Computing
https://doi.org/10.1007/s00607-024-01324-8

SPECIAL ISSUE ARTICLE

Analysis of strategies for scalable transaction creation in blockchains

Ole Delzer[1] · Richard Hobeck[1] · Ingo Weber[2,3] · Dominik Kaaser[4] · Michael Sober[4] · Stefan Schulte[4]

Received: 19 February 2024 / Accepted: 10 July 2024

Abstract

The growing popularity of blockchains highlights the need to improve their scalability. While previous research has focused on scaling transaction processing, the scalability of transaction creation remains unexplored. This issue is particularly important for organizations needing to send large volumes of transactions quickly or continuously. Scaling transaction creation is challenging, especially for blockchain platforms like Ethereum, which require transactions to include a sequence number. This paper proposes four different methods to scale transaction creation. Our experimental evaluation assesses the scalability and latency of these methods, identifying two as feasible for scaling transaction creation. Additionally, we provide an in-depth theoretical analysis of these two methods.

Keywords Blockchain · Distributed ledgers · Ethereum · Scalability · Transactions

✉ Stefan Schulte
stefan.schulte@tuhh.de

Ole Delzer
ole.delzer@campus.tu-berlin.de

Richard Hobeck
richard.hobeck@tu-berlin.de

Ingo Weber
ingo.weber@tum.de

Dominik Kaaser
dominik.kaaser@tuhh.de

Michael Sober
michael.sober@tuhh.de

[1] Technische Universität Berlin, Berlin, Germany

[2] Technical University of Munich, Munich, Germany

[3] Fraunhofer Gesellschaft, Munich, Germany

[4] Christian Doppler Laboratory for Blockchain Technologies for the Internet of Things, Institute for Data Engineering, Hamburg University of Technology, Hamburg, Germany

Published online: 29 July 2024

Mathematics Subject Classification 68M14 · 68Q85

1 Introduction

Since the advent of smart contracts, blockchains have been explored as a foundation for Decentralized Applications (dApps) and multi-party business processes [3]. Transaction throughput is widely discussed as a factor impeding the scalability of blockchain applications [12]. The limited throughput scalability of early blockchain platforms, like Bitcoin, motivated numerous proposals of consensus algorithms and blockchain platforms, including Ripple with 1 500 Transactions per Second (TPS) [1] and the RedBelly Blockchain with 30k TPS [4].

In some blockchain use cases, organizations may need to send a high volume of transactions in a short time frame or continuously. This includes, among others, manufacturers of high-volume products, e.g., in application scenarios like the traceability of food or pharmaceutical products. To register each product with an Individual Identifier (ID) on a blockchain, processing 100 million transactions daily requires a throughput of approximately 1 158 TPS, assuming one transaction per product ID.

However, conventional approaches to creating transactions do not scale easily within a single machine to such throughput rates. It is important to note that while alternative architectures can be designed for individual use cases, our focus here is on the general class of problems where high throughput in transaction *creation* is required. Layer 2 technologies, while often considered as potential scalability solutions, may not be viable if the goal is to scale transaction creation. This limitation underscores the need for alternative approaches. Thus, Thus, Transaction (TX)-creating machines need to be scaled horizontally, i.e., using more (or fewer) machines to create transactions, instead of vertical scaling, i.e., using a machine with more resources. This is non-trivial since blockchains typically require a unique identifier for each transaction to prevent replay attacks and maintain the order of transactions. Many blockchains, e.g., Ethereum, Avalanche, and Polygon, require that each transaction includes a sequence number, the so-called *nonce*, where the combination of the sender account and the sequence number is unique.

A solution for scalable transaction creation needs to consider the following properties: horizontal scalability, throughput, latency, and fairness. Horizontal scalability allows transaction creation to be distributed across multiple machines to create transactions concurrently. A horizontally scalable solution allows adding more machines to the system to increase throughput. The throughput should increase sufficiently, proportionally, and predictably when adding additional machines to justify the extra cost for additional machines and the introduced complexity for enabling horizontal scalability. Further, all transactions should take approximately the same time from creation to inclusion in the blockchain and maintain the same order to ensure fairness. To summarize our work, we investigate the following research questions:

1. Which approaches enable scalable transaction creation for blockchains?
2. How do these approaches perform regarding throughput, latency, and fairness?
3. How do the results align with already existing theoretical concepts?

To answer these questions, we propose four alternative approaches for achieving such scalability, partly by taking advantage of the specifics of the blockchain environment, like employing smart contracts. We implement the approaches and conduct experiments to study and contrast their properties, particularly regarding scalability, latency, and fairness. Summarizing the evaluation results, we find that two approaches scale well, offering high transaction throughput, and—with suitable parameter settings—achieve good fairness regarding the distribution of transaction inclusion latency. In addition, we present a formal model based on "balls-into-bins games" that enables us to perform a theoretical analysis of our approaches. We limit the theoretical discussion of the approaches to the two most feasible approaches, i.e., the ones with the best results.

The remainder of the paper is structured as follows. After providing the necessary background in Sect. 2, we present the four alternative approaches in Sect. 3 and their implementation in Sect. 4. The empirical evaluation is presented in Sect. 5. Section 6 provides our theoretical analysis. The results and related work are discussed in Sect. 7 and Sect. 8, respectively. Finally, Sect. 9 concludes the paper.

2 Blockchain technology

The first practical application of blockchain technology is Bitcoin [16], a cryptocurrency that operates without a trusted third party. For this, Bitcoin uses a blockchain, i.e., a distributed ledger of transactions managed by a peer-to-peer network of nodes to maintain a consistent shared state. Blockchains exhibit many valuable properties, such as transparency, immutability, security, pseudonymity, and decentralization [10], to allow for their application in areas of mutually distrustful parties.

A blockchain stores the state in a data structure of the same name, representing a linked list of blocks connected through hash pointers [16]. A block consists of a block header and a body. The block header includes metadata about the block and consensus-specific data, while the body includes the transactions. A transaction describes specific actions that alter the state of the blockchain, e.g., transferring value or executing smart contracts. A transaction consists of the sender, receiver, amount of cryptocurrency, and other blockchain-specific data, including, e.g., a unique transaction sequence number, which is essential for protecting against replay attacks and maintaining transaction order. In Ethereum and other blockchains, the transaction sequence number is called the nonce and reflects the total number of transactions the sender created and that were successfully included. The nonce also determines the order in which transactions can be processed. Accordingly, for a transaction with the nonce x to be mined, there need to be transactions with the nonces 0 to $x - 1$ from the same sender already included in

the blockchain or the miner's transaction pool. There also must be no other transaction with the nonce x from this sender in the blockchain.

In blockchains, each network node usually maintains its local copy of the ledger, requiring the network to apply a consensus mechanism to decide on the canonical blockchain. The consensus mechanism is essential to maintaining the integrity and security of the blockchain. While there are several consensus mechanisms, the most popular are Proof of Work (PoW) and Proof of Stake (PoS), applied in Bitcoin [16] and Ethereum [23], respectively. In PoW, participants must solve a cryptographic puzzle to add a new block to the blockchain. Participants search for a random number, so the block's hash falls below a particular target value [16]. The target value depends on the network's current difficulty, which dynamically changes to ensure a specific inter-block time. Solving this puzzle requires a lot of computational resources due to the properties of cryptographic hash functions, which only allow for a brute-force search to find the correct random number. In PoS, on the other hand, the blockchain relies on validators staking cryptocurrency as collateral to gain the right to propose new blocks and validate transactions [17]. The blockchain selects validators based on the number of tokens, staking duration, and randomization. A block proposer builds and broadcasts a new block for the validators to verify and add to the blockchain. Additionally, the block proposer receives a reward. Misbehaving validators are subject to punishments that result in losing part or all of the staked collateral. Both PoW and PoS strive for decentralization, making it costly and inefficient for malicious participants to alter the blockchain and incentivize honest behavior.

The second generation of blockchains also provides extensive programmability. These blockchains, e.g., Ethereum [23], enable the creation of so-called smart contracts [18], deterministic programs stored on the blockchain. Executing a smart contract results in every network participant executing the code, which needs to be able to reach the same result. Ethereum, e.g., uses the Ethereum Virtual Machine (EVM), a stack-based quasi-Turing-complete virtual machine, to execute smart contracts to alter the state of the blockchain. For that, developers can use specific programming languages for smart contracts, e.g., Solidity or Vyper, to create a smart contract. A compiler translates the smart contract to byte code, which the EVM executes. For executing specific instructions, i.e., opcodes, Ethereum uses *gas* to measure the computational cost to execute these transactions. A user has to pay transaction fees depending on the complexity of the executed smart contract, the so-called gas costs. Also, gas costs occur when deploying a new smart contract to a blockchain. Other blockchains use similar concepts to prevent the free usage of blockchain resources for computational purposes.

3 Approaches

This section describes our four different approaches to scaling transaction creation horizontally. We describe the general architecture in Sect. 3.1 and explain each approach in detail in Sects. 3.2.

