CS 162 Operating Systems and Systems Programming

Professor: Anthony D. Joseph

Spring 2002

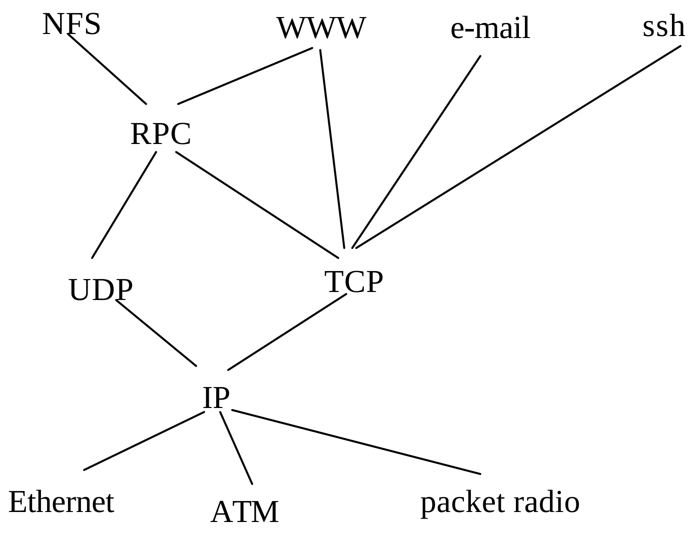
Lecture 21: Network Protocols (and 2 Phase Commit)

21.0 Main Point

Protocol: agreement between two parties as to how information is to be transmitted.

Example: system calls are the protocol between the operating system and applications

Another example: Alphabet soup. Will explain all these acronyms as we go along.



Physical Reality: packets	**Abstraction: messages**
Limited size	Arbitrary size
Unordered (sometimes)	Ordered
Unreliable	Reliable
Machine-to-machine	Process-to-process
Only on local area net	Routed anywhere
Asynchronous	Synchronous
Insecure	Secure

Illustrates **layering** – build services on simpler services

21.1 Arbitrary size messages

Arbitrary size messages on top of limited size ones

Send N little messages – split up message into fixed size packets

abcdefgh -> 1 of 3/abc   2 of 3/def   3 of 3/gh

Checksum can be computed on each fragment, or on whole message

21.2 IP – Internet Protocol

Deliver messages unreliably from one machine in internet to another.

a. Routes packets from one machine through internet to another

b. Some intermediate links may have limited size.
   Fragments on demand, re-assembles at destination

c. Unreliable, unordered, machine->machine

21.3 Process-process communication

User process communication on top of machine to machine communication

Mailbox (or “port”) address – include in each message, the destination mailbox. Allows you to direct each message to correct process

21.4 UDP – Unreliable Data Protocol

Unreliable, unordered, user-to-user communication

Built on top of IP

21.5 Ordered messages

Ordered messages on top of unordered ones

IP can re-order packets – send A, B arrives: B, A

How do we fix this? Assign sequence numbers to successive packets –

0, 1, 2, 3, ...

If arrive out of order, don’t deliver #3 to user application until get #2.

Sequence numbers specific to a connection – for example, the machine-machine (or mailbox-mailbox) pair. This means put “source” as well as “destination” in each header.

21.6 Performance considerations

Overhead – CPU time to put packet on wire
Latency – how long to send one byte packet
Throughput – maximum bytes per second

21.6.1 Example

How long to send 4KB packet over various networks? Typical overhead to send a packet: 1 ms.

Ethernet (**10 –** 1000Mb/s) within Soda:

Latency: speed of light = 1 ns / foot, implies < 1 microsecond.

Throughput delay: packet doesn't arrive until all its bits get there! So 4KB/10 Mb/s = 3 milliseconds (roughly as long as a disk!)

ATM (155 Mb/s) within Soda:

Latency same.

Throughput delay: 4KB/ 155 Mb/s = 200 microseconds.

ATM cross-country?

Latency: 3000 miles * 5000 ft/mile => 15 milliseconds.

Throughput delay: same as above.

How many bits are in transit at the same time?

15 ms * 155 Mb/s => 280 KB

Key to good performance: in local area, minimize overhead, improve bandwidth. In wide area, keep pipeline full.

21.7 Reliable message delivery

Reliable message delivery on top of unreliable delivery

All of these networks can garble, drop messages.

1. Physical media – if transmit close to maximum rate, get more throughput, even if some messages get lost
2. Congestion – what if no place to put incoming message (no buffer space)?
    - In point-to-point network, at each switch
    - What if two hosts try to use same link?
    - In any network, at destination
    - What if sender sends faster than receiver can process?

So what can we do?

See lecture from the beginning of the term on how to implement streams between cooperating processes.

1. Detect garbling at receiver via checksum, discard if incorrect
2. Receiver ack's if received properly
3. Timeout at sender. If no ack, retransmit

Some questions:

If the sender doesn't get an ack, does that mean the receiver didn't get the original message?

No.

What if ack gets dropped? Or if message gets delayed.

