CS5412: TWO AND THREE PHASE COMMIT

Lecture XI

Ken Birman

Continuing our consistency saga

- Recall from last lecture:
  - Cloud-scale performance centers on replication
  - Consistency of replication depends on our ability to talk about notions of time.
    - Lets us use terminology like "If B accesses service S after A does, then B receives a response that is at least as current as the state on which A's response was based."
    - Lamport: Don't use clocks, use logical clocks
    - We looked at two forms, logical clocks and *vector* clocks
  - We also explored notion of an "instant in time" and related it to something called a consistent cut

Next steps?

- We'll create a second kind of building block
  - Two-phase commit
  - It's cousin, three-phase commit
- These commit protocols (or a similar pattern) arise often in distributed systems that replicate data
- Closely tied to "consensus" or "agreement" on events, and event order, and hence replication

The Two-Phase Commit Problem

- The problem first was encountered in database systems
- Suppose a database system is updating some complicated data structures that include parts residing on more than one machine
- So as they execute a "transaction" is built up in which participants join as they are contacted

… so what's the "problem"?

- Suppose that the transaction is interrupted by a crash before it finishes
  - Perhaps, it was initiated by a leader process L
  - By now, we've done some work at P and Q, but a crash causes P to reboot and "forget" the work L had started
    - Implicitly assumes that P might be keeping the pending work in memory rather than in a safe place like on disk
    - But this is actually very common, to speed things up
    - Forced writes to a disk are *very* slow compared to in-memory logging of information, and "persistent" RAM memory is costly
  - How can Q learn that it needs to back out?

The basic idea

- We make a rule that P and Q (and other participants) treat pending work as transient
  - You can safely crash and restart and discard it
  - If such a sequence occurs, we call it a "forced abort"

- Transactional systems often treat commit and abort as a special kind of keyword

A transaction

- L executes:

  Begin

  {

  Read some stuff, get some **locks**

  Do some **updates** at P, Q, R…

  }

  Commit

- If something goes wrong, executes “Abort”

Transaction…

- Begins, has some kind of system-assigned id
- Acquires pending state
  - Updates it did at various places it visited
  - Read and Update or Write locks it acquired
- If something goes horribly wrong, can Abort
- Otherwise if all went well, can *request* a Commit
  - But commit can fail. This is where the 2PC and 3PC algorithms are used

The Two-Phase Commit (2PC) problem

- Leader L has a set of places { P, Q, … } it visited
  - Each place may have some pending state for this xtn
  - Takes form of pending updates or locks held
- L asks "Can you still commit" and P, Q … must reply
  - "No" if something has caused them to discard the state of this transaction (lost updates, broken locks)
  - Usually occurs if a member crashes and then restarts
  - No reply treated as "No" (handles failed members)

What about "Yes"?

- If a member replies "Yes" it moves to a state we call *prepared to commit*
  - Up to then it could just abort in a unilateral way, i.e. if data or locks were lost due to a crash/restart (or a timeout)
  - But once it says "I'm prepared to commit" it must not lose locks or data. So it will probably need to force data to disk at this stage
  - Many systems push data to disk in background so all they need to do is update a single bit on disk: "prepared=true" but this disk-write is still considered costly event!

- Then can reply "Yes"

Role of leader

- So.... L sends out "Are you prepared?"
- It waits and eventually has replies from {P, Q, ... }
  - "No" if someone replies no, or if a timeout occurs
  - "Yes" only if that participant actually replied "yes"and hence is now in the prepared to commit state

- If all participants are prepared to commit, L can send a "Commit" message. Else L must send "Abort"
  - Notice that L could mistakenly abort. This is ok.