3.1 General architecture

All four approaches are designed to accept application requests, which may be distributed over the available TX-creating machines, e.g., by a load balancer component. These TX-creating machines are part of an off-chain backend that interacts with an on-chain backend. The off-chain backend consists of the actual TX-creating machines and additional components necessary to synchronize generating transaction sequence numbers. The on-chain backend includes the actual smart contract, including the business logic that the TX-creating machines want to invoke. Additionally, the on-chain backend can include other components for authorizing transaction senders, which is important for TX-creating machines that do not share the same blockchain account. In the following paragraphs, we describe the role of each component in more detail:

A **TX-creating machine** is any component that creates transactions to transfer value or invoke smart contracts on a blockchain. A TX-creating machine needs an account on the used blockchain, which tracks a transaction sequence number to prevent replay attacks. Hence, a TX-creating machine assigns each transaction a correct sequence number. Otherwise, the blockchain network does not accept the transaction. TX-creating machines using the same account communicate with a transaction sequencer to assign each transaction a sequence number. The TX-creating machines communicate directly with the blockchain network or through a transaction sequencer as middleware.

A **transaction sequencer** manages the transaction sequence number for TX-creating machines. This component takes over the task of synchronizing the transaction sequence number among multiple TX-creating machines to allow multiple machines to use the same blockchain account for creating transactions. The transaction sequencer is a push- or pull-based component. In our approaches, the middleware and Sequence Number Manager (SNM) take the role of the transaction sequencer. The middleware not only assigns each transaction sent by a TX-creating machine a sequence number but also signs the transaction on behalf of it. Conversely, the SNM enables the TX-creating machines to retrieve sequence numbers or contingents of sequence numbers to create transactions.

The **blockchain network** is a peer-to-peer network of nodes managing a distributed ledger of transactions. The blockchain network follows an account-based model that utilizes transaction sequence numbers to prevent replay attacks and maintain the correct order of transactions. Further, the blockchain network is the execution platform for the smart contracts executing the business logic for dApps.

The **smart contracts** implement the functions for the business logic that the TX-creating machines want to invoke. The blockchain can host additional smart contracts for authorizing different accounts if TX-creating machines do not share the same account, requiring a more sophisticated approach to authorizing transactions than a single account.

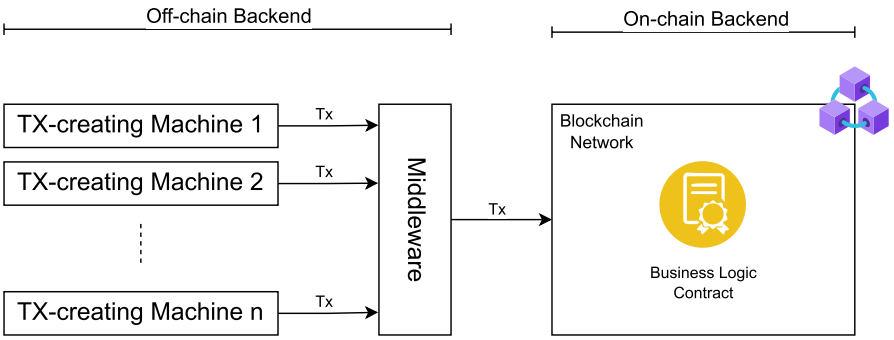

Fig. 1 Approach 1: Externalizing the sequence number management and signing the transaction in a separate, dedicated *middleware* with a single blockchain account

3.2 Approach 1

The first approach is arguably the most straightforward one. As depicted in Fig. 1, the on-chain backend consists of a smart contract. The smart contract implements the business logic, e.g., a registry of product IDs for an organization producing a high volume of goods for which IDs are needed. The off-chain backend includes the TX-creating machines as per above and a *middleware* component between the TX-creating machines and the blockchain system. The middleware receives transaction objects from the TX-creating machines. These transaction objects are missing the sequence number and are still unsigned. The middleware then adds the current sequence number, signs the transaction with its blockchain account, and forwards it to the blockchain network. Note that the transaction can only be signed once the sequence number has been set. This design allows us to keep track of the sequence number locally at the middleware. The middleware fetches the current sequence number of the blockchain account from the blockchain network once at startup and stores it as a local variable. Then, the middleware increments it by one every time it finalizes a new transaction. Since the sequence number is irrelevant when the transaction object is first created, we can scale the TX-creating machines horizontally without concern for the sequence number.

The downside of this approach is that transactions can be created concurrently from multiple machines, but the middleware is a singleton: setting the sequence number and signing the transactions on a single machine. Hence, there is a limit regarding the number of transactions the middleware can process per second, conceivably posing a bottleneck.

3.3 Approach 2

Figure 2 depicts our second approach. As it can be seen, there is no longer a middleware. Instead, the TX-creating machines are responsible for setting the sequence number and forwarding transactions to the blockchain network. We

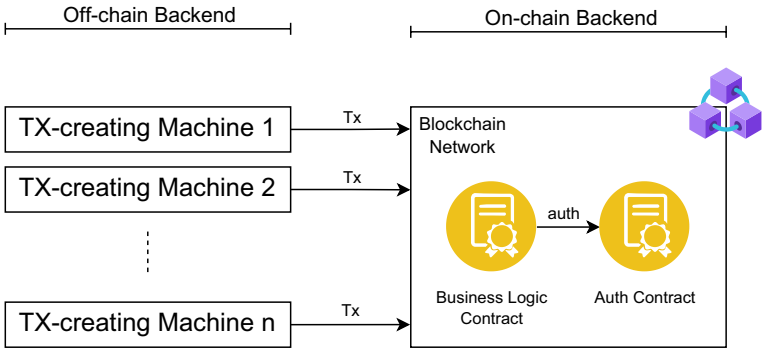

Fig. 2 Approach 2: Allocating an individual account to each TX-creating machine

circumvent the sequence number problem by equipping each TX-creating machine with its own blockchain account to sign transactions. This way, all machines can keep track of their individual sequence number in a local variable. They can create transactions completely independent from each other, which allows for easy scaling without synchronization in the off-chain backend.

However, using multiple blockchain accounts introduces a new problem concerning *authorization*: the smart contract containing the business logic has to be able to decide which invocations to accept (cf. the Embedded Permission pattern [24]). In all other approaches, the TX-creating machines share a single designated blockchain account, so the smart contract simply checks if a transaction originates from this account. We want to dynamically scale our TX-creating machines according to the current workload. Hence, the number of accounts that should be permitted to send transactions to our smart contract cannot be static. The number of authorized accounts must be increased or decreased as needed, and the smart contract must be able to authorize the transaction sender.

We achieve this by introducing a second smart contract, which we refer to as *Auth Contract*. The Auth Contract maintains a list of the addresses of all authorized accounts. It offers a function to check if a specific address is in this list, i.e., belongs to an authorized user account. The list can be updated dynamically with a dedicated *master account*. Upon being invoked, the smart contract containing the business logic calls the Auth Contract to check whether the transaction's sender account is authorized. The smart contract only executes the called function if the transaction sender is authorized. Otherwise, an error is logged, and no other state changes occur.

The disadvantages of this approach are the extra complexity introduced by using a variable number of distinct user accounts and slightly higher transaction fees (for deploying the Auth Contract, updating the variables, and conducting the checks). The list containing the authorized accounts' addresses has to be updated whenever the number of TX-creating machines changes, and the key pairs for all accounts have to be managed well. We must also store the associated private keys in a secure place and create new accounts if the number of machines reaches a new maximum.

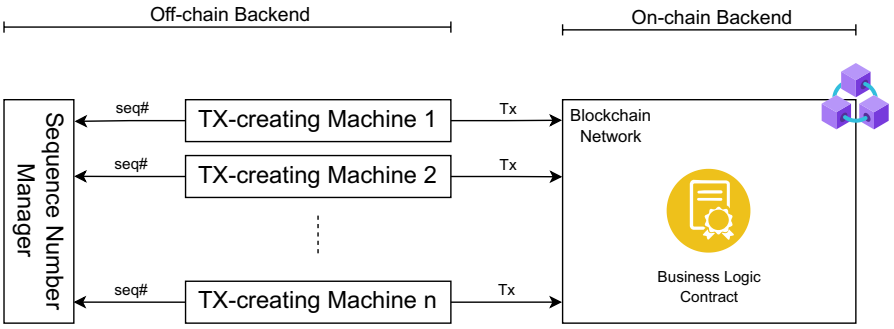

Fig. 3 Approach 3: Outsourcing the sequence number management

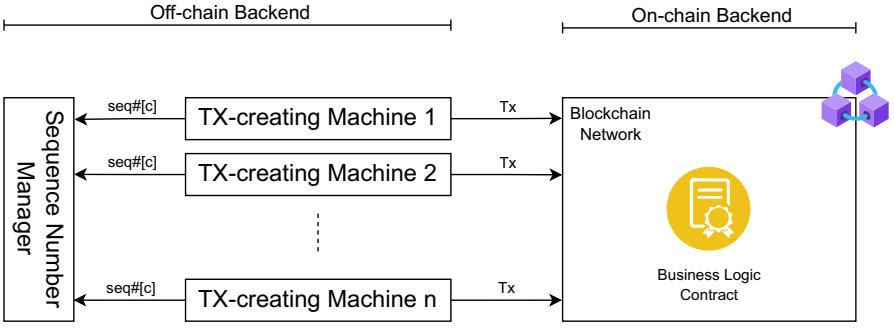

Fig. 4 Approach 4: Assigning *sequence number contingents*

3.4 Approach 3

For Approach 3, to horizontally scale our TX-creating machines, we outsource the management of sequence numbers to a new singleton component called *SNM* (see Fig. 3). As a singleton, only one instance is shared by all TX-creating machines. Therefore, instead of maintaining the current sequence number locally at the TX-creating machine, it is only stored at the SNM. Whenever the TX-creating machines create a new transaction, they request a sequence number from the SNM. The SNM then responds with the current value of the sequence number and increments it by one afterward. The TX-creating machine sets the sequence number to the received value for the new transaction, which is then signed and sent to the blockchain network.

