INSTITUTO SUPERIOR DE ENGENHARIA DO PORTO

MESTRADO EM ENGENHARIA INFORMÁTICA

Exploração de algoritmos de consenso no Quorum

JOÃO PEDRO ASCENSÃO LOPES

junho de 2023

POLITÉCNICO
DO PORTO

Exploring consensus algorithms in Quorum

João Lopes

MSc in Computer Engineering, Specialisation Area of Software Engineering

Supervisor: Isabel Azevedo

Porto, June 29, 2023

Dedicatory

To my parents, for all the support over the years.

Statement of Integrity

I hereby declare having conducted this academic work with integrity.

I have not plagiarized or applied any form of undue use of information or falsification of results along the process leading to its elaboration.

Therefore, the work presented in this document is original and authored by me, having not previously been used for any other end.

I further declare that I have fully acknowledged the Code of Ethical Conduct of P.PORTO.

ISEP, June 29, 2023

João Pedro Ascensão Lopes

Abstract

As blockchain technology matures, more industries are becoming interested in evaluating if the technology can answer their needs for decentralized systems that guarantee data immutability and traceability.

Quorum is a blockchain platform that accommodates enterprise use-cases by extending Ethereum to support private transactions and a higher transaction throughput. To achieve this, Quorum replaced Ethereum's proof-of-stake consensus mechanism with proof-of-authority ones, supporting four different algorithms: Raft, Clique, IBFT 1.0, and QBFT.

This work explores Quorum's consensus algorithms and how they affect performance and fault-tolerance, in order to assess the best use cases for each and what should drive their choice.

A GoQuorum network was set up, and benchmarks were run against this system under different scenarios while only changing the consensus algorithm for each scenario.

Results showed that Raft is the most performant consensus algorithm in Quorum in both private and public transactions. Additionally, QBFT achieved the same performance as IBFT, and Clique was the worst performer across the board, particularly due to having high resource-usage. Regarding fault-tolerance, it was found that bringing validator nodes down at random, when the network has high-availability, had no impact on networks under any of the consensus algorithms.

Keywords: Blockchain, Quorum, Consensus Algorithms, IBFT, QBFT, Clique, Raft

Resumo

Com *blockchain* a entrar numa fase de maturidade, cada vez mais indústrias procuram avaliar se esta tecnologia responde às suas necessidades de sistemas distribuídos que garantam a imutabilidade e rastreabilidade dos seus dados.

Quorum é uma plataforma *blockchain* que procura acomodar os casos de uso destas empresas ao extender Ethereum para suportar transações privadas e um maior número de transações por segundo. Para esse efeito, o Quorum substituiu o mecanismo de consenso *proof-of-stake* do Ethereum por um mecanismo de *proof-of-authority*, onde quatro algoritmos são suportados: Raft, Clique, IBFT 1.0, e QBFT.

Este trabalho explora os algoritmos de consenso suportados pelo Quorum de modo a determinar como estes afetam o desempenho e tolerância a falhas das redes, e consequentemente perceber os melhores casos de uso para estes algoritmos e que fatores ter em conta aquando a sua escolha.

Foi criada uma rede de GoQuorum, e vários testes de desempenho foram corridos contra a rede sob diferentes cenários, onde para cada cenário a única variável foi o algoritmo de consenso.

Os resultados mostraram que o Raft foi o algoritmo de consenso com melhor desempenho, tanto em transações públicas como privadas. Adicionalmente, o QBFT e o IBFT atingiram o mesmo desempenho, e o Clique o pior de todos, particularmente pelo seu alto uso de recursos do sistema. Quanto a tolerância a falhas, foi concluído que trazer nós validadores abaixo aleatóriamente enquanto o sistema está configurado com alta disponibilidade não tem impacto nas redes, independentemente do algoritmo de consenso utilizado.

Acknowledgement

I would like to show my sincerest gratitude to my thesis supervisor, Isabel Azevedo, for the dedication and availability she has to her students, for pointing me to the right places when researching, and for the sincerity when giving feedback. Her guidance has been invaluable.

Contents

List of Figures

List of Tables

List of Source Code

List of Acronyms

AHP	Analytic hierarchy process.
BFT	Byzantine fault tolerant.
DLT	Distributed Ledger Technology.
EVM	Ethereum Virtual Machine.
GQM	Goal, question, metric.
NCD	New concept development.
NPD	New product development.
P2P	Peer-to-peer.
PoS	Proof-of-stake.
SUT	System under testing.
TLS	Transport Layer Security.

Chapter 1

Introduction

In this chapter, the problem statement is initially described, stating the project's goals. Following this, the execution plan to be followed is presented, and, finally, an overview of this documents structure is given.

1.1 Problem statement

More industries are becoming interested in exploring solutions for their problems using blockchain technologies, as these are seen as strong candidates for solutions in multiple fields and use cases due to providing traceability by default. Examples of such use cases include secure information exchange, asset tracking and management, and financial flow traceability [1]. Consequently, there is an increasing need for more data on how these technologies perform and how they can evolve to accommodate these industries' needs.

Blockchain was originally introduced and implemented in 2014 by Bitcoin [2] to power its decentralized digital currency. It is an open and permissionless network that assures trust and consensus among unknown entities using cryptography.

The interest in this technology has had ups and down over the years, usually tied to the cryptocurrency market that it supports. Figure 1.1 shows the evolution of interest in blockchain based on Google Trends[1] data.

[1]https://trends.google.com/trends/explore?date=2014-01-18%202023-02-18&q=blockchain

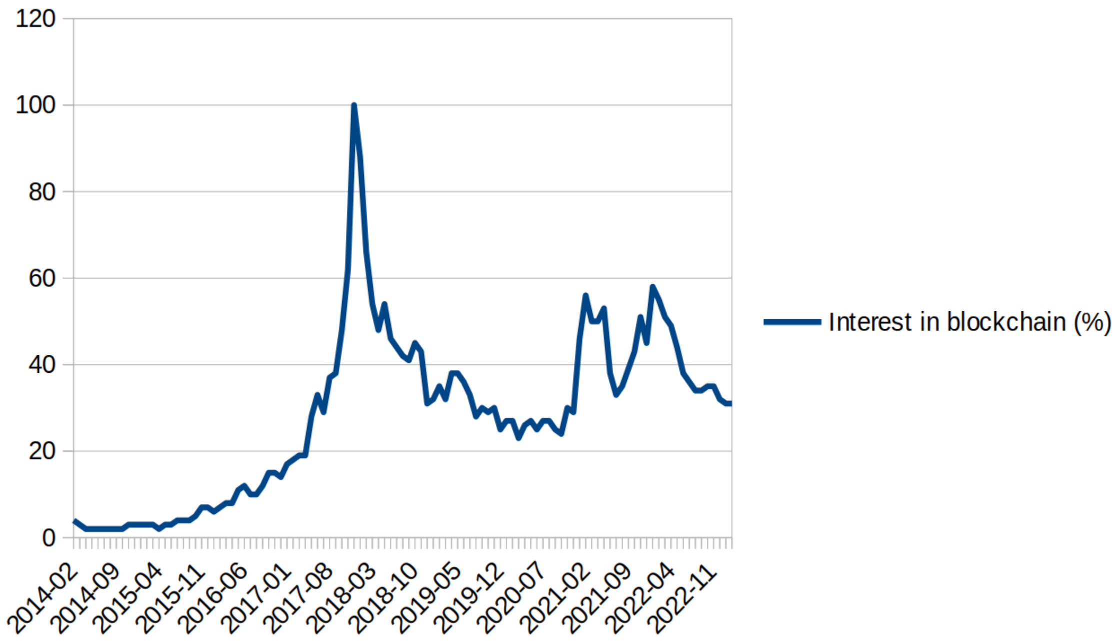


Figure 1.1: Interest in blockchain over the years

Although there are still recent peaks of relative interest, this metric peaked in 2017 where the technology was still emerging and largely unknown. At the moment, the concept of a distributed ledger is already known in high-level terms to most of the software industry, and as it continues being explored, it enters its early phase of maturity. This means that more industries outside the initial early adopters are assessing if blockchain can offer solutions to existing problems for which previous software frameworks were inept [3].

However, due to its nature, Bitcoin's original implementation of this technology and similar ones are generally unfit for enterprise use-cases, which need privacy when handling their data and control over what each party can access. To overcome this, several blockchain platforms and protocols have emerged to cater to enterprises. Enterprise use-cases solved by these technologies include ensuring data integrity across multiple parties and transactions, or allowing a distributed system of nodes with a single source of truth and data sharing [4], [5].

Supply chain tracking, for example, is a big use case for private blockchain platforms, and has already been implemented and been in use for companies such as Renault[2] and Walmart[3] [6].

Enterprise consultancy companies have also been investing heavily into blockchain. Cloud providers like IBM, Amazon and Oracle have also started offering Blockchain-as-a-Service (BaaS) solutions, and continue to innovate in that field [7]. IBM is reported to have had 1500 employees working on blockchain in 2018 [8], and Deloitte reports to be heavily invested in permissioned blockchains to support use cases beyond financial services [9]. According to the same survey by Deloitte, 80% of the participants say their industries will see new revenue streams from blockchain solutions, and the banking industry is leading the adoption

[2]https://www.ibm.com/case-studies/renault/
[3]https://www.hyperledger.org/learn/publications/walmart-case-study

of the new technology, followed by telecommunications, entertainment, manufacturing and healthcare [9].

Forbes magazine also reports as recently as February 2023, in [10], that top enterprises continue investing in enterprise blockchain, despite its hype cycle having peaked, which demonstrates maturity, real-world potential to solve existing problems and customer trust.

Despite this interest, the technologies and underlying consensus algorithms are still being iterated on and relatively under-explored, making adopting the technology risky, and contributing to estimates that say that up to 92% of all blockchains projects fail in their early stages [11].

It is thus important to have more independent data on these technologies and provide concrete, verifiable metrics that the industry can use to assess which solution to implement. This work aims to provide data on how different consensus algorithms for permissioned networks behave under different scenarios, so that interested parties can better evaluate how to choose them for their systems.

The three main permissioned blockchain platforms are R3 Corda [12], Hyperledger Fabric[13], and Quorum [14], [15] . Despite aiming to provide the same features, these platforms work in very different ways, and thus the way they achieve consensus across the network is also different [16].

Hyperledger Fabric relies on deterministic consensus, where a block validated by a peer is guaranteed to be final and correct. To do this, it uses an ordering service, which is a collection of orderer nodes that, as the name implies, ensure the order the transactions and packages them into blocks, which later get committed to the ledger. As of version 2.x, Hyperledger Fabric only supports and recommends using Raft as the ordering service implementation [17].

In R3 Corda, consensus is achieved by proving that transactions are both valid and unique before being committed to the ledger. Validity is done by *walking the chain* and ensures that transactions are properly signed and all smart contracts involved in their input and output accept them. Meanwhile, uniqueness prevents double-spend by ensuring that the same input is not used in more than one transaction. This is guaranteed by using a notary service, which is a cluster of notary nodes. Notary clusters are pluggable, and a Corda network can have multiple ones, all running different consensus algorithms. However, adding custom notaries is considered experimental, and only their in-house JPA notary implementation is available and recommended for high-availability needs, with even the sample integrations to Raft and SmartBFT clusters being tagged as experimental and deprecated on their documentation [18]–[20].

Quorum, on the other hand, provides four consensus mechanisms that can be easily replaced by configuration [21], making it the ideal testing platform for the project.

1.2 Research methodology

To assess how Quorum's consensus algorithms compare, a controlled experiment will be conducted. A Quorum network will be set up under several scenarios, and benchmarks will be run against each while only changing the consensus algorithm. The results obtained will then be used to take conclusions on how each consensus algorithm performs with public and private transactions, how they scale, and how they handle faulty networks.

1.3 Goals

This work's main goal is then to explore the effect that different consensus algorithms have on fault tolerance and performance in Quorum. Based on these findings, it should be possible to have a good understanding of which use-cases each protocol fits better, namely for different industries and development cycles.

Although there are other metrics to explore in regard to these algorithms, such as security, those will not be explored in this work due to time constraints and instead will be left for future work.

The research questions for this work are thus the following:

- RQ1: What is the effect of the consensus algorithm on performance in Quorum blockchains?
- RQ2: What is the effect of the consensus algorithm on fault-tolerance in Quorum blockchains?

1.4 Structure of the document

This paper features six chapters in total: Introduction, State of the art, Analysis and Design, Implementation and benchmarking, Experimentation and Evaluation, and Conclusions.

This first chapter introduced the problem being explored and the project's goals and scope.

In the second chapter, the current knowledge on the topic is explored. This is done by explaining what Quorum is, how its consensus algorithms work, and the current state of benchmarking technologies for it. A systematic mapping study is also presented to understand the current research on the topic.

In the third chapter, the tools and techniques that can be used to benchmark Quorum's consensus mechanisms are explored and chosen for the project. The design and architecture of the system to be tested and the monitoring system is also detailed.

In the fourth chapter, the implementation of the benchmarking framework and proof of concept domain to run against the benchmarks is described in detail. Then, the results obtained from the benchmarks are detailed and analyzed to draw conclusions.

In the fifth chapter, the evaluation definition and methodology for the work are described. Afterward, the methodology is applied to assess if the work done achieved the goals it set.

The sixth chapter details the conclusions taken from the project. A summary of the goals reached is presented, as well as any limitations and difficulties felt. The chapter ends with a personal appreciation of the work done.

Chapter 2

State of the art

In this chapter, first, a dive into Quorum is done, showcasing the technology's current state and how it works. Then, a study into the current knowledge on permissioned blockchains and particularly Quorum is done. The study summarizes knowledge on Quorum's consensus mechanisms and compares their theoretical benchmarks. Lastly, benchmarking tools for Quorum are summarized, and their applicability for this project is assessed.

2.1 Quorum

Quorum is an open-source protocol layer for a permissioned blockchain network. It is also a blockchain platform, which encapsulates the low-level intricacies of the network behind well-defined APIs. It was originally developed by J.P. Morgan Chase in 2016, and later sold to ConsenSys in 2020 [22], [23]. It extends the Ethereum protocol by introducing features that accommodate enterprise use-cases [24].

Ethereum is an open-source protocol and implementation for a decentralized, distributed platform that enables the execution of smart contracts. A smart contract is a stateful program that resides at a specific address in the Ethereum blockchain, and always executes the same block of code whenever called by a message or transaction [4], [25].

To maintain consensus on the order of transactions and account balances across all the nodes in the blockchain, Ethereum, first used the proof-of-work mechanism and recently switched to proof-of-stake. Both mechanisms target public networks, where all the transaction data and account information are publicly available, and this makes Ethereum prohibitive for most enterprise use-cases [26], [27].

Additionally, by virtue of being open, public, and permissionless, Ethereum requires complex cryptography to maintain security and consensus across the multiple nodes in the network. Security and consensus are maintained with recourse to the Proof-of-Stake protocol and Ether, the blockchain's cryptocurrency. All transactions done by smart contracts have an associated Gas fee, paid for in Ether, which discourages denial-of-service and brute force attacks. Consensus on the network's state is then achieved by having pseudo-randomly selected nodes *stake* part of their Ether on the validity of transactions, losing their Ether if they approve invalid transactions. This effectively prevents malicious actors from attempting to steal and creating a fake blockchain state, barring a 51% attack, in which more than half of the network's processing power agree on a new, incorrect state [28].

Quorum, by extending Ethereum to introduce private transactions via a permission system, aims to make this technology viable for enterprise solutions. It achieves its goals by introducing private transactions, permissioned access to data, different consensus mechanisms, account management via account plugins, and a general higher transaction throughput [29].

Quorum splits the Ethereums Patricia Merkle Tree [30] into a public one and multiple private ones. Then, permissions are added to notes by using cryptography to ensure nodes cannot access data from a transaction they are not part of. The Ethereum consensus mechanism is replaced with a proof-of-authority one, for which there are currently four supported protocols: Raft, Clique, QBFT, and Istanbul BFT [21], [29].

There are two open-source clients available to develop for Quorum: GoQuorum [31] and Hyperledger Besu [32]. GoQuorum is developed by ConsenSys and is a fork of GoEthereum, the official Go language implementation of Ethereum's protocol. On the other hand, Hyperledger Besu is an Ethereum client written in Java and designed to be enterprise-friendly by supporting private transactions and permissioned consensus mechanisms. It is developed by the Hyperledger Foundation, a collaborative effort supported by the Linux Foundation.

In this work only the GoQuorum client will be explored, and all mentions of Quorum in the next chapters refer to GoQuorum.

2.1.1 Private transactions

In Ethereum transaction data are publicly available in the network, which is a requirement for its consensus mechanisms to work [4]. However, enterprise solutions also require private transactions in which only specified nodes are privy to the transaction data.

To support these use cases, Quorum supports private transactions by using the Tessera private transaction manager, an open-source project also developed by ConsenSys [33].

Tessera stores and allows retrieval of private transactions without having access to private keys. All cryptographic functionalities, namely encryption, decryption, and key management, are delegated to an enclave, which is a processing environment that works as a black box. Enclaves can either be local or remote, the latter of which is communicated with over HTTP [33], [34].

Tessera also supports multi-tenancy by being stateless and featuring peer discovery. This means that adding a key to a multi-tenant Tessera node and restarting it results on the key being propagated to all other nodes in the network [34].

With this, Quorum nodes have a public state, accessible to all nodes in the network, and a private state reserved for nodes with specific permissions. Public transactions function the same way as Ethereum ones. If a node receives a public transaction, each participant will execute the smart contract code and their state is updated accordingly.

Private transactions, however, function differently. An example lifecycle of a private transaction in Quorum can be seen in Figure 2.1 [35].

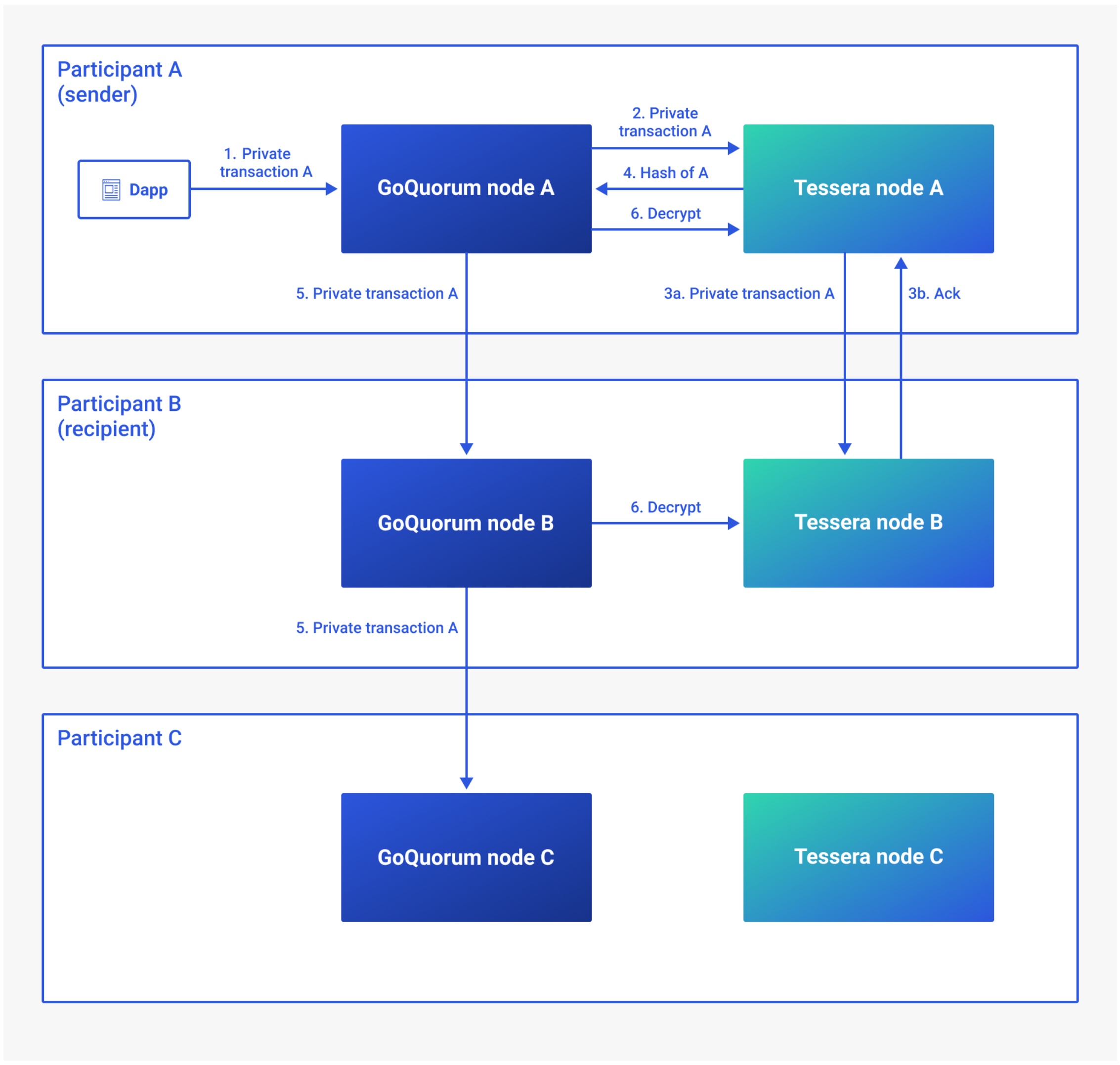


Figure 2.1: Private transaction lifecycle [35]

A node executing a private transaction must first specify which nodes can access the information. Then, before the transaction is sent and propagated to the rest of the network, the original payload is replaced with a hash containing the key for the location of the encrypted payload, stored in Tessera. Afterwards, the transaction is propagated across the network, and participants with access to the information can replace the key with the original payload by retrieving it from their Tessera instance.

