Performance and scalability testing for blockchain consensus protocols: an empirical framework

Alked Ejupi[1], Stefano De Angelis[2,*] and Vladimiro Sassone[1]

[1]*University of Southampton, University Rd, SO171BJ Southampton, United Kingdom*
[2]*University of Salerno, Fisciano (SA), 84084, Italy*

Abstract

Blockchain unlocks several use cases leveraging unique characteristics of trustless peer-to-peer transactions and decentralised computing. Reaching consensus in blockchains is crucial to ensure a consistent state of the ledger across the network participants. However, consensus also introduces performance and scalability gaps concerning traditional centralised systems. Understanding those gaps is paramount, especially if blockchains are established as the main computing infrastructure for future digital services.

In this paper, we propose an empirical approach to performance and scalability testing in blockchain systems. We provide a framework that unlocks measurements of well-established metrics under different configuration scenarios. This framework establishes a systematic methodology based on simulated private blockchain networks. To this extent, we first provide a standardisation for performance and scalability metrics. Then, we describe the testing methodology that establishes a reproducible environment for running quantitative tests with efficient, asynchronous, results computation. We implement the framework with two blockchain platforms using different consensus protocols, respectively Proof of Work and Proof of Stake. We demonstrate that the performance of both protocols is impacted by different configuration settings like the difficulty parameter in Proof of Work and the gossip protocol in Proof of Stake.

Keywords

Blockchain, Consensus, Performance testing, Scalability,

1. Introducton

Blockchain technology provides a trustless infrastructure for both transactional, peer-to-peer systems and distributed computing systems. Being decentralised, blockchains do not need to entrust third-party providers to deliver secure and reliable services. Since the advent of Bitcoin [1], blockchains have been considered as an infrastructure for building a decentralised economy [2], disrupting traditional computation and storage systems [3], and decentralising both digital and physical systems [4]. Achieving these use cases requires secure and efficient platforms able to operate under various configuration settings and scenarios. In this context, several blockchains have been proposed in an attempt to surpass the others with better performance while preserving their principle of decentralisation and trust [5].

Providing high-performance decentralised systems is not trivial. In this context, a critical component is the consensus protocol. Consensus ensures that the decentralised parties reach an agreement on a certain state or the next computing operations. However, this comes at the cost of heavy communications and computation that undermine the whole system's performance [6]. Various consensus implementations exist today, each one providing a tradeoff between decentralisation and performance. In particular, it is unclear how blockchain systems can scale to larger networks without suffering performance degradation [7]. Some systems claim to provide outstanding performance without properly demonstrating their claims. This brings confusion in the industry, undermining the technological credibility of several existing platforms. To this extent, an approach to performance and scalability measuring is urgently needed. Although some proposals already exist in literature [6, 8], they lack of a common methodology creating ambiguous results.

6th Distributed Ledger Technology Workshop, May 14–15, 2025, Torino, Italy
*Corresponding author.
ae7g18@soton.ac.uk (A. Ejupi); sdeangelis@unisa.it (S. De Angelis); vsassone@soton.ac.uk (V. Sassone)
0000-0002-1168-9064 (S. De Angelis); 0000-0002-6432-1482 (V. Sassone)
© 2022 Copyright for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

In this paper, we present an empirical framework for testing the performance and scalability of blockchain platforms. The framework is based on a systematic methodology that leverages simulation techniques to execute comprehensive testing scenarios. It implements simulated private blockchains as System Under Test (SUT) running under controlled execution environments. A controlled SUT ensures flexible and reproducible testing scenarios under various system configurations. The methodology is quantitative, providing empirical testing under different workload scenarios, and is asynchronous for the data collection and results generation. This framework has been designed to be optimum, in the sense that provides accurate results without injecting unexpected overheads and is adaptable to any blockchain system. In this work, we implemented and tested the framework with two blockchain platforms, Algorand (Proof of Stake) and Ethereum (Proof of Work). We used the framework to outline the performance and scalability limitations of both protocols when deployed in different configuration scenarios and varying the node parameters.

Contributions. The main contributions of this paper are:

- an empirical framework for testing performance and scalability of blockchain platforms based on a systematic methodology that applies a quantitative measurement approach with varying workloads and systems configurations, and asynchronous data collection;
- an implementation of the above framework with two famous blockchain platforms, namely Algorand and Ethereum, running respectively a Proof of Stake and Proof of Work consensus protocols;
- a performance and scalability evaluation of both implementations with varying workloads and configurations.

2. Related Works

In recent years, blockchain performance and scalability tradeoffs have received particular attention from the scientific community. A comprehensive survey of the existing performance studies has been presented by Caixiang et al. in [6]. The survey outlines state-of-the-art techniques to measure performance with empirical and analytical models, thus presenting existing bottlenecks in blockchain performance and future challenges. In the literature exist several empirical performance evaluations of mainstream blockchains. Zheng et al. [8], propose a real-time performance monitoring framework using a log-based approach. The authors evaluate their framework with Ethereum and Hyperledger Fabric blockchains, however, the study does not consider different workloads and network configurations. Seamlessly, Hao et al., present in [9] a quantitative analysis approach to measure throughput (transactions per second) and latency in Hyperledger Fabric and Ethereum. The study shows how consensus represents a performance bottleneck with varying workloads. More focussed on Hyperledger Fabric, in [10] Sukhwani et al., propose a model based on Stochastic Rewards Nets (SRN) to measure throughput, latency, mean queue lengths, and resources utilisation of the Fabric's "execute-order-validate" architecture. Baliga et al. [11], evaluate the performance features of the GoQuorum blockchain under different configurations and with two different consensus protocols, namely Raft and IBFT. As an approach, this work adapts the Caliper benchmarking tool to enable distributed workloads and measure the transactions throughput and latencies in GoQuorum. In [12], the authors provide an analytical model to estimate transaction confirmation latencies in PoW consensus protocols. The model suggests optimal transaction fees respect mempool state and workloads. Conversely from our empirical evaluation, this work does not consider parameters affecting block production rate and only assumes constant block finalisation times.

Related to benchmarks, several tools exist. A survey proposed by Shäffer et al. provides an overview of the most prominent benchmarks [13]. Other popular tools are BTCMark [14] and Gromit [15]. BTCMark is a framework to assess different blockchains through various application scenarios and different emulated infrastructures. It has been used to evaluate the performance and resource consumption of Ethereum and Hypoerledger Fabric blockchains. Gromit is a tool for systemic testing on specific blockchain components. It has been used to test the validator's performance of several blockchains, including Ethereum, Algorand, and Avalanche. Finally, a broader analysis of performance and scalability has been introduced by Schäffer et al. with [13]. The study considers the variation of performance in Ethereum blockchains varying different configuration parameters. From the study emerged that misconfigured nodes are a major cause of underperforming systems.

