Foundations of Blockchains
Lecture #1: Introduction and Overview[*]

Tim Roughgarden[†]

1 The Upshot (of Lecture 1)

1. This lecture series is about the science and technology of blockchain protocols and the applications built on top of them, with an emphasis on fundamental principles rather than specific protocols.
2. We're witnessing a new area of computer science blossom in real time, and future generations will be jealous of your opportunity to get in on the ground floor.
3. Roughly 60% of the lectures are about "layer 1," including consensus (agreeing on a sequence of transactions) and compute (executing those transactions).
4. Roughly 20% of the lectures concern "layer 2," where the goal is to scale up layer-1 functionality (e.g., transactions processed per second) by orders of magnitude.
5. Roughly 20% of the lectures focus on the application layer (smart contracts and the user-facing applications that interact with them), with a particular focus on decentralized finance (DeFi).
6. For us, blockchains will not be about digital money (except as means to an end) but rather a new computing paradigm—a programmable computer that lives in the sky, that is not owned by anyone and that anyone can use.
7. We will always assume the existence of the Internet (semi-reliable point-to-point communication) and cryptography (specifically, cryptographic hash functions and secure digital signatures).
8. A user of a digital signature scheme uses their private key to sign messages, and such signatures can be verified by anyone who knows the corresponding public key.

[*]©2021–2022, Tim Roughgarden.

[†]Email: tim.roughgarden@gmail.com. The preparation of these notes was partially supported by a seed grant from the Columbia-IBM Center for Blockchain and Data Transparency.

9. Such a scheme is secure if it's impossible (or at least computationally infeasible) to forge signatures without knowing the private key.

10. State machine replication (SMR) is the consensus problem most immediately relevant to blockchains. In this problem, a bunch of nodes run a protocol to stay in sync on an ever-growing ordered sequence of transactions that have been previously submitted by clients.

11. A "solution" to the SMR problem is a protocol (i.e., code deployed at each node to control local computation and communication) that satisfies consistency (no pair of nodes ever disagrees on the relative order of a pair of transactions) and liveness (submitted transactions eventually get processed).

2 About These Lectures

This lecture series is about the science and technology of blockchain protocols and the applications built on top of them. There will not be any hype (OK, maybe two minutes of hype, below), and these lectures assume that you are already sufficiently interested in or curious about blockchains to spend a number of hours thinking hard about how they work. We will not discuss anything about investing (the future price of Bitcoin, etc.), nor about startups and entrepreneurship. We will avoid nitty-gritty engineering aspects (e.g., we won't being going line-by-line through any smart contract code), except as it's needed to appreciate more general issues.

Instead, this lecture series will focus on the fundamental principles of blockchain design and analysis, such as they are in 2022 (it's still early days. . .). The goal is to equip you with the tools and concepts to evaluate and compare existing technologies (cutting through the rampant marketing crap), understand fundamental trade-offs between different properties one would want from a protocol or application, and perhaps even create something new and important in the near future (because it's early days, you can have a tremendous impact on the area's future trajectory).

(Here's the hype part.) It's worth recognizing that we're currently in a particular moment in time, witnessing a new area of computer science blossom before our eyes in real time. It draws on well-established parts of computer science (e.g., cryptography and distributed systems) and other fields (e.g., game theory and finance), but is developing into a fundamental and interdisciplinary area of science and engineering in its own right. Future generations of computer scientists will be jealous of your opportunity to get in on the ground floor of this new area—analogous to getting into the Internet and the Web in the early 1990s. I cannot overstate the opportunities available to someone who masters the material covered in this lecture series—current demand is much, much bigger than supply.

And perhaps these lectures can also serve as a partial corrective to the misguided coverage and discussion of blockchains in a typical mainstream media article or water cooler conversation, which seems bizarrely stuck in 2013 (focused almost entirely on Bitcoin, its environmental impact, the use case of payments, Silk Road, etc.). A surprising number of people, including a majority of computer science researchers and academics, have yet to grok the modern vision of blockchains: a new computing paradigm with the potential to enable the next incarnation of the Internet and the Web, along with an entirely new generation of applications.

3 Overview of Lecture Series

3.1 The "Blockchain Stack"

To explain the organization of these lectures, it's useful to keep in mind a cartoon version of a "blockchain stack" that comprises a number of layers (Figure 1).[1] Starting from the bottom and moving on up:

(0) For us, layer 0 will basically be the Internet. That is, it provides at least a semi-reliable method for point-to-point communication between untrusted parties.

