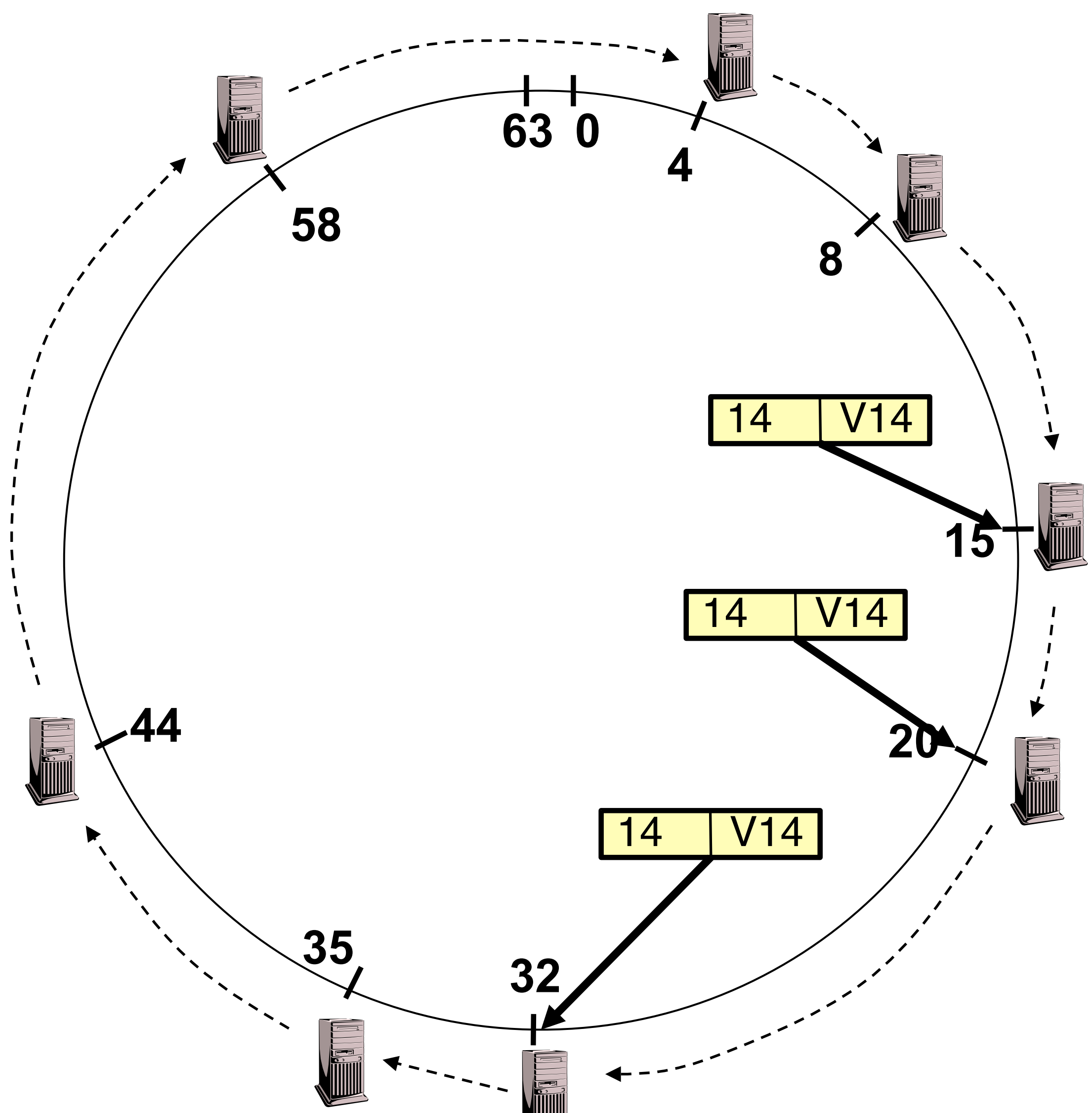
ith entry at peer with id n is first peer with id >= $n + 2^i \pmod{2^m}$

Achieving Fault Tolerance for Lookup Service

- To improve robustness each node maintains the k (> 1) immediate successors instead of only one successor
- In the pred() reply message, node A can send its k-1 successors to its predecessor B
- Upon receiving pred() message, B can update its successor list by concatenating the successor list received from A with its own list
- If k = log(M), lookup operation works with high probability even if half of nodes fail, where M is number of nodes in the system