The state-of-the-art in blockchain performance testing provides insightful experimentations, analytical models, and tools. However, existing solutions are either tailored to a specific platform or only focus on certain components of the system. In our work, we provide a systematic methodology to define an empirical framework for testing the performance and scalability of any blockchain. The framework defines a standardised approach for testing under different systems' configurations, workload generation and measuring collected data without introducing computing overheads.

3. Blockchain as a System Under Test

In this work, we model a blockchain as a benchmarking *System Under Test (SUT)* [16]. To introduce the SUT, we first provide some basic blockchain concepts that will be used throughout this paper.

We consider a *blockchain* as a distributed ledger replicated across independent nodes of a decentralised network [17]. The nodes run a distributed protocol, namely the *blockchain protocol* to collectively process operations and maintain a consistent state. The *ledger* data structure is a list of records, called *blocks*, cryptographically linked together. Blocks are of fixed dimension, *block-size*, and include information such as the cryptographic hash of its predecessor, a timestamp, and a list of transactions. A *transaction* is an operation processed on the blockchain protocol between two parties. Users send transactions for different tasks, for example exchanging cryptocurrencies like Bitcoin [1], or executing smart contracts like in Ethereum [18]. Users interact with the protocol with cryptographic identities, called *accounts*. Transactions sent over a user requests on the blockchain are said *submitted*. The blockchain protocol associates to each account a *balance* of the native token. Those tokens can be used to manage the governance of the blockchain or as a cryptocurrency. The total supply of native tokens and the initial distribution is usually defined within the genesis block. The *genesis block* is the first block of the blockchain, and it specifies parameters like the *block-size*, the list of participation nodes, and more.

Blockchains rely on *consensus* to achieve a consistent view of the ledger. Consensus determines the rules on which nodes agree to append new blocks on the blockchain and their frequency, i.e. *block-period*. Not all consensus protocols can guarantee the same level of consistency at any point in time. Some protocols allow the creation of forks. A blockchain *fork* happens when the nodes of the network have different views of the ledger. When forks are unlikely to happen for a certain block, that block is said *final* or *finalised*. Thus, we refer to the transactions of a *final* block as *finalised*.

We define a SUT as a private blockchain network composed of a fixed number of nodes running a specific blockchain protocol. A node of a private network can be of two types, namely *network node* or *participation node*. The former handles communication routing between the latter. Both node types execute the blockchain protocol.

4. Performance and Scalability Framework

In this section, we present our framework for testing the performance and scalability of blockchain systems. We first present the testing methodology that the framework implements, and its limitations. Then, we define performance and scalability metrics, and an overview of the framework's architecture and its components.

4.1. Testing Methodology

The methodology defines an empirical approach to performance and scalability testing. As intuition, a test (or experiment)[1] consists of a three-step process: (i) deploy a controlled SUT from a provided configuration file, (ii) spawn a workload of SUT's transactions, and (iii) collect data artefacts from the

[1]From now on, the terms "experiment" and "test" will be used interchangeably.

SUT nodes and compute the test results. This methodology is said “systematic”, in the sense that follows a standardised approach and ensures reproducibility, “quantitative”, allowing empirical performance and scalability analysis under various workloads, and “asynchronous”, avoiding collecting data with inefficient, real-time, monitoring processes that introduce computation overheads [8].

The methodology assumes a SUT deployed under a local testing facility. The effectiveness and comprehensiveness of results depend on the capacity of such a facility. Moreover, being the facility deployed in a single geographical area, the SUT cannot reproduce latencies caused by nodes’ communication delays. We consider an interesting future direction for the evaluation of our methodology under realistic deployment scenarios.

Deploy a controlled SUT. A fresh SUT is deployed. The SUT is said "fresh" starting from a baseline configuration. The configuration takes as input SUT parameters, such as the testing blockchain platform, the underlying consensus protocol, a number N of participation nodes and a number M of network nodes. The SUT is a private blockchain network running in a controlled and reproducible playground environment. With this approach, it is possible to provide various testing options, varying for example the number of network or participation nodes while testing the system’s scalability. Finally, the SUT runs in separate and dedicated hardware. No other processes run on the SUT apart from the SUT itself. The rationale is to maximize the computing and networking capabilities of SUT while avoiding overhead caused by other processes.

Workload Generation. The workload generation simulates batches of transactions, i.e. the *load*, that periodically flood the SUT. The load is equally balanced across the participation nodes. This avoids unrealistic load distribution over single nodes and reproduces a homogeneous distribution of transactions across the network. The workload generator accepts custom configurations for various load settings. The configurable load parameters are the *load duration*, i.e. the continuous period in which a workload constantly generates new transaction requests; the *input rate*, namely the constant transaction delivery rate; the *batch size* and *batch count*, respectively the number of transaction requests in a single batch and the number of batches to spawn throughout the load duration.

Data collection. The data collection step follows an asynchronous approach. Once the workload generation terminates, a script collects the logs artefacts from the participation nodes. Those artefacts contain consensus data about all the processed transactions. Being the data collection asynchronous, it waits for all pending transactions to be processed (and finalised) before collecting the test artefacts. This approach does not leverage real-time performance monitoring or any pooling service to the nodes’ APIs. In this way, no overheads to nodes’ performance are introduced, thus the obtained results remain accurate [8]. The artefacts are parsed and merged into a single dataset. As a result, a 6-tuple dataset gets generated: $\langle node_{id}, tx_h, t^s, t^f, block_n, batch_k \rangle$, where:

- $node_{id}$: is the node identifier (id) in the SUT;
- tx_h: is the unique identifier (hash h) of a submitted transaction processed by the SUT;
- t^s: is the transaction submission timestamp;
- t^f: is the transaction finalisation timestamp, i.e., the timestamp when the block containing the transaction tx_h is considered final;
- $block_n$: is the number of the $n - th$ block on the ledger; $block_0$ is the first finalised block of the ledger (also known as the genesis block);
- $batch_k$: is the $k - th$ transactions *batch* submitted to the system.

4.2. Evaluation Metrics.

The following metrics are defined:

- *Throughput*: is the number of transactions finalised by the SUT within a determined period. Given two times, t_1 and t_2, such that $t_2 > t_1$, we identify with T_{t_1,t_2} the total number of finalised transactions $\{tx_{h1}, ..tx_{hn}\}$ in that period. The throughput is measured as transactions (finalised) per second *TPS* with:

$$TPS = \frac{T_{t_1,t_2}}{t_2 - t_1} \tag{1}$$

  where:
  T_{t_1,t_2}: number of final transactions in period (t_1, t_2);
  (t_1, t_2): transactions finalisation period;
  $t_2 - t_1$: duration (seconds) of the finalisation period.

