Lecture N4: 2-phase commit

****
Review -- 1 min
****
Motivation
Basic NW communication
3 problems
- performance
- reliability
- security

Case study: Distributed file systems

****
Outline - 1 min
****
- General's paradox
- 2-phase commit
- Reliable message queues

****
Preview - 1 min
****
If time permits: security

****
Lecture - 20 min
****

1. TBD Finish file systems

2. Reliability

Lamport: "A distributed system is a system where I can't get any work done if a machine I've never heard of crashes."

3. General's paradox

Want to be able to reliably coordinate activity on two different machines (e.g., both do the same thing at same time, exactly once semantics, atomically update state on two different machines, etc.)

e.g., atomically move directory from file server A to file server B
e.g., atomically move $100 from my account to Visa account

Challenge:

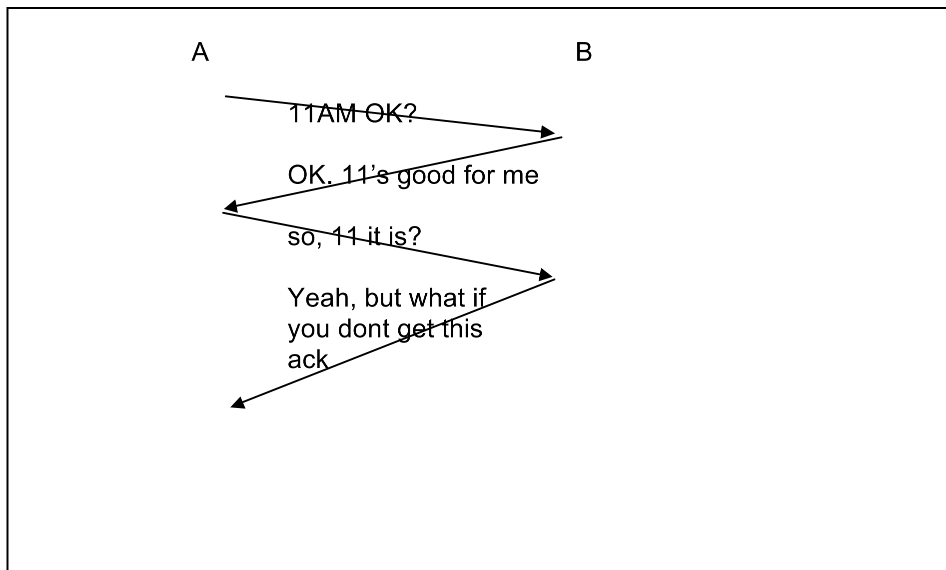
- messages can be lost
- machines can crash

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

Remarkably, no. Even if all messages end up getting through. Even if no machines crash.

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

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



Even if all messages are delivered, can't coordinate (B/c a chance that the last message doesn't get through). Can't simultaneously get two machines to agree to do something at same time

No solution to this – one of the few things in CS that is just impossible.
Proof: by induction

3.1 Network failures

Since I cannot solve General's Paradox, let me solve a related problem: at least once delivery

For now, assume no machine failures. Just network failures.
(1) communication interruption

- lost message
- lost reply
- cut wire
- …

Simple solution:
Request/acknowledge protocol
Common case:
1) Sender sends message (msg, msgId) and sets timer
2) Receiver receives message and sends (ack, msgId)
3) Sender receives (ack, msgId) and clears timer

If timer goes off, goto (1)

How does this work? What does it guarantee?
- What if msg 1 lost?
- What if ack lost?

Guarantees *at least once* semantics ***assuming no machines crash or otherwise discontinue protocol***
- Receiver guaranteed to recv message at least once

- Receiver may recv message multiple times. Receiver MAY use sequence number to filter repeated transmissions so that each is acted upon just once
-

3.1.1 At least once delivery

safety: If call at sender returns, message was processed by receiver at least once
liveness: if sender repeatedly sends until call returns and network eventually repaired and operates correctly long enough for a send/receive to occur, then eventually message is processed by receiver (at least once)

[[until call returns => no crash, no timeout/give up]]

Example: NFS "idempotent" requests

3.1.2 Exactly once delivery

Example: TCP/IP reliable stream

safety: If call at sender returns, message was processed by receiver exactly once

liveness: if sender repeatedly sends until call returns and network eventually repaired and operates correctly long enough for a send/receive to occur, then eventually message is processed by receiver (exactly once)

[[note: implementation typically requires sender and receiver to maintain state; cannot lose state in crash...]