2.1.2 Permissioning

Quorum is considered a permissioned blockchain, which means it enables fine-grained access to data by implementing a network permissioning model, of which there are two types: Basic and Enhanced. With the basic permissioning model, developers can specify which particular nodes a given node can connect to, and which particular nodes it can receive connections from. Configuration for this is done by specifying an allowlist in a JSON file named *permissioned-nodes.json*. An example of this file can be seen in listings 2.1 [36], [37].

```
[
  "enode://remotekey1@ip1:port1",
  "enode://remotekey2@ip2:port2",
  "enode://remotekey3@ip3:port3"
]
```

Listing 2.1: Basic permissioned nodes file [37]

With the enhanced permissioning model, additional flexibility and options are given. Accounts and account management are greatly extended by using smart contracts to apply the permissioning rules at the time of transaction entry and block minting [36], [37]. Figure 2.2 shows the key definitions of the enhanced permissions model.

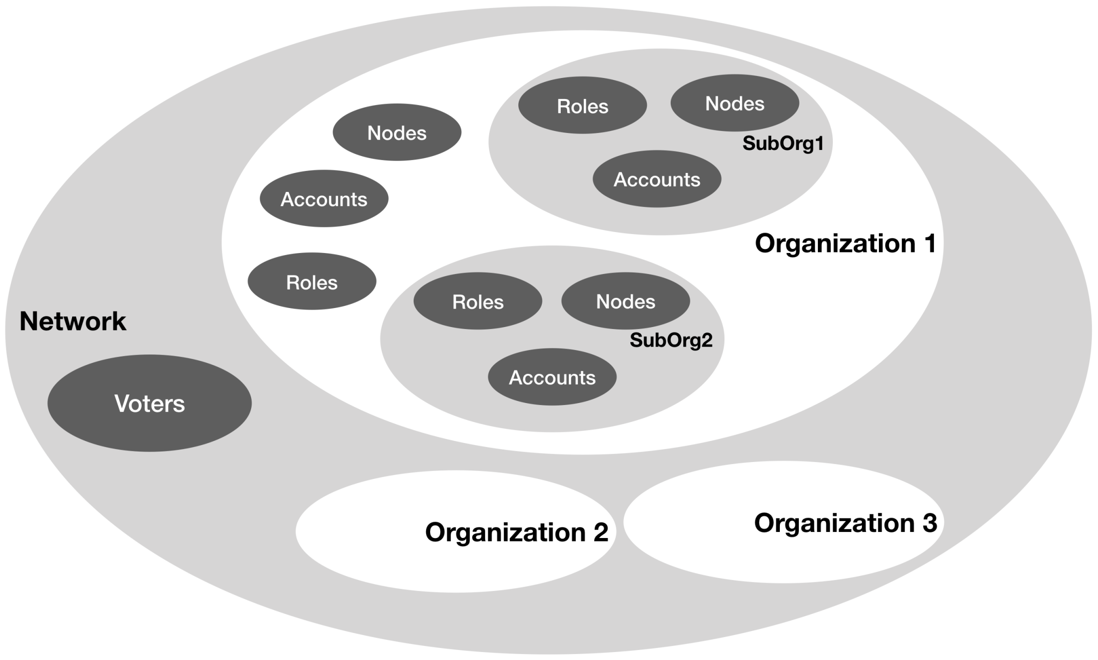


Figure 2.2: Permissions enhanced model [36]

The network represents the entire blockchain, and consists of an interconnected set of nodes, constituting a group of organizations. The network administrator can propose and approve new organizations, as well as assign administrator accounts to these. Each organization's administrator can then create sub-organizations, roles and accounts for their organization. Each sub-organization can also have their own administrators, and its own specific roles and accounts [36], [38].

An account is an externally owned Ethereum account. Each account has a role assigned, and its permissions derive from it. A Voter is any account that can vote for specific actions but is not necessarily part of any organization [36], [38].

To configure this, developers need to first setup the network by deploying the initial nodes and the *PermissionsUpgradable.sol* contract. Once all contracts are deployed, the JSON configuration file *permission-config.json* must be created to set up the network administrators. An example of this file can be seen in listings 2.2.

```json
{
  "permissionModel": "v2",
  "upgradableAddress": "0x1932c48b2bf8102ba33b4a6b545c32236e342f34",
  "interfaceAddress": "0x4d3bfd7821e237ffe84209d8e638f9f309865b87",
  "implAddress": "0xfe0602d820f42800e3ef3f89e1c39cd15f78d283",
  "nodeMgrAddress": "0x8a5e2a6343108babed07899510fb42297938d41f",
  "accountMgrAddress": "0x9d13c6d3afe1721beef56b55d303b09e021e27ab",
  "roleMgrAddress": "0x1349f3e1b8d71effb47b840594ff27da7e603d17",
  "voterMgrAddress": "0xd9d64b7dc034fafdba5dc2902875a67b5d586420",
  "orgMgrAddress": "0x938781b9796aea6376e40ca158f67fa89d5d8a18",
  "nwAdminOrg": "ADMINORG",
  "nwAdminRole": "ADMIN",
  "orgAdminRole": "ORGADMIN",
  "accounts": [
    "0xed9d02e382b34818e88b88a309c7fe71e65f419d",
    "0xca843569e3427144cead5e4d5999a3d0ccf92b8e"
  ],
  "subOrgBreadth": 3,
  "subOrgDepth": 4
}
```

Listing 2.2: Enhanced permissioning config file [38]

All further configurations of creating organizations, sub-organizations, and roles are done by executing transactions via smart contracts.

2.1.3 Consensus mechanisms

Blockchains are networks of distributed nodes, all of which must agree on a certain shared state. Consensus mechanisms are the ways in which all the nodes in the network can agree on the shared state after a series of transactions occur.

Ethereum uses the proof-of-stake consensus protocol, which is unnecessary and impossible to implement in a permissioned network where all participants are known and transaction data are private. As such, Quorum required consensus algorithms that could work in this setting, while also providing faster consensus and transaction finality to speed up enterprise processes.

Currently, Quorum provides the Raft, Clique, IBFT 1.0, and QBFT consensus algorithms, all of which are proof-of-authority.

In proof-of-authority algorithms, a small number of nodes are given the power to validate transactions and include new blocks in the blockchain. Validators are able to vote on new validators or remove existing ones, and consequently the system works on a reputation basis, where validators have the incentive to correctly process transactions so that they retain their authority. Because the system is based on reputation, there is no need for an underlying currency, and the computational power required by the blockchain is lower than the more common proof-of-stake and proof-of-work mechanisms [39].

IBFT, QBFT, and Clique are considered to be byzantine fault-tolerant. Byzantine fault-tolerance is an important term when evaluating blockchain consensus algorithms. It refers to the Byzantine generals problem, where a number of generals who are physically separated are attacking a fortress and need to decide whether to attack or retreat. Some generals want to attack while others want to retreat, and they need to vote on a coordinated solution so that they all perform the same action. If they fail at coordinating, a half-hearted attack will be done, resulting in certain defeat. The problem also assumes that there can be malicious generals who are intentionally trying to sabotage the strategy.

Given this formulation, a distributed system is considered byzantine fault-tolerant when it can continue working safely despite part of its nodes failing, and others acting maliciously.

Raft

Raft is a consensus algorithm developed to handle replicated logs or state machines. It was specifically designed to be easy to understand, and so it is decomposed into multiple independent sub-problems: leader election, log replication, and safety [40].

In a Raft cluster, nodes can be in one of three states: leader, follower, and candidate. On a common deployment, there is a single leader, and all other nodes serve as followers. All followers are passive nodes and simply redirect all client requests to the leader, which is always assumed to be correct. Time is divided into terms of arbitrary length, and at the end of each term a new election occurs, in which candidate nodes try to become the leader for the new term. Figure 2.3 shows the election lifecycle of nodes in a Raft cluster [40].

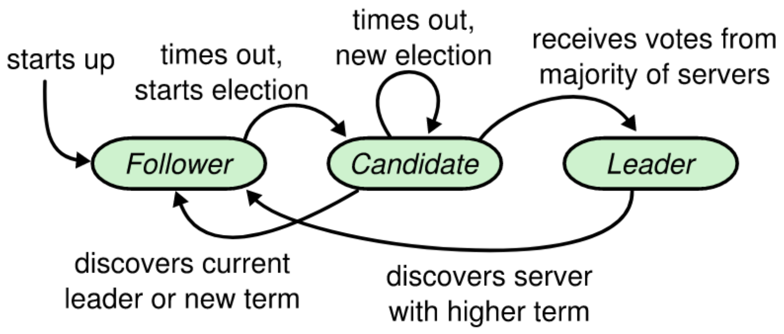

starts up
times out, starts election
times out, new election
receives votes from majority of servers
Follower
Candidate
Leader
discovers current leader or new term
discovers server with higher term

Figure 2.3: Raft election lifecycle [40]

Leaders maintain their status by sending heartbeats to followers. When a follower stops receiving heartbeats for a certain period of time, it assumes there is no viable leader and begins an election by transitioning to the candidate state. The node then votes for itself and requests votes from other nodes. The node will continue in this state until it wins the election, or another node wins the election, or a randomized period of time passes with no winner. In the later case, a new election will start until a single leader is elected.

Due to these properties, Raft is considered to be fault-tolerant, but not byzantine fault-tolerant [21], [40]. This is because during the election process, two malicious nodes can ignore the timeouts and switch leaders between them, preventing the system from leaving the election phase [41].

All client requests are first stored on the leader's logs and then replicated to other nodes. This is done by having the leader continuously firing events until all followers replicate the data in their logs [40].

Each log contains the command executed, the log's index, and the term in which it was executed. A log is committed by the leader when the majority of nodes in the network successfully synced it to their status. Once the log is committed, the leader executes the command and reports back to the client. Because commands are executed in order, committing a log will also commit any previous one [40].

Inconsistencies are handled by forcing followers to replicate the leader's logs in case of conflicting entries [40].

This system is secured by ensuring that there can be only one leader at any given time, and that its logs are append- only. Additionally, during the election phase, a node will refuse to vote for a candidate whose latest log is older than its own [40].

The minimum number of nodes required to perform an action is 2/n+1, where n is the total number of nodes in the cluster. This means that Raft can tolerate up to half the nodes in the cluster being down [21], [40].

Clique

Clique is a proof-of-authority consensus algorithm that originated as an improvement proposal for Ethereum's testnet in 2017. At the time of Clique's proposal, Ethereum used the proof-of-work consensus mechanism, which made it vulnerable to attacks on networks with low computing capacity. This made creating a testnet, which is an instance of the blockchain used to experiment new features and changes, particularly hard because they were being easily attacked. Clique aims to provide a simple and standardized proof-of-authority mechanism that can be embedded into any Ethereum client, and as such, it is proposed by repurposing already existing Ethereum headers and without resorting to smart contracts [42].

There is a set of trusted nodes named Signers which are responsible for validating transactions and new blocks at fixed intervals. Signers take turns to create blocks, and can also vote to add or remove other signers. Spamming is prevented by restricting the minting frequency of a given signer to 1 out of N/2, where N equals the total number of signers. Although malicious signers can cause some damage, this is limited by the above restrictions, and the malicious signer can be voted out by the remaining ones. The system can also continue working normally up until half its miners stop working, which makes the algorithm both byzantine-fault-tolerant and fault-tolerant [42].

The main problem with this algorithm is that signers are allowed to mint blocks concurrently, which can cause race conditions and create network forks. The problem is made rare by making all signers add a random time offset to when they release a new block, but it does not prevent the problem completely. Small forks can be easy to solve, but large forks can become irresolvable in the network [42], [43].

IBFT

Istanbul Byzantine Fault Tolerant is a consensus algorithm based on the Practical Byzantine Fault Tolerant state machine replication algorithm. It is implemented by modifying Ethereum's headers to include additional information [44].

There are two versions of IBFT supported by the Quorum platform. This section will only focus on version 1.0, which is the one supported by the goquorum. Version 2.0 is only supported by Hyperledger Besu, which superseded the previous one by fixing chain forking issues the original implementation had [45].

IBFT clusters have a set of nodes named validators or leaders, whose responsibility is to determine if a proposed block can be added to the chain. Validators can in turn propose the blocks to be added to blockchain, in which case they are called proposers [44], [46].

In each consensus round, a validator is randomly selected to become a proposer, at which point it must construct a block and share it with the group. The group then votes on the validity of the new block, which requires a super-majority of the votes to be considered valid. A super-majority is obtained with 2F + 1 votes, where F represents the number of faulty nodes in the network. If the block is considered invalid, the proposer changes and a new round begins. Otherwise, the same proposer can continue unless the group votes to change proposer. To ensure only one block is appended to the state machine, blocks are immediately locked after being approved, before their command runs on the chain. This makes the algorithm have immediate finality since it prevents blockchain forks [44], [46].

IBFT algorithm can be represented as a state machine, as shown in Figure 2.4.

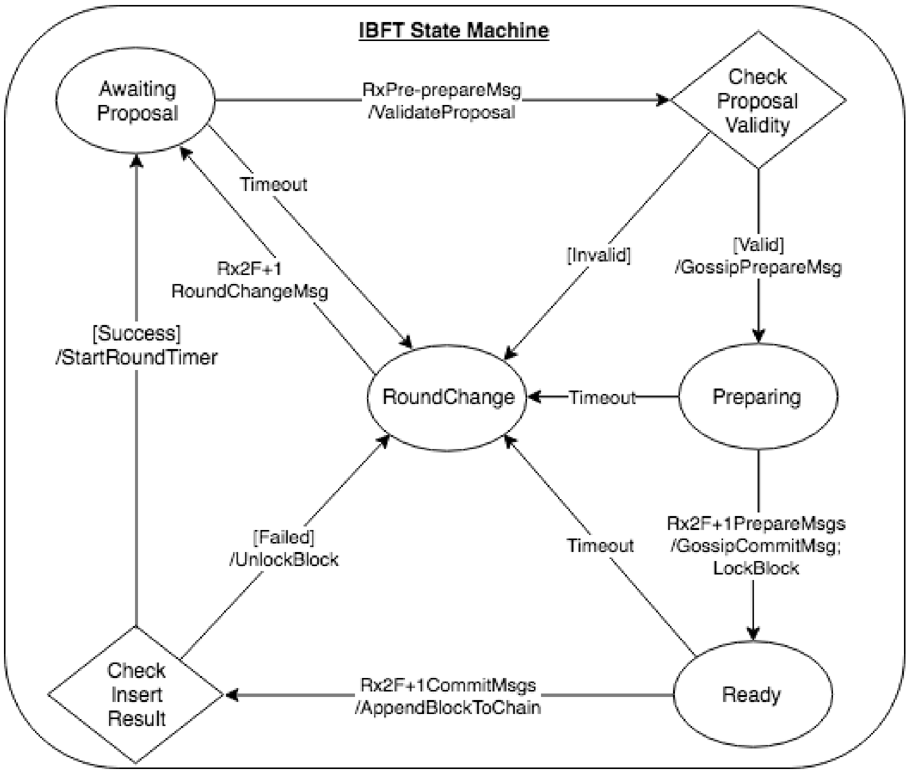


Figure 2.4: IBFT state machine [44]

First, validators wait to receive a proposal from the proposer node. Once received, validators notify their validator-peers and wait for these to notify if they accept the block. Once they receive the notification that their peers have accepted the block, the validators jump into ready state and wait until other validators are in the same state so that they can commit the block. During this time, the block is already locked and can not be replaced. If the round times out or the block somehow failed to be inserted, validators go into the Round Change state, where they must agree on the next round number. If enough validators reach the ready state, then the block is inserted on the chain and validators move back into the Awaiting Proposal state, incrementing the round number [44].

Validators can also vote to add or remove others, where at least N/2 + 1 votes are needed for approval, where N equals the total number of validators in the network [44].

Given the algorithm's behavior, it is considered to be byzantine-fault-tolerant, since malicious validators can be voted out of the network and out of making block proposals. The network implementing this algorithm can continue working properly until up to a third of its nodes are faulty, making it also fault-tolerant [44].

Correctness analyses of the algorithm have found that it is possible for it to enter a deadlock state, making its liveness weak. This happens because there is no process for nodes to unlock from a given block until another block is approved for the same height, because once a block locks on a proposed block for a certain height, it can only vote on that block for that height [47].

A simple scenario to cause this, where F is the number of faulty nodes in the network, is that if a block is proposed and validator A receives 2F+1 notifications of acceptance, it will lock the block. In the same round, if other validators receive less than 2F+1 notifications of acceptance, the round will timeout, and the validators will go into the round change state. In the next round, a new block is proposed, but validator A will be unable to vote on the new block because it is locked on the previous one. If a different validator gets locked on the new block and the round times out again, and there are exactly F faulty validators, it will no longer be possible to get super-majorities in votes, effectively locking the network until validators recover.

QBFT

QBFT is a consensus algorithm developed by ConsenSys to overcome the identified limitations of IBFT. It is a revision of IBFT, and thus functions very similarly to it, following the same three-phase commit strategy, but with five modifications to implementation details, and a bigger modification to ensure liveness.

First, the protocol IBFT uses to transmit finalized blocks is modified to make nodes query their peers regularly on the availability of new finalized blocks. Then, to improve security, nodes now only consider commit-messages signed by the sender. Block verifications are now embedded to pre-prepare messages to enable adding finalization proof the block earlier in the commit chain. Following this, failure to create a finalization proof is removed from the list of conditions that can lead to the round change state. A guarded command is removed from the original specification, as it was considered it provided no value. And, finally, IBFT's block-finalization-protocol is modified to require 2N / 3 validators to obtain a super majority, where N equals the total number of validators in the network [46], [48].

To ensure liveness, the QBFT revision paper makes two suggestions: a PBFT-like solution, or a Tendermint-like solution, both of which are also consensus algorithms.

The PBFT-like solution suggests four modifications. First, remove the locking logic from IBFT. Then, a Prepared Certificate is added to the round change transition message, which includes a set of 2N/3 prepare messages received for the same round of a given instance. If a validator received multiple sets of 2N/3 *prepare* messages for different rounds of the same instance, then its prepared certificate must only include the ones for the highest round number. Finally, a new round message is introduced, which is sent by the proposer once 2N/3 round change messages are received by it [46].

Removing the locking logic effectively solves the liveness issue, and the rest of the modifications are done to ensure safety is not compromised and the network does not fork during round changes [46].

Alternatively, concepts from the Tendermint consensus protocol, namely the concept of *relocking*, can be applied to IBFT to ensure liveness. The specification suggests two modifications to implement this. First, add a locked round value to the pre-prepare message, which represents the latest round number the validator locked on a given block. Allow a validator to relock on a new block if it receives 2N/3 prepare messages for round *r*, one pre-prepare message with the locked round number *r* and *r* is higher than the current round number [46].

The proposal further suggests that both solutions should have little differences in performance as long as the proposer for round 0 is honest and the network delay is less than the round timer.

The solution implemented by ConsenSys for QBFT was based on the Tendermint protocol [49].

2.2 Mapping study

In order to collect and understand the current knowledge of Quorum's consensus algorithms, it was important to conduct a review of literature on the topic. This section describes the review done, detailing both the methodology used the results obtained.

2.2.1 Methodology

There are multiple known strategies to conduct a review of literature. In this instance, the goal is to understand what is currently known about Quorum's consensus algorithms, what has been studied and to what depth. Considering the goal, the most appropriate strategy to adopt is to perform a systematic mapping study, which allows synthesizing information on a field in the form of maps, thus creating a structure and classification system for it. Because the search strategy is explicit, the study can be replicated [50].