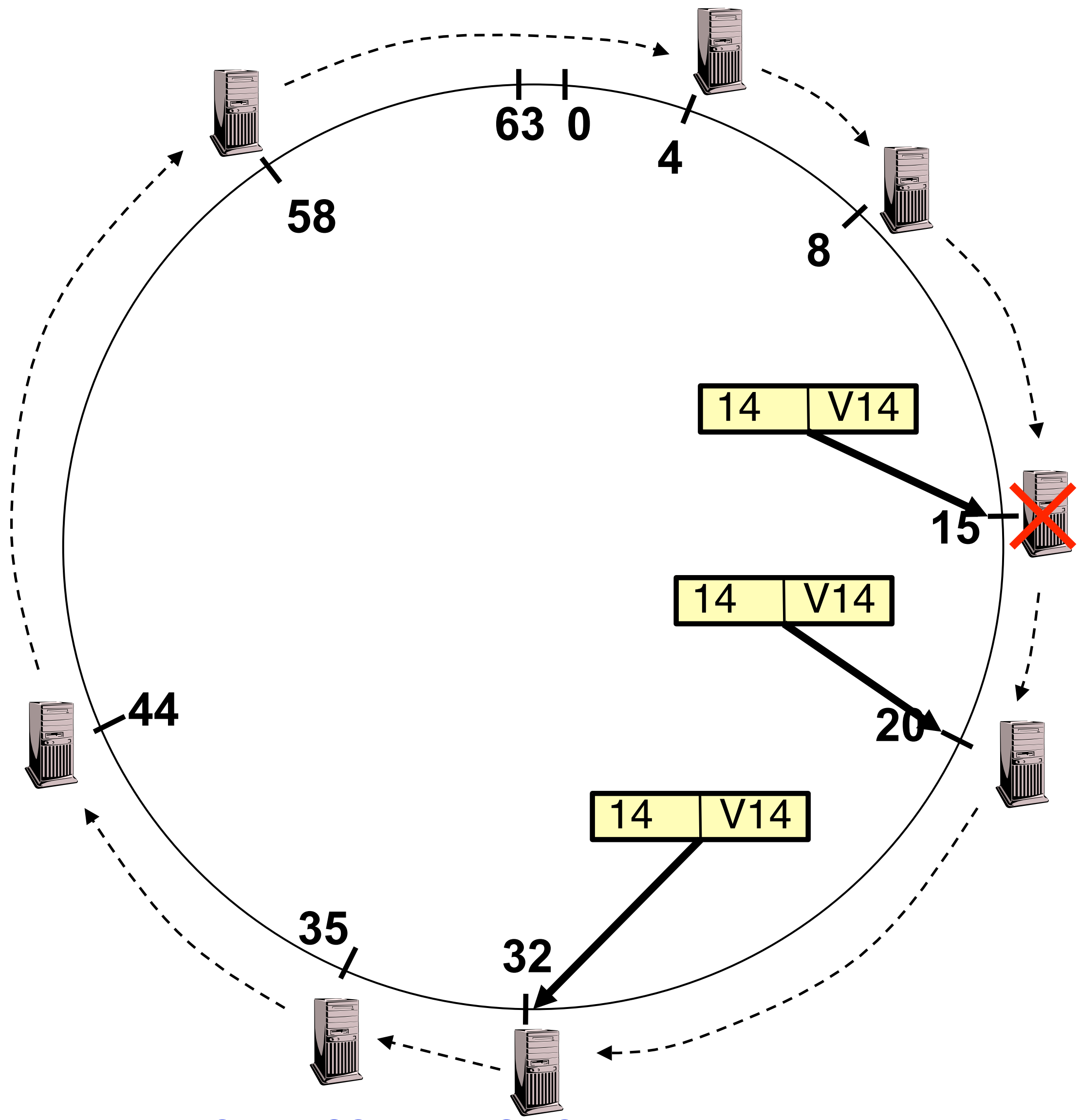
Storage Fault Tolerance

- Replicate tuples on successor nodes
- Example: replicate (K14, V14) on nodes 20 and 32