Participant receives a commit/abort

- If participant is prepared to commit it waits for outcome to be known
  - Learns that leader decided to Commit: It "finalizes" the state by making updates permanent
  - Learns that leader decided to Abort: It discards any updates
  - Then can release locks

Failure cases to consider

- Two possible worries
  - Some participant might fail at some step of the protocol
  - The leader might fail at some step of the protocol
- Notice how a participant moves from “participating” to “prepared to commit” to “commited/aborted”
- Leader moves from “doing work” to “inquiry” to “commited/aborted”

Can think about cross-product of states

- This is common in distributed protocols
  - We need to look at each member, and each state it can be in
  - The system state is a vector (S_L, S_P, S_Q, …)
  - Since each can be in 4 states there are 4^N possible scenarios we need to think about!

- Many protocols are actually written in a state-diagram form, but we'll use English today

How the leader handles failures

- Suppose L stays healthy and only participants fail
- If a participant failed before voting, leader just aborts the protocol
- The participant might later recover and needs a way to find out what happened
  - If failure causes it to forget the txn, no problem
  - For cases where a participant may know about the txn and want to learn the outcome, we just keep a long log of outcomes and it can look this txn up by its ID to find out
  - Writing to this log is a role of the leader (and slows it down)

What about a failure after vote?

- The leader also needs to handle a participant that votes "Yes" and hence is prepared, but then fails

- In this case it won't receive the Commit/Abort message
  - Solved because the leader logs the outcome
  - On recovery that participant notices that it has a prepared txn and consults the log
  - Must find the outcome there and must wait if it can't find the outcome information

- Implication: Leader must log the outcome *before* sending the Commit or Abort outcome message!

Now can think about participants

- If a participant was involved but never was asked to vote, it can always unilaterally abort

- But once a participant votes "Yes" it must learn the outcome and can't terminate the txn until it does
  - E.g. must hold any pending updates, and locks
  - Can't release them without knowing outcome

- It obtains this from L, or from the outcomes log

The bad case

- Some participant, maybe P, votes "Yes" but then leader L seems to vanish
  - Maybe it died… maybe became disconnected from the system (partitioning failure)
  - P is "stuck". We say that it is "blocked"
- Can P deduce the state?
  - If log reports outcome, P can make progress
  - What if the log doesn't know the outcome? As long as we follow rule that L logs outcome before telling anyone, safe to commit in this case

So 2PC makes progress with a log

- But this assumes we can access either the leader L, or the log.
- If neither is accessible, we're stuck
- In any real system that uses 2PC a log is employed but in many textbooks, 2PC is discussed *without* a log service. What do we do in this case?

2PC but no log (or can't reach it)

- If P was told the list of participants when L contacted it for the vote, P could poll them
  - E.g. P asks Q, R, S… "what state are you in?"
- Suppose someone says "pending" or even "abort", or someone knows outcome was "commit"?
  - Now P can just abort or commit!
- But what if N-1 say "pending" and 1 is inaccessible?

P remains blocked in this case

- L plus one member, perhaps S, might know outcome
- P is unable to determine what L could have done
- Worse possible situation: L is both leader and also participant and hence a single failure leaves the other participants blocked!

Skeen & Stonebraker: 3PC

- Skeen proposed a 3PC protocol, that adds one step (and omits any log service)

- With 3PC the leader runs 2 rounds:
  - "Are you able to commit"? Participants reply "Yes/No"
  - "Abort" or "Prepare to commit". They reply "OK"
  - "Commit"

- Notice that Abort happens in round 2 but Commit only can happen in round 3

State space gets even larger!

- Now we need to think of 5^N states
  - But Skeen points out that many can't occur
  - For example we can't see a mix of processes that are in the Commit and Abort state
    - We could see some in "Running" and some in "Yes"
    - We could see some in "Yes" and some in "Prepared"
    - We could see some in "Prepared" and some in "Commit"
  - But by pushing "Commit" and "Abort" into different rounds we reduce uncertainly

3PC recovery is complex

- Skeen shows how, on recovery, we can poll the system state
- Any (or all) processes can do this
- Can always deduce a safe outcome… provided that we have an *accurate failure detector*
- Concludes that 3PC, without any log service, and with accurate failure detection is non-blocking

Failure detection in a network

- Many think of Skeen's 3PC as a practical protocol
- But to really use 3PC we would need a perfect failure detection service that never makes mistakes
  - It always says "P has failed" if, in fact, P has failed
  - And it never says "P has failed" if P is actually up
- Is it possible to build such a failure service?

