1st Cut Reliable UDP Protocol

TCP/IP class

Jim Binkley

a mystery lecture

Jim Binkley

the problem

- network/link layer problems include
  - **loss of packets** due to congestion, collisions, noise (ethernet detects bad crc, shoots packet), or no space at sender (buffer overrun)
  - **data corruption** due to not enough CRC, no CRC, or bad memory
- that’s fine bucky, what do we do?

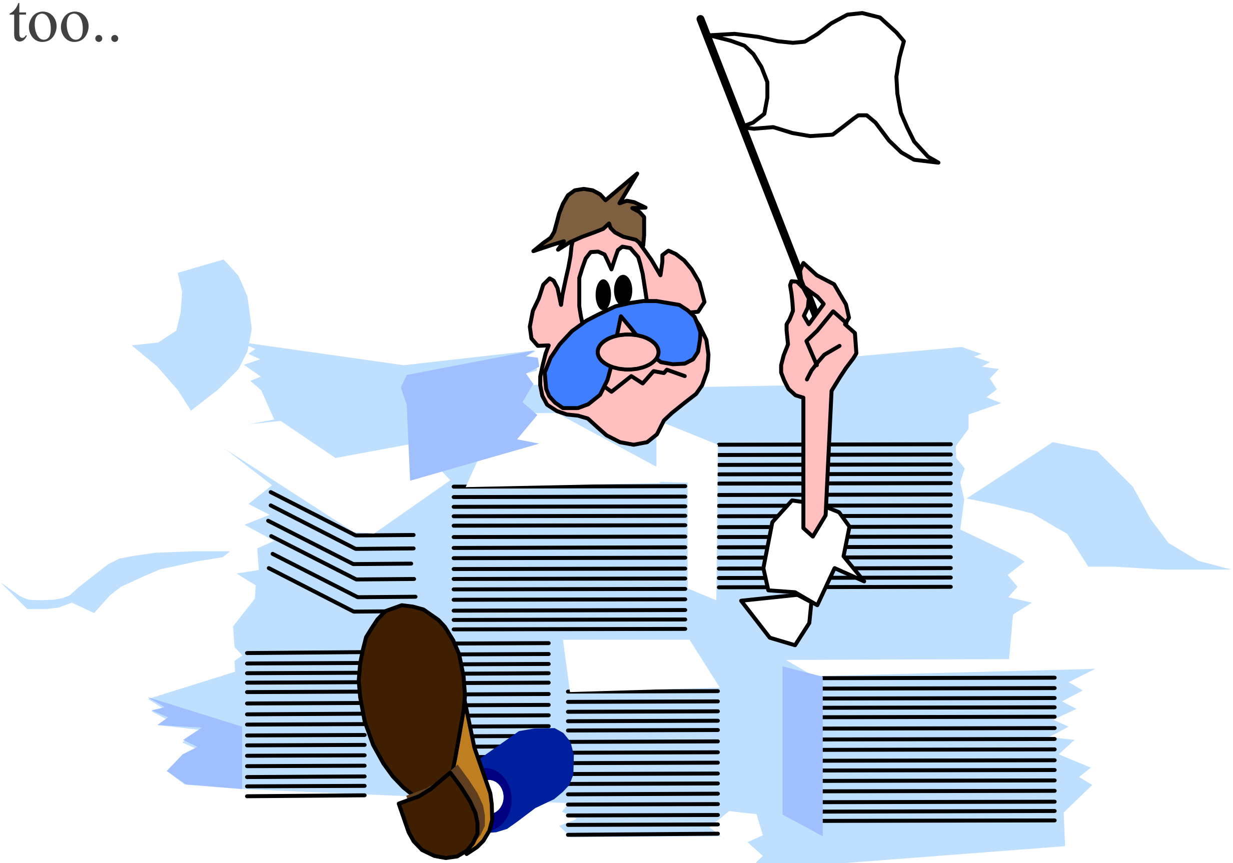
  design an end-end reliable protocol

qualities of said protocol

- keep in mind that TCP is a possible model. It has many complex features and may be too complex
- UDP doesn’t qualify (“fire and forget”)
- design criteria might include:
  - reliable!
  - efficient
  - no deadlock and both sides can start asynch
  - point to point connected (TCP) OR
    N to 1 or 1 to N datagram-style (UDP)?

286 after download from Cray

flow control too..



Jim Binkley

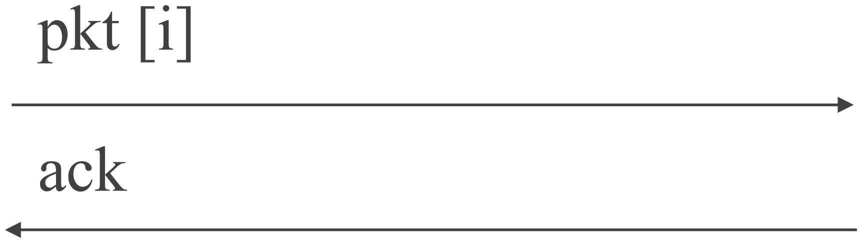
first assumptions

- assume a 1-way channel, writer to reader

- we will use **positive acknowledge with retransmission** (as opposed to NAKs)

pos. ack with retransmission

- send the packet, get back an explicit acknowledgement.



pkt [i]

ack

- we need to time the ACK and resend if it doesn’t come back

ACKs mean new problems

- assume a fixed TIMEOUT N, followed by a resend? What problems does this introduce?
  - 1. we may have **duplicate packets** on the net
  - 2. packets may get **out of order** due to more than one path through routers (with different link delays)
- we need a packet header with a sequence number

packet header is at least

- rup hdr {

  ```
        unsigned int seqno;
        unsigned long csum;
  }
  ```

- a checksum too so we can deal with the problem of damaged data. We can just use the IP checksum algorithm (which is weaker than a link CRC, but will do)

sequence number notions

- worry about what it does when it “wraps”

```
recv:
        if recv_seqno > current_seqno
              then OK!
```

- what if the recv_seqno is MAXINT?
- for a simple protocol the seqno range can be [0..1]
  - if 0, then 1, if 1, then 0

ACKS need sequence numbers

- assume you just send back an ACK

send	recv
send pkt[i] and wait for ACK	
pkt[i] goes slow route and timeout occurs	
retransmit, now takes fast route	
	got #2, send ACK
send pkt[i+1], wait for ACK	
pkt[i+1] is lost	
	slow pkt[i] arrives
	send ACK
get wrong ACK for pkt[i+1]	oops ... sorry ...
but assume it got there	

more problems

- fixed timer may be permanently too slow for a given end to end path
  - make it too long, and you are inefficient if a packet is lost
  - too short and it will never work
- what if two processes both write a data packet and then read for an ACK?
- how does the server handle > 1 client

UDP versus TCP?

- grasshopper: “Master, isn’t UDP more efficient than TCP?”
- master: “Sure, if monologues are better than dialogs!”
- IMHO - TCP vs UDP is a big case of “it depends” both have pros/cons
- interesting problem: only have 1-way channel, how do you make it reliable?

some study questions

- efficiency? what does tcp do here?
- what more does TCP do that we haven't touched on?
- is pos. ack. with retransmission good for a reliable multicast protocol (1 to N)?
- how do you detect that your end point is down?
- why does TCP use a 3-way handshake to initialize the connection?
- why can't you use a 3-way handshake at the end of a connection?