The disadvantage of this approach is that it entails a service invocation every time a transaction is created. This increases the time it takes to create a transaction, decreasing the TPS. The impact depends on the network latency between a TX-creating machine and the SNM and the load of the latter: the SNM is shared by all TX-creating machines and naturally subject to limited bandwidth and computing power. The SNM also poses a single point of failure. However, the SNM

only implements the functionality of a data store containing a single key-value pair to achieve high performance and resiliency.

3.5 Approach 4

As we can see in Fig. 4, Approach 4 is very similar to Approach 3. In fact, Approach 4 is a special case of Approach 3, as will be described in the following. We also externalize sequence number management to a dedicated singleton SNM. But instead of having the TX-creating machines requesting the sequence number from the SNM for every new transaction individually, we use the SNM to allocate *contingents* of sequence numbers to the machines. Therefore, Approach 3 is a special case of Approach 4 with $c = 1$.

If a particular TX-creating machine A requests the next sequence number contingent from the SNM with a contingent size of $c = 100$ and the current sequence number is 1 500, the latter responds with the contingent of sequence numbers 1 500 to 1 599. The SNM then increments the current sequence number by the contingent size of $c = 100$, so the new value is 1 600. A can now create transactions with sequence numbers from 1 500 to 1 599. Afterward, A requests another contingent from the SNM, and the cycle repeats.

This leads to situations where, e.g., another TX-creating machine B creates transactions with sequence numbers 1 600 and above while A has not yet used all sequence numbers between 1 500 and 1 599, i.e., there are sequence numbers lower than 1 600 yet unused. For transaction creation itself, this is not a problem. But once it is sent to the blockchain network, a transaction with a sequence number higher than a yet-unused sequence number has to wait in the miner's transaction pool—until all lower sequence numbers are used in a mined or pending transaction. Hence, larger contingent sizes increase the expected waiting time (latency) of the transactions and lead to a higher variance in the distribution of waiting times. The advantage compared to Approach 3 is that the frequency and volume of requests to the SNM are much lower when requesting whole sequence number contingents instead of single sequence numbers, which allows the SNM to serve more TX-creating machines.

4 Implementation

In this section, we describe the implementations of the approaches described in Sect. 3. The code for the TX-creating machines and the experiments are available on GitHub[1].

[1] https://github.com/OleDe/ethereum-tx-scaling.

4.1 Private Ethereum network

We implement the four approaches on Ethereum due to its widespread adoption and the availability of robust frameworks and tools. It should be noted that the conceptual approaches presented in this paper can also be applied to other blockchain technologies. For instance, the approaches can be used with any other EVM-based blockchain that uses transaction sequence numbers, including BNB chain, Polygon, and Avalanche.

For our evaluation, we use a private Ethereum network consisting of a single *Geth node* acting as a miner for our experiments. Geth[2] is the most used Ethereum client on the public Ethereum network (the *Mainnet*), written in Go.

We opted for Proof of Authority (PoA) consensus in our implementation. Here, only authorized nodes can act as miners, and mining new blocks does not require computation power or "work" [11]. The advantage of PoA for our implementation is that we can simply configure the *inter-block time*, i.e., the time that passes between the mining of two blocks. Since we are foremost interested in the scaling of transaction creation, which happens off-chain, we want to reduce other interfering factors as much as possible. Hence, the stable mining rate of PoA is more suited for our case than PoW or PoS.

4.2 Smart contracts and sequence number manager

The language we used for writing the needed smart contracts is Solidity. Currently, Solidity is the quasi-standard choice for writing smart contracts in Ethereum.

Listing 1 Code Extract from the AuthContract

```
contract AuthContract {
  address public owner;
  mapping(address => bool) public validAccounts;
  constructor() {
    owner = msg.sender;
  }
  function addAccount(address accountAddress) external {
        require(owner == msg.sender, "unauthorized");
        validAccounts[accountAddress] = true;
  }
  function removeAccount(address accountAddress) external {
        require(owner == msg.sender, "unauthorized");
        validAccounts[accountAddress] = false;
  }
  function checkAuthorization(address accountAddress)
  external returns (bool) {
        bool isAuthorized = validAccounts[accountAddress];
        return isAuthorized;
  }
}
```

[2] https://geth.ethereum.org/.

Listing 1 shows a shortened version of our implementation of the *AuthContract*. The AuthContract is used in Approach 2 to support a dedicated Ethereum account per TX-creating machine. It keeps track of the authorized accounts and can be used to check if a certain address is authorized. During construction (lines 4–5), we store the creator of the smart contract in a public variable called *owner* (line 2). This will be our *master account* to update the list of authorized user accounts. The list is maintained by a public variable called *validAccounts*, which maps data of type *address* to data of type *bool*. In addition, we define two functions called *addAccount* (lines 7–10) and *removeAccount* (lines 11-14). Both can only be called by the master account (line 8, line 12) and accept an address as an argument. As one might expect, they set the *validAccounts* mapping for the specified address to true or false, respectively. To allow others to check the authorization status for a specific account, we define another method called *checkAuthorization* that consumes the address of the account to be checked and returns the associated bool value. Since bool is false by default, this method will return false for addresses that have neither been used in the *addAccount* method nor the *removeAccount* method, just like it would for addresses where the *validAccounts* mapping has been explicitly set to false through the *removeAccounts* method.

Listing 2 shows a simplified version of the smart contract that acts as the backend of our dApp and contains the actual business logic. Hence, we simply call it *DappBackend*. To use the AuthContract, we define an abstract contract called *AuthContractProxy* (lines 17–20). AuthContract is an internal variable, i.e., only the contract (and derived contracts) can access it (line 3). The proxy needs to define a method with the same signature to call the actual *checkAuthorization* method of the AuthContract. Since we are not interested in the AuthContract's other methods or variables, we can leave them out of our proxy.

Our DappBackend needs to know the address of the AuthContract to delegate authorization checks, so we implement a method called *setAuthContract* that creates a reference to the AuthContract using the AuthContractProxy and an address argument (lines 7–10). After the reference is established, the business logic can be executed through transactions, represented by an exemplary function called *processTransaction* (lines 11–15). In this method, we can see how the AuthContract's *checkAuthorization* method is called (line 12) and how the execution is aborted or continued depending on the outcome (line 13).

Listing 2 Code Extract from the DappBackend

```
contract DappBackend {
  address public owner;
  AuthContractProxy auth;
  constructor() {
    owner = msg.sender;
  }
  function setAuthContract(address contractAddress) external {
    require(msg.sender == owner, "unauthorized");
        auth = AuthContractProxy(contractAddress);
  }
  function processTransaction() external {
        bool authorized = auth.checkAuthorization(msg.sender);
        require(authorized, "unauthorized");
        ...
  }
}
abstract contract AuthContractProxy {
        function checkAuthorization(address accountAddress)
        external virtual returns (bool);
}
```

For Approaches 3 and 4, the TX-creating machines rely on the *SNM* to keep track of the nonces. We implemented the singleton SNM using Redis,[3] an in-memory database storing the data in key-value format. Hence, the nonce is stored inside our Redis instance as a simple key-value pair.

4.3 Transaction creating machines

The TX-creating machines are Spring applications.[4] Spring is a popular framework for the Java programming language and consists of a whole portfolio of different projects, offering solutions for many common problems. Spring WebFlux was used to create a simple REST API to interact with the TX-creating machines via HTTP. This allows the start and stop of transaction creation or the configuration of certain parameters at runtime. Spring Data includes a Redis client by default, so it is used in the implementations of Approaches 3 and 4 to allow the TX-creating machines to interact with our Redis instance, i.e., the singleton SNM.

For Approaches 2, 3, and 4, the TX-creating machines sign the transactions themselves and directly forward them to the Ethereum node. That is achieved through Web3j.[5] Web3j allows our TX-creating machines to interact with the Geth node in the private Ethereum network. In particular, Web3j lets our TX-creating machines create raw transactions, sign them with our Ethereum account, and send them to the Geth node to call our smart contract.

[3] https://redis.io/.

[4] https://spring.io/.

[5] https://docs.web3j.io/4.8.7/.

Since the TX-creating machines of Approach 1 only create the transaction objects without signing them or forwarding them to the Geth node, they do not use the Web3j library. Instead, we implemented an additional component for Approach 1 called *middleware*. The middleware is also a Spring Boot application and uses the Web3j library. The TX-creating machines send the unfinished transaction objects to the middleware, which adds the nonce to the transaction, signs it, and then sends it to the Geth node. The middleware is a singleton, so no matter how much we horizontally scale the TX-creating machines, there will always be only a single instance of the middleware.

5 Evaluation

In our experimental setup, up to three TX-creating machines are used on individual Virtual Machines (VMs). The private Ethereum network, consisting of a single Geth node, was operated on another, separate VM. To allow high transaction throughput, the node was operated with PoA, at 400 M gas per block and an inter-block time of 5 s, resulting theoretically in up to 3 800 TPS. This proved sufficient for all our experiments, i.e., the blockchain was not the bottleneck. There also was a separate VM for the middleware in Approach 1 and for the SNM in Approaches 3 and 4, respectively. We use Microsoft Azure as the cloud computing platform. All VMs were of the size *Standard_A1_v2*, which comprises a single vCore,[6] 2 GiB of RAM, a download bandwidth of 1 500 Mbit/s and an upload bandwidth of 250 Mbit/s. The latency between the VMs hosting the TX-creating machines and the VM hosting the Geth node was approximately 2.2 ms on average.

