Introduction to Computer Networks

COSC 4377

Lecture 6

Spring 2012

February 6, 2012

Announcements

- HW3 due this week
- Start working on HW4
- In-class student presentations
- No TA office hours this week
  - Makeup hours next week
- Fast HTTP transfer competition

Student presentations

Topic	Date
FTP vs HTTP	2/8/2012
www.webpagetest.org	2/13/2012
DHCP	2/15/2012
Dynamic DNS	2/20/2012
SPDY	2/22/2012

What problem does it solve?

How does it work?

What are its limitations?

Today's Topics

- Peer to Peer (P2P) Networks
- Transport Layer

Structured P2P systems

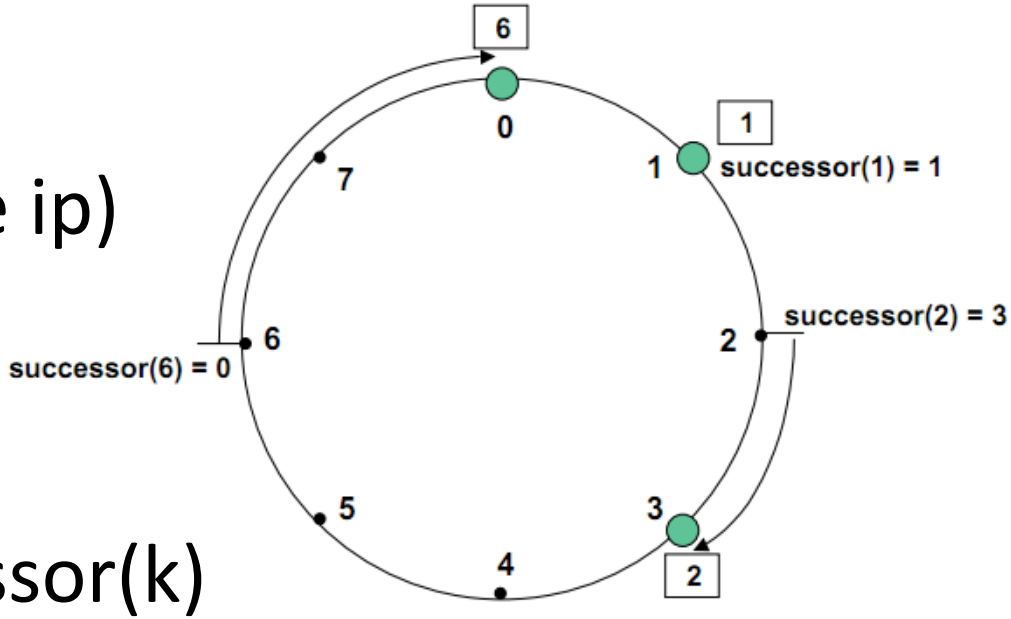
- Distributed Hash Table (DHT)
  - Efficient Key, value storage
  - Approach: map the id to a host
- Challenges
  - Scale to millions of nodes
  - Churn
  - Heterogeneity

DHTs

- IDs from a *flat* namespace
  - Contrast with hierarchical IP, DNS
- Metaphor: hash table, but distributed
- Interface
  - Get(key)
  - Put(key, value)
- How?
  - Every node supports a single operation:
    Given a *key*, route messages to node holding *key*

Consistent Hashing

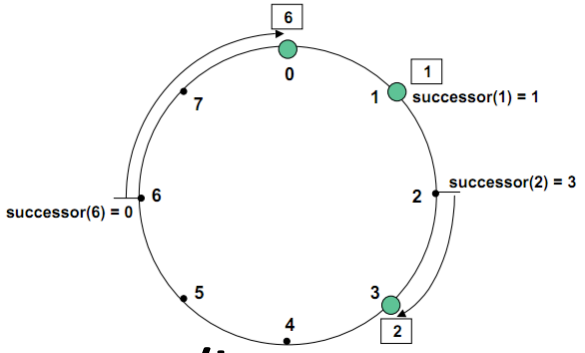
- Map keys to nodes
- nodeid = hash(node ip)