- *Latency*: it is the average transaction latency measured on all submitted transactions. Transaction latency is the difference between the transaction finalisation time and the transaction submission time:

$$latency = t^f - t^s \tag{2}$$

  where:
  t^f: transaction finalisation time;
  t^s: transaction submission time.

- *Scalability*: measured as the variation of throughput and latency measurements obtained by altering the number of nodes and the workload.

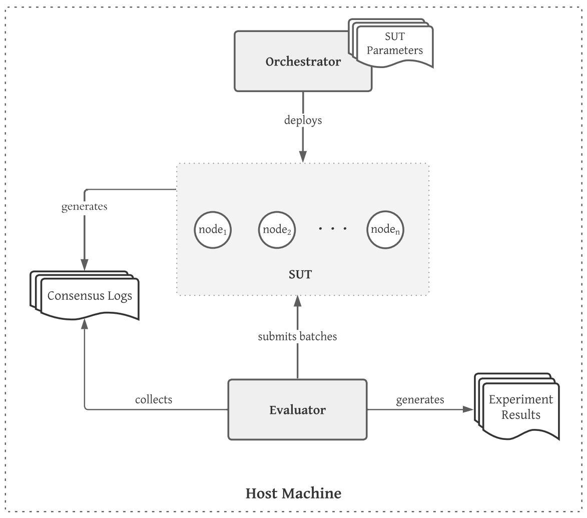


Figure 1: Framework architecture. The arrows indicate the interaction between each component, whereas at the centre there is the SUT.

4.3. Framework Architecture

The framework architecture is shown in figure 1. It is composed of two components called *Orchestrator* and *Evaluator*. The former is responsible for deploying the SUT, while the latter generates the transaction workload and handles data collection. Both components, along with the System Under Test (SUT), operate on the same host machine. They rely on process containerisation to ensure an isolated and consistent execution environment. This approach provides a minimum and reproducible testing system trading-off with resource limitations. The decoupling of the SUT from the Orchestrator and Evaluator is left as future work.

Orchestrator. It deploys the SUT for the experiment. It takes as input the SUT parameters that specify the SUT blockchain platform and consensus, along with the numbers N and M, of participation and network nodes respectively, and an arbitrary array of additional configuration options that can be tuned with the nodes. The Orchestrator takes care of configuring the SUT environment and deploying the network.

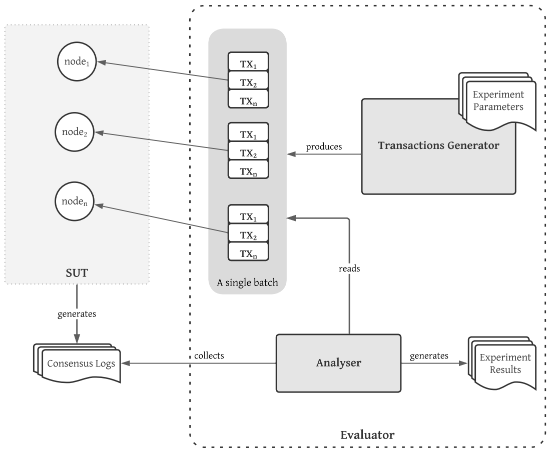


Figure 2: A close-up view on the Evaluator. The Transactions Generator creates a distributed workload, while the Analyser collects and parses the logs and produces the experiment results.

Evaluator. It generates the workload and collects data. Once the experiment is completed, it generates the experiment results by computing the performance and scalability metrics. Figure 2 illustrates a close-up view of the Evaluator. As shown, it embeds two sub-components, the *Transactions Generator* and the *Analyser*. The Transactions Generator accepts an Experiment Parameters file which specifies the workload parameters, such as the load duration, input rate, and batches. For each transaction request, it generates an *TX entity* object that is an HTTP request toward the nodes of the SUT. The Transactions Generator integrates the blockchain platforms' SDKs (Software Development Kit) to build transaction requests according to their specification. TX entities are organised in batches. Given a $batch_k$ and a TX entity TX_i, we say that TX_i is the ith transaction in $batch_k$, with $0 < i < batch\_size$. Transaction entities are off-chain serialised objects. A blockchain node receiving a transaction entity TX_i verifies its validity and appends it to the transactions queue, i.e. *mempool*. Given a transaction entity TX_i, the node responds with a transaction identifier tx_{hi}.

TX entities are load-balanced across the nodes of the SUT. The Analyser monitors the network and collects the artefacts from the nodes containing consensus logs. Finally, it parses the artefacts and returns a data object as a result, which represents the metrics measurement from the generated dataset.

5. Implementation

We implement the framework with a combination of Bash and Python scripts and a Java routine. The SUT is deployed using Docker, as a container-based virtual network, in which each Docker container runs a node of the SUT. Containers embed the blockchain protocol of the supported platforms. We integrate two blockchain platforms, namely Algorand [19] and Ethereum [18]. Both platforms support the execution of smart contracts by integrating a virtual machine.

Ethereum comes with several implementations for its participation nodes and different consensus protocols. In our implementation, we use the Geth [20] Ethereum software to run a PoW network by executing the Ethash [21] protocol[2]. On the other hand, Algorand is used to run a PoS network with its consensus protocol Pure PoS [19]. This choice was driven by the intention of testing the framework with two different consensus protocols, specifically the PoW and PoS.

5.1. Orchestrator

The Orchestrator executes Bash commands that interact with the Algorand and Ethereum CLIs (Command Line Interfaces), respectively `goal` and `geth`. It is characterised by three different shell scripts that control the creation, initialisation, and stopping of a SUT. The scripts are:

- `createNetwork.sh`: it reads from the configuration file the SUT parameters, such as the blockchain platform, consensus, number of participation and network nodes, and additional node options. This script configures the blockchain nodes, the network topology, and the genesis file according to the passed configuration;
- `startNetwork.sh`: it accesses the environment in which the SUT has been created, thus it starts the network by running the startup command on every node;
- `stopNetwork.sh`: it accesses the environment in which the SUT has been created and therefore stops the network. It is responsible for storing the node logs in a repository external to the SUT environment.

Networks Topologies. The network topology is dictated by the number of participants and network nodes. In particular, given N participation nodes, at least one network node is required to manage the communications. The network nodes are called *relay* and *bootnode* respectively by Algorand and Ethereum, whereas participation nodes are called *non-relay* (or participation) in Algorand, and *miner* in Ethereum. Ethereum's bootnodes bootstrap the peer-to-peer communication across the network, conversely on Algorand the participation nodes cannot directly talk.

