This document is a draft; feedback is welcome.

LazyLedger: A Distributed Data Availability Ledger With Client-Side Smart Contracts

Mustafa Al-Bassam[1]

[1]Department of Computer Science
University College London
m.albassam@cs.ucl.ac.uk

Abstract

We propose LazyLedger, a design for distributed ledgers where the blockchain is optimised for solely ordering and guaranteeing the availability of transactions. Responsibility of executing and validating transactions is shifted to only the clients that have an interest in those transactions. As the core function of the consensus system of a distributed ledger is to order transactions and ensure their availability, consensus participants do not necessarily need to be concerned with the content of those transactions. This reduces the problem of block verification to data availability verification, which can be achieved probabilistically without downloading the whole block. The amount of resources required to reach consensus can thus be minimised, as transaction validity rules can be decoupled from consensus rules. We also implement and evaluate several example LazyLedger applications, and validate that the workload of clients of specific applications does not significantly increase when the workload of other applications increase.

1 Introduction

So far, blockchain-based distributed ledger platforms such as Bitcoin [1] and Ethereum [2] have adopted similar consensus design paradigms, where the validity of the blocks proposed by block producers is determined by (*i*) whether it is the block producer's turn to propose a block and (*ii*) whether the transactions in the block are valid according to some state machine. Traditional consensus protocols such as Practical Byzantine Fault Tolerance [3] have also taken a similar approach, where consensus nodes (replicas) process transactions according to a state machine.

The scalability issues that have plagued decentralised blockchains [4] can be attributed to the fact that in order to run a node that validates the blockchain, the node must download, process and validate every transaction included in the chain. As a result, various scalability efforts have emerged including on-chain scaling via sharding [5, 6], which aims to split the state of the blockchain into multiple shards so that transactions can be processed by different consensus groups in parallel, and off-chain scaling via state channels [7, 8], which takes the approach of moving transactions off-chain and using the blockchain as a settlement layer.

However, it is also worth exploring alternative blockchain design paradigms that may be suitable for different types of applications, where nodes that need to validate the blockchain in order to determine the correct chain do not need to validate the contents of the blocks. Instead, the end-users of applications that store information on the blockchain can be concerned with the validation of such contents. This would remove the bottleneck where nodes need to validate

everyone else's transactions, and reducing the problem of validating the blockchain to simply verifying that the contents of the block are available (the data availability problem [5]), so that end-users can meaningfully access the information needed to apply state transitions on their applications. In such a paradigm, the blockchain is used solely for ordering and making available messages, rather than executing and verifying the state machine transitions of transactions. Because messages for applications are executed by end-users off-chain, the logic of these applications does not need to be defined on-chain, and thus application logic can be written in any programming language or environment, and changing the logic does not require a hard-fork of the chain.

An interesting result of reducing blockchain validation to the data availability problem is that one can fully achieve consensus on new messages without downloading the entire set of messages, using probabilistic data availability verification techniques [5], as consensus participants do not need to process the contents of messages.

In this paper, we make the following contributions:

- We design a blockchain, LazyLedger, where consensus and transaction validity is decoupled, and describe two alternative block validity rules which just ensure that block data is available. One is a simple rule where nodes simply download the blocks themselves, and the other is probabilistic but more efficient as nodes do not need to download entire blocks.
- We build an application-layer on top of our proposed blockchain, where end-user clients can efficiently query the network for data relating only to its applications, and only need to execute transactions related to its applications.
- We implement and evaluate several example LazyLedger applications; including a currency, a name registrar and a petitions system.

Outline: Section 2 presents a background of the technical concepts LazyLedger relies on; Section 3 presents the LazyLedger threat model, node types, and blockchain model; Section 4 presents the block validity rules of the LazyLedger blockchain; Section 5 presents the LazyLedger application model and how applications can be built on top of its blockchain; Section 6 presents an evaluation of a prototype of LazyLedger and some example applications; Section 8 presents a comparison with related work; and Section 9 concludes.

2 Background

2.1 Blockchains

The data structure of a blockchain consists of a chain of blocks. Each block contains two components: a header and a list of transactions. In addition to other metadata, the header stores at minimum the hash of the previous block (thus enabling the chain property), and the root of the Merkle tree that consists of all transactions in the block.

Blockchain networks implement a consensus algorithm [9] to determine which chain should be favoured in the event of a fork, *e.g.,* if proof-of-work [1] is used, then the chain with the most accumulated work is favoured. They also have a set of transaction validity rules that dictate which transactions are valid, and thus blocks that contain invalid transactions will never be favoured by the consensus algorthim and should in fact always be rejected.

Full nodes (also known as 'fully validating nodes') are nodes which download both the block headers as well as the list of transactions, verifying that all transactions are valid according to some transaction validity rules. This is necessary in order to know which blocks have been accepted by the consensus algorithm.

There are also 'light' clients which only download block headers, and assume that the list of transactions are valid according to the transaction validity rules. These nodes verify blocks against the consensus rules, but not the transaction validity rules, and thus assume that the consensus is honest in that they only included valid transactions. They therefore do not fully execute the consensus algorithm to know which blocks are accepted, and may end up in a situation where they accept blocks that contain invalid

transactions, that full nodes have rejected.