- K mapped to successor(k)
- Successor(k) = node equal to or follows K

Consistent Hashing Properties

- Designed for node join/leave with minimal churn in key mapping

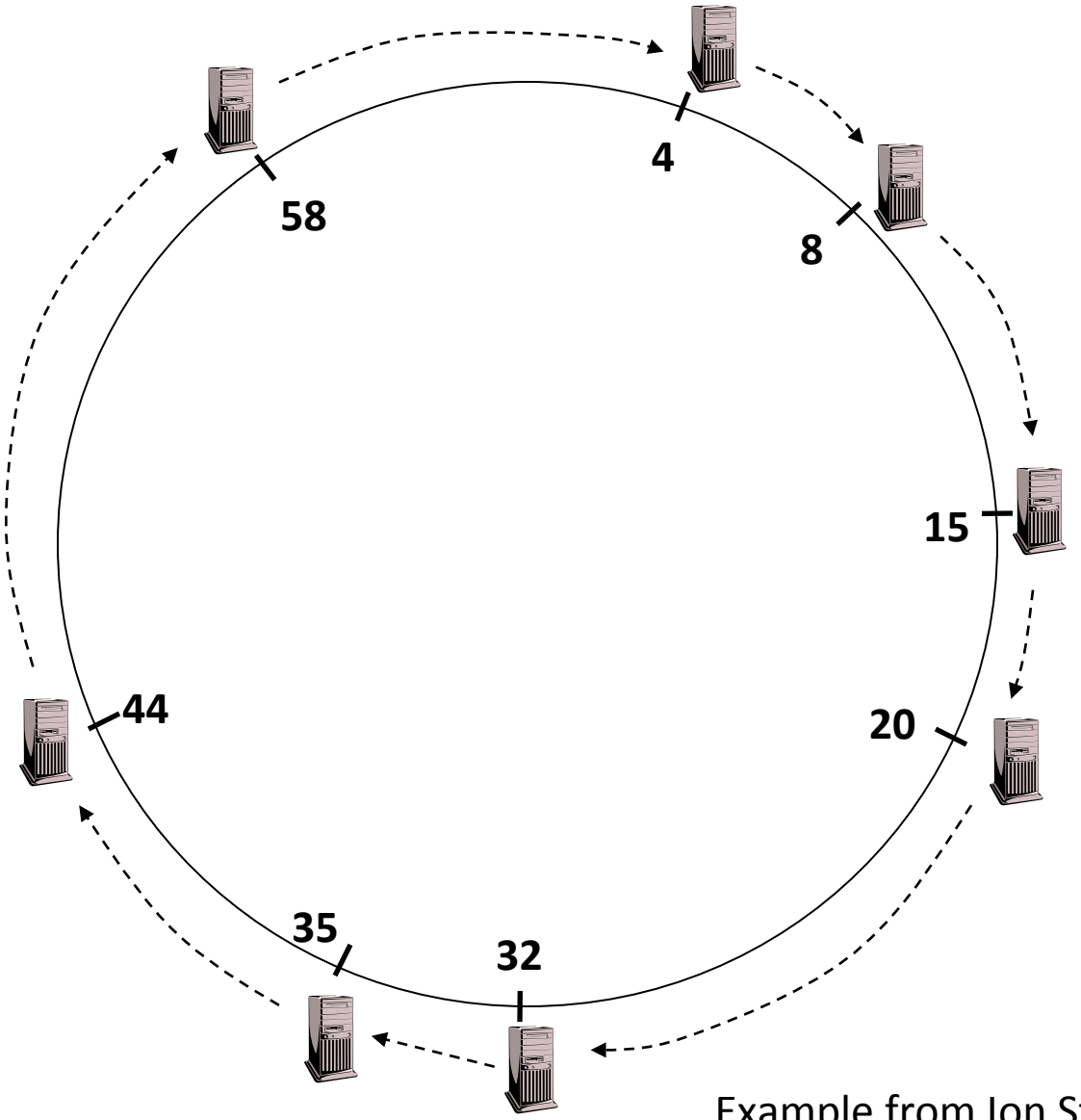


- K/N keys per node
- K/N keys change hands during join/leave

Identifier to Node Mapping Example

- Node 8 maps [5,8]
- Node 15 maps [9,15]
- Node 20 maps [16, 20]
- …
- Node 4 maps [59, 4]

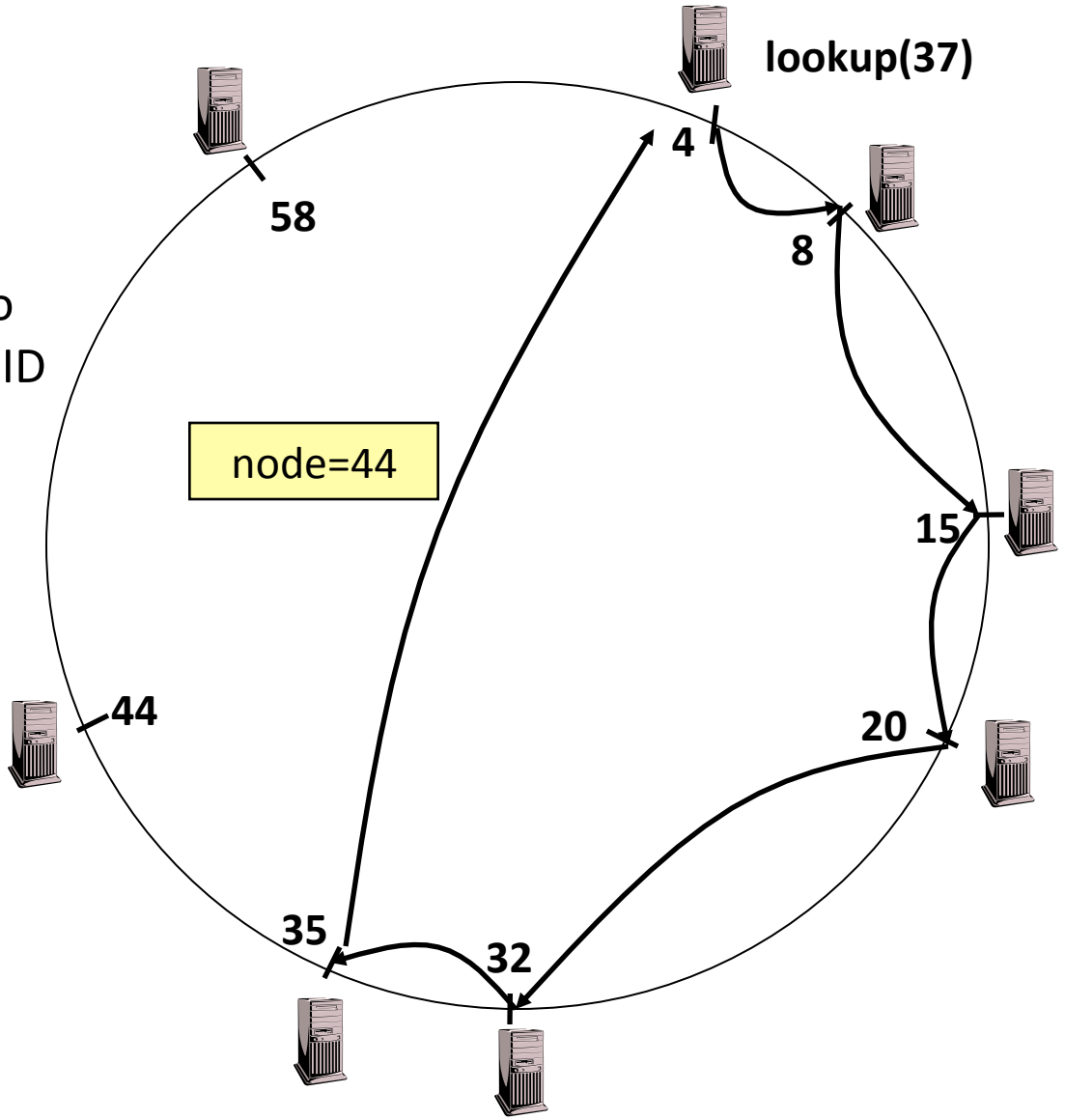
- Each node maintains a pointer to its successor