The steps to perform a systematic mapping study can be seen in Figure 2.5.

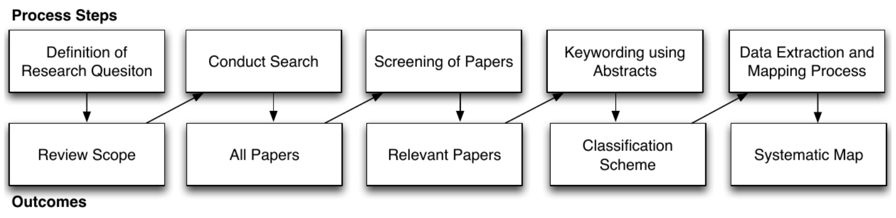


Figure 2.5: Systematic mapping study [44]

Research questions must first be defined in order to define the research scope. Then, research strings must be derived from the research questions and used on the search engines of choice, in order to collect all papers that match them [50].

Once all papers are collected, they must be screened for inclusion and exclusion criteria to get only the relevant papers. A classification scheme must then be created so that the relevant papers can then be sorted into it. With all the data extracted, it can be mapped into frequency tables or bubble plots for better visualization [50].

2.2.2 Research Questions

As mentioned, the goal of the study is to understand what research has been done on Quorum's consensus protocols and in what quantity and depth. As such, the research questions formulated for this study can be seen in Table 2.1.

Table 2.1: Research questions

	Question
RQ1	What performance metrics were explored for each consensus algorithm?
RQ2	What fault-tolerance metrics were explored for each consensus algorithm?
RQ3	What benchmarking frameworks or strategies were used when exploring consensus algorithms?

2.2.3 Search keywords

Keywords to be used on the search engines must be derived from the research questions previously defined.

The search engines chosen can be seen in the list below.

- ScienceDirect [1]
- ACM Digital Library [2]
- IEEE Xplore [3]

Each search engine supports key searching in a slightly different format, so the search syntax must be adapted for each.

[1] https://www.sciencedirect.com/
[2] https://dl.acm.org/
[3] https://ieeexplore.ieee.org/

Based on the research questions, the search terms found in Table 2.2 were created to conduct the search.

Table 2.2: Search keywords for each engine

Search engine	Search
ScienceDirect	permissioned AND blockchain AND consensus AND (algorithm OR protocol OR mechanism)
ACM Digital Library	Abstract:(permissioned AND blockchain AND consensus AND (algorithm OR protocol OR mechanism)) OR Keyword:(permissioned AND blockchain AND consensus AND (algorithm OR protocol OR mechanism)) OR Title:(permissioned AND blockchain AND consensus AND (algorithm OR protocol OR mechanism))
IEEE Xplore	"All Metadata":"permissioned" AND "All Metadata":"blockchain" AND "All Metadata":"consensus" AND ("All Metadata":"algorithm" OR "All Metadata":"protocol" OR "All Metadata":"mechanism")

Because Quorum's consensus algorithms are platform-agnostic and can be implemented on any Ethereum client, the search did not filter down to Quorum blockchain in this phase. The keywords are searched only in the title, abstract, and author-specified keywords of each paper, in order to eliminate articles that only mention the algorithms in passing from the results.

Filtering specifically by "permissioned" blockchain technologies means that some older papers might be filtered out, because the keyword "permissioned" had not yet seen widespread use, and instead the keyword "private" was more common [51]. However, filtering with the "private" keyword resulted in many unrelated papers, and because the algorithms being explored are fairly recent, particularly QBFT, a compromise was reached to exclude these older papers.

2.2.4 Screening of papers

Using the search criteria previously explained, the results in Table 2.3 were obtained.

Table 2.3: Initial search results

Search engine	Number of papers	Date range
ScienceDirect	26	2018–2023
ACM Digital Library	35	2017–2023
IEEE Xplore	140	2016–2023

To further filter the results for relevancy, the inclusion and exclusion criteria found in Table 2.4 were defined.

Table 2.4: Inclusion and exlusion criteria

Inclusion	The abstract explicitly mentions exploring the QBFT, IBFT, Clique and/or Raft protocols.
Exclusion	The paper is not available in its full format or only mentions the consensus algorithms in passing and does not focus on them on its body of work.

By applying the criteria to the results, we obtain the summary of relevant papers found in table .

Table 2.5: Relevant papers result

Search engine	**Number of relevant papers**
ScienceDirect	2
ACM Digital Library	0
IEEE Xplore	4

The vast majority of the initial papers explored different consensus algorithms or proposed new ones, making no references to the ones used in Quorum and Hyperledger Besu, resulting in a total of six relevant papers.

2.2.5 Data extraction

Based on the research questions and knowledge gained by reading the abstracts of the relevant papers, three facets were created to structure the topic. The consensus mechanisms facet will simply represent the consensus algorithms explored in each paper. The Metric explored facet will represent the metrics explored in each paper, detailing the algorithms were explored in terms of performance, resource-usage, security, correctness, and liveness. Correctness and liveness directly affect both fault-tolerance and performance because correctness represents the blockchain's ability to have the correct blocks mined and liveness ensures that as long as a theoretical threshold is not reached, the network will continue processing transactions [52].

Finally, the benchmarking framework facet represents how the data for the exploration of the algorithms were obtained.

Figure 2.6 shows a bubble plot representing the systematic map of the papers into the facets defined.

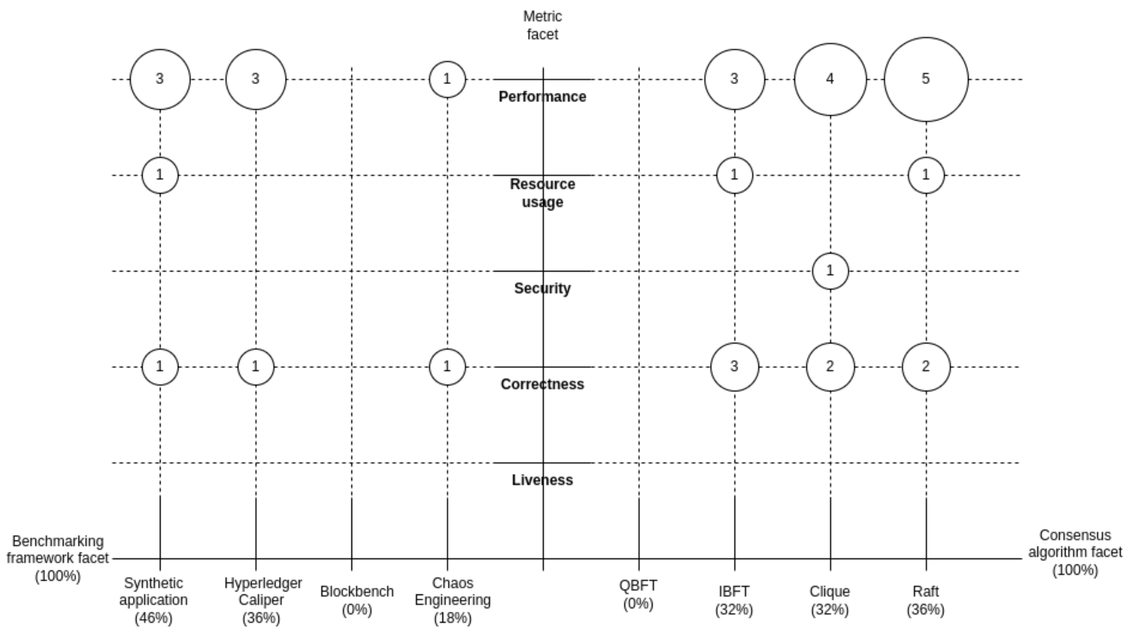


Figure 2.6: Systematic map

From the chart, it can be concluded that there are few papers exploring the consensus algorithms. None explores the QBFT consensus algorithm yet, while the other three algorithms have a similar share of papers published on them.

The most explored metrics are performance and fault-tolerance, while security and resource usage have less data on them. Regarding benchmarking tools, the two most common strategies are deploying a synthetic application and using Hyperledger Caliper. Synthetic applications are instances of a system running a blockchain platform, where tests can be run against them. No papers used Blockbench for benchmarking despite mentioning it when evaluating which framework to use, and one paper used a novel approach of using a synthetic application together with chaos engineering to benchmark it for performance and correctness.

In conclusion, there is very little research published on the topic at the time of writing, particularly for QBFT, which was initially released at the end of 2022. All relevant papers were published after 2020, and so, given their recency, their research and results can help set baselines and expectations for Raft, Clique and IBFTs metrics.

2.3 Benchmarking tools

With the need to test and benchmark systems running on Ethereum-based blockchain platforms, the community has developed several tools and frameworks for benchmarking private networks over the years.

ConsenSys develops quorum-profiling[4] specifically for Quorum. The previous mapping study also revealed Hyperledger Caliper[5] and Blockbench[6] as widely used. In addition to these,

[4]https://github.com/ConsenSys/quorum-profiling
[5]https://github.com/hyperledger/caliper
[6]https://github.com/ooibc88/blockbench

there have been several papers published that detail and showcase novel benchmarking tools for permissioned blockchains. The most notable examples found in a brief search are Chainhammer[7], gromit[8], BCTMark[9], and Diablo[10]. These tools collect different performance indicators and in general have different sets of pros and cons, as shown in Table 2.6.

Table 2.6: Benchmarking tools comparison

Tool	**Success rate**	**Throughput**	**Latency**	**Resource usage**	**Comprehensive documentation**	**Active maintenance**	**Custom workloads**
quorum-profiling	X	X		X	X		
Blockbench	X	X	X	X	X		X
Hyperledger Caliper	X	X	X	X	X	X	X
Chainhammer		X					
gromit	X	X	X	X			
BCTMark	X	X	X	X			
Diablo	X	X	X	X	X		X

ConsenSys quorum-profiling is a toolset that connects to a Quorum network and runs stress tests on it using JMeter, collecting TPS, total transactions and block counts from all nodes. It also collects resource usage information, like CPU and RAM usage, and the results of the JMeter tests being run against the system. All metrics are pushed to an InfluxDB time-series database, and can be queried from a Prometheus[11] endpoint, which can then be used to display the data visually on a Grafana[12] instance.

Blockbench was one of the first frameworks for benchmarking permissioned blockchains, and it supports benchmarking many performance indicators [53]. However, the tool has been mostly superseded by Hyperledger Caliper, which supports the same benchmarks but with better documentation and extensibility, and consequently better community support and engagement.

Hyperledger Caliper is a framework that connects to a system under test (SUT), sends requests to it and then monitors responses, before generating a report with the data collected [54]. Figure 2.7 shows the high-level architecture diagram of the framework.

[7]https://github.com/drandreaskrueger/chainhammer
[8]https://github.com/grimadas/gromit
[9]https://gitlab.inria.fr/dsaingre/bctmark
[10]https://github.com/NatoliChris/diablo-benchmark
[11]https://prometheus.io/
[12]https://grafana.com/

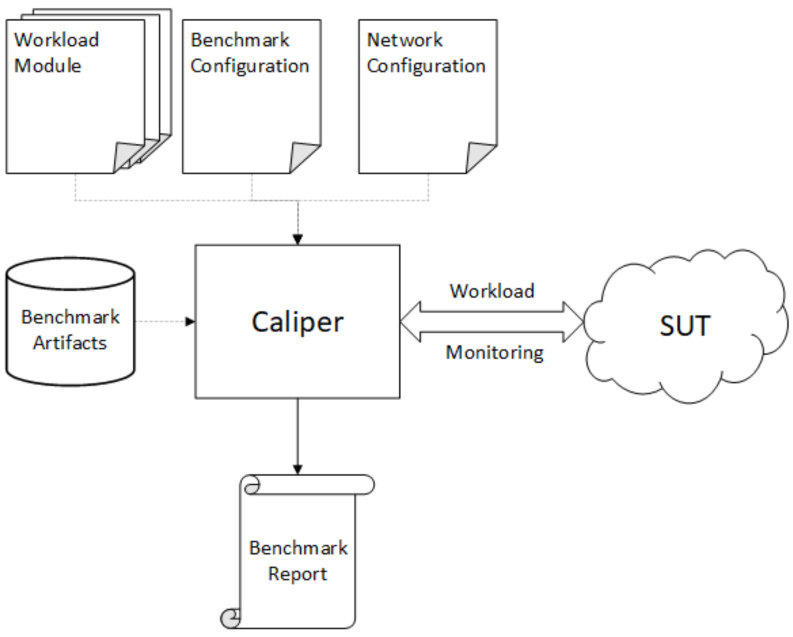


Figure 2.7: Hyperledger Caliper architecture [54]

The framework can be used as a standalone docker image, or it can be embedded into an application using Node.js. Only Hyperledger Besu, Ethereum, Hyperledger Fabric, and FISCO BCOS platforms are supported natively [55]. However, the framework itself is platform agnostic and supports custom connectors to interact with SUTs on different platforms [55]. For example, in [56], an older version of Caliper was extended to support benchmarking Quorum.

Chainhammer focuses solely on transactions per second (TPS) and measures internal blockchain metrics like gas used, block time and block size. It is not being actively maintained, with the latest functional commit being three years ago, and it also lacks comprehensive documentation [57].

Gromit is based on another tool called Gumby. It supports multiple blockchain platforms, which include Quorum, and provides a set of experiments that can be run and customized [58]. However, documentation is lacking because the author pointed to Gumby's original documentation, which is no longer available on GitHub. Additionally, it has not seen recent activity, with the latest functional commit being over a year ago, around the time the paper detailing it was published.

BCTMark was developed in 2020 and also collects all the relevant metrics for this work [59]. However, there is no documentation for it outside of the initial release paper, and there is no activity in its repository for three years, indicating it has been abandoned.

The paper presenting and describing Diablo was released in May of 2023, but the source code for the project indicates that it was developed over 2 years ago. The tool supports collecting all the performance metrics and resource usage, and it sets itself apart by providing real-world workloads for the blockchain, including complex video-games, Youtube, and Uber. The tool also supports adding custom workloads and has detailed documentation on how it works and how to run it. However, the source code has not been updated in over 2 years, and for Quorum only the IBFT algorithm is supported natively [60]. Support for the rest of the algorithms could theoretically be added by modifying the source code, but the tool is written in the Go programming language[13], which the author is not familiar with, making it unfeasible to use given the project's time constraints.

[13]https://go.dev/

Chapter 3

Experimentation design

In this chapter, the methodology for the benchmarking is described, followed by the design and architecture of the implementation.

3.1 Methodology

The main objective of comparing consensus algorithms requires benchmarking the network under different scenarios, in a way that promotes speed of development, fairness, data reliability, and transparency. As such, a methodology to follow was drafted, which can be seen in Figure 3.1.

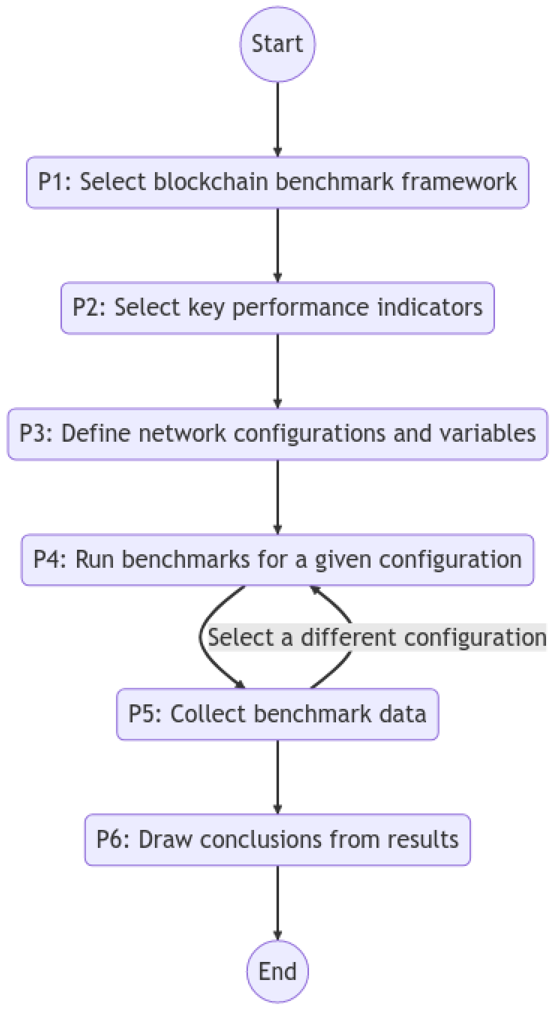

Start
P1: Select blockchain benchmark framework
P2: Select key performance indicators
P3: Define network configurations and variables
P4: Run benchmarks for a given configuration
Select a different configuration
P5: Collect benchmark data
P6: Draw conclusions from results
End

Figure 3.1: Flowchart of the benchmarking methodology

First, a blockchain benchmark framework compatible with the system must be chosen. This should be done based on a study of the current solutions and what features they provide.

Then, the key performance indicators that will be measured must be selected, followed by network and workload configurations. In these phases, it is important to keep the goal and scope of the project present so that only relevant variable system parameters and measured metrics are included.

Once these are defined, the benchmarks must be run for each combination of network variables in order to collect all the relevant data, which in turn will be used to compare the consensus algorithms and draw conclusions.

The sections that follow go over each phase of the methodology as it was applied.

3.2 Benchmark framework

As explored in the previous chapter, there are multiple tools that can be used to benchmark a quorum network. The one chosen for this project is Hyperledger Caliper because it supports all the metrics needed, its documentation is more extensive and complete than the alternatives, it has more community support and active development, being supported by the Hyperledger community, and is written in a programming language known to the author. Figure 3.2 shows the multiple layers of the framework.

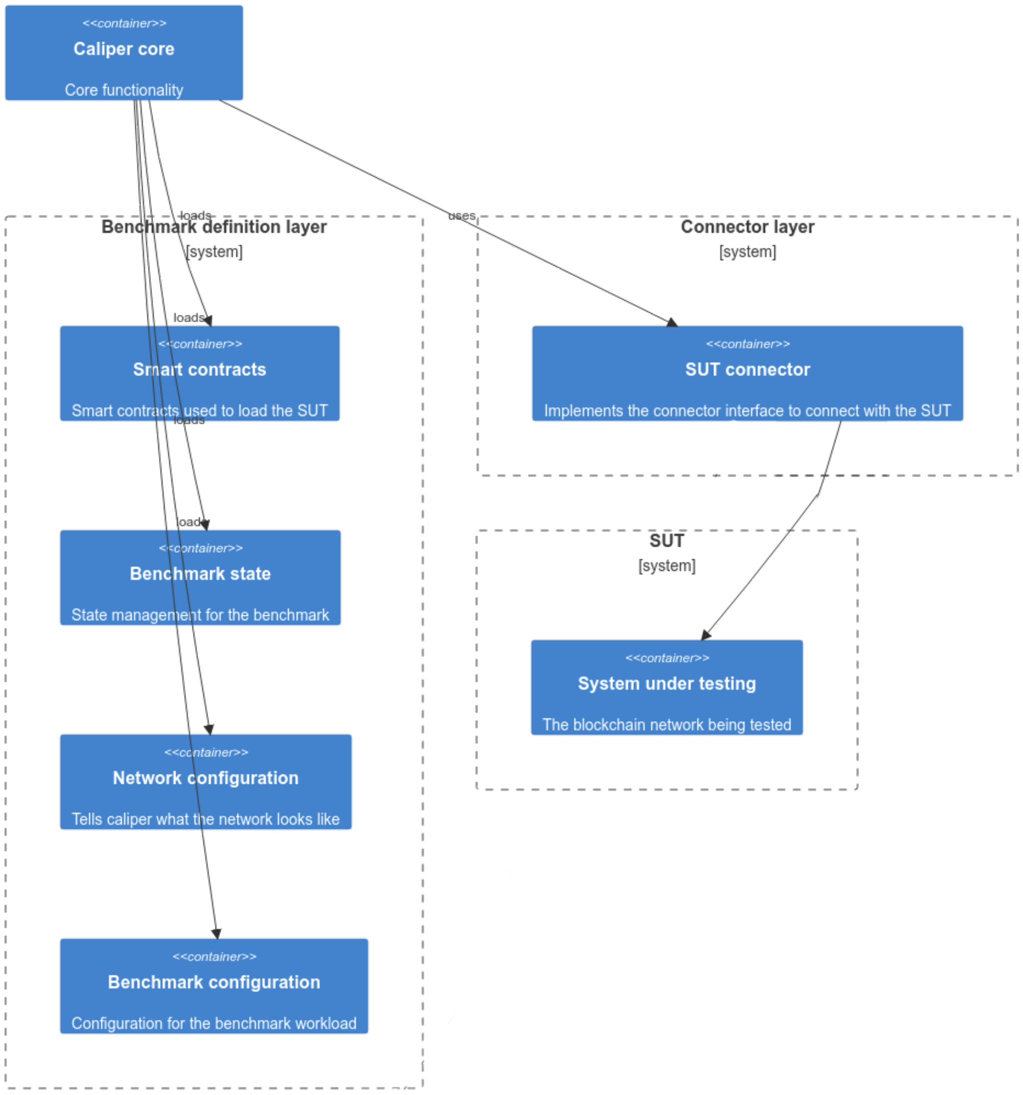


Figure 3.2: Hyperledger Caliper layers

As previously mentioned, Hyperledger Caliper does not support the GoQuorum client by default. However, because it is designed to be extended, a custom connector can be created for it and added to the connector layer, which will allow the framework's core to interact with the system under test [61].