Notions of failure

- This leads us to think about failure "models"

Best: "Fail-stop" with trusted notifications

Many things can fail in a distributed system

- Network can drop packets, or the O/S can do so
- Links can break causing a network partition that isolates one or more nodes
- Processes can fail by halting suddenly
- A clock could malfunction, causing timers to fire incorrectly
- A machine could freeze up for a while, then resume
- Processes can corrupt their memory and behave badly without actually crashing
- A process could be taken over by a virus and might behave in a malicious way that deliberately disrupts our system

Worst: Byzantine

“Real” systems?

- Linux and Windows use timers for failure detection
  - These can fire even if the remote side is healthy
  - So we get “inaccurate” failure detections
  - Of course many kinds of crashes can be sensed accurately so for those, we get trusted notifications

- Some applications depend on TCP, but TCP itself uses timers and so has the same problem

Byzantine case

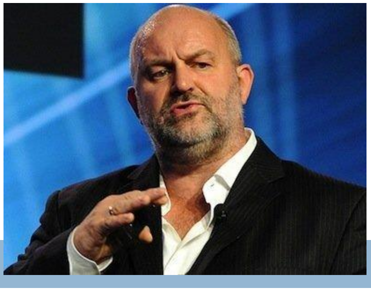
- Much debate around this
- Since programs are buggy (always), it can be appealing to just use a Byzantine model. A bug gives random corrupt behavior... like a mild attack
- But Byzantine model is hard to work with and can be costly (you often must "outvote" the bad process)

Failure detection in a network

- Return to our use case

- 2PC and 3PC are normally used in standard Linux or Windows systems with timers to detect failure
  - Hence we get *inaccurate* failure sensing with possible mistakes (e.g. P thinks L is faulty but L is fine)
  - 3PC is also blocking in this case, although less likely to block than 2PC
  - Can prove that any commit protocol would have blocking states with inaccurate failure detection

World-Wide Failure Sensing



- Vogels wrote a paper in which he argued that we really could do much better
  - In a cloud computing setting, the cloud management system often "forces" slow nodes to crash and restart
    - Used as a kind of all-around fixer-upper
    - Also helpful for elasticity and automated management
  - So in the cloud, management layer is a fairly trustworthy partner, if we were to make use of it
    - We don't make use of it, however, today

The Postman Always Rings Twice

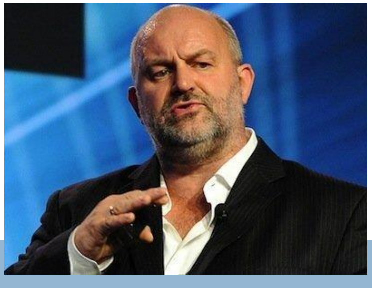
- Suppose the mailman wants to see you…
  - He rings and waits a few seconds
  - Nobody comes to the door… should he assume you've died?

- Hopefully not
- Vogels suggests that there are many reasons a machine might timeout and yet not be faulty

Causes of delay in the cloud

- Scheduling can be sluggish
- A node might get a burst of messages that overflow its input sockets and triggers message loss, or network could have some kind of malfunction in its routers/links
- A machine might become overloaded and slow because too many virtual machines were mapped on it
- An application might run wild and page heavily

Vogels suggests?



- He recommended that we add some kind of failure monitoring service as a standard network component
- Instead of relying on timeout, even protocols like remote procedure call (RPC) and TCP would ask the service and it would tell them
- It could do a bit of sleuthing first… e.g. ask the O/S on that machine for information… check the network…

Why clouds *don't* do this

- Hamilton: In the cloud our focus tends to be on keeping the "majority" of the system running
  - No matter what the excuse it might have, if some node is slow it makes more sense to move on
  - Keeping the cloud up, as a whole, is way more valuable than waiting for some slow node to catch up
  - End-user experience is what counts!