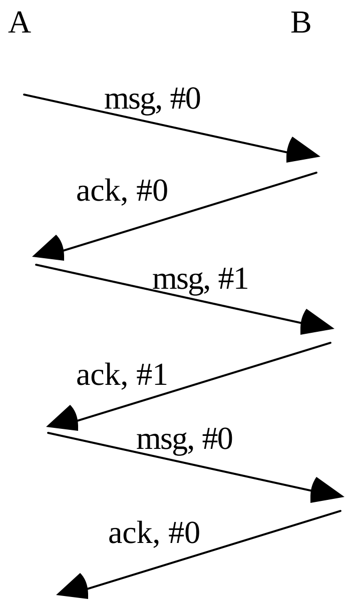
Sender doesn't get ack; retransmits. Receiver gets message twice, ack's each.

Solution: put sequence number in message to identify re-transmitted packets. Receiver checks for duplicate sequence #'s. If so, discards.

1. Sender must keep copy of every message that has not been ack'ed yet (easy)
2. Receiver must keep track of every message that could be a duplicate (hard! How does receiver know when it's ok to forget about received messages?)

Several approaches to maintaining state at sender/receiver:

a. Alternating bit protocol. One bit sequence number. Send one message at a time; don't send next message until ack received. Sender only keeps copy of last message; receiver keeps track of sequence # of last message received.



Pros & cons:

+ Simple

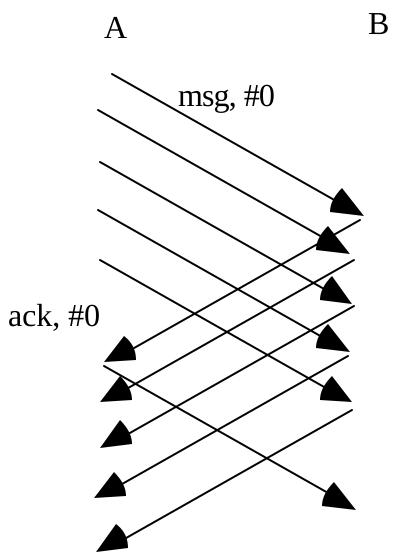
+ Small overhead

– Poor performance

b. Window-based protocol (TCP). Send up to N messages at a time, without waiting for ack.

"Window" reflects storage at receiver – sender shouldn't overrun receiver's buffer space.

Each message has sequence number. Receiver can say, "I've ack'ed up to message #X -> any message below X is a duplicate.



What if message gets garbled/dropped? Receiver will get messages out of order!

- Discard any messages that arrive out of order?
  - Simple, worse performance
- Keep copy until sender fills in the missing piece?
  - Reduces # of retransmits, more complex

What if ack gets dropped? Timeout and resend just the un-acknowledged message.

21.8 TCP: transmission control protocol

Reliable byte stream between two processes on different machines over Internet (read, write, flush).

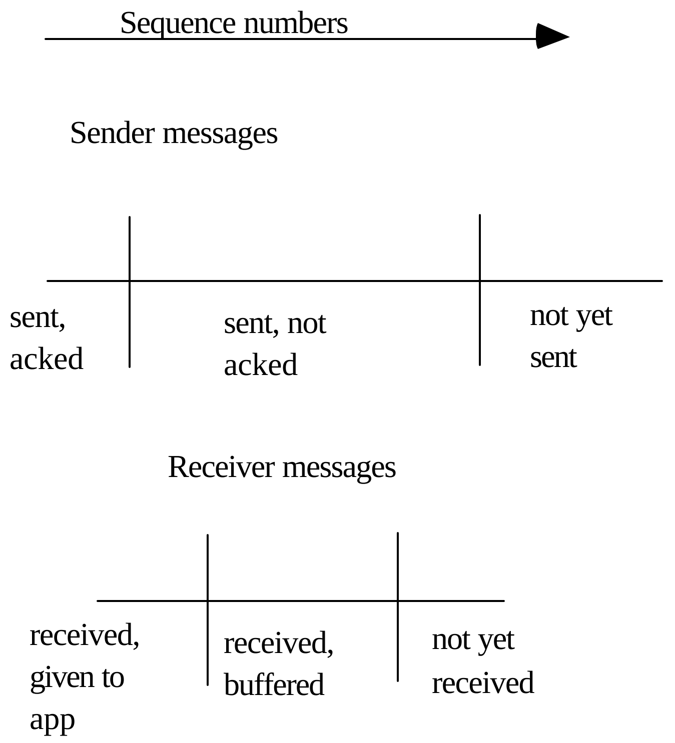
Fragments byte stream into packets, hands packets to IP.

Uses window-based protocol (to minimize state at sender and receiver): send up to N messages at a time, without waiting for ack.

“Window” reflects storage at receiver – sender shouldn’t overrun receiver’s buffer space.

Sender has three regions: sent and ack’ed, sent and not ack’ed, not yet sent

Receiver has three regions: received and ack'ed (given to application), received and buffered, not yet received (or received and discarded because out of order)



Each ack says: "got all messages up to #". What happens if ack is delayed, arrives out of order? OK in this scheme. Just discard.