The `createNetwork.sh` script interacts with a Python standalone program that deals with the creation of the SUT network. This program creates a Docker configuration file, namely the *docker-compose.yml*, that specifies the containerised Docker network. This file is used to start the SUT via the Docker Compose daemon. We configure the Docker Compose to allocate a space of the host's filesystem for every container, i.e., docker *volume*. The Orchestrator uses *volume*s to store the generated artefacts, such as the node and network configuration files, and their logs.

Nodes Configuration. The `createNetwork.sh` script interacts with the SUT to set up the configuration according to the system specification. It includes the methods to generate the configuration data necessary for a node to participate in the consensus protocol, including its existing *account*s, *balance*, and the blockchain *genesis file*s. In particular, the genesis file includes the type of *consensus* protocol, a network identifier called *network id*, and the initial *balance* allocation to accounts. The genesis file represents the initial block of the blockchain and it is the same for every node.

Algorand Configuration. The Algorand CLI `goal` provides commands to deploy an Algorand private network using a template file. The command is `goal network`. It prepares the configuration files and the genesis file of each Algorand node taking as parameters:

- *Network name*: a simple string representing the unique network identifier. We used the parameter N to assign the network name, for instance, a network with $N = 7$ would have the name *7nodes-net*;
- *Total supply*: total supply of ALGOs, i.e., Algorand's native token;

[2]The Ethash protocol is currently used in the Ethereum Classic network

Table 1
`geth` options used to deploy a PoW miner node in Ethereum.

Option	Value	Description
`-mine`	–	Mining enabled
`-miner.threads`	1	Number of CPU threads to use for mining
`-miner.gasprice`	0	Minimum gas price for mining a transaction
`-miner.gaslimit`	12500000	Maximum gas ceiling for mined blocks
`-miner.target`	12500000	Target gas for mined blocks

- *Accounts*: set of accounts to use at the genesis. For each account, we specify the ALGOs distribution, and a boolean status indicating the account's registration for PPoS consensus. Accounts without a balance are not able to join the consensus. We create one consensus account per participation node with equal ALGO allocation;
- *Network topology*: number M of *relay* nodes and N of *non-relay* nodes.

Algorand is characterised by two main processes, namely *algod* and `kmd`. The former is the main Algorand process for handling the blockchain like message exchanges and transactions processing, the latter handles the cryptographic operations for generating wallets and accounts private keys, and for signing transactions. These processes are configured using a JSON file. Algorand *relay* nodes do not host wallets and therefore the kmd is not needed. We implement the `startNetwork.sh` script to start the *algod* processes of each configured node, and the respective *kmd* process for *non-relay* nodes. Similarly, the `stopNetwork.sh` kills those processes and stops the SUT.

Ethereum Configuration. The `createNetwork.sh` takes care of creating and deploying the Ethereum environment. It creates the node configuration files, accounts, and genesis files according to the Geth specifications provided with the software documentation. For each node, a new identifier is created, called *keystore*. The genesis file is created with `puppeth`, an Ethereum command line tool to create and customise Ethereum genesis files. There exist several node parameters that can be tuned in Ethereum. However, the most relevant parameters are the *gasLimit* and *difficulty* [7]. Those impact the Ethereum gas, i.e. the amount of tokens that users have to pay to process Ethereum transactions, and the consensus performance. `puppeth` provides default values for both params. We changed those values to reflect realistic network conditions:

- `gasLimit`: it determines the maximum block size. To obtain realistic measurements, we used the *gasLimit* adopted at the time of writing by the Ethereum main network. Therefore, the default value assigned by `puppeth` was changed from 524,288 to 12,500,000;
- `difficulty`: it determines the block period in a PoW network. It defines the difficulty of the PoW puzzle and the rate at which miners can solve it. We fixed the default `difficulty` generated by `puppeth` to align with Algorand's block period.

Additional Ethereum parameters can be configured with the `createNetwork.sh` script. Table 1 shows the list of supported commands, such as (i) the number of threads used by the node to process transactions, (ii) the transactions fees, called *gasPrice*, (iii) the *gasLimit*, and (iv) the *target* gas usage. We chose those parameters as they can be tuned to obtain different performances [13].

5.2. Evaluator

The Evaluator is implemented as a standalone Java application. To interact with Algorand and Ethereum nodes, the Evaluator imports respectively the AlgoSDK and web3j libraries.

Transactions Generator. It accepts a workload configuration and generates the respective transactions. The cryptographic operations of signing a transaction are delegated to the respective SDKs before sending the workload. In this way, the blockchain nodes do not have to waste performance computing cryptographic operations, as shown in [13].

The Algorand Transaction Generator interacts with the AlgoSDK to create and sign *TX entities* objects using the `kmd` daemon. To guarantee the uniqueness of concurrent transactions, we use the `note` field adding a random identifier (`UUID.randomUUID()` method from the `java.util` package). Similarly to Algorand, Ethereum's transactions need to be distinguished to avoid concurrency errors. We used the `nonce` field of the Ethereum transactions for uniqueness. For each transaction, we compute the `nonce` as $nonce = \gamma + cNonce$, where $cNonce$ indicates the value of the previous `nonce` and γ is the batch size. The workload generation functions as a multi-thread process. For each batch of *TX entity*s, the Transactions Generator initiates a Java thread. We adjust the timeout settings of Java threads to ensure they remain active until all queued transactions are processed by the nodes. We used the `CountDownLatch` from the `java.util.concurrent` package.

Algorithm 1 Pseudocode of Transactions Generator Workload

```
Require: $N$, $\Omega$, $R_n$, $\Delta$, $\beta$, $\tau$, $\gamma$
Ensure: $\mathcal{T}$, the set of all transactions processed by the network.
1: $\mathcal{T} \leftarrow \emptyset$
2: for $\beta$ batches do
3:     $\Pi \leftarrow \emptyset$
4:     for all $node \in N$ do
5:         for $\gamma$ iterations do
6:             $sender \leftarrow \Omega(node)$
7:             $receiver \leftarrow SelectRandomReceiver(R_{node})$ //selects any $r$ from $R_{node}$
8:             $TXentity_\gamma \leftarrow \Delta$ (node, sender, receiver)
9:             $\Pi \leftarrow \Pi \cup \{TXentity_\gamma\}$
10:        end for
11:    end for
12:    for all $TXentity \in \Pi$ do
13:        $tx_h \leftarrow \Phi(TXentity)$
14:        $\mathcal{T} \leftarrow \mathcal{T} \cup \{tx_h\}$
15:    end for
16:    $timeout(\tau)$
17: end for
```