Example from Ion Stoica

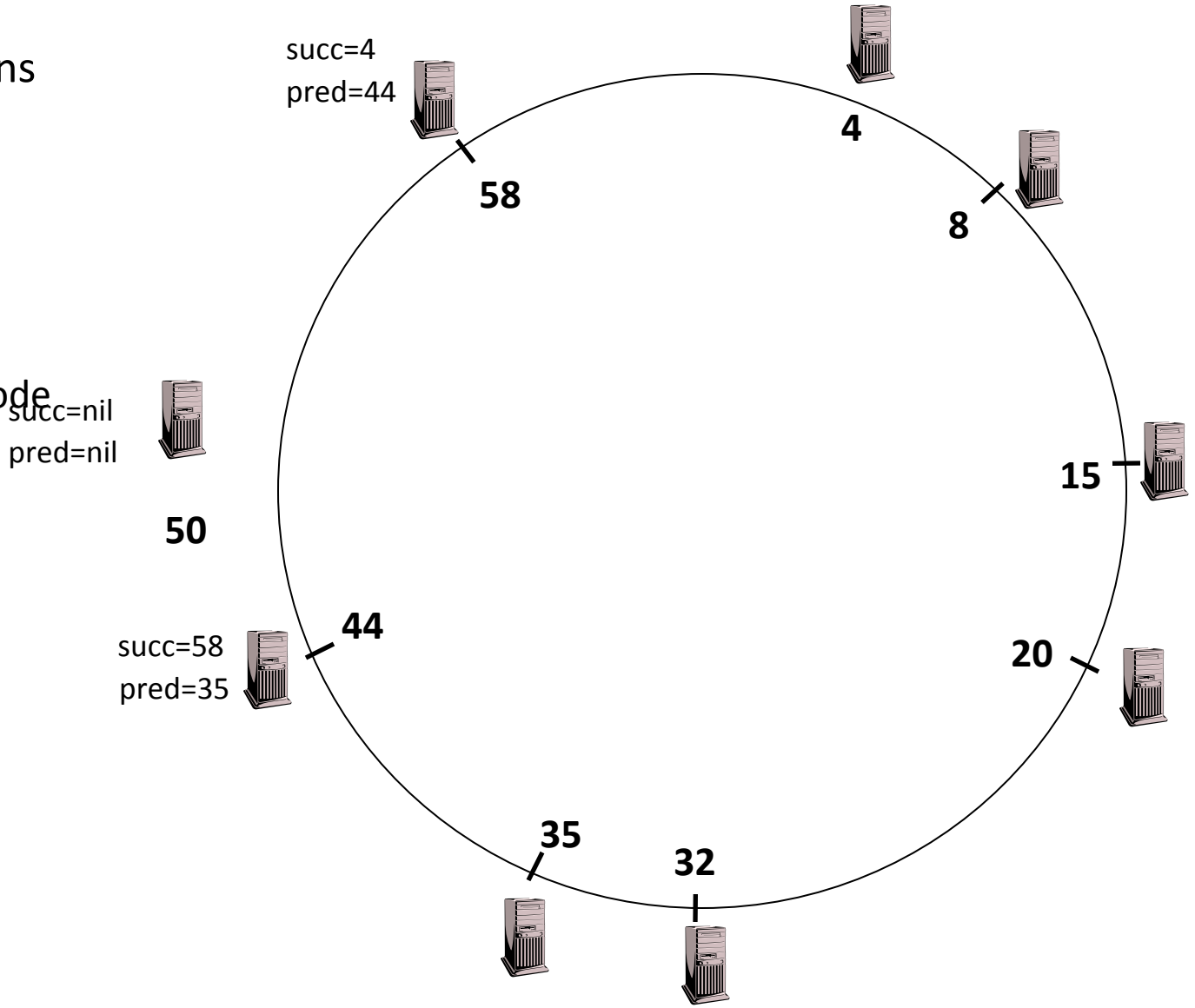
Lookup

- Each node maintains its successor
- Route packet (ID, data) to the node responsible for ID using successor pointers



