Lecture 16: Bridging BFT protocols with the longest chain protocol

Principles of Blockchains, University of Illinois,
Professor: Pramod Viswanath
Scribe: Suryanarayana Sankagiri

March 23, 2021

Abstract

In the past two lectures, we have studied BFT protocols that are an alternative to longest chain protocols. The BFT protocols offer strong safety guarantees (at times even when the network is not synchronous) while deprioritizing liveness; the exact reverse is true in the longest chain protocol. In this lecture, we study how to construct blockchains that have both protocols, BFT and longest chain, as constituents; the goal is to have both strong safety and liveness properties, along with forensic capabilities.

Introduction

Blockchain protocols must satisfy two basic properties: safety and liveness. Roughly speaking, safety says that all parties will have consistent views of the blockchain, while liveness says that new blocks will get included in the blockchain at a regular rate. Safety and liveness are jointly referred to as security properties.

For any given protocol, security properties cannot be guaranteed unconditionally. Rather, they are guaranteed to hold only under some assumptions, e.g., the system has honest majority (or 2/3 super-majority) and the network is synchronous. Naturally, we would like safety and liveness to hold under as large a set of conditions as possible; e.g., safety and liveness guarantees under 1/2 honest majority would be preferable over these guarantees under 2/3 honest majority. We begin by examining the different assumptions required for security by the different blockchain protocols we have seen in the lectures.

A major strength of the (Proof-of-Work) longest-chain protocol is that it works in a truly permissionless setting. Put differently, it handles variable participation levels; parties may join or leave the system at any time. The key point here is that it remains secure even if the number of active participants drops to one (as long as we have honest majority at all times). This property is called *adaptivity*. On the flip side, safety guarantees in the longest chain protocol are probabilistic, not deterministic. More importantly, the protocol does not remain secure during extended periods of asynchrony. During such periods, the adversary can easily build many blocks and overturn blocks that are k-deep, thereby violating the safety of the k-deep confirmation rule. It also loses liveness.

In the last two lectures, we have seen permissioned (committee-based) BFT protocols that offer deterministic safety (so-called *finality*). Crucially, the safety property holds even during extended periods of asynchrony. By definition, permissioned protocols do not have adaptivity; i.e., the protocol will not be live when not enough nodes participate (although safety is inviolable). Tables 1 and 2 summarize the properties of adaptive protocols (like the longest-chain protocol) and the finality-offering protocols (like HotStuff).

In summary:

Table 1: Adaptive rule

	Asynchrony	Synchrony
Static Participation	Neither	Safe, Live
Variable Participation	Neither	Safe, Live

Table 2: Finality-preserving rule

	Asynchrony	Synchrony
Static Participation	Safe	Safe, Live
Variable Participation	Safe	Safe

> The longest chain protocol prioritizes liveness over safety; BFT protocols prioritize safety over liveness. In other words, BFT protocols offer finality but not dynamic availability; longest chain protocols offer dynamic availability but not finality.

A natural question is whether it is possible to construct a blockchain protocol that offers *both* properties: finality *and* adaptivity. In this lecture, we will first see how *finality gadgets* give us a blockchain system with two different confirmation rules: one guaranteeing adaptivity, and the other, finality. Finality gadgets are interesting because they neatly combine a BFT protocol with the longest-chain protocol, in order to create a protocol with the best-of-both-worlds. We will also discuss the *CAP theorem*, which tells us that it is impossible to have a single blockchain protocol with both properties. This is a fundamental result with wide implications in distributed systems, some of which we discuss in the last part of this lecture.

Beyond adaptivity versus finality, there are other dimensions that contrast the longest-chain protocol from committee-based protocols. One important factor is the latency in confirming blocks. The Nakamoto consensus protocol is slow to confirm blocks. One must wait for blocks to be k-deep before they can be confirmed. Blocks arrive as per the mining rate, which is set proportional to the inverse of the maximum network delay Δ. Thus, the latency of the longest-chain protocol is $O(k\Delta)$. We have seen how protocols like Prism can help reduce this to $O(\Delta)$. However, protocols like HotStuff offer even better latency of $O(\delta)$, where δ is the real network delay (which can be much smaller than the worst-case upper bound). This property is called *responsivity*; the protocol responds to the real network delay. Note that not every committee-based protocol is responsive, e.g., Streamlet is not.