To quickly deploy our implementations to the VMs, we use Docker.[7] Docker allows us to build images containing our application, which can be executed in isolated environments called Docker containers. Docker also helps to orchestrate these containers across the VMs so that we can quickly scale our TX-creating machines, i.e., increasing or decreasing the number of instances as needed.

We configured the mining node to mine a new block every 5 seconds. For Approach 1, the total number of TX-creating machines impacts their performance, so we tested Approach 1 with three configurations, i.e., one to three machines m. Approach 4 was also tested with different configurations, at first with a nonce contingent size of $c = 100$, then $c = 1\,000$, and lastly, $c = 10\,000$. Simulations for Approaches 2 and 3 did not have different configurations. The load was simulated to exert the transaction creation throughput continuously.

[6] Intel Xeon Platinum 8272CL CPU @ 2.60 GHz.

[7] https://www.docker.com/.

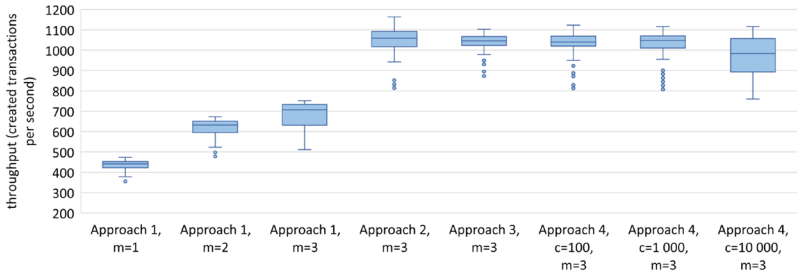

Fig. 5 Comparison of the total transaction throughput

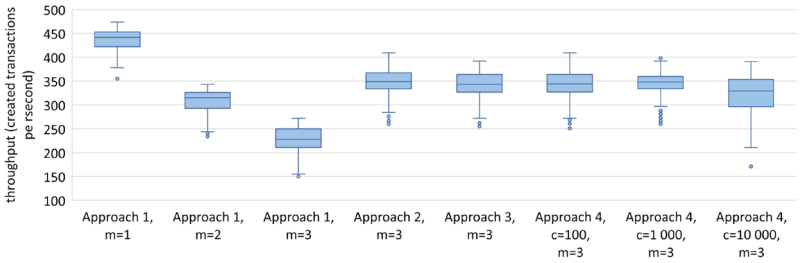

Fig. 6 Comparison of the transaction throughput per machine

5.1 Transaction throughput

In this section, we discuss the evaluation of the transaction throughput of our approaches to scale TX-creating machines. In this case, throughput refers to how many transactions are created per second, i.e., TPS.

Figure 5 compares the total transaction throughput, i.e., the transaction throughput of all machines combined, between the four approaches. We can see that Approach 1 performed the worst. Even with $m = 3$ TX-creating machines running simultaneously, this approach yielded a median of only about 680 TPS. Also, the increase in throughput is sub-linear with a growing number of machines. In contrast, all the other approaches achieved significantly higher throughput rates, with medians ranging from 980 to 1 045 TPS. It appears that the middleware used in Approach 1 already poses a severe bottleneck when we only use a small number of machines. We observed the highest throughput for Approach 2 with 1 045 TPS (median). That corresponds to our expectations because the TX-creating machines in Approach 2 can locally keep track of the current nonce. In contrast, Approaches 3 and 4 must regularly make external requests to the SNM. Still, the difference in performance between Approaches 2, 3, and 4 is only marginal. Notably, the median transaction throughput for Approach 3 with 1 030 TPS was only slightly smaller than that of Approach 2, even though the TX-creating machines had to make a request to the SNM for every single transaction they created. This may be due to the reason that the SNM does not execute resource-intensive tasks since each TX-creating machine still signs its transactions. Figure 5 and 6 show a similar spread of the

distributions for all settings, with two exceptions: Approach 1, $m = 3$ and Approach 4, $c = 10\,000$. The former is only suboptimal, while the latter is interesting for further discussions (see Sect. 5.2).

Figure 6 shows the transaction throughputs per machine for the different approaches and configurations. Here, we can see even more clearly that the middleware in Approach 1 becomes a bottleneck when we try to scale transaction creation horizontally. When only a single TX-creating machine was running, Approach 1 achieved a median throughput of 435 TPS, the highest observed throughput across all machines and configurations. The reason for this is that the machines in Approach 1 do not have to sign the transactions but instead only create and forward "unfinished" transactions to the middleware. For all other approaches, the machines are responsible for signing the transactions, leading to lower performance in the individual machine. However, once we consider $m = 2$ TX-creating machines, the throughput per machine of Approach 1 drops to a median of 307 TPS. At $m = 3$ machines, Approach 1 only offers a median throughput per machine of 230 TPS, which is significantly lower compared to the other approaches, whose median throughput rates per machine range from 323 TPS to 348 TPS.

Summarized, Approaches 2, 3, and 4 offer adequate throughput rates that increase proportionally when adding more TX-creating machines. For Approach 1, on the other hand, the middleware poses a bottleneck when scaling horizontally, so its throughput does not increase proportionally when the number of machines is increased. For all four approaches, the throughput was mostly stable, i.e., the variance of the throughput over time was low. In addition, the load was distributed evenly across all active machines; no single machine had significantly higher/lower throughput than others during the same run.

5.2 Latency and waiting periods

In this section, we examine the transactions' latency, measured as *waiting periods*. With the term *waiting period* we describe the time it takes from a transaction's creation at the TX-creating machine to it being included in the Ethereum blockchain. Note that we do not use the term *commit time* from the literature [21] since that is measured from transaction announcement to the network, which is less suitable for our purposes.

Figure 7 shows the distribution of waiting periods for Approach 2; observations for Approaches 1 and 3 were almost identical, hence the following also applies to them. Especially, for all of these three approaches, we could identify the "dip" for the 5-second-mark (see discussion below), and the frequencies increase and decrease at very similar waiting period times (x-axis). Compared to Approaches 1–3, Approach 4 incurs additional waiting times since the miner needs to wait until a specific TX-creating machine used its whole contingent before including transactions from other TX-creating machines with higher sequence numbers.

The shortest (longest) observed waiting periods were approximately 1.8 (7) s, respectively. In between this interval, the waiting periods follow roughly a uniform

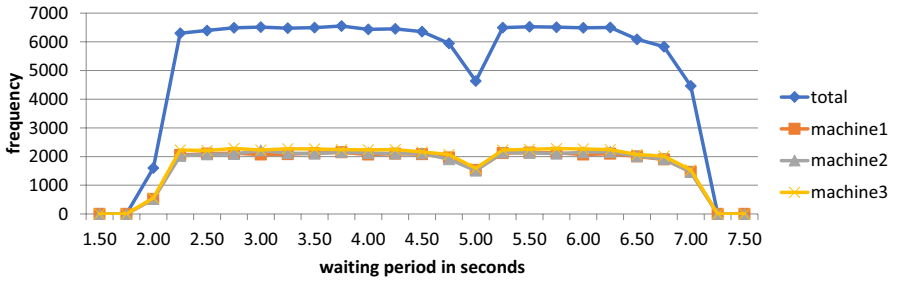

Fig. 7 Approach 2: Aggregated waiting periods with bin sizes of 0.25 seconds

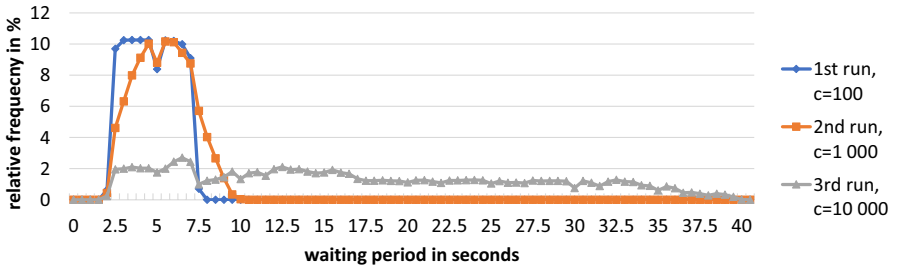

Fig. 8 Approach 4: Aggregated waiting periods with bin sizes of 0.5 seconds for different contingent sizes

distribution. This was expected because we continuously create new transactions while creating new blocks, including transactions in the blockchain every 5 s. Hence, the interval length corresponds to the configured inter-block time of 5 s. On average, it took 4.4 s from a transaction's creation to its inclusion in the blockchain.

The frequency of observed waiting periods noticeably drops at the 5-second-mark, as can be seen in Fig. 7. Through further testing, we could confirm that this drop depends on the configured inter-block time, i.e., when changing the inter-block time to a specific value, the drop could then be observed around that new inter-block time. Given that we are not focused on the behavior of Ethereum clients, we did not investigate this particularity further.

Figure 8 compares the distribution of waiting times of Approach 4 for contingent sizes of $c = 100$, $c = 1\,000$, and $c = 10\,000$. In general, the observed waiting periods were lower when smaller contingent sizes were used because assigning nonce contingents leads to situations where transactions with higher nonces have already reached the transaction pool of our Ethereum node, while some transactions with smaller nonces have not yet been created. The transactions with the higher nonces then have to wait in the queue until all transactions with lower nonces have been created and sent to the Ethereum node. For larger nonce contingents, this effect intensifies, and the average length of the waiting period increases.