Additionally, because GoQuorum is based on Ethereum, and Hyperledger Caliper is open-source, the Ethereum connector that is provided by Caliper can be used as the base for the custom one for Quorum. The source code for Caliper's Ethereum connector can be found in their GitHub repository[1], and it follows the interface shown in Listing 4.1.

The Ethereum connector already has support for private transactions, but only for the Hyperledger Besu client. This is done using the *web3-eea*[2] package, but it has since been deprecated in favor of *web3js-quorum*[3], both developed by ConsenSys. To ensure compatibility with the latest version of Quorum and facilitate development, the new connector will be built on top of the Ethereum one, modifying it to use the latest version of *web3s-quorum* and all the changes required for it.

3.3 Key performance indicators

The project has the aim of measuring the performance and fault-tolerance of each consensus algorithm in Quorum, and thus the key performance indicators are all related to these two metrics.

The goal, question, and metric (GQM) approach defines a measurement model that can be used to establish the goals and metrics to measure them. This model has three distinct levels: conceptual level, operational level, and quantitative level [62].

The conceptual level defines the goal for the project, which, as defined in 1.3, is to explore the effect that different consensus algorithms have on fault tolerance and performance in Quorum.

Based on this goal, the operational level details a set of questions that map to concrete components of the system being evaluated. The questions previously defined are:

- **Q1** What is the effect of the consensus algorithm on performance?
- **Q2** What is the effect of the consensus algorithm on fault-tolerance?

The Hyperledger Performance and Scale Working Group (PSWG) has proposed a set of basic terms and metrics to serve as a standard when evaluating the performance of blockchains and distributed ledger technologies (DLTs) [63].

Latency and throughput for reads and transactions are identified as the main metrics to assess the system's performance. In addition to those, resource consumption and scalability will also be measured to assess how the system behaves under different needs, resulting in the following list of metrics:

- M1: Transaction throughput.
- M2: Read throughput.
- M3: Transaction latency.
- M4: Read latency.

[1]https://github.com/hyperledger/caliper/tree/main/packages/caliper-ethereum
[2]https://github.com/ConsenSys/web3js-eea
[3]https://consensys.github.io/web3js-quorum/latest/index.html

- M5: Resource consumption.
- M6: Scalability: relationship between number of nodes and transaction throughput.

As for fault-tolerance, the only metric that will be monitored is:

- M7: Transaction success rate.

As for fault-tolerance, the only metric that will be monitored is the transaction success rate.

With the goal, questions, and metrics defined, the resulting GQM map can be seen in Figure 3.3.

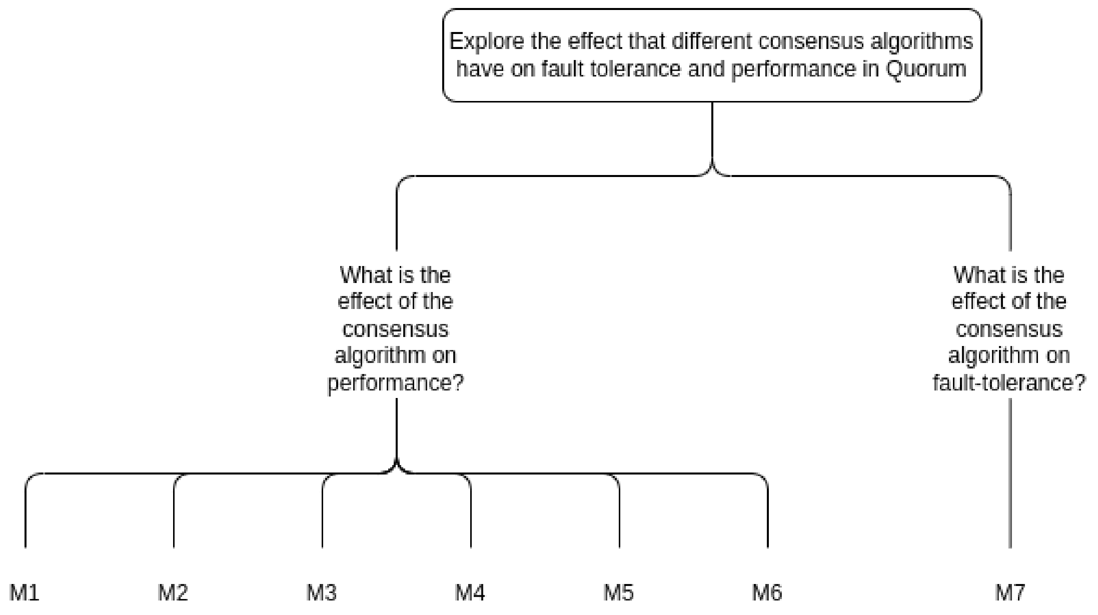

Explore the effect that different consensus algorithms have on fault tolerance and performance in Quorum
What is the effect of the consensus algorithm on performance?
What is the effect of the consensus algorithm on fault-tolerance?
M1
M2
M3
M4
M5
M6
M7

Figure 3.3: GQM map

A transaction is considered a network state transition, where some data in the blockchain was changed to a new value. Transactions are submitted by clients and evaluated by the system, which will commit them if it considers them valid. Invalid transactions are not taken into account for latency and throughput metrics because they are not considered meaningful by Hyperledger [63].

Transaction throughput is the rate at which valid transactions are committed by the SUT in a given time frame, thus representing the rate of the entire network, and not of each individual node [63].

$$\textit{Transaction throughput} = \frac{\textit{Total transactions committed}}{\textit{Total time in seconds}} \tag{3.1}$$

Transaction latency is also measured at the network level, and represents the amount of time it takes for a transaction to go from being submitted by a client to having its result available across the network. This means that the consensus mechanism's propagation and settling times are included in the latency.

$$\textit{Transaction latency} = \textit{Confirmation time x network threshold} - \textit{Request submission time} \tag{3.2}$$

The network threshold is basically the percentage of the network's processing power that needs to have acknowledged the transaction. In this project, since all consensus algorithms are deterministic, the threshold will always be considered 100%.

Reads, on the other hand, are measured on the clients directly, and thus are much simpler to define. Read throughput represents how many read operations are completed within a given time-frame.

$$\textit{Read throughput} = \frac{\textit{Total read operations}}{\textit{Total time in seconds}} \tag{3.3}$$

Read latency, consequently, is simply the difference between the time when the request was sent and the time the response was received.

$$\textit{Read latency} = \textit{Response received time} - \textit{Request submission time} \tag{3.4}$$

It is important to note that for each latency metric the average, maximum, and minimum values will be gathered and compared.

Resource consumption will look into the CPU and RAM usage of each GoQuorum node over the course of the benchmark. Hyperledger Caliper measures these metrics for each container in the network. To get a single measurement for the whole network, the highest average CPU and RAM usage value measured for the run will be used. This will be done instead of calculating the average for all nodes because, if one node is having much higher resource consumption than the rest, then it can be a bottleneck for the whole network.

Scalability will be measured by analyzing how the transaction throughput and latency evolve with the number of nodes in the blockchain. As the number of nodes increases, each transaction will need to propagate across more nodes, and thus it is expected that the throughput will do down and the latency up.

Fault tolerance will be measured by introducing a 10-second delay to network requests to validator nodes in the system at random and evaluating if transaction processing, throughput, and latency are affected. This will be achieved by using pumba[4], a chaos testing tool for Docker[5] that allows interacting with contains at random in a docker network using the docker socket. The pumba service entry and command that allows this can be seen in Listing 3.1.

```
chaos-delay:
  image: gaiaadm/pumba
```

[4]https://github.com/alexei-led/pumba
[5]https://www.docker.com/

```
    volumes:
      - /var/run/docker.sock:/var/run/docker.sock
    command: "--log-level debug --interval 20s --random netem --tc-image gaiadocker/
    iproute2 --duration 10s delay re2:^network-validator"
```

Listing 3.1: Pumba pause service

Each test run will then have a scenario where validator nodes being paused at random, and another where all containers are stable.

3.4 Network and environment configuration

The network and environment configurations impact the system's performance, and thus it is important to determine which parameters will be treated as constants and which ones will serve as variables for the benchmarks. This includes configuration parameters of the Quorum system itself and decisions on how the network and benchmark environment is set up when building the test environment.

Starting with the system, there are multiple configurations outside the consensus algorithm that can affect its performance, such as the gas limit or block mining period. However, because the goal of this project is only to evaluate the consensus algorithms, all these configurations will be treated as constants. Table 3.1 shows the values used for each of Quorum's network parameters in its genesis file[6], and which consensus algorithms they apply to.

Table 3.1: Quorum network configuration constants

Property	**Algorithms**	**Value**
txnSizeLimit	All	64 (kb)
difficulty	All	0x1
gasLimit	All	0xFFFFFF
epoch	Clique, IBFT, QBFT	30000
policy	Clique, IBFT, QBFT	0
ceil2Nby3Block	IBFT, QBFT	0
blockperiodseconds	IBFT, QBFT	5
emptyblockperiodseconds	IBFT, QBFT	60
requesttimeoutseconds	IBFT, QBFT	10
period	Clique	10

All the values are the default ones when using *quorum-genesis-tool*[7] to generate a GoQuorum network. It is also important to note that this project used the latest versions of Quorum and Tessera available at the time of writing, which is version 23.4.0 for both.

As for the network and environment setup, three main alternatives were identified:

- A: Run all nodes locally.

[6]https://docs.goquorum.consensys.net/configure-and-manage/configure/genesis-file/genesis-options
[7]https://github.com/ConsenSys/quorum-genesis-tool

- B: Run all nodes on a single cloud instance.
- C: Divide nodes across multiple cloud distances.

Alternative A is free and very simple to set up, but limits the reproducibility of the benchmark results while also affecting the results directly if the machine does not have enough CPU threads to dedicate one per node.

Alternative B is also simple to set up, only requiring a few more extra steps for configuring the cloud instance and connecting remotely to it, while also improving benchmark reproducibility. However, it incurs some monetary costs since most cloud providers will charge for instances with enough resources to run the benchmarks comfortably.

Alternative C also guarantees reproducibility, and is better at simulating an actual real-world scenario of a network where the nodes are distributed across different geographical locations. However, it is much harder to set up since it involves setting up communication across multiple cloud instances, and it also requires running more benchmarks because network latency becomes a variable outside the tester's control. It also incurs higher monetary charges since it involves multiple cloud instances.

Taking these alternatives into consideration, option B was chosen as a good compromise between cost, reproducibility, and speed of development. For the cloud provider, an AWS EC2[8] instance was chosen, since the author already has experience working with this provider, speeding up the setup process.

The EC2 instance type is *c5a.8xlarge*, which is configured with the values found in Table 3.2.

Table 3.2: EC2 instance configuration

Property	**Value**
Operating System	Ubuntu 22.04
vCPUs	32
CPU clock speed	2.8Ghz
RAM	64Gb

Inside the instance, nodes will run inside Docker containers. This technology was chosen because of its *compose* tool[9], which greatly simplifies the process of tearing down and setting up a new network to run the benchmarks.

Regarding TLS usage, this protocol is already known to introduce performance overhead to all requests, while also slightly increasing CPU usage [64]. Since TLS overhead is introduced per request, the more requests a network has, the more noticeable this overhead will be, which means that this latency is expected to increase as the number of nodes in a blockchain network increases. Consequently, it might be interesting to explore how this overhead relates to other variables, namely if different consensus mechanisms result in more or less network requests. However, this exploration falls outside this project's scope because manually setting up TLS for so many nodes is complex and time-consuming, and many companies and systems in production do not use TLS for communication internal to their network. Regardless, it

[8]https://aws.amazon.com/ec2/
[9]https://docs.docker.com/compose/

could be explored in future works, perhaps more focused on the security aspects of these systems.

Regarding the number of benchmark workers, Hyperledger Caliper will use four workers to run transactions against the system, in order to simulate multiple clients sending requests at the same time. Each worker will thus simulate one client. Caliper supports this natively and divides the configured transactions and transactions per second across all workers. This means that increasing the number of workers, which make the requests in parallel, does not have an impact on the number of transactions per second sent, thus ensuring the transactions per second are the ones explicitly configured. Furthermore, Caliper allows each worker to connect to a node at random by using the *fromAddressSeed* property, which uses BIP-44 key derivation to generate an address for each worker [65].

In an enterprise scenario, it is expected that only a subset of nodes will be exposed to clients, and the network needs to propagate these requests across all its members. Thus, four clients is a good middle ground to simulate this enterprise scenario and test how each consensus algorithm is affected by different transactions being propagated across the network in parallel.

Finally, the number of nodes will be a variable in the benchmark in order to assess how each consensus algorithm scales. All consensus algorithms being tested require at least four validator nodes to achieve high availability. As such, the system will initially be configured with seven nodes: three members and four validators. Then, in order to horizontally scale the network, member nodes will be increased in steps of one, up to 6, and validator nodes will be increased in steps of four, up to 16. This will result in networks of 7, 12, 17, and 22 nodes. The system is not scaled any further because past 6 member nodes because the number of errors when processing transactions becomes too high, making it pointless to assess its performance since it is not feasible to deploy such a network in a production environment. The main reason for this lack of scaling is that the Tessera private transaction manager is a bottleneck on the system, as it will be explored in chapter 4.

In addition to member and validator nodes, an RPC node will also be present in the network to interact with the blockchain.

3.5 Workload

Hyperledger provides a repository with benchmark examples compatible with multiple systems, which can be found in [66]. There are two benchmark scenarios available that simulate a simple bank[10]. One called *smallbank* targets Hyperledger Fabric, and another called *simple* targets Ethereum. Despite their names, the operations they perform are identical, and the only changes are in the targeted systems.

Because Quorum extends Ethereum, the *simple* scenario was chosen as the workload for the benchmarks since it is the most compatible one. In this scenario there are three different smart contracts used:

- *open*: write operation that creates a bank account, which is a randomly generated string with an amount of money associated.

[10]https://github.com/hyperledger/caliper-benchmarks/tree/main/benchmarks/scenario

- *query*: read operation that, given an account string, returns the amount of money associated with it.
- *transfer*: read-write operation that transfers money across two accounts.

This is a good benchmark workload because it supports the three possible types of operations, it allows parallel requests, and exemplifies a finance use case, which is the most common one for permissioned blockchains. However, as it is, it only supports Ethereum public transactions, and so it must be modified to also support private Quorum transactions.

Alternatives to using this benchmark could involve creating a new scenario from scratch, but the extra work was not considered worth it when the existing one already covers all the benchmark needs. Creating a more complex benchmark would also bring no benefits to the goal of comparing the consensus algorithms, since a more complex smart contract logic would only impact running the code in the EVM. Implementing a scenario for an industry outside of finance was also not considered necessary because all three operation types are being measured, which will allow interested parties to extrapolate the results and conclusions to their use cases.

Regarding how many requests will be sent and at which rate, Hyperledger Caliper allows configuring both of these numbers using the properties *txNumber* to define the number of transactions to be sent of each smart contract and *tps* to define the number of transactions per second. The transactions per second property will be incremented across test runs to assess if different consensus algorithms can handle specific request rates better than others. The number of transactions will start at 50, then increase to 100, 200, and finally 300. Similar to what happens with network scaling, there is no point in increasing the TPS past 300 because the system is unable to handle them and results in most transactions failing.

Listing 3.2 shows an example of a benchmark configuration file used in the project for the smart contracts previously shown.

```yaml
bankArgs: &bank-args
  initialMoney: 10000
  moneyToTransfer: 100
  numberOfAccounts: &number-of-accounts 1000

test:
  name: bank
  workers:
    number: 1
  rounds:
    - label: open
      description: >-
        Test description for the opening of an account through the deployed
        contract.
      txNumber: *number-of-accounts
      rateControl:
        type: fixed-rate
        opts:
          tps: 50
      workload:
        module: benchmarks/scenario/bank-private/open.js
        arguments: *bank-args
    - label: query
      description: Test description for the query performance of the deployed contract.
      txNumber: *number-of-accounts
      rateControl:
        type: fixed-rate
        opts:
          tps: 100
      workload:
        module: benchmarks/scenario/bank-private/query.js
        arguments: *bank-args
    - label: transfer
      description: Test description for transferring money between accounts.
      txNumber: 50
      rateControl:
        type: fixed-rate
        opts:
          tps: 5
      workload:
        module: benchmarks/scenario/bank-private/transfer.js
        arguments:
          << : *bank-args
          money: 100
```

Listing 3.2: Benchmark configuration example

As it can be seen, transactions per seconds and the total number of transactions can be configured globally using a variable or per smart contract.

3.6 Overview

Table 3.3 shows a summary of the main benchmark properties and how variables will be incremented.

Table 3.3: Benchmark variables summary

Property	**Value**
Number of validators	4–16, steps of 4
Number of members	3–6, steps of 1
Number of benchmark workers	4
Consensus mechanisms	Clique, IBFT, RAFT, QBFT
Transaction rate (tps)	50,100–300, steps of 100
Number of transactions	5000
Transaction type	Public and Private
Chaos testing	Disabled and Enabled

Figure 3.4 shows the system architecture, using UML notation[11].

[11] https://www.omg.org/spec/UML/2.5.1/About-UML#inventory-links

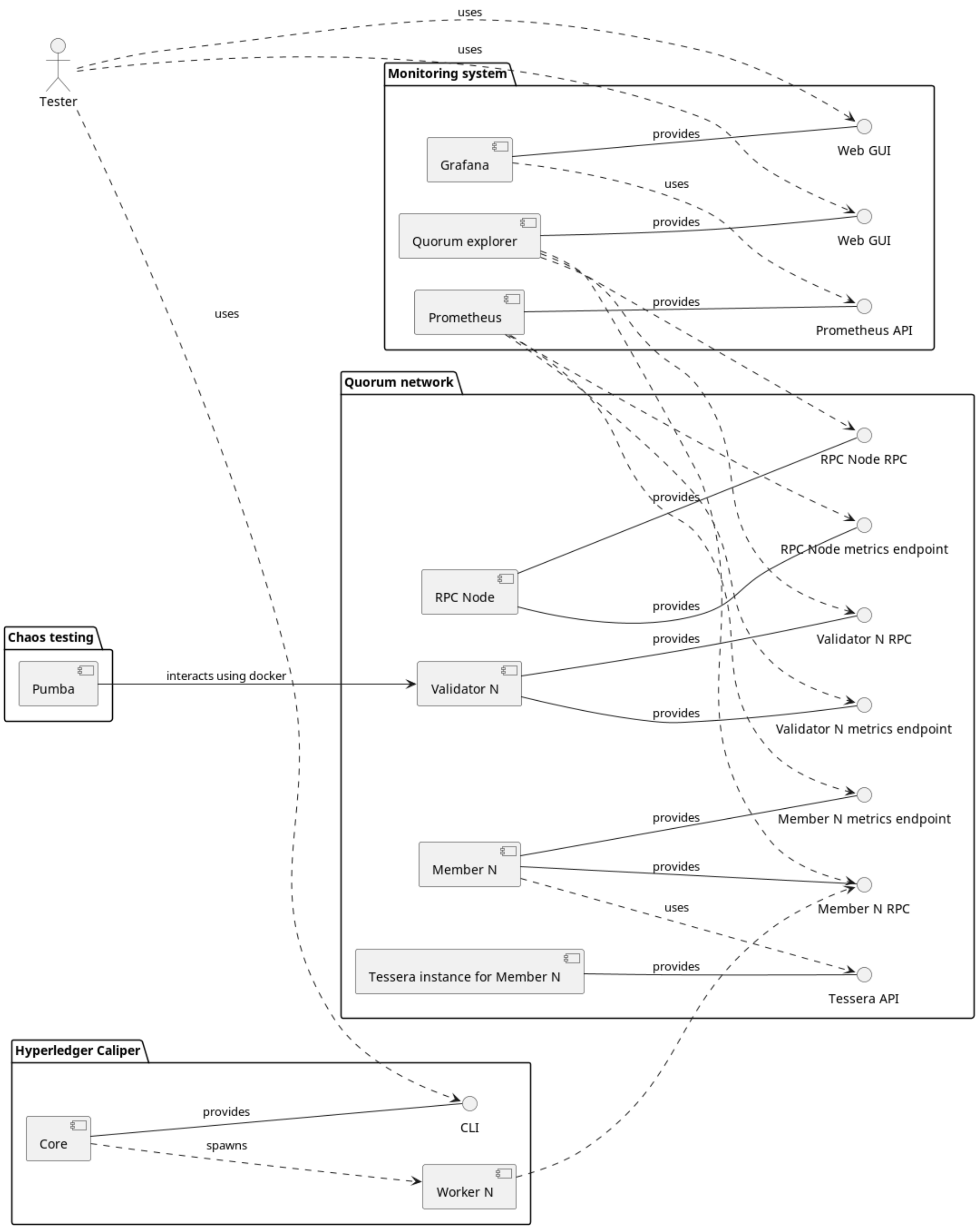


Figure 3.4: Benchmarking system architecture

Inside a docker network are two main boundaries: the monitoring system and the SUT itself.

