Point-to-Point Links

Hongwei Zhang

http://www.cs.wayne.edu/~hzhang

It is a mistake to look too far ahead. Only one link in the chain of destiny can be handled at a time.

--- Winston Churchill

Acknowledgement: this lecture is partially based on the slides of Dr. Larry Peterson
Outline

- Encoding
- Framing
- Error detection
- Reliable transmission
Outline

- Encoding
- Framing
- Error detection
- Reliable transmission
Outline

- Encoding
- Framing
- Error detection
- Reliable transmission
Outline

- Encoding
- Framing
- Error detection
- Reliable transmission
Cyclic Redundancy Check?

- Add k bits of redundant data to an n-bit message
  - want \( k << n \)
  - e.g., \( k = 32 \) and \( n = 12,000 \) (1500 bytes) --- CRC-32

- Represent n-bit message as \( n-1 \) degree polynomial
  - e.g., MSG=10011010 as \( M(x) = x^7 + x^4 + x^3 + x^1 \)

- Let k be the degree of some \textit{divisor polynomial}
  - e.g., \( C(x) = x^3 + x^2 + 1 \)
CRC (cont)

- Transmit polynomial $P(x)$ that is evenly divisible by $C(x)$
  - shift left $k$ bits, i.e., $M(x)x^k$
  - subtract remainder of $M(x)x^k / C(x)$ from $M(x)x^k$
    - Modulo 2 arithmetic: subtract $\equiv$ XOR
    - Let’s do an exercise: $MSG = 10011010$, $C(x) = x^3 + x^2 + 1$ (p.98)

- Receiver polynomial $P(x) + E(x)$
  - $E(x) = 0$ implies no errors

- Divide $(P(x) + E(x))$ by $C(x)$; remainder zero if:
  - $E(x)$ was zero (no error), or
  - $E(x)$ is exactly divisible by $C(x)$ (error goes undetected in this case)
Selecting $C(x)$

- Can detect
  - All single-bit errors, as long as the $x^k$ and $x^0$ terms have non-zero coefficients.
  - All double-bit errors, as long as $C(x)$ contains a factor with at least three terms
  - Any odd number of errors, as long as $C(x)$ contains the factor $(x + 1)$
  - Any ‘burst’ error (i.e., sequence of consecutive error bits) for which the length of the burst is less than $k$ bits.
    - Most burst errors of larger than $k$ bits can also be detected

- See Table 2.5 on page 101 for common $C(x)$
Internet Checksum Algorithm

- View message as a sequence of 16-bit integers; sum using 16-bit ones-complement arithmetic; take ones-complement of the result.
  
  ```c
  u_short cksum(u_short *buf, int count) {
    register u_long sum = 0;
    while (count--)
      { sum += *buf++;
        if (sum & 0xFFFF0000) {
          /* carry occurred, so wrap around */
          sum &= 0xFFFF;
          sum++;
        }
      }
    return ~(sum & 0xFFFF);
  }
  ```

In ones complement arithmetic,
- a negative integer -x is represented as the complement of x
- a carryout from the most significant bit needs to be added to the results
Outline

- Encoding
- Framing
- Error detection
- Reliable transmission
Reliable transmission in spite of bit and burst errors?

- Forward error correction does not guarantee reliable transmission or is not cost effective

- “Acknowledgment (from receiver)” and “timeout (at sender)” as the basic mechanism for reliable transmission
  - Stop-and-wait
  - Sliding window
  - Concurrent logical channels
Stop-and-Wait

4 different scenarios
Stop-and-wait (contd.)