(1) Layer 1 is the *consensus layer*, and its job is to keep a bunch of computers (potentially scattered all over the globe) in sync, despite possible network failures and attacks. For example, Bitcoin and Ethereum are both layer-1 protocols. For smart contract platforms like Ethereum, it can also be useful to separate out the *compute layer*, with the consensus layer merely deciding which instructions (smart contract function calls, etc.) should be executed and in what order, and the compute layer responsible for actually carrying out those instructions and updating the global state. (E.g., full nodes running the Ethereum protocol participate simultaneously in both consensus and compute.)[2]

(2) For us, layer 2 will be the *scaling layer*. The goal here is basically to implement the same functionality exported by a layer-1 protocol, but a lot more of it. For example, Bitcoin and Ethereum can only process so many transactions (roughly 5 per second and 15-20 per second, respectively), and the point of a layer-2 protocol is to scale this capacity up by at least a couple of orders of magnitude.

(3) Finally, on top there is an *application layer* (as there is in the Internet stack), which refers to the applications built on the functionality provided by the previous layers. (Decentralized exchanges like Uniswap and NFT marketplaces like OpenSea are examples you might be familiar with.) Here again we're grouping together two logically distinct things, the actual smart contracts that live in the blockchain (sometimes called

[1]For an analogy, when you study computer networking, you learn about the layers of the Internet stack (network layer, transport layer, etc.). Ask 10 researchers in computer networking about the details of the stack and you'll get 11 different answers, and the situation is even more extreme in the blockchain world. . . .

[2]One can also further subdivide the functionality of the consensus layer into: (i) the selection of an ordered sequence of transactions; and (ii) the storage of that sequence. There is increasing momentum for separating out the second responsibility from the first via a "data availability layer." This idea will make more sense later, when we discuss approaches to scaling a smart contract platform such as Ethereum.

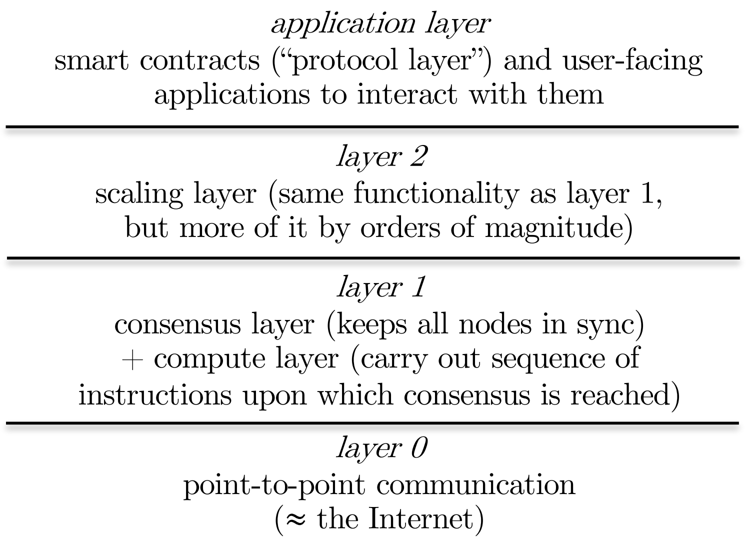
application layer
smart contracts ("protocol layer") and user-facing applications to interact with them
layer 2
scaling layer (same functionality as layer 1, but more of it by orders of magnitude)
layer 1
consensus layer (keeps all nodes in sync) + compute layer (carry out sequence of instructions upon which consensus is reached)
layer 0
point-to-point communication ($\approx$ the Internet)

Figure 1: A cartoon version of the "blockchain stack" to explain the organization of this lecture series. (Warning: not standardized, in flux, porous boundaries.)

the "protocol layer") and the user-facing icing on top (e.g., a Web interface). For example, Uniswap is really two things, its smart contracts and its Web interface to interact with those contracts. These are different things: For example, you can interact with the Uniswap contracts directly (as one would via a function call from a different contract, for example) rather than going through the standard Web interface.

Again, don't take this taxonomy too seriously—it's in flux and not at all standardized. For example, "layer 2 solution" often means something more specific than an arbitrary scaling solution (as we'll discuss in a future lecture). Also, the clean picture of a stack is misleading, as the boundaries between layers are porous. Ideally the layers would be insulated from each other, with protocols for one layer depending only on the provided functionality of the layer below (and independent of the latter's implementation). Currently this is far from true, unfortunately; for example, if you're implementing a decentralized exchange at the application layer, it's really useful to know exactly how the layer-1 consensus protocol works. Open question: Will we eventually come up with relatively clean separations between layers, or is there something about blockchains and their applications that forces layers to interact?