2.2 Sampling-Based Data Availability

The 'data availability problem' asks how a client–such as a light client–that only downloads block headers, but not the corresponding block data (*e.g.,* list of transactions), can satisfy itself that the block data is not being withheld by the producer of the block (*e.g.,* a miner), and that the full data is indeed available to the network.

Al-Bassam *et al.* [5] propose a solution to this problem based on erasure coding and random sampling. The solution was proposed in the context of state transition fraud proofs, however it is of independent interest. We summarise the scheme here.

Erasure codes are error-correcting codes [10] that can transform a message consisting of k shares (*i.e.,* pieces) into a bigger extended message of n shares, such that the original message can be recovered by any subset k' of the n shares. The ratio $\frac{k'}{n}$ (the code rate) depends on the erasure code used and its parameters. For example, Reed-Solomon erasure codes [11] can support $k' = \frac{n}{2}$, which means that only half of the erasure coded data is needed to recover the original data.

To allow clients to be sure that block data is available, block headers contain a pointer to the Merkle tree of the erasure coded version of the data. In order for an adversarial block producer to withhold any part of the block, they must withhold at least k' out of n shares of the block (*e.g.,* with standard Reed-Solomon coding this would be at least 50% of the block). Clients can then sample multiple random shares from the block, and if it does not receive a response for one of its samples because it is unavailable, then it considers the whole block to be unavailable, and does not accept the block. If an adversarial block producer has withheld k' out of n shares, then there is a high probability that the client will sample an unavailable piece and reject the block.

However because the block producer may incorrectly compute the erasure code or the Merkle tree, thus making the block data unrecoverable, it is necessary to allow clients to receive 'fraud proofs' from full nodes to alert them that the erasure code is incorrect, causing the client to download the block data, recompute the erasure code and verify that it does not match the Merkle root.

To prevent clients from needing to download the entire block data (which would defeat the goal of data availability proofs being more efficient than downloading the whole data yourself), two-dimensional erasure coding is used, which limits these fraud proofs to a specific axis as only one row or column needs to be downloaded, thus the fraud proof size would be $O(\sqrt{n})$ for a block with n shares. However this also requires clients who want data availability guarantees to download a Merkle root for each row and column as part of the block header, rather than a single Merkle root of the entire data, thus the number of hashes that need to be downloaded increases from 1 to $2\sqrt{n}$.

It also worth noting that for this scheme to provide any guarantees, there must be a minimum number of clients in the network that are making enough samples to force the block producer to release more than k' shares to satisfy all of those samples, as if less than k' shares are released, the block data may not be recoverable from those shares. This is because clients gossip downloaded samples to 'full' nodes that request it, so that they can recover the full block with enough shares.

3 Model

3.1 Threat Model and Nodes

We consider the following types of nodes in LazyLedger:

- **Consensus nodes.** These are nodes which participate in the consensus set, to decide which blocks should be added to the chain.
- **Storage nodes.** These are nodes which store a copy of all of the data in the blockchain and its blocks.
- **Client nodes.** These are effectively the end-users of the blockchain system. They participate in applications or use cases that use the

blockchain, and receive transaction data or messages from storage nodes relevant to their applications.

These nodes are all connected to each other in a peer-to-peer network, *e.g.,* all node types may have some connections with any other node type and the topology of the network is non-hierarchical. However, client nodes may wish to ensure that they are connected to at least one storage node if they wish to utilise their services.

We assume that honest nodes not under an eclipse attack [12] and are thus connected to at least one other honest node; that is, a node that will follow the protocols described in Section 4 and relay messages. This implies that the network is not split, so that there is always a network path between two honest nodes. Additionally, there is at least one honest storage node in the network.

We also assume that there is a maximum network delay δ so that if an honest node can receive a message from the network at time T, then any other honest node can also do so at time $T' \leq T + \delta$.

3.2 Block Model

We assume a blockchain data structure that at minimum consists of a hash-based chain of block headers $H = (h_0, h_1, ...)$. Each block header h_i contains the root mRoot_i of a Merkle tree of a list of messages $M_i = (m_i^0, m_i^1, ...)$, such that given a function $\mathsf{root}(M)$ that returns the Merkle root of a list of messages M, then $\mathsf{root}(M_i) = \mathsf{mRoot}_i$. This is not an ordinary Merkle tree, but an ordered Merkle tree we refer to as a 'namespaced' Merkle tree which we describe in Section 5.2. A block header h_i is considered to be valid if given some boolean function

$$\mathsf{blockValid}(h) \in \{\mathsf{true}, \mathsf{false}\}$$

then $\mathsf{blockValid}(h_i)$ must return true.

If a block is valid, then it has the potential to be included in the blockchain. We assume that the blockchain has some consensus rules to decide which valid blocks have consensus to be included in the blockchain, and resolve forks. A block header h_i is considered to have consensus if given some boolean function

$$\mathsf{inChain}(h, \{H_0, H_1, ...\}) \in \{\mathsf{true}, \mathsf{false}\}$$

then $\mathsf{inChain}(h_i, \{H_0, H_1, ...\})$ must return true, each H_j is a distinct chain of block headers and $\{H_0, H_1, ...\}$ is the set of distinct chains observed (there may be multiple in the event of a fork).

Note that computing $\mathsf{inChain}$ on h_i can only return true if and only if $\mathsf{blockValid}(h_i)$ returns true, regardless of the forks to pick from. Apart from this constraint, the specific consensus rules used by $\mathsf{inChain}$ are arbitrary and are out of scope for the design of LazyLedger. For example, $\mathsf{inChain}$ may use proof-of-work or proof-of-stake with the longest chain rule [1, 9].