For a contingent size of $c = 100$, the distribution of waiting periods is similar to that of Approaches 1, 2, and 3. The observed waiting periods range from 1.8 to

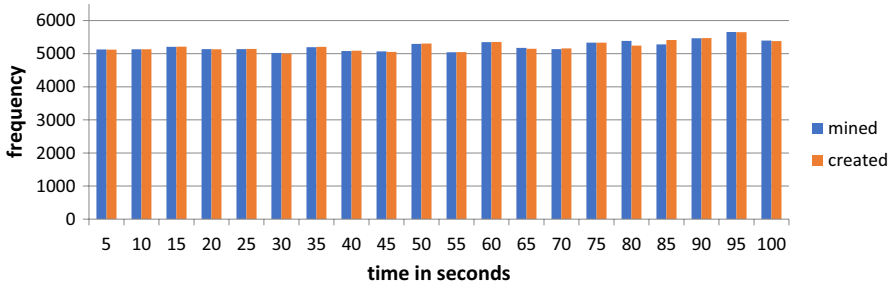

Fig. 9 Approach 2: Comparison between the number of created transactions and the number of transactions included in the blockchain over intervals of 5 s

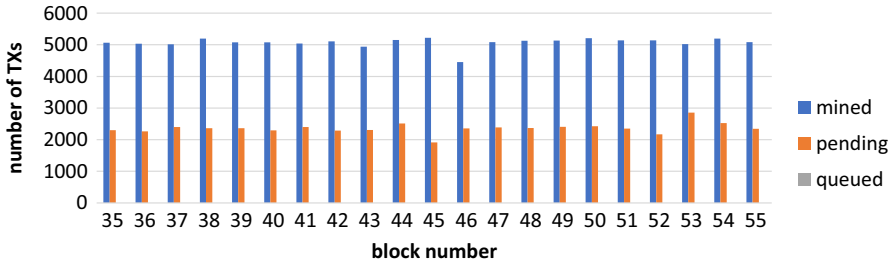

Fig. 10 Approach 2: Transaction count per block compared to the number of pending and queued transactions right after the block was mined

7.2 s. Within this interval, they approximately follow a uniform distribution, except for the drop in frequency around the 5-second-mark, which we also observed in the other approaches. When the contingent size is increased to $c = 1\,000$, the average length of the waiting periods slightly rises. However, we observe a drastic upsurge for contingent sizes of $c = 10\,000$, with transactions waiting up to 40 s from their creation to being included. With a transaction throughput of about 330 TPS on average, the TX-creating machines are fast enough to completely deplete a whole contingent for $c = 100$ and $c = 1\,000$ within the 5 s inter-block time. But for $c = 10\,000$, a machine needs about 30 s until it has used all nonces of a contingent. Accordingly, for larger contingent sizes, the inter-block time is less and less the determining factor for a transaction's estimated waiting period, but instead, the average time it takes for all transactions with lower nonces also to reach the miner becomes the decisive factor. This shows the limitations of contingency sizes, i.e., they should not become too large.

Nevertheless, as long as the contingent size is not configured to be overly large, Approach 4 offers comparable latency/waiting periods to the other approaches. Across all approaches, waiting periods were low and independent from the TX-creating machine from which a transaction originated.

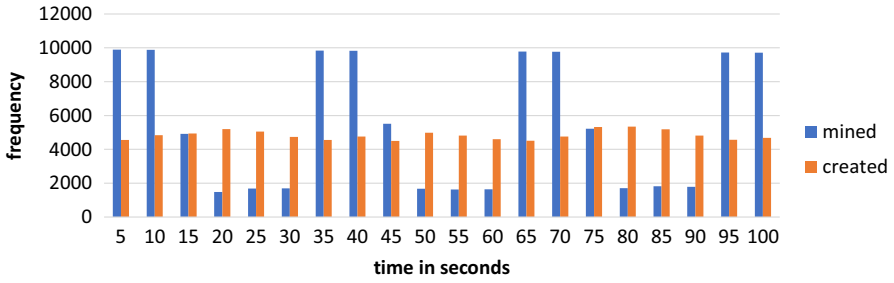

Fig. 11 Approach 4, $c = 10\,000$: Comparison between the number of created transactions and the number of transactions included in the blockchain over intervals of 5 s

5.3 Performance of the mining node

In this section, we present additional data regarding the inclusion of transactions in the blockchain and the status of our miner in general. Like in Sect. 5.2, we discuss results for the first three approaches and then regard Approach 4 separately. Again, Approaches 1, 2, and 3 behaved very similarly. Hence, Figs. 9 and 10 are representative of Approaches 1 and 3 despite being based on data from Approach 2. As already discussed in Sect. 5.2, this difference is because Approach 4 leads to transactions staying longer in the mempool until other TX-creating machines with lower sequence numbers have used up their contingents. However, this is not the case for Approaches 1–3, which explains why they behave very similarly.

Figure 9 depicts a comparison between the number of created transactions and the number of transactions included in the blockchain over intervals of 5 s, respectively. Again, we chose 5 s as the interval size because it corresponds to the inter-block time. We can see that both the mining and creation rates are quite stable, with no noteworthy fluctuations. In addition, the mining rate is approximately identical to the creation rate, so the miner was able to constantly keep up with the TX-creating machines. That is also visible in Fig. 10, where we can see the transaction count per block compared to the size of the transaction pool (pending and queued transactions) right after the block was mined. The observed number of pending transactions did not increase over time but almost stayed constant, so the transactions were included in the blockchain just as fast as they were created. The number of queued transactions was always zero. This is expected since, for Approaches 1-3, the transactions are always created in sequence according to their nonces. Therefore, we observed no instance of a transaction with a higher nonce being in the transaction pool before a transaction with a lower one at the points of observation.

The same does *not* apply for Approach 4, especially for larger contingent sizes, as can be seen in Fig. 11. While the transaction creation rate is fairly stable, the contingent size of $c = 10\,000$ led to heavy, periodic fluctuations in the rate at which transactions are included. Here, we observed a standard deviation of $\sigma = 3\,800$.

The reason for these fluctuations can be seen in Fig. 12: when using larger contingent sizes c, transactions with higher nonces remain in the transaction pool longer, in the *queued* status. They have reached the Ethereum node but cannot be included

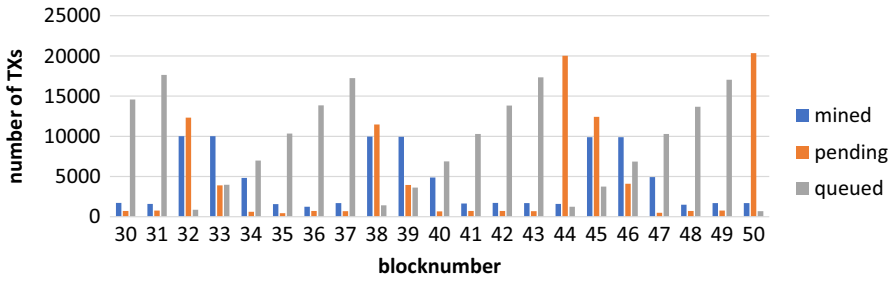

Fig. 12 Approach 4, $c = 10\,000$: Transaction count per block compared to the number of pending and queued transactions right after the block was mined

(or "mined") because not all transactions with lower nonces have yet been created and sent to the Ethereum node. The maximum possible number of these temporarily "missing" transactions is higher for larger contingent sizes c. More specifically, with $m = 3$ TX-creating machines running at roughly the same speed, the maximum is $2c$. The higher the value of c, the longer it takes the TX-creating machines to create all the missing transactions, and the longer the queued transactions with the higher nonces have to wait, and their number is building up. Eventually, when all missing transactions have been created and sent to the Ethereum node, the queued transactions can be included in the blockchain. Therefore, they are simultaneously shifted to the *pending* status. They are then included in the next block (or in the next multiple blocks if their combined amount of gas is higher than the gas limit for a single block) to be mined and become a part of the blockchain. As a result, the mining rates periodically fluctuate above and below the creation rates in Fig. 11. The theoretical maximum of simultaneously queued transactions is $2c$, assuming we have $m = 3$ TX-creating machines with the same throughput.

6 Theoretical analysis

This section presents a theoretical analysis of Approach 3 and Approach 4. The reason why we focus only on these two approaches is the following: Regarding Approach 1, from a theoretical point of view, the off-chain backend with multiple machines submitting transactions via a middleware cannot be distinguished from a single large TX-creating machine. Therefore, the additional insights from a theoretical analysis are limited. Regarding Approach 2, the additional overhead and the higher gas usage introduced by the Auth Contract appear to be a major drawback. Nevertheless, from a theoretical point of view, Approach 2 appears to be a special case of our analysis of Approach 3. Indeed, the additional overhead leads to much larger constant factors, which are hidden in our asymptotic analysis. Since the connection between the actual process and Approach 3 is much cleaner, we therefore focus in the following on the analysis of Approach 3 (see Sect. 6.1) and an extension that covers Approach 4 (see Sect. 6.2).

6.1 Analysis of approach 3: balls into bins

This approach's major drawbacks include a single point of failure and an additional service call for each transaction. Nevertheless, we still present a detailed analysis of this approach due to the following observation: The simple structure of the service call to the SNM allows us to model the interactions via a simple and well-known mathematical model.

From a mathematical point of view, allocation processes are commonly modeled by the classical *balls-into-bins* game. These purely theoretical models are extensively studied in theoretical computer science, probability theory, and combinatorics [14]. They are commonly used to model load balancing and resource allocation tasks. In addition, they have many applications in hashing and networking, see, e.g., [22]. In the context of load balancing in a large parallel computation, the balls typically correspond to jobs or tasks, and the bins correspond to servers. A balls-into-bins game then describes the allocation of balls to a fixed number of bins to distribute the balls to the bins as evenly as possible.