Once again, a natural question to ask is can we have a responsive protocol in a permissionless setting? One method to achieve this is via a protocol design called Hybrid Consensus, which neatly combines a BFT protocol with the longest chain protocol.

Hybrid Consensus

The main goal of the Hybrid Consensus approach is to develop a Proof of Work permissionless system with fast confirmation. In particular, we want protocols that are *responsive*, i.e., the confirmation delay is proportional to the true network delay than the estimated upper bound. In Algorand, which works in a Proof of Stake system, Verifiable Random Functions were used to elect a committee, which then ran a fast-confirmation protocol. Taking a cue from Algorand, the crucial task at hand is to elect a committee of a fixed size in a PoW system, and moreover, do so periodically.

Let us first see how to elect one committee of size csize. Let us run the Nakamoto consensus protocol until csize $+ k$ blocks are mined. Then all parties are guaranteed to agree on the first csize blocks (except with negligible probability). Each block includes the miner's public key. Lo and behold, we have elected a set of csize nodes (to be precise, csize public keys, multiple of which could be held by the same person/entity). This committee is chosen in a fair, decentralized manner

(due to PoW) and is agreed upon by everyone (because of the security of the PoW longest chain protocol). Once a committee is chosen, it executes a responsive BFT protocol (like HotStuff) to confirm transactions. Thus, transactions are actually confirmed by the BFT protocol, not the longest-chain protocol! Figure 1 illustrates this idea.

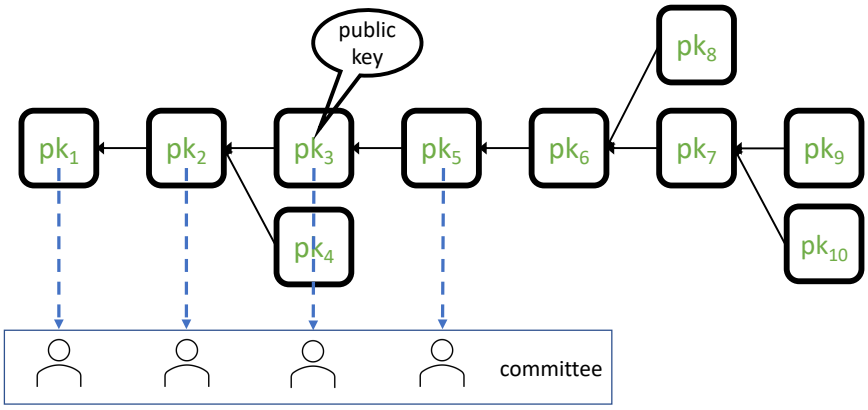

Figure 1: The Idea Behind Hybrid Consensus

Having covered the basic idea behind the protocol, we examine it in more detail. Firstly, note that even if we wanted to elect a single committee, honest miners cannot afford to stop mining; if they do, the adversarial miners can create a longer chain with purely adversarial blocks, which will then decide the committee. Secondly, to run a responsive BFT protocol like HotStuff, we would like an honest super-majority of $> 2/3$ csize parties in the committee. This must be reflected in the honest-to-adversarial ratio in the PoW protocol. In fact, if we use the plain longest chain protocol, we need a 3/4 honest majority, because of imperfect chain quality. Rather, if we use FruitChains, (see Lecture 7), we can do with a 2/3 honest majority.

It is important to be able to keep rotating the committee, for two reasons. First, we cannot rely on the committee to remain online continuously for an extended period of time. Secondly, an adversary can corrupt the committee after it is elected (although presumably, with some lag). By rotating the committee frequently, we can mitigate attacks by an adaptive adversary. The rotation is conducted in a straightforward manner: once a committee is formed, it keeps confirming transactions for a fixed period of time, until which time the PoW blockchain grows by csize more blocks. Once this happens, the old committee hands over the responsibility of transaction confirmation to the new committee. Note that all parties in the protocol have the same view of the committee.

