Building Protocols for Scalable Decentralized Applications

Kai Mast

Abstract Blockchain protocols are a promising technology in the abstract, but, in reality, fall short of the promise of supporting arbitrary decentralized applications. For example, Bitcoin supports <10 transactions per second and Ethereum's gas limit prevents computationally expensive applications to execute on its chain. This chapter provides an overview of mechanisms that have been proposed to overcome these limitations. In particular, we describe novel consensus protocols, sharding mechanisms, state and payment channels, subchains, and federated protocols. Additionally, we give insight into the tradeoffs and benefits of the different approaches.

1 Introduction

Blockchains [45, 64], or more broadly decentralized ledgers, enable applications to execute across a trustless peer-to-peer infrastructure. We consider a system decentralized if individual nodes cannot influence its execution as long as they do not control a threshold of the network. This means that decentralized architectures protect against malicious adversaries in addition to simple crash failures. As a result, decentralized ledgers allow for online services to operate without reliance on a trusted party. Figure 1 outlines this stark contrast to previous architectures, where each user's data is in full control of a single organization.

While blockchain protocols are a promising technology in the abstract, they fall short in critical ways. For example, the Ethereum blockchain has roughly the processing power of a portable calculator or about 35k floating-point operations per

K. Mast (✉)
Department of Computer Sciences, University of Wisconsin-Madison, Madison, WI, USA
e-mail: kaimast@cs.wisc.edu

D. A. Tran et al. (eds.), *Handbook on Blockchain*, Springer Optimization and Its Applications 194, https://doi.org/10.1007/978-3-031-07535-3_7

second.[1] The culprit for these limitations is that decentralization requires massive replication of computation and data. This massive replication results in high computation, communication, and storage overheads, which in turn, hurts throughput and latency.

In this chapter, we first give an overview of the decentralized ledger model and the protocols that implement it. Then, we discuss different avenues for improving the performance of such protocols to support real-world workloads. Throughout the chapter, we will also give insight into the limitations and open problems of existing mechanisms.

We describe four different avenues for scaling blockchains. First, we discuss how the consensus protocol itself can be made faster. Second, we discuss sharding, which allows running multiple consensus protocols in parallel. Third, we provide an overview of layer-2 solutions, such as payment channels. Finally, we discuss federated blockchains, which can be viewed as a hybrid of layer-2 and sharding solutions. Our discussion is mainly focused on safety, i.e., that the consistency and integrity guarantees of the blockchain system are not broken, and availability, i.e., that the current and past states of a blockchain can always be retrieved and inspected.

2 Decentralized Ledger Abstraction

Each decentralized architecture, in essence, provides the abstraction of an append-only ledger with semantics that goes beyond the mere storage of data and execution of programs. These semantics are key to building applications with high integrity in a decentralized setting and it is important to understand them before modifying existing or creating new protocols for decentralized ledgers. For the rest of this chapter, we will refer to this abstraction as decentralized ledgers and the underlying protocols as decentralized ledger technologies (DLTs).

We extend the formalism of Adya [1], which defines a database $\mathcal{D}$ consisting of a history $\mathcal{H}_\mathcal{D}$ of transactions and a set of objects $\mathcal{O}_\mathcal{D}$, each associated with a totally ordered set of *object versions*. Each transaction is a set of operations applied to a particular object, such as a read, write, or append. Each object's version history initially only consists of the $\perp$ value, indicating that it has not been created yet.

Transactions affecting the same object(s) and their operations can be ordered with respect to each other. We say a transaction T precedes another transaction T' if it appears earlier in the database's history, denoted as $T \leftarrow T'$. This relationship is transitive, i.e., if $T_1 \leftarrow T_2$ and $T_2 \leftarrow T_3$ hold, then $T_1 \leftarrow T_3$ holds as well. Similarly, we say an operation op precedes another operation op' if the object versions it accesses precedes that of op'. Transactions affecting two disjoint sets of objects may

[1] With the current gas limit, Ethereum can do about two million floating-point multiplications per block, which are published about once a minute [21].

not be able to be ordered with respect to each other, denoted as $T \leftrightarrow T'$. Similarly, operations affecting two distinct objects cannot be ordered with respect to each other, denoted as $op \leftrightarrow op'$.

2.1 Consistency

Like many conventional database management systems, decentralized ledgers allow enforcing application-specific constraints on the data and provide strict serializability for all operations. Serializability ensures that all transactions execute atomically, i.e., in a serial or equivalent to serial order. In other words, if two transactions T_1 and T_2 are applied to two distinct objects, they must be applied in the same order to both objects. Strict serializability extends this notion of a real-time order: if T_1 started before T_2, its operations should also be applied before those of T_2 (see Eq. 1). This, in turn, not only ensures the integrity of a system's state but also makes it much easier for developers to build applications, because they do not have to reason about concurrency.

$$\forall T_1, T_2 \in \mathcal{H}, \forall op_1 \in T_1, op_2 \in T_2.\ T_1 \leftarrow T_2 \Rightarrow op_1 \leftarrow op_2 \vee op_1 \leftrightarrow op_2. \quad (1)$$

2.2 Immutability

Decentralized ledgers are eidetic: they maintain a record of all transactions ever processed by the system. From this record, any past state of the system can be regenerated and inspected. As a result, the ledger can serve as a notary or an impartial witness, by providing a reliable record of past information.

Formally, successfully applying a transaction T to a database $\mathcal{D}$ yields a new database $\mathcal{D}'$ with T appended to its history. Similarly, the version history of each object modified by T will be appended with its new version.

We then define immutability as a constraint on the allowed state transitions from $\mathcal{D}$ to $\mathcal{D}'$. Thus, if a database state $\mathcal{D}$ predates another state $\mathcal{D}'$, i.e., $\mathcal{D} \leftarrow \mathcal{D}'$, all of its transactions and object versions are contained in $\mathcal{D}'$. This means the successor state can only add new transactions and object versions and not remove or reorder them, denoted in Eq. (2).

$$\forall \mathcal{D}, \mathcal{D}'.\ \mathcal{D} \leftarrow \mathcal{D}' \Rightarrow \mathcal{H}_{\mathcal{D}'} \subseteq \mathcal{H}_{\mathcal{D}} \quad (2)$$

Immutable systems thus are, unlike the term *immutability* suggests, able to change their state, but will only allow state transitions that extend the state without removing existing information. Further, they might enforce other data policies to ensure the integrity of a particular application. For example, a cryptocurrency usually wants to ensure that no transaction is spent more than once.

2.3 Auditability

Auditability enables participants to join the network at any point in time and verify all states relevant to them up to the current point without having to trust a particular remote party. Formally, we say there exists a publicly available function $verify(\mathcal{D}, \mathcal{D}')$ that certifies a transition from $\mathcal{D}$ to $\mathcal{D}'$ is valid. Auditors can then recreate and verify the entire system execution by verifying all database state transitions, starting with the initial state consisting of an empty transaction history.

3 Decentralized Ledger Technologies

At the core of DLTs lies consensus protocols, used for *state machine replication (SMR)* to maintain a unified database. The definition of a state machine comprises a set of potential states the machine can be in and a set of admissible state transitions that allow moving from one state to another. SMR decides which state transitions to perform and replicates this decision across all participants of the protocol. As a result, all non-faulty participants maintain the same state at any point in time.

Most consensus protocols, while varying greatly in their implementation, are leader-based. This class of protocols first appoints a particular node to be a leader (sometimes called a primary), which then proposes state transitions to the system. These state transitions are then subject to approval by the rest of the network. The existence of a singular leader ensures that transactions are proposed in an order that ensures serializability. Finally, leader-based protocols can react to failures or bad performance at any point in time by appointing a different entity to be a leader.

3.1 Assumptions and Attack Model

Distributed ledgers are designed to be resilient against Byzantine failures, a model that encompasses both benign failures and those caused by malicious intent. A Byzantine actor may want to change the network's behavior to their advantage or break the network entirely. To achieve this, attackers may issue invalid or conflicting messages, and delay or hide communication. Correct nodes, on the other hand, follow the protocol as prescribed.

To ensure that correct nodes faithfully follow the protocol, distributed ledger protocols typically assume that the majority of network participants behave rationally and provide incentives for these rational actors to advance the protocol. These incentives can take the form of direct payments, where parties that process transactions receive compensation in the form of block rewards or transaction fees. Further, incentives can be based on collateral, where parties that misbehave are penalized financially.

DLTs may rely on different network assumptions. The three most common are synchronous, partially synchronous, and asynchronous [18]. In the synchronous setting, messages will be delivered within a known and fixed time-bound. In the partially synchronous setting, messages will be delivered in an unknown, but finite, time-bound. Finally, in the asynchronous model messages may take an unbounded amount of time to be delivered.

3.2 Data and Transaction Models

DLTs require a different data model than conventional databases because they execute in a trustless environment. Here, each user is associated with a set of cryptographic keys and must sign off transactions spending their cryptocurrencies with those keys. Nodes participating in a decentralized protocol need to prove that a transaction has been signed off by a particular set of users for it to be deemed valid. Additionally, transactions issued by a client might be conflicting. For example, Alice might request to spend $5 each on Bob and Claire, but only have $6 in her account.

Data models in decentralized ledgers are focused around the notion of payments and cryptocurrencies, as this was their initial application and cryptocurrencies are still the basis for incentive mechanisms in almost every DLT. We discuss the two most common ones: UTXO and Accounts.

The UTXO Model

Bitcoin represents a user's account balance as a set of *Unspent Transaction Outputs (UTXOs)*. Transactions in the UTXO model work similarly to a voucher system in which some input vouchers are exchanged for new vouchers of the same or lesser value. Figure 1 outlines how transactions consume UTXOs (the unspent outputs of a previous transaction) and produce new UTXOs. Note, that in a real system some transaction's input would go towards a transaction fee.

Bitcoin, like many other DLTs, relies on Merkle hash trees [40] to provide authentication of the blockchains state. Merkle trees can be generated for any arbitrary set of objects. To do this, these objects are first arranged in some pre-defined order. The tree is then constructed by recursively combining k hashes, where k is some branching factor of the tree, and generating a new hash value from the resulting value. A Merkle proof then allows verifying an object's state against the root of the tree without having access to the entire tree. The proof is the particular branch from the object to the tree root. An example for a Merkle proof and its associated tree is given in Fig. 2. The verifier here just recomputes and checks the correctness of every hash in the branch to ensure the proof's integrity. These proofs are virtually impossible to forge as it is very hard to find collisions, i.e., to input values that map to the same output value, for cryptographic hash functions [55].

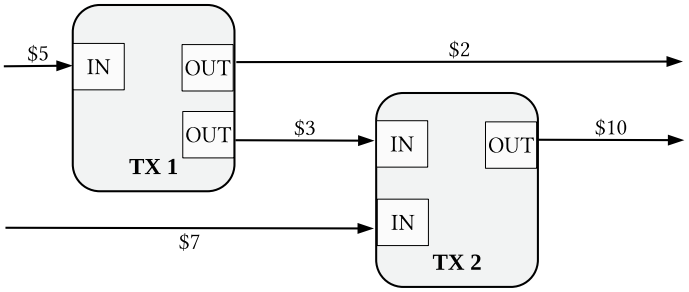


Fig. 1 Sketch of the UTXO model. Each transaction consumes one or multiple unspent transaction outputs and generates at least one new transaction output. Outputs are owned by a particular public key

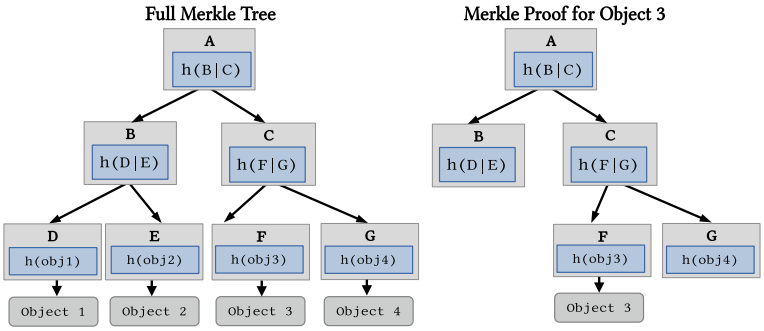


Fig. 2 Sketch of a Merkle tree and an associated proof. To prove the authenticity of Object 3 against the root A, we only need to provide a branch leading from A to the object

The key advantage of the UTXO model is that one can succinctly prove the existence of an unspent transaction output. Each block in Bitcoin contains the Merkle root of the current UTXO set, which allows verifying the current state without each block having to contain the entire set. Protocol participants just locally compute the current state by executing all previous blocks, generate the hash tree, and then verify the root against the public chain. Additionally, third parties that do not maintain the entire state of the blockchain, so-called “light clients”, can verify the existence of a particular UTXO by verifying a Merkle proof.