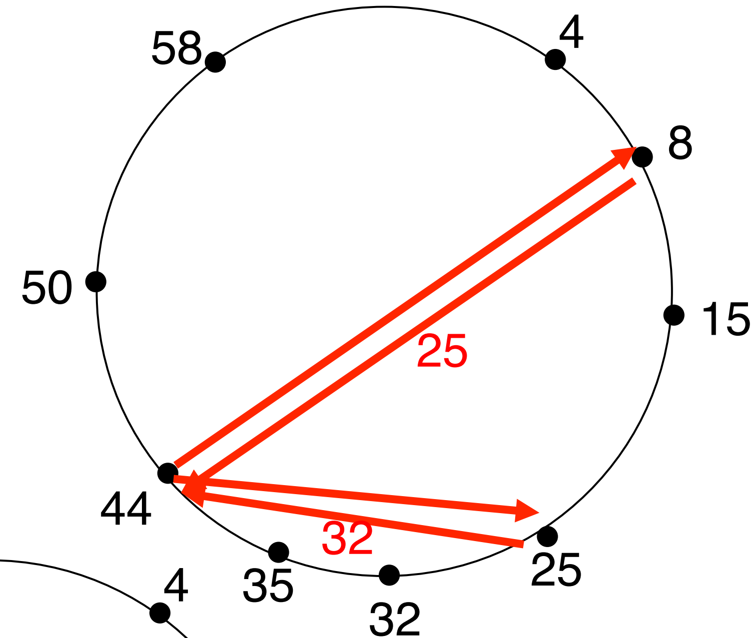


Storage Fault Tolerance

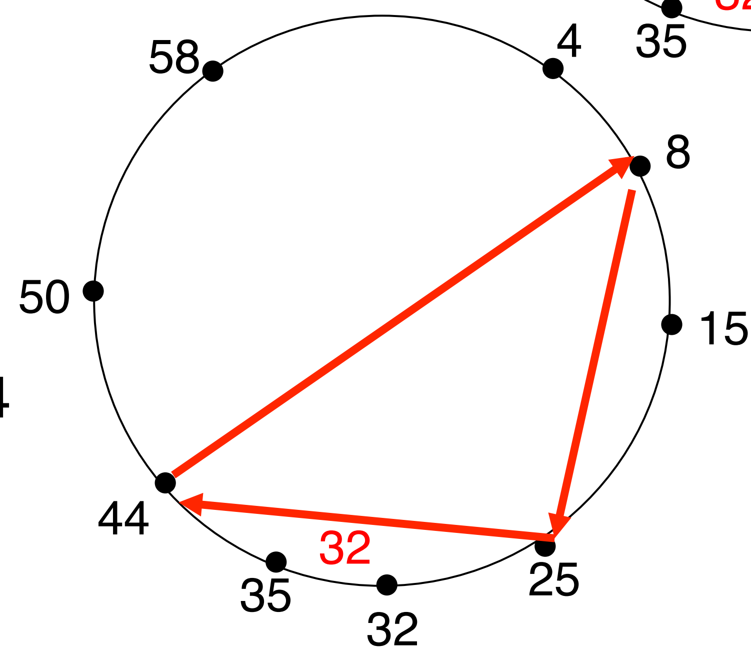
- If node 15 fails, no reconfiguration needed
  - Still have two replicas
  - All lookups will be correctly routed
- Will need to add a new replica on node 35



Iterative vs. Recursive Lookup



- Iteratively:
  - Example: node 44 issue query(31)



- Recursively
  - Example: node 44 issue query(31)

Conclusions: Key Value Store

- Very large scale storage systems
- Two operations
  - put(key, value)
  - value = get(key)
- Challenges
  - Fault Tolerance → replication
  - Scalability → serve get()'s in parallel; replicate/cache hot tuples
  - Consistency → quorum consensus to improve put() performance

Conclusions: Chord

- Highly scalable distributed lookup protocol
- Each node needs to know about O(log(*M*)), where *m* is the total number of nodes
- Guarantees that a tuple is found in O(log(*M*)) steps
- Highly resilient: works with high probability even if half of nodes fail

Project 3 (Single Node K/V Store) You are expected to learn

- Networking concepts
- Using synchronization primitives
- How to use threading in Java
- Cache replacement policies
- Message formats (XML)
- Using EC2

Project 3 Parts

- Set up EC2 + Simple network echo program
- XML Parsing and data marshalling
- Create a client for request generation
- Implement a ThreadPool
- Create an LRU Cache
- Putting it all together: Create a K/V Server with caching and asynchronous data servicing

5min Break

Networking: This Lecture's Goals

- What is a protocol?
- Layering

Many slides generated from my lecture notes by Vern Paxson, and Scott Shenker.

What Is A Protocol?