The basic mathematical model of balls-into-bins games is as follows. We are given n identical and indistinguishable balls and m bins that are initially empty. Each ball is "thrown" independently and randomly into one of the bins. More formally, this means that each ball is assigned to one of the m bins chosen independently and uniformly at random. The objective is to analyze the distribution of balls among the bins to derive bounds on the performance of the corresponding load balancing process. Indeed, if we assume that each ball corresponds to a job with the same amount of work, the maximum number of balls in any bin gives a bound on the make span of the distributed computation.

Interestingly, this model can be applied in the context of Approach 3 to facilitate a mathematical analysis. Each TX-creating machine in this approach contacts the SNM. The singleton SNM accepts incoming requests and replies with an updated sequence number. The main question in this context is whether the machines creating transactions are blocked for a significant amount of time, reducing the approach's overall performance. To answer this question, we use a model similar to a balls-into-bins-based load balancing approach presented in [2].

In our model, we assume that at the singleton SNM, we have n threads accepting connections. These threads correspond to our bins. Time progresses in discrete time steps (e.g., seconds) of a fixed time period length. In each time step, λn new transactions are generated, where λ is a parameter of our model. For each transaction, a request is sent to the SNM. These requests correspond to our balls. For our theoretical analysis, we assume that the requests are randomly assigned to a thread chosen independently and uniformly. We assume that every thread accepts exactly one request in each time period. The remaining requests stay in the queue $\mathcal{Q}$. In the following, we prove that, by choosing a suitable parameter

λ, the system remains *stable* in the following sense: In a stable system that is initially empty, the number of requests that are currently unanswered is bounded at any arbitrary point of time.

The main implication of stability is quite strong. Indeed, if we let the system run for an indefinite number of time steps, our analysis will show that at an arbitrary point of time, the system is not overloaded *with high probability (w.h.p.)*, that is, a probability of at least $1 - n^{-\Omega(1)}$. Our first task is to prove the following lemma for the process formally defined in Algorithm 1. The following analysis is a simplified version of the proof of the first case analyzed in [2].

Algorithm 1 balls-into-bins process

```
for each time step t do
    generate λn balls and add them to the queue Q
    each ball in Q selects a bin independently and uniformly at random
    Let S_i be the set of balls that have selected bin i.
    for each bin i with S_i ≥ 1 do          ▷ each bin processes one incoming request
        select a ball uniformly at random from S_i and remove it from Q
```

Theorem 1 *Consider the balls-into-bins process defined in Algorithm* 1. *Let t be an arbitrary time step, assuming that* $0 \leq \lambda \leq 1 - 1/n$. *Then it holds w.h.p. for the size of the queue* $\mathcal{Q}$ *at time t that*

$$|\mathcal{Q}(t)| \leq \mathrm{O}\Big(n + \log\Big(\frac{1}{1-\lambda}\Big) \cdot n\Big).$$

Proof Let $q(t) = |\mathcal{Q}(t)|$ be the size of the queue $\mathcal{Q}$ at time t and let $q^* = (1 + \ln(2)) \cdot n + \ln\left(\frac{1}{1-\lambda}\right) \cdot n$. We first define for a time step t the event

$$\mathcal{A}_t = \{\, q(t) \geq 2 \cdot q^* \,\}.$$

Intuitively, the event $\mathcal{A}_t$ means that at time t, the queue size $q(t)$ exceeds $2 \cdot q^*$. Our goal is to show that $\mathcal{A}_t$ is a low probability event for any time step t. To do so, we will use the following events $\mathcal{B}_{r,t}$.

Let $r \in \mathbb{N}_0$ be a non-negative integer. We define the events

$$\mathcal{B}_{r,t} = \{\, q(t-r) \leq q^* \text{ and } q(i) > q^* \text{ for } t - r < i \leq t \,\}.$$

Intuitively, the event $\mathcal{B}_{s,t}$ means that at time $t - r$ the queue size was below q^*, starting with time $t - r + 1$ the queue size $q(t)$ exceeds q^* and it does not drop below q^* for the entire time interval $[t - r + 1; t]$ of length r. We observe that for each event

$\mathcal{A}_t$ there is exactly one interval length r such that $\mathcal{B}_{r,t}$ occurs. Similarly to [2], this allows us to partition the probability space such that we get

$$\mathcal{A}_t = \biguplus_{r \in \mathbb{N}_{N0}} \mathcal{A}_t \cap \mathcal{B}_{r,t}.$$

We will use the following technical claim in the remainder of this proof.

Claim 1 *For any $r \in \mathbb{N}$ we have* $\Pr\left[\mathcal{A}_t \cap \mathcal{B}_{r,t}\right] \leq 2^{-q^*} \cdot 2^{-(1-\lambda)\cdot n \cdot r}$.

Using Claim 1 allows us to compute

$$\begin{aligned}\Pr\left[\mathcal{A}_t\right] &= \sum_{r \in \mathbb{N}_0} \Pr\left[\mathcal{A}_t \cap \mathcal{B}_{r,t}\right] \\ &\leq \sum_{r \in \mathbb{N}_0} 2^{-q^*} \cdot 2^{-(1-\lambda)\cdot n \cdot r} \\ &\leq 2^{-q^*} \cdot \sum_{r \in \mathbb{N}_0} \left(2^{-1}\right)^r \leq 2^{-q^*},\end{aligned}$$

where we use $\lambda \leq 1 - 1/n$ and the bound on the geometric series. It remains to prove Claim 1.

Proof of Claim 1 In each time step, λn balls are generated according to the definition of the process. Therefore, in the time interval $[t - r + 1; t]$, exactly $r \cdot \lambda n$ balls are generated. The number of balls allocated to each bin in time step i follows a multinomial distribution $\text{Mult}(q(i); p_1, \ldots, p_n)$ with $p_j = 1/n$ for $1 \leq j \leq n$. Let $X_{i,j}$ with $1 \leq j \leq n$ be an indicator random variable that is 1 if no ball is allocated in bin j and 0 otherwise. Note that these indicator random variables $X_{i,j}$ are stochastically dominated by the same indicator random variables $\hat{X}_{i,j}$ for a multinomial distribution $\text{Mult}(\min\{q^*, q(i)\}; p_1, \ldots, p_n)$. Intuitively, this means that our process has several failed allocations that is at most as large as in the same process where at least q^* many balls are thrown in each time step. Due to our choice of q^* we get

$$\Pr\left[X_{i,j}\right] \leq \left(1 - \frac{1}{n}\right)^{\min\{q(i), q^*\}} \leq \left(1 - \frac{1}{n}\right)^{q^*} \leq e^{-\frac{q^*}{n}} = \frac{1-\lambda}{2e}.$$

Then, the total number of failed allocations is

$$Z_{r,t} = \sum_{i=t-r+1}^{t} \sum_{j=1}^{n} X_{i,j} \quad \text{with expected value} \quad \mathrm{E}\left[Z_{r,t}\right] \leq n \cdot r \cdot \frac{(1-\lambda)}{2e}.$$

Since the number of allocated balls in each bin follows a multinomial distribution, the indicator random variables $X_{i,j}$ for the empty bins are negatively associated [8]. This allows us to apply Chernoff bounds to $Z_{r,t}$. Since $q^* + (1-\lambda) \cdot n \cdot r \geq 2e \cdot \mathrm{E}\left[Z_{r,t}\right]$ we get from Lemma 2 (see Appendix A) that

$$\Pr\left[\mathcal{A}_t \cap \mathcal{B}_{t-r,t}\right] \leq \Pr\left[Z_{r,t} \geq q^* + (1-\lambda)\cdot n \cdot r\right] \leq 2^{-q^*} \cdot 2^{-(1-\lambda)\cdot n \cdot r}.$$

□

This concludes the proof of Theorem 1. □

Our analysis shows that the process is stable such that the total number of balls awaiting allocation remains bounded at an arbitrary point of time w.h.p. Unfortunately, from a practical point of view, the overhead of the additional service call can become substantial. In the next section, we analyze the improved Approach 4, where the SNM allocates a contingent of sequence numbers to each transaction-generating machine. In our mathematical analysis of this approach, we will see that this has a positive effect on the *waiting time* of each ball, i.e., the number of time steps it takes until a ball is processed by a bin.

We remark that the same balls-into-bins game can also be used to model Approach 2. However, in Approach 2, there is a significant overhead required to handle user accounts and sign transactions, and while the corresponding constant factors do not impact our asymptotic analysis, the connection between the balls and the bins becomes less obvious. In Approach 3, balls correspond to requests sent to the SNM, and the bins correspond to threads on the SNM. In contrast, in Approach 2, balls correspond to transactions signed by an authorized user account, and bins correspond to nodes in the network. The main difference is that balls are only accepted to bins if the authorization is validated, resulting in much larger constant factors hidden in our asymptotic analysis.

6.2 Analysis of approach 4: introducing buffers

In the following, we extend our mathematical analysis of Approach 3 to the setting of Approach 4, where each request allocates a contingent of multiple sequence numbers. Our main concern w.r.t. Approach 4 is the time it takes to execute a transaction. In the following we will provide a bound on the processing time that hints at the existence of a non-trivial trade-off based on the size of the contingents. To this end, we start with the following warm-up result.