The UTXO model significantly reduces the complexity of the data model that transactions execute on, but limits storing custom data. Participants in this protocol merely have to maintain the UTXO set to track the state of the blockchain and processing a transaction only involves adding and removing UTXOs to the set. As a result, platforms that are focused mostly on monetary transactions, such as Bitcoin or ZCash, often still rely on the UTXO model due to its simplicity.

The Accounts Model

Ethereum, in contrast to Bitcoin, relies on a data model focused on the notion of accounts. Intuitively, an account has a non-negative balance and can be owned by a particular user. Additionally, accounts can hold other data as well, which enable more complex applications.

Decentralized ledgers implementing the accounts model must provide additional measures for preventing double-spending and other conflicting transactions. First, instead of merely verifying the existence of a particular UTXO, the transaction must be verified against the account's state and ensure applying it to the account will not violate consistency constraints. Second, as long as there are sufficient funds in the spending account, a malicious node might use the same request to issue multiple transactions. To address this, Ethereum transaction requests contain a unique number, or nonce, and the protocol only admits one transaction for each combination of account identifier and nonce.

One drawback of the accounts model is that the authentication of state is more complex. Here, DLT nodes usually maintain three Merkle trees per block instead of just one as in the UTXO model. The first hash tree provides information about the resulting state of the system, the second hash tree represents the set of all transactions contained in this block, and the third hash tree represents the set of all changes.

In Ethereum and most other Account-based systems, state is represented in the form of Patricia Merkle trees [43]. Patricia trees have two key advantages over conventional Merkle trees: there exists a maximum bound on their height and updates are relatively inexpensive. This is achieved by storing data in a compact trie structure. Unlike when updating a conventional Merkle tree, where a new entry might reorder the set and rebuild the entire tree, inserting new values in Patricia trees only needs to update the affected subtree. An upper bound of the tree height is ensured by using hash values as object keys, which are guaranteed to be a certain length.

3.3 Smart Contracts

The client-server model, where applications perform computation locally and then write the resulting state to the database, does not apply to the decentralized setting. In the client-server model, Byzantine actors could attempt storing invalid application results in the globally replicated ledger and, thus, violate consistency. To prevent such attacks, DLTs provide the means to execute arbitrary applications directly on the ledger itself, similar to stored procedures in conventional database systems.

Ethereum introduced the notion of *Smart Contracts*, stateful programs that are stored and executed entirely on the decentralized ledger. While the previous system already provided some notion of programmability, such as Bitcoin Script, Ethereum smart contracts were the first to provide full Turing completeness and, as a result, the possibility to support arbitrary programs. Smart contracts are usually written in a

high-level language, such as Solidity, and then compiled to byte code, such as EVM byte code or WebAssembly, before being stored and executed on the ledger.

Smart contracts reside on a particular address on the blockchain, analogous to how each user's account is assigned an address. Contracts may hold currency and contain a key-value store to store arbitrary data. Users call functions of a smart contract by issuing a transaction that contains a function call and some amount of currency to pay for the computation.

Ethereum replaced fixed transaction fees with a notion of "gas cost", which covers the cost of processing and executing the transaction. Each transaction request contains a gas limit, representing the maximum number of computational steps the issuer is willing to pay for. Like with transaction fees, the cost of a single unit of gas is determined by the market. If the transaction runs out of gas during execution, it aborts. Any unused gas is refunded to the party that issued the transaction.

Contracts can modify their local state directly while executing or invoking functions of other contracts. The latter allows reusing existing code and interaction between different applications. For example, one can implement a custom token on top of Ethereum that can be used as a form of payment by other contracts.

3.4 Committee-Based Consensus

Classical consensus protocols achieve state machine replication among a fixed set, or committee, of nodes. They were first introduced by Leslie Lamport [35], among others [47, 56]. These protocols now form the foundation for most fault-tolerant applications. For example, a web service might be implemented across three data centers. If one of the data centers fails, the consensus protocol ensures that operation can continue by shifting computation to the other two data centers.

While consensus protocols were initially intended to tolerate only benign failures, the introduction of Byzantine fault-tolerant consensus protocols allowed for more complex use cases. For example, a node in the committee might not simply become unavailable but encounter a software bug that makes it behave in ways not originally intended by the software developers. While such failures might be much more unlikely than a crash, it is still important to be resilient against them for safety-critical applications.

Recently, committee-based Byzantine fault-tolerant consensus protocols, such as *Practical Byzantine Fault Tolerance (PBFT)* [8], have received new attention in the context of decentralized ledgers. Because these protocols cannot only protect against software bugs or hardware failures but also against a malicious human adversary controlling a subset of the committee, they are suitable for implementing applications where mutually distrusting parties are trying to agree on a consistent state.

While committee-based, or *permissioned*, protocols allow for greater tolerance against Byzantine failures, they are not sufficient to provide full decentralization. In particular, such protocols often only work well with a small number of participants. As a result, small committees of nodes can quickly devolve into an oligopoly. Here,

while not a single entity controls the system, a small number of participants can collude to take over control of the system. Similarly, committee members can collude to artificially increase transaction fees or impose censorship.

3.5 Sybil Detection

In the *permissionless*, or public, setting, such as that of Bitcoin or Ethereum, protocols must be resilient to Sybil attacks, where a single entity is creating multiple identities to gain more control over the system. These attacks are feasible because without a trusted third party there is no straightforward way to authenticate user identities.

Consensus protocols rely on either computational barriers or stakes to prevent such Sybil attacks. Stake-based systems manage membership information as part of their protocol. In committee-based consensus protocols stake usually is binary, which means only members of the committee are allowed to vote and each committee member has the same voting power. Here, each change to the committee must be approved by all participants. Recent protocols have introduced the notion of variable stake, often bound to how much cryptocurrency a certain party holds. In this setting, cryptocurrency can be passed on to other participants to dynamically reallocate voting power.

Systems that rely on computational barriers for Sybil detection do not manage any form of global membership information. Instead, participants have to perform a certain task to become, or have the chance to become, a leader. Most commonly, this task involves solving a cryptographic puzzle, where an input to a hash function has to be found such that the functions' output is below a specified threshold.

These particular cryptographic puzzles are better known as *Proof of Work (PoW)* [17]. The underlying intuition is simple: every attempt to solve the puzzle requires a constant amount of computation and the chance to solve the puzzle is independent of earlier attempts. PoW, thus, provides a very reliable means of Sybil detection, albeit being a very wasteful mechanism.

3.6 Nakamoto Consensus

The Bitcoin paper introduced the Nakamoto consensus, a consensus protocol that builds on top of *Proof of Work (PoW)*. There are two core differences between protocols described so far and the Nakamoto consensus. First, the use of PoW allows it to be a public protocol that allows participants to join and leave at any point in time. To take part in the protocol one does not have to register with some global mechanism, but merely starts attempting to solve the crypto-puzzle. Second, the Nakamoto consensus operates non-deterministically, where the current state is known to be agreed upon by the global network with some high, but not absolute, probability.

Systems based on Nakamoto consensus rely on Gossip protocols [15] to broadcast messages, such as transactions or blocks, because they execute across a peer-to-peer network with no pre-defined topology or membership. Instead of being connected to the full network, participants of a Gossip protocol only talk to a few peers. When receiving a new message, they forward it to all their peers. To make gossip efficient, participants usually keep track of which messages they already sent to or received from a particular peer. As a result, messages eventually spread to the entire network, without the network being fully connected.

Nakamoto consensus performs leader election using PoW through a process called mining. Once a party has solved the cryptographic puzzle, they forward their solution in form of a block to the network to become a leader. Instead of proposing transactions after becoming leader, they directly include a set of serialized transactions in the published block. Once participants start mining, their chance of becoming miner is directly proportional to the processing power available to them, because each attempt to solve the crypto-puzzle is independent of past attempts.

Nakamoto consensus achieves consensus by picking the longest chain of proposed chains. There can always be multiple competing blocks or chains because mining is a random process. Honest participants pick the longest chain of valid blocks they received and, as result, will all eventually converge on the same prefix of the blockchain. However, this means that one has to wait a significant amount of time for the block to be "buried" deep enough in the chain for it to be considered finalized and immutable. For example, in Bitcoin, one usually waits for depth for 10 blocks (about 60 min).

Figure 3 outlines how, at a particular point in time, there might be multiple competing chains. Here, while the prefix of the chain is considered stable and abandoned forks have been removed, at the head of the chain, multiple forks are competing for the longest chain. At any point in time, the protocol might switch to a different branch, potentially reverting multiple blocks. These switches are sometimes also called *reorganizations*.

Protocols based on Nakamoto consensus with open membership, usually assume a strong bound on the network latency. This ensures that a block will be visible to all network participants after some fixed time. More concretely, systems like Bitcoin assume that this bound is about 5 min. If this assumption was not made, there could potentially be an undetectable longer chain due to a network partition.

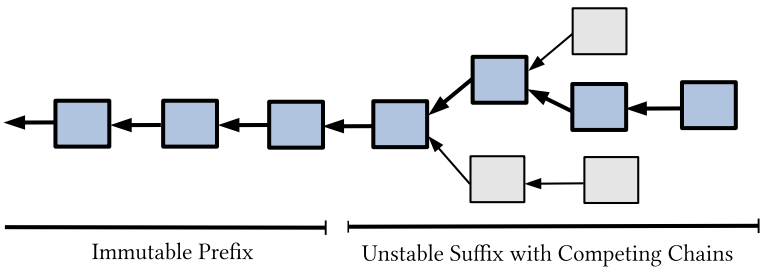


Fig. 3 Sketched structure of a Bitcoin-like blockchain. The currently winning chain is highlighted

3.7 Bottlenecks

We now discuss the major bottlenecks of blockchains: *execution*, *verification*, and *communication*. Essentially, electing protocol leaders and ordering transactions in a globally replicated manner require massive replication of both data and computation. Thus, what the network can process as a whole is limited by the fact that every participant needs to process, forward, and execute all transactions.

Execution

Transactions in decentralized ledger systems differ significantly from those in conventional database systems. Every participant of the protocol maintains its local state in an authenticated data structure to be able to verify and process future blocks. In particular, DLT nodes usually calculate and store some form of hash tree of the state, and every block contains the root hash of the current state. These hash trees can be used both to verify blocks and to provide succinct proofs of some substate of the system. Executing transactions in such an authenticated manner requires more computation and storage. This is one of the reasons why systems such as Ethereum employ a limit on how many computational steps a block can contain ("gas limit"). Previous work has demonstrated that an improved storage engine can mitigate this bottleneck to some amount [50].

Verification

Blockchains rely on digital signatures to ensure the correctness and authenticity of messages. Intuitively, checking every transaction request and block generates a high computational workload as digital signatures are rather complex to verify. Increasing the frequency of transactions included by the system, thus, significantly increases the burden for every node in the network to participate in the protocol.

Privacy-preserving decentralized ledger may also rely on more advanced cryptography, such as zero-knowledge proofs, which increases the verification burden significantly.