3.1.3 Limitation: What if a machine crashes?

Do we still get "at least once" semantics if machines can crash?
NFS/RPC: no.
NFS Solution – blocking calls – don't return until remote operation completes.

Note: after a crash, operation may have happened zero, once, or ten times.

Do we still get exactly once semantics if machine can crash?
TCP: no
TCP solution:

(1) If sender or receiver crash or network partition causes either to give up, **no guarantee of "at least once"** – local send may complete before data received by remote machine
→ if send request not return, data may be received 0 or 1 time
→ if send request does return, data may be received 0 or 1 time

(2) *at most once semantic – if crash causes sender to reuse sequence numbers, no guarantee of at most once…s* hacks to make it very unlikely – pick sequence numbers unlikely to overlap with prev attempts; don't re-use port numbers until "pretty sure" both sides know connection is closed (two generals)
→ very unlikely that after receiver crashes, a resend will be accepted as a first send (~at most once semantics…)

Don't just worry about crashes. What about "giving up." Suppose I try to send for 10 seconds and get no reply – should I report "failure" to the user? 1 minute? 10 hours?

What are at least once/at most once semantics now?

Bottom line:
If machines can crash or give up (e.g., during a network partition), then messages can be received 0, 1, or N times
→ these things help
→ but still have corner cases to worry about

- These corner cases sometimes OK (e.g., TCP/IP – if one side gives up, eventually tear down the connection and hand an error up to higher level – let the higher level protocol recover (or exit)
- Sometimes they require recovery protocols (e.g., AFS callback recovery)

Can we provide a more powerful abstraction?

4. Machine failures

Several variations:

- ♦ user level bug causes address space to crash
- ♦ machine failure, kernel bug causes all AS on same machine to fail
- ♦ power outage causes all machines to fail

Before, whole system would crash. Now: one machine can crash, while others stay up.
Now, one machine can crash, while others stay up. If file server goes down, what do the other machines do?

Example: simple send/ack protocol above -- Difficult to deal with machine crashes

- If sender crashes (or if sender gives up because it has tried 100 times in a row) what is the post condition?
  - Receiver may or may not have received message
- If receiver crashes, filtering repeated messages to act on them exactly once is tricky → carefully design protocol to either (a) tolerate *at least once* semantics or (b) detect/avoid replication even across sender/receiver failures

Outline:
(1) 2-phase commit – distributed atomic update
(2) persistent message queues

5. 2-phase commit

Since I cannot solve General's Paradox, let me 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)

example: my account is at NationsBank, yours is at Wells Fargo. How to transfer $100 from you to me? (Need to guarantee that both banks agree on what happened).
Example: file system – move a file from directory A on server a to directory B on server b
Example: replication -- run k copies of file server so that if one fails, others can continue operation and files stay available -- need each replica to execute same series of requests

Two-phase commit protocol does this. Use log on each machine to keep track of whether commit happened

Ground rules/assumptions
-- all correct nodes must take same action (eventually)
-- reliable network -- if message sent to a node that is up, it is received in bounded time
-- correct nodes can crash and recover (losing memory but retaining disk storage) (and failing to send messages for some period of time)

Protocol
Phase 1: coordinator requests
1. coordinator logs REQUEST; sends REQUEST to all participants

   e.g. C→S1 "delete foo from /", C→S2 "add foo to /"

2. participants recv request, execute transaction locally, write REQUEST, RESULT, VOTE_COMMIT or VOTE_ABORT to local log,
   and send VOTE_COMMIT or VOTE_ABORT to coordinator

Failure case	*Success case*
S1 decides OK, writes "rm /foo; VOTE_COMMIT" to log, and sends VOTE_COMMIT *S2 decides no space on device and writes and sends VOTE_ABORT*	*S1 and S2 decide OK and write updates and VOTE_COMMIT to log, send VOTE_COMMIT*

Phase 2: coordinator decides
3. case 1: coordinator recv VOTE_ABORT or timeout

→ coordinator write GLOBAL_ABORT to log, and send GLOBAL_ABORT to participants

case 2: coordinator recvs VOTE_COMMIT from all participants
→ coordinator write GLOBAL_COMMIT to log, and send GLOBAL_COMMIT to participants

4. participant receives decision; write GLOBAL_COMMIT or GLOBAL_ABORT to log

What if

- Participant crashes at 2? Wakes up, does nothing. Coordinator will timeout, abort transaction, retry
- Coordinator crashes at 3? Wakes up,
  - Case 1: no GLOBAL_* in log → Send message to participants "abort"
  - Case 2: GLOBAL_ABORT in log → send message to participants "abort"
  - Case 3: GLOBAL_COMMIT in log → send message to participants "commit"
- Participant crashes at 4? On recovery, ask coordinator what happened and commit or abort

This is another example of the idea of a basic atomic operation. In this case – commit needs to "happen" at one place

Liveness

Limitation of 2PC – what if coordinator crashes during 3 and doesn't wake up? All nodes block forever

You would like to be able to declare the coordinator dead and either commit or abort the transaction (so that we can release the locks and move on...)
-- e.g., crash during transfer of cash from my account at BofA to your account at Wells Fargo -- my account should not be locked forever!