3.2 Outline of Lectures

We can describe the arc of this lecture series in terms of the layers above. We'll skip layer 0 and assume that we have the functionality of the Internet (such as it is). There's plenty of interesting work about blockchain-friendly layer-0 protocols (e.g., designing a peer-to-peer gossip protocol to make denial-of-service attacks harder), but it's outside our scope.

The majority of the lectures—perhaps 60% of them—will be about layer-1 protocols.

(Evidently, there's a lot to say here.) In the first 10 or so lectures, we'll discuss classical consensus protocols (and how they inspired Tendermint), Nakamoto/longest-chain consensus (one of Bitcoin's key innovations), proof-of-work and proof-of-stake sybil-resistance mechanisms, difficulty adjustment, and "selfish mining." The last four or so lectures on layer-1 protocols will be a deep dive on the world's biggest two blockchains, Bitcoin and Ethereum. Even though nitty-gritty details are not a focus of this lecture series, you'll learn quite a bit about how these two protocols work—both because it's useful to know, and because it's a prerequisite for understanding how layer-2 scaling protocols work (scaling solutions must inevitably adapt to the idiosyncrasies and limitations of the underlying layer-1 protocol).

Speaking of which, the next 20% or so of the lectures will be about layer-2 protocols (an extremely active area in 2021–2022). We'll discuss the Lightning network, the principal solution thus far for scaling up Bitcoin, and "rollups" (both "optimistic" and "zk/validity"), which appear to be the way forward for scaling up Ethereum. Time permitting, we'll also cover some newer layer-1 protocols that strive for high throughput out of the box, potentially obviating or at least delaying the need for layer-2 solutions.

The final 20% of the lectures will focus on the application layer, and primarily on "DeFi" (decentralized finance), which is where a lot of the action has been over the past couple of years. We'll have a couple lectures on DeFi primitives (stablecoins, price oracles) and a couple on applications built on top of those primitives (borrowing and lending, trading via automated market makers). We'll also talk about "MEV" (for "miner extractable value"), which is a great case study of the current lack of separation between the consensus and application layers.

3.3 Three Comments

Reflecting on the lecture series plan, let me single out three big differences between these lectures and a majority of the "introductions to blockchain" that you might come across.

A new computing paradigm. For us, blockchains are not about cryptocurrencies or payments per se. They're about a new computing paradigm—a programmable computer that lives in the sky, that is not owned by anyone (or rather, is owned by thousands of people all over the globe, including yourself if you like) and that anyone can use. (There might be a usage fee you have to pay, but there's no access control—you don't need anyone's permission.) When thought of this way, how could blockchains not unlock a totally new generation of applications?[3]

Not about digital money. Many blockchain introductions (especially the Bitcoin-focused ones) start by asking "what is money, anyway?" You learn about its uses (store of value, medium of exchange, unit of account), Bitcoin's interpretation as a form of "digital gold" with programmatic scarcity, and maybe even the rai stones from the island of Yap. This is

[3]Remember, for a couple decades there, no one really knew what to do with the Internet other than send emails and transfer files—it was "just" a very sped-up version of the postal service. Obviously, people eventually figured out some pretty cool things to build on top of it!

all cool stuff (if you haven't read about it, look it up), but in 2022 this is an outdated and overly narrow perspective. For us, cryptocurrency is the means, not the ends—a tool that helps us implement the functionality that we really want, by charging for blockchain usage and/or rewarding actors that contribute to the protocol and keep it running.

Principles over protocols. We won't start by explaining how one specific blockchain protocol like Bitcoin or Ethereum works. (Though we will learn a lot about those two protocols later.) Rather, we'll start with the fundamental principles of consensus protocol design and analysis (safety, liveness, etc.), and will then understand specific protocols through the lens of these principles.

4 Digital Signature Schemes

4.1 Permanent Assumptions

Consensus—keeping multiple machines (usually called *nodes*) synced up, despite failures and attacks—is the fundamental problem that must be solved by any blockchain protocol. Next lecture, we'll see how to solve this problem under a long list of assumptions. The subsequent lectures work hard to relax these assumptions. But there are two assumptions (both palatable, fortunately) that we'll never relax:

1. The Internet exists. (That is, there is a semi-reliable mechanism for point-to-point communication between untrusted parties.)