The process of workload generation used with the Transaction Generator is described with the Algorithm 1. The parameters and the functions used in the algorithm are detailed below:

- $N := \{n_1, n_2, \ldots, n_n\}$ is the finite set of participation nodes;
- $A := \{a_1, a_2, \ldots, a_n\}$ is the finite set of accounts;
- $\Omega : N \twoheadrightarrow A$, is a one-to-one function that accepts a participation node and returns its account; given a node n_1, $\Omega(n_1)$ returns the account of a_1;
- β is the number of workload *batch*es;
- γ it the size of a workload *batch*, i.e., the number of *TX entities* in a *batch*;
- $\Pi := \{\pi_1, \pi_2, \ldots, \pi_i\}$ is a finite set of transactions ready to be submitted, i.e. signed *TX entities*. The size of Π depends on γ and the size of the network ($|N|$), thus $|\Pi| = |N| \times \gamma$.
- τ defines *batch*es issuance rate.
- $R_n := \{r_1, r_2, \ldots, r_\gamma\}$ defines the set of *receivers* (accounts) for a given node $n, n \in N$; for a node n_1 and its account a_1 derived from $\Omega(n_1)$, then $R_n = \{A - \{a_1\}\}$;
- $\Delta : (N \times A \times R) \rightarrow \Pi$, is a function executed by each node to create a *TX entity*; it takes as input a 3-tuple $\langle n_1, \Omega(n_1), r_1 \rangle$ (a node, a sender and a receiver) and it returns a signed *TX entity* π_1;
- $\mathcal{T} := \{tx_{h1}, tx_{h2}, \ldots, tx_{hn}\}$: it represents a list of transactions identifiers;
- $\Phi : \Pi \rightarrow TX$, is a function executed by the nodes to process submitted *TX entities*; it takes as input a signed *TX entity* and returns the transaction receipt.

Table 2
Framework dataset

$node_{id}$	tx_h	t^s	$block_n$	$batch_k$	t^f
1	0x54a5aa1862...	1618940343328	23	1	1618939903046
2	0xc6c4a43dd3...	1618940345193	21	1	1618245653328
...	...	...	...	...	...

Analyser. The Analyser is implemented as a single Java process responsible for collecting the logs of the SUT nodes and processing them to generate the dataset. It retrieves the list of *TX entities* and the list of transaction receipts $\mathcal{T}$. Therefore, it starts a new process `waitTransactionsToBeProcessed()`, which waits for the execution of every transaction in $\mathcal{T}$. This method checks the transaction queues, i.e. the *mempools*, and terminates when all the *mempools* are empty. The Analyser iterates over the list of transaction receipts and, for each tx_{hi}, parses the values $node_{id}$, tx_{hi}, t^s, $block_n$, and the batch number $batch_k$.

The finalisation time t^f is then derived from the node logs. It collects the blocks' finalisation timestamps from each node of the SUT and computes the average times to assign a final value to t^f. In Ethereum, the PoW consensus finalisation time is displayed by the node logs with the string similar to `INFO [04-04|10:41:46.601] Block reached canonical chain number=517 ....` We implement the log parser so that it stores the timestamp when the logs show that particular message for the expected block number.

On the other hand, Algorand's PPoS proceeds in rounds and for each round, one block is proposed. PPoS provides instant finality; blocks get finalised when the round terminates. We parse the logs identifying the rounds termination times, i.e., when the log entry `Type=RoundConcluded`. The parser collects N logs and computes the average finalisation time of a block. Finally, we consider the round number the same as the block number as the protocol generates one block per round.

At the end of the parsing phase, the Analyser creates a dataset similar to table 2.

6. Experimental Evaluation

The environment used to deploy and evaluate the framework was composed of a PowerEdge R730xd rack server with 56 logical processors Intel(R) Xeon(R) CPU E5-2695 v3 2.30GHz running the VMware ESXi hypervisor. We ran one virtual machine with *68-Core Intel Core i7 2.6 GHz* with *132 GB 2667 MHz DDR4 RAM* and 1T storage. We deployed the SUT running $N + M$ containers with 2 CPUs and 4GB RAM each. The SUT ran with Docker *v18.09.7* and Docker Compose v1.24.0. We used *Algorand v2.8.0 stable* and the Ethereum *Geth v1.10.6*. Although more recent releases were available, they introduced changes in the consensus protocol that might have affected our evaluation. As a future investigation, we aim to extend the framework with the latest versions to compare changes from the latest and upgraded versions of protocols.

6.1. PoW Difficulty Configuration

We tuned the `difficulty` parameter of the Ethereum PoW. The main purpose was to make Ethereum's experiments comparable with the Algorand block period. We ran an experiment for both platforms deploying a private network of $N = 5$ and $M = 1$ nodes, with a workload of $\beta = 20$ *batches* at the *input rate* of 40 tx/s γ/τ. We compute the average block *block-period* (BP) with the following equation 3:

$$BP_b = finalised_b - finalised_{b-1} \qquad (ms) \tag{3}$$

where:
b: The block number
$finalised_b$: finalisation timestamp of block b.

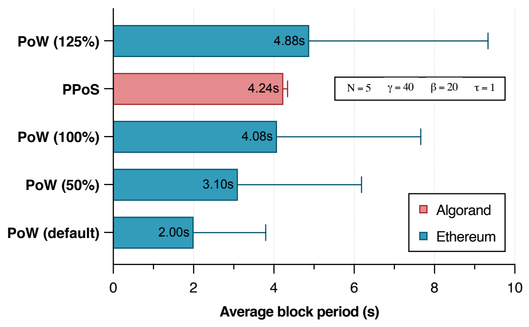


Figure 3: Average block period of Algorand's PPoS and Ethereum's PoW when submitting 100 ($N \times \gamma$) transactions for 40 (β) times every second (τ). Default PoW difficulty is increased by a factor of 50, 100 and 125%

Table 3
Experiment setting for performance evaluation.

Parameter	Value
N	5
γ	100
τ	1
β	120

The first block is equal for every node and it is computed straightforwardly from the genesis file, hence $BP_0 = 0s$. To calculate the average BP, we compute $\overline{BP}$ with equation 4:

$$\overline{BP} = \frac{\sum_{n=1}^{N} BP_n}{N} \qquad (ms) \tag{4}$$

where:
N: quantity of all blocks generated during an experiment.