Termination Protocol

What if a participant time out waiting in step 4 for coordinator to say what happened. It can make some progress by asking other participants

1. if any participant has heard "GLOBAL_COMMIT/ABORT", we can safely commit/abort
2. if any participant has said "VOTE_ABORT" or has made no vote, we can safely abort
3. if all participants have said "VOTE_COMMIT" but none have heard "GLOBAL_*", can we commit?
   A: **no** – coordinator might have written "GLOBAL_ABORT" to its disk (e.g., local error or timeout)
   Turns out – 2PC always has risk of indefinite blocking

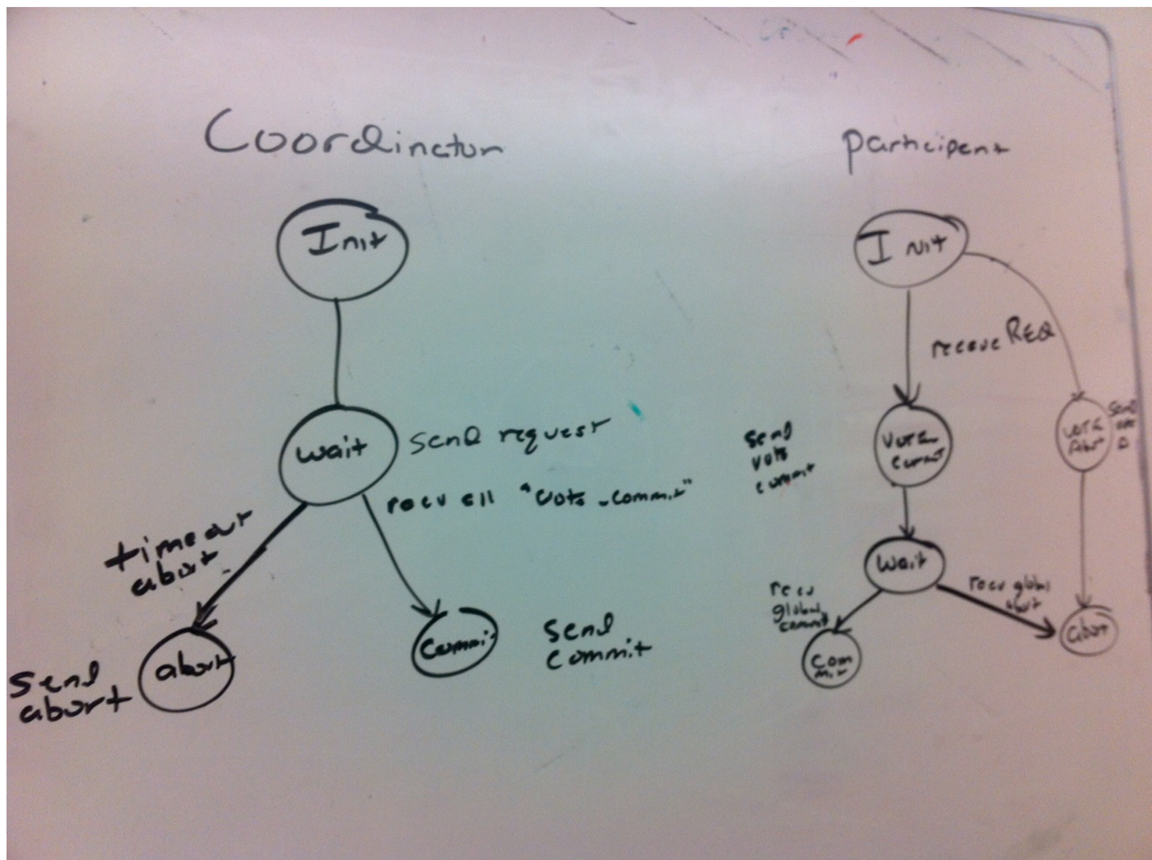
   Problem: Can't tell if coordinator has crashed (in which case, OK to abort) or is just slow/disconnected/temporarily crashed (in which case we need to do whatever it said to do)