The main advantage Hybrid Consensus brings over the longest-chain protocol is that it is a responsive protocol. Let us now examine some of its limitations. Firstly, if we would like a responsive BFT protocol, the adversarial threshold it can tolerate is 1/3, down from 1/2. This is inevitable; any responsive protocol can tolerate at most 1/3 Byzantine adversaries. While there are BFT protocols that tolerate 1/2 adversaries, their latency is $O(\Delta)$. Secondly, the protocol completely breaks down under asynchrony, because the committee election mechanism itself loses both safety and liveness. If there is no consensus on who belongs to the committee, there can be no consensus on the transactions confirmed by them. For similar reasons, security of the whole scheme is probabilistic, not deterministic. Lastly, honest committee members must remain active throughout the duration of time that it is a member of the committee. If a large fraction of them stop participating, the protocol will stall. Thus, in terms of guaranteeing safety and liveness under a wide variety of conditions, Hybrid Consensus actually does worse than the longest-chain protocol – it does not guarantee liveness under variable participation.

Finality Gadgets

A finality gadget, as the name suggests, is designed to provide deterministic safety guarantees, i.e., "finality" into a PoW blockchain. Ideally, the blockchain endowed with the gadget should provide finality in addition to, and not at the expense of, adaptivity. The natural approach is to combine a committee-based BFT protocol with the longest-chain protocol. After all, we are seeking "the best of both worlds", i.e., adaptivity, like the longest-chain protocol, and finality, like a BFT protocol. Let us examine how we may combine these two protocols to get the desired benefits.

A two-layer design

In its simplest form, a finality gadget is a layer-two committee-based BFT protocol, which runs on top of a (layer-one) longest-chain protocol. One can use any adaptive protocol instead of the longest-chain protocol, but for this lecture, we shall focus on the longest-chain protocol. To elaborate, there are two sets of nodes in the system: *miners* and *checkpointers*. Miners produce new blocks, carrying transactions, using PoW and the longest-chain rule. There could be any number of miners, but we assume that the mining rate of the system remains constant. The checkpointers are a distinguished committee of n nodes, separate from the miners. The checkpointers do not produce any blocks of their own. Rather, they execute a committee-based BFT protocol by voting on blocks produced by the PoW mining process. Thus, there are two consensus protocols being executed in parallel, to achieve consensus on the *same* set of blocks (transactions).

Checkpointing. The notion of checkpointing is new, and we elaborate on it a little more here. Suppose the longest chain protocol proceeds for some time, and the block tree grows to a certain height. Consider a block that is six-deep; a user could confirm it, with the hope that all future blocks would be descendants of this block. However, the confirmation guarantee would be probabilistic, as we know. How could we make the confirmation guarantee deterministic? In other words, how can we checkpoint this block?

It comes the checkpointing committee. The committee's job is to checkpoint blocks, by issuing *checkpoint certificates* for blocks. A checkpoint certificate for a block B is a collection of at least $2n/3$ signed messages of the form $\langle \mathsf{finalized}, B \rangle$; such a block is said to be checkpointed. These messages arise naturally in nearly all committee-based BFT protocols. Any node that receives this certificate is assured that this block is confirmed deterministically. This node could be a miner, a checkpointer, or simply an observer of the system. Naturally, the checkpoint blocks, issued by the layer-two gadget, should lie on the longest chain, to maintain consistency with the layer-one protocol. How can this be done?

The checkpointers keep track of the blocks being mined. At regular intervals, they execute a (single-shot) BFT protocol, where inputs to this protocol are blocks produced by the miners. Honest nodes input blocks that lie on the longest chain, but have not been checkpointed yet. Typically, blocks at a certain fixed depth are chosen; say, a depth of 6. This would ensure that if nodes have chains of the same length and have 6-deep common prefix, their inputs would be the same. Checkpointers exchange votes on each other's inputs, and finally agree on one particular input; this forms the checkpointed block. The overall system is illustrated in Figure 2.

