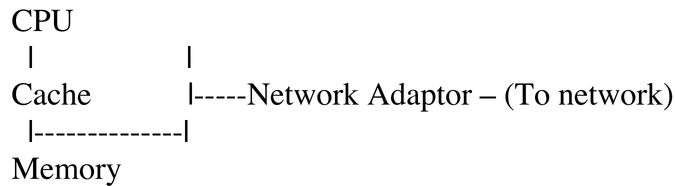
Csc72010

Parallel and Distributed Computation and Advanced Operating Systems

Lecture 2

Notes based on Peterson and Davies, Chapter 2

Nodes



CPU speeds increase faster than memory speeds: networking is memory-speed bound!

Links (physical layer)

Speed of light
Full duplex versus half duplex
Cat 5 most common right now – what others do you know?

- Fiber
- Wireless (Bluetooth, 802.11b/g at 2.4 GHz and 802.11a at ?? GHz)
- Twisted pair
- Cable

Encodings

NRZ: 1 is high signal, 0 is low signal
    Problems: baseline wander, recovering clock
NRZI: Transition for a 1, stay at current signal for a 0
Manchester: XOR NRZ-encoded data and the clock, where clock alternates between high and low, with one low-high pair being a cycle.

Framing

Data link layer: the data units are called frames (at network layer, called packets). The network adaptor must determine where frames start and end.

[Format for describing “data units”: A sequence of labeled fields, with a number above each indicating the length. Transmitted from left to right.]

Byte-oriented

Sentinel approach: start and end with special characters
Problem: special character may appear in data
Solution: escape

Start byte + count

Bit-oriented

Similar to byte-oriented: distinguished bit sequences at beginning & end – use "bit stuffing" to avoid them appearing in the middle of a frame, i.e., 01111110 is the distinguished bit sequence for HDLC. After 5 consecutive 1's, must insert a 0. On receiving side, if bit after 5th 1 is a 0, it was stuffed and is discarded; if it was a 1, then either this is the dbs or there's an error.

Clock-based framing

SONET: Frame is 810 bytes long; there is a special two-byte sequence at beginning. Receiver looks for the two-byte sequence at 810 byte intervals.

Error detection

Two-dimensional parity

One-dimensional: add 1 bit to a 7-bit code (ASCII) or 8-bit (EBCDIC)
Two-dimensional: do it in both directions
See Figure 2.16

Internet Checksum

Add up all transmitted words, and then transmit the sum.
Receiver performs the same computation and compares the result.

Current technique in the Internet: Consider data as a sequence of 16-bit integers. Add them together using 16-bit 1's complement. (-x has every bit of x inverted; when adding, a carryout from the most significant bit is added to the result. Example: -5 and –3:

5 0101 -5 1010
3 0011 -3 1100
0111 (one's complement of 8 = 1000)
Weak protection, easy to implement

Cyclic Redundancy Check

Think of a *message as being represented by a polynomial. Coefficient is value of bit. Sender and receiver agree on a divisor polynomial of some degree k. To send a (n+1)-bit message, send* also k additional bits

Multiply M(x) by x^k, that is, add k zeroes to the end of the message. This zero-extended message is T(x).
Divide T(x) by C(x) and find the remainder.
Subtract the remainder from T(x).
Send what is left – this is exactly divisible by C(x).

Facts (page 96):
CRC can find all single-bit errors, as long as the x^k and x^0 terms of C(x) are non-zero.
CRC can find all double-bit errors, as long as C(x) has a factor with at least three terms.
CRC can find any odd number of errors, as long as C(x) contains the factor (x+1).
CRC can find any "burst" error for which the length of the burst is less than k.

Reliable Transmission

When frames arrive that can't be corrected –
Acknowledgment: a control frame that says a data frame has been received correctly
Timeout: Retransmit if there's no acknowledgment before timer goes off
Generically, Automatic Repeat Request, or ARQ strategy.

Stop-and-wait:

Send and wait for ACK. If no ACK by timeout, resend. *Standard method for reliable send: require an ACK (return receipt) for each message, and resend if no ACK received.* See cases in Figures 2.19.

There's a duplicate problem if the ACK is lost or delayed, because the receiver acknowledged it but the sender re-sends it. Use 1-bit sequence number alternating between 0 and 1. See Figure 2.20

Only one frame on a link at a time – way below link capacity.
Run through computation in book.