3.3 Goals

With this threat model in mind, LazyLedger has the following goals:

In the text below, 'messages relevant to the application' means messages that are necessary to compute the state of an application, and is discussed in more depth in Section 5.1.1.

1. **Availability-only block validity.** The result of $\mathsf{blockValid}(h_i)$ should be true if the data behind mRoot_i is available to the network. This consequently means that consensus nodes should not need to execute messages in blocks.

2. **Application message retrieval partitioning.** Client nodes must be able to download all of the messages relevant to the applications they use from storage nodes, without needing to downloading any messages for other applications.

3. **Application message retrieval completeness.** When client nodes download messages relevant to the applications they use from storage nodes, they must be able to verify that the messages they received are the complete set of messages relevant to their applications, for specific blocks, and that there are no omitted messages.

4. **Application state sovereignty.** Client nodes must be able to execute all of the messages relevant to the applications they use to compute the state for their applications, without needing to execute messages from other applications, unless other specific applications are explicitly declared as dependencies.

4 Block Validity Rule Design

The key idea of LazyLedger is that the result of $\mathsf{blockValid}(h_i)$ should only depend on whether the data required to compute mRoot_i is available to the network or not, rather on whether any of the messages in the block correspond to transactions that satisfy the rules of some state machine (Goal 1 in Section 3.3). This way, we can decouple transaction validity rules from the consensus rules, as the result of inChain does not depend on the contents of the messages in the block M_i, when $\mathsf{blockValid}(h_i)$ is computed (recall inChain on h_i can only return true if and only if $\mathsf{blockValid}(h_i)$ returns true).

We consider that checking the availability of the data necessary to recompute mRoot_i is the bare minimum necessary requirement to have a useful functioning blockchain. This is because, as we shall see in Section 5, clients need to know the transactions that have occurred in the blockchain in order to know the state of applications on the blockchain and thus do anything useful. If the data behind a block is unavailable, clients would not be able to compute the state of their applications.

We provide definitions for data availability soundness and agreement, adapted from [5] for the threat model described in Section 3.

Definition 1 (Data Availability Soundness)**.** *If an honest node accepts a block as available, then at least one honest storage node has the full block data or will have the full block data within some known maximum delay $k \times \delta$ where δ is the maximum network delay.*

Definition 2 (Data Availability Agreement)**.** *If an honest node accepts a block as available, then all other honest nodes will accept that block as available within some known maximum delay $k \times \delta$ where δ is the maximum network delay.*

We offer two possible validity rules with different trade-offs. Section 4.1 describes a simple validity rule that satisfies Definition 1 and Definition 2 with 100% probability, for an $O(n)$ bandwidth cost where n is the storage size of the block, because the node must download the entire block data to confirm that it is available. Section 4.2 describes a probabilistic validity rule that satisfies Definition 1 and Definition 2 with a high but less than 100% probability, but with a lower bandwidth cost because only samples from the block need to be downloaded.

4.1 Simplistic Validity Rule

In the Simplistic Validity Rule, $\mathsf{blockValid}(h_i)$ returns true if and only if upon receiving a block header h_i from the network, the node is also able to download M_i from the network and authenticating that the Merkle root of the downloaded M_i is mRoot_i, by checking that $\mathsf{root}(M_i) = \mathsf{mRoot}_i$.

Upon $\mathsf{blockValid}(h_i)$ returning true, the node must distribute h_i and M_i to the nodes it is connected to, should the nodes request the data if they do not have it. The node should thus store M_i for at least δ, the maximum network delay.

Theorem 1. *The Simplistic Validity Rule satisfies Definition 1 (Soundness).*

Proof. If $\mathsf{blockValid}(h_i)$ returns true on an honest node, then the node will distribute M_i to the nodes it is connected to, of at least one of which is honest, and will also run $\mathsf{blockValid}(h_i)$ and distribute M_i, and so on. Thus a storage node will receive M_i within the maximum network delay δ, which there exists at least one of which is honest. □

Theorem 2. *The Simplistic Validity Rule satisfies Definition 2 (Agreement).*

Proof. If $\mathsf{blockValid}(h_i)$ returns true on an honest node, then the node will distribute M_i to the nodes it is connected to, of at least one of which is honest, and will also run $\mathsf{blockValid}(h_i)$ and distribute M_i, and so on. Thus all honest nodes will receive M_i within the

maximum network delay δ, and $\mathsf{blockValid}(h_i)$ will thus return true, causing them to accept h_i as an available block. □

4.2 Probabilistic Validity Rule

For the Probabilistic Validity Rule, $\mathsf{blockValid}(h_i)$ utilises the probabilistic data availability scheme based on random sampling the erasure coded version of the block data M_i described by Al-Bassam *et al.* [5] and summarised in Section 2.2. Proofs for Definition 1 and Definition 2 are provided in [5]. Unlike the Simplistic Validity Rule, this scheme is probabilistic in satisfying these definitions, however it is more efficient because only a part of the block data needs to be downloaded to obtain high probability guarantees that the data is available.