-Problem: how to keep the pipe full

-Example
  -1.5Mbps link × 45ms RTT = 67.5Kb (8KB)
  -1KB frames implies 1/8th link utilization
Sliding Window

- Allow multiple outstanding (un-ACKed) frames
- Upper bound on un-ACKed frames, called *window*
SW: Sender

- Assign sequence number to each frame ($SeqNum$)

- Maintain three state variables:
  - send window size ($SWS$): depends on channel capacity
  - last acknowledgment received ($LAR$)
  - last frame sent ($LFS$)

- Maintain invariant: $LFS - LAR \leq SWS$

- Advance $LAR$ when ACK arrives

- Buffer up to $SWS$ frames
SW: Receiver

- Maintain three state variables
  - receive window size (\textit{RWS}): depends on local memory size, and \textless\textless SWS
  - largest acceptable frame (\textit{LAF})
  - last frame acknowledged (\textit{LFA})

- Maintain invariant: \textit{LAF} - \textit{LFA} \textless\textless \textit{RWS}

- Frame \texttt{seqNum} arrives:
  - if \texttt{LFA} < \texttt{seqNum} \textless\textless \texttt{LAF} \rightarrow accept
  - if \texttt{seqNum} \textless\textless \texttt{LFA} or \texttt{seqNum} > \texttt{LAF} \rightarrow discarded

- Send cumulative ACKs
A diversion: 
Formal protocol specification


- Our case study here: cumulative ACK
Scenario

P  q
  data
   --->
  data
   --->
  data
   --->

q must deliver the data to its application without loss, reorder, or corruption.
Notation & variables

data(i), where i is an integer (i.e. unbounded) sequence number.

ack(i) acknowledges from data(0) up to data(i-1). Note that it does not acknowledge data(i).

There are two important variables in process p:

1. ns - which is the next sequence number to send
2. na - which is the next sequence number to be acknowledged.

sequence numbers in 0 .. ns-1 have been sent by p, p has received an ack for data messages 0 .. na-1.
Process q has a single important variable:

nr - next sequence number to be received by q
data(0) up to data(nr-1) have been received by q.

(note: data(nr) has not been received yet, since otherwise we would have incremented nr)
Intuition diagram
Abstract Protocol (AP) representation:
sender

process p
const w
var
    na, ns, i : integer
begin
    ns < na + w \rightarrow \text{send data}(ns) \text{ to } q;
    ns := ns + 1

    \[]
    \text{rcv ack}(i) \text{ from } q \rightarrow \text{na} := \text{max}(na, i)

    \[]
    na < ns \rightarrow \text{send data}(na) \text{ to } q
end
AP representation: receiver

```plaintext
process q
const w
inp
  wr : integer  {1 ≤ wr ≤ w}
var
  nr, j : integer
  rcvd : array [0 .. wr-1] of boolean
  ack : boolean
begin
  rcv data(j) from p →
    if j < nr →  ack := true  {old message}
    [] j ≥ nr+wr →  skip  {no buffer space}
    [] nr ≤ j < nr+wr →
      rcvd[j mod wr] := true;
      do  rcvd[nr mod wr] →  
          {deliver data(nr) }
          rcvd[nr mod wr], nr, ack := false, nr+1, true
      od
  fi
  []
  ack → send ack(nr) to p; ack := false
end
```
Potential drawback of the protocol?

- Sender process $p$?
- Fix: retransmit only after timeout
Sequence Number Space

- **SeqNum** field is finite; sequence numbers wrap around
- Sequence number space must be larger than number of outstanding frames
- Is $SWS \leq |SeqNumSpace| - 1$ sufficient? Not always
  - suppose 3-bit **SeqNum** field (0..7)
  - $SWS=RWS=7$
  - sender transmit frames 0..6
  - arrive successfully, but ACKs lost
  - sender retransmits 0..6
  - receiver expecting 7, 0..5, but receives second incarnation of 0..5
- when $SWS == RWS$,
  - $SWS \leq |SeqNumSpace| / 2$ is the correct rule
  - Intuitively, **SeqNum** “slides” between two halves of sequence number space
Q: General rule on SWS, RWS, seq. number space?