Communication

Finally, for every node to be able to process every block and transaction, all transactions and blocks must be propagated to the entire network. Intuitively, this creates a high network communication overhead. Decentralized ledgers usually execute across a geo-distributed peer-to-peer network. Here, a larger state that needs to be synchronized will further increase the considerably high propagation latencies.

Even worse, scalability mechanisms may harm decentralization, a key promise of decentralized ledgers. For example, a naive attempt for increasing the throughput of a ledger is a higher block frequency or block size. Either, will cause a higher propagation delay of messages and, in turn, increase the likelihood of forks. Additionally, bigger block sizes raise the CPU and storage requirements for nodes participating in the network. This problem is especially salient for new nodes joining the network the need to verify *all* blocks in the chain before processing new transactions. As a result, only participants with strong hardware that is well connected may participate in the protocol, causing a more centralized network layout.

4 Improved and Novel Consensus Mechanisms

4.1 Improved Committee-Based Consensus Protocols

Figure 4 sketches the message exchange in the Paxos protocol [36], excluding its leader election, one of the most prominent mechanisms for SMR. Once a leader is elected (not shown in the figure), clients can submit transaction requests to it. The leader then proposes the transaction to its followers (`accept?`), which then each forwards their response (`accept!`) to the network. If a majority of the nodes accept the transaction, it is considered accepted by the system as a whole and the result is forwarded to the client. So-called Multi-Paxos pipelines this mechanism, by only electing a leader every so often and having that leader propose many transactions in sequence.

Practical Byzantine Fault Tolerance (PBFT) [8] was one of the first Byzantine fault-tolerant consensus protocols and is still widely used today. Figure 5 outlines how the protocol accepts a transaction. PBFT adds another round of messages to the

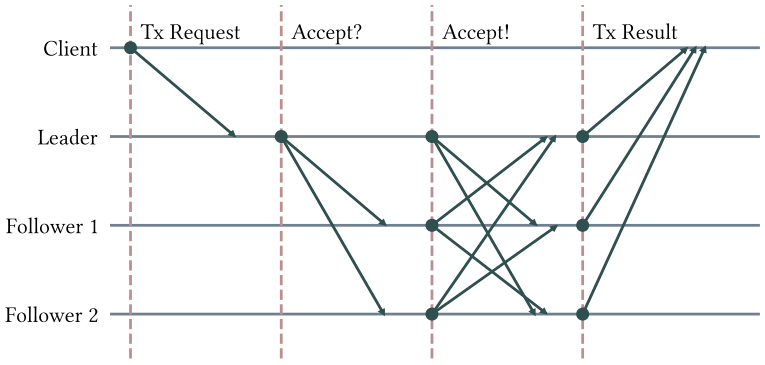


Fig. 4 The voting phase of the Paxos protocol: The leader proposes new transactions to the system, which then need to be approved by the majority of the network

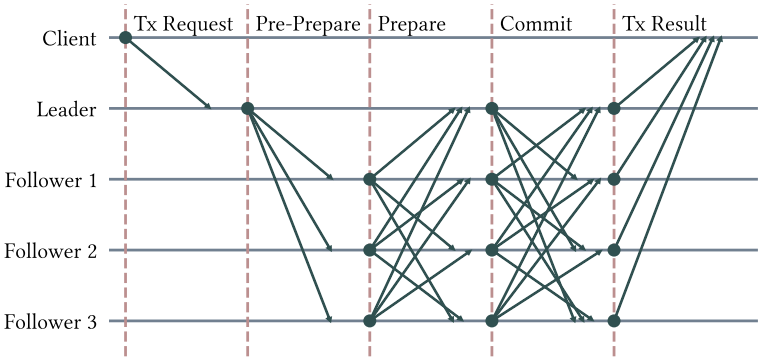


Fig. 5 Protocol diagram of PBFT: compared to Paxos an additional phase is added to account for Byzantine behavior. Note that also additional replicas are needed to tolerate Byzantine failures

protocol that confirms the receipt of the `prepare` message—which is equivalent to the `accept!` message in Paxos—by a majority of the network. This is necessary because nodes might send conflicting messages to different participants in the network. For example, follower 1 might send a `prepare` message for one transaction to the leader while also sending a prepare for a conflicting transaction to follower 2.

Similar to Multi-Paxos, PBFT can let a single leader propose many transactions to speed up the protocol. Leaders are usually only switches during failure. In the context of PBFT, this mechanism is usually called a *view change*.

Several minor improvements to PBFT have been proposed over the last few years. For example, Zyzzyva [32] avoids the third round of messages in the absence of failures using speculative execution. Aardvark [9] adds additional robustness by making clients digitally sign their requests and frequently rotating leadership. Tendermint [6] reduces communication complexity using Gossip protocols. We describe more complex modifications and entirely new permissioned protocols below.

Another key factor in making PBFT (and similar protocols) scale is *batching*. Similar to blocks in permissionless systems, a large set of transactions is bundled together. This allows to reduce the amount of communication required per transaction, but, in turn, increases the latency of the protocol.

Stellar

The Stellar Consensus Protocol (SCP) [37] is a variation Byzantine Agreement where each participant may have different levels of trust in other participants. Here, each participant picks a weight for peers they trust. The weight indicates their level of trust. Classical BFT consensus can be seen as a special case where each participant picks the same weight for all peers.

For SCP to reach consensus, a quorum must contain a *quorum slice* for each of its non-faulty members. Each node can define one or more quorum slices, of which at least one must be met for a valid quorum. A quorum slice consists of a set of nodes S and a certain threshold, for example, $\frac{3}{4}$, of how many members of S's members have to agree. Finally, for a quorum to be valid, each of its quorum slices must overlap with one another.

Similar to PBFT, SCP executes in three phases: `NOMINATE`, `PREPARE`, and `COMMIT`. The `NOMINATE` phase acts as a filter by identifying valid candidates for a consensus value. Each node can nominate multiple values, but must not nominate new values once it has confirmed the `NOMINATE` statements of a peer.

The `NOMINATE` phase is then followed by one or multiple rounds of ballots. Multiple rounds of ballots may be needed as it is not possible to determine whether a ballot got "stuck" due to a failure or if there is just a large network delay. Here, for the n-th ballot, the `PREPARE(n,x)`-message ensure that no value other than x is chosen for any ballot $\leq n$. A `COMMIT(n,x)`-message then states that the value x was chose at ballot n.

HotStuff

HotStuff [66] is a novel consensus protocol that improves upon PBFT by reducing message complexity by employing a star topology. HotStuff is of particular interest, as it is intended to be used by Facebook's Diem (formerly "Libra") cryptocurrency.

Figure 6 outlines the message exchange in HotStuff. HotStuff allows for a start communication pattern with linear complexity, not quadratic like PBFT, by passing all messages through the leader. To account for malicious leaders, another round of messages is needed compared to PBFT. In the so-called `Decide`-phase, the leader notifies all participants that a particular commit message has been received and accepted by a quorum.

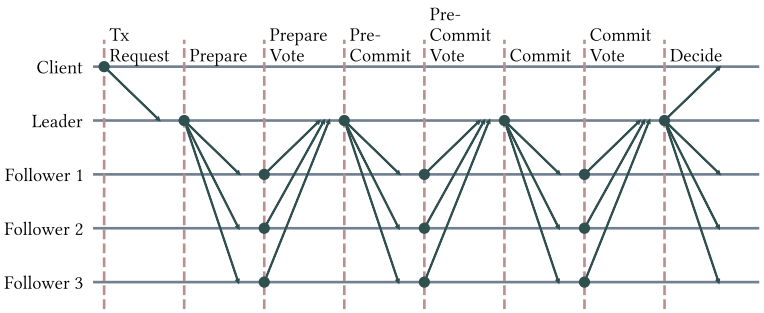


Fig. 6 Protocol diagram of HotStuff: the leader includes Quorum Certificates (signatures from all members of the quorum) from the previous phase to reduce message complexity

HotStuff reduces communication complexity through a primitive called *Quorum Certificates (QCs)*. Instead of having every participant exchange messages with each other, the committee communicates in a start topology with the leader at its center. The leader then includes QCs in its messages, which are certificates that prove the receipt and acceptance of a proposal by a quorum. This is achieved by having OCs contain a *threshold signature*, which, in essence, is a signature signed by multiple private keys that can be verified against a single public key. A key advantage of threshold signatures is that they do not grow in size with the size of the quorum.

To achieve even higher performance, the protocol can be pipelined, where multiple phases of the protocol (for different batches of transactions) execute in parallel. This is possible by including leader election in every prepare phase so that view changes do not interrupt pipelining.

Byzantine Ordered Consensus

For the committee-based setting, Zhang et al. [69] propose a mechanism that establishes transaction order outside of leader election. Their protocol, pompē, prevents the so-called "front-running", which allows malicious leaders to reorder transactions to their advantage. However, their results also indicate that pre-ordering transactions can enhance the performance of a permissioned system significantly. Here, instead of letting the leader decide the order of transactions directly, transactions are ordered before they are serialized.

To order transactions, a node n proposes a command c and asks a quorum of the participants to assign a timestamp for that command. Nodes first order all pending commands (or transactions) locally and then reply with a timestamp that honers this ordering. Node n then picks the median timestamp from the replies. Because there are at most f malicious nodes and a quorum consists of $2f + 1$ nodes, this median value is guaranteed to be within the valid range of time stamps.

HoneyBadger BFT

HoneyBadger [42] is a leaderless and asynchronous consensus protocol. At a high level, this protocol operates in three steps.

First, *each* node proposes a set of transactions, which are then broadcast to all other nodes. To do this, they maintain a local pool of transaction requests from client and sample some subset of this pool. The exact mechanism of how this subset is chosen is important for performance and safety but goes beyond the scope of this book. A *Reliable Broadcast Protocol (RBC)* ensures that each non-faulty node's proposal is propagated through the network.

Second, nodes send and acknowledgement for each valid set of transactions being proposed. They do this using threshold encryption, the same primitive as is used in

the Stellar protocol. Encryption serves two purposes: it allows participants to sign on to a set without increasing the size of the message significantly and it prevents adversary to censor transactions they do not like.

Third, nodes agree on which sets to accept. This last step is important, because faulty nodes might not propose a transaction set and, as a result, could cause the protocol to wait forever or become inconsistent. To address this, nodes vote to accept a set of transactions once it has been signed by enough participants until at least $N - f$, where N is the number of nodes and f is the maximum allowed number of failures. Afterward, they will vote to reject all other transaction sets.

To retrieve the final batch of transaction to be accepted, nodes form a "common subset" of all accepted transaction sets. Nodes add all transactions for each accepted transaction set to the batch. Then, they sort the transactions lexicographically, e.g., by their transaction identifier, and remove duplicates.

This protocol is teared towards the UTXO model where the probability of two transactions conflicting is low. Thus, one potential drawback of this design can be in a setting where there might be multiple conflicting transactions.

4.2 Minor Changes to Nakamoto Consensus

Concurrency Inside Blocks

Bitcoin and Ethereum enforce a serial order of operations in a single block. This limits the execution of a block to a single logical core. TTOR (Topological Transaction Ordering Rule), which was used for some time in Bitcoin Cash, loosens this requirement and only enforces a partial order among transactions, and, thus, enables validating transactions of a single block concurrently.

However, in our experiments, we observed that the bulk of work performed by blockchain nodes is involved in validating and generating digital signatures, which can already be performed concurrently as transaction requests are usually received and generated out of band. Bitcoin Cash eventually switched from TTOR to another ordering mechanism as its benefits to block propagation were limited [53].

Block Size and Frequency

As mentioned in Sect. 3.7, one attempt to scale blockchains is to increase block size allowing for more transactions to be processed with the same number of blocks. The propagation delay depends on its size and performance of the underlying peer-to-peer network. In particular, a larger block takes longer to propagate between two peers as outlined in the equation below.

$$Latency_{block} = Latency_{network} + \frac{BlockSize}{Throughput_{network}}.$$

Block sizes that are too large might take longer to propagate than it takes to mine the next block [14, 24]. As a result, larger block sizes result in a higher likelihood of blockchain forks, which hurt performance. Once a fork is resolved, only one of the branches is considered part of the chain, and the rest is discarded.