- A protocol is an agreement on how to communicate
- Includes
  - Syntax: how a communication is specified & structured
    - » Format, order messages are sent and received
  - Semantics: what a communication means
    - » Actions taken when transmitting, receiving, or when a timer expires

Examples of Protocols in Human Interactions

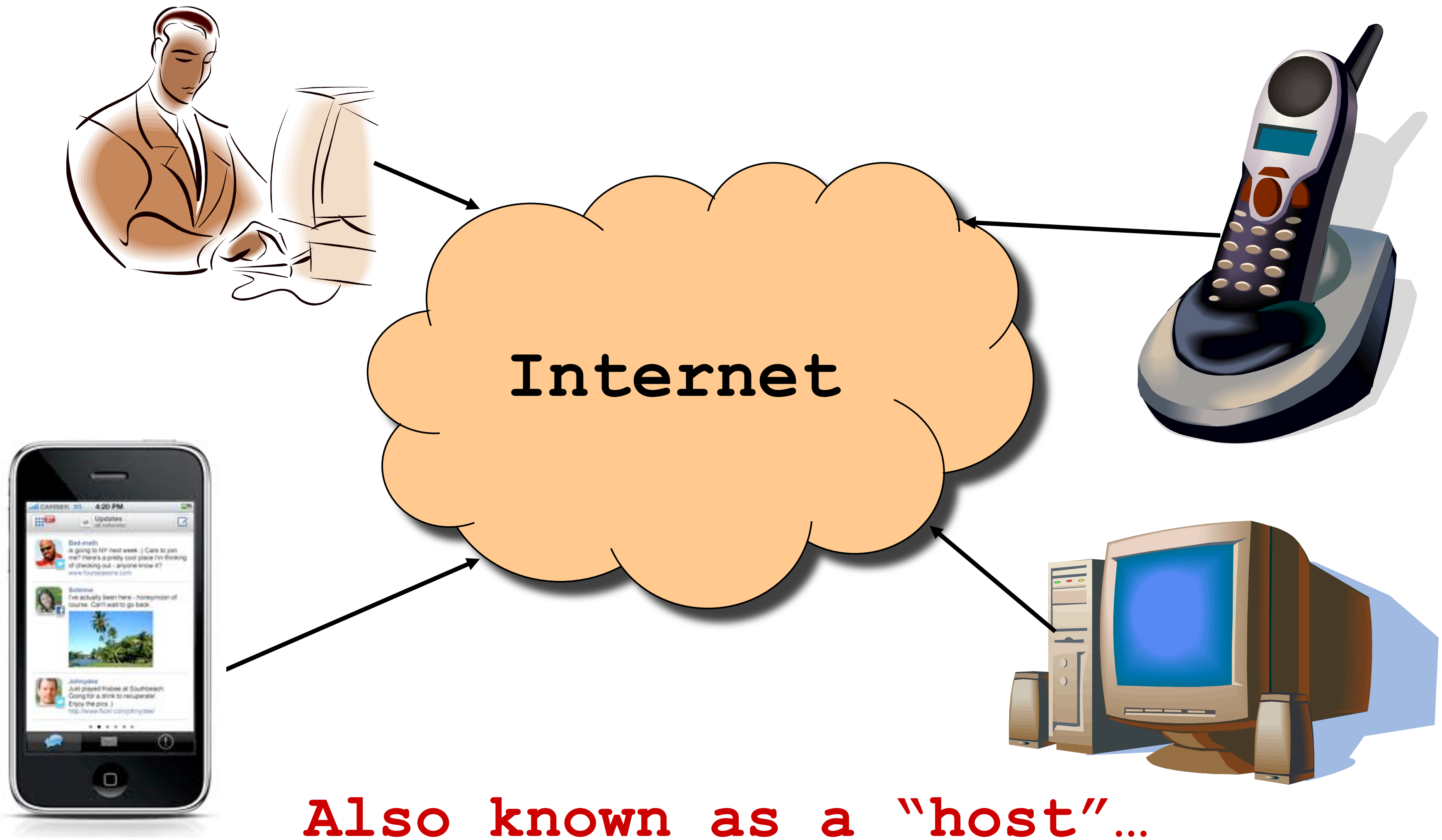
- Telephone
  1. (Pick up / open up the phone.)
  2. Listen for a dial tone / see that you have service.
  3. Dial
  4. Should hear ringing …
  5. Callee: "Hello?"
  6. Caller: "Hi, it's Alice …."
     Or: "Hi, it's me" (← what's *that* about?)
  7. Caller: "Hey, do you think … blah blah blah …" **pause**
  8. Callee: "Yeah, blah blah blah …" **pause**
  9. Caller: Bye
  10. Callee: Bye
  11. Hang up

Examples of Protocols in Human Interactions

- Asking a question
  1. Raise your hand.
  2. Wait to be called on.