In normal conditions, Algorand v2.x experienced a $\overline{BP}$ is $\approx 4.5s$ according to recent benchmarks [15, 22]. Conversely, in Ethereum's PoW, the BP value depends on the `difficulty` params set at genesis. We tune the PoW's `difficulty` such that $BP_{PoW} \approx BP_{PPoS}$. The default difficulty set by the Ethereum's command `puppeth` is `0x80000`, i.e., `524288` in decimal format. Figure 3 shows that the experiment measured for Algorand a $\overline{BP} = 4.24s$ as expected. Then, we ran the same experiment for Ethereum. We first tested the platform using the default `difficulty` value and we obtained $\overline{BP} = 2s$. Then we increased it by 50, 100 and 125% to find the closest $\overline{BP}$ to Algorand. The chart shows that increasing the default value by 100% gives us a $\overline{BP} = 4.08s$.

Figure 3 also reveals to us that $\overline{BP}$'s PoW can be variable and not stable as for Algorand, reaching a maximum of 7.6s. This is because the PoW forks are common, and if they occur, the process to confirm a block is delayed. On the other hand, the probability of forking is negligible [19].

6.2. Performance: PoW versus PPoS

We compared PPoS and PoW performance measuring their *throughput* and *latency* over time. We used a fixed input rate and network size. As shown in table 3, we deployed a network of 5 nodes and we tested it with a workload of 500 tx/s for 2 minutes. A SUT with one single network node was used in all tests.

Throughput Evaluation. Figure 4 shows the transactions throughput over time. We measured the throughput by counting the number of transactions finalised in a block b and its BP_b. The y-axis represents the average throughput normalised to a 5 second range. PPoS achieves a throughput of 400 TPS, outperforming PoW which only reached 100 TPS. This result provides evidence that PPoS

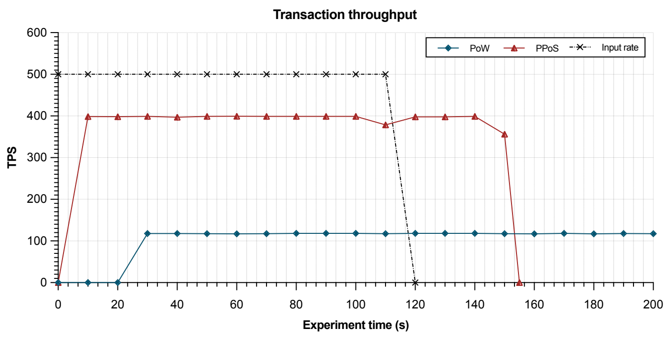


Figure 4: PPoS and PoW TPS 200s experiment

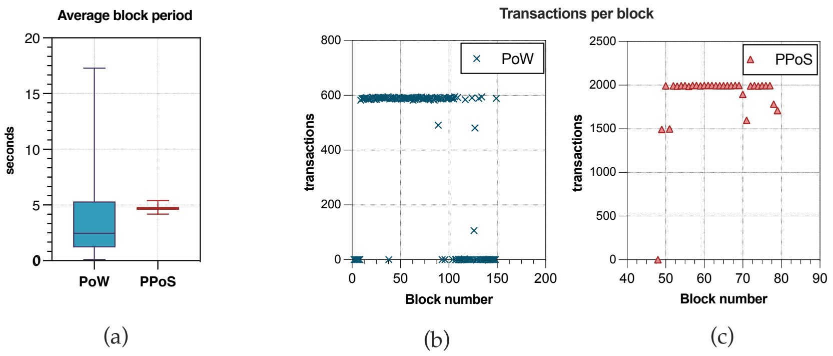


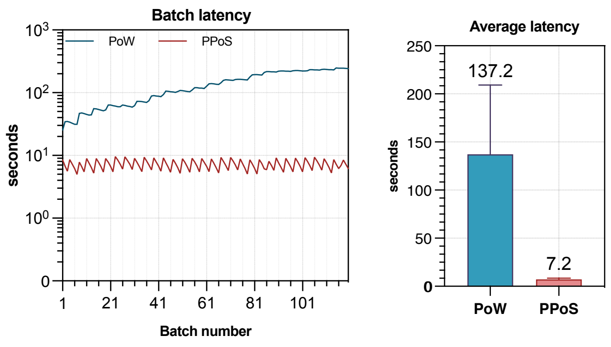
Figure 5: Comparison of PPoS and PoW average block period and number of transactions per block

generates blocks at a constant rate, whereas PoW blocks' generation is variable due to mining and eventual forks. To further understand this throughput rate, we measured the average block period and the number of transactions per block. Figure 5(a) shows that PPoS maintains its block period stable, finalising an average of 1859.5 transactions per block, which means that the input rate is immediately processed and finalised. On the other hand, PoW's blocks were generated with different block periods ranging from less than 1 second up to 18 seconds resulting in even higher peak throughput than PPoS due to low block periods. This suggests that blocks were generated by multiple miners, which increased the probability of forking. Looking at the number of transactions in Figure 5(b), PoW produced empty blocks (with no transactions) between block 75 and block 150, delaying the entire experiment time. This provides evidence that a fork occurred, and miners struggled to synchronise on the longest chain. To prove this claim, we examined the logs of the nodes to find out whether some blocks got refused, i.e., *uncle blocks* [18]. We found that five blocks were classified as *uncle blocks* during the experiment, which means the transactions contained inside were reverted to the transaction mempool. These transactions got finalised in the last 100 seconds of the experiment.

Latency Evaluation. We measured the average transaction latency per batch. Figure 6(a) shows the latency of 120 batches submitted sequentially with a rate of 500 tx/s. In PoW, the latency linearly increased as the batches were delivered, reaching an average latency of 137s, while PPoS presents a constant behaviour with an average transaction latency of 7.2s. Hence, a constant input rate does not impact PPoS throughput and latency, whereas the PoW suffers from unstable throughput that leads to a linear increase in latency over time.

6.3. Scalability: PoW versus PPoS

Input Rate Variation. The following experiment shows how the variation of input rate may affect scalability in both protocols. Figure 4 summarises the experiments we conducted. For each experiment,

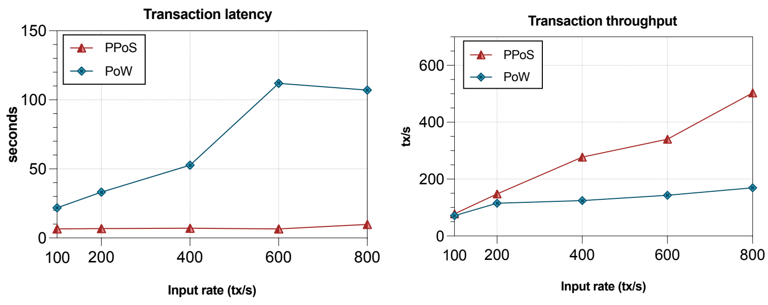


(a) (b)