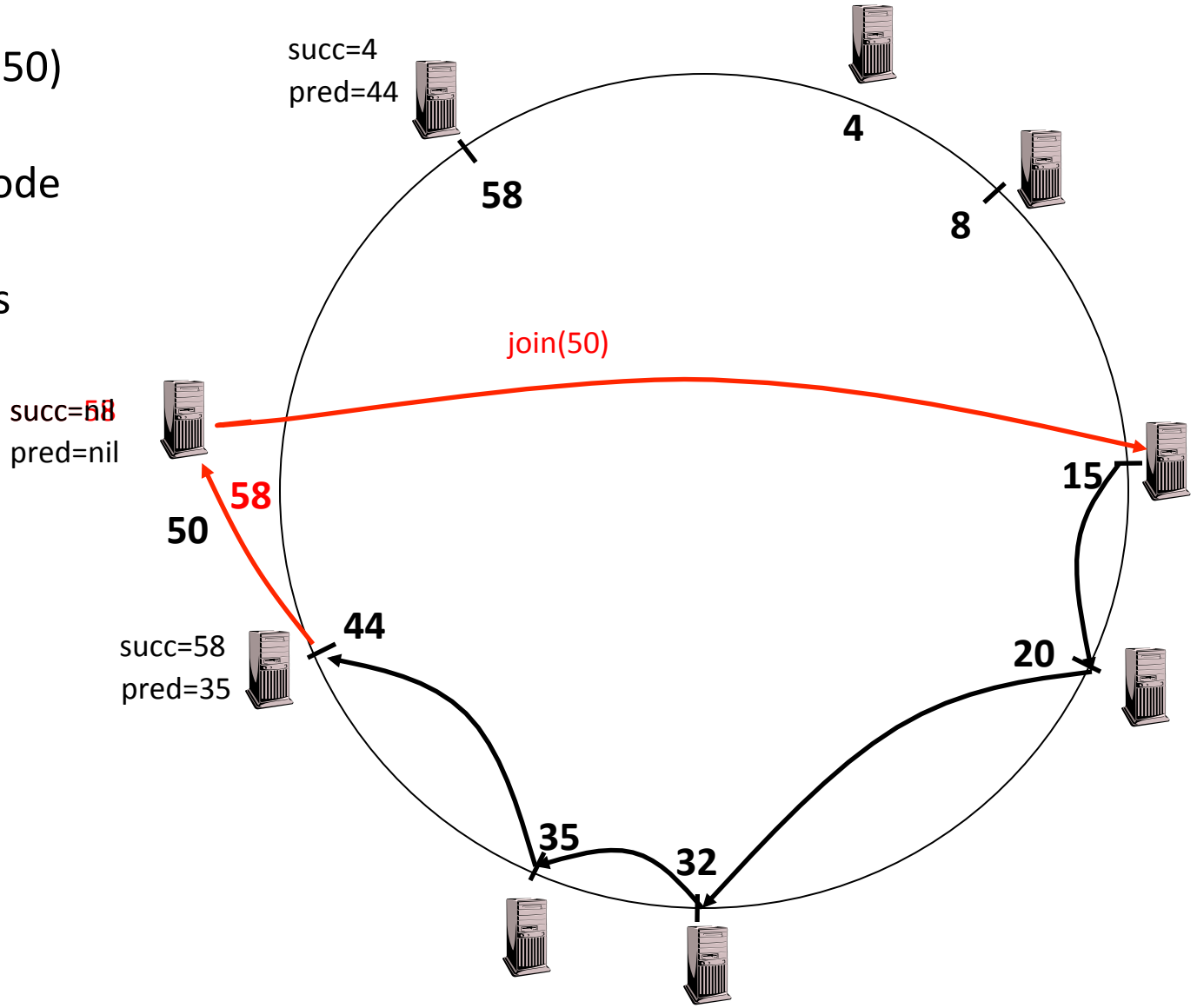
Joining Operation

- 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



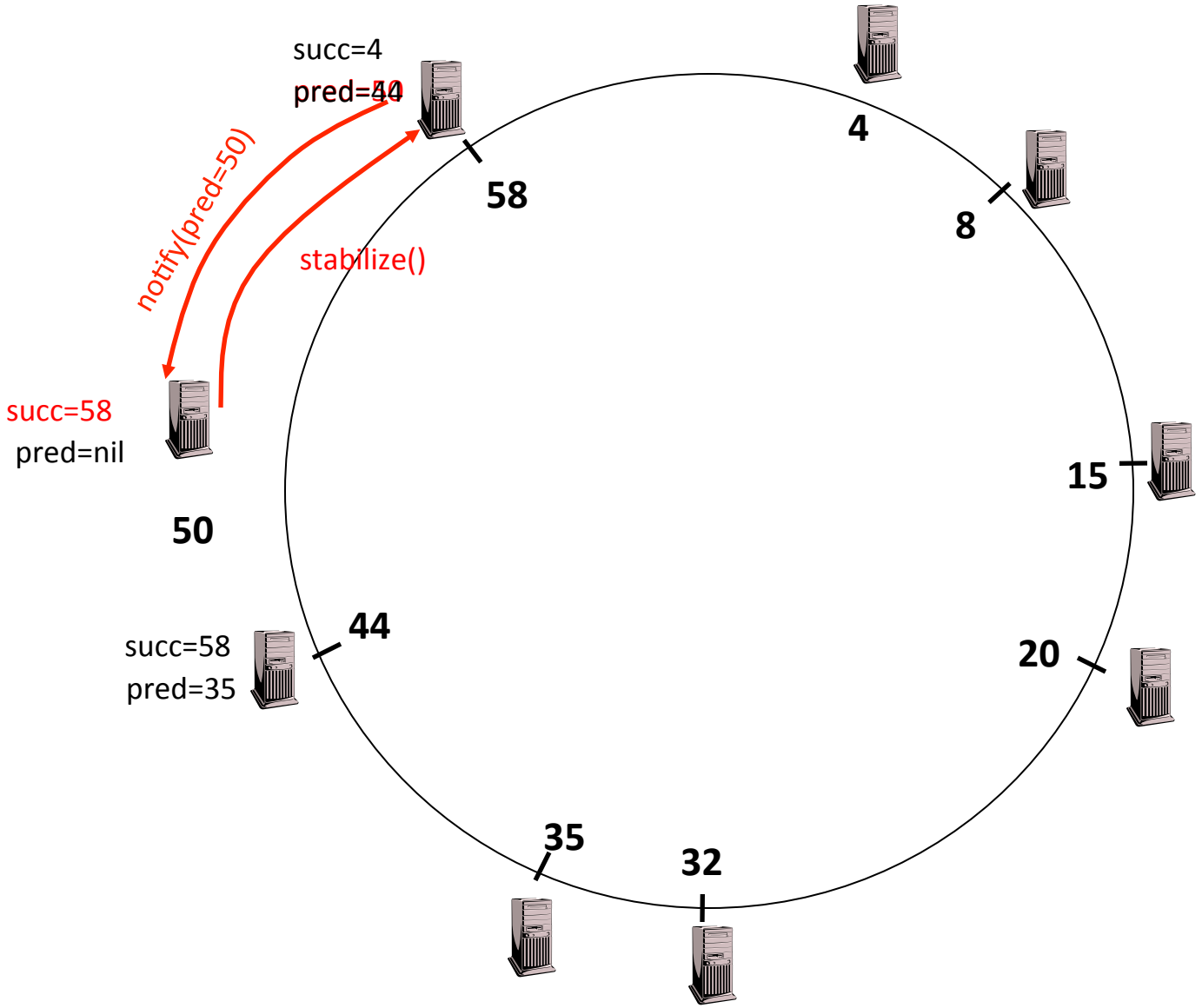
Joining Operation

- Node 50: send join(50) to node 15
- Node 44: returns node 58
- Node 50 updates its successor to 58



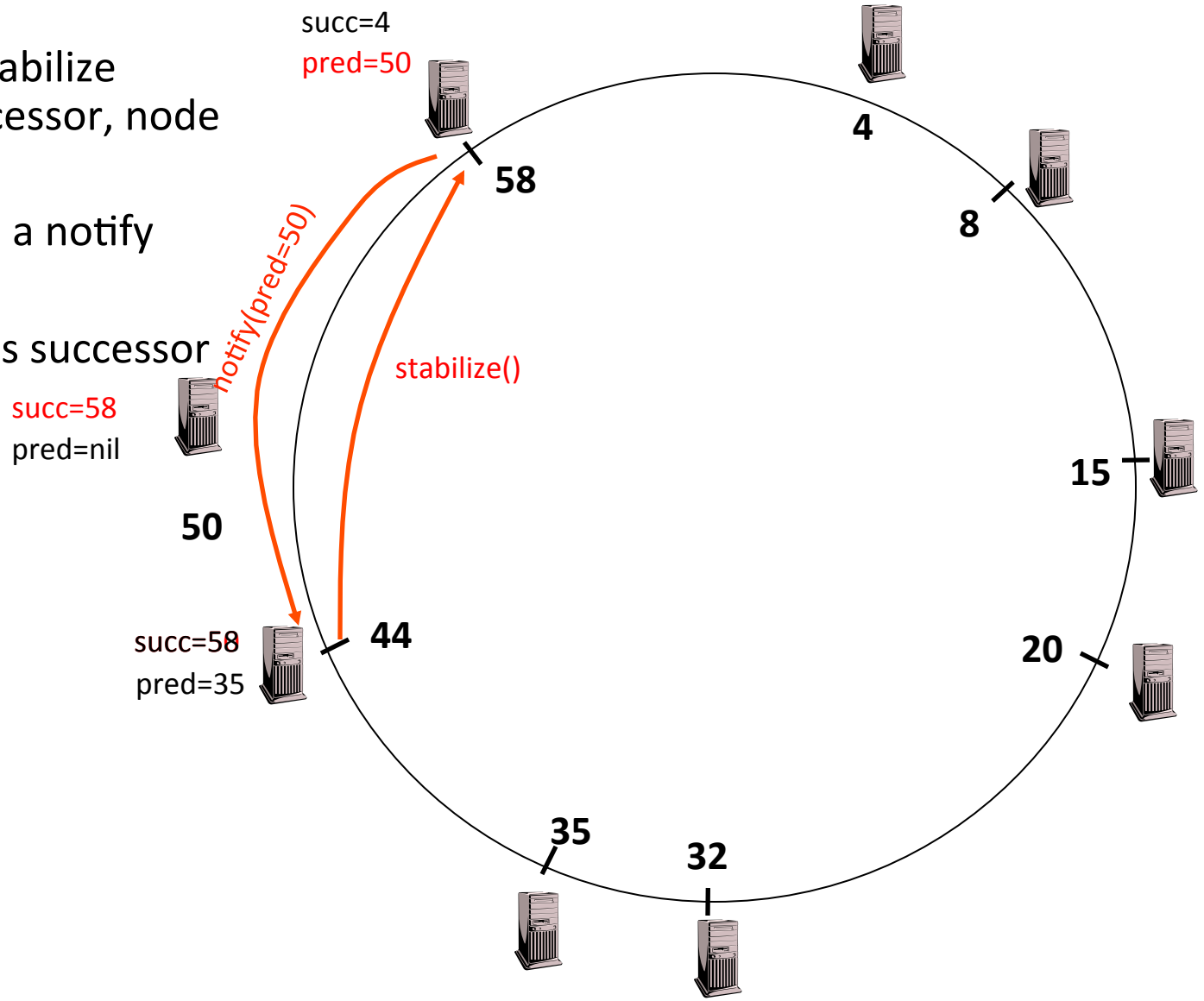
Joining Operation

- Node 50: send stabilize() to node 58
- Node 58:
  - update predecessor to 50
  - send notify() back