-- 2PC has undesirable property -- if nodes can permanently fail it can block indefinitely if coordinator fails, even if all participants are up and able to communicate

3PC/Non-blocking commit

Crash at wrong time --> 2PC is stuck forever (requires manual intervention to fix and restart)

Fundamental problem: nodes can be in a state where they could transition directly to either commit or abort --> if can't talk to coordinator, don't know which should happen



3PC

-- always ensure that working participants can complete, as long as a majority are functioning

-- assumes reliable network -- max delay from when message sent to received

-- If you are paranoid enough about the corner case for 2pc, then "reliable network" is an aggressive assumption; better to use asynchronous, unreliable network model (e.g., Paxos; see below)

Key idea: Get rid of direct transition from "uncertain" to "commit" or "abort"

--> 3PC can complete as long as at most one node is unresponsive (disconnected, slow, or crashed)

"precommit" state -- process knows it will commit unless it fails

1. Coordinator sends VOTE-REQ to all participants

2. Participant recvs VOTE-REQ and logs and sends decision (VOTE_COMMIT or VOTE_ABORT); if VOTE_ABORT, done.
Timeout: log, send VOTE_ABORT
State: COMMIT->UNCERTAIN; ABORT->ABORTED

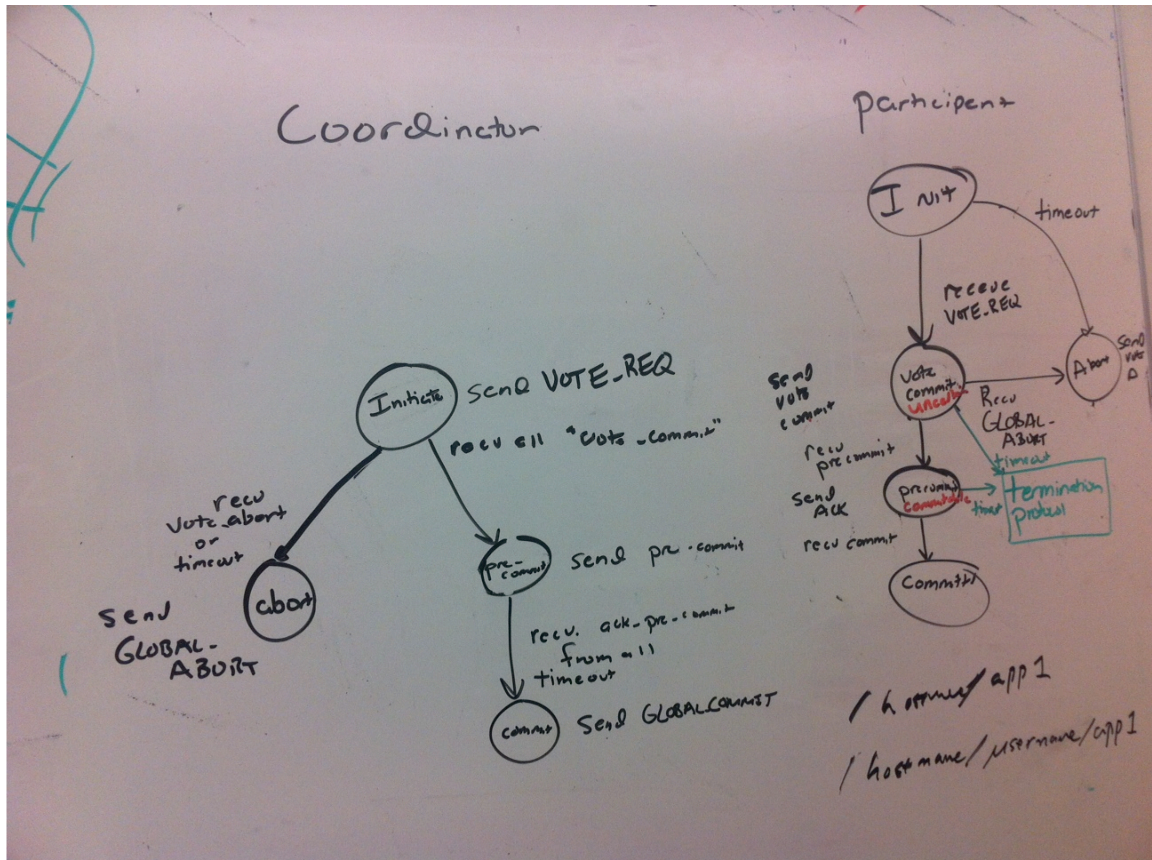
3. (a) If recv VOTE_COMMIT from all participants, log PRE-COMMIT, and send PRE-COMMIT to all participants
(b) recv VOTE_ABORT, log GLOBAL_ABORT and send GLOBAL_ABORT to all participants
Timeout: log, send GLOBAL_ABORT