Lemma 1 *Consider the balls-into-bins process defined in Algorithm* 1. *Let b be a ball created in an arbitrary but fixed time step t and assume that $0 \leq \lambda \leq 1 - 1/n$. Let q^* be the bound on the queue size $\mathcal{Q}$ from Theorem* 1. *In expectation, it takes at most $\mathrm{O}(q^*/n)$ many time steps until b is processed.*

Proof Recall that from Theorem 1, we know that at time t, we have at most

$$|\mathcal{Q}(t)| \leq \mathrm{O}\left(n + \log\left(\frac{1}{1-\lambda}\right) \cdot n\right)$$

many balls in the queue $\mathcal{Q}$. Let q^* be this bound on $\mathcal{Q}(t)$ from Theorem 1. In the following, we assume that $q^* \geq n$ Further, in our asymptotic analysis, we assume that n is large enough.

Fix a time step t and assume that we have $q(t)$ many balls in the queue at time t. Let b be a ball generated at time t and, without loss of generality, assume that ball b is assigned to bin 1. Let now $X_{1,t}$ be a random variable for the total number of balls also allocated to bin 1 at time t. Clearly, $X_{1,t}$ has a binomial distribution with parameters $\mathrm{Bin}(q(t), 1/n)$. We condition on the high-probability event that $q(t)$ is bounded by q^* and observe that $X_{1,t}$ has expected value $\mathrm{E}\left[X_{1,t} \mid q(t) \leq q^*\right] \leq q^*/n$. We apply Chernoff bounds (Lemma 2 in Appendix A) to $X_{1,t}$ and obtain

$$\Pr\left[X_{1,t} \geq \frac{2eq^*}{n} \,\middle|\, q(t) \leq q^*\right] \leq 2^{-\frac{2eq^*}{n}}.$$

Let now $\mathcal{E}_{b,t}$ be the event that ball b is processed by its bin in round t conditioned on the two events that $X_{1,t} < \frac{2eq^*}{n}$ and $q(t) \leq q^*$. We have

$$\Pr\left[\mathcal{E}_{b,t} \,\middle|\, X_{1,t} < \frac{2eq^*}{n},\ q(t) \leq q^*\right] \geq \frac{n}{2eq^*}.$$

In other words, if the number of balls in the queue does not exceed the bound from Theorem 1 and the bin of ball b does not get significantly more balls than expected, there is a $1/\ell$ chance that ball b is processed, where ℓ is the total number of balls in b's bin. We use the law of total probability to remove the conditioning and get

$$\begin{aligned}
\Pr\left[\mathcal{E}_{b,t}\right] &= \Pr\left[\mathcal{E}_{b,t} \,\middle|\, X_{1,t} < \frac{2eq^*}{n},\ q(t) \leq q^*\right] \cdot \Pr\left[X_{1,t} < \frac{2eq^*}{n} \,\middle|\, q(t) \leq q^*\right] \cdot \Pr\left[q(t) \leq q^*\right] \\
&\quad \underbrace{+ \Pr\left[\mathcal{E}_{b,t} \,\middle|\, X_{1,t} \geq \frac{2eq(t)}{n} \cup q(t) > q^*\right] \cdot \ldots}_{\geq 0} \\
&\geq \frac{n}{2eq^*} \cdot \left(1 - 2^{-\frac{2eq^*}{n}}\right) \cdot (1 - \mathrm{o}(1)) \geq \frac{n}{6q^*}.
\end{aligned}$$

The lemma now follows from the observation that the time until ball b is processed is stochastically dominated by a geometric distribution $\mathrm{Geom}(n/(6q^*))$ with expected value $\mathrm{O}(q^*/n)$.

Finally, we consider the modified process where a contingent of size c is requested. We assume exactly the same theoretical process as in Algorithm 1 with the only difference that in each time step, every bin selects c balls at the same time to be removed from the queue $\mathcal{Q}$. However, the selected balls are only processed once, and every "older" ball is also processed. Intuitively, this model shows that each contingent can be used to submit c transactions to the network, but a transaction is only executed once all lower sequence numbers are used in a mined or

pending transaction. Our next theorem focuses on analyzing each transaction's waiting time in this extended model.

Theorem 2 *Consider modifying the balls-into-bins process from Algorithm* 1 *where each bin processes c balls in each time step. Let b be a ball created in an arbitrary but fixed time step t and assume that* $0 \leq \lambda \leq 1 - 1/n$. *Let* q^* *be the bound on the queue size* $\mathcal{Q}$ *from Theorem* 1. *In expectation it takes at most* $\mathrm{O}\left(\frac{q^{*2} \log n}{c \cdot n}\right)$ *many time steps until ball b is processed.*

Proof The first part of the proof follows along the lines of the proof of Lemma 1. With regard to the queue size at time t, the same analysis from Theorem 1 applies in the case where in each time step c balls are processed. Indeed, a straightforward coupling shows that the queue size cannot become larger if additional balls are removed. The main difference to the proof of Lemma 1 is that for the event $\mathcal{E}_{b,t}$ that ball b is selected in time step t we have

$$\Pr\left[\mathcal{E}_{b,t} \,\middle|\, X_{1,t} < \frac{2eq^*}{n},\ q(t) \leq q^*\right] \geq \frac{c \cdot n}{2eq^*}.$$

Indeed, if the number of balls in the queue does not exceed the bound from Theorem 1 and the bin of ball b does not get significantly more balls than expected, then there is now a c/ℓ chance that ball b is selected. Hence we get

$$\Pr\left[\mathcal{E}_{b,t}\right] \geq \frac{c \cdot n}{6q^*}.$$

Let b be an arbitrary but fixed ball created at time t. Observe that b is processed once it is selected and all other balls created prior to ball b have been selected. For simplicity, we assume that in every time step, there are exactly q^* many balls in the system. As we have shown in Theorem 1, this overestimates the actual size of the queue w.h.p.

We follow along the lines of the analysis of the classical coupon collector's problem [14, 15] and define T_i as the time until the i-th ball is selected. The time until ball i is selected once ball $i-1$ has been selected is dominated by a geometric distribution $T_i \sim \mathrm{Geom}\left(\frac{c \cdot n \cdot (q^* - i)}{6q^{*2}}\right)$. Let now T be the time until all balls are selected. We have by linearity of expectation

$$\mathrm{E}[T] \leq \sum_{i=0}^{q^*-1} \mathrm{E}\left[T_i\right] = \sum_{i=0}^{q^*-1} \frac{6q^{*2}}{c \cdot n \cdot (q^* - i)} = \sum_{i=1}^{q^*} \frac{6q^{*2}}{c \cdot n \cdot i} = \mathrm{O}\left(\frac{q^{*2} \log n}{c \cdot n}\right).$$

□

7 Discussion

Referring to the research questions outlined in Sect. 1, we have presented four approaches to facilitate scalable transaction creation for blockchains, addressing our first research question. For the second research question, which focuses on the performance of these approaches in terms of throughput, latency, and fairness, we discuss each approach in the following paragraphs:

Overall, the first approach, which adds the sequence number to a singleton middleware, displayed the weakest scalability. While the transaction throughput increased proportionally for additional TX-creating machines in the other approaches, this was not the case for Approach 1. Instead, the throughput increase caused by adding additional machines quickly declined, so scaling from two to three machines only increased throughput by 10.8%. When scalability is the objective, Approach 1 should be avoided.

In contrast, Approach 2 displayed a much more favorable performance, with a stable throughput that sufficiently increases when adding TX-creating machines. The key advantage of Approach 2 is that the TX-creating machines do not depend on any other component to set the sequence number or to forward transactions to the blockchain node, which is why it offers the overall highest throughput of 1 045 TPS. As a downside, Approach 2 requires supplying a distinct Ethereum account for each TX-creating machine and a slightly higher gas usage. This might introduce higher complexity for dynamically scaling the number of machines and realizing authentication and authorization. In summary, Approach 2 is a solid design choice to implement horizontal scalability for TX-creating machines, provided that using multiple Ethereum accounts or higher gas usage is acceptable in a given context.

For Approach 3, it is harder to give a clear recommendation. During our simulation run with three TX-creating machines, the transaction throughput of our implementation was similar to Approach 2. With an average of 1 030 TPS, it even outperformed Approach 4—albeit slightly—for all three tested contingent sizes. However, the singleton SNM will eventually become a bottleneck when too many TX-creating machines are running concurrently, so Approaches 3 and 4 cannot be scaled indefinitely. The big disadvantage of Approach 3 is that the TX-creating machines will have to make c-times as many requests to the SNM as they would in Approach 4. This means the SNM will become a bottleneck in Approach 3 considerably sooner, so the maximum number of machines running concurrently is drastically higher in Approach 4. In contrast to the other approaches, if the contingent size c is too large, Approach 4 is suboptimal in latency, particularly the distribution of waiting times, which can be understood as *fairness*. By selecting a smaller c, this issue can be avoided. Given these points, Approach 4 should generally be preferred over Approach 3, even though Approach 3 had a slightly higher average throughput.

Approaches 2 and 4 are decent choices for horizontal scaling of transaction creation and offer similar performance. However, while Approach 2 requires a way to deal with account management and authorization (see Sect. 3.3), Approach 4 necessitates implementing an additional component—the SNM—which poses a single

point of failure, so there need to be adequate measures for mitigation and recovery. For a given context, the choice comes down to finding the better trade-off between the drawbacks and advantages, in most contexts, likely between Approaches 2 and 4.