Two ledgers and their properties. The salient feature of a finality-gadget based system is that it provides *two different confirmation rules*. Firstly, the k-deep rule continues to remain a viable confirmation rule in the system, since blocks are being mined as per the longest-chain protocol. A node that is simply observing the blocks mined in the system can easily follow this rule to confirm blocks. Such a node may remain completely oblivious to the checkpointing protocol. This rule provides *adaptivity*. The second confirmation rule comes from the checkpointers, who issue *checkpoint certificates* for certain blocks at regular intervals. This collection of messages certifies

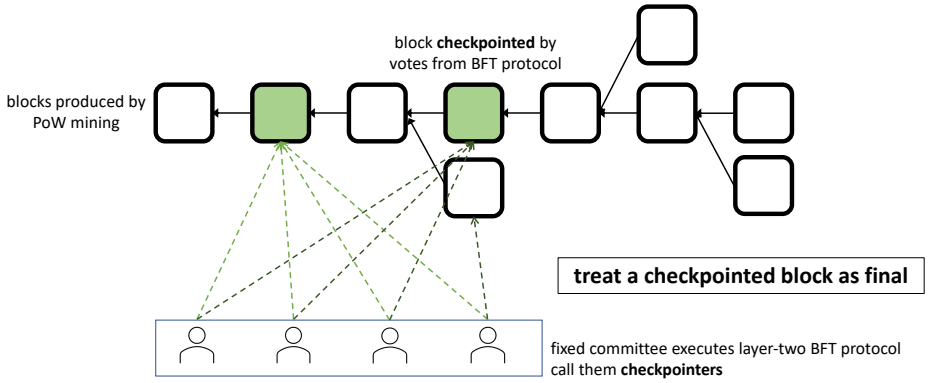

Figure 2: Finality gadget.

that block B has been agreed upon by the checkpointers. The confirmation rule for the finality gadget is to simply confirm the latest checkpointed block, and all its ancestors. This rule provides *finality*.

What does it mean for a confirmation rule to provide adaptivity or finality? To understand this, we need to appreciate the following points:

- Every confirmation rule generates a ledger: it is the sequence of all transactions in the order of the blocks confirmed by the particular rule.
- Over a long time, the two ledgers are consistent: all transactions that appear in one ledger also appear in the other, and they are in the same order.
- At any given point in time, one ledger could be slightly ahead of the other (i.e., confirm a few more transactions). Typically, we expect the k-deep rule (for small/moderate k) to confirm blocks more optimistically than the checkpoint-based ledger. Thus, the ledger produced by the k-deep rule will be a bit ahead of the checkpoint-based ledger.

Safety and liveness are defined for *each ledger* in a natural way: safety means that once a transaction appears in the ledger, it stays there forever, while liveness means that new transactions keep getting added to the ledger. It is indeed possible that under certain conditions, different ledgers have different security guarantees. For example, under variable participation, an adaptive ledger will be both safe and live, whereas a finality-based ledger will only be safe, not live. To summarize, the notions of adaptivity and finality actually apply to each confirmation rule (i.e., ledger), rather than the entire protocol.

Under optimal conditions (i.e., static and synchronous), both ledgers grow at the same rate. Under variable participation, the finality-preserving ledger stalls, while the adaptive ledger keeps growing. As soon as static participation returns, the finality-preserving ledger catches up.

Two confirmation rules. The notion of multiple confirmation rules may seem surprising at first, but if we think about it, it is a natural concept. In fact, the longest-chain protocol alone is equipped with a multitude of confirmation rules! To understand this better, let us make a distinction between the block production rule and the confirmation rule of the protocol. The block production rule is to propose a new block at the tip of the longest chain one has seen so far; this is the rule that gives the longest-chain protocol its name. The confirmation rule is to confirm a block that is buried k-deep, along with all the ancestors of this block. All players follow the same block production rule. However, each user can choose its own confirmation rule, i.e., they each choose their value of

k independently, depending on the level of security they prefer and the latency they are willing to tolerate.