Another intuitive attempt of increasing the throughput of a blockchain system is to increase the frequency of blocks. Similar to block sizes, a higher block frequency results in an increased chance of forks as blocks are created faster than they are being propagated through the network. This in turn also leads to more centralization of mining, as large-scale mining pools have a higher chance of receiving and processing blocks in time.

Ethereum relies on a mechanism called Greedy Heaviest Observed Subtree (GHOST) [58] to disincentivize centralization. Here, if a miner is aware of a fork will reference not only a block's direct predecessor but also the heads of competing chains (known as "uncle blocks"). As a result, miners receive a partial reward if they ended up mining on a fork.

Increasing Efficiency of Block Propagation

The nature of peer-to-peer protocols results in significant communication overheads when propagating data. Gossip communication inherently requires additional communication, because data does not flow in a straight path but spreads in multiple directions through a peer-to-peer network. As a result, peers might receive the same messages from multiple parties. This problem is exacerbated in Bitcoin as transactions are propagated through the network twice: as a transaction request and as part of a block.

One line of work is to improve the efficiency of Gossip protocols. Compact blocks [10] do not contain a full list of transactions as their payload but merely shortened transaction identifiers. Upon receiving a compact block, peers only request the transaction they have not seen yet.

Bloom filters can be used to efficiently keep track of which data a peer has already received [49]. Essentially, bloom filters are a lightweight data structure (usually only a few bytes) that provides a heuristic about whether a set contains some data item or not. When forwarding compact blocks, peers rely on Bloom filters to estimate which transactions the remote party already holds and forward only the ones that it probably does not have yet. This, in turn, avoids an additional round-trip time, where the remote party has to request transactions.

Another line of work is to augment peer-to-peer networks with a fast relay network. Relay networks are usually not a good fit for the decentralized ledger setting, as they have sparse topologies, and, thus, contain multiple points of failure. However, they can be used in addition to a fault-tolerant peer-to-peer network, to allow for faster propagation of blocks in the common case [11, 29].

4.3 Decoupling Mining from Transaction Serialization

Consensus protocols generally perform two distinct tasks. `LEADER` picks the next participant to be the leader of the protocol, i.e., the entity that propose the next block(s), and `ORDER` decides on the order of transaction inside those block(s).

In most permissionless protocols, these tasks are bundled together, which harms performance. In particular, blocks in Bitcoin or Ethereum can only hold a certain amount of transactions and are published at a low frequency. Additionally, as outlined in Sect. 4.2, increasing block size or frequency does not always result in higher throughput of the blockchain.

Bitcoin-NG

Bitcoin-NG [26] breaks down the process of mining in traditional Nakamoto consensus into its constituent processes to increase throughput. The Bitcoin-NG `LEADER` process proceeds as follows: Miners solve a PoW puzzle and broadcast a special block called a *keyblock* with the solution to the rest of the network, signaling their status as the protocol leader. At that point, the winning miner performs an `ORDER` process by grouping transactions into *microblocks* and broadcasting them into the network. This separation of key and microblocks is outlined in Fig. 7. The entity that mined the most recent keyblock creates and broadcasts microblocks so long as they are the leader. Solely the network speed and how quickly the leading miner can sequence them limit the flow of transactions.

While Bitcoin-NG improves throughput over the conventional Bitcoin protocol, it is still limited to the bandwidth of a single entity executing the `ORDER` process. Also, a single high-throughput chain harms decentralization as every participant of the protocol needs to possess the processing power and network bandwidth to process the chain in its entirety. Further, these long-lived leaders can be subject to Denial-of-Service (DoS) attacks.

ByzCoin

ByzCoin [30] follows the same observation as Bitcoin-NG, but, instead, establishes a committee of leaders. ByzCoin relies on some underlying *identity blockchain*. The committee is then chosen by picking the entities that mined the last n blocks on the

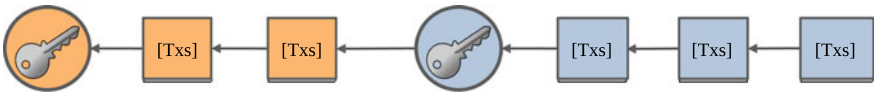


Fig. 7 Structure of the Bitcoin-NG blockchain: Keyblocks (square) hold leadership information, while microblocks (circle) hold serialized transactions

identity chain, where n is the size of the committee. The entity that mined the most recent block, is the designated leader. Each set of transactions proposed by the leader needs to be approved by a majority of the committee.

A key advantage of ByzCoin over Bitcoin-NG is that it has almost instant finality. Bitcoin-NG, on the other hand, has a similar latency as the unmodified Bitcoin protocol. The instant finality of ByzCoin is possible because, for a sufficiently large n, the majority of the committee is guaranteed to consist of honest miners.

4.4 Novel Proof-of-Stake Protocols

Proof of Stake (PoS) is a mechanism intended to be an energy-efficient replacement for PoW. At a high level, voting power here is not dependent on a party's processing capabilities but on the amount of funds they hold in the cryptocurrency, which, in turn, allows avoiding unnecessary computation. The key challenge in PoS is the "nothing at stake" problem: if block generation does not require mining, an adversary can easily generate many, potentially conflicting, blocks.

Ouroboros

Ouroboros [28] is a provably correct PoS protocol that powers the Cardano blockchain.[2] The protocol's execution is divided into constant-size epochs, each consisting of some number of time slots. At the beginning of an epoch, a seed is generated from the values of the previous epoch, to generate a pseudo-random assignment of participants to slots, where the likelihood of being assigned to a slot is directly proportional to the amount of currency a participant is holding.

In every slot, the selected participant is allowed to propose a block containing a set of transactions. Although an adversary here can still generate conflicting blocks, assuming the majority of the participants are honest, a sequence of correct blocks will eventually constitute the longest chain. However, in this scheme, it is still possible to anticipate who the next leader will be and launch a DoS attack on them. Like Bitcoin, Ouroboros requires blocks to propagate within a bounded amount of time and loosely synchronized clocks.

Algorand

Algorand [23] addresses some challenges in PoS using a verifiable random function (VRF). Here, each participant locally runs the VRF which takes some global data and their private key as an input. Depending on the input, the function may return

[2] Note that this section describes the initial version of Ouroboros outlined in the CRYPTO 2017 paper.

a certificate that the particular user is allowed to propose a block. Like in Bitcoin, multiple users may be allowed to propose blocks and Algorand provides a mechanism to sort certificates of concurrent blocks. Protocol participants then discard all blocks except the one with the highest priority to prevent forks.

Algorand prevents DoS attacks by making this random function unpredictable and switching participants after every round of voting. This unpredictability is achieved by taking the user's private key as an input. Later users can prove they executed the VRF correctly using their public key. Additionally, the protocol assumes the absence of network partitions to prevent malicious users from successfully proposing conflicting blocks.

Avalanche

Avalanche [54] is a probabilistic leaderless consensus mechanism with low communication overhead. Here, nodes periodically query a constant-size random set of peers about which transaction they accepted. Depending on their peers' responses, they adjust their confidence in the transaction being accepted by the network as a whole. Acceptance of a particular transaction will then eventually propagate through the network. Avalanche works well with the UTXO model as transactions do not need to be totally ordered and conflicting transactions are rare. The protocol needs to be combined with another mechanism, such as PoS, to ensure Sybil resistance.

```
def on_query(v, new_col):
    if col == None:
        col = new_col

    respond(v, col)

def slush_loop(u, col0 in [R, B, None]):
    # Initialize with red blue or nothing
    col = col0

    for _ in range(m):
        # if None, skip until on_query sets the color
        if col == None: continue

        K = sample_nodes(k)
        P = [query(v, col) for v in K]

        for ncol in [R, B]:
            if P.count(ncol) >= alpha*k:
                col = ncol

    accept(col)
```

Listing 1 Slush: a simplified version of the avalanche protocol

Listing 1 displays a simplified version of the Avalanche protocol without fault tolerance, dubbed "slush". Here, the network aims to decide on a single color `col`. To do this, each node queries, a random sample of `k` other nodes `m` times. In the outlined code, `on_query` is called whenever a node is queried by some other node and `slush_loop` is executed repeatedly by each node. After each set of queries, nodes participating in the slush protocol either decide to stick with the color it has currently accepted, or, if more than $\alpha * k$ (where $\alpha > 0.5$) other nodes have accepted a different color, switch over to that color.

A key advantage of this protocol is that it requires almost no state to be maintained at each node (only the currently accepted state) and that it involves communication with a small subset of, instead of a majority of, the network. More concretely, communication complexity per node is constant independent of the size of the network, because the sample size `k` does not grow with the size of the network.

Gasper and Ethereum 2.0

Casper is a "finality gadget": it allows ensuring that a block in Nakamoto consensus is finalized and cannot be undone due to a reorganization. Here, stakers endorse blocks they consider part of the longest chain using their stake. Because the total amount of stake is known, at some point, if a block is sufficiently endorsed, it can safely be considered final.

Ethereum 2.0 relies on *Gasper*, a protocol that combines mechanisms from Casper with GHOST. Gasper performs leader election similar to Ouroboros: time is segmented into slots, each having a leader that is defined by some randomized mechanism. For every k slots, the protocol uses the Casper mechanism to achieve finality. Like in Ouroboros and Bitcoin, this requires loosely synchronized blocks. Unlike those mechanisms, the protocol will not be unsafe in a partially synchronous setting, but may not make progress. Competing blocks can still exist, due to network delays or malicious leaders, but GHOST's notion of referencing "uncle blocks" allows quick convergence to a singular chain in this case.

Gasper relies employs a chain selection rule based on the amount of stake attached to a particular chain. Here, the "heaviest chain" is the chain with the most stake attached to it. This ensures that the protocol will converge on the chain that is considered finalized by the majority of the network, not adversarial chain that is potentially longer.

4.5 Summary

The Bitcoin protocol and derivatives are not sufficient to support any demanding workload and waste massive amounts of energy. We do need new protocols, or radically improve existing ones to overcome these limitations.

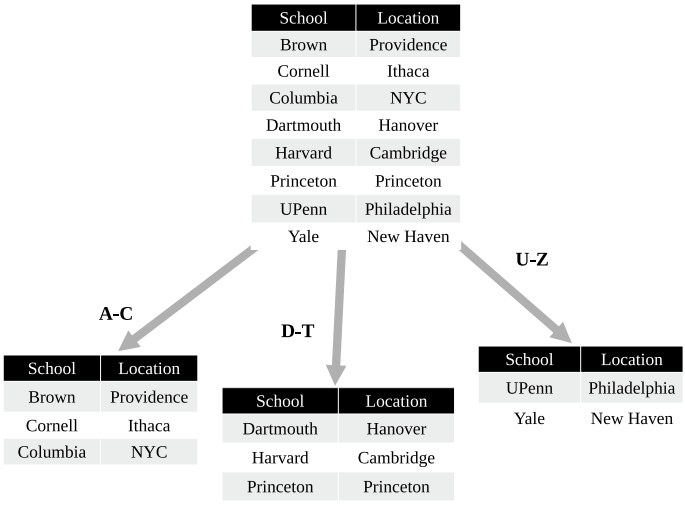


Fig. 8 Sketch of how a table of all Ivy League schools could be sharded alphabetically

Multiple potential contenders exist to replace outdated ledger technologies, each with different properties and tradeoffs. Further evaluation and benchmarking is necessary to determine the "winner" among these protocols.

5 Sharding Blockchains

At a high level, sharding breaks the keyspace of a database into multiple "shards".[3] Sharding is usually done using a hash function or breaking the keyspace into evenly sized pieces. Figure 8 sketches how a table can be broken apart by assigning different ranges of starting letters to different shards. Hash functions are usually the preferred mechanism of sharding as they assign objects to shards pseudo-randomly and thus spread the workload of a system more evenly across shards. Updates and queries for a particular shard can then be processed without involving other shards.

