Lecture 2

Consensus Basics

- recap: building the computer in the sky (i.e., what are we trying to build, again?)
  - one way to think about it: a shared computer but with its hardware swapped out for the Internet
  - another way: put a virtual machine on top of the Internet
  - can again visualize as a three-layer stack:
    - top layer: user-facing applications (in this context, sometimes called "smart contracts"), e.g. Uniswap, OpenSea, Farcaster, etc.
    - bottom layer: the Internet (in effect, Web3 infrastructure piggybacks on existing Internet infrastructure)
    - intermediate layer: blockchain protocol (e.g., Ethereum, Solana, etc.)
  - strong parallels between the role of a blockchain protocol and that of the operating system of a computer:
    - acts as a "master program" that coordinates all applications/smart contracts (e.g., determines which one is currently be executed by the virtual machine)
    - exports a VM to application/smart contract developers; programmers can code applications as if their program would be run a physical machine corresponding to the exported VM
  - like the Internet, the product of coordination between many physical machines (ideally with different owners/operators)—in this sense, "decentralized"
  - why take many physical computers and use them to simulate only one? only the latter is decentralized and, e.g., suitable for attesting to ownership of digital assets
- goals for lecture 2:
  - blockchain protocol basics: validators, transactions, blocks, consensus vs. execution

        * need to start internalizing relevant language and concepts, will hear about these over and over
    - state machine replication (SMR), consistency, liveness
        * the specific consensus problem relevant to blockchain protocols and the basic guarantees that we want/need
    - challenges to consensus: faulty validators (crash vs. Byzantine), message delays (bounded vs. unbounded)
        * consensus is hard because validators may not behave as expected, and the communication network may not behave as expected
        * will give us a road map to solve SMR in more and more challenging and practically relevant settings
    - (if time) security thresholds
        * the fraction of validators at which consensus flips from possible to impossible
        * threshold depends on assumptions (could be 33%, 50%, etc.)

- some terminology:
    - validator = physical machine running a blockchain protocol [a.k.a. "node"]
        * crucial to have many validators (meaning of "decentralization")
        * cf., Web servers (all running the http protocol)
        * canonically, for now, maybe 22 or 100 validators
    - transaction = user-submitted action
        * cf., a command like "crop an image" in Photoshop
        * from user perspective, the basic unit of computational activity
        * could be an action to be taken by the blockchain itself (analogous to directly asking your laptop's OS to do something) or by a smart contract/program running in its virtual machine
        * translates to a snippet of low-level code to be executed in the blockchain's virtual machine
    - block = a sequence of transactions
        * i.e., a batch of transactions, ordered
        * for now, allow blocks to have unbounded size (but if you want, think of e.g. 100 transactions in a block)

- responsibilities of a blockchain protocol:
    - **consensus:** decide on a sequence (a.k.a. "chain" of blocks)

        * topic of the next 5 lectures
        * since blocks are themselves sequences of transactions, a chain of blocks is a transaction sequence (which will then correspond to a sequence of operations carried out in the blockchain's VM)
        * crucial that all validators agree on the same sequence, else could disagree about the state of the blockchain's VM (e.g., on who owns what)
        * think of a blockchain protocol as running forever (like your desktop), validators should keep agreeing on new blocks as long as users keep submitting transactions
        * fundamentally, can think of the blockchain consensus layer as a sequencer: it gets slammed with transactions from lots of different clients and determines the ultimate order of operations (and also, if there isn't room for everybody, which transactions get included at all)
        * at the consensus layer, pay no attention to what the transactions might mean (they are just 0s and 1s), transaction semantics (as virtual machine instructions) are the job of the execution layer
    - **execution:** keep the state of the virtual machine up-to-date
        * topic of lectures 8 and 9 (concludes Part I)
        * when a new block is added to the running sequence chain, must carry out the corresponding lines of VM code on the VM (e.g., reading from and writing to virtual memory, etc.)
        * so, a validator (i.e., physical machine) has a local representation of the blockchain's virtual machine (e.g., perhaps a key-value store that represents the current state of the virtual machine's memory, etc.), and must update that local representation whenever there are new VM instructions to be executed

- Consensus: Getting Started
    - goal of the consensus section of the course is to develop a strong understanding of the most fundamental concepts that come up *all the time* when discussing blockchain protocols, like consistency-liveness trade-offs, synchronous vs. asynchrony, security thresholds in different settings, etc.
    - informally, the goal of consensus is to keep multiple validators in sync (e.g., on who owns what) despite failures and attacks (e.g., denial-of-service attacks, hacked validators, etc.)
    - the fundamental problem that must be solved by any blockchain protocol—the consensus component acts as the glue between the "Internet hardware" and the application-facing virtual machine [in effect, turns the Internet into a (decentralized) transaction sequencer]