Validity conditions. A point to note is that there is some flexibility in choosing the *inputs* to the BFT protocol and in deciding what inputs are *valid* ones. Some validity conditions must be introduced, or else malicious checkpointers could propose arbitrary blocks for checkpointing, and these might get checkpointed. A validity criterion would help honest nodes ignore invalid blocks. Introducing the right validity conditions is a challenging design question. For example, we would like checkpoints to be issued for relatively new blocks. Checkpointing an old block is not of much use, for it will already be confirmed with very high probability using the k-deep rule. At the same time, one should not confirm a block very close to the tip, as the longest-chain may deviate from there. Checkpointing a d-deep block, for some moderate value of d, is a possible trade-off.

Permissioned or Permissionless? In the design so far, the checkpointers are a permissioned set of nodes. Can we make the checkpointing mechanism permissionless? In principle, they could be rotated regularly from a large set of players using a PoS mechanism, as done in Algorand. Even so, some level of a permissioned system is inevitable. A pure PoW system cannot offer the finality properties we desire (the closest we could do was what was done in Hybrid Consensus, but that does not meet our requirements). For simplicity, in this lecture, we just assume that the checkpointers are fixed for the duration of the protocol.

Examining Adaptivity and Finality. We introduced finality gadgets with the aim of combining a committee-based BFT protocol and the longest-chain protocol to get a system with both adaptivity and finality. The system design gives us two confirmation rules, and leads us to believe that the k-deep rule would provide adaptivity, and the checkpoint-based rule would provide finality. Let us examine closely whether we indeed achieve this.

In our two-layer design, adaptivity of the k-deep rule clearly holds, because blocks are mined just as they are in the regular longest-chain protocol. The layer-two checkpointing protocol has no bearing on the layer-one protocol. How does the checkpointing protocol fare under variable participation? If enough number of checkpointers are not actively participating, the protocol simply stalls. Once all honest checkpointers are back online, they resume checkpointing blocks. Thus, under variable participation, the checkpointing protocol turns off and then back on for arbitrary durations, depending on the participation level. This merely affects the liveness of the protocol; the safety remains intact at all times.

Let us now turn to finality. We want the finality gadget to protect the protocol under periods of asynchrony. In other words, even after a period of asynchrony, all blocks until the last check-pointed block should remain confirmed. We do not expect that any new blocks will be checkpointed in this period. Once synchrony resumes, the protocol should start checkpointing new blocks, and thus regain liveness. Does the aforementioned design satisfy this requirement?

Unfortunately, it does not. Consider an extended period of asynchrony. The adversary can create a new chain that forks back from before the last checkpointed block, and becomes the longest chain. Once synchrony resumes, all miners will find the adversarial chain as the longest chain and will mine on that. There will be no new blocks below the last checkpointed block! At this point, the checkpointers must either stall completely or (consciously) break safety by switching chains. Either of these is undesirable.

The key reason for this issue is that we have created a two-layer solution. This means that the behavior of miners does not depend on the checkpointers. For a finality gadget to be effective, miners must respect the checkpointed blocks. In particular, miners should mine below the latest checkpointed block. Of course, we would like to have some longest-chain-like properties as well. A natural suggestion is that miners follow the *checkpointed longest chain rule: extend the longest*

chain below the latest checkpoint block. Since the latest checkpoint block is also the checkpointed at the latest height, this rule could also be stated as: mine on the longest block that contains all the checkpoint blocks. Under synchrony, all checkpoints would be on the longest chain by design. Following the checkpointed longest chain rule would be the same as following the longest chain rule. Thus, the new protocol would inherit the security guarantees of the old protocol. On the other hand, the checkpoints protect the system during an extended period of asynchrony. If the adversary mines blocks deviating before the previous checkpoint, all honest nodes would simply dismiss such blocks as invalid. For all practical purposes, every new checkpoint acts like a new genesis block.

The design of checkpointed longest-chain protocol requires some care, because the checkpointing protocol actually affects the miners. One area of vulnerability is the behavior of the checkpointing protocol during variable participation. It is possible that the checkpointing protocol is stalled for a very long time, due to low participation. Once participation resumes, we expect the protocol to resume checkpointing quickly, and close to the tip of the longest chain. At all times, the protocol must ensure that the checkpoints are always on the longest chain. Doing so requires paying special attention to the validity conditions mentioned beforehand. We refer to this paper for more details.