The SUT is a Quorum network which will have, in each benchmark, the pre-configured number of nodes. All member nodes have a corresponding Tessera transaction manager in order to support private transactions.

Then, the monitoring system consists of a Prometheus [67] instance which will query each node's metrics endpoint to fetch information exported, a Grafana [68] instance to present visualizations on top of this information, and a Quorum Explorer instance to explore the network.

Prometheus is an open-source systems monitoring toolkit that pulls data from running instances and stores them in real-time. Grafana, in turn, is an open-source data visualization platform that can fetch data from Prometheus and present it in multiple dashboards with custom queries. ConsenSys has made available a pre-configured Grafana dashboard to GoQuorum nodes that export Prometheus metrics on a pre-configured endpoint [69].

Quorum Explorer is a simple webpage that connects to the system using RPC and allows developers to visualize the system in real time and interact with it.

As previously mentioned, one of the metrics to be measured is resource usage, which is supported by Hyperledger Caliper via Prometheus or via Docker [70]. Since all the benchmark services are running on a single Docker network and the Caliper workers are also local to it, the Docker monitoring will be used, since it is fully automated to configure, whereas the Prometheus one requires writing each query individually. As such, resource usage results will be derived from this monitoring, and Prometheus, Grafana, and Quorum Explorer will only be used during the development phase to ensure the benchmarks are being run correctly and transactions are being added to the chain successfully.

Finally, pumba will be used as a chaos testing tool to randomly introduce network delays and prevent validator nodes from sending requests. Delays will be run every 20 seconds and have a duration of 10 seconds, effectively causing a validator node to go temporarily offline for the duration.

Chapter 4

Benchmark implementation

In this chapter, the implementation of the system that was used for benchmarks is initially described. Then, the Quorum connector implemented for Hyperledger Caliper is presented, and, finally, the workload used in Caliper to run against the system is also described.

4.1 Hyperledger Caliper Quorum connector

As mentioned in chapter 3, Hyperledger Caliper does not support GoQuorum as a target platform by default. However, it does support an extensible connector interface that can be implemented to support any platform. Listing 4.1 shows the connector interface exported by the *@hyperledger/caliper-core* package, for *Node.js*, that must be implemented to write a custom connector.

```
class ConnectorInterface extends EventEmitter {
    getType() {}
    getWorkerIndex() {}
    async init(workerInit) {}
    async installSmartContract() {}
    async prepareWorkerArguments(number) {}
    async getContext(roundIndex, args) {}
    async releaseContext() {}
    async sendRequests(requests) {}
}
```

Listing 4.1: Caliper connector interface

In essence, the connector handles three tasks: deploying public and private smart contracts to the system, sending both public and private transactions, and storing the benchmark's state.

Since GoQuorum is built on top of Ethereum, which is supported by the framework, the connector for it can be adapted to also support GoQuorum. Hyperledger Caliper connector for Ethereum can be found in Appendix B.

This connector already has support for Hyperledger Besu private transactions using the *web3-eea* package developed by ConsenSys, however, this package has been deprecated since January 2022 and was effectively replaced by *web3js-quorum*. Documentation for *web3js-quorum* fully details the migration process from *web3-eea* to the new package, which can be found in [71].

As such, implementing the connector started by copying the Ethereum connector to a new file and then following the migration guide to have it use the new package. Once this was done, each connector task had to be slightly adapted to fully support GoQuorum.

4.1.1 Contract deployment

the deployment of private transactions had to be adapted to support all the fields introduced by the new library and required by the GoQuorum client. Deploying private contracts consists of the following steps:

1. Get the contract deployer account.
2. Create the payload with the transaction data and privacy settings.
3. Send the transaction and wait for a successful response.

In the first step, a random deployer account is created. This is done instead of using the same account to deploy all contracts in order to avoid nonce issues when deploying both public and private contracts.

In the second step, first, the current nonce for the account is fetched by counting the number of transactions for the deployer account. Then, the privacy configuration for the contract is fetched by defining the *privateFor* and *privateFrom* fields. To simulate the worst case scenario for every network configuration, the *privateFrom* field is set to the deployer node, and the *privateFor* field specifies all member nodes in the network, which means that every member must decode each transaction using their Tessera instance.

In addition to these, the payload also defined the gas price and limit, the deployer account as the transaction sender, and the contract's bytecode as its data.

Finally, with the payload the defined the payload is sent. If the deployment is successful, the benchmark continues; otherwise it fails by throwing an error. The code that handles this can be seen in Listing 4.2.

```
async deployPrivateContract(contractData, privacy) {
    const web3 = this.web3;
    // Using a randomly generated account to deploy private contract to avoid public/
    private nonce issues
    const deployerAccount = web3.eth.accounts.create();
    const txCount = await web3.eth.getTransactionCount(`${deployerAccount.address}`);

    const transaction = {
        data: contractData.bytecode,
        nonce: txCount,
        gasPrice: 0,
        gasLimit: 0x24a22,
        value: 0,
        from: deployerAccount,
        isPrivate: true,
        privateKey: deployerAccount.privateKey,
        privateFrom: privacy.privateFrom,
        privateFor: privacy.privateFor
    };

    try {
        const txHash = await web3.priv.generateAndSendRawTransaction(transaction);
        return new web3.eth.Contract(contractData.abi, txHash.contractAddress);
    } catch (err) {
        logger.error('Error deploying private contract: ', err.stack);
        throw(err);
    }
}
```

Listing 4.2: Deploy private contract method

As for public contracts, these are identical in GoQuorum and Ethereum, and thus deploying these and sending public transactions is supported without any changes required. The code for deploying public contracts can be found in Listing 4.3.

```
async deployContract(contractData) {
    const web3 = this.web3;
    const contractDeployerAddress = this.ethereumConfig.contractDeployerAddress;
    const contract = new web3.eth.Contract(contractData.abi);
    const contractDeploy = contract.deploy({
        data: contractData.bytecode
    });

    try {
        return contractDeploy.send({
            from: contractDeployerAddress,
            gas: contractData.gas
        });
    } catch (err) {
        throw(err);
    }
}
```

Listing 4.3: Deploy public contract method

The pre-configured deployer address is used to deploy the contract using its bytecode gas required data. If the contract deployment fails, the benchmark stops by throwing an error.

4.1.2 Sending transactions

Sending a private transaction consists of encoding the transaction to ABI, then fetching the sender and privacy configuration to generate the payload and then send the transaction to the network. The transaction's success status is then stored in the benchmark's state to generate the final report. The code that handles this can be seen in Listing 4.4.

```
async _sendSinglePrivateRequest(request) {
    const context = this.context;
    const web3 = context.web3;
    const contractInfo = context.contracts[request.contract];
    const privacy = request.privacy;
    const sender = privacy.sender;

    const status = new TxStatus();

    const onFailure = (err) => {
        status.SetStatusFail();
        logger.error(`Failed private tx on ${request.contract}; calling method: ${request
    .verb}; private nonce: ` + 0);
        logger.error(err);
    };

    const onSuccess = (rec) => {
        status.SetID(rec.transactionHash);
        status.SetResult(rec);
        status.SetVerification(true);
        status.SetStatusSuccess();
    };

    let payload;
    if (request.args) {
        payload = contractInfo.contract.methods[request.verb](...request.args).encodeABI
    ();
    } else {
        payload = contractInfo.contract.methods[request.verb]().encodeABI();
    }

    const transaction = {
        to: contractInfo.contract._address,
        data: payload,
        gasPrice: 0,
        gasLimit: 0x24a22,
        value: 0,
        isPrivate: true,
        nonce: sender.nonce,
        privateKey: sender.privateKey,
        from: sender,
        privateFrom: privacy.privateFrom,
        privateFor: privacy.privateFor
    };

    try {
        const result = await web3.priv.generateAndSendRawTransaction(transaction);
        if (result.status) {
            onSuccess(result);
        } else {
            onFailure(result);
        }
    } catch (err) {
        onFailure(err);
    }

    return status;
}
```

Listing 4.4: Send private transaction method

Similarly to contract deployments, the sender account is created automatically for each request in order to avoid nonce issues when using multiple workers. The transaction receiver is also randomly selected from a list of accounts that is predefined in the benchmark.

All private transactions are marked to be sent from the same pre-configured member and for all other members, which is the worst case scenario for the network, since all members will need to decode the transaction.

For public transactions, the sender's address is automatically generated from a seed using BIP-44 key derivation[1]. This works by pre-configuring the field *fromAddressSeed* and then fetching a wallet from this seed using the derivation path *m/44'/60'/<workerIndex>'/0/0*, where *workerIndex* ranges from 1 to 4 and is unique for each worker. This effectively allows each worker to have their own sender account, which also prevents eventual nonce issues.

The code that handles processing the seed can be seen in Listing 4.5.

```
let hdwallet = EthereumHDKey.fromMasterSeed(this.ethereumConfig.fromAddressSeed);
let wallet = hdwallet.derivePath('m/44\'/60\'/' + this.workerIndex + '\'/0/0').getWallet
    ();
context.fromAddress = wallet.getChecksumAddressString();
context.nonces[context.fromAddress] = await this.web3.eth.getTransactionCount(context.
    fromAddress);
this.web3.eth.accounts.wallet.add(wallet.getPrivateKeyString());
```

Listing 4.5: From address seed handler

Other than the sender's address, the only other fields specified are the gas price, gas required, and chain id. The gas price and chain id are relevant to the network itself and could be fetched in real time if not specified here, but this would cause transactions to be reordered, possibly resulting in nonce failures.

The gas required for the contract is estimated using web3 and then added 1000, to ensure that there's enough for the operation.

The code that handles sending public transactions can be seen in Listing 4.6.

[1]https://github.com/bitcoin/bips/blob/master/bip-0044.mediawiki

```
async _sendSingleRequest(request) {
    const context = this.context;
    let status = new TxStatus();
    let params = {from: context.fromAddress};
    if (request.hasOwnProperty('value') && request.value > 0) {
        params.value = request.value;
    }
    let contractInfo = context.contracts[request.contract];
    let receipt = null;
    let methodType = 'send';
    if (request.readOnly) {
        methodType = 'call';
    } else if (context.nonces && (typeof context.nonces[context.fromAddress] !== 'undefined')) {
        let nonce = context.nonces[context.fromAddress];
        context.nonces[context.fromAddress] = nonce + 1;
        params.nonce = nonce;
        params.gasPrice = context.gasPrice;
        params.chainId = context.chainId;
    }
    const onFailure = (err) => {
        status.SetStatusFail();
        logger.error(`Failed tx on ${request.contract}; calling method: ${request.verb}; nonce: ${params.nonce}`);
        logger.error(err);
    };
    const onSuccess = (rec) => {
        status.SetID(rec.transactionHash);
        status.SetResult(rec);
        status.SetVerification(true);
        status.SetStatusSuccess();
    };
    if (request.args) {
        if (contractInfo.gas && contractInfo.gas[request.verb]) {
            params.gas = contractInfo.gas[request.verb];
        } else if (contractInfo.estimateGas) {
            params.gas = 1000 + await contractInfo.contract.methods[request.verb](...request.args).estimateGas();
        }
        try {
            receipt = await contractInfo.contract.methods[request.verb](...request.args)[methodType](params);
            onSuccess(receipt);
        } catch (err) {
            onFailure(err);
        }
    } else {
        if (contractInfo.gas && contractInfo.gas[request.verb]) {
            params.gas = contractInfo.gas[request.verb];
        } else if (contractInfo.estimateGas) {
            params.gas = 1000 + await contractInfo.contract.methods[request.verb].estimateGas(params);
        }
        try {
            receipt = await contractInfo.contract.methods[request.verb]()[methodType](params);
            onSuccess(receipt);
        } catch (err) {
            onFailure(err);
        }
    }
    return status;
}
```

Listing 4.6: Send public transaction method

Like in private transactions, if the request fails, the transaction status is marked as a failure; otherwise it is marked as success.

4.2 System under test

The system under test is a Quorum network that needs to be set up and teared down using different variables for each test run. To get an initial working setup, the tool *quorum-dev-quickstart* was used. This tool is developed by ConsenSys and allows generating a Quorum network with monitoring tools for either GoQuorum or Hyperledger Besu clients [72]. The resulting project uses Docker to manage all the services provided and includes shell scripts to bring the network up and down. The list of services included is:

- Four Quorum validator instances
- Three Quorum member instances
- Three Tessera instances
- One RPC node instance
- One Quorum explorer instance
- One Prometheus instance
- One Grafana instance
- One Loki instance
- One Promtail instance

Loki is a log aggregator system that is configured to store the logs from all the services [73]. Promtail is an agent included in each service instance that sends the logs to Loki [74]. Because logs can be seen directly using the *docker logs <container_id>* command, these two services were removed to save computer processing power for the network. The other services have already been explored in previous chapters and were kept for their intended purposes.

The generated project also included a set of smart contracts and a distributed app simulating a pet shop that can be deployed to manually test the network. However, these were also deleted as they were not needed for the benchmarking project.

To configure the network parameters, a *.env* file is included, as seen in Listing 4.7.

```
# This file defines environment variables defaults for Docker-compose
# but we also use it for shell scripts as a sourced file

BESU_VERSION=22.10.3
QUORUM_VERSION=22.7.5
TESSERA_VERSION=22.10.1
ETHSIGNER_VERSION=22.1.3
QUORUM_EXPLORER_VERSION=4f60191

LOCK_FILE=.quorumDevQuickstart.lock

# GoQuorum consensus algorithm
# istanbul, qbft, raft
# !!! lower case ONLY here
GOQUORUM_CONS_ALGO=qbft

# Besu consensus algorithm
# IBFT, QBFT, CLIQUE
# PLEASE NOTE: IBFT used here refers to IBFT2.0 and not IBFT1.0 More information can be
    found https://besu.hyperledger.org/en/latest/HowTo/Configure/Consensus-Protocols/IBFT
    /
# We use IBFT here to keep the API names consistent
# !!! upper case ONLY here
BESU_CONS_ALGO=QBFT
```

Listing 4.7: *quorum-dev-quickstart* initial environment file

This file was adapted by removing unused variables, changing the services' version to their latest ones and including a variable that allows configuring the number of nodes in the network. The final environment file that was used can be seen in Listing

```
# This file defines environment variables defaults for Docker-compose
# but we also use it for shell scripts as a sourced file

QUORUM_VERSION=23.4.0
TESSERA_VERSION=23.4.0

LOCK_FILE=.quorumDevQuickstart.lock

# GoQuorum consensus algorithm
# ibft, qbft, raft, clique
# !!! lower case ONLY here
GOQUORUM_CONS_ALGO=raft
# 7, 12, 17 or 22
NODES_NUMBER=7
```

Listing 4.8: Network environment file

The generated project contains a configuration directory with the necessary data for each service. This includes the prometheus configuration file, which can be seen in Appendix C, and also the configuration data for Tessera and Quorum nodes.

Tessera nodes require a JSON file named *tessera-config* that contains the server's ports, list of peers and the paths to the private and public keys it will use. The Tessera configuration file for the network of seven nodes can be seen in Listing 4.9.

```json
{
    "mode": "${TESSERA_MODE}",
    "useWhiteList": false,
    "jdbc": {
        "username": "sa",
        "password": "",
        "url": "jdbc:h2:./data/tm/db;MODE=Oracle;TRACE_LEVEL_SYSTEM_OUT=0",
        "autoCreateTables": true
    },
    "serverConfigs": [
        {
            "app": "ThirdParty",
            "enabled": true,
            "serverAddress": "http://${HOSTNAME}:9080",
            "communicationType": "REST"
        },
        {
            "app": "Q2T",
            "enabled": true,
            "serverAddress": "http://${HOSTNAME}:9101",
            "sslConfig": {
                "tls": "OFF"
            },
            "communicationType": "REST"
        },
        {
            "app": "P2P",
            "enabled": true,
            "serverAddress": "http://${HOSTNAME}:9000",
            "sslConfig": {
                "tls": "OFF"
            },
            "communicationType": "REST"
        }
    ],
    "peer": [
        {
            "url": "http://member0tessera:9000"
        },
        {
            "url": "http://member1tessera:9000"
        },
        {
            "url": "http://member2tessera:9000"
        }
    ],
    "keys": {
        "passwords": [],
        "keyData": [
            {
                "privateKeyPath": "/config/keys/tessera.key",
                "publicKeyPath": "/config/keys/tessera.pub"
            }
        ]
    },
    "alwaysSendTo": [],
    "bootstrapNode": false,
    "features": {
        "enableRemoteKeyValidation": false,
        "enablePrivacyEnhancements": true
    }
}
```

Listing 4.9: Tessera configuration

For networks with more nodes, the new members' tessera instances are added to the list of peers. The hostname is always *localhost* and the Tessera mode is always *tessera*.

For Quorum nodes, multiple configuration files are required:

- Genesis JSON file
- Disallowed nodes list
- Permissioned nodes list
- Static nodes list
- Keys
  - Account keystore
  - Account password
  - Account private key
  - Account address
  - Address
  - Private nodekey
  - Public nodekey
  - Tessera public key
  - Tessera private key

The genesis file contains information about the network itself and must be common for all the Quorum nodes in it. There are multiple network properties that can be configured in this file, with the relevant ones for this project being the ones mentioned in the previous chapter. Additionally, this file must also specify the account address list of all the nodes in the network and their initial balance.

The disallowed nodes list specifies the public nodekeys which can not connect to a given node, and in this project it is always an empty array. Meanwhile, the permissioned nodes list specifies which nodes can connect to a given node, and in this project it always includes all the network's nodes, thus allowing connections between all of them.

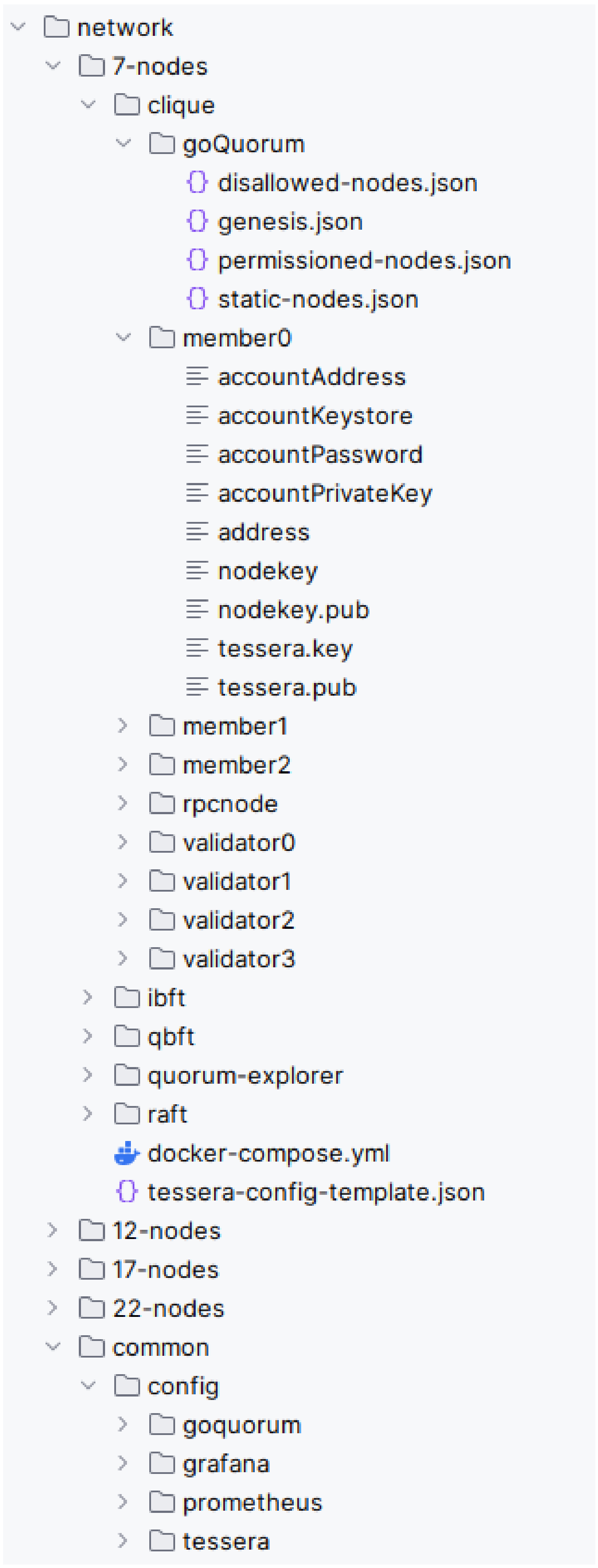
The static nodes list allows the network to discover nodes. An alternative to this would be to use use *bootnodes*, which are nodes whose purpose is to discover other nodes in the network. Because the network in this project is not dynamic, specifying the pre-configured list of static nodes is faster and provides better benchmark reproducibility.