Figure 6: Comparison of PPoS and PoW transactions latency and averages

Table 4
Experiments setups to measure PPoS and PoW scalability varying input rate and fixed network size

Experiment name	N	γ	β	τ
8 nodes - 100tx/s	8	13	60	1
8 nodes - 200tx/s	8	25	60	1
8 nodes - 400tx/s	8	50	60	1
8 nodes - 600tx/s	8	75	60	1
8 nodes - 800tx/s	8	100	60	1

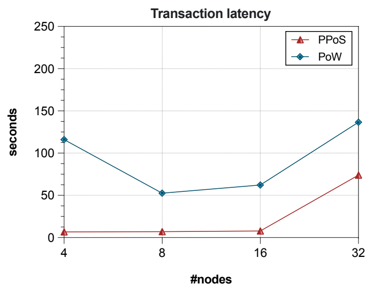


(a) (b)

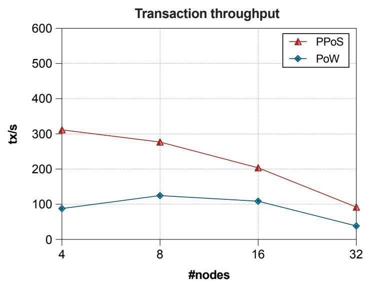
Figure 7: Comparison of PPoS and PoW average throughputs and latencies with input rates ranging from 100tx/s to 800tx/s, and a network size of 8 nodes

we estimate the overall throughput by considering the number of transactions confirmed during the experiment time, that is, the period from the first transaction submitted to the last transactions finalised by the network. The transaction latency is measured by taking the average of all transaction latencies. Figure 7 shows the transaction latency and transaction throughput versus the input rate ranging from 100tx/s to 800tx/s. The graph 7(a) shows that the average transaction latency in PPoS always stays under 10s regardless the input rate. By contrast, increasing the input rate from 400 tx/s to 600tx/s doubled PoW's latency from around 50s to just over 110s. When the input rate from 600tx/s to 800tx/s, PoW's high latency declined steadily, whereas PPoS's latency slightly increased from 6.5s to 9.8s. In comparison, PoW's latency is about 10× higher when delivering 800 tx/s.

Looking at the transaction throughput in 7(b), PPoS achieved the best performance, improving its throughput linearly while maintaining the same latency. Conversely, in PoW, the throughput slightly increased but always remained under 200 TPS for all input rates, whereas PPoS achieved a maximum throughput of 500 TPS with an input rate of 800 tx/s. The lower PoW throughput is caused by the `gasLimit` parameter that we assigned to Ethereum nodes. In our configuration, each block had a

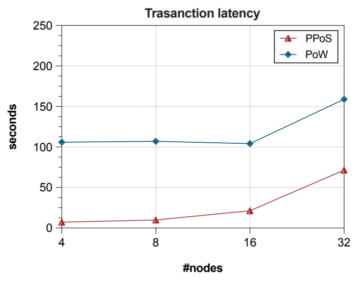


(a)

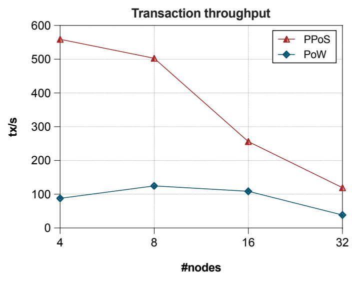


(b)

(a) Input rate 400 tx/s



(a)



(b)

(b) Input rate 800 tx/s

Figure 8: Comparison of PPoS and PoW average throughputs and latencies with input rates of 400 tx/s and 800 tx/s, and network sizes of 4, 8, 16, and 32 nodes

theoretical cap of 595 transactions [3] transactions per block. We observe in figure 7(a) a sharp increase in transaction latency when delivering 600 tx/s, which means that the network must generate two blocks to fit 600 transactions, instead of just one; therefore, this also explains the latency 2× higher drifting the input rates from 400 tx/s to 600 tx/s and 800 tx/s.

Input Rate and Network Size Variation. We varied both the network size and input rate. Figure 8 illustrates a comparison of the protocols under input rates of 400 tx/s and 800 tx/s. We ran four experiments with $N = 4$ up to $N = 32$ per input rate. Overall, the two graphs (8a-(b) and 8b-(b)) show that the horizontal scaling of the system lowered the throughput of both protocols, reaching roughly the same value moving from 16 to 32 nodes. This was caused by the type of network topology used, in which the network node, i.e., *relay* for Algorand and *bootnode* for Ethereum, represented a bottleneck of the blockchain network delaying the propagation of blocks and transactions to all nodes. We leave as future work the evaluation of the same experiment in a more complex topology without centralised network nodes. Looking closely at figure 8a-(a), the transaction latency of both protocols sharply increases when raising the number of nodes from 16 to 32. Figure 8b-(a), however, shows that with a higher input rate, only PoW's latency increased, whereas PPoS's latency remained under 10s. This behaviour was caused by the fact that in PoW more miners generate more blocks simultaneously hence the probability of forks increases. As a result, the PoW had to run more iterations to resolve the forks, introducing additional delays to transaction finalisation. Differently, figure 8a-(a) shows that switching from 16 to 32 nodes, with an input rate of 400tx/s, caused in PPoS a drastic increase in latency. This result was caused by the relay node failing to verify messages. Specifically, these issues occurred during the *Block Proposal* phase where nodes broadcast the blocks using the Algorand gossip protocol [19]. In this phase, only one block is selected. However, in the case of *relay* overloading, some messages may fail and some rounds skipped with zero transactions finalised - no block finalised.

[3]Value obtained by dividing the `gasLimit` by the standard Ethereum transaction's gas value of 21k: $12500000/21000 = 595.2$

7. Conclusion

In this paper, we presented a framework to measure the performance and scalability of blockchain systems. It adopts an empirical approach based on a novel methodology for testing blockchain as a System Under Test (SUT). The methodology is systematic with the SUT deployment environment and testing reproducibility. It allows the deployment of configurable blockchain networks with custom parameters. The framework unlocks accurate quantitative analysis based on ad-hoc workload generation and efficient data collection. The framework has been implemented and tested with two blockchain platforms running different consensus protocols, namely Ethereum (Proof of Work) and Algorand (Proof of Stake). The analysis measured performance under various network dimensions and workloads. It emerged that the configuration parameters of blockchain nodes are crucial when it comes to performance testing. We showed that Algorand's consensus achieves better results than Ethereum's Proof of Work with different loads and network sizes. However, we also demonstrate that, in large networks, the adoption of a communication relayer can introduce performance bottlenecks, and thus it is not preferable in realistic deployment scenarios. As a future direction, we aim to extend the framework to explore how larger networks and more flexible configurations can affect the measurement of results, also considering other blockchain platforms and consensus protocols. Moreover, we plan to decouple the SUT from the Orchestrator and Evaluator components with the aim of minimising computation overheads and bottlenecks in the experiments.