2. Cryptography exists. For the most part, we won't need anything too exotic, primarily the existence of cryptographic hash functions (discussed in a later lecture) and secure digital signature schemes (details below).

4.2 Definition of a Digital Signature Scheme

A *digital signature scheme* is defined by three (computationally efficient) algorithms:

1. **Key generation algorithm:** takes as input a random seed r, and returns a public key-private key pair (pk, sk). As the terminology would suggest, sk should be kept private (it will let you sign digital documents) while pk should be posted in public view (it allows anyone to verify your signature). The two keys are inextricably linked—indeed, typically pk can be derived directly from sk.

   [In a typical implementation, the algorithm might well take no input and generate its own random seed. E.g., type in `ssh-keygen` (with no arguments) at a Unix command line.]

2. **Signing algorithm:** takes as input a message m and a private key sk, and returns a signed version of the message (m, sig). (Here sig denotes bits that are appended to

the end of the message. The signature length is independent of the message length, and in a blockchain context is typically 520 bits or thereabouts.)

Note that the *signature depends on the message* (and on the identity of the signer, i.e., the provided private key). This is totally different from IRL signatures—your pen-and-paper signature is the same, no matter what the document contents. This property is obviously necessary for digital signatures—a document-independent signature could be easily copied and pasted to forge other signed documents.

3. **Verification algorithm:** takes as input a message m, someone's public key pk, and an alleged signature sig of m by the person who knows the corresponding private key sk. The algorithm answers "yes" or "no," according to whether the signature is valid (i.e., whether or not running the signing algorithm with message m and the private key sk corresponding to pk really would generate the signature sig).

Note that only the holder of the private key is in a position to run the signing algorithm, while everyone (who knows the public key) can run the verification algorithm.

4.3 Defining "Security"

In practice, secure digital signature schemes exist; in fact, some have been known since the late 1970s.[4] What do we mean by "secure"? For simplicity, we'll work with the most extreme version.

Assumption (ideal signatures): it is impossible to forge a signature without knowing the private key, even if you've seen a huge collection of examples of messages that have been signed with that key (with the example messages potentially chosen by the would-be attacker). That is, given such examples and a new (previously unseen) message m, without knowledge of the private key sk, it is impossible to generate a signature sig such that the verification algorithm would respond "yes" to the input (m, sig, pk) (where pk is the public key corresponding to sk).

This assumption is a good approximation of reality (assuming you use a well-implemented digital signature scheme with an appropriate key length). But as a theorist, it's my duty to point out that, strictly speaking, the assumption as stated is false. (If you're OK with taking the ideal signatures assumption on faith, feel free to skip the rest of this section.) For example, it's possible in principle to break a signature scheme by brute-forcing the private key—given a message m and signature sig generated by the private key sk, an adversary could enumerate all possibilities for sk and try each one, waiting until it manages to regenerate the signature sig (at which point the adversary knows that its current guess actually is sk), and then using the reverse-engineered private key to sign any other messages that it wants.

[4]RSA (Rivest-Shamir-Adelman) signatures are well known but not generally used in blockchain protocols on account of the relatively long lengths of its signatures and keys. ECDSA ("elliptic curve digital signing algorithm") is perhaps the most common, with Schnorr signatures gaining an increasing amount of traction.

Why doesn't this caveat bother us? Because, assuming the key length ℓ is at least several hundred bits, this brute-force attack would need to enumerate over 2^ℓ possibilities and would be completely unimplementable—the search literally wouldn't finish before the collapse of our sun. (For reference, the estimated number of atoms in the known universe is something like 2^{265}.) Turning this into a theorem thus requires a (modest) assumption, that everybody (including our adversaries) are computationally bounded. For example, we could assume that there exists some polynomial function p (say, $p(x) = x^{20}$ or $p(x) = x^{100}$) such that no adversary can perform more than $p(\ell)$ computer operations, where ℓ denotes the key length. (Every polynomial function is asymptotically smaller than every exponential function, so this assumption precludes brute-force search for sufficiently long keys.)