For examples if using the 2D Reed-Solomon coding scheme with a $\frac{1}{4}$ code rate described in [5] in a block that has been divided into 1024 shares, only 15 samples corresponding to 0.4% of the block data needs to be downloaded by a node to achieve a 99% guarantee that the block data is available [5]. Further analysis will be provided in Section 6.

5 Application-Layer Design

5.1 Application Model

Recall in Section 3 that LazyLedger has client nodes which read and write messages in blocks relevant to their application, and that the contents of blocks have no validity rules, and thus any arbitrary message can be included in a block. LazyLedger applications are akin to smart contracts, with the primary difference being that they are executed by end-user clients rather than consensus participants. Thus, application logic is defined and agreed upon entirely offchain by clients of that application, and may therefore be written in any programming language or environment. A client can submit a message to be recorded on the blockchain that specifies a transaction for a specific application, which will then be read and parsed by other clients of that applications, which may then modify the state of that application.

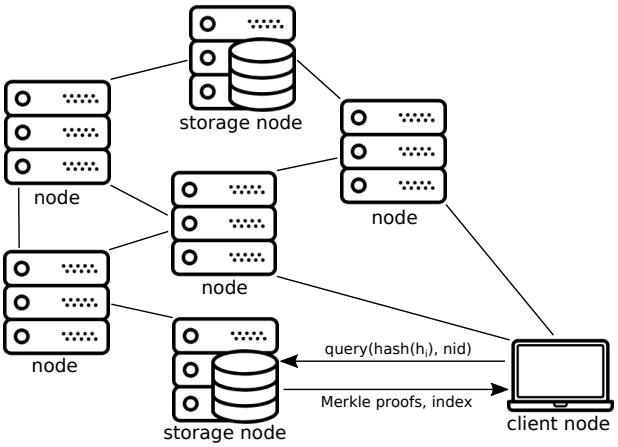

Figure 1: Overview of interaction between client nodes and storage nodes.

Applications are identified by their own 'namespace', and well-formed messages associated with a specific application can be parsed to determine their application namespace. We assume a function $\mathsf{ns}(m)$ that takes as input a message m and returns its namespace ID. Therefore if a client is a user of an application with ID nid, it is interested in reading all messages m in the ledger such that $\mathsf{ns}(m) = \mathsf{nid}$, in order to determine the state of its application.

Because the consensus of the blockchain does not require checking the validity of any transactions included in the blockchain, the ledger may include transactions that are considered invalid according to the logic of certain applications. Therefore we define a state transition function that LazyLedger applications should use that does not return an error. Given an application with ID nid:

$$\mathsf{transition}_{\mathsf{nid}}(\mathsf{state}, \mathsf{tx}) = \mathsf{state}'$$

$\mathsf{transition}_{\mathsf{nid}}(\mathsf{state}, \mathsf{tx})$ cannot return an error because if an adversarial actor included an invalid tx in a block, then the state of the application would end up in a permanently erroneous state. Therefore if tx is considered erroneous by the logic of the transition function, it should simply return the previous state, state, as the new state.

Clients who use an application with ID nid should

agree with each other on the logic or code of $\mathsf{transition}_{\mathsf{nid}}$. If for example, one client decides to use different logic for $\mathsf{transition}_{\mathsf{nid}}$, then that client would reach a different final state for that application than everyone else, which in effect means that they would be using a different application, but it would not effect anyone else.

Interestingly, this means that it is possible for users of an application to decide to change the logic of that application without requiring a hard-fork of the blockchain that would effect other applications. However if immutability of the logic is important, the creator of the application may decide for example that the namespace identifier of the application should be the cryptographic hash of the application's logic.

5.1.1 Cross-Application Calls

Some applications may want to call other applications (*i.e.,* a cross-contract call). There are two scenarios in which an application may want to do this: either as a pre-condition or a post-condition.

Recall in Section 3.3 that Goal 4 of LazyLedger is application state sovereignty, which means that users of an application should not have to execute messages from other irrelevant applications. An application B however becomes relevant to users of application A if is B is a dependency of A, however if A is not a dependency of B, then A is not relevant to the users of B.

In the case of a pre-condition, an application may have a function that can only be executed if another application that it depends is in a certain state. In such a case, in order to validate that these pre-conditions are met, clients of an application must also download and verify the state of the application's dependency applications; however the clients of the dependency application do not need to download the state of the applications which depend on it.

For example, consider a name registrar application where clients can register names only if they send money to a certain address in a different currency application. The clients of the name registrar application would have to also become clients of the currency application, in order to verify that when a name is registered, there is a corresponding transaction that sends the funds to pay for the name to the correct address.

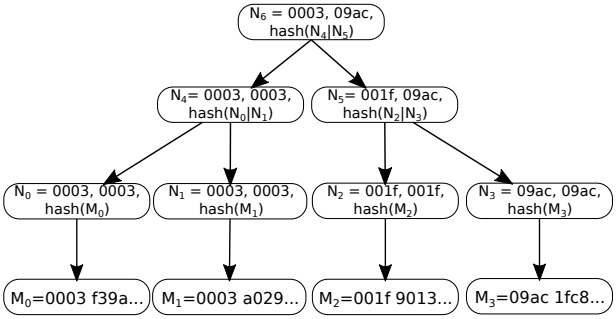

Figure 2: An example of a namespaced Merkle tree.