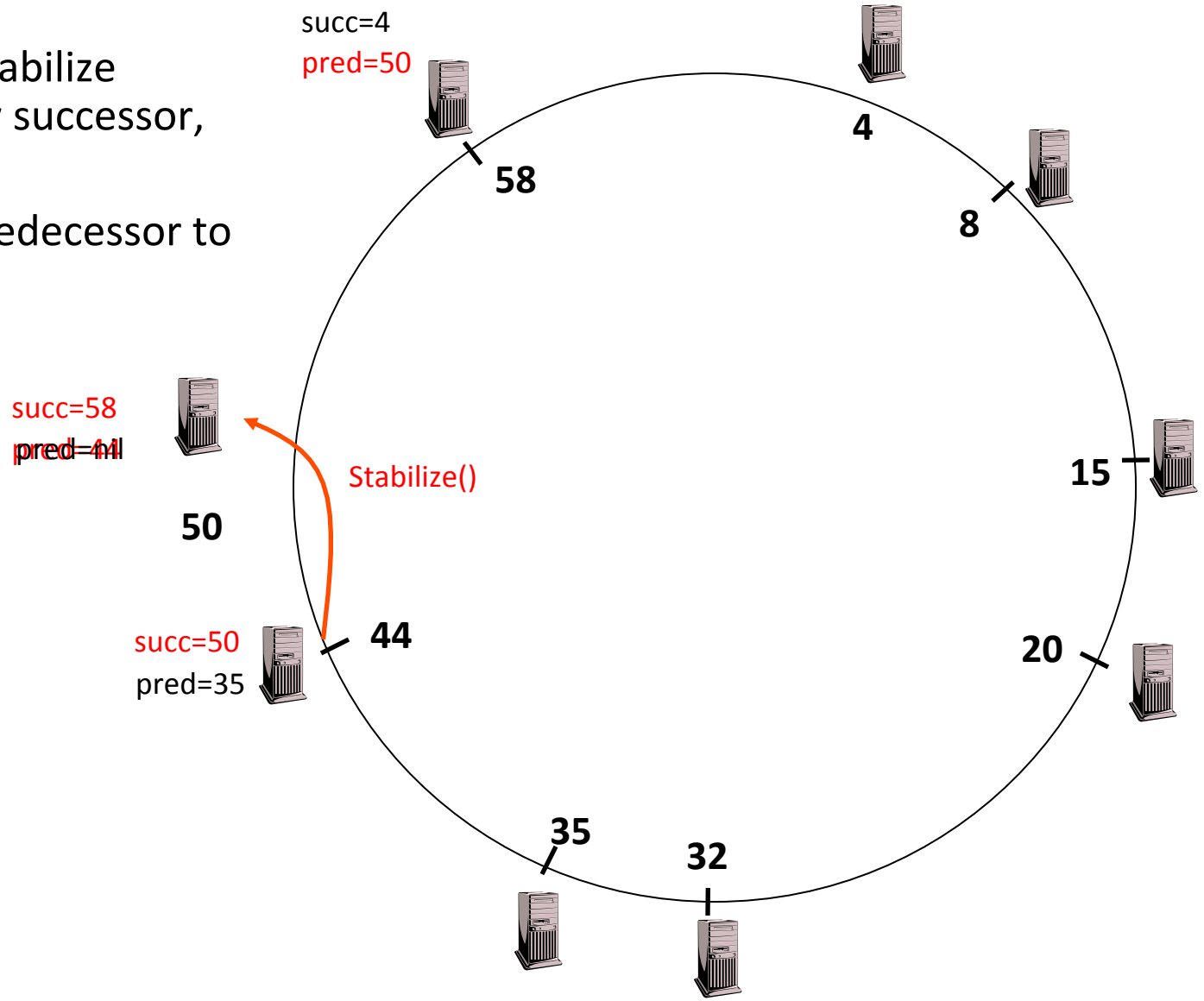
Joining Operation (cont'd)

- Node 44 sends a stabilize message to its successor, node 58
- Node 58 reply with a notify message
- Node 44 updates its successor to 50



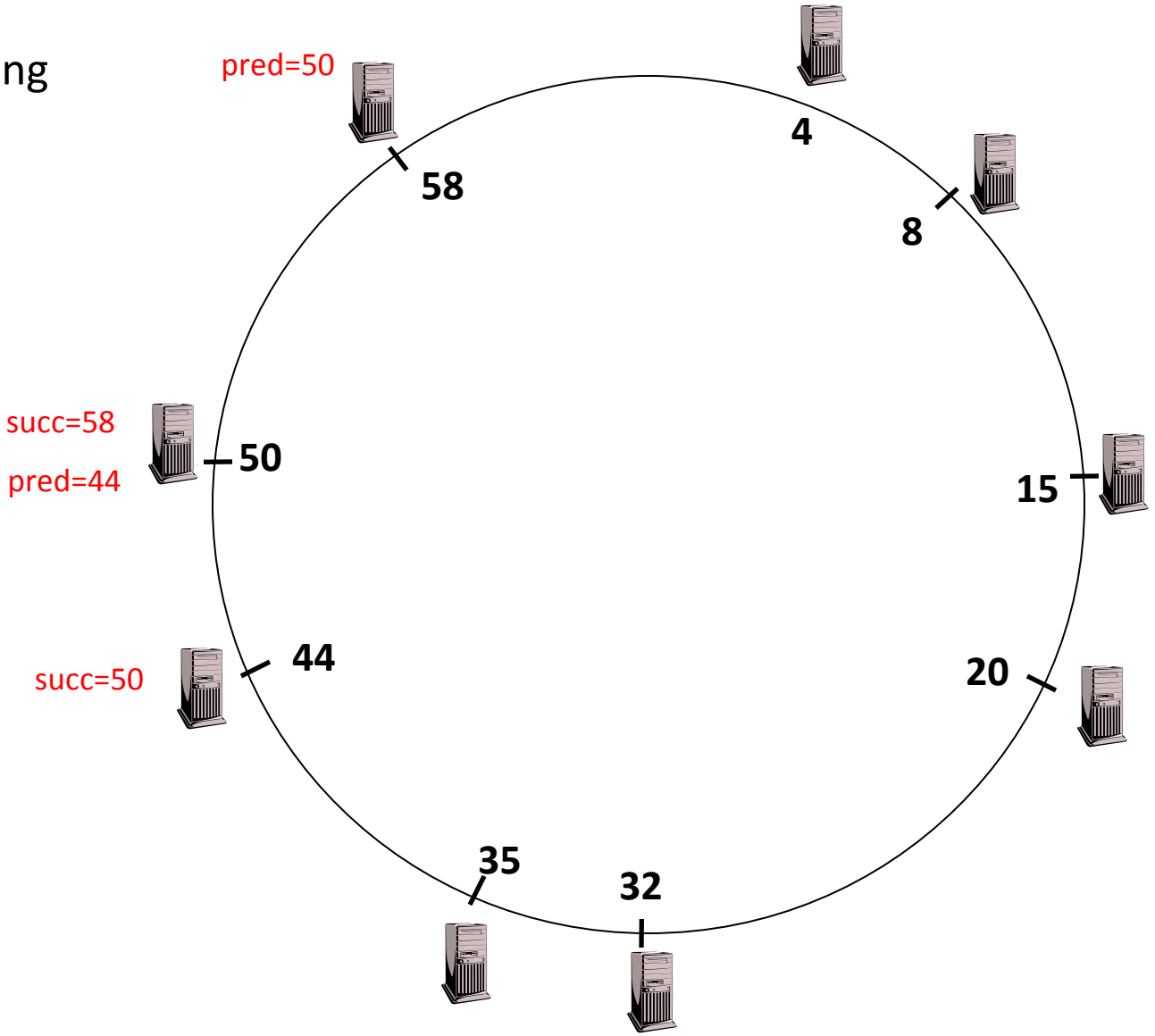
Joining Operation (cont'd)

- Node 44 sends a stabilize message to its new successor, node 50
- Node 50 sets its predecessor to node 44



Joining Operation (cont'd)

- This completes the joining operation!