  3. Or: wait for speaker to **pause** and vocalize

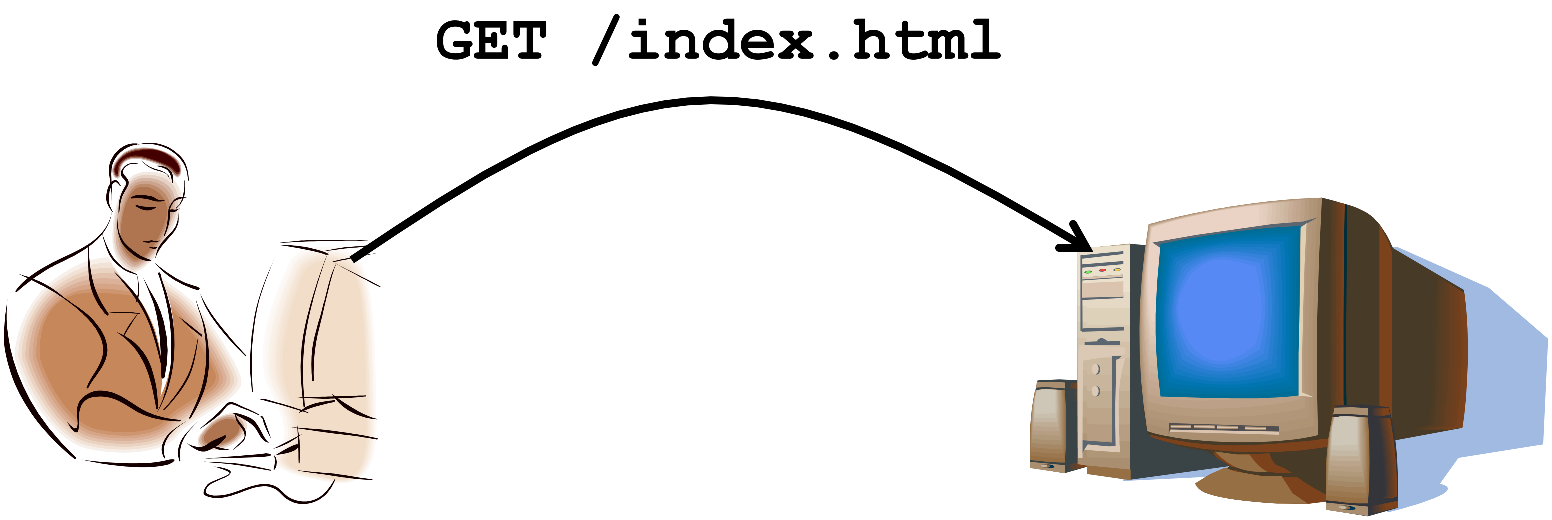
End System: Computer on the ‘Net



Also known as a “host”…

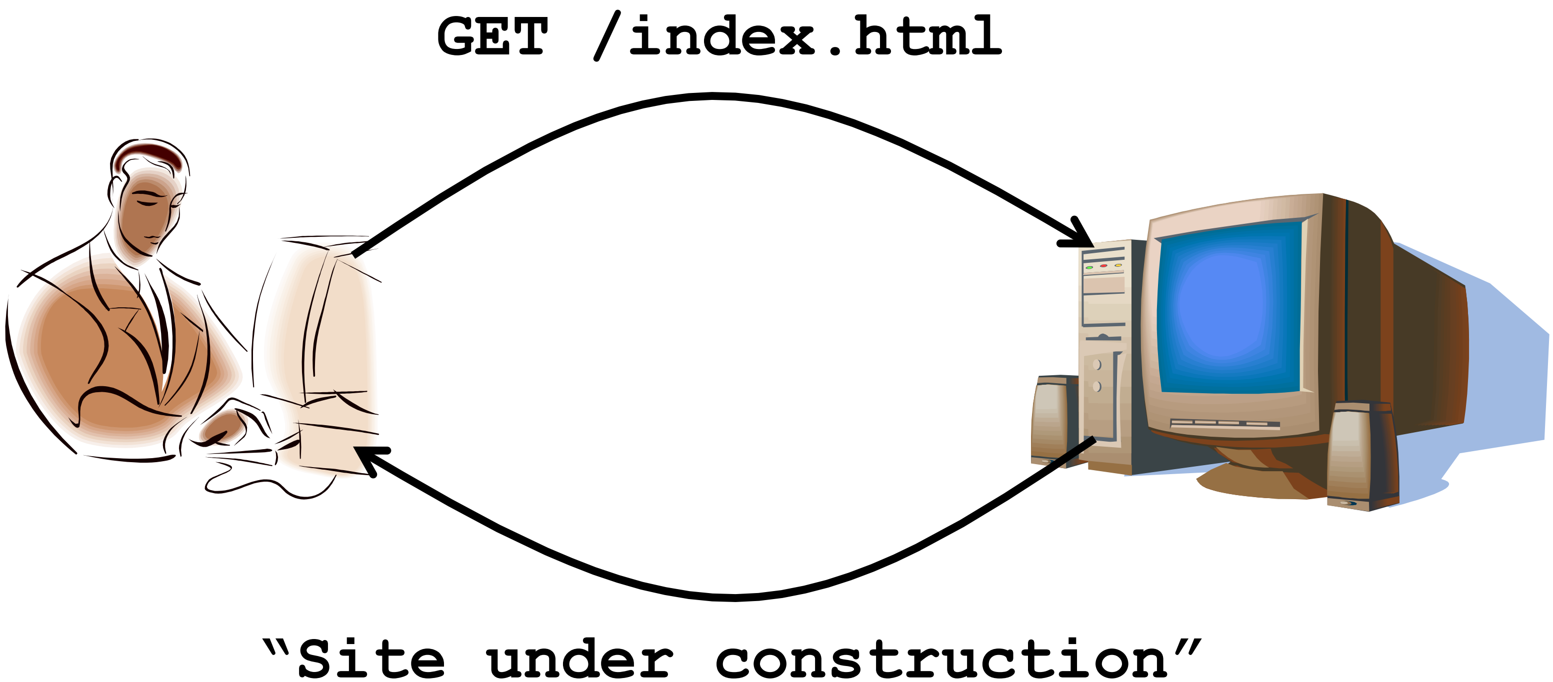
Clients and Servers

- Client program
  - Running on end host
  - Requests service
  - E.g., Web browser



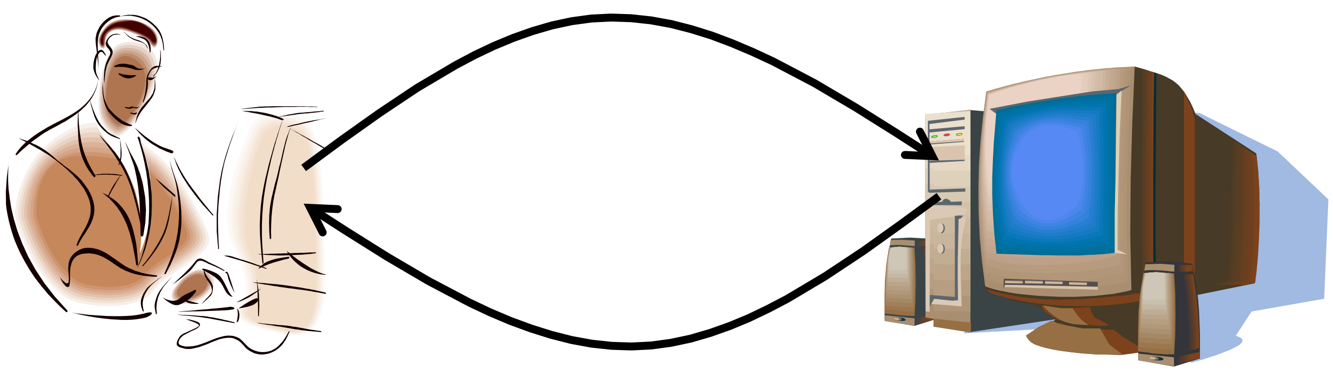
Clients and Servers

- Client program
  - Running on end host
  - Requests service
  - E.g., Web browser
- Server program
  - Running on end host
  - Provides service
  - E.g., Web server



Client-Server Communication

- Client "sometimes on"
  - Initiates a request to the server when interested
  - E.g., Web browser on your laptop or cell phone
  - Doesn't communicate directly with other clients
  - Needs to know the server's address
- Server is "always on"
  - Services requests from many client hosts
  - E.g., Web server for the *www.cnn.com* Web site
  - Doesn't initiate contact with the clients
  - Needs a fixed, well-known address