In the case of a post-condition, an application may want to modify the state of another application after a transaction. Post-conditions are only possible if the application whose state is being modified has explicitly set the application that is executing the post-condition as a dependency application to the post-condition application. This is because in order to execute the post-condition, the clients of the post-condition application would have to download and verify the state of the application executing the post-condition, so if any application was allowed to execute a post-condition in any application, then it would mean that clients would have to download and verify other applications against their will, thus violating Goal 4 in Section 3.3. Post-conditions can however be executed indirectly through sidechain mechanisms such as federated pegs [13].

5.2 Storage Nodes and Namespaced Merkle Tree

In order to satisfy Goal 2 in Section 3.3 to allow client nodes to be able to retrieve all the messages relevant to the application namespaces they are interested in without having to download and parse the entire blockchain themselves (*e.g.,* if they use the Probabilistic Validity Rule, or simply assume that the consensus has a honest-majority that only accepts available blocks), they may query storage nodes for all of the messages in a particular application names-

pace for particular blocks. The storage node can then return Merkle proofs the relevant messages being included in the blocks.

In order to allow storage node to prove to clients that they have returned the complete set of messages for a namespace included in a block's Merkle tree of messages (Goal 3 in Section 3.3), we use a 'namespaced' Merkle tree described below, which is a standard Merkle tree that uses a modified hash function so that each node in the tree includes the range of namespaces of the messages in all of the descendants of each node.

In a namespaced Merkle tree, each non-leaf node in the tree contains the lowest and highest namespace identifiers found in all the leaf nodes that are descendants of the non-leaf node, in addition to the hash of the concatenation of the children of the node. This enables Merkle inclusion proofs to be created that prove to a verifier that all the elements of the tree for a specific namespace have been included in a Merkle inclusion proof.

The Merkle tree can be implemented using standard unmodified Merkle tree algorithms, but with a modified hash algorithm that depends on an existing hash function, that prefixes hashes with namespace identifiers. Suppose $\mathsf{hash}(x)$ is a cryptographically secure hash function such as SHA-256. We define a wrapper function $\mathsf{nsHash}(x)$ that produces hashes prefixed with namespace identifiers. A namespaced hash has the format $\mathsf{minNs}||\mathsf{maxNs}||\mathsf{hash(x)}$, where minNs is the lowest namespace identifier found in all the children of the node that the hash represents, and maxNs is the highest.

The value of minNs and maxNs in the output of $\mathsf{nsHash}(x)$ depends on if the input x is a leaf or two concatenated tree nodes, as illustrated by Figure 2. If x is a leaf, then $\mathsf{minNs} = \mathsf{maxNs} = \mathsf{ns}(x)$, as the hash contains only one leaf with a single namespace.

If x is two concatenated tree nodes, then $x = \mathsf{left}||\mathsf{right}$ where $\mathsf{left} = \mathsf{leftMinNs}||\mathsf{leftMaxNs}||\mathsf{hash(x)}$ and $\mathsf{right} = \mathsf{rightMinNs}||\mathsf{rightMaxNs}||\mathsf{hash(x)}$. Thus in the output of $\mathsf{nsHash}(x)$, $\mathsf{minNs} = \mathsf{min}(\mathsf{leftMinNs}, \mathsf{rightMinNs})$ and $\mathsf{maxNs} = \mathsf{max}(\mathsf{leftMaxNs}, \mathsf{rightMaxNs})$.

An adversarial consensus node may attempt to produce a block that contains a Merkle tree with children that are not ordered correctly. To prevent this, we can set a condition in nsHash such that there is no valid hash when $\mathsf{leftMaxNs} \geq \mathsf{rightMinNs}$, and thus there would be no valid Merkle root for incorrectly ordered children. Therefore $\mathsf{blockValid}(h_i)$ would return false in the simplistic and probabilistic validity rules as no possible M_i where $\mathsf{root}(M_i) = \mathsf{mRoot}_i$. Additionally, recall that $\mathsf{root}(M_i) = \mathsf{mRoot}_i$ and thus $\mathsf{blockValid}(h_i)$ would also return false if the Merkle root of the tree is constructed incorrectly, *e.g.,* if the minimum and maximum namespaces for a node in the tree are incorrectly labelled.

Because only the hash function is being modified in the Merkle tree, the Merkle tree is generated, and Merkle proofs are verified using standard algorithms. However, during Merkle proof verification, and extra step is necessary in order to verify that the proofs covers all of the messages for a specific namespace.

A client node can send a query $\mathsf{query}(\mathsf{hash}(h_i), \mathsf{nid})$ to a storage node to request all of the messages in block h_i that have namespace ID nid. The storage node replies with a list of Merkle proofs $\mathsf{proofs} = (\mathsf{proof}_0, \mathsf{proof}_1, ..., \mathsf{proof}_n)$ and an index index that specifies the index in the tree in which proof_0 is located. In addition to the client node verifying all the proofs, the client node also verifies that the highest namespace in all of the left siblings included in proof_0 are smaller than nid, and the lowest namespace in all of the right siblings included in proof_n are larger than nid.

If a block has no messages associated with nid, then only one proof proof_0 is returned which corresponds to the child in the tree where the child to the left of it is smaller than nid but the child to the right of it is larger than nid. The actual message in the child does not need to be included in the proof as the purpose of the proof would just be to show that there are no messages in the tree for nid.