Acknowledgments

This work has been financially supported under the National Recovery and Resilience Plan (NRRP), Mission 4, Component 2, Investment 1.1, Call for tender No. 104 published on 2.2.2022 by the Italian Ministry of University and Research (MUR), funded by the European Union – NextGenerationEU– Project Title "PARTHENON" – CUP D53D23008610006 - Grant Assignment Decree No. 959 adopted on June 30, 2023 by the Italian Ministry of Ministry of University and Research (MUR)

References

[1] S. Nakamoto, Bitcoin: A peer-to-peer electronic cash system, 2008. URL: https://bitcoin.org/bitcoin.pdf.

[2] O. Ali, M. Ally, Clutterbuck, Y. Dwivedi, The state of play of blockchain technology in the financial services sector: A systematic literature review, International Journal of Information Management 54 (2020) 102199. doi:https://doi.org/10.1016/j.ijinfomgt.2020.102199.

[3] N. Zahed Benisi, M. Aminian, B. Javadi, Blockchain-based decentralized storage networks: A survey, Journal of Network and Computer Applications 162 (2020) 102656. doi:https://doi.org/10.1016/j.jnca.2020.102656.

[4] D. Sarkar, Generalised depin protocol: A framework for decentralized physical infrastructure networks, 2023. arXiv:2311.00551.

[5] Y. Xiao, N. Zhang, W. Lou, Y. T. Hou, A survey of distributed consensus protocols for blockchain networks, IEEE Commun. Surv. Tutorials 22 (2020) 1432–1465. doi:10.1109/COMST.2020.2969706.

[6] C. Fan, S. Ghaemi, H. Khazaei, P. Musilek, Performance evaluation of blockchain systems: A systematic survey, IEEE Access 8 (2020) 126927–126950. doi:10.1109/ACCESS.2020.3006078.

[7] M. Schäffer, M. di Angelo, G. Salzer, Performance and scalability of private ethereum blockchains, in: C. Di Ciccio, R. Gabryelczyk, L. García-Bañuelos, T. Hernaus, R. Hull, M. Indihar Štemberger, A. Kő, M. Staples (Eds.), Business Process Management: Blockchain and Central and Eastern Europe Forum, Springer International Publishing, 2019, pp. 103–118.

[8] P. Zheng, Z. Zheng, X. Luo, X. Chen, X. Liu, A detailed and real-time performance monitoring framework for blockchain systems, in: F. Paulisch, J. Bosch (Eds.), Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice, ICSE (SEIP)

2018, Gothenburg, Sweden, May 27 - June 03, 2018, ACM, 2018, pp. 134–143. doi:10.1145/3183519.3183546.

[9] Y. Hao, Y. Li, X. Dong, L. Fang, P. Chen, Performance analysis of consensus algorithm in private blockchain, 2018, pp. 280–285. doi:10.1109/IVS.2018.8500557.

[10] H. Sukhwani, N. Wang, K. S. Trivedi, A. Rindos, Performance modeling of hyperledger fabric (permissioned blockchain network), in: 17th International Symposium on Network Computing and Applications, NCA 2018, Cambridge, Nov 1-3, IEEE, 2018, pp. 1–8.

[11] A. Baliga, I. Subhod, P. Kamat, S. Chatterjee, Performance evaluation of the quorum blockchain platform, CoRR (2018).

[12] I. Malakhov, A. Marin, S. Rossi, Analysis of the confirmation time in proof-of-work blockchains, Future Generation Computer Systems 147 (2023) 275–291. doi:https://doi.org/10.1016/j.future.2023.04.016.

[13] M. Schäffer, M. di Angelo, G. Salzer, Performance and scalability of private ethereum blockchains, in: C. Di Ciccio, R. Gabryelczyk, L. García-Bañuelos, T. Hernaus, R. Hull, M. Indihar Štemberger, A. Kő, M. Staples (Eds.), Business Process Management: Blockchain and Central and Eastern Europe Forum, Springer International Publishing, Cham, 2019, pp. 103–118.

[14] D. Saingre, T. Ledoux, J.-M. Menaud, Bctmark: a framework for benchmarking blockchain technologies, in: 2020 IEEE/ACS 17th International Conference on Computer Systems and Applications (AICCSA), 2020, pp. 1–8. doi:10.1109/AICCSA50499.2020.9316536.

[15] B. Nasrulin, M. De Vos, G. Ishmaev, J. Pouwelse, Gromit: Benchmarking the performance and scalability of blockchain systems, in: IEEE International Conference on Decentralized Applications and Infrastructures (DAPPS), 2022, pp. 56–63. doi:10.1109/DAPPS55202.2022.00015.

[16] R. Almeida, M. Poess, R. Nambiar, I. Patil, M. Vieira, How to advance tpc benchmarks with dependability aspects, in: R. Nambiar, M. Poess (Eds.), Performance Evaluation, Measurement and Characterization of Complex Systems, Springer, Berlin, Heidelberg, 2011, pp. 57–72.

[17] X. Xu, I. Weber, M. Staples, L. Zhu, J. Bosch, L. Bass, C. Pautasso, P. Rimba, A taxonomy of blockchain-based systems for architecture design, in: 2017 IEEE International Conference on Software Architecture (ICSA), 2017, pp. 243–252. doi:10.1109/ICSA.2017.33.

[18] G. Wood, Ethereum: A secure decentralised generalised transaction ledger, Ethereum Project Yellow Paper (2014).

[19] Y. Gilad, R. Hemo, S. Micali, G. Vlachos, N. Zeldovich, Algorand: Scaling byzantine agreements for cryptocurrencies, in: Proceedings of the 26th Symposium on Operating Systems Principles, SOSP17, Association for Computing Machinery, New York, NY, USA, 2017, p. 51–68. doi:10.1145/3132747.3132757.

[20] Ethereum, Ethereum geth client, https://geth.ethereum.org, 2016.

[21] Ethereum, Ethereum mining - ethash, https://ethereum-org-fork.netlify.app/vi/developers/docs/consensus-mechanisms/pow/mining-algorithms/ethash, 2016.

[22] K. Korkmaz, J. Bruneau-Queyreix, S. Ben Mokhtar, L. Réveillère, Alder: Unlocking blockchain performance by multiplexing consensus protocols, in: 2022 IEEE 21st International Symposium on Network Computing and Applications (NCA), volume 21, 2022, pp. 9–18. doi:10.1109/NCA57778.2022.10013556.