- So the cloud is <u>*casual*</u> about killing things
- ... and avoids services like "failure sensing" since they could become bottlenecks

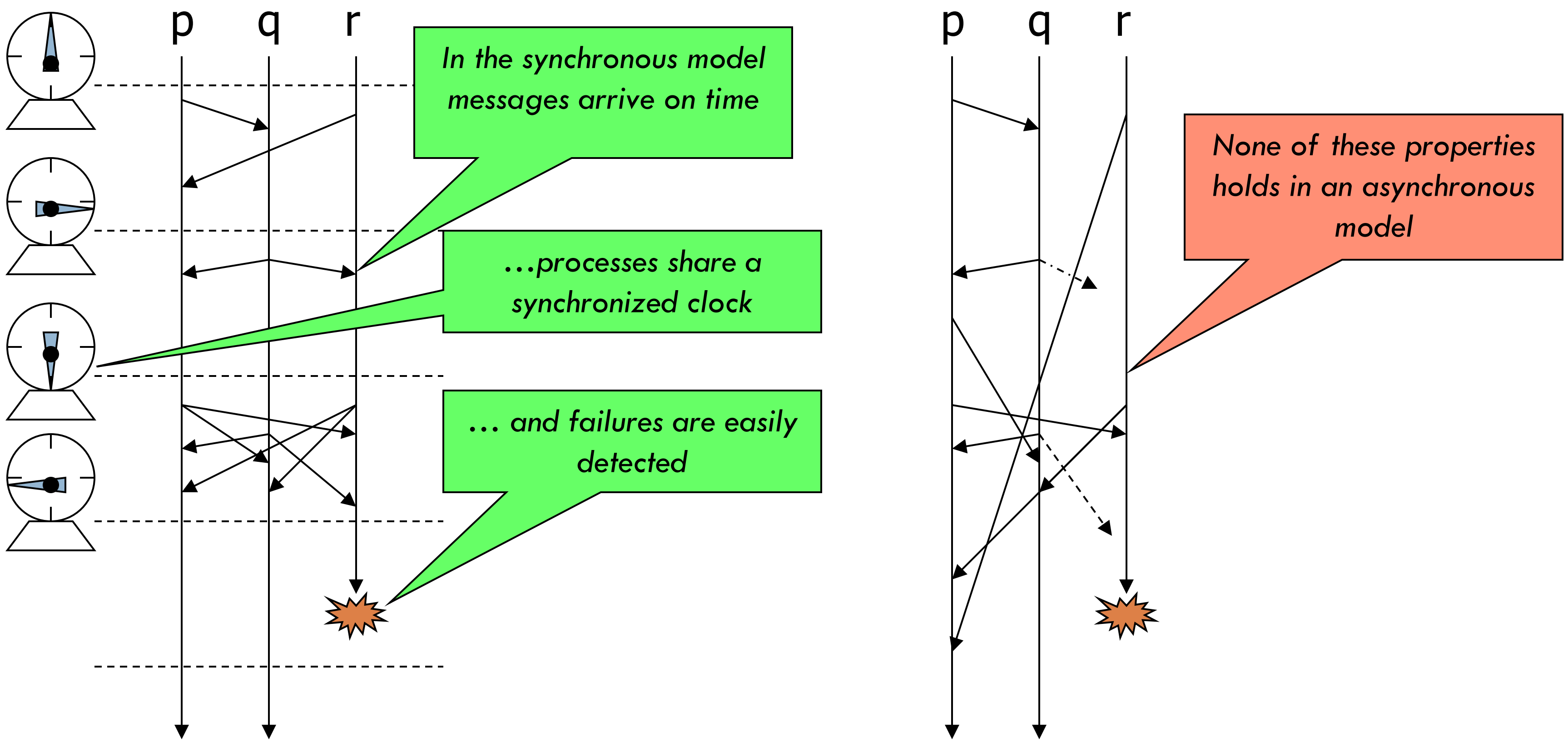
Also, most software is buggy!