Theorem 3. *Assuming the Merkle tree is correctly constructed, an incomplete set of Merkle proofs* $\mathsf{proofs} = (\mathsf{proof}_0, \mathsf{proof}_1, ..., \mathsf{proof}_n)$ *for a request for the messages of* nid *can always be detected.*

Proof. Let us assume that an adversary returns an incomplete set of correct proofs $\mathsf{proofs} =$

$(\mathsf{proof}_0, \mathsf{proof}_1, ..., \mathsf{proof}_n)$ for nid, and index is the index in the tree that proof_0 is located at.

If an omitted message for nid has an index lower than index, then proof_0 will contain a left sibling node with a maximum namespace maxNs where maxNs > nid, thus proving that there is an omitted message to the left of the proof set.

If an omitted message for nid has an index higher than index$+n$, then proof_n will contain a right sibling node with a minimum namespace minNs where nid < minNs, thus proving that there is an omitted message to the right of the proof set. □

5.3 DoS-resistance

In the design of LazyLedger, consensus nodes are not responsible for validating transactions, and thus an adversarial client may submit many invalid transactions for namespaces, forcing clients to download many invalid transactions. In a permissioned system, consensus nodes can choose which clients can submit transactions. However in a permissionless system, there ought to be a way to prioritise transactions and to make it expensive to conduct DoS attacks.

Consensus nodes can thus choose to prioritise transactions that include transaction fees. However, any transaction fee system must not require client nodes that read messages from the application namespaces they are interested in, to also validate the application that implements the currency system that transaction fees are paid in. To achieve this, when a message is submitted to consensus nodes for inclusion in a block, the submitter of the message can also submit 'transaction fee' message for a different currency system application running in parallel, and also specify the hash of the 'dependency' message that the fee is paying for, where the fee can only be collected if the message behind the specified hash is included in the same block. The client nodes of the currency system application do not have to download the message itself, but simply verify an inclusion proof that the dependency message is included in the same block.

Interestingly there does not need to be a native currency to the system, as consensus nodes can choose to accept transaction fees in any currency application that they choose to recognise.

6 Implementation and Performance

We implemented a prototype of LazyLedger in 2,865 lines of Golang code. The code has been released as a free and open-source project.[1]

As well as the core LazyLedger system, we also implemented (and released) several example applications using LazyLedger. Each application's state is implemented as one or more key-value stores that can be read from or modified. Applications include:

- A currency application where clients publish messages that are transactions for the transfer of funds between addresses that are elliptic curve public keys. Transactions are signed by the public keys of senders, and specify the amount of funds to send and the recipient address. In the key-value store, keys are public keys, and values are the corresponding balance of each public key, which is updated after each valid transaction.

- A name registration application where clients can (*i*) send a balance top-up transaction to the registrar's public key using a dependent currency application, so that clients can pay for name registrations using their balance with the registrar and (*ii*) send a registration transaction to register a specified name to their public key, which reduces the balance of the registrant, if their balance is sufficient. The registration application has one key-value store representing the in-app topped-up balance of each public key, and another key-value store where each key represents a registered name, and each value represents the public key the name has been registered by.

- A simple petition application where clients can (*i*) add new petitions by providing a description of what the petition is about and (*ii*) sign petitions with their public key, to increase their signatory counts. In the key-value store, each

[1] `https://github.com/musalbas/lazyledger-prototype`

key represents the ID of the petition, which is incremented each time a new petition is added, and each corresponding value represents the total number of signatures.

- A dummy application for testing purposes which adds arbitrary sized specified key-value pairs to the state.

We present an evaluation of LazyLedger's performance and scalability properties.

Figure 3 compares how much data needs to be downloaded to execute the Simple Validity Rule and the Probabilistic Validity Rule to verify data availability, for varying block sizes. As expected, there is a linear relationship between block size and data downloaded for the Simple Validity Rule, as this requires downloading all of the block data to ensure that it is available. However, we can see that the relationship between block size and data downloaded for the Probabilistic Validity Rule is logarithmic. This is because in order to execute the Probabilistic Validity Rule, nodes download a fixed number of samples and their corresponding Merkle proofs. The size of the Merkle proofs increase logarithmically with the size of the blocks.

Figure 4 compares the response size of queries for a specific namespace to a storage node (application proofs), for varying amounts of messages of different namespaces (measured by total bytes) that are not relevant to that query. We use currency application messages as the relevant queried namespace (although any other application could be used), fixing the number of currency messages in the block to 10, but increasing the total size of dummy application messages. We can observe that that for both simple blocks and probabilistic blocks, the size of the application proofs for the relevant application only increases logarithmically, because although messages that are not in the relevant namespace do not need to be downloaded, the size of the Merkle proofs for those messages increase logarithmically as the number of total messages in the block increases. The size of the application proofs for probabilistic blocks are smaller because a two-dimensional erasure code is used, where each column and row gets its own Merkle tree, and thus the Merkle proofs are smaller because there are less items in each tree.

Figure 5 follows the same setup as Figure 4, however instead of comparing the size of the applications proofs for the currency application, we compare the size of the state that needs to be stored by users of the relevant application (in this case, the currency application). As expected, we observe that as the size of the state of other applications increase, the size of the state that needs to be stored for the currency application remains static.