The CAP theorem

Blockchains and the CAP theorem

We have seen how finality gadgets give us dual-ledger protocols, with one ledger providing adaptivity and the other, finality. A natural question to ask is, can we have a *single ledger* that gives both adaptivity and finality? Unfortunately, this is not possible. The impossibility is a consequence of a celebrated result in distributed systems, known as the CAP theorem (CAP stands for Consistency, Availability and Partition tolerance). Roughly speaking, the CAP theorem states that during a network partition, a distributed system must make a choice between availability (liveness) and consistency (safety); it cannot offer both. Moreover, this choice must be encoded in its design itself. Thus, systems can be classified as availability favoring or consistency favoring, based on their design.

Blockchains, being distributed systems, also inherit the trade-offs implicated by the CAP theorem. It states that a protocol cannot be adaptive and, at the same time, offer finality. The essence of the impossibility result is that it is difficult to distinguish network asynchrony from a reduced number of participants in the blockchain system. Hence, a protocol's behavior must be similar under both these conditions. The CAP theorem also sheds some light on why we see two totally different classes of protocols in today's blockchain space. Protocols based on the longest-chain idea favor liveness, which lends them the adaptivity property. Committee-based BFT protocols (such as Hotstuff) favor safety, which makes them finality-guaranteeing.

The CAP theorem does, in fact, allow for resolving the adaptivity-finality trade-off at a *user level*. We have seen that a carefully constructed finality gadget gives us two distinct confirmation rules: one that guarantees adaptivity, and the other, finality. Such a protocol lets clients make a *local choice* between availability and finality. Depending on the nature of the transactions, clients can make a choice of which confirmation rule they would like to apply. For example, for low-value transactions such as buying a coffee, one may prefer adaptivity over finality. In contrast, for high-value transactions such as buying a Tesla, it is natural to choose finality over adaptivity.

Historical notes

The CAP theorem was originally proposed in the late 1990s, in the context of a web service, which is an example of an internet scale distributed system. Traditionally, distributed systems were spread over a small physical area and over a small number of computers. Consistency was considered to be of foremost importance, and people were willing to trade-off some degree of availability for it during

times of faulty communication. This was not a problem, as network partitions were rare and quickly fixed.

When designing internet-scale distributed systems, researchers realized that insisting on perfect consistency was causing inefficiency in terms of availability (being quick to generate responses). The CAP theorem was the result of the realization that one had to give up some degree of consistency in order to achieve the necessary availability. This is not always a bad thing! It also showed that consistency can be recovered during good network conditions.

A couple of real-world examples helps us understand why we may want to trade-off consistency for availability. When we scroll through any social media such as Instagram or Twitter, we would like it to generate a feed instantly (i.e., be available). These platforms are able to do so by generating the feed ahead of time, based on data locally available at the server close to where we are accessing them from. We do not care that much about whether our feeds are consistent, i.e., whether the same piece of news, or the same photo shared by a friend, appears at the same time, on all of our feeds. Another example would be the behavior of the COVID vaccine registration websites. When we open the website, they might show that a few vaccines are available, allowing us to sign up. Since many people are signing up in parallel, it is quite possible that the few slots available are gone by the time we finish. However, the website does not keep checking with all of its servers to see if doses have run out every second. If it did, the web page would freeze up! Instead, websites choose to remain available, providing a smooth experience for its users. At the time of confirming, it does a consistency check, and provides a confirmation only after making sure that doses have not been overbooked. That's why this step takes time. A similar experience is seen in many other websites as well, such as in Amazon (e-retail) or Instacart (groceries).

The CAP theorem is nuanced enough to allow for great flexibility in design. For example, different "components" of the system can have different behavior: some components may favor safety, while others may favor liveness. These components could be different kinds of operations, data, or users. For example, an e-retail server may favor availability for browsing goods, but would favor consistency when it comes to actually purchase one. A newspaper website may favor availability in terms of its news content, but consistency when it comes to storing user's passwords and subscription details. Lastly, we have seen how finality gadgets in blockchains give users a choice between safety and liveness.