- A mix of “Bohrbugs” and “Heisenbugs”
  - Bohrbugs: Boring and easy to fix. Like Bohr model of the atom
  - Heisenbugs: They seem to hide when you try to pin them down (caused by concurrency and problems that corrupt a data structure that won’t be visited for a while). Hard to fix because crash seems unrelated to bug
- Studies show that pretty much all programs retain bugs over their full lifetime.
  - So if something is acting strange, it may be failing!

Worst of all… timing is flakey

- At cloud scale, with millions of nodes, we can trust timers at all
- Too many things can cause problems that manifest as timing faults or timeouts
- Again, there are some famous models… and again, none is ideal for describing real clouds

Synchronous and Asynchronous Executions



Reality: neither one

- Real distributed systems aren't synchronous
  - Although a flight control computer can come close
- Nor are they asynchronous
  - Software often treats them as asynchronous
  - In reality, clocks work well… so in practice we often use time cautiously and can even put limits on message delays
- For our purposes we usually start with an asynchronous model
  - Subsequently enrich it with sources of time when useful.
  - We sometimes assume a "public key" system. This lets us sign or encrypt data where need arises

Thought problem

- Ron and Hermione will meet for lunch. They'll eat in the cafeteria unless both are sure that the weather is good
  - Hermione's cubicle is in the crypt, so Ron will send email
  - Both have lots of meetings, and might not read email. So she'll acknowledge his message.
  - They'll meet inside if one or the other is away from their desk and misses the email.
- Ron sees sun. Sends email. Hermione acks's. Can they meet outside?

Ron and Hermione

Ron

Hermione

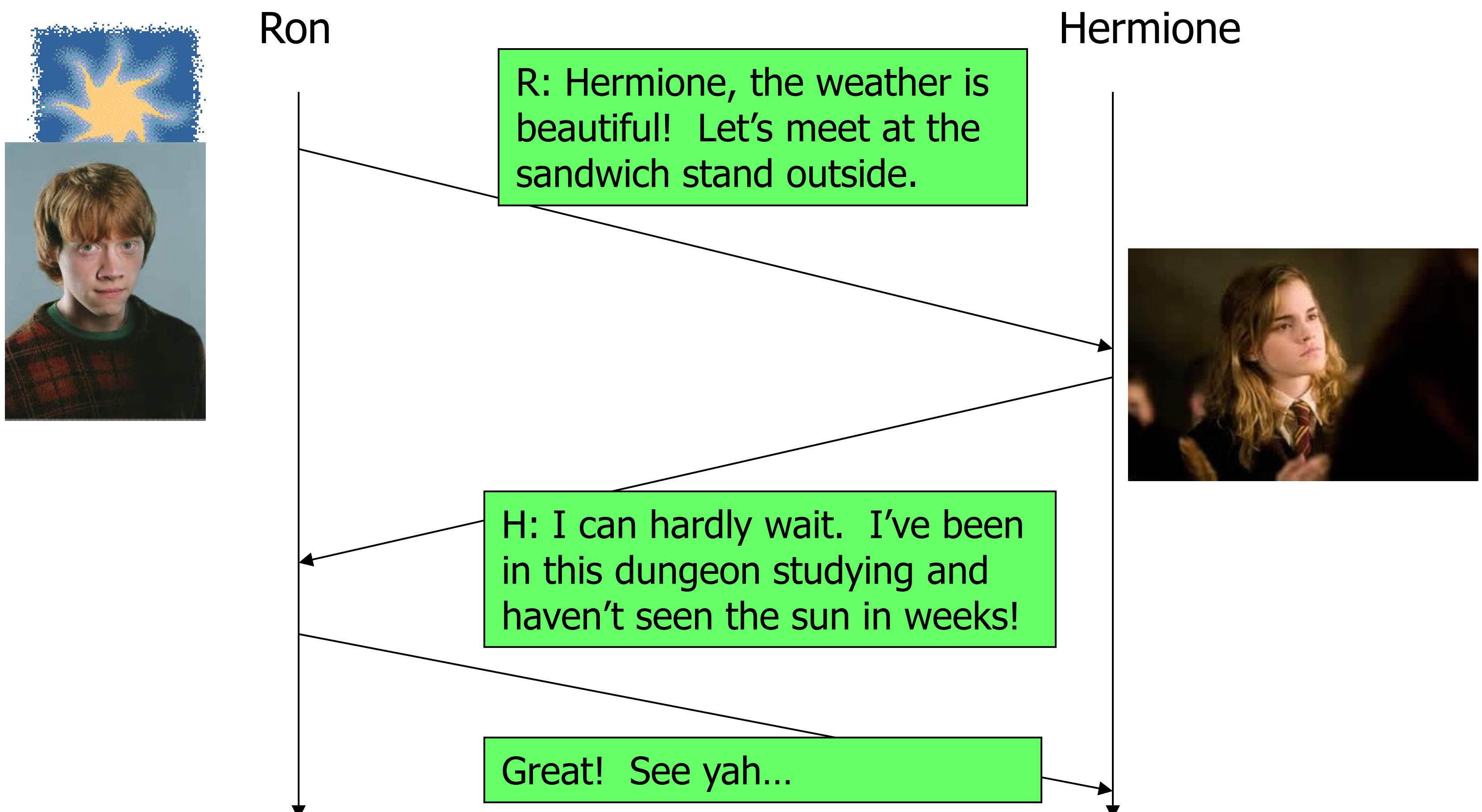
R: Hermione, the weather is beautiful! Let's meet at the sandwich stand outside.