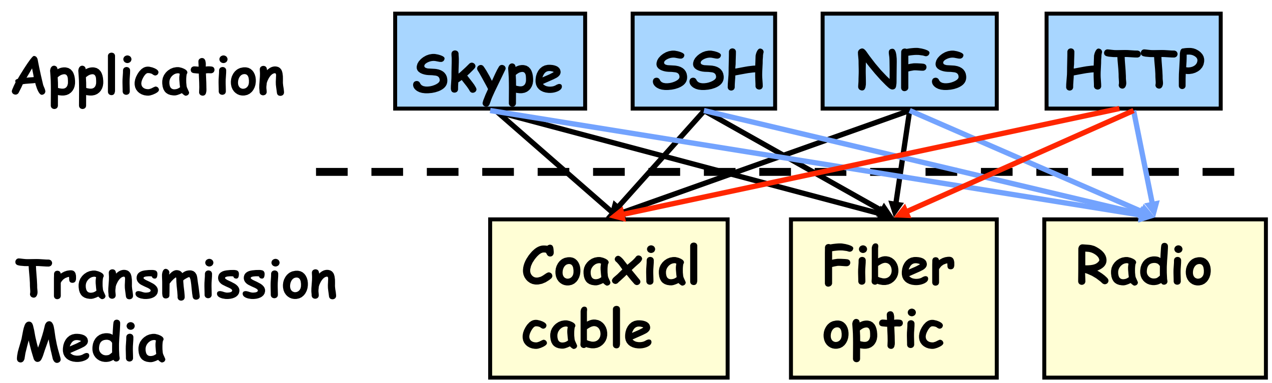
Peer-to-Peer Communication

- Not always-on server at the center of it all
  - Hosts can come and go, and change addresses
  - Hosts may have a different address each time
- Example: peer-to-peer file sharing
  - Any host can request files, send files, query to find where a file is located, respond to queries, and forward queries
  - Scalability by harnessing millions of peers
  - Each peer acting as both a client and server

The Problem

- Many different applications
  - email, web, P2P, etc.
- Many different network styles and technologies
  - Wireless vs. wired vs. optical, etc.
- How do we organize this mess?

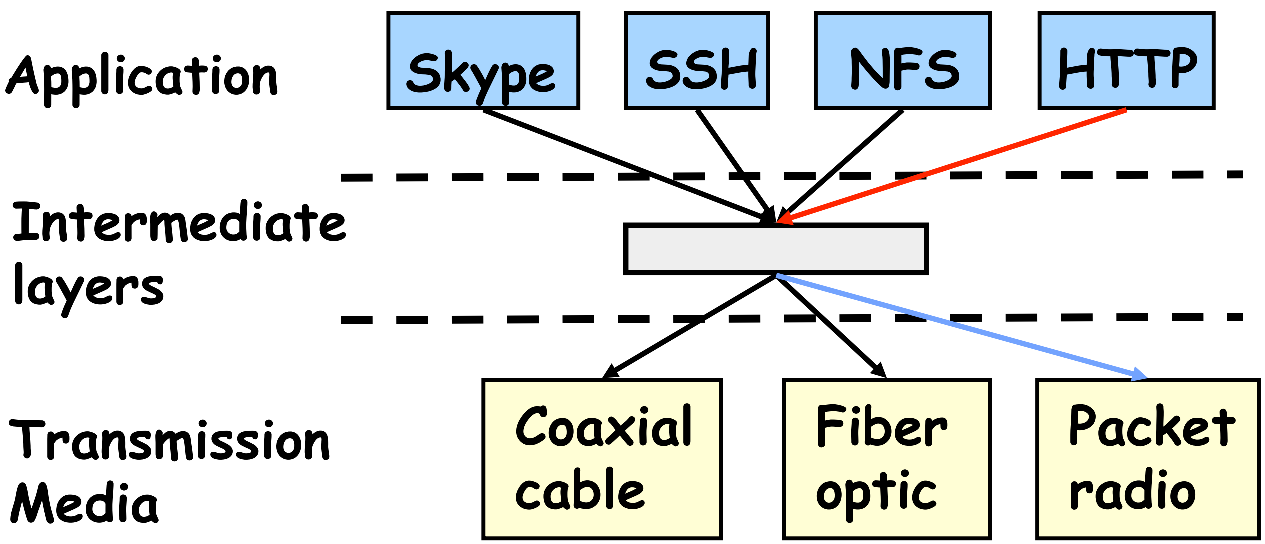
The Problem (cont'd)



- Re-implement every application for every technology?
- No! But how does the Internet design avoid this?

Solution: Intermediate Layers

- Introduce intermediate layers that provide set of abstractions for various network functionality & technologies
  - A new app/media implemented only once
  - Variation on "add another level of indirection"



Software System Modularity

Partition system into modules & abstractions:

- Well-defined interfaces give flexibility
  - ***Hides*** implementation - thus, it can be freely changed
  - Extend functionality of system by adding new modules
- E.g., libraries encapsulating set of functionality
- E.g., programming language + compiler abstracts away not only how the particular CPU works …
  - … but also the basic computational model
- Well-defined interfaces hide information
  - Isolate assumptions
  - Present high-level abstractions
  - **But can impair performance**

Network System Modularity

Like software modularity, but:

- Implementation distributed across many machines (routers and hosts)
- Must decide:
  - How to break system into modules
    - **Layering**
  - What functionality does each module implement
    - **End-to-End Principle**
- We will address these choices next lecture