Remember bandwidth * delay product – we would like to send 8 1-KB frames if the pipe can hold 8 KB.

Concurrent logical channels in ARPANET

Use stop-and-wait on each of multiple logical channels – so if you can fill a 1-KB pipe with one logical channel using Stop-And-Wait, use 8 logical channels on an 8-KB pipe.

Sliding Window – Data Link Layer Version

Suppose we are permitted to send 8 frames before an ACK, then we wait for the ACK – i.e., we can keep 8 frames "ahead" of the ACK.

We call the distance we can get ahead of the ACK's the *window size*. To keep track of where we are, we give the frames sequence numbers. So if the window size is 8 frames,

then we can send frames 1-8 before we receive the ack for 1; after we get the ack for 1, we can send frame 9.

Both the sending and receiving sides have a "window" – usually same size – consisting of some number of frames – for concreteness, we'll say 8 frames

The receiver keeps track of its window size (RWS), the last frame received (LFR), and the last acceptable frame (LAF). What's the last frame the receiver can accept?

Answer: Every time the receiver receives a frame, it checks whether it can accept it (is it inside the window, ie, its seqno is less than LAF?).

If it can accept it, it puts it in the correct position in the buffer (size = window size*frame size) and sends an ack for the latest *consecutive* frame received (*SeqNoToAck*). This is a cumulative acknowledgment, i.e., when the receiver acknowledges frame 15, it is saying it has received all frames up to and including 15.

Noice that LAF – LFR <= RWS

Example with out-of-order frames.

Example A. Sender sends 1,2,3,5,6,4,7,8
Acks are 1 (after 1) 2 (after 2) 3 (after 3) 6 (after 4) 7 (after 7) and 8 (after 8)

Example B. Sender sends 1,3,5,7,8,6,2,4.
Acks are 1 (after 1) 3 (after receiving 2) and for 8 (after 4).

Now, let's consider what's going on at the sender:

The sender keeps frames in its buffer (window). It also keeps track of the window size (SWS), the last frame sent (LFS), and the last frame for which it received an ack (LAR). The sender hangs on to each frame until it has been acknowledged. The sender checks to see if it can send a frame (if its buffer window is full of unacknowledged frames, it can't send any more). When it sends a frame, it sets a timer & if the timer expires before it sees an ack, it re-sends the frame. When it receives an ack, it "slides" the window so there's room for another frame.

Example: Suppose the sender has a window size of 3.

Then in example A,
Let's say this is the order in which things happen:
Send 1, Send 2, Send 3, Ack 1, Send 4, Ack 2, Send 5, Ack 3, Send 6, Ack 6, Send 7, Send 8, Ack 7, Ack 8

In example B: suppose the window size is 8
Send 1, Send 2, Send 3, Ack 1, Send 4, Send 5, Send 6, Send 7, Send 8, Ack 3, Ack 8

Notice that LFS – LAR <= SWS

Example with acks not arriving
Window size 3: Send 1, Send 2, Send 3, Ack 1 (lost), Ack 2, Send 4, …

Sequence number wrapping:
Suppose we have k bits for sequence numbers: 2^k possible sequence numbers. Must wrap.

This causes problems if the # messages outstanding > (2^k+1)/2 = 2^(k-1) + 1/2

What can happen?
Example: Suppose we have 3 bits, so there are 8 sequence numbers, and we have window sizes of 7 on both sides.
Worst case behavior: the sender could send frames 0...6 (that's 7 frames), and the receiver acks them all, but all acks are lost. Then the receiver is expecting frame 7, followed by new frames 0,1,… but the sender re-sends old frames 0, …, 6. It looks to the receiver like frame 7 was lost or delayed.

Book: If SWS=RWS, then they must be < (MaxSeqNum+1)/2 – otherwise, all the acks for outstanding frames could be lost

Must be big enough to avoid confusion.

Note the functions that sliding window performs:
1) Guarantees reliable transmission (if window sizes are not too large for sequence numbers)
2) Keeps frames in order

Under some circumstances, it can be used for flow control: throttles the sender because it must wait for ACKs (but not the above algorithm)

Switches and Bridges

Interfaces

Forwarding

My intent with these algorithms was to illustrate simple Ethernet LAN algorithms that bridges and switches use. Links in these networks are bidirectional. When I discussed these algorithms today, I was thinking bidirectional edges, as I think most of you were (otherwise, there should have been a few more questions).