Then, the list of keys is unique for each node, and they specify all the required information for each node to work correctly. Validator and RPC nodes do not contain the Tessera keys, but all other ones must be included.

Each number of nodes and consensus algorithm combination requires a unique genesis file. However, with the quickstart tool, only data for the network of seven nodes with either Raft, IBFT, or QBFT are available. To generate Quorum configuration files for all combinations previously specified, the quorum genesis tool was used [75].

This tool was also developed by ConsenSys, and it allows generating genesis files for a configurable number of member nodes with Tessera keys, validator nodes, and all Quorum consensus algorithms. Besides the genesis files, all required keys are also generated for each node.

To support running all the different configurations by only changing the environment file, the structure found in Figure 4.1 was created.



```
network
  7-nodes
    clique
      goQuorum
        disallowed-nodes.json
        genesis.json
        permissioned-nodes.json
        static-nodes.json
      member0
        accountAddress
        accountKeystore
        accountPassword
        accountPrivateKey
        address
        nodekey
        nodekey.pub
        tessera.key
        tessera.pub
      member1
      member2
      rpcnode
      validator0
      validator1
      validator2
      validator3
    ibft
    qbft
    quorum-explorer
    raft
    docker-compose.yml
    tessera-config-template.json
  12-nodes
  17-nodes
  22-nodes
  common
    config
      goquorum
      grafana
      prometheus
      tessera
```

Figure 4.1: Network file structure

Inside the *network* directory is a folder named *common*, which contains the docker images and non-environment specific configuration files. Then, a subfolder was created for each number of nodes, and inside each of these is the *docker-compose* file, the Tessera configuration file, and four folders, one for each consensus algorithm. The *docker-compose* file for a seven-node network can be seen in Appendix E. The files for other networks follow the exact same format, but with extra member and validator nodes. Then, inside each consensus algorithm folder are the files generated by the *quorum-genesis-tool*, with the IP addresses in the static and permissioned nodes list updated to match the ones of the docker network.

In the *network* root folder, there is another *docker-compose* file containing the Grafana, Prometheus, and pumba services, as well as the docker network configuration, which can be seen in Appendix D.

The script that starts the network can be seen in Listing 4.10.

```bash
#!/bin/bash -u

NO_LOCK_REQUIRED=true

. ./.env
. ./.common.sh

mkdir -p logs/besu logs/quorum logs/tessera

echo "docker-compose.yml" > ${LOCK_FILE}

echo "Start network"
echo "--------------------"

echo "Starting network..."
docker compose -f docker-compose.common.yml -f "${NODES_NUMBER}-nodes/docker-compose.yml"
     build --pull
docker compose -f docker-compose.common.yml -f "${NODES_NUMBER}-nodes/docker-compose.yml"
     up --detach

./list.sh
```

Listing 4.10: Start network script

As it can be seen, the environment variables are used to dynamically call the pre-configured network environment. The same logic is used in the Tessera and GoQuorum node's Dockerfiles, which can be seen in listings 4.11 and 4.12.

```
ARG QUORUM_VERSION
FROM quorumengineering/quorum:${QUORUM_VERSION}

ARG NODES_NUMBER
ARG GOQUORUM_CONS_ALGO

RUN apk add --no-cache curl

COPY common/config/goquorum/data data
COPY ${NODES_NUMBER}-nodes/${GOQUORUM_CONS_ALGO}/goQuorum/ data
COPY common/config/goquorum/docker-entrypoint.sh /usr/local/bin/

ENTRYPOINT ["docker-entrypoint.sh"]
```

Listing 4.11: GoQuorum Dockerfile

```
ARG TESSERA_VERSION

FROM quorumengineering/tessera:${TESSERA_VERSION}

ARG NODES_NUMBER

RUN if [ -e /sbin/apk ] ; then apk add gettext --no-cache ; else apt-get update && apt-
    get install -y gettext && rm -rf /var/lib/apt/lists/* ; fi

ENV JAVA_OPTS="-Dlogback.configurationFile=/data/logback.xml"

COPY common/config/tessera/docker-entrypoint.sh /usr/bin/
RUN mkdir data
COPY ${NODES_NUMBER}-nodes/tessera-config-template.json data/tessera-config-template.json

ENTRYPOINT ["docker-entrypoint.sh"]
```

Listing 4.12: Tessera Dockerfile

Using this setup, it is possible to tear down a network and run a new one with a different consensus algorithm or number of nodes by just altering the properties in the environment file, which facilitates benchmarking so many variables.

4.3 Workload

As explored in the previous chapter, Hyperledger Caliper's *simple* scenario was used as a workload for the benchmarks. This scenario simulates a small bank where accounts are open, queried, and transactions between accounts are made.

The smart contracts used for these operations are deployed to the blockchain at the start of every benchmark run. The solidity code for them can be seen in Listing 4.13.

```
pragma solidity >=0.4.22 <0.6.0;

contract bank {
    mapping(string => int) private accounts;

    function open(string memory acc_id, int amount) public {
        accounts[acc_id] = amount;
    }

    function query(string memory acc_id) public view returns (int amount) {
        amount = accounts[acc_id];
    }

    function transfer(string memory acc_from, string memory acc_to, int amount) public {
        accounts[acc_from] -= amount;
        accounts[acc_to] += amount;
    }
}
```

Listing 4.13: Bank solidity code

As seen, the code is straightforward and thus creates as little overhead as possible on top of the network.

To automate running all the tests without having to manually change the required variables, the script in Listing 4.14 was developed.

```bash
#!/bin/bash -u

NETWORK_ENV_FILE='./network/.env'
TIMESTAMP=$(date +%s)

RESULTS_DIR=./results/"$TIMESTAMP"

mkdir -p "$RESULTS_DIR"

for NODES_NUMBER in 7 12 17 22; do
  sed -i 's/NODES_NUMBER=.*/NODES_NUMBER='$NODES_NUMBER'/' $NETWORK_ENV_FILE
  for CONSENSUS_ALGO in raft clique ibft qbft; do
    sed -i 's/GOQUORUM_CONS_ALGO=.*/GOQUORUM_CONS_ALGO='$CONSENSUS_ALGO'/' $NETWORK_ENV_FILE
    for TYPE in public private; do
      for TPS in 50 100 200 300; do
        cd ./network || exit 1
        ./run.sh
        cd ..
        sleep 90 # wait for network to init
        sed -i 's/tps:.*/tps: '$TPS'/' ./caliper/benchmarks/scenario/bank-"$TYPE"/config.yaml
        echo "Executing benchmark with $NODES_NUMBER, $CONSENSUS_ALGO, $TPS, $TYPE"

        cd ./caliper || exit 1
        if [[ "$TYPE" = "public" ]]; then
          npm run launch-bank-public
        else
          npm run launch-bank-private-"$NODES_NUMBER"-nodes
        fi

        cd ..
        sleep 5
        cp ./caliper/report.html "$RESULTS_DIR"/report-$CONSENSUS_ALGO-$NODES_NUMBER-$TYPE-$TPS.html
        cd ./network || exit 1
        ./remove.sh
        cd ..
        echo "Benchmark ran successfully"
      done
    done
  done
done
```

Listing 4.14: Script to run all benchmarks

The script is essentially four nested *for loops*, iterating over every possible combination of number of nodes, consensus algorithm, type of transaction, and transaction per second. The network is restarted before every benchmark, and the script waits 90 seconds before each run to ensure every node is up and running. The 90-second sleep time was chosen despite the network tending to be ready in a few seconds to be safe and avoid having to re-run benchmarks.

Each benchmark run generates a report HTML file containing all the data collected. To avoid overwriting previous runs, a folder with the timestamp of when the run was started is also created, and the report files are stored inside it. The name of each file is the four variables appended to facilitate collecting the results after.

The script was run with and without Pumba enabled to compare results between networks where some validators were being brought offline at random and ones where the validators were online during the whole test.

All the source code is available in GitHub [76].

Chapter 5

Benchmarks and interpretation

This chapter presents the method used to evaluate the results obtained. Then, the hypotheses are defined, the approach to the evaluation is detailed, and then the results obtained are presented and explored.

The goals, metrics and methodology used to benchmark the system have been explored in chapter 3.

5.1 Hypotheses

To evaluate if the goals previously defined were met, the following hypotheses were specified for each research question:

- Q1:
  - **Hypothesis 1 (HA0)**: The consensus algorithm used influences performance
  - **Hypothesis 0 (HA1)**: The consensus algorithm used does not influence performance
- Q2:
  - **Hypothesis 1 (HB1)**: The consensus algorithm used influences fault-tolerance
  - **Hypothesis 0 (HB0)**: The consensus algorithm used does not influence fault-tolerance

5.2 Approach

A Quorum network was developed and stress-tested while being monitored to collect relevant data.

To ensure the hypotheses are verified correctly, data was collected while tweaking different variables individually, and each test ran multiple times to ensure data consistency and avoid outliers.

Additionally, to ensure relevancy when validating the hypothesis, a data point for a consensus mechanism is only considered better or worse than another if there is a 5% difference between them.

5.3 Results

This section presents the results after running the benchmarks previously described.

For all consensus algorithms, private transactions are expected to have a lower throughput than public ones due to the overhead of Tessera encrypting and decrypting messages. Networks with a lower number of nodes are also expected to outperform ones with a higher number, due to spending less time propagating transactions across the network. The key points of comparison, thus, are how each consensus algorithm compares with each other under the same network scenarios.

Results will then first be presented for each consensus algorithm, to analyze how each one scales and how private transactions compare to public ones. Then, the different consensus algorithms will be compared with each other for the same scenarios.

5.3.1 Raft

This subsection will present the results obtained for the Raft consensus algorithm for both performance and fault-tolerance.

Performance

Figures 5.1, 5.2, and 5.3 show the throughput results for Raft, for all numbers of nodes with public and private transactions. The results are divided into the different operations benchmarked: *open*, *query*, and *transfer*.

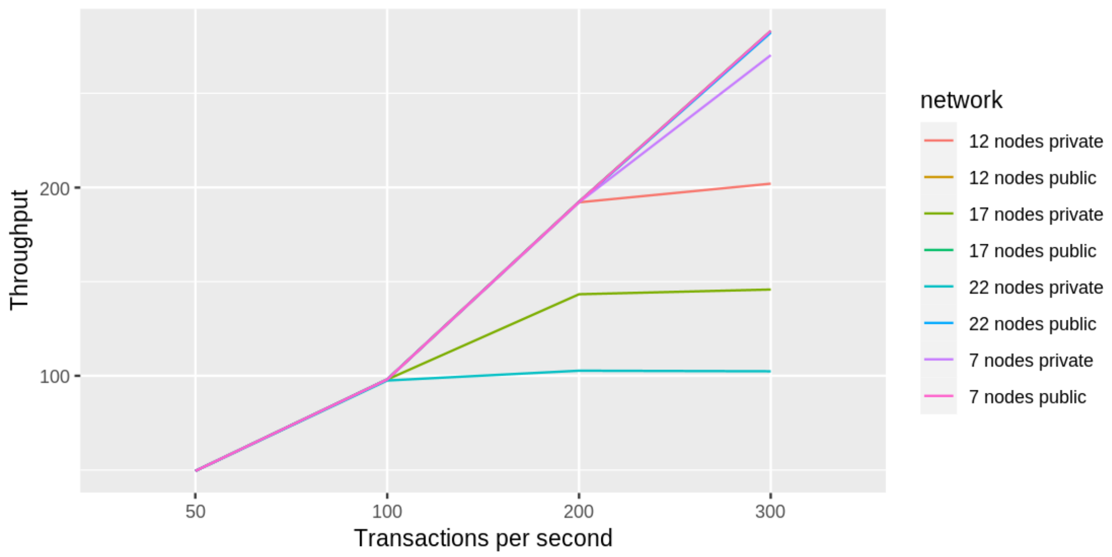


Figure 5.1: Raft open throughput

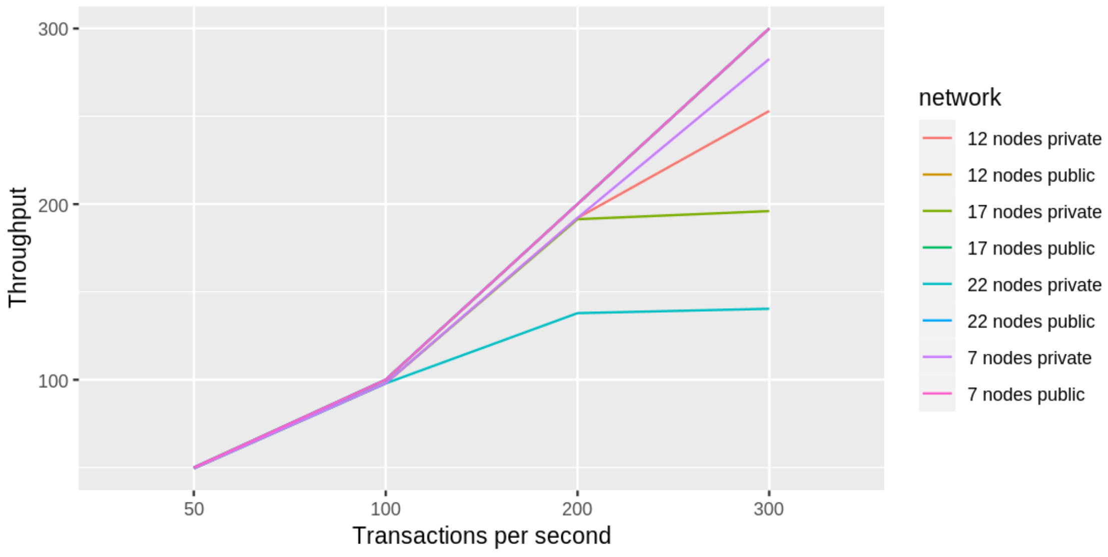


Figure 5.2: Raft query throughput

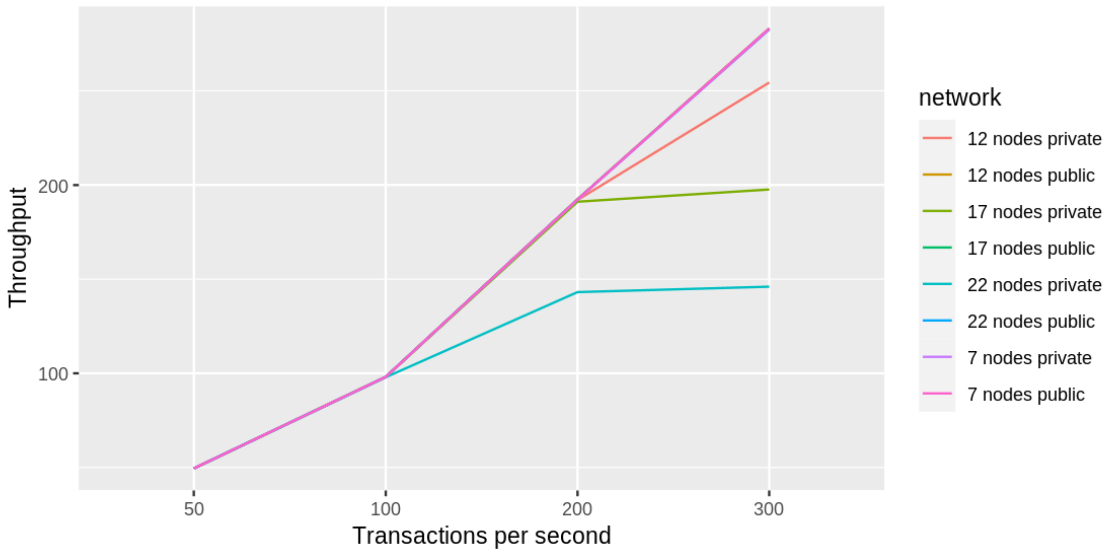


Figure 5.3: Raft transfer throughput

In all operations and for both privacy configurations, until 100 TPS, all network configurations using Raft managed to reach near the maximum throughput of 100. With 200 and 300 TPS, however, the throughput varies greatly across configurations.

Public networks managed to achieve roughly the same throughput regardless of the number of nodes in the network, with 283 for both *open* and *transfer*, and 300 for *query*.

With private transactions, however, a higher number of nodes consistently resulted in lower throughput.

For *open*, the throughput on the 7-node network is on par with the public ones on 200 TPS, but sees a slight decrease at 300 TPS. For *query*, the throughput is slightly lower than the public ones across the board, but never decreasing too much, and for *transfer* the throughput is exactly on par with the public ones.

With 12 nodes, the throughput of *open* until 200 TPS is also on par with the public networks, but it decreases greatly at 300 TPS, to only 200. For *query*, the results are the same as the 7-node private network until 200 TPS, but the throughput dips to 253 at 300 TPS. For the *transfer* operation, results were again on par until 200 TPS, but fell to 254 at 300 TPS.

The 17 and 22-node networks showed a much lower throughput at 200 TPS, which plateaued until 300 TPS, with the 17-node network reaching only 145 and the 22-node one 100 processed transactions per second. For *query* and *transfer*, the 17-node one managed to achieve the same throughput as the rest at 200 TPS, but fell greatly at 300 TPS. Meanwhile, the 22-node one already fell short at 200 TPS, then plateauing until 300 TPS.

Results for latency can be seen in Figures 5.4, 5.5, and 5.6.