Figure 6 and Figure 7 illustrate how the size of application proofs may vary for an application that has a dependency application. In this case we use the name registration application as an example, which requires users to follow the state of a currency application so that balance top-up transactions to the registrar can be verified. In the two graphs, we setup multiple instances of the name registration application for multiple registrar, but the user is only interested in following one of them. In Figure 6 we can observe that as the number of top-up transactions for the irrelevant name registration applications increase, the size of the application proofs for the relevant name registration application increases linearly, because the user must also download application proofs for the currency application, which has transactions being added to it by users of the other name registration applications. This extreme case defeats any scalability gains of LazyLedger, since all transactions require transactions in dependency applications that other users may follow.

However, Figure 7 shows the same but in the case of name registration transactions instead of balance top-up transactions. Here we see that irrelevant name registration transactions do not linearly increase the size of application proofs that need to be downloaded for other users, because only users of the relevant name registration application need to have knowledge of the registered names, and no dependency application is impacted.

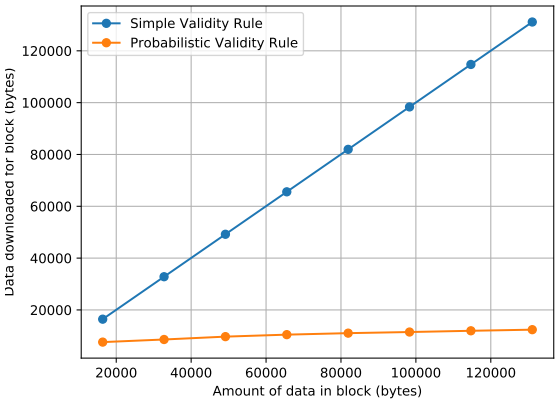

Figure 3: A graph showing how much data needs to be downloaded to execute the block validity rule to validate data availability versus the size of the block. For the Probabilistic Validity Rule, 15 samples are used.

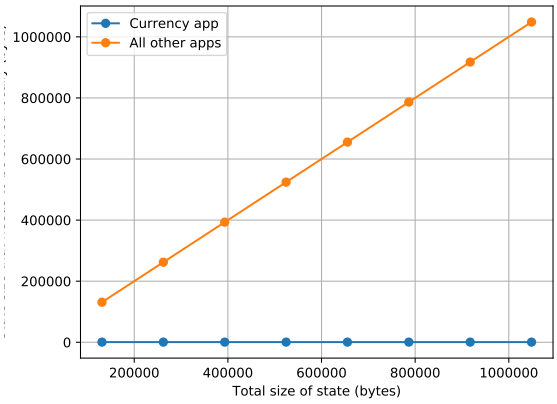

Figure 5: A graph showing the size of the state that needs to be stored after a block versus the total size the state of all apps in the block, for a currency app and all other apps.

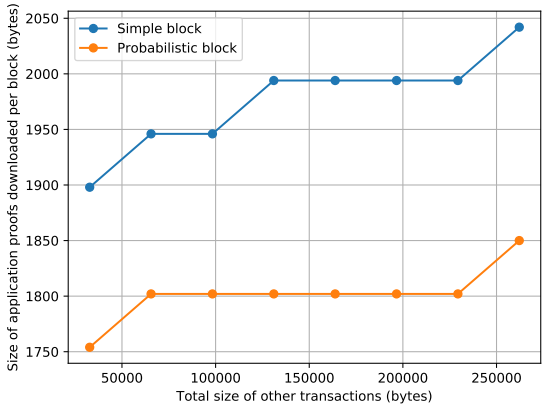

Figure 4: A graph showing the size of application proofs in a block for a currency application with 10 transactions versus the total size of all of the other transactions in the block.

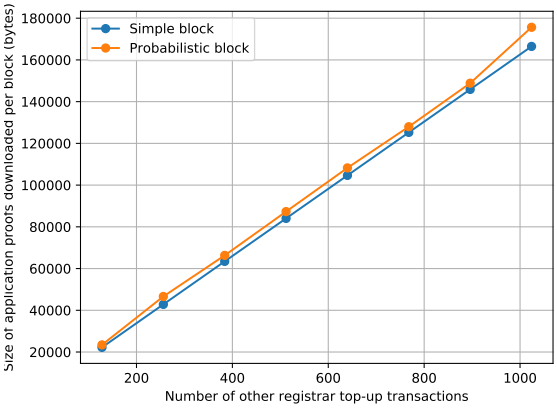

Figure 6: A graph showing the size of application proofs in a block for an instance of a registrar application with 10 top-up transactions versus the number of top-up transactions for other registrar application instances in the block.

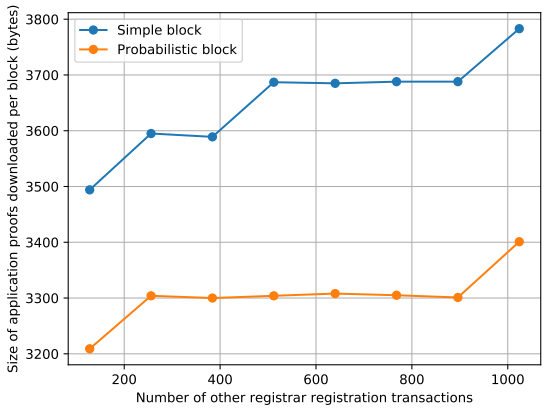

Figure 7: A graph showing the size of application proofs in a block for an instance of a registrar application with 10 registration transactions versus the number of registration transactions for other registrar application instances in the block.