Sharding for blockchains is usually implemented in the following way [62]. Some mechanism keeps track of a set of identities, e.g., by examining the last k miners of a PoW chain [30]. The protocol then assigns shard some subset of these nodes. Each shard then locally runs a consensus mechanism, such as PBFT [8], and a distributed transaction protocol, such as a two-phase commit, handles cross-shard transactions. Finally, some scheme is in place to periodically "merge" the state of all shards.

[3] Note that some protocols shard transactions, not state. These mechanisms are significantly different and not covered by this section.

5.1 Challenges in Sharding Blockchains

Blockchain enthusiasts have long hoped that sharding will solve the scalability problem. In essence, sharding allows every participant to only process a subset of all transactions of the network. Ideally, this allows to linearly scale the throughput of the system *without* increasing the burden on any particular participant. However, so far, no sharding protocol has been deployed in a real-world setting. The reason for this is complex but, at a high level, sharding decentralized ledgers faces four major challenges: *reduced safety*, *reduced availability*, *loss of network decentralization*, *reduced consistency*, and *lack of economic incentives*.

Maintaining Safety

The essence of decentralized ledgers is that they protect some application, e.g., a cryptocurrency, against a strong Byzantine adversary. A basic requirement for protection against such an adversary is to have a Sybil detection mechanism, which is usually based on how much stake an entity has or how much computational work is done.

For example, a core assumption in Bitcoin is, that $<50\%$ (or 25% in some cases) of the entire network is controlled by adversaries. A shard intuitively has much less total stake (or computational work) than the system as a whole. Thus, some mechanisms must be in place to ensure that a single shard is safe as the network as a whole.

Ensuring Availability

When splitting upstate among subsets of the network, less participants hold a copy of a particular transaction. For example, if a transaction executes locally on a particular shard, there is no need for other shards to know about that transaction, let alone storing it on their machines. As a result, shard state could get lost during failure. This problem is exacerbated by the fact that an adversary might be incentivized to hide (parts of) a shard's state. For example, they might have misbehaved and want to hide the evidence, or they might want to revert the shard to a state that is more advantageous to them.

Ensuring Consistency

In systems such as Ethereum, a serial order of transactions is enforced to ensure consistent updates to contract states. Unfortunately, enforcing a total order for all transactions is very difficult if shards execute mostly independently.

To ensure consistency, a sharding protocol needs some mechanism to consistently apply transactions to multiple shards. Protocols for distributed transactions, which

ensure consistent and atomic updates across shards, have been explored extensively in the systems community. However, adopting such protocols to a permissionless setting with Byzantine failures is a challenge.

Maintaining Decentralization

So far, we have discussed decentralization as an abstract system property. More concretely, ensuring decentralization means keeping the burden of joining the network and participating in the consensus protocol low. Ideally, anybody with a computing device should be able to join the system.

Even current non-sharded systems that promise decentralization are not very decentralized in practice. For example, Bitcoin and Ethereum are controlled by only a handful of entities [22]. The underlying reason for this is that decentralization often conflicts with the goal of scalability. If a network supports many participants of varying locations and processing power, data takes longer to be propagated across the network. As a result, control of many decentralized ledger protocols tends to centralize around nodes with access to large amounts of processing power and fast network connections.

Providing Sound Incentive Mechanisms

Miners (or stakers) participate in a consensus protocol because they receive some monetary reward or want to secure the value of their assets stored on the ledger. Bitcoin and Ethereum have fairly straightforward incentive mechanisms, where the miner of a new block gets some new currency and transaction fees as a reward.

Incentive mechanisms tend to get more complicated when introducing sharding, as there may exist distinct shard chains and transactions can execute across multiple shards. For example, OmniLedger [31], while providing a safe sharding protocol, does not provide sound incentives for the large set of validators required to power the protocol.

5.2 Foundations

Several sharding solutions have been proposed for permissionless and permissioned blockchain systems that are built on previous work on sharding databases and distributed transactions. In fact, the concepts of sharding and distributed transactions have been widely studied concerning conventional database systems. While the failure assumptions are vastly different in conventional systems, the underlying motivation of reducing coordination and increasing parallelism to achieve higher throughput is the same.

Sharding Databases

Sharding was first popularized by systems like Chord [59] and Mercury [5]. In such systems, usually, a hash function is applied to an object key to map it to a specific shard. Some systems also have the notion of virtual shards. Here, a large number of virtual shards is mapped to a considerably smaller number of nodes. Virtual shards can then be remapped to different nodes if the workload changes.

Later work introduced systems, such as Chubby [7], which provide serializable transactions on top of sharded systems. More recent work aims to improve performance by reducing coordination even further [13, 44] or relying on loosely synchronized clocks [12].

Distributed Transaction Protocols

The most prominent mechanism to apply transactions in a consistent and atomic fashion to multiple shards is a two-phase commit [4]. Here, in the first phase, a transaction first locks all relevant data objects, ensuring that no concurrent updates are made. In the second phase, the transaction applies all changes and releases the locks.

Two-phase commit can be separated into two variants. First, a conventional (or pessimistic) two-phase commit acquires locks gradually while executing a transaction. If a lock is already held by another transaction, some mechanism such as wound-wait must be in place to avoid deadlocks. Optimistic concurrency control [33], on the other hand, first executes transactions without holding locks, then submits the transactions as a set of operations to the involved servers in the first phase of the protocol. The main advantage of OCC is that it keeps the time a lock is held short allowing for higher concurrency. Pessimistic concurrency control usually works better in update heavy workloads and in settings where latencies are high.

5.3 Public Blockchain Sharding Protocols

To our knowledge, virtually all blockchain sharding protocols apply to public (or permissionless) blockchains. While private blockchains can leverage sharding as well to increase performance, the problem of low performance is less severe there as they operate committee-based protocols with a small number of participants. For example, HotStuff can process thousands of transactions per second, while Ethereum can only process tens.

Monoxide

Monoxide [63] breaks up the workload across independent consensus zones, each having its own set of miners. Monoxide does not support generalized transactions, but only money transfers between exactly two accounts. For a cross-zone transaction, the transactions are first processed in the source account's zone and then forwarded to the target account's zone together with a Merkle proof of the transaction's inclusion. At some point, the transaction will be included in the source and the target zone, however, the protocol does not provide an upper time-bound for this.

Furthermore, the transaction processing scheme proposed in monoxide is susceptible to recursive invalidation of dependent transactions in the case of zone-forks. Another challenge with Monoxide's design is that its independent zones naturally partition the mining power of the blockchain system, which dilutes the overall security of the system. The authors address this by assuming the majority of miners will work in all zones at the same time, which requires miners to possess large amounts of processing power for verification to maintain the same security guarantees as Bitcoin. This encourages mining centralization for high throughput, giving up the key property of blockchains.

Elastico, RapidChain, and OmniLedger

Elastico [38] and OmniLedger [31] in a similar class of scalability solutions that propose dividing the nodes in a system into small committees, each of which performs a Byzantine consensus protocol for intra-shard consensus. In these protocols, there exists a single identity blockchain, similar to that in ByzCoin, as well as, a distinct blockchain for each shard. The Elastico protocol, the first of such solutions, proceeds in the following fashion: protection against Sibyls is achieved using an identity chain based on PoW. It then pseudo-randomly assigns nodes to committees that perform PBFT in rounds until all the nodes in the system agree on a final changeset to be committed. The protocol then re-assigns committees and restarts the process for the next set of transactions.

OmniLedger makes further improvements on top of Elastico, such as using RandHound [60] to better seed for randomness in shard assignments and helps ameliorate some security compromises introduced by Elastico's small committee sizes. However, OmniLedger still adds several layers of complexity to public blockchains. This complexity is especially salient when examining the need for OmniLedger to have day-long epochs because of the amount of overhead required for bootstrapping at the beginning of an epoch, which makes it susceptible to quick-responding attackers.

Additionally, OmniLedger allows for atomic cross-shard transactions using Atomix, a variant of a two-phase commit. Here, clients have to first lock funds of the affected shards in phase one. They collect proofs of inclusion of the lock message in the shard, or, respectively, proofs that the transaction could not be included in the shard. In phase two, if all shards lock the transaction successfully, they issue a

unlock message that commits the transaction. Otherwise, they issue an unlock message that will abort the transaction. Elastico, on the other hand, has no notion of atomic cross-shard transaction.

Note that, in the Atomix protocol, if a client fails, the transaction is "stuck". The authors argue that the client has an incentive to finalize their transaction as, in the UTXO model, their funds are locked while the transaction is in progress. This is similar to how Avalanche incentivizes clients to not issue conflicting transactions, as it would lock up their own funds. However, this makes it difficult to implement a similar mechanism for smart contracts, where the incentive structure is not as clear.

RapidChain [67], among other changes, replaces the Atomix protocol with one that does not rely on the behavior of particular clients. Instead, the transaction is assigned to a particular shard by hashing its identifier. The output of the transaction, i.e., the generated UTXOs, are then also stored on that particular shard. The shard then contacts all shards that hold inputs for the particular transaction. To make this scheme efficient, shards are not connected to every other shard, but instead, route messages through a shard network.

Zilliqa

Zilliqa [70] shards transactions, but not state. This protocol relies on a similar mechanism as OmniLedger for assigning nodes to shards but uses a different cross-shard commit protocol. Instead of splitting the state of the system across shards, they only split the transaction workload and replicate state among all nodes.

Each shard then processes a subset of all transactions for a specific epoch and merges their resulting state with other shards at certain checkpoints. At a high level, the protocol allows a particular shard to lock parts of the state to prevent concurrent modification of the same data entries. Zilliqa employs a data flow-based programming model to implement this scheme efficiently.

Ethereum 2.0

Ethereum 2.0 [61] introduces a sharding scheme among other major changes to the protocol. This mechanism borrows ideas from both off-chain mechanisms and randomness-based protocols like OmniLedger. Here, nodes participate in the protocol by putting down a deposit. A verified random function then assigns each node to a particular shard.

Additionally to the random assignment, the protocol ensures safety and availability by punishing nodes that sign an invalid block or respond too slowly. At the time of writing this chapter, the Ethereum developers have not yet decided on a protocol for cross-shard transactions and it is unclear whether the protocol will support serializable cross-shard transactions.

5.4 Summary

Sharding is almost certainly necessary to make decentralized ledgers scale. However, it is a problem that is still in the process of being solved without losing any of the core guarantees that blockchains provide.

6 Layer-2 Solutions

Instead of scaling the blockchain protocol itself, the so-called "Layer 2" protocols can be layered on top of existing systems to improve performance. These protocols are usually orthogonal to previously mentioned approaches, such as sharding, as they build on top of an existing DLT.

Payment channels lock funds on the global ledger and facilitate fast transactions between parties through an off-chain protocol. Only the amount locked on the base chain is allowed to be exchanged in these systems, and a tally of balances is kept for when it is time to settle. On settling, the amount apportioned to the settler as denoted by her balance in the subchain is unlocked on the main chain and returned to the settler. State channels extend this scheme from cryptocurrency funds to the arbitrary state.

6.1 Building Blocks

Layer-2 solutions rely on a common set of cryptographic primitives to implement their functionality securely. We outline the most important ones here.

Merkle Proofs

Merkle trees allow creating a succinct tree of cryptographic hashes that represent a system's state. Such trees are constructed by hashing all objects of the state and then recursively merging hashes by applying the hash function to them again. Usually, only the root of such trees are stored on the blockchain and the rest of the tree can either be constructed on the fly by clients or is provided in the form of Merkle proofs.

Merkle proofs then allow showing the validity of a system's (sub-)state by providing the branch of the tree from the affect objects to the root. This proof can then be verified against the root hash on the blockchain. Because these proofs rely on cryptographic hashes, it is virtually impossible to forge a Merkle proof against the same root for a different state.

