AGREEMENT IN DISTRIBUTED SYSTEMS

We now turn to an important and broad problem in distributed systems: agreement. *A set of nodes in a DS need to agree on a decision, value of a variable, or order of events.* It is pervasive in distributed systems, coming in different flavors. We've seen examples of the problem already:

1) Lamport's distributed mutual exclusion protocol, discussed in the Clocks lecture: nodes agree on who has the lock at any time.
2) HW 2 asks you to design a replication protocol, where the primary and secondary replicas of a key/value store agree on the sequence of values written at each key.
3) Also in HW2, you are trying to design a protocol by which the nodes will agree on who is the primary at any time.
4) The ATM example from the RPC lecture: ATM front-end and banking service need to agree on whether to commit or abort my cash extraction transaction.

The agreement problem comes in two major flavors, each with its own protocols and application domains:[1]

- **Consensus problem:** participants need to agree on a value, but they are willing and capable to accept any value.
- **Atomic commitment problem:** participants need to agree on a value, but they have specific constraints on whether they can accept any particular value.

Where does each of the preceding DS examples (1)-4)) fit? 1)-3) are instances of consensus; 4) is an instance of atomic commitment. Distributed transactions on sharded databases, which we'll discuss in a future lecture, are also an example of the atomic commitment problem.

Each of these problems have intuitive correspondents in real life:

- A group's decision on *when* to meet is probably an atomic commitment problem, because each participant has his/her own calendar constraints.
- A group's decision on *where* to meet (say, which specific room on campus of those that are of suitable size) can probably be cast as a consensus problem: most likely no one cares where they meet, but they all need to agree on the same value. (There may be preferences on specific locations, but probably no hard constraints specific to each participant.)

Both of these problems are "hard," in the sense that it's impossible to guarantee agreement is reached in finite time under *all failure scenarios*. But the consensus problem is "doable" in practice: there are consensus protocols that can ensure the protocol makes progress under the vast majority of failure scenarios. That's not the case for the atomic commitment problem. The intuition behind that is that the atomic commitment problem imposes some extra constraints, and if participants have each their own constraints, then the only way to make progress is that we need all of them be up and running to make a decision on the value that's good for everyone (if there's one of course). It turns out that for the consensus problem, you only need *majorities* to choose a value. When you operate on majorities, you can make progress in the context of many more failure scenarios than if you require the entire set of nodes to agree on something.

[1] The way we differentiate between these flavors in these notes is not standard, but pedagogically, it is a good way to frame the discussion of the agreement protocols we will study in future lectures.

Interestingly enough, the protocols we use to solve atomic commitment problems (e.g., 2-phase commit) are conceptually simpler then the protocols we use to solve consensus problems (e.g., Paxos). In what follows, we'll formalize the consensus problem and prove it is impossible to solve in the asynchronous system model and under arbitrary failure scenarios.

THE CONSENSUS PROBLEM

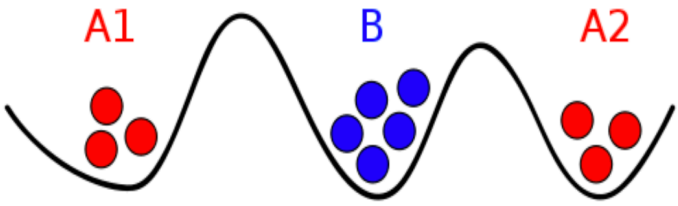
We'll start with a fictitious example from olden times (**I**). Then we'll define the consensus problem abstractly and in terms closer to distributed systems (**II**). We'll give examples of real-life instances of the problem, which abound, as previously noted (**III**). Finally, we'll end this lecture with a somber note: a result showing the impossibility of achieving consensus in asynchronous networks (**IV**). Despite this negative result, it's worth noting that future lectures tackle a very well known, approximate solution for consensus in asynchronous networks, which works very well in practice (or variations of it do) and is being used successfully in many distributed systems out there.

I. The Two Generals Problem

Two armies want to attack a fortified city. Both armies need to attack at the same time in order to succeed. The armies can only communicate through messengers, but unfortunately, the messengers can be captured, so message delivery is not reliable.

A solution must have three properties:

- **Consistency**: both armies decide to attack at the same time
- **Validity**: the time to attack was proposed by one of the armies
- **Termination**: each army decides to attack after a finite number of messages



Unfortunately, this is provably impossible in the general case. Let's see why.