There are networks that have unidirectional links (SONET, wireless), but the properties of these algorithms are much more complicated for unidirectional links, and they're unlikely to be used in that environment. So, using in-nbrs and out-nbrs is misleading, and

I have changed the model to collapse in-nbrs and out-nbrs into nbrs. I also assume that the network graph is an undirected graph for this particular purpose. I encourage you to think about what happens in a directed network graph, but make sure you understand the bidirectional case first.

Forwarding algorithm idea

Each message received on an interface of a bridge is sent out every other interface.

Forwarding algorithm messages

I distinguish between the initial end-to-end messages sent by hosts and the collections of end-to-end messages sent by switches.

Let M be the set of all possible end-to-end messages. Members of M are uninterpreted symbols, and a switch can't do anything with them but copy them and collect them in vectors.

We want to consider both hosts and switches as nodes of the network graph, without differentiating between them. To make this work, the type of message sent on a host-to-switch link is the same as the type sent on a switch-to-host link. This means a host must encapsulate a message d in a vector <d> when sending it. Also, the switch sends vectors of messages to each connected host. We assume that the host can figure out which messages are addressed to it.

Forwarding Algorithm

Idea: A switch sends each incoming message out all other ports.

Correctness

Correctness conditions that we would like to have are:

Each message eventually arrives at its destination.

No message continues being forwarded forever.

The second correctness condition will hold only for graphs without cycles, that is, trees. A message would that is forwarded forever would have to be sent over the same link twice, so in an acyclic graph this depends on knowing that the message never goes backwards. This is true since it never gets sent back out the port it came in

The first correctness condition will hold for this algorithm if the network graph is connected (i.e., there is an undirected path from each node to every other node) and there are no changes to the graph during the execution. The intuition is that each message eventually goes from each bridge to all of its neighbors. Then looking at the graph, we see that it must go to every location in the graph. Therefore it has to reach its destination.

Intermediate statements that you can use to prove this is:

A message in position i of an input buffer moves up to position i-1 of the buffer after at most k1 transitions of the automaton.
A message at the head of an input buffer is moved to every output buffer within k2 transitions.
A message in position i of an output buffer moves up to position i-1 of the buffer within k3 transitions
A message at the head of an output buffer is sent within k4 transitions.

These conditions all depend on "liveness," i.e., each task gets enough chances to execute.

Complexity:
Communication: In a connected graph, each message traverses every edge of the network graph at least once. If the graph is a tree, the message traverses each edge exactly once. So communication complexity is |E| in a tree and infinite if there are cycles in the graph.

Time: The time complexity for a message to go from src to dst is distance(src, dst) rounds from the time the originating host sends the message until it arrives at the destination host. Worst case time complexity for a message to go from src to dst is diam(G).

Learning bridges

Messages have destination address, source address, and data.

Learning bridge idea

The bridge associates the source address on an incoming message with the interface; sends messages destined for that source out that interface only. Note that this is not much use in an arbitrary directed graph. It requires bidirectional links.

Learning bridge algorithm

transitions
output send(m:Message, p:Nat)
input receive(m:Message, p:Nat)
internal copy(p:Nat, q:Nat)

states
inbuf: Array[Nat,Seq[M]]
outbuf: Array[Nat,Seq[M]]
mac-address-table: Array[Nat, Nat]\

transitions
output send(p)
take a message out of the outbuf for the port p

input receive(p)
add a message to the inbuf for the port p
also, remember the address of the src in the mac-addr-table for looking up the port

internal copy(p,q)
move a message from the head of p's inbuf to:
all other outbufs if the dest addr is not in the mac-addr-table
the designated outbuf if the dest addr is in the mac-addr-tablew

Homework

For Thursday, February 22

1. Write a tioa that implements the sliding window algorithm.

Show that the algorithm has the following desirable properties:

a. Messages are delivered to the receiving application in the order that they were sent.
b. Each message is delivered to the receiving application at most once.
c. If the channel drops only a finite number of messages, then all messages are delivered to the receiving application.

Make the proof as formal as you can, but I will require only that you make a reasonable argument that presents the key insights. We will discuss how to formalize the proof in subsequent classes.

2. Fill in the details in the learning bridge algorithm.
   a. Discuss what happens if switches forward messages out all ports, instead of all ports except the incoming port.
   b. Discuss what happens if switches are connected to each other in a cycle.

Both tioa should be syntactically correct (they pass the tioa checker), but you do not need to simulate them.