Network Applications

- Centralized and Peer-to-peer architectures
- How to design and write network applications
- Case Studies
  - HTTP
  - DNS
  - P2P applications

The applications we discussed need

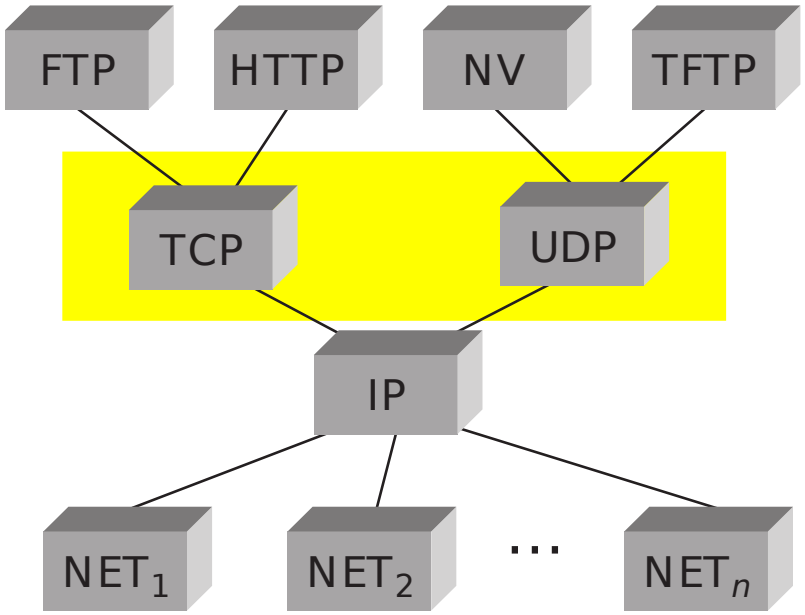
a reliable method

to send information

across the network.

Transport Layer provides that service.

Transport Layer



- Transport protocols sit on top of network layer and provide
  - Application-level multiplexing (“ports”)
  - Error detection, reliability, etc.

Error Detection

- Idea: add redundant information to catch errors in packet
- Three examples:
  - Parity
  - Internet Checksum
  - CRC

Parity Bit

- Parity
  - Can detect odd number of bit errors
  - No correction

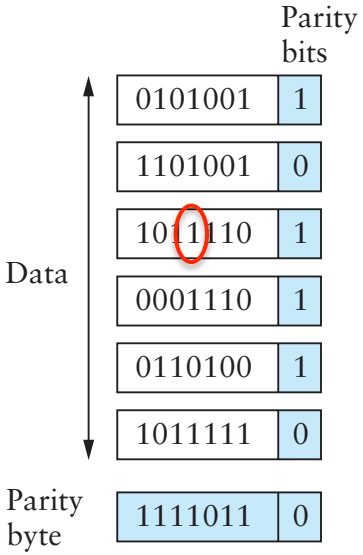
  Data: 1101101

  Even Parity: 1

  Transmit: 11011011

http://en.wikipedia.org/wiki/Parity_bit

2-D Parity



- Add 1 parity bit for each 7 bits
- Add 1 parity bit for each bit position across the frame)
  - Can correct single-bit errors
  - Can detect 2- and 3-bit errors, most 4-bit errors

Checksum

- Algorithm
  - Set Checksum field to 0
  - Sum all 16-bit words, adding any carry bits to the LSB (one's complement sum)
  - Flip bits to get checksum (one's complement)
- Transmit: Data + Checksum
- To check: sum whole packet, including sum, should get 0xffff

http://www.ietf.org/rfc/rfc1071.txt

How good is it?

- 16 bits not very long
  - Probability 1-bit error not detected?
- Checksum does catch any 1-bit error
- But not any 2-bit error
  - E.g., increment word ending in 0, decrement one ending in 1

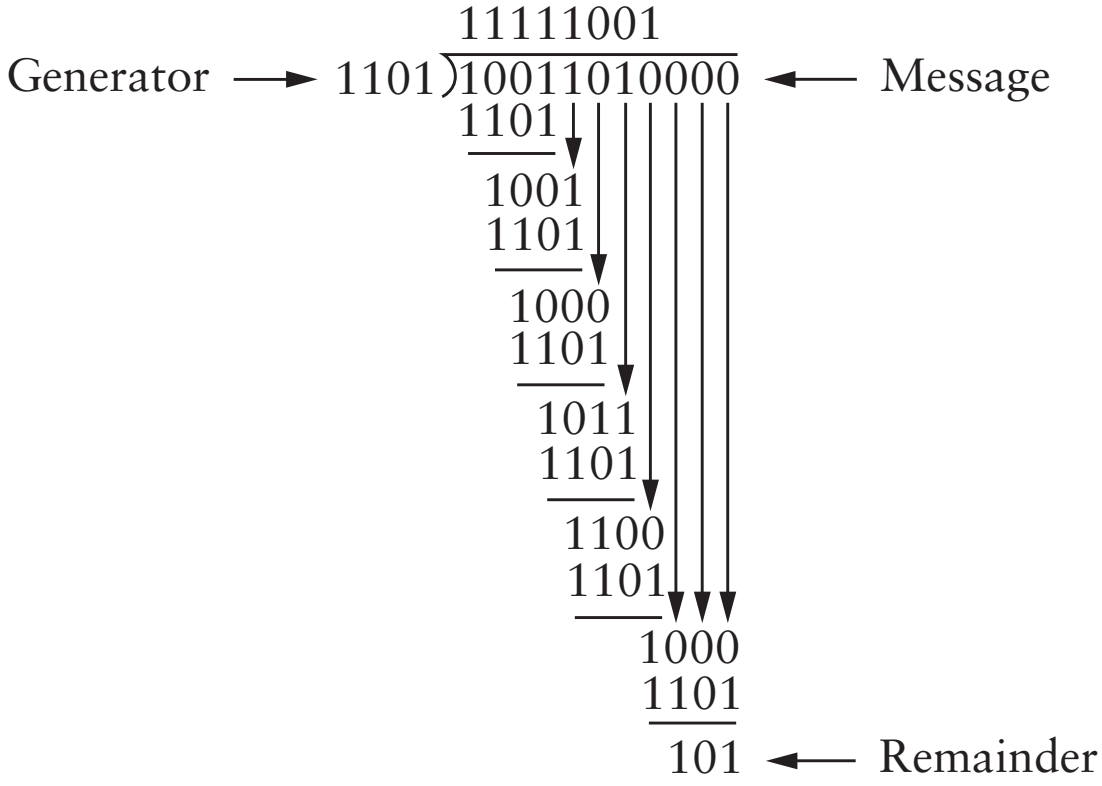
CRC – Error Detection with Polynomials

- Consider message to be a polynomial in $Z_2[x]$
  - Each bit is one coefficient
  - E.g., message 10101001 -> $m(x) = x^7 + x^5 + x^3 + 1$
- Can reduce one polynomial modulo another
  - Let $n(x) = m(x)x^3$. Let $C(x) = x^3 + x^2 + 1$
  - Find $q(x)$ and $r(x)$ s.t. $n(x) = q(x)C(x) + r(x)$ and degree of $r(x)$ < degree of $C(x)$
  - Analogous to taking 11 mod 5 = 1