How would one achieve the above properties? What's the protocol?
CASE 1: Known delays, delivery is reliable (known as **synchronous system model**):

- The protocol:
  - Pre-agree on either A1 or A2 generals proposing the time to attack. Say A1 is the one to propose. A2 will be the one to accept.
  - A1 sets the time of attack to communication delay + some extra time to account for A2's preparation for response.
- So the problem is solvable in this case.

CASE 2: Unknown delays OR unreliable delivery (known as **asynchronous system model**):

- Achieving consistency, validity, and termination is provably impossible.

- Proof sketch: need acks, but acks could be arbitrarily delayed/lost, too. Therefore I need more acks at every step. Therefore, one general can never be sure that the other will attack. So they can't be guaranteed to reach agreement.

II. Consensus Problem Formulation

For distributed systems, the consensus problem is defined as follows:

- A collection of processes, Pi.
- They propose values Vi (e.g., time to attack, client update, lock requests, …), and send messages to others to exchange proposals.
- Different processes may propose different values, but they can all accept any of the proposed values.
- Only one of the proposed values will be "chosen" and eventually (once all failures are addressed) all of the nodes learn that *one chosen value*.

Requirements:

- **consistency:** once a value is chosen, the chosen value of all working processes is the same.
- **validity:** the chosen value was proposed by one of the nodes.
- **termination:** eventually they agree on a value (a.k.a., a value is "chosen").

III. Examples from Real Life

1. The two generals problem – they agree on a time to attack
2. Agreeing on order of operations to a set of replicas (e.g., updates to a replicated DB).
3. One solution to the preceding problem is the primary/secondaries replication architecture. There are several replicas, one of which is designated as the primary. Reads and writes are executed at the primary, which establishes an order for all of these operations. There are variants of this, but they all reduce to one core consensus question: how does one choose the primary? Agreeing on who's the primary (or master or leader) among a set of replicas is a consensus problem. It's also called leader election.
4. Grabbing a lock for mutual exclusion. Previously we looked at a specific algorithm for distributed mutual exclusion. But at its core, it's a consensus problem, which can be addressed with a generic consensus algorithm.
5. Reliable and ordered multicast: all members of a group agree on a set and order of messages to receive.

All of these problems are instances of the same problem, which if we address, we've solved them all. Illustrate the consensus requirements in the preceding six cases.

IV. The FLP Impossibility Result

FLP (Fischer, Lynch, Paterson) result of 1985:

- **In an asynchronous system (unordered messages, unbounded communication delays, asynchronous processors), no protocol can guarantee consensus within a finite amount of time if even a single process can fail by stopping.**

Doesn't mean that consensus won't be achieved, just that it's not guaranteed. Also, as it's often the case with impossible problems, there are pretty good approximate solutions for the consensus problem, which will reach consensus in 99.999…% of the cases. We'll look at one solution, Paxos, in future lectures. For now, let's ask a more optimistic question:

When CAN consensus be guaranteed?

It turns out that consensus is possible in some circumstances, and impossible (not guaranteed to each all the requirements) in others. There are several axes:

- Processors: synchronous vs. asynchronous
  - bounded ratio of rate at which processors make forward progress
- Communication delay: bounded vs. unbounded
  - unbounded means messages have finite but unbounded delivery latency
- Ordered vs. unordered messages
- Broadcast vs. point-to-point messages

Processors	Message Order: Unordered		Ordered		Communication
Asynchronous	No	No	Yes	No	Unbounded
	No	No	Yes	No	Bounded
Synchronous	Yes	Yes	Yes	Yes	
	No	No	Yes	Yes	Unbounded
Transmission	Point-to-point	Broadcast		Point-to-point	

Table credit: Turek&Shasha, 1992

Two cases most distributed systems and protocols focus on:

- **asynchronous system**: messages are unordered, communication delay is unbounded, and processors are unbounded execution delays and can fail at any time;
- **synchronous system**: messages are unordered, communication delay is bounded and delivery is guaranteed, processors have bounded execution delays.

Consensus is solvable for synchronous but not for asynchronous. Asynchronous is more realistic.

Key Papers

For further details (and proper proofs) about these results, please consult the original papers that introduced them:

- FLP paper: Michael Fischer, Nancy Lynch, and Michael Paterson. *Impossibility of Distributed Consensus with One Faulty Process.* In *Journal of the ACM*, 1985.

- Paper that mapped possible/impossible settings for distributed consensus (the preceding table): John Turek, Dennis Shasha. *The many faces of consensus in distributed systems.* In *IEEE Computer*, 1992.

Acknowledgements

The preceding notes contain portions adapted from the University of Washington distributed systems course, Steve Gribble's edition:
https://courses.cs.washington.edu/courses/csep552/13sp/lectures/5/intro.pdf.