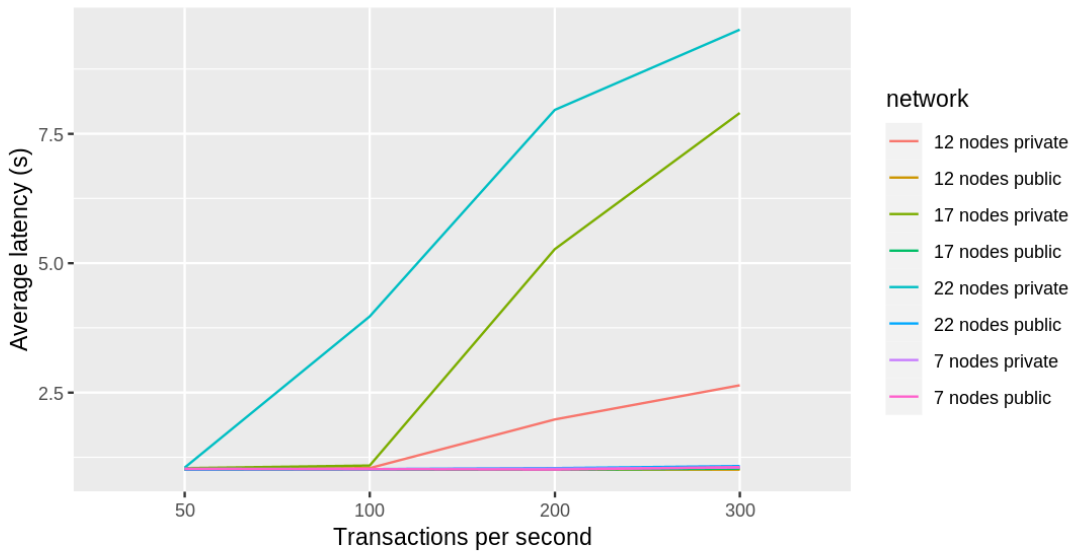


Figure 5.4: Raft *open* latency

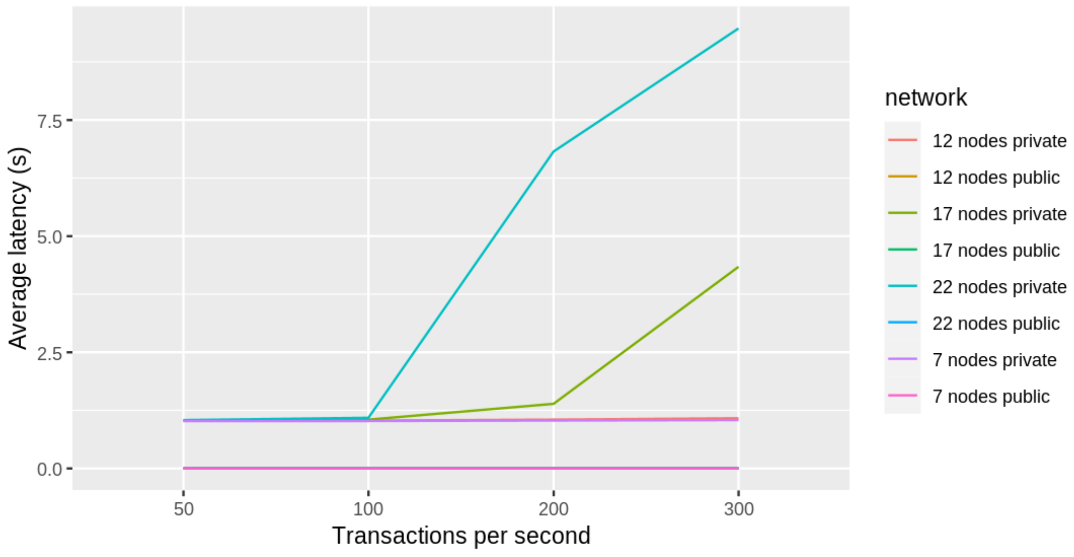


Figure 5.5: Raft *query* latency

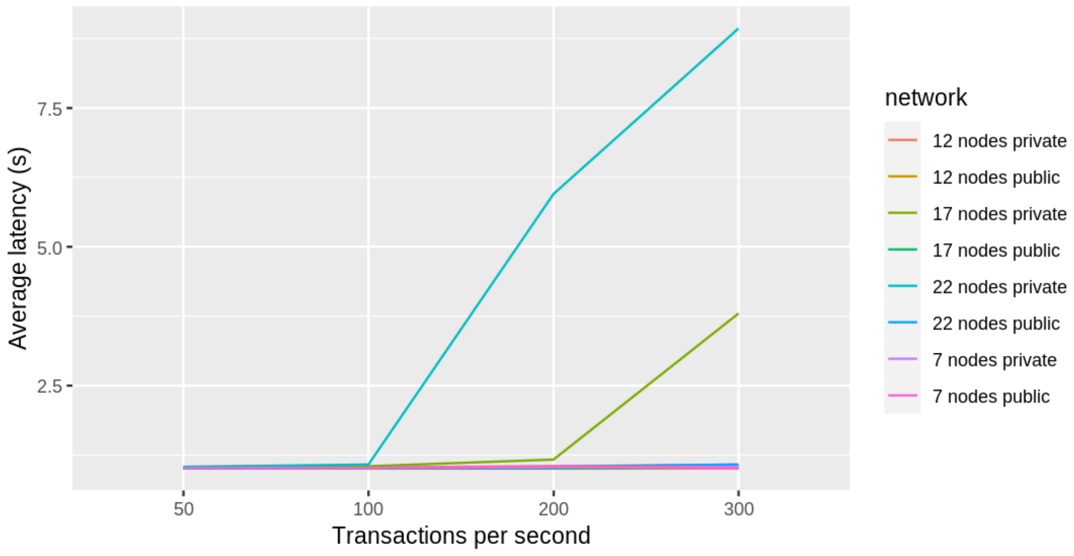


Figure 5.6: Raft *transfer* latency

For public transactions latency stays consistently low, around 1s for *open* and *transfer*, and 0s for *query*. However, with private transactions, the results vary more.

The 7-node private network manages latency numbers comparable to the public ones across all operations.

With 12 nodes, the latency stays at 1s up to 100 transactions per second, increasing to round 2.5s with 300 transactions per second. For the query and transfer operations, latency stays at 1s across all TPS configurations.

With 17 nodes, the latency for *open* is noticeably worse at 200 TPS, with an average of 5.3s, which increases to 7.9s with 300 TPS, an increase of 49% in this step. For the *query* and *transfer* latency stays low up until 200 TPS, only increasing to 4.3 and 3.8 seconds, respectively, at 300 TPS.

With 22 nodes, for the *open* operation, the latency still manages to stay low at 50 TPS, but immediately increases to 4s at 100 TPS, 8s at 200 TPS, and 9.51s at 300 TPS. For *query* and *transfer*, the latency stays low up to 100 TPS, only increasing at 200 TPS and 300 TPS, thus further confirming the overhead that the open transaction has over the others.

Results show that the throughput of the *open* operation is much lower than the *query* or *transfer* ones. Additionally, and as expected, public transactions achieve much greater performance than private ones, with the latter seeing their processing power plateauing at a certain number of nodes and transactions per second.

Figures 5.7, 5.8, and 5.9 show the CPU usage for Raft across all operations.

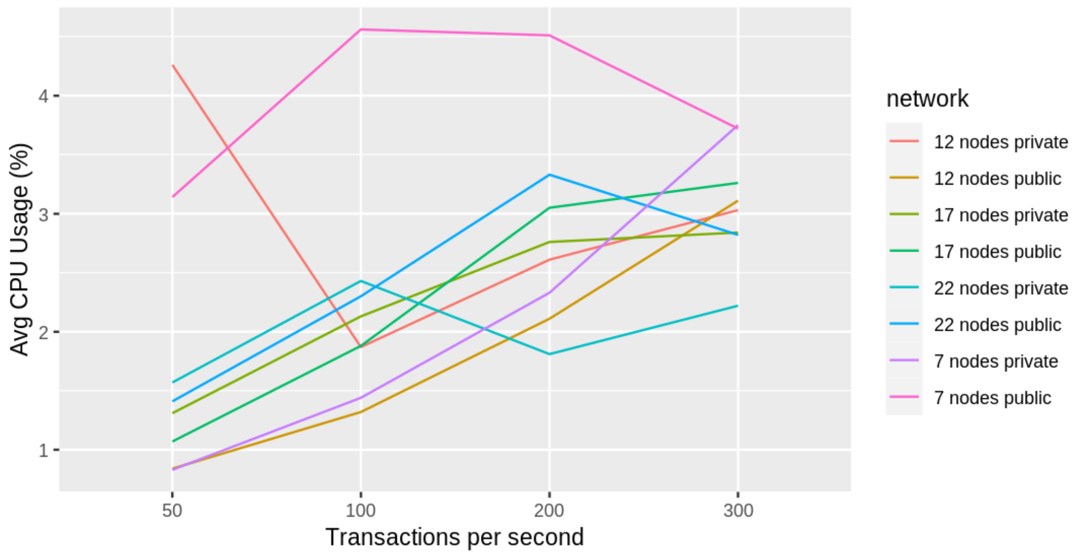


Figure 5.7: Raft *open* CPU usage

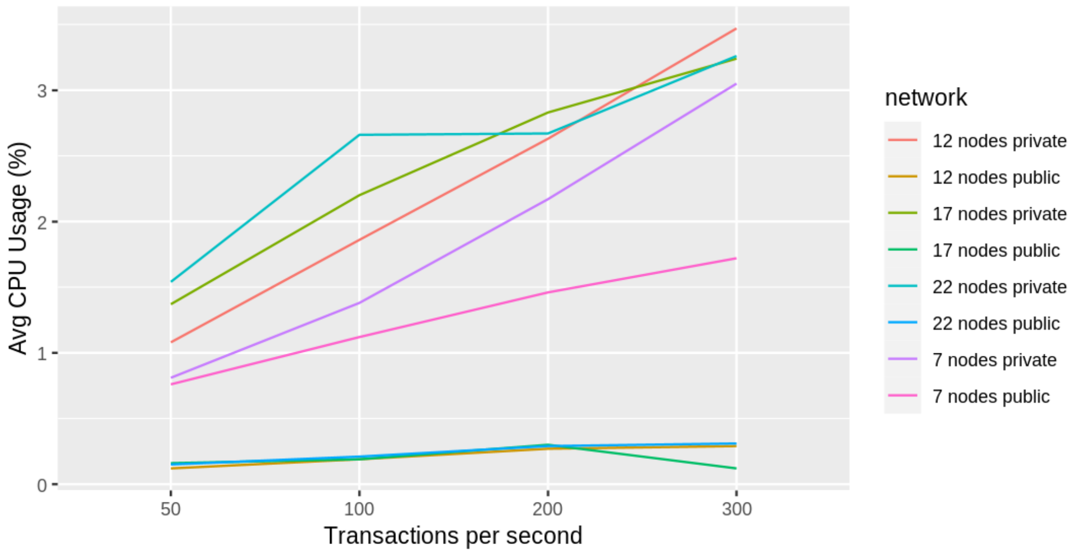


Figure 5.8: Raft *query* CPU usage

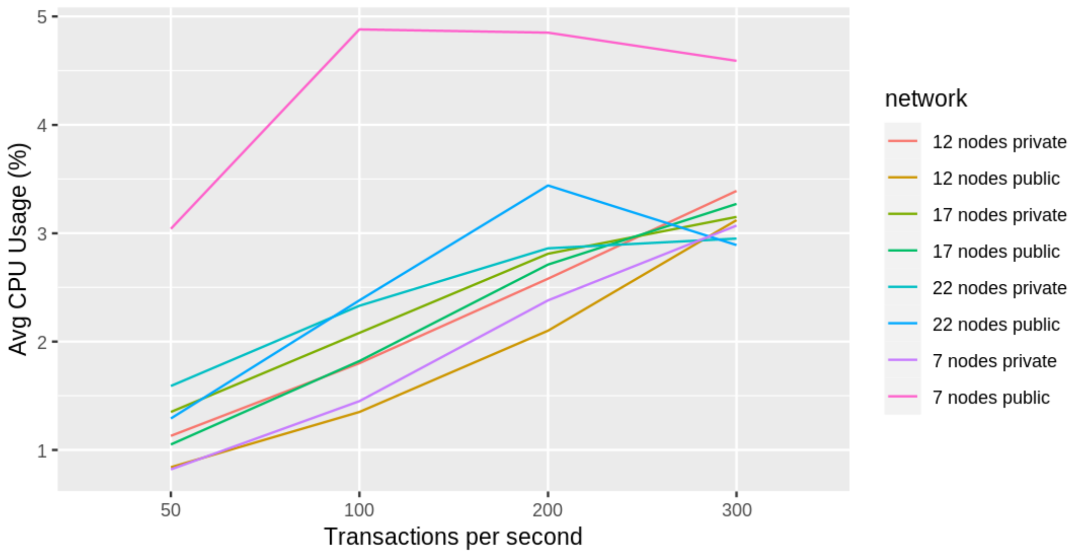


Figure 5.9: Raft *transfer* CPU usage

As it can be observed, Raft's CPU usage is very low across the board, reaching a maximum of 4.6% on the network with 7-nodes using public transactions. In all operations, CPU usage increases slightly as the number of transactions per second increases, but only by a few percentage points. There is no clear correlation found between the number of nodes or transaction type and CPU usage.

Results for RAM usage can be found in Figures 5.10, 5.11, and 5.12.

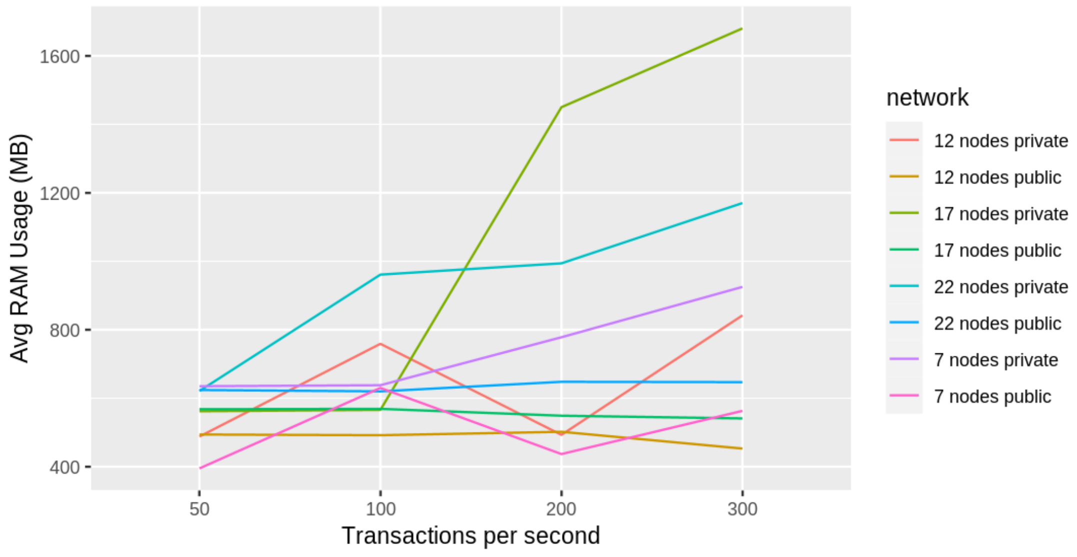


Figure 5.10: Raft *open* RAM usage

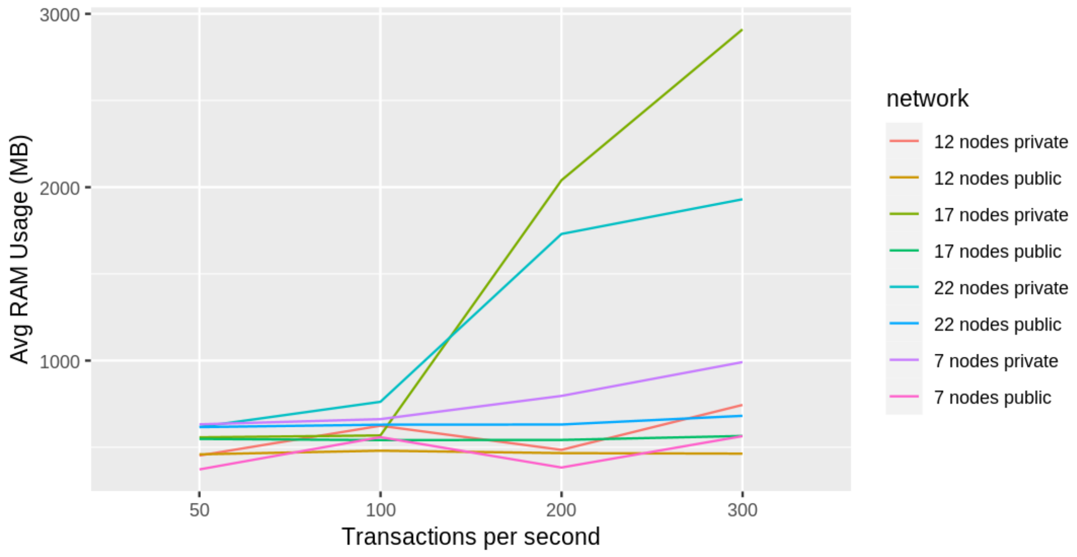


Figure 5.11: Raft *query* RAM usage

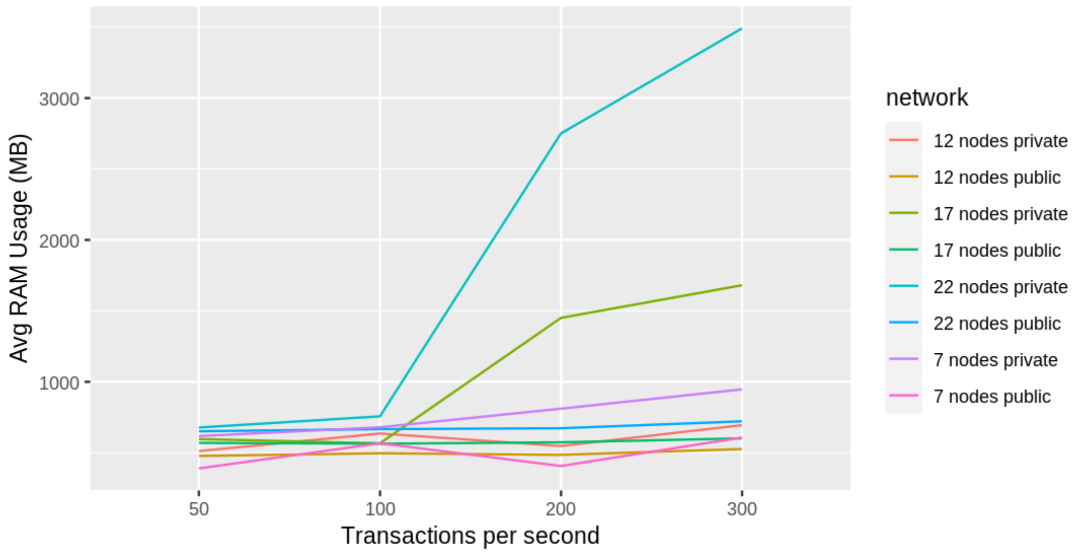


Figure 5.12: Raft *transfer* RAM usage

In all operations, RAM usage increases as the number of transactions per second also increases. For the 17 and 22-node private networks, the jump in RAM usage at 200 and 300 TPS is particularly high, jumping from under 1Gb of RAM to more than double. All other scenarios saw incremental increases in RAM usage but not as steep.

It can also be observed that private networks have higher RAM usage than public ones. In a production scenario, this effect is aggravated by the fact that Tessera instances are required for these types of networks, and they also have a relatively high RAM usage.

Read operations saw a consistent higher RAM usage than write ones, which can be explained by the fact that nodes can answer reads at much faster speed, thus keeping more information in memory at a given time.

Fault-tolerance

Without introducing validator failures with pumba, Raft never had failures on public transactions, and only saw failures with private ones in the scenarios found in Table 5.1.

Table 5.1: Raft transaction errors without chaos testing

Scenario	**Number of failures out of 5000**
12-node, 300 TPS, *open*	3
17-node, 200 TPS, *open*	819
17-node, 300 TPS, *open*	923
17-node, 300 TPS, *query*	299
17-node, 300 TPS, *transfer*	153
22-node, 100 TPS, *open*	64
22-node, 200 TPS, *open*	2821
22-node, 200 TPS, *query*	684
22-node, 200 TPS, *transfer*	624
22-node, 300 TPS, *open*	2837
22-node, 300 TPS, *query*	1006
22-node, 300 TPS, *transfer*	988

By introducing random 10 second latency to validator nodes, once again there were no failures in public transactions, and the results for private ones can be found in Table 5.2.

Table 5.2: Raft transaction errors with chaos testing

Scenario	**Number of failures out of 5000**
12-node, 300 TPS, *open*	3
17-node, 200 TPS, *open*	806
17-node, 300 TPS, *open*	988
17-node, 300 TPS, *query*	260
17-node, 300 TPS, *transfer*	201
22-node, 100 TPS, *open*	64
22-node, 200 TPS, *open*	2634
22-node, 200 TPS, *query*	645
22-node, 200 TPS, *transfer*	542
22-node, 300 TPS, *open*	2649
22-node, 300 TPS, *query*	920
22-node, 300 TPS, *transfer*	854

Analysing at the results, there is no noticeable difference in the number transaction failures, and throughput and latency were also not affected by this change in any benchmarked scenario, leading to the conclusion that validators being disabled did not affect the error rate of transactions, nor its general performance.

5.3.2 Clique

This subsection will present the results obtained for the Clique consensus algorithm for both performance and fault-tolerance.

Performance

Figure 5.13 shows the throughput results for the *open* transaction with Clique, for all numbers of nodes with public and private transactions.