http://en.wikipedia.org/wiki/Cyclic_redundancy_check

Polynomial Division Example

- Just long division, but addition/subtraction is XOR



CRC

- Select a divisor polynomial C(x), degree k
  - C(x) should be *irreducible* – not expressible as a product of two lower-degree polynomials in $Z_2[x]$
- Add k bits to message
  - Let $n(x) = m(x)x^k$ (add k 0's to m)
  - Compute $r(x) = n(x) \bmod C(x)$
  - Compute $n(x) = n(x) - r(x)$ (will be divisible by C(x)) (subtraction is XOR, just set k lowest bits to r(x)!)
- Checking CRC is easy
  - Reduce message by C(x), make sure remainder is 0

Why is this good?

- Suppose you send m(x), recipient gets m'(x)
  - E(x) = m'(x) – m(x) (all the incorrect bits)
  - If CRC passes, C(x) divides m'(x)
  - Therefore, C(x) must divide E(x)
- Choose C(x) that doesn't divide any common errors!
  - All single-bit errors caught if x^k, x^0 coefficients in C(x) are 1
  - All 2-bit errors caught if at least 3 terms in C(x)
  - Any odd number of errors if last two terms (x + 1)
  - Any error burst less than length k caught

Common CRC Polynomials

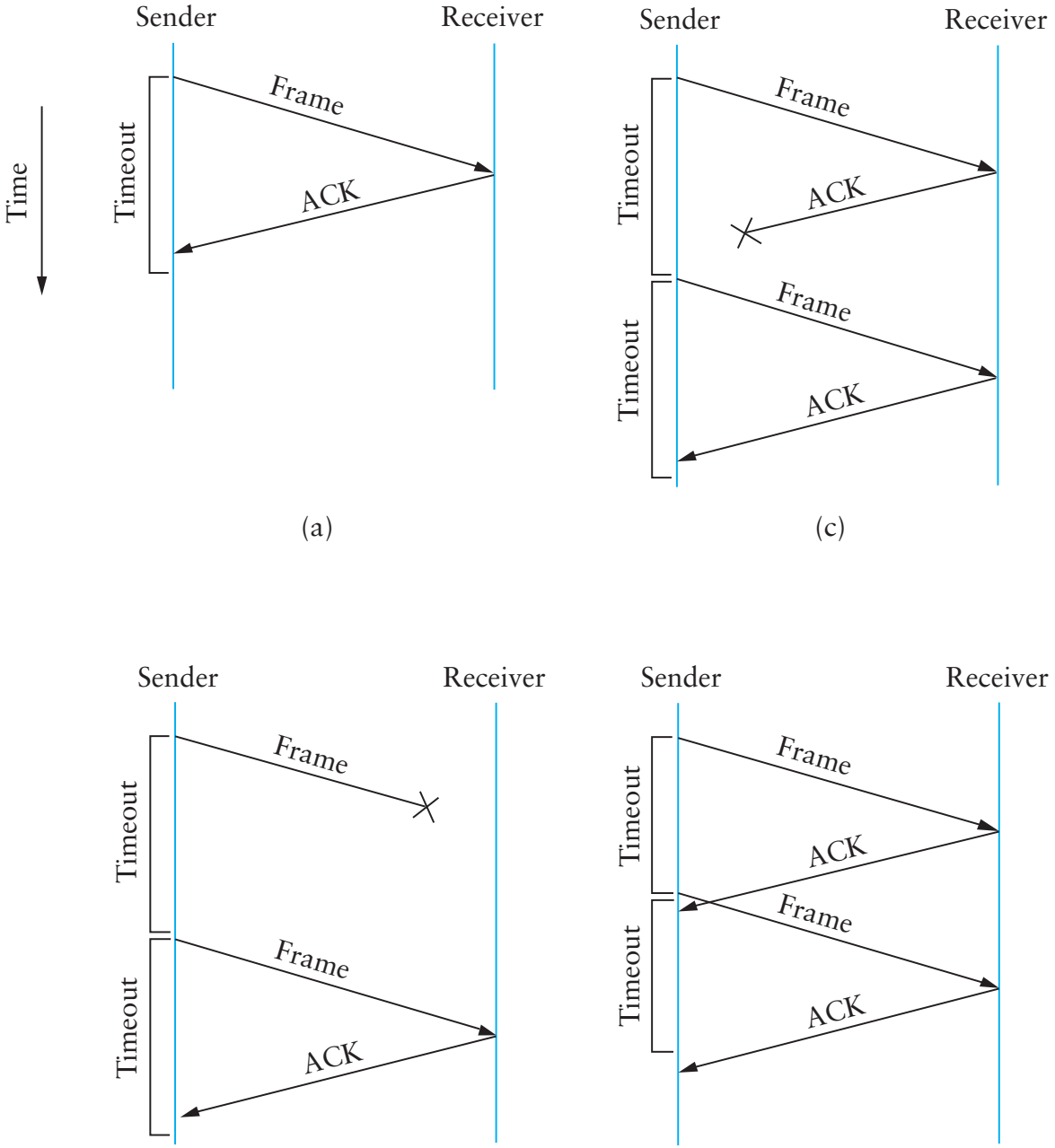
- CRC-8: $x^8 + x^2 + x^1 + 1$
- CRC-16: $x^{16} + x^{15} + x^2 + x^1$
- CRC-32: $x^{32} + x^{26} + x^{23} + x^{22} + x^{16} + x^{12} + x^{11} + x^{10} + x^8 + x^7 + x^5 + x^4 + x^2 + x^1 + 1$
- CRC easily computable in hardware

Reliable Delivery

- Error detection can discard bad packets
- Problem: if bad packets are lost, how can we ensure reliable delivery?
  - Exactly-once semantics = at least once + at most once

At Least Once Semantics

- How can the sender know packet arrived *at least once*?
  - Acknowledgments + Timeout
- Stop and Wait Protocol
  - S: Send packet, wait
  - R: Receive packet, send ACK
  - S: Receive ACK, send next packet
  - S: No ACK, timeout and retransmit