Cryptographic Commitments and Fraud Proofs

Analogous to a promise in real life, participants can provide a *commitment* in the form of a statement that is signed with their private key. For example, one can generate a hash $H = h(S)$ of the current state S of a system and sign it with a cryptographic key. This enables any holder of the commitment to prove later that the state of the system was indeed S.

If a party detects misbehavior, they can then raise a *fraud proof* that shows two conflicting cryptographic commitments by a particular party. Layer-2 protocol often rely on fraud proofs to punish misbehaving party, as well as, to recover from failure.

Time Locks

Time locks allow for a certain transaction or statement to become invalid if not included on the blockchain in a specified timebound. In Bitcoin, this mechanism is implemented using the `CheckSequenceVerify` and `CheckLockTimeVerify` protocol extensions. Here, time can be expressed either as real-world time or as the length of the blockchain. Again, in systems like Ethereum, similar functionality can be implemented using a smart contract.

This mechanism enables layer-2 protocols to recover in case of participants becoming unresponsive. For example, in a payment or state channel funds could be lost if a party refuses to cooperate. Some protocols, thus, release funds after a certain amount of time if no progress is made.

Hash-Locked Transactions

In Bitcoin, *Hash-Locked Transactions* allow locking funds, which can then be retrieved using a custom key. More concretely, such hash locks define under which conditions a particular transaction output can be spent. These locks are implemented using Bitcoin Script and they can be implemented similarly in other programmable blockchains, such as Ethereum.

In the context of layer-2 protocols, this primitive is especially useful, as it allows to lock funds for a channel or subchain and later release it only if a certain condition is met. For example, it allows extending time-locks with a fraud-proof mechanism from Sect. 6.1, which we will outline later.

6.2 Payment Channels

At a high level, payment channels lock funds on some existing systems and facilitate fast transactions between parties through an off-chain protocol. Only the amount locked on the base chain is allowed to be exchanged in these systems, and a tally of

balances is kept for when it is time to settle. This flow is outline in Fig. 9. At the time of writing, the most prominent payment channel protocols are Lightning [52] for Bitcoin and Plasma [51] for Ethereum, respectively. Payment channels do not rely on additional consensus mechanisms but, instead, on cryptographic commitments and time locks.

We now outline the Lightning protocol at a high level.

Creating and Updating Channels

First, a `funding`-transaction is created, which records the initial funds deposited into the payment channel. This initial funding transaction creates a single transaction output that can only be unlocked using a transaction signed by both parties.

Then, whenever the balance of the payment channel is updated, a new `commitment`-transaction is created that records the new state. `commitments` are not immediately stored on the blockchain, but saved by the participating parties for later use. Each new commitment contains an `revocation` for the previous commitment.

The `funding`-transaction is signed and stored on the main chain once the first `commitment` has been created. The latter ensures that channels can always be terminated, as termination requires a `commitment`-transaction to exist.

Terminating Channels (One-Sided)

In Lightning, either party can close the channel at any time by storing the most recent `commitment`-transaction on the main chain. While the other party has immediate access to the released funds, the closing party needs to wait for a certain amount of time before their funds can be used.

This waiting time is implemented using hashed time-locks. There are two potential outcomes of this hashed time-lock. First, if the closing party has attempted to close the channel with an outdated `commitment`, the other party can reveal the `revocation` for the outdated `commitment`, which serves as a fraud proof. If

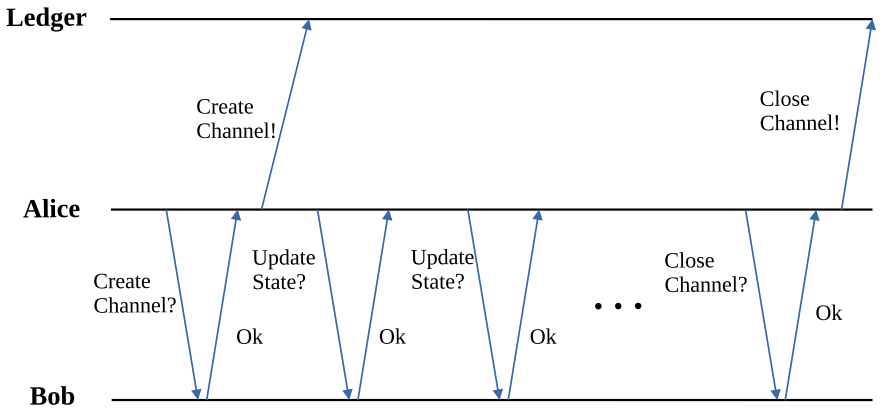


Fig. 9 Flow of a payment channel: only the opening and closure of the channel are recorded on the global chain

they do so, they will claim all funds stored in the channel and, thus, punish the misbehaving party.

Terminating Channels (Cooperatively)
If both parties are available and well behaving, they can cooperatively close the channel. Because the original funding transaction is a single UTXO that can be unlocked by a transaction signed by both parties, this is much easier to do.

A cooperative channel termination is then just a regular transaction that consumes the channel's UTXO and distributes the funds among the participants. No contest time or hash lock is required.

Increasing or Decreasing a Channel's Funds
In Lightning, a channel must be closed and recreated in order to change the number of funds it has access too. Some other solutions, such as that provided by Miller at al. [41] allow restocking funds.

Payment Networks

In most implementations, payment channels allow an arbitrary number of payments, with only two transactions stored on the blockchain. However, one-to-one channels, like the ones described in the previous section, require a new channel to be established whenever one wants to transact with a new participant. This makes their use somewhat limited as establishing a new payment channel is costly. Payment networks address this limitation.

At a high level, payment networks allow transacting with another party through many intermediates. For example, if Alice wants to send money to Bob, but they do not have a channel established between each other, Alice can rely on a third party that has a channel established with her and Bob. Payment networks then provide a protocol to send money through that third party, or a series of third parties in the general case. This protocol has to allow these third parties to be *untrusted* to maintain decentralized properties of a blockchain system.

In Lightning, this mechanism is ensured as follows. Consider the topology from Fig. 10. Here, Alice wants to pay Bob but does not have a direct channel established with him. To do this, Alice first notifies Bob about her intention to pay him and Bob responds with a value H, where H is the cryptographic hash of some other value R. Bob keeps R secret. Alice then promises, through a cryptographic commitment, that she will pay Carol once she reveals R to her. Carol similarly tells Dave she will pay him once he reveals R to her. Dave tells the same to Bob, except Bob knows the

Fig. 10 Example topology for a payment network: Alice wants to pay Bob through Carol and Dave

value of R. Bob can then reveal R to Dave to initiate the payment process. The other participants do the same to pass the money through the chain.

This mechanism is safe as R will not be revealed before a chain of commitments has been established. If Bob reveals R ahead of time, the money would be sent through the network partially and not reach him. As a result, Bob is financially incentivized to wait until Dave has created a cryptographic commitment to him.

Payment networks need to provide a routing mechanism, that allows discovering the current topology and establishing a path between two parties. This problem is exacerbated by the fact that not all paths are valid for a particular payment, as a particular path may contain nodes that do not hold sufficient funds to process the payment. One promising approach to this problem is that of Sivaraman et al. [57], which, among other mechanisms, improves the flow of payments by breaking them into smaller "packets".

A challenge with payment networks is to prevent them from becoming too centralized. For example, the network could devolve into a topology consisting of a few large nodes that route most payments. Such large nodes would introduce single points of failure that could harm the reliability of the network.

6.3 State Channels

Payment channels can generalize to *state channels* [41], that support arbitrary smart contracts. For example, one can implement a chess game using a state channel, where all moves are processed by the channel and only the final result is stored on the blockchain itself. In most implementations, similarly as before, participants sign off every state change using cryptographic commitment.

At the time of writing, Dziembowski et al. [19] provide the only sound mechanism for state channel networks, which relies on the notion of virtual channels. While, in regular state channels, one relies on the blockchain to resolve conflicts, in virtual channels a third entity serves in this role. The key challenge here is that, unlike the blockchain, this third party is untrusted. As a result, virtual channels can still fall back to the underlying blockchain if the third party is faulty.

Figure 11 outlines how virtual channels can be constructed on top of other virtual channels, as well, which allows building a more complex state channel networks. Regular state channels, such as that between Alice and Carol or Dave and Bob, are constructed using a *State Channel Contract (SCC)* on the blockchain itself. Here, for example, if Alice becomes unresponsive, Carol will rely on the SCC to "forcefully" close the channel. Virtual channels, such as y_1, are then constructed using what the authors call a *Virtual State Channel Contract (VSCC)*. Similar to an SCC, Carol only becomes involved, during the creation and closing of y_1. In the case that Alice becomes unresponsive, Dave can "forcefully" close the channel using Carol. If Carol is unresponsive as well, he can leverage the SCC between him and Carol to recursively close both channels.

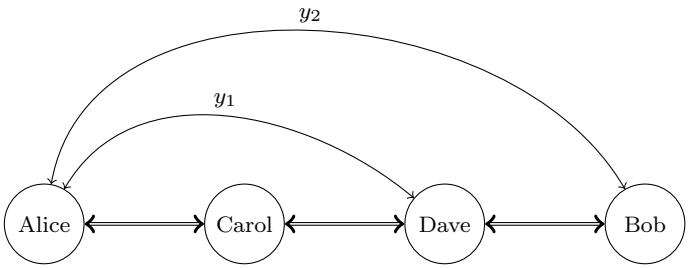


Fig. 11 Sketch of how virtual channels are constructed recursively. Alice first constructs a virtual channel y_1 to Dave, via Carol. Then she uses y_1 to construct a second virtual channel y_2 to Bob via Dave

Note, that this concept of virtual channels is tailored towards applications that have exactly two participants. To our knowledge, no sound state channel (network) construction exists yet that allows for a larger number of users to interact.

6.4 Watchtowers

A major drawback of layer-2 solutions is that they rely on a constant audit of the blockchain to prevent malicious behavior. In many implementations, if a party misbehaves, other participants must raise a fraud proof within a certain time-bound. However, not all participants might be online and actively participating in the blockchain consensus at all times.

Watchtowers [2, 3, 16, 39] enable outsourcing this task to a third party. At a high level, these mechanisms provide a reward to third parties, the watchtowers, if they detect misbehavior. More advanced solutions link this reward to the payment channel itself, to prevent malicious parties to bribe the watchtower(s).

6.5 Subchains

Systems such as BlockchainDB [20], Plasma [51] or Arbitrum [27] maintain authenticated data structures outside the blockchain and solely rely on it in case of failures. Such an authenticated data structure can be a Merkle tree, where the root is stored on the parent blockchain, as outlined in Fig. 12. This design aims to combine the advantages of a centralized system with that of a decentralized one. In the common case, the state of the system can be updated at a small number of sites without involving the blockchain. If the system fails, users issue fraud proofs to the main chain.

A core challenge with many subchain solutions is ensuring availability of blocks. As usually, only the Merkle root of the subchains state is stored on the blockchain

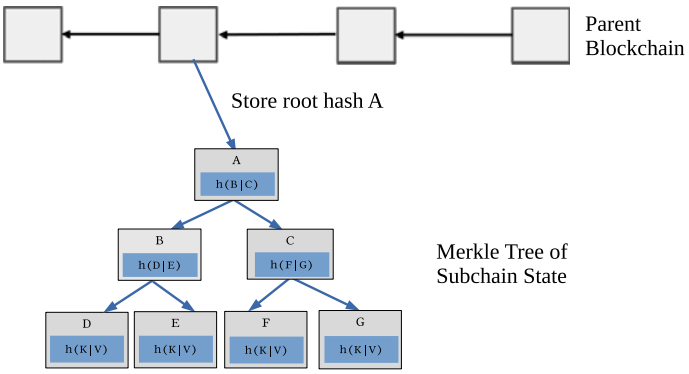


Fig. 12 Simplified concept of a subchain. The side chain's state is recorded by a Merkle tree and the tree's root stored on the parent blockchain

itself, the provider(s) of the subchain can hide the state to prevent audits. To address this, Arbitrum assumes that at least one participant in a database replica set behaves honestly and always remains available. Similarly, BlockchainDB assumes clients trust the particular database instance they are connected to.