7 Discussion

Light Clients

One of the limitations of LazyLedger is that per-application light clients are not possible because application messages are not validated by consensus nodes, thus clients cannot assume that an honest majority has validated them.

Hard Forking

One of the interesting aspects of the LazyLedger design is its consequences on blockchain governance, in particular hard forks. Traditionally, hard forks have been used in the past to change transaction protocol rules [14] or to reverse damage caused popular by smart contracts being compromised, such as the DAO hack [15].

However with LazyLedger, as there are no transaction-specific protocol rules and blocks may contain any arbitrary data, hard forks to change transaction rules or change the state of the system are not possible or necessary, as the interpretation of transactions are left to the device of the end-user clients rather than the consensus. Thus if users of a specific application decide they want to change the state of or 'upgrade' an application, they can do so without the permission of the consensus or any on-chain effects or changes.

8 Related Work

The namespaced Merkle tree in LazyLedger is inspired by the 'flagged' Merkle tree concept by Crosby and Wallach [16], where each node in the tree is has a flag that represents the attributes that its leafs has.

Mastercoin (now OmniLayer) [17] is a blockchain application system predating Ethereum [2], which uses Bitcoin has a protocol layer for posting messages. This is similar to LazyLedger in the sense that the blockchain can be used to post arbitrary messages that are interpreted by clients, however in Mastercoin all nodes must download all Mastercoin messages as the Bitcoin base layer does not support efficient data availability schemes such as the Probabilistic Validity Rule. Additionally, as Mastercoin uses Bitcoin as the base layer, it does not support queries for complete sets of messages by specific applications by clients. Finally, Mastercoin has a set of hardcoded applications, and does not support arbitrary applications. In contrast, LazyLedger examines what an ideal new blockchain would look like for use as a base layer in a system where the base layer is only for posting messages and data availability.

9 Conclusion

We have presented LazyLedger, an alternative blockchain design paradigm where the base layer is only used a mechanism to guarantee the availability of on-chain messages, and transactions are interpreted and executed by end-users.

Acknowledgements

Mustafa Al-Bassam is supported by a scholarship from The Alan Turing Institute.

Thanks to George Danezis for helpful discussions about the design of LazyLedger, and Alberto Sonnino for comments.

References

[1] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system," 2008.

[2] V. Buterin, "Ethereum: The ultimate smart contract and decentralized application platform (white paper)," 2013.

[3] M. Castro and B. Liskov, "Practical byzantine fault tolerance," in *Proceedings of the Third Symposium on Operating Systems Design and Implementation*, OSDI '99, (Berkeley, CA, USA), pp. 173–186, USENIX Association, 1999.

[4] K. Croman, C. Decker, I. Eyal, A. E. Gencer, A. Juels, A. Kosba, A. Miller, P. Saxena, E. Shi, E. G. Sirer, *et al.*, "On scaling decentralized blockchains," in *International Conference on Financial Cryptography and Data Security*, pp. 106–125, Springer, 2016.

[5] M. Al-Bassam, A. Sonnino, and V. Buterin, "Fraud proofs: Maximising light client security and scaling blockchains with dishonest majorities," *CoRR*, vol. abs/1809.09044, 2018.

[6] E. Kokoris-Kogias, P. Jovanovic, L. Gasser, N. Gailly, E. Syta, and B. Ford, "OmniLedger: A secure, scale-out, decentralized ledger via sharding," in *Proceedings of IEEE Symposium on Security & Privacy*, IEEE, 2018.

[7] J. Poon and T. Dryja, "The Bitcoin Lightning network: Scalable off-chain instant payments," 2016.

[8] A. Miller, I. Bentov, R. Kumaresan, and P. McCorry, "Sprites: Payment channels that go faster than Lightning," *CoRR*, vol. abs/1702.05812, 2017.

[9] S. Bano, A. Sonnino, M. Al-Bassam, S. Azouvi, P. McCorry, S. Meiklejohn, and G. Danezis, "Consensus in the age of blockchains," *CoRR*, vol. abs/1711.03936, 2017.

[10] W. W. Peterson, W. Wesley, E. Weldon Jr Peterson, E. Weldon, and E. Weldon, *Error-correcting codes*. MIT press, 1972.

[11] S. B. Wicker, *Reed-Solomon Codes and Their Applications*. Piscataway, NJ, USA: IEEE Press, 1994.

[12] E. Heilman, A. Kendler, A. Zohar, and S. Goldberg, "Eclipse attacks on bitcoins peer-to-peer network," in *24th {USENIX} Security Symposium ({USENIX} Security 15)*, pp. 129–144, 2015.

[13] A. Back, M. Corallo, L. Dashjr, M. Friedenbach, G. Maxwell, A. Miller, A. Poelstra, J. Timón, and P. Wuille, "Enabling blockchain innovations with pegged sidechains," 2014.

[14] S. Larson, "Bitcoin split in two, here's what that means," 2017.

[15] V. Buterin, "Hard fork completed," 2016.

[16] S. A. Crosby and D. S. Wallach, "Efficient data structures for tamper-evident logging," in *Proceedings of the 18th Conference on USENIX Security Symposium*, SSYM'09, (Berkeley, CA, USA), pp. 317–334, USENIX Association, 2009.

[17] J. R. Willett, M. Hidskes, D. Johnston, R. Gross, and M. Schneider, "Omni protocol specification," 2012.