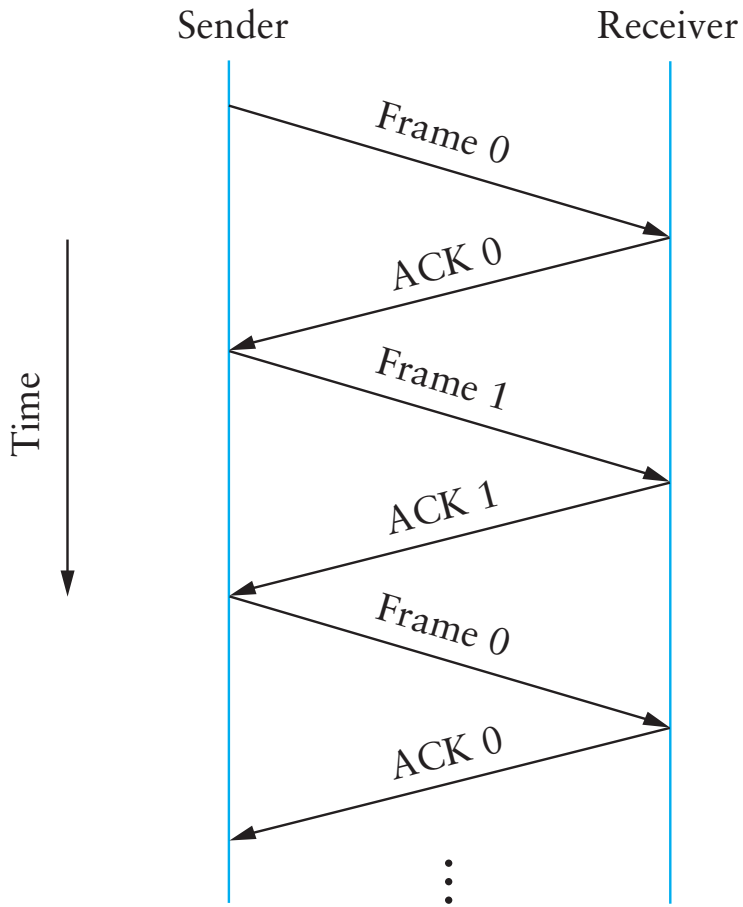

Sender
Receiver
Time
Timeout
Frame
ACK
(a)
Sender
Receiver
Timeout
Frame
ACK
Timeout
Frame
ACK
(c)
Sender
Receiver
Timeout
Frame
Timeout
Frame
ACK
(b)
Sender
Receiver
Timeout
Frame
ACK
Timeout
Frame
ACK
(d)

Stop and Wait Problems

- Duplicate data
- Duplicate acks
- Can’t fill pipe
- Difficult to set the timeout value

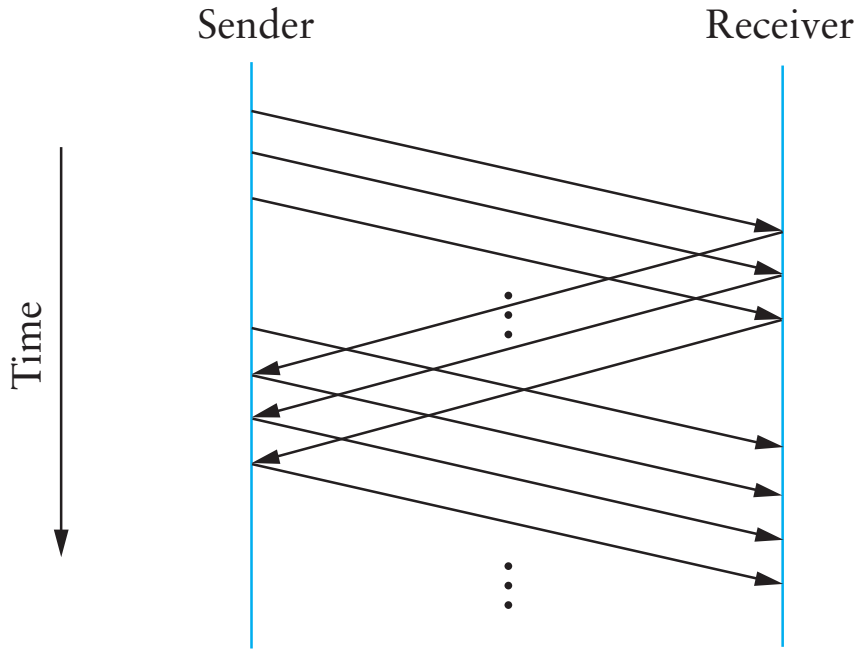
At Most Once Semantics

- How to avoid duplicates?
  - Uniquely identify each packet
  - Have receiver and sender remember
- Stop and Wait: add 1 bit to the header
  - Why is it enough?


Sender
Receiver
Frame 0
ACK 0
Frame 1
ACK 1
Frame 0
ACK 0
Time

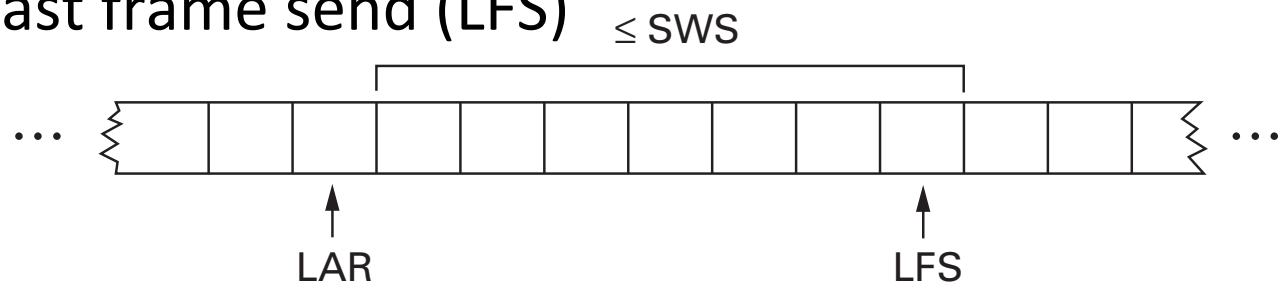
Sliding Window Protocol

- Still have the problem of keeping pipe full
  - Generalize approach with > 1-bit counter
  - Allow multiple outstanding (unACKed) frames
  - Upper bound on unACKed frames, called *window*



Sliding Window Sender

- Assign sequence number (SeqNum) to each frame
- Maintain three state variables
  - send window size (SWS)
  - last acknowledgment received (LAR)
  - last frame send (LFS)



- Maintain invariant: LFS – LAR ≤ SWS
- Advance LAR when ACK arrives
- Buffer up to SWS frames

Sliding Window Receiver

- Maintain three state variables:
  - receive window size (RWS)
  - largest acceptable frame (LAF)
  - last frame received (LFR)

- Maintain invariant: LAF – LFR ≤ RWS
- Frame SeqNum arrives:
  - if LFR < SeqNum ≤ LAF, accept
  - if SeqNum ≤ LFR or SeqNum > LAF, discard
- Send *cumulative* ACKs