H: I can hardly wait. I've been in this dungeon studying and haven't seen the sun in weeks!

They eat *inside!* Ron reasons:

- “Hermione sent an acknowledgement but doesn’t know if I read it
- “If I didn’t get her acknowledgement I’ll assume she didn’t get my email
- “In that case I’ll go to the cafeteria
- “She’s uncertain, so she’ll meet me there

Ron had better send an Ack



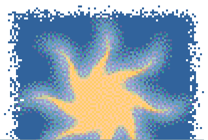
Why didn't this help?

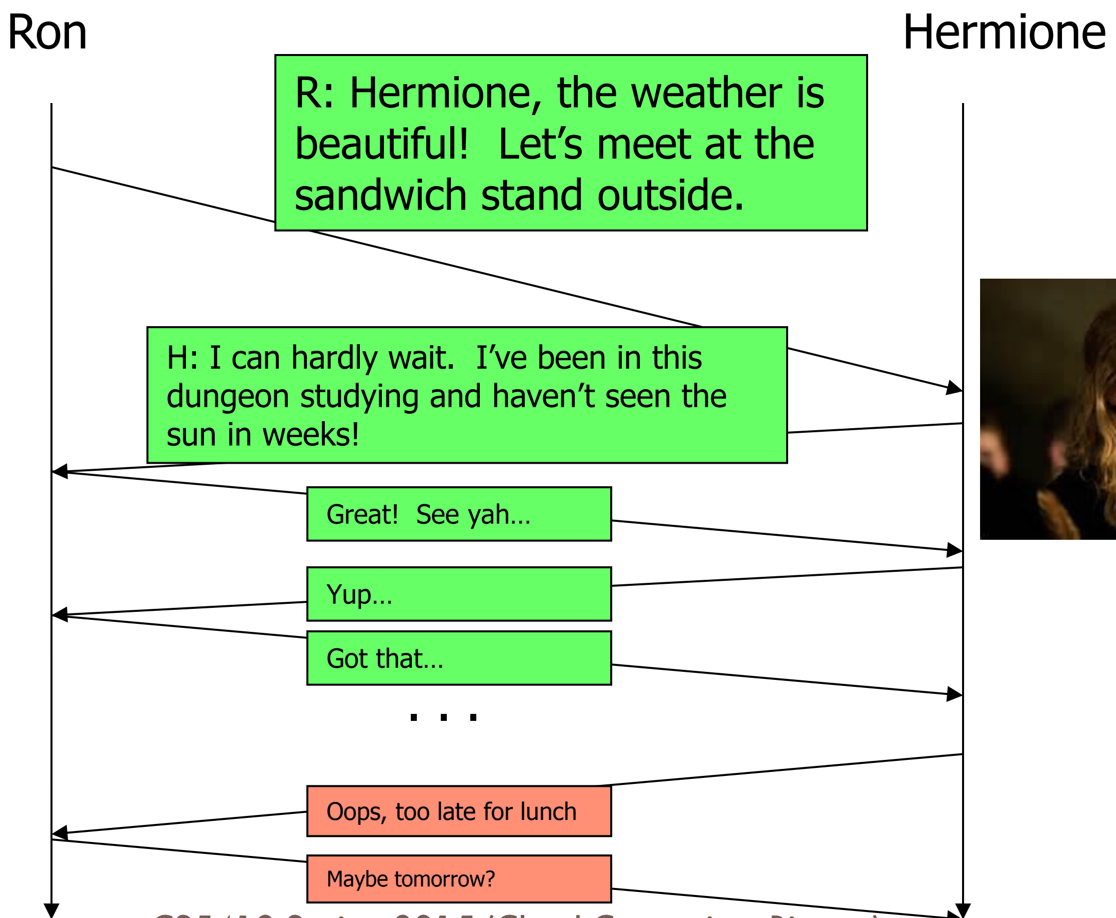
- Hermione got the ack… but she realizes that Ron won't be sure she got it
- Being unsure, he's in the same state as before
- So he'll go to the cafeteria, being dull and logical. And so she meets him there.

New and improved protocol

- Hermione sends an ack. Ron acks the ack. Hermione acks the ack of the ack....
- Suppose that noon arrives and Hermione has sent her 117'th ack.
  - Should she assume that lunch is outside in the sun, or inside in the cafeteria?

How Ron and Hermione's romance (should have) ended





H. K. Rowling

- “I’ve been feeling that I made a mistake… I really wonder if Hermione shouldn’t have ended up with Harry Potter”
- “I hope I’m not breaking some little girl’s heart saying this, but ever since I married her to Ron I’ve just been feeling that they aren’t right for each other…”

Moral of the story?

- Logicians are dull people and have miserable lives.

Your illogical approach to chess does have its advantages on occasion, Captain.
--Spock in Star Trek

Moral of the story?

- ~~Logicians are dull people and have miserable lives.~~
- The real world demands leaps of faith: pure logic isn't enough.
- For our computing systems, this creates a puzzle, since software normally behaves logically!

How do real people meet for lunch?

- They send one email, then go outside
  - Mishaps happen, now and then, but we deal with those.
  - In fact we know perfectly well that we can't achieve perfect agreement, and we cope with that
  - In some sense a high probability of meeting outside for lunch is just fine and we don't insist on more

Things we just can't do

- We can't detect failures in a trustworthy, consistent manner
- We can't reach a state of "common knowledge" concerning something not agreed upon in the first place
- We can't guarantee agreement on things (election of a leader, update to a replicated variable) in a way certain to tolerate failures

Back to 2PC and 3PC

- Summary of the state of the world?
  - 3PC would be better than 2PC in a perfect world
  - In the real world, 3PC is more costly (extra round) but blocks just the same (inaccurate failure detection)
  - Failure detection tools could genuinely help but the cloud trend is sort of in the opposite direction
  - Cloud transactional standard requires an active, healthy logging service. If it goes down, the cloud xtn subsystem hangs until it restarts
- We'll be using both 2PC and 3PC as a building block but not necessarily to terminate transactions.