We're still not done, as who said that an adversary won't do anything more clever than brute-force search? When you study algorithms, you learn tons of problems for which brute-force search would take an exponentially long time and yet clever algorithms can cut through the clutter and identify a solution in polynomial (often, near-linear) time. (The single-source shortest-path problem and the minimum spanning tree problem are two classic examples.) Who's to say a clever adversary can't find a clever short cut and reverse engineer a private key sk from a collection of signed messages much faster than brute-force search? In response, we need to make more assumptions—called *(computational) complexity assumptions* or *hardness assumptions*—that certain problems cannot be solved efficiently (meaning in time polynomial in the input size).[5] For the digital signature schemes most commonly used in blockchain protocols, security is based on the assumption that there is no polynomial-time algorithm for the discrete logarithm problem in a suitably chosen group (i.e., the problem of reverse engineering the exponent x from the terms g and g^x, where g is a group generator and g^x denotes g multiplied by itself x times).

Finally, we need to deal with the fact that an attacker may use a randomized algorithm, and so in principle (with nonzero but extremely low probability) might get lucky and randomly guess our key. For this reason, any formal statement must tolerate a nonzero but negligible failure probability. Thus, the formal statement of the security of a digital signal scheme would look something like this: assuming a polynomial-bounded (randomized) attacker and suitable complexity assumptions (like the hardness of discrete log), an attacker that knows a collection of messages signed with the private key sk has only a negligible probability of generating a valid signature (the one that would be generated by signing with sk) on a new message m. (The example messages can be chosen by the attacker, and can depend on m.) Security statements of this form have been proved (under the stated assumptions) for the digital signature schemes used in practice.

Now that my theorist's conscience is clear, we'll go back to using the ideal signatures assumption for the rest of the lecture series. Again, this is a close approximation of reality.[6]

[5]It would of course be nice to actually prove mathematically that such assumptions are true, but doing so would resolve the "P vs. NP" conjecture (by showing that the complexity classes P and NP are different), an event that nobody is anticipating anytime soon.

[6]The digital signature schemes currently used in blockchain protocols are not "post-quantum secure," meaning that they will be broken once we have sufficiently large and powerful quantum computers. That's a ways off though, and meanwhile cryptographers have designed a new generation of digital signature schemes

4.4 Digital Signatures in Consensus Protocols

You can probably see why digital signatures are important for various blockchain use cases (e.g., signing off on a transfer of your funds), but they can also be tremendously useful in the design of the underlying consensus protocol. For example, they prevent a node A from credibly claiming to a node B that a third node C sent a message m to A some time in the past—as long as all the nodes are signing their messages, B will only believe A if A can exhibit a version of m that has been signed by C. With digital signatures, they can't make up fake messages from other nodes; all nodes can do is repeat without modification what they've heard. Unless otherwise noted, when we discuss consensus protocols, we will assume that every message sent by one node to another is signed by the sender.

Default assumption: in a consensus protocol, every node signs every message that it sends.

5 The State Machine Replication (SMR) Problem

5.1 Context

There are several notions of "consensus"; we'll see at least three. We'll start with the version most immediately relevant to blockchain protocols, called the state machine replication (SMR) problem.[7] What's a "state machine"? If you've ever studied automata (e.g., deterministic finite automata (DFAs)) you have a good sense of what it means (states and a state transition function). If not, some examples might help.

One of the old-school applications of the SMR problem is managing a replicated database. Think of a big company like IBM, with some database of valuable information. Suppose they want to charge customers for access to it (e,g., via queries and updates), but also want to promise 99.999% uptime. You won't get that level of uptime if the database is stored on a single computer (hardware and software failures being too common). An obvious idea for boosting uptime is to have multiple copies of the database, with each copy stored on a different machine and in a different location (so that machine failures are somewhat independent). But as soon as you have two or more copies of the data, you've got a new problem—keeping them in sync with each other. (E.g., if a customer writes an update to one copy, the update must also be reflected in the other copies, so that a corresponding read returns the same answer no matter which copy you ask.) Here, "state" means the current contents of the database, and each write to the database would effect a "state transition," moving from one state to a new one (which reflects the new write operation).

In a blockchain context, "state" will refer to the current status of the blockchain and its users (e.g., the current balance of each account, the local state managed by smart contracts,

that *are* post-quantum secure (under suitable assumptions), waiting to be deployed when needed.

[7]If the name sounds archaic, you're right, it's from the 1980s. It's kind of amazing—and a testament to the power and utility of theory and abstraction—that the same essential difficulty of certain applications from the 1980s are also central to the challenges of blockchain protocol design.

etc.). Executing a transaction (e.g., a payment from one account to another) effects a state transition (e.g., with the new state reflecting the post-transfer account balances). Unlike in the database example, where the only reason for replication is to increase uptime, in a blockchain context, the primary motivation for replication is "decentralization," meaning to ensure that responsibility for the protocol is distributed over many machines, with no one actor having significant control over its state and execution.