4. Participant recvs PRE-COMMIT or GLOBAL_ABORT, logs it, and sends ACK to coordinator
Timeout: See below
State: PRE-COMMIT->COMMITABLE; GLOBAL ABORT -> ABORTED

5. Coordinator receives all ACKs --> log GLOBAL_COMMIT and send GLOBAL_COMMIT to all participants
Timeout: log, send GLOBAL_COMMIT

6. Participant recvs GLOBAL_COMMIT, logs it, and is done
Timeout: See below
State: COMMITTED

Strange: receiver knows what messages will be before receiving them. Why send them at all?
B/c messages allow nodes to track other nodes progress...
-- Key idea: Make sure no process is COMMITTED while any process is UNCERTAIN



Termination protocol -- timeout at step 4 or 6
I. Election protocol --> new coordinator
II. New coordinator sends STATE_REQ to all (live) participants
III. Coordinator collects responses
(a) Some process ABORTED --> log and send GLOBAL_ABORT to all
(b) Some process COMMITTED --> log and send GLOBAL_COMMIT to all
(c) All processes UNCERTAIN --> log and send GLOBAL_ABORT tot all
(d) All processes COMMITTABLE --> log and send PRE_COMMIT to all; continue steps 4, 5, 6 above

Key idea: Participants are always in compatible states:

	Aborted	Uncertain	Committable	Committed
Aborted	Y	Y	N	N
Uncertain	Y	Y	Y	N
Committable	N	Y	Y	Y
Committed	N	N	Y	Y

[table from: Chapter 7, "Concurrency Control in Database Systems", Bernstein, Hadzilacos, Goodman]

Better than 3PC -- paxos; also allows unreliable network and majority progress

BUT

Requires reliable network, no spurious timeouts

Can we avoid this? Yes
(a) Good protocol: Paxos
(b) I'll show a simple variation
[[see 3pc.txt]]

Bottom line

If you come to a place where you need to do something across multiple machines, don't hack

- use 2PC (or Paxos)
- if 2PC, identify circumstances under which indefinite blocking can occur (and decide if acceptable engineering risk)

In practice 2PC usually good enough – but be aware of the limits

up until recently non-blocking commit was seldom used in practice; recently, becoming not uncommon

QUESTION: is 2PC "at most once", "at least once", "exactly once" or "none of the above"?

6. Persistent message queues

MQSeries, etc.

Use 2-phase commit for message passing – guarantee exactly once delivery even across machine failures, long partitions

```
Send:
    Add msgID++, msg to log
    Send <msgID, msg> on NW (keep repeatedly sending
                                          all items in log)

Recv <msgID, msg>
    If <msgID> != largest stored msgID + 1
        If <msgID> <= largest stored msgID
            Send ack <msgId> to sender;
        Drop message;
        break;
    Add <msgId, msg> to log
    Send ack <msgId> to sender

Recv ack:
    Remove <msgID, msg> from log and stop retransmission

Process next msg:
    Transaction begin
        remove next msg from log
        process message
    transaction commit
```

E.g., AFS consistency state recovery – how would this now work?

QUESTION: is basic persistent message queue “at most once” “at least once” “exactly once” or “none of the above”?

How would you make it “exactly once?” (combine “at least once” with local transaction?)

7. Summary

RPC – “transparent” way to change local program into distributed program

- Generalization RMI, CORBA, SOAP – object-oriented versions of this

Case against RPC – RPC provides wrong abstraction – implies that local and remote programs can be/should be similarly structured

- focuses attention/abstraction on “common case” of everything works
- Some argue – this is wrong way to think of distributed programs. “Everything works” is the easy case –RPC encourages you to think about that case. But, the case of partial failures is the case you should focus your attention on.
- E.g., don’t assume that each request will get a reply, etc.
- “Exception paths” need to be as carefully considered as the “normal case” procedure call/return paths → RPC wrong abstraction

Lower-level message passing abstraction may help program writer avoid making implicit “everything usually works” assumption and may encourage structuring programs to handle failures elegantly

Persistent message queues can greatly simplify message passing (but at a potentially significant overhead.)