- plan: start simple with relatively strong assumptions, gradually weaken assumptions and design more and more sophisticated, robust and practical protocols (culminating with Tendermint, a popular blockchain consensus protocol originating with Cosmnos and now used by many other projects)
- standing assumptions:
    - fixed and known set of n validators (e.g., $n = 22$ or $n = 100$), each with a known ID and IP address
        - i.e., when we design a protocol, can think of the validator set and their IP addresses as hard-coded into it
        - a.k.a. "permissioned" or "proof-of-authority" blockchains (i.e., randos can't just join the validator set)
        - amazingly, all the most famous blockchain protocols are "permissionless" (i.e., they *do* allow randos to join the validator set!), and accordingly will relax this assumption in Part III (proof-of-work vs. proof-of-stake, etc.)
        - blockchain protocol design already quite difficult in the permissioned case, as we'll see in Parts I and II
    - all validators have a common notion of time (i.e., "synchronized clocks")
        - in effect, a shared global clock (like GMT)
        - in practice, doable (at least to an accuracy acceptable for blockchain protocol-relevant timescales)
        - also, a protocol like Tendermint (see Lecture 6) can be modified (with modest additional complexity) to accommodate a bounded amount of heterogeneity across the clock rates of different validators
        - won't discuss this issue further in this course

- state machine replication (SMR)
    - lots of different consensus problems ("Byzantine agreement" probably the most famous); for a perpetually running blockchain protocol, *state machine replication (SMR)* is the most relevant consensus problem
    - "state machine" — like a DFA (as you might study in automata theory of compilers); for us, "state" = state of a blockchain's virtual machine (virtual memory contents, etc.), execution of a single VM instruction corresponds to a (deterministic) state transition
    - "replication" — all the validators perform the same sequence of state transitions (and so agree on the state of the VM)
    - definition of the SMR problem:
        - "clients" submit transactions (abbreviated "txs") to validators
            - canonical client: you, using e.g. an app on your phone on a software wallet on your desktop that interacts with a blockchain like Ethereum

        · canonical transactions: make a payment, mint an NFT, etc.
    * each validator maintains an append-only list of finalized transactions (a.k.a. a "log" or "history")
  - solution to the SMR problem: a *protocol* that guarantees *consistency* and *liveness* [three terms we need to define]
  - protocol = the code run by each validator (will see several concrete examples over forthcoming lectures)
    * local computations
    * receive messages from other validators and transactions from clients
    * send messages to other validators (remember, list of validators and their IP addresses are baked into the protocol)
  - consistency = all validators should agree on the history (i.e., on the same tx sequence)
    * OK if some validator lags behind the others and just needs to catch up (e.g., local histories $B_1 \to B_2 \to B_3$ and $B_1 \to B_2$)
    * but no disagreements about tx ordering are allowed (not OK: $B_1 \to B_2 \to B_3$ and $B_1 \to B_2 \to B_3'$)
    * equivalently: all validators' local chains should be prefixes of a common chain
    * equivalently (you check): if you superimpose all validators' local chains, there are no "forks"
  - note: consistency by itself trivial to achieve (just do nothing!) so also need:
  - liveness = every valid tx submitted by a client eventually added to validators' histories
    * in practice, also want a concrete bound on the delay until inclusion/finalization, known as the *latency* of the protocol (an important performance metric, think about it from the user perspective!)

• why is consensus hard?

  - let's try a first stab (which is not a bad idea, actually) and see what goes wrong
  - Protocol A:
    * target say one block per second
      [fast, but achievable by state-of-the-art protocols]
    (1) validators take turns as "leader" (round-robin, one per second)

      [idea: different validators know about different transactions; rather than compare notes, just elect a dictator to coordinate everybody]

      [because of our assumptions of a shared global clock and a known set of validators, everyone agrees on the leader of a given second without any communication]

(2) current leader decides on a block

    [e.g., the new transactions it has heard of, ordered by time of arrival]

(3) leaders sends its block to all the other validators

- note Protocol A would seem to consistent and live, all validators operate in lock-step, adding the same new block (namely, the leader's proposal) in each second
- what could go wrong?
- question: what if a validator doesn't hear from the current leader within one second? (not suppose to happen, but what if it does?)
    - perhaps due to a problem with the leader (e.g., it crashed)
    - perhaps due to a problem with the network (e.g., congestion)
- and these are indeed the key challenges to consensus
    - faulty validators (i.e., for whatever reason, don't behave as expected)
        - and protocol doesn't automatically know which ones are faulty (otherwise could just ignore the faulty and proceed with the rest)
    - unreliable communication network (i.e., for whatever reason, doesn't behave as expected)
- these challenges are unavoidable in practice, so much design consensus protocols to work despite them
- revised goal for SMR: a protocol that guarantees consistency and liveness, despite faulty validators and an unreliable communication network
- can therefore get easier/harder versions of the SMR problem by allowing less/more severe forms of faults and network unreliability

- faulty validators/unreliable network:
    - faulty validators (easy mode): crash faults
        - every validattor dutifully follows the protocol but may crash (forever) at some point
        - could crash e.g. while in the middle of sending a bunch of messages to all other validators
    - faulty validators (hard mode): Byzantine faults
        - history buffs: see "Byzantine Generals" paper for backstory
        - i.e., worst-case faults: Byzantine validators can act arbitrarily (pretend to have crashed, swap out the protocol code for malicious code, lie about what they know or the messages they've heard, etc.)
        - seems paranoid, no?