5.2 Problem Definition

In both the database and blockchain examples, the goal is to keep a bunch of nodes in sync, meaning all of them make the same sequence of state transitions (database operations/transaction executions) and hence agree on the current state of the state machine (database contents/blockchain state). This is the SMR problem. Summarizing:

1. There is a set of *nodes* responsible for running a consensus protocol, and a set of *clients* who may submit "transactions" to one or more of the nodes.

2. Each node maintains a local append-only data structure—an ordered list of transactions that only grows over time—which we'll call its *history.*[8]

Note that *order matters.* If two writes to a database conflict, it matters which one is carried out first. In a blockchain context, if two submitted transactions spend the same coins but with two different recipients (an attempted "double-spend"), it matters which transaction is executed first (as the second one will fail on the grounds of insufficient funds).

Informally, the goal in the SMR problem is to deploy code that keeps all the nodes in sync, with the same local histories (same ordered sequences of transactions). But what does this actually mean?

First, what form would a "solution" to the SMR problem take? Answer: a *protocol.* We won't bother with an overly formal definition, but think of a protocol as a piece of code that is to be run by each of the nodes. This code manages both the computations and the communications performed by the node as the protocol runs. Specifically, each node can:

- maintain local state, and perform local computations that depend on or affect that state;

- receive messages from other nodes and from clients;

- send messages to other nodes.[9]

[8]Many people use the word *ledger* for this data structure. To my taste, "ledger" too strongly connotes the use case of payments which, as we've said, seriously undersells the full potential of blockchain protocols as a new computing platform.

[9]Generally, nodes to not send messages to clients.

The code is event-driven, meaning that when some event occurs (e.g., receiving a new message from a client or another node), it can trigger a response from the node (e.g., some local computations followed be sending out new messages to one or more other nodes).[10]

What does it mean for a protocol to be a "correct" solution to the SMR problem?[11] That is, what guarantees do we want from a protocol? We can distinguish between *safety* guarantees, which promise that a certain bad event never happens, and *liveness* guarantees, which promise that a certain good event eventually happens. We'll focus on one of each.

Goal #1: Consistency. We say that a protocol satisfies *consistency* if all the nodes running it always agree on the history (i.e., the same ordered transaction sequence). Actually we'll be a little more flexible—if there's a node in Siberia that always finds out about the latest transactions later than everyone else, it's OK if its local history lags, as long as it's always a *prefix* of other nodes' histories (i.e., just needs to catch up). What *absolutely cannot happen* is for two nodes to disagree on the relative order of two different transactions. Consistency is the safety property promising that this bad event never occurs.

If we only cared about consistency, our lives would be easy. After all, the empty protocol (with all nodes maintaining an empty history forevermore) satisfies consistency! So we also need a guarantee that work eventually gets done.

Goal #2: Liveness. Every transaction submitted to at least one node is eventually added to every node's local history.[12]

Are there protocols that solve the SMR problem, in the sense of satisfying both consistency and liveness? As we'll see, the answer depends on a number of factors, including the reliability of the underlying communication network and the number of compromised nodes. In the following lectures you'll learn the key possibility and impossibility results for SMR consensus.[13] We'll eventually see how these theoretical results give us a lens through which to compare different layer-1 protocols (e.g., some of which favor liveness, others of which favor consistency). Next lecture, we'll assume a super-reliable communication network and give a protocol that solves the SMR problem even in the face of an overwhelming number of compromised nodes. Later lectures will discuss protocols that solve the SMR problem under weaker assumptions about the communication network (but stronger assumptions about the number of compromised nodes).

[10]Depending on the exact computational model, nodes may also be able to keep track of (global) time and use it in their decisions of what to compute and communicate (e.g., sending certain messages only after a timeout).

[11]With a single-shot problem like, say, single-source shortest-paths, it's clear what correctness means—an algorithm should always identify the actual shortest path in a given input graph. For protocols that run forever, without a clearly defined "output," defining "correctness" is a much more subtle problem.

[12]For now, think of all transactions as always being valid and eligible for inclusion. Obviously, liveness does not apply to invalid transactions (e.g., those lacking an appropriate digital signature or sufficient funds).

[13]It's also interesting to minimize the amount of computation and communication required by a protocol—and much of the past and present research literature focuses on exactly this—but such efficiency considerations will be largely outside our scope.