(3) With regard to our third research question, our theoretical analysis of Theorem 1 shows that the transaction creation process is stable in the following sense: We prove that at an arbitrary point in time, the total number of requests awaiting allocation remains bounded by $O(n + \log(1/(1-\lambda)) \cdot n$ with high probability. This applies to Approach 2 (which is, from a theoretical point of view, a special case of Approach 3), Approach 3, and Approach 4 (which uses an extended model of Approach 3). Regarding Approach 4, Theorem 2 shows an interesting gap between the performance of Approach 3 and Approach 4, wherein in the former, transactions are processed immediately, and in the latter, the contingent size may cause transactions to stall. This effectively forces every selected ball to wait until all older balls have also been selected. We remark that our theoretical findings based on an asymptotic analysis align with our empirical data based on real-world instance sizes; indeed, for a suitable choice of the buffer size c, we observe that Approach 4 is among the fastest approaches in theory and practice.

8 Related work

In 2016, Croman et al. were among the first to scientifically explore ways for scaling blockchains [5]. They also explained why earlier methods for scaling—increasing the block size so that more transactions can be included in a single block while decreasing the inter-block time—are limited. That is why other, more drastic changes are necessary to enable scalability to industrial use cases. Another early paper by Vukolić [19] discussed different proposals for blockchain scalability, including the option to rely on alternatives to PoW.

Considering the popularization of smart contracts in Ethereum, Dickerson et al. [6] proposed that miners speculatively process transactions (and thereby execute smart contracts) concurrently. If conflicts arise, the affected contracts are rolled back and serially re-executed. The resulting execution schedule is populated alongside the mined block so that validators can execute the smart contracts in parallel.

Early blockchain protocols achieved only a limited number of TPS. For instance, Bitcoin achieves ca. 7 TPS. PoW-based Ethereum achieved 15 TPS in 2018 [1] and could, due to increased block "size" and frequency at the time of writing, achieve a theoretical 120 TPS. More recent protocols achieve higher numbers, e.g., the Red-Belly Blockchain achieved 30k TPS in a globally distributed network [4], with the authors stating that the bottleneck in this experiment was the load generation, which is the focus of our work. The applied consensus mechanism primarily affects the possible TPS. The usage of PoW naturally leads to a low TPS, while (Delegated) PoS enables increasing the TPS to higher numbers. For instance, Ethereum 2.0 applies PoS and aims at 200k to 300k TPS.

Furthermore, blockchain benchmarking is also relevant to the work at hand. Benchmarking of blockchains concerning scalability and general performance has been an important research topic in recent years [20]. For instance, Gervais et al. [9] present a simulation framework for analyzing the security and performance constraints of PoW blockchains. Dinh et al. [7] implement BLOCKBENCH, a framework for analyzing private blockchains. For this, workloads are defined. Lobmaier et al. present another blockchain simulator focusing on communication overhead [13]. Finally, Hyperledger Caliper[8] is used to benchmark the performance of Hyperledger-based blockchains.

To the best of our knowledge, none of the discussed solutions considers the scalability of creating transactions. They focus mainly on increasing the transaction throughput of blockchains and, therefore, only consider the transaction processing by miners and validators. These approaches are not applicable for improving the scalability of the client side of dApps, i.e., for creating transactions. Therefore, the work at hand can be used to extend existing frameworks and simulators.

9 Conclusion

Following the increased throughput scalability of blockchain systems, high-volume applications with a single participant issuing thousands of TPS become possible. However, the horizontal scaling of TX-creating machines has received little attention in research to date. We study this subject and propose four approaches to this end. Prototypical implementations of the approaches allow us to evaluate them experimentally. Our work demonstrates that it is feasible to scale transaction creation horizontally, and two approaches—Approaches 2 and 4—achieve both good scalability and fair latency distributions. Approach 2 introduces additional complexity because it requires on-chain account management and slightly higher gas usage but offers the highest throughput. Approach 4 relies on the SNM, which could become a single point of failure and achieves a throughput slightly below that of Approach 2.

In future work, we want to consider the failure of transactions and ordering constraints on the incoming requests. Currently, we assume no dependencies exist between transactions other than the sequence number of their associated blockchain account. However, business logic might require transactions to be processed in a specific sequence, and that sequence would then need to be accounted for when setting the sequence number. This might require changes to the transaction creation approaches or even render Approach 4 unsuitable. Last but not least, we have not regarded the batching of transactions so far.

[8] https://hyperledger.github.io/caliper/, accessed 2024-02-10.

Appendix A: Concentration inequalities

The following Chernoff bound is based on Theorem 4.4 in [14].

Lemma 2 (Chernoff Bounds [14]) *Let* $X_1, \ldots, X_n$ *be independent Bernoulli trials such that* $\Pr\left[X_i = 1\right] = p_i$. *Let* $X = \sum_{i=1}^{n} X_i$. *Then*

$$\Pr\left[X \geq R\right] \leq 2^{-R} \quad \text{if} \quad R \geq 2e\mathrm{E}[X].$$

Author's contribution OD conducted the basic research leading to the results presented in this paper. He also provided a report, which was then condensed by RH, IW, and StS into the final paper. OD project was supervised by RH and IW, who developed the research idea. StS led the paper writing efforts for both the original paper as well as this revision. DS provided the theoretical analysis. Both IW and SS extended the discussion of the related work, based on OD original report. MS contributed heavily to the revision, adding the general information about blockchains and aligning the complete paper.

Funding Open Access funding enabled and organized by Projekt DEAL. The financial support by the Austrian Federal Ministry for Digital and Economic Affairs, the National Foundation for Research, Technology and Development as well as the Christian Doppler Research Association for the Christian Doppler Laboratory for Blockchain Technologies for the Internet of Things is gratefully acknowledged.

Data availability No datasets were generated or analysed during the current study.

Declarations

Conflict of interest The authors declare no competing interests.

References

1. Bach LM, Mihaljević B, Zagar M (2018) Comparative analysis of blockchain consensus protocols. In: International convention on information and communication technology, electronics and microelectronics (MIPRO). IEEE, pp 1545–1550
2. Berenbrink P, Friedetzky T, Hahn C, et al (2021) Infinite balanced allocation via finite capacities. In: 41st IEEE international conference on distributed computing systems (ICDCS). IEEE, pp 965–975
3. Bratanova A, Devaraj D, Horton J, et al (2019) Blockchain 2030: a look at the future of blockchain in Australia. Tech rep, Data61, CSIRO, Brisbane, Australia
4. Crain T, Natoli C, Gramoli V (2021) Red belly: a secure, fair and scalable open blockchain. In: IEEE symposium on security and privacy (SP). IEEE, pp 466–483

5. Croman K, Decker C, Eyal I, et al (2016) On scaling decentralized blockchains. In: Financial cryptography and data security. Springer, pp 106–125
6. Dickerson T, Gazzillo P, Herlihy M, et al (2017) Adding concurrency to smart contracts. In: ACM symposium on principles of distributed computing. ACM, pp 303–312
7. Dinh TTA, Wang J, Chen G, et al (2017) BLOCKBENCH: A framework for analyzing private blockchains. In: 2017 ACM International conference on management of data. ACM, pp 1085–1100
8. Dubhashi DP, Ranjan D (1998) Balls and bins: a study in negative dependence. Random Struct Algorithms 13(2):99–124
9. Gervais A, Karame GO, Wüst K, et al (2016) On the security and performance of proof of work blockchains. In: 2016 ACM SIGSAC conference on computer and communications security. ACM, pp 3–16
10. Guo H, Yu X (2022) A survey on blockchain technology and its security. Blockchain Res Appl 3(2):100067
11. Joshi S (2021) Feasibility of proof of authority as a consensus protocol model. CoRR abs/2109.02480
12. Khan D, Jung LT, Hashmani MA (2021) Systematic literature review of challenges in blockchain scalability. Appl Sci 11(20):9372
13. Lobmaier D, Konlechner R, Schulte S, et al (2024) Assessing routing algorithms for payment channel networks. Distributed Ledger Technologies: Research and Practice 3:6:1–6:27
14. Mitzenmacher M, Upfal E (2005) Probability and computing: randomized algorithms and probabilistic analysis. Cambridge University Press
15. Motwani R, Raghavan P (1995) Random Algorithms. Cambridge University Press, Cambridge
16. Nakamoto S (2008) Bitcoin: a peer-to-peer electronic cash system. http://www.bitcoin.org/bitcoin.pdf
17. Saleh F (2021) Blockchain without waste: proof-of-stake. Rev Financ Stud 34(3):1156–1190
18. Szabo N (1997) Formalizing and securing relationships on public networks. First Monday 2(9)
19. Vukolić M (2016) The quest for scalable blockchain fabric: proof-of-work vs. BFT replication. In: Open problems in network security. Springer, pp 112–125
20. Wang R, Ye K, Xu C (2019) Performance benchmarking and optimization for blockchain systems: a Survey. In: Second international conference on blockchain. Springer, pp 171–185
21. Weber I, Gramoli V, Staples M, et al (2017) On availability for blockchain-based systems. In: IEEE international symposium on reliable distributed systems. IEEE, pp 64–73
22. Wieder U (2017) Hashing, load balancing and multiple choice. Found Trends Theor Comput Sci 12(3–4):275–379
23. Wood G et al (2014) Ethereum: a secure decentralised generalised transaction ledger. Ethere Project Yellow Paper 151(2014):1–32
24. Xu X, Pautasso C, Zhu L, et al (2018) A pattern collection for blockchain-based applications. In: European conference on pattern languages of programs. ACM, pp 1–20