21.9 Arbitrary Size Messages (revisited)

Face similar issues as in TCP when building big messages on small ones, when messages can get dropped.

1. Ack each fragment? Lots of acks
2. One ack for entire big message? Re-transmit all fragments, even if only one gets dropped
3. "Blast protocol" – send one ack, tells sender which pieces were missing. Selective retransmit.

21.10 Initialization

How do you know which sequence # to start with? When machine boots, ok to start with #0?

No. Could send two separate messages with the same serial #!

Two solutions:

1. Time to live: each TCP packet has a deadline. If not delivered in X seconds, then dropped. Thus, can re-use sequence numbers if wait for all packets in flight to be delivered or to expire.
2. Epoch # – uniquely identifies **which** set of sequence numbers are being used. Put in every message, epoch # incremented on crash and/or when run out of sequence #'s, and stored on disk.

21.11 Congestion

How long should timeout be for re-sending messages?

Too long? Wastes time if message is dropped.

Too short? Retransmit even though ack will arrive shortly.

Stability problem: more congestion -> ack is delayed -> unnecessary timeout -> more traffic -> more congestion

TCP solution: "slow start". Originally, window size = buffer space on remote end. Now, window size = control on how much to add to congestion.

Start sending slowly. If no timeout, slowly increase window size (throughput). If a timeout occurs, it means there's congestion, so cut window size (throughput) in half.

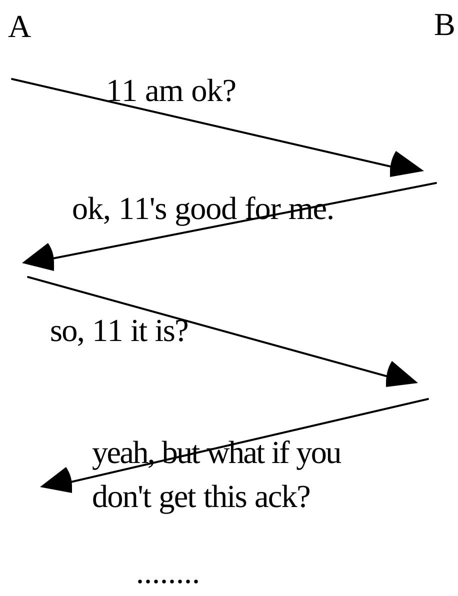
21.12 General's Paradox

Can I use messages and retries over an unreliable network to synchronize two machines so that they are guaranteed to do some operation at the same time?

Remarkably, no, even if all messages get through.

General's paradox: two generals, on separate mountains. Can only communicate via messengers; the messengers can be captured.

Need to coordinate the attack; if they attack at different times, then they all die. If they attack at the same time, they win.



Even if all messages are delivered, can't coordinate! Can't simultaneously get two generals or two machines to agree to do something at the same time.

No solution to this – one of the few things in CS that's just impossible.

21.13 Two phase commit

Since we can't solve the General's Paradox (i.e., simultaneous action), let's solve a related problem.
Abstraction: distributed transaction – two machines agree to do something, or not do it, atomically (but not necessarily at exactly the same time).

Two phase commit protocol does this. Use a persistent, stable log on each machine to keep track of whether commit has happened. If a machine crashes, when it wakes up it first checks its log to see what state the world was in at the time of the crash.

First phase, ask if each can commit – for instance, transfer of funds from one bank to another.

A writes, "Begin transaction" to log
A -> B: OK to transfer funds to me?

Not enough cash:

B-> A: transaction aborted
A writes "Abort" to log

Enough cash:

1. B: Write new X account balance to log
2. B->A: OK, I can commit

Second phase, A can decide for both, whether they will commit.

3. A: Write new Y account balance to log
4. Write commit to log
5. Send message to B that commit occurred
6. Write "Got commit" to log

What if:

- B crashes at 1? Wakes up, does nothing. A will timeout, abort transaction, retry.
- A crashes at 3? Wakes up, sees transaction in progress. What transaction, sends message to B, abort.
- B crashes at 3? B will come back up, look at log, so that when A sends it "Commit" message, it will say, oh, ok, commit.

One problem with 2PC is "Blocking": That is, a site gets stuck in a situation where it cannot continue until some other site (usually the coordinator) recovers. How could this happen?

- Participant site B writes a "prepared to commit" record to its log, sends a "yes" vote to the coordinator (site A) and crashes.
- Site A crashes
- Site B wakes up, checks its log and realizes that it had voted "yes" on the update. It sends a message to site A, asking what happened. At this point, B cannot change its mind and decide to abort, because the update may have

committed while it was crashed, thus it is *blocked*. (note, B may be able to learn the fate of the update by asking some of the other participants.)

Blocking is problematic because a blocked site must hold resources (for example, locks on updated items, pages pinned in memory, etc.) until it learns the fate of the update.

Question: Can blocking be avoided? Answer: yes! If you are willing to impose some constraints on the way that voting is done, an algorithm called "Three Phase Commit" can solve the problem. 3PC is not generally used in practice, however, due to performance reasons and the low instance of blocking in practice.