Layering: A Modular Approach

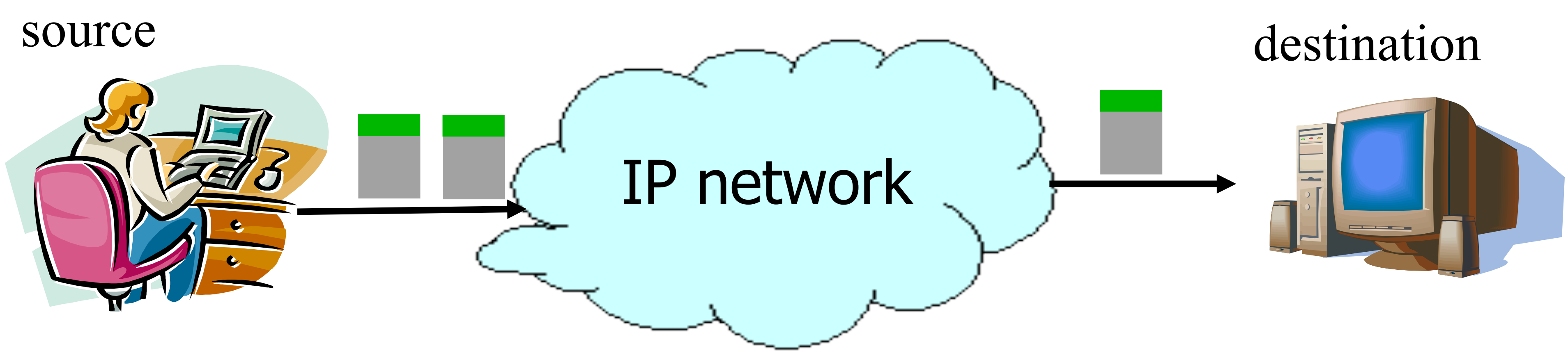
- Partition the system
  - Each layer solely relies on services from layer below
  - Each layer solely exports services to layer above
- Interface between layers defines interaction
  - Hides implementation details
  - Layers can change without disturbing other layers

Protocol Standardization

- Ensure communicating hosts speak the same protocol
  - Standardization to enable multiple implementations
  - Or, the same folks have to write all the software
- Standardization: Internet Engineering Task Force
  - Based on working groups that focus on specific issues
  - Produces "Request For Comments" (RFCs)
    - Promoted to standards via rough consensus and running code
  - IETF Web site is ***http://www.ietf.org***
  - RFCs archived at ***http://www.rfc-editor.org***
- De facto standards: same folks writing the code
  - P2P file sharing, Skype, <your protocol here>…

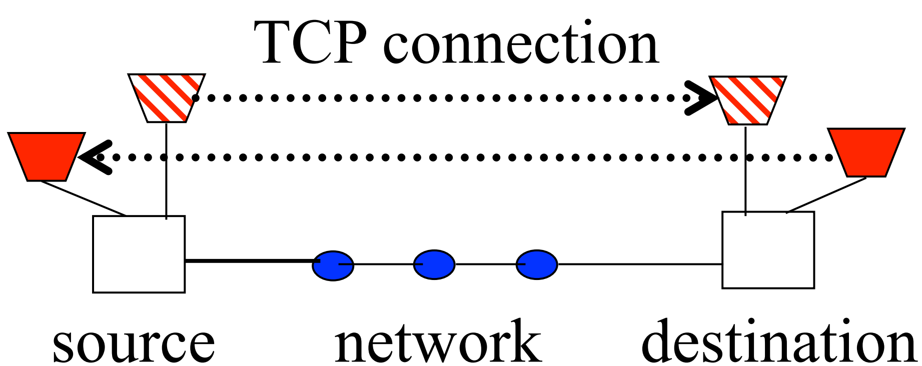
Example: The Internet Protocol (IP): "Best-Effort" Packet Delivery

- Datagram packet switching
  – Send data in packets
  – Header with source & destination address
- Service it provides:
  – Packets may be lost
  – Packets may be corrupted
  – Packets may be delivered out of order



Example: Transmission Control Protocol (TCP)

- Communication service
  - Ordered, reliable byte stream
  - Simultaneous transmission in both directions
- Key mechanisms at end hosts
  - Retransmit lost and corrupted packets
  - Discard duplicate packets and put packets in order
  - Flow control to avoid overloading the receiver buffer
  - Congestion control to adapt sending rate to network load



Summary

- Roles of
  - Standardization
  - Clients, servers, peer-to-peer
- Layered architecture as a powerful means for organizing complex networks
  - Though layering has its drawbacks too
- Next lecture
  - Layering
  - End-to-end arguments