6.6 Optimistic Rollups

Optimistic rollups are a special kind of subchain, where transactions are recorded and ordered on the main chain, but executed on a subchain. The key advantage here is that no tradeoff in terms of availability is made: if needed the subchains state can be recovered by re-executing all transactions. Examples of such systems are Optimistic Ethereum [48] and Arbitrum One [46].[4]

Rollups are mainly useful for computationally expensive transactions, such as complex smart contracts. While there exist mechanisms to batch transactions together, they usually still require all transaction data to be on-chain. This means that for simple payment transactions, the performance gain is negligible.

6.7 Summary

Layer-2 solutions are a great mechanism to *augment* other scaling solutions. For example, payment and state channels allow for instant confirmation through the exchange of cryptographic commitments. Subchains enable bundling many transactions into one on-chain transaction for efficiency. Additionally, subchains can enable

[4] Arbitrum One is not to be confused with the version of Arbitrum described in the previous section.

even high performance by making availability tradeoffs. The latter might be acceptable for applications of low financial value. Finally, optimistic rollups can overcome the computational limitations of current blockchains.

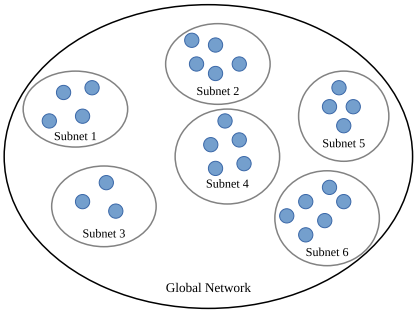
7 Federated Chains

Federated chains attempt to scale blockchains by allowing multiple separate chains to work together through a global relay chain or cross-chain swaps. While sharding usually has a fixed mechanism that dictates how and when shards are created and who they are assigned to, blockchain federation allows creating new chains organically.

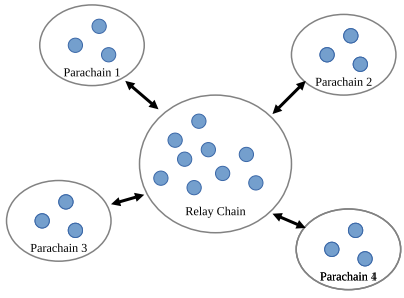
While very similar to sharding, at first sight, this mechanism is much more similar to sidechains. In most designs, there exist a global chain that takes a similar role as the blockchain in side-chain protocol, in that it processes cross-chain communication and handles failures. However, one common point with sharding is that there usually exists a mechanism to update the state on multiple chains atomically, similar to cross-shard transactions (Fig. 13).

7.1 Cross-Chain Swaps

Cross-chain swaps [25, 68] allow exchanging cryptocurrencies between two separate blockchains without the involvement of a third party. Here, funds are locked on both chains for a certain amount of time. If a chain is provided with proof that the funds on the other chain are locked as well, it considers the transaction as successful,



(a) Avalanche Subnetworks: All nodes participate in the global network, while some may also participate in other networks.



(b) Polkadot: A set of validators maintain a global relay chain. Each parachain has its own "collators" that bundle and forward parachain state to the validators.

Fig. 13 Comparision of Polkadot and Avalanche subnetworks

otherwise, it aborts and releases the funds on timeout. This, again, assumes a strong bound on network latency.

A key advantage of cross-chain swaps is that it allows to federate existing blockchains with minimal modifications. As a result, the primitive can be leveraged to build decentralized cryptocurrency exchanges. For example, one can trade Ethereum for Avalanche tokens using cross-chain swaps.

7.2 Polkadot

Polkadot [65] is a self-described "scalable heterogeneous multi-chain". It provides a single *relay chain* that handles cross-chain transactions and multiple *parachains* that rely on the relay chain for security. Polkadot additionally introduces the notion of *bridges*, special subchains that connect to other blockchain systems, such as Ethereum.

The Polkadot architecture relies on four different roles for nodes: nominators, validators, collators, and fishermen. We outline these roles in Fig. 14. Nominators are entities holding tokens on the relay chain, who appoint validators to process the relay chain. The validators then form the committee for the underlying consensus algorithm of the relay chain. Collators serve a similar role as validators, but for a specific parachain. Finally, fishermen check validators for correctness.

Each parachain is then assigned a random subset of all validators. These validators do not have to process the entire state of that parachain. In fact, they might not be able to as they are frequently reassigned to a different parachain. Instead, they rely on parachain collators to propose block candidates to them.

Collators do not need to run a consensus protocol for a parachain. Instead, collators can compete for the validators' "trust", e.g., through a history of good behavior or by providing the blocks containing the most transaction fee revenue. In addition to the

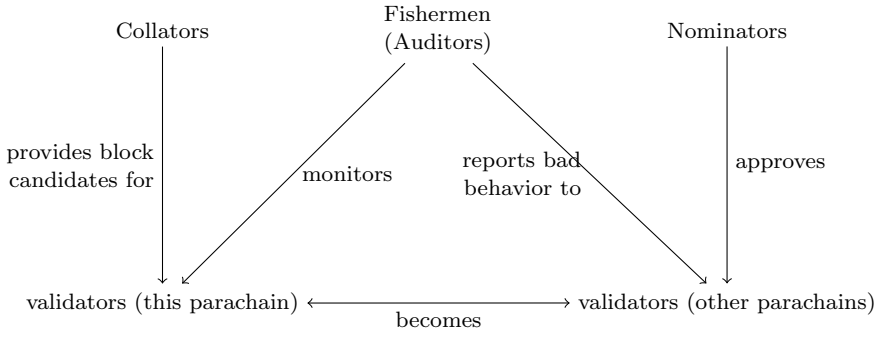


Fig. 14 Roles in the Polkadot network. Collators propose blocks for a specific parachain, which is then approved by a subset of the relay chain's validator. This approval is then inspected by the fishermen who report misbehavior to the validator set as a whole

block itself, collators provide a zero-knowledge proof that the contents of the block are correct and do not violate the parachain state. Additionally, they can provide funds that can be withheld if the block turned out to be faulty. The validators then include the headers of the accepted parachain block in the relay chain.

Because only a subset of validators processes a particular relay chain, they need to be checked by fishermen for correctness. Fishermen are somewhat similar to watchtowers in layer-2 protocols, in that, they look for misbehavior in a particular parachain or bridge, and generate a fraud proof if needed. As a result, parachains operate mostly independently from the relay chain, except when recovering from failure.

Validators additionally participate in the relay chain consensus. Here, they process and approve relay chain block, which contains parachain block headers and cross-chain transaction information, as a whole. Periodically, these validators are (re-)appointed by the nominators.

To ensure consistency of cross-chain communication, the relay chain processes all messages between parachain. Each parachain block contains an egress set of messages sent by the particular chain, and an ingress set of those messages received and process by the chain. This ensures that cross-chain transactions execute on all involved shards, and are correctly ordered. However, depending on how many cross-chain transactions there are, this can constitute a bottleneck of the system.

7.3 Avalanche Subnetworks and Cosmos Zones

Avalanche also provides the notion of federated chains through its subnetworks concept. Similar to Polkadot, there exists a global chain handling cross-chain and global transactions. Additionally, any set of entities can create a new subchain that operates independently from the global chain.

Cosmos [34] is a federated blockchain system leveraging the Tendermint consensus protocol. Here, a global "hub" processes cross-chain transactions, while there can be many "zones" that operate independently from each other. Each zone and the hub run their own instance of the Tendermint protocol and can have a different set of validators. Similar to Avalanche and Polkadot, there then exists a mechanism for cross-shard messages and coin swaps.

Both, Avalanche and Cosmos, to our knowledge, have no mechanism to recover from subnetwork (or zone) failure. This means that these systems' subnetworks (or zones) have weaker availability and safety guarantees than its global chain. On the other hand, this design can potentially allow for higher performance, as the global chain is less involved in the particular shard execution.

7.4 Summary

The key advantage of the federation is that it allows connecting blockchain systems that rely on different consensus protocols, currencies, and even different virtual machines for smart contract execution.

On the other hand, federation usually trades for safety by splitting stake or mining power into multiple independent systems. Thus, it might be safe to federate a handful of large blockchains, but not hundreds or thousands.

8 Conclusion

This chapter gave an overview of different scaling approaches to distributed ledger protocols, from the network level to off-chain solutions. Each of these mechanisms has unique advantages, disadvantages, and challenges. Note that, aside from scalability, availability, and safety, decentralized ledger technologies face many challenges that were not discussed in this chapter at all, such as ensuring user privacy or providing mechanisms for governance.

As none of the described mechanisms is a solution to all problems, we believe only a combination of multiple mechanisms can address the scalability limitations of current decentralized applications. The ledger's underlying network needs to be fast to allow for low latency transmission of blocks and transactions. The consensus mechanism must have high throughput to enable managing the global state and processing fraud proofs. Sharding allows processing even more global state for applications that cannot be executed well on layer-2. Finally, layer-2 protocols are necessary to achieve low-cost low-latency transactions with high throughput for end-users, and blockchain federation to allow for interoperability between different architectures and virtual machines.

References

1. Adya, A.: Weak consistency: a generalized theory and optimistic implementations for distributed transactions. Ph.D. thesis, Massachusetts Institute of Technology (1999)
2. Avarikioti, Z., Litos, O.S.T., Wattenhofer, R.: Cerberus channels: incentivizing watchtowers for Bitcoin. In: Financial Cryptography and Data Security, pp. 346–366. Kota Kinabalu, Sabah, Malaysia, February 2020
3. Avarikiotia, Z., Kokoris-Kogias, E., Wattenhofer, R., Zindros, D.: Brick: asynchronous incentive-compatible payment channels. In: International Conference on Financial Cryptography and Data Security (2021)
4. Bernstein, P.A., Goodman, N.: Concurrency control in distributed database systems. ACM Comput. Surv. (CSUR) **13**(2), 185–221 (1981)
5. Bharambe, A.R., Agrawal, M., Seshan, S.: Mercury: supporting scalable multi-attribute range queries. In: SIGCOMM Conference, pp. 353–366, Portland, Oregon, August 2004