Proof of CAP Theorem

It is relatively straightforward to prove the CAP theorem for a toy model of a distributed system. Consider a system in which there are two servers, p_1 and p_2. Both servers store a certain variable V. Initially, the variable V has value x. Then, they are partitioned into two disconnected parts of the network. During partition, some client sends a write request to server p_1, requesting that V be set to y. Since p_1 cannot communicate to p_2, it is faced with the following two choices:

- It can act as "available" to the client, i.e., it can send an "ok" response to the client and set $V = y$. p_2 will still have V set to x. In this case, when some other client contacts either p_1 or p_2, they will get inconsistent responses, i.e., consistency is lost.
- If it fails to establish communication to p_2, it will not respond "ok" response to the client and set $V = x$. In this case, p_1 prioritizes consistency over availability, since it remains unavailable to the client.

The same trade-off is faced by p_2 as well, when a client issues a read request. Ideally, it should establish contact with p_1 to check if V has been re-written or not. However, it cannot establish a contact due to the network partition. It has the choice to either eventually return a response (and risk returning the wrong response) or to never return a response. Thus, if communication is asynchronous (i.e., processes have no a priori bound on how long it takes for a message to be

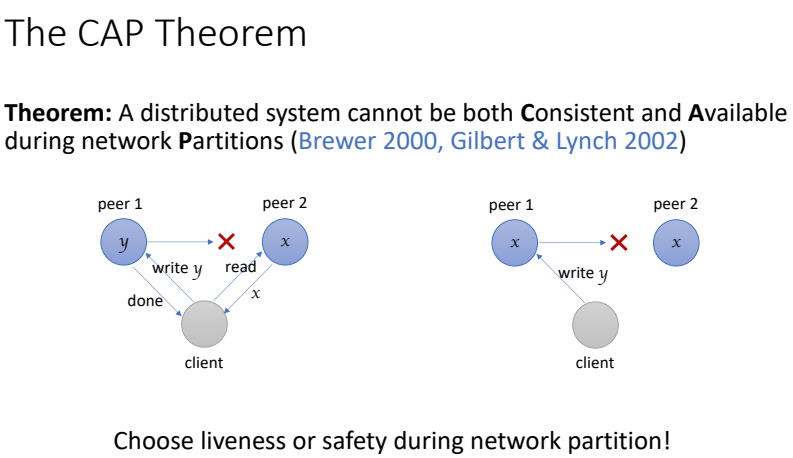

Figure 3: A visualization of the trade-offs implicated by the CAP theorem

delivered), it is impossible for the system to guarantee both consistency (safety) and availability (liveness).

References

The ideas behind Hybrid Consensus have appeared in a few works, such as ByzCoin. It was formally studied for the first time in this paper. The idea of a finality gadget was first proposed by Buterin and Griffith in the form of Casper FFG. This work did not specify a complete system, but provided some of the main ideas that are still in use. More recent works, such as Afgjort and GRANDPA, provide a full system design. Moreover, they provide a more principled approach to building finality gadgets, identifying many of the issues that we identified in this paper. However, both these works do not provide all the necessary security guarantees. Afgjort, being a layer-two design, cannot provide security under asynchrony, while GRANDPA is vulnerable to attacks in the variable participation setting, due to malicious checkpointers.

Two concurrent works, titled Ebb-and-Flow and the checkpointed longest chain, identify the adaptivity-finality trade-off as implicated by the CAP theorem, and provide a solution to resolve the trade-off at the user level. Among these solutions, Ebb-and-Flow provides a more general solution, allowing for *any* BFT protocol to be combined with *any* adaptive protocol. However, the protocol does not provide intrinsic validity: to get a complete and consistent ledger, the protocol requires a certain post-processing to discard invalid (or duplicate) transactions. The checkpointed longest chain solution overcomes this issue, at the expense of generality.

The CAP theorem for blockchains was stated and proven by Lewis-Pye and Roughgarden in this paper. Strictly speaking, it precludes the possibility of achieving finality in the Proof of Work setting. It does not explicitly classify protocols on the basis of their adaptivity-finality properties. It leaves open the question of whether both finality and adaptivity can be achieved in a more homogeneous fashion in the Proof of Stake setting.