    - originally studied in 1980s by researchers who protocols robust to software (as opposed to hardware) faults—not clear how to model a machine with buggy software, so ideally don't commit to any one model and design a protocol that works for arbitrary behavior
    - true killer app of Byzantine fault-tolerant consensus is blockchain protocols (especially those securing valuable assets), where validators might literally be controlled by bad actors (directly, or through hacking/intrusion) who want to interfere with the protocol (appropriate to be paranoid, as it is in e.g. cryptography)
    - lots of other fault/adversary models studied as well, we'll only have time for the crash/Byzantine cases
  - unreliable network (easy mode): synchronous network
    - a priori bounded delays — i.e., for a known parameter Δ, every message delivered within Δ time steps (e.g., perhaps corresponding to 1 or 2 seconds)
    - message delivery time still unpredictable (anywhere between 1 and Δ), but bounded
    - can hard-code Δ into the protocol description, if desired
    - reasonable assumption for the Internet on a good day (and the bigger the value of Δ, the more likely the assumption is true be true), but not when there are network outages or attacks (e.g., a denial-of-service attack)
  - unreliable network (hard mode): asynchronous network
    - all messages eventually delivered (i.e., delays finite, but could be arbitrarily large)
    - as with Byzantine faults, this is a pessimistic assumption, but the Internet is unpredictable place, might not want to commit to any one model of it
  - (we'll actually work a lot in a hybrid synchronous-asynchronous setup known as partial synchrony, see Lecture 4)

- road map: (will solve successively harder and more practical versions of the SMR problem)
  - Lecture 3: SMR with crash faults and a synchronous network
  - Lecture 4: SMR with crash faults and an asynchronous (actually, partially synchronous) network
    - will learn the essence of a famous protocol (used e.g. in lots of data centers) called Paxos (or Raft, in a more modern incarnation)
  - Lecture 6: SMR with Byzantine faults and an asynchronous (actually, partially synchronous) network

    - will learn the essence of a famous protocol (used in many blockchain projects) called Tendermint
  - expectations:
    - hopefully positive results (i.e., SMR protocols guaranteed to be consistent and live) at the beginning of the road map when in easy mode
    - gotta be ready for impossibility results (i.e., SMR unsolvable by any protocol) at the end of the road map when in hard mode
    - expect protocols to go from relatively simple to more sophisticated as we progress along the road map
  - aside: would obvious want a perfect protocol (simple, no assumptions, all the guarantees we want), but alas doesn't exist, any blockchain protocol must make some tough trade-offs
  - debates about the relative merits of different major blockchain protocols often boil down to different design choices about how to resolve these trade-offs (on which reasonable people can disagree)
  - next 5 lectures will give you a strong understanding of these (very practical) issues

- security thresholds [ran out of time, will cover in future lecture]
  - we've talked about two dimensions along which can make a consensus problem easier/harder (less/more severe faults and message delays); a third is to vary the number of misbehaving validators
  - intuition: [largely correct]
    - consensus is easy with 0% faulty validators
      - Protocol A already basically good enough, even in asynchrony (why?)
  - consensus is impossible with $\approx 100\%$ faulty validators
    - intuition clearest in the case of Byzantine faults
    - e.g., imagine that 98 Byzantine validators collude to convince the other two (non-faulty) validators of different blocks, how can the latter distinguish the one lone honest voice from the 98 liars?
      - some subtleties here, actually, but still accurate in spirit
  - security threshold = fraction of faulty validators at which solving the SMR problem flips from possible to impossible
    - value generally depends on your assumptions (e.g., on severity of faults and message delays)
    - typical values: 50% (i.e., strict non-faulty majority necessary and sufficient for consensus) or 33% (i.e., super-majority necessary and sufficient)

    * hear about these thresholds *all the time* in practice (e.g., on the possibility of some bad actor gaining control of 51% of Bitcoin's hashrate of 34% of staked ETH)

- one takeaway: a credible blockchain protocol requires a validator set of mostly reliable operators (you don't have trust any given validator, but you do have to believe that "on average" they are competently and correctly operating the protocol)