6. Buchman, E., Kwon, J., Milosevic, Z.: The latest gossip on BFT consensus (2018). arXiv preprint arXiv:1807.04938
7. Burrows, M.: The Chubby lock service for loosely-coupled distributed systems. In: Symposium on Operating System Design and Implementation, pp. 335–350, Seattle, Washington, November 2006
8. Castro, M., Liskov, B.: Practical byzantine fault tolerance. In: Symposium on Operating System Design and Implementation, pp. 173–186, New Orleans, Louisiana, February 1999
9. Clement, A., Wong, E.L., Alvisi, L., Dahlin, M., Marchetti, M.: Making Byzantine fault tolerant systems tolerate byzantine faults. In: Symposium on Networked System Design and Implementation, pp. 153–168, Boston, Massachusetts, April 2009
10. Corallo, M.: Compact block relay. https://github.com/TheBlueMatt/bips/blob/master/bip-0152.mediawiki. Accessed June 2020
11. Corallo, M.: The fast Internet Bitcoin Relay Engine (FIBRE). http://www.bitcoinfibre.org/. Accessed June 2020
12. Corbett, J.C., Dean, J., Epstein, M., Fikes, A., Frost, C., Furman, J.J., Ghemawat, S., Gubarev, A., Heiser, C., Hochschild, P., Hsieh, W.C., Kanthak, S., Kogan, E., Li, H., Lloyd, A., Melnik, S., Mwaura, D., Nagle, D., Quinlan, S., Rao, R., Rolig, L., Saito, Y., Szymaniak, M., Taylor, C., Wang, R., Woodford, D.: Spanner: Google's globally distributed database. ACM Trans. Comput. Syst. **31**(3), 8-1 (2013)
13. Cowling, J., Liskov, B.: Granola: low-overhead distributed transaction coordination. In: USENIX Annual Technical Conference (2012)
14. Croman, K., Decker, C., Eyal, I., Gencer, A.E., Juels, A., Kosba, A.E., Miller A., Saxena, P., Shi, E., Sirer, E.G., Song, D., Wattenhofer, R.: On scaling decentralized blockchains—(A Position Paper). In: Financial Cryptography and Data Security, pp. 106–125, Christ Church, Barbados, February 2016
15. Demers, A.J., Greene, D.H., Hauser, C., Irish, W., Larson, J., Shenker, S., Sturgis, H.E., Swinehart, D.C., Terry, D.B.: Epidemic algorithms for replicated database maintenance. In: ACM Symposium on Principles of Distributed Computing, pp. 1–12, Vancouver, Canada, August 1987
16. Drya, T. (2016) Unlinkable outsourced channel monitoring. In: Scaling Bitcoin Milan (2016)
17. Dwork, C., Naor, M.: Pricing via processing or combatting junk mail. In: Annual International Cryptology Conference, pp. 139–147, Santa Barbara, California, August 1992
18. Dwork, C., Lynch, N.A., Stockmeyer, L.J.: Consensus in the presence of partial synchrony. J. ACM **35**(2), 288–323 (1988)
19. Dziembowski, S., Faust, S., Hostáková, K.: General state channel networks. In: Computer and Communications Security, pp. 949–966, Toronto, Canada, October 2018
20. El-Hindi, M., Binnig, C., Arasu, A., Kossmann, D., Ramamurthy, R.: BlockchainDB—a shared database on blockchains. Proc. VLDB Endowm. **12**(11), 1597–1609 (2019)
21. Etherscan: Ethereum average gas limit chart. https://etherscan.io/chart/gaslimit. Accessed June 2020
22. Gencer, A.E., Basu, S., Eyal, I., van Renesse, R., Sirer, E.G.: Decentralization in Bitcoin and Ethereum networks. In: Financial Cryptography and Data Security, pp. 439–457, Porta Blancu, Curaçao, February 2018
23. Gilad, Y., Hemo, R., Micali, S., Vlachos, G., Zeldovich, N.: Algorand: scaling byzantine agreements for cryptocurrencies. In: Symposium on Operating Systems Principles, pp. 51–68, Shanghai, China, October 2017
24. Göbel, J., Krzesinski, A.E.: Increased block size and Bitcoin blockchain dynamics. In: 27th International Telecommunication Networks and Applications Conference, ITNAC 2017, Melbourne, Australia, November 22–24, 2017, pp. 1–6 (2017)
25. Herlihy, M.: Atomic cross-chain swaps. In: ACM Symposium on Principles of Distributed Computing, pp. 245–254, London, United Kingdom, July 2018
26. Ittay E., Gencer, A.E., Sirer, E.G., van Renesse, R.: Bitcoin-NG: a scalable blockchain protocol. In: Symposium on Networked System Design and Implementation, pp. 45–59, Santa Clara, California, March 2016

27. Kalodner, H.A., Goldfeder, S., Chen, X., Matthew Weinberg, S., Felten. Arbitrum, E.W.: Scalable, private smart contracts. In: USENIX Security Symposium, pp. 1353–1370, Baltimore, Maryland, August 2018
28. Kiayias, A., Russell, A., David, B., Oliynykov, R.: Ouroboros: a provably secure proof-of-stake blockchain protocol. In: Annual International Cryptology Conference, pp. 357–388, Santa Barbara, California, August 2017
29. Klarman, U., Basu, S., Kuzmanovic, A., Sirer, E.G.: bloXroute: A scalable trustless blockchain distribution network. Bloxroute White Paper (2018)
30. Kogias, E.K., Jovanovic, P., Gailly, N., Khoffi, I., Gasser, L., Ford, B.: Enhancing bitcoin security and performance with strong consistency via collective signing. In: USENIX Security Symposium, pp. 279–296, Austin, Texas, August 2016
31. Kogias, E.K., Jovanovic, P., Gasser, L., Gailly, N., Syta, E., Ford, B.: OmniLedger: a secure, scale-out, decentralized ledger via sharding. In: IEEE Symposium on Security and Privacy, pp. 583–598, San Francisco, California, May 2018
32. Kotla, R., Alvisi, L., Dahlin, M., Clement, A., Wong, E.L.: Zyzzyva: speculative byzantine fault tolerance. In: Symposium on Operating Systems Principles, pp. 45–58, Stevenson, Washington, October 2007
33. Kung, H.T., Robinson, J.T.: On optimistic methods for concurrency control. In: International Conference on Very Large Data Bases, p. 351, Rio de Janeiro, Brazil, October 1979
34. Kwon, J., Buchman, E.: Cosmos whitepaper. https://cosmos.network/resources/whitepaper. Accessed March 2021
35. Lamport, L.: Using time instead of timeout for fault-tolerant distributed systems. ACM Trans. Program. Lang. Syst. **6**(2), 254–280 (1984)
36. Lamport, L.: The part-time parliament. ACM Trans. Comput. Syst. **16**(2), 133–169 (1998)
37. Lokhava, M., Losa, G., Mazières, D., Hoare, G., Barry, N., Gafni, E., Jove, J., Malinowsky, R., McCaleb, J.: Fast and secure global payments with Stellar. In: Symposium on Operating Systems Principles, pp. 80–96, Huntsville, Ontario, Canada, October 2019
38. Luu, L., Narayanan, V., Zheng, C., Baweja, K., Gilbert, S., Saxena, P.: A secure sharding protocol for open blockchains. In: Computer and Communications Security, pp. 17–30, Vienna, Austria, October 2016
39. McCorry, P., Bakshi, S., Bentov, I., Meiklejohn, S., Miller, A.: Pisa: arbitration outsourcing for state channels. In: Advances in Financial Technologies, pp. 16–30, Zürich, Switzerland, October 2019
40. Merkle, R.C.: A digital signature based on a conventional encryption function. In: Annual International Cryptology Conference, pp. 369–378, Santa Barbara, California, August 1987
41. Miller, A., Bentov, I., Kumaresan, R., McCorry, P.: Sprites: payment channels that go faster than lightning (2017). arXiv:1702.05812
42. Miller, A., Xia, Y., Croman, K., Shi, E., Song, D.: The honey badger of BFT protocols. In: Computer and Communications Security, pp. 31–42, Vienna, Austria, October 2016
43. Morrison, D.R.: PATRICIA—practical algorithm to retrieve information coded in alphanumeric. J. ACM **15**(4), 514–534 (1968)
44. Mu, S., Cui, Y., Zhang, Y., Lloyd, W., Li, J.: Extracting more concurrency from distributed transactions. In: Symposium on Operating System Design and Implementation, pp. 479–494, Broomfield, Colorado, October 2014
45. Nakamoto, S.: Bitcoin: a peer-to-peer electronic cash system (2008)
46. Offchain Labs: Arbitrum rollup basics. https://developer.offchainlabs.com/docs/rollup_basics. Accessed January 2022
47. Oki, B.M., Liskov, B.: Viewstamped replication: a general primary copy. In: ACM Symposium on Principles of Distributed Computing, pp. 8–17, Toronto, Canada, August 1988
48. Optimism PBC: Optimistic ethereum documentation. https://community.optimism.io/. Accessed Jan 2022
49. Pinar Ozisik, A., Andresen, G., Levine, B.N., Tapp, D., Bissias, G., Katkuri, S.: Graphene: efficient interactive set reconciliation applied to blockchain propagation. In: SIGCOMM Conference, pp. 303–317, Beijing, China, August 2019

50. Ponnapalli, S., Shah, A., Tai, A., Banerjee, S., Chidambaram, V., Malkhi, D., Wei, M.: Scalable and efficient data authentication for decentralized systems (2019). arXiv preprint arXiv:1909.11590
51. Poon, J., Buterin, V.: Plasma: scalable autonomous smart contracts. White Paper (2017)
52. Poon, J., Dryja, T.: The bitcoin lightning network: scalable off-chain instant payments (2016). https://lightning.network/lightning-network-paper.pdf
53. Redman, J.: BCH upgrades: What's new and What's next (2018). https://news.bitcoin.com/bch-upgrades-whats-new-and-whats-next/. Accessed June 2020
54. Rocket, T., Yin, M., Sekniqi, K., van Renesse, R., Sirer, E.G.: Scalable and probabilistic leaderless BFT consensus through metastability (2019). arXiv preprint arXiv:1906.08936
55. Rogaway, P.: Formalizing human ignorance: collision-resistant hashing without the keys. IACR Cryptol. ePrint Arch. **2006**, 281 (2006)
56. Schneider, F.B.: Implementing fault-tolerant services using the state machine approach: a tutorial. ACM Comput. Surv. **22**(4), 299–319 (1990)
57. Sivaraman, V., Venkatakrishnan, S.B., Ruan, K., Negi, P., Yang, L., Mittal, R., Fanti, G.C., Alizadeh, M.: High throughput cryptocurrency routing in payment channel networks. In: Symposium on Networked System Design and Implementation, pp. 777–796, Santa Clara, California, February 2020
58. Sompolinsky, Y., Zohar, A.: Secure high-rate transaction processing in bitcoin. In: Financial Cryptography and Data Security, pp. 507–527, San Juan, Puerto Rico, January 2015
59. Stoica, I., Morris,R.T., Karger, D.R., Frans Kaashoek,M., Balakrishnan, H.: Chord: a scalable peer-to-peer lookup service for internet applications. In:SIGCOMM Conference, pp. 149–160, San Diego, California, August 2001
60. Syta, E., Jovanovic, P., Kokoris-Kogias, E., Gailly, N., Gasser, L., Khoffi, I., Fischer, M.J., Ford, B.: Scalable bias-resistant distributed randomness. In:IEEE Symposium on Security and Privacy, pp. 444–460, San Jose, California, May 2017
61. The Ethereum Foundation. Ethereum 2.0 Specifications. https://github.com/ethereum/eth2.0-specs. Accessed August 2020
62. Wang, G., Shi, Z.J., Nixon, M., Han, S.: SoK: sharding on blockchain. In: Advances in Financial Technologies, pp. 41–61, Zürich, Switzerland, October 2019
63. Wang, J., Wang, H.: Monoxide: scale out blockchains with asynchronous consensus zones. In: Symposium on Networked System Design and Implementation, pp. 95–112, Boston, Massachusetts, February 2019
64. Wood, G.: Ethereum: a secure decentralised generalised transaction ledger. Ethereum project yellow paper **151**, 1–32 (2014)
65. Wood, G.: Polkadot: vision for a heterogeneous multi-chain framework. White Paper (2016)
66. Yin, M., Malkhi, D., Reiter, M.K., Golan-Gueta, G., Abraham, I.: HotStuff: BFT consensus with linearity and responsiveness. In: ACM Symposium on Principles of Distributed Computing, pp. 347–356, Toronto, Canada, July 2019
67. Zamani, M., Movahedi, M., Raykova., M.:Rapidchain: scaling blockchain via full sharding. In: Computer and Communications Security, pp. 931–948, Toronto, Canada, October 2018
68. Zamyatin, A., Harz, D., Lind, J., Panayiotou, P., Gervais, A., Knottenbelt, W.J.: XCLAIM: trustless, interoperable, cryptocurrency-backed assets. In: IEEE Symposium on Security and Privacy, pp. 193–210, San Francisco, California, May 2019
69. Zhang, Y., Setty, S.T.V., Chen, Q., Zhou, L., Alvisi, L.: Byzantine ordered consensus without byzantine oligarchy. In: Symposium on Operating System Design and Implementation, pp. 633–649 (2020)
70. Zilliqa Team: The Zilliq technical whitepaper. Technical Report (2017)