- Expected frames should not overlap (in a wrap-around manner) with retransmitted frames

- Worst case: all ACKs are lost

- General rule: \( SWS + RWS \leq |\text{SeqNumSpace}| = MaxSeqNum + 1 \) (i.e., size of seq. # space)
  - Note: the book uses \( MaxSeqNum \) to denote \( |\text{SeqNumSpace}| \)
Other roles that sliding window protocol can play

- In-order frame delivery
- Flow control to avoid overwhelming receiver
  - Subject to constraint of the channel capacity and receiver memory size
- More important at higher layers such as TCP and application
Concurrent Logical Channels (ARPANET)

- Multiplex 8 logical channels over a single link
- Run stop-and-wait on each logical channel
- Maintain 3 state bits per channel
  - channel busy
  - current sequence number out
  - next sequence number in
- Header: 3-bit channel num, 1-bit sequence num
  - 4-bits total
  - same as sliding window protocol
- Separates *reliability* from *order*
Error control in sensor networks

RETTRANSMITION does not help much, and may even decrease reliability and throughput.

Similar observations when adjusting contention window of B-MAC and using S-MAC.

<table>
<thead>
<tr>
<th>Metrics</th>
<th>RT= 0</th>
<th>RT= 1</th>
<th>RT= 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reliability (%)</td>
<td>51.05</td>
<td>54.74</td>
<td><strong>54.63</strong></td>
</tr>
<tr>
<td>Latency (sec)</td>
<td>0.21</td>
<td>0.25</td>
<td>0.26</td>
</tr>
<tr>
<td>throughput (pkt/sec)</td>
<td>4.01</td>
<td>4.05</td>
<td>3.63</td>
</tr>
</tbody>
</table>
Problem statement

- How to achieve
  - close to 100% reliability?
  - close to optimal event throughput (implies real-time packet delivery)?

- Elements of the solution RBC
  - Window-less block acknowledgment
  - Differentiated contention control
  - Fine-grain timer management

Window-less block ACK

Non-blocking window-less queue management

- Unlike sliding-window based block ACK, in order packet delivery is not considered
  - Packets have been timestamped

- For block ACK, sender and receiver maintain the “order” in which packets have been transmitted
  - “order” is identified without using sliding-window, thus there is no upper bound on the number of un-ACKed packet transmissions
Sender: queue management

static physical queue

VQ0

VQ1

VQ_M

VQ_M+1

ID of buffer/packet

M: max. # of retransmissions

ranked virtual queues (VQ)

occupied

empty

high

low

Sender: queue management
Sender: gets a packet from an upper layer

empty queue buffer?
Sender: transmits a packet

- **fresher**
  - $VQ_0$: 1 → 2 → ...
  - $VQ_1$: 3 → 4 → 5 → ...
  - $VQ_M$: ...
  - $VQ_{M+1}$: 6 → ...

- **earlier**
- **order of transmission**
- **later**

- $\langle 1, 2 \rangle$
**Receiver: loss detection**

- If no packet loss, expecting packet \( j \) at \( i' \).

- Check if \( i' \neq j \):
  - If equal, no loss.
  - If not equal, some loss.
Receiver: block ACK

\[ \langle i, i \rangle \ \langle i, j \rangle \ \langle i, k \rangle \ \cdots \ \langle i, k' \rangle \]

ACK replication!
Sender: processes a block ACK
Differentiated contention control

- Schedule channel access across nodes

- Higher priority in channel access is given to
  - nodes having fresher packets
  - nodes having more queued packets
Implementation of contention control

- The rank of a node $j = \langle M - k, |VQ_k|, ID(j) \rangle$, where
  - $M$: maximum number retransmissions per-hop
  - $VQ_k$: the highest-ranked non-empty virtual queue at $j$
  - $ID(j)$: the ID of node $j$

- A node with a larger rank value has higher priority

- Neighboring nodes exchange their ranks
  - Lower ranked nodes leave the floor to higher ranked ones
Fine tuning retransmission timer

- **Timeout value**: tradeoff between
  - delay in necessary retransmissions
  - probability of unnecessary retransmissions

- **In RBC**
  - Dynamically estimate ACK delay
  - Conservatively choose timeout value; also
    reset timers upon packet and ACK loss