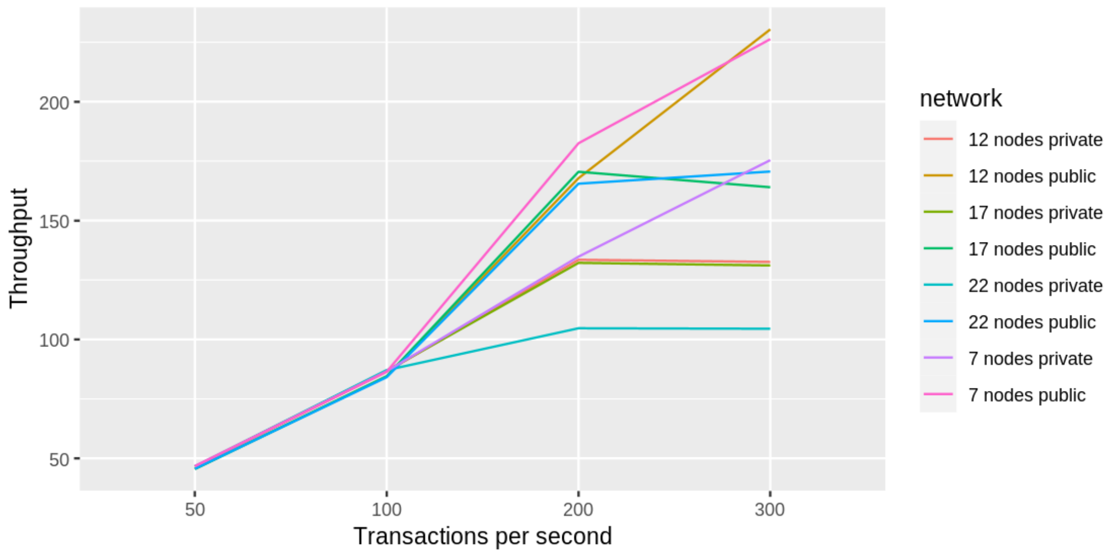


Figure 5.13: Clique open throughput

Until 100 TPS, all network configurations using Clique reached roughly the same throughput for *open*, varying between 84.4 and 87.1. With 200 and 300 TPS, however, the throughput varied greatly across configurations.

Public transactions achieved a higher throughput than private ones across the board for the same number of nodes. The 7 and 12-node public configurations outperformed the rest under all TPS configurations, as expected. However, the 17 and 22-node public networks, despite outperforming all the private ones up until 200 TPS, had their throughput plateau past this at just 164.0 and 170.6 on 300 TPS, respectively. This caused them to have a lower throughput than the 7-node private network at 300 TPS.

Analyzing the private transactions, the 7-node network outperformed the rest, which was also expected. The 12 and 17-node networks performed near identically across the board, with the 12-node one being slightly having slightly more throughput. Then, the 22-node network performed the worse, only reaching 104.7 throughput.

The throughput results for the *query* operation can be seen in Figure 5.14.

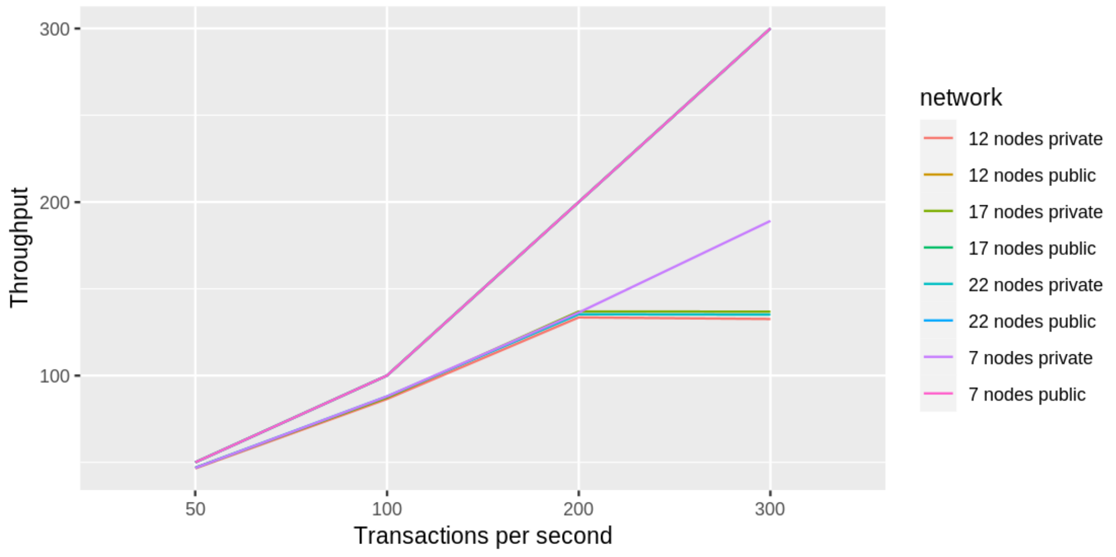


Figure 5.14: Clique query throughput

As it can be seen, all public network configurations achieved the maximum possible result across all scenarios.

With private transactions, all network configurations performed similarly until 200 TPS, always being at a difference lower than 4 transactions per second. At 300 TPS, the 7-node network manages to increase its throughput to 189.1, but the rest are unable to scale and continue processing at the same rate.

For the *transfer* operation, results can be seen in Figure 5.15.

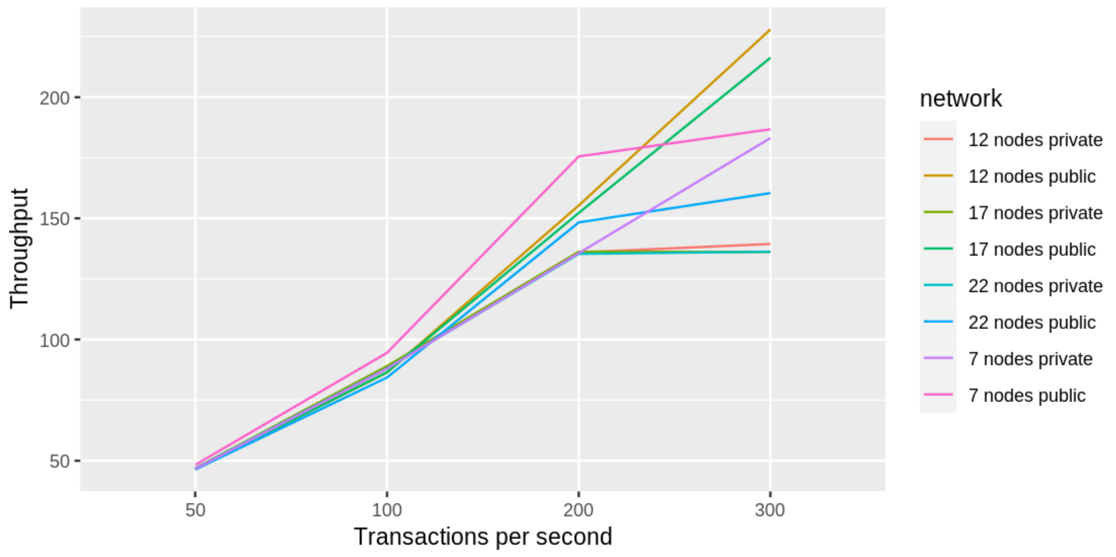


Figure 5.15: Clique transfer throughput

These are similar to the ones obtained in the *open* operation, with all the public networks outperforming the rest except at 300 TPS, where the 7-node private scenario achieves a higher throughput than the 22-node public one.

These results also showcase some inconsistency with Clique, as the 12 and 17-node public networks managed to achieve a higher throughput than the 7-node public one, which was unexpected.

Private networks all achieve roughly the same results until 200 TPS, with the 7-node scenario then achieving a throughput of 183.1 at 300 TPS, whereas the rest remained flat with results between 136.1 and 139.4 at the same TPS.

Results for latency can be seen in Figures 5.16, 5.17, and 5.18.

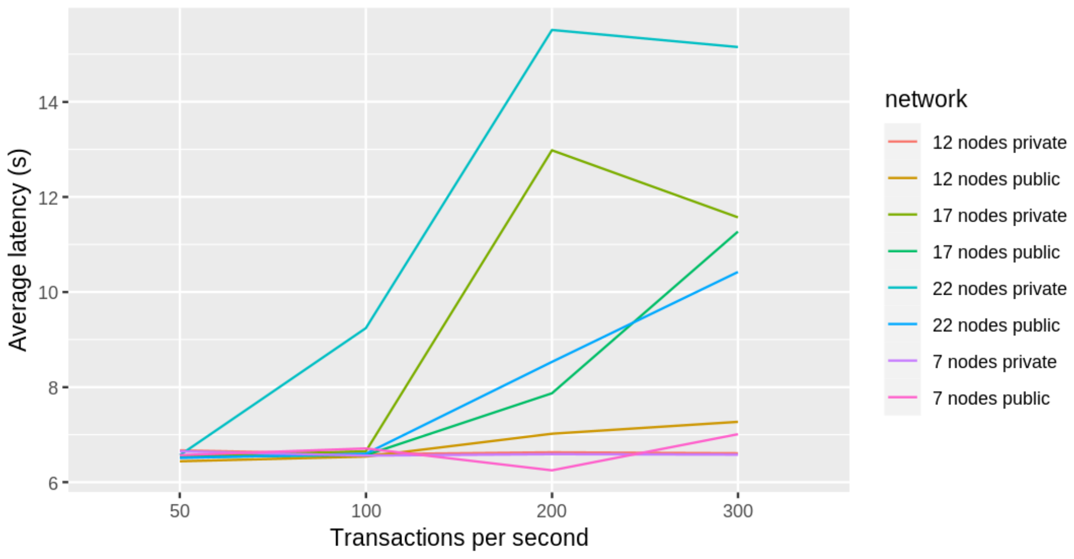


Figure 5.16: Clique *open* latency

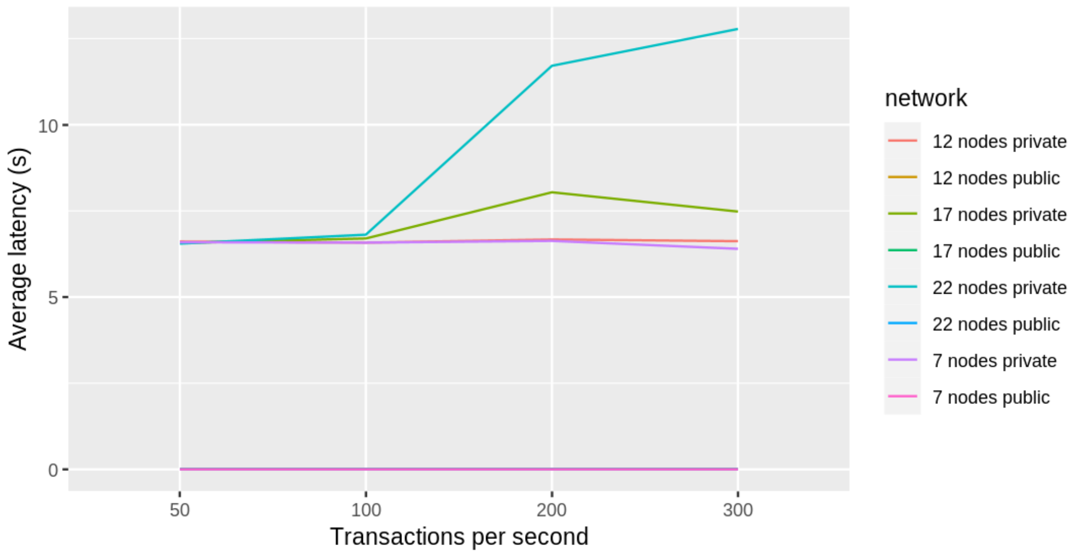


Figure 5.17: Clique *query* latency

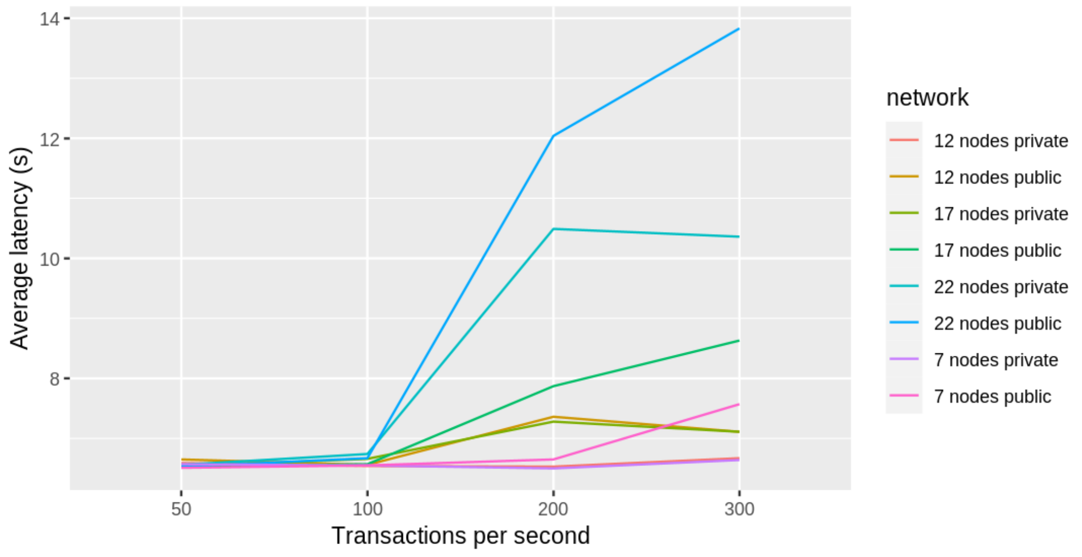


Figure 5.18: Clique *transfer* latency

With public transactions latency starts at 6.5 seconds and increases up to 13 seconds at 300 TPS for *open* and *transfer*, and is always 0 for *query*.

With private transactions, the results are similar to the public ones for *open* and *transfer*. For *open*, all node configurations start at 6.5 seconds and reach 15.15 seconds on the 22-node network. This network is also an outlier by consistently having more latency than the rest across all TPS.

For *transfer*, latency starts at 6.5 for all configurations, and reaches 10.46 seconds at worse, also on the 22-node network. Unexpectedly, for *transfer*, the latency results are worse with public transactions than they with for private ones.

For *query*, results with private transactions are as expected, with the latency increasing as the number of nodes in the network increases. The latency for the 22-node network is once again particularly high, reaching 12.78 seconds at 300 TPS.

Figures 5.19, 5.20, and 5.21 show the CPU usage for Clique across all operations.

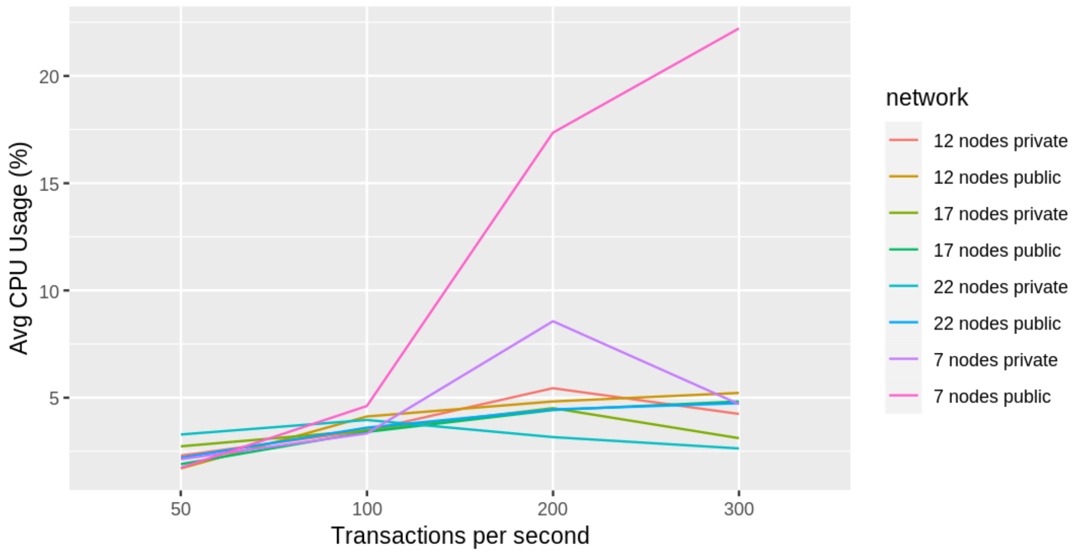


Figure 5.19: Clique *open* CPU usage

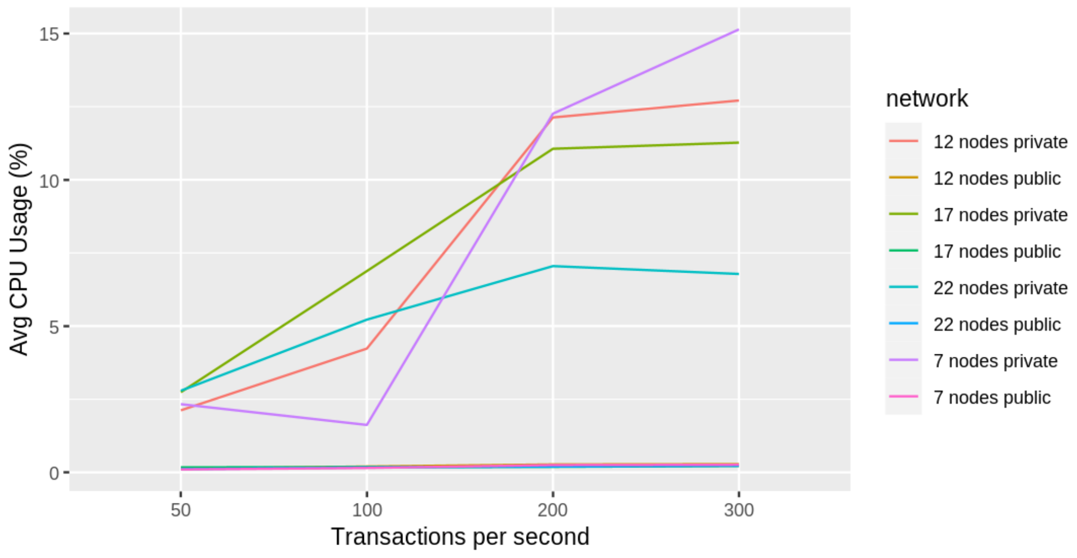


Figure 5.20: Clique *query* CPU usage

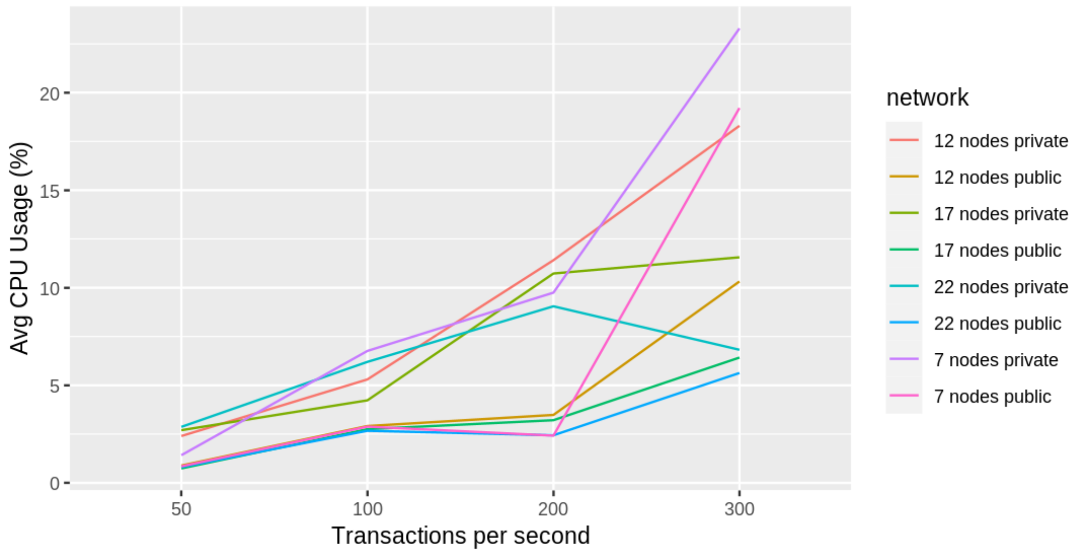


Figure 5.21: Clique *transfer* CPU usage

For *open*, CPU usage starts at 2 to 3% for all network configurations, and tends to increase as the number of transactions increases, peaking at around the 5% mark. Despite this tendency, there are some exceptions due to unusual peaks, like the 7-node private network reaching 8.6% at 200 TPS. The outlier is the 7-node public network which reached 17.35% CPU usage at 200 TPS and 22.21% at 300 TPS. This is partly explained because in networks with fewer nodes, each node will have to process more transactions, resulting in increased resource usage.

For *query*, CPU usage for public networks was consistently low, between 0.1 and 0.28%.

In private networks, results were as expected: CPU usage increased with the number of TPS, and networks with fewer nodes saw consistently higher CPU usage. This correlation is somewhat inconsistent until 100 TPS, but becomes noticeable at 200 and 300 TPS.

The *transfer* operation saw CPU usage increase as TPS increased, and generally networks with less number of nodes again saw higher CPU usage than ones with less, although again there were some exceptions.

Results for RAM usage can be found in Figures 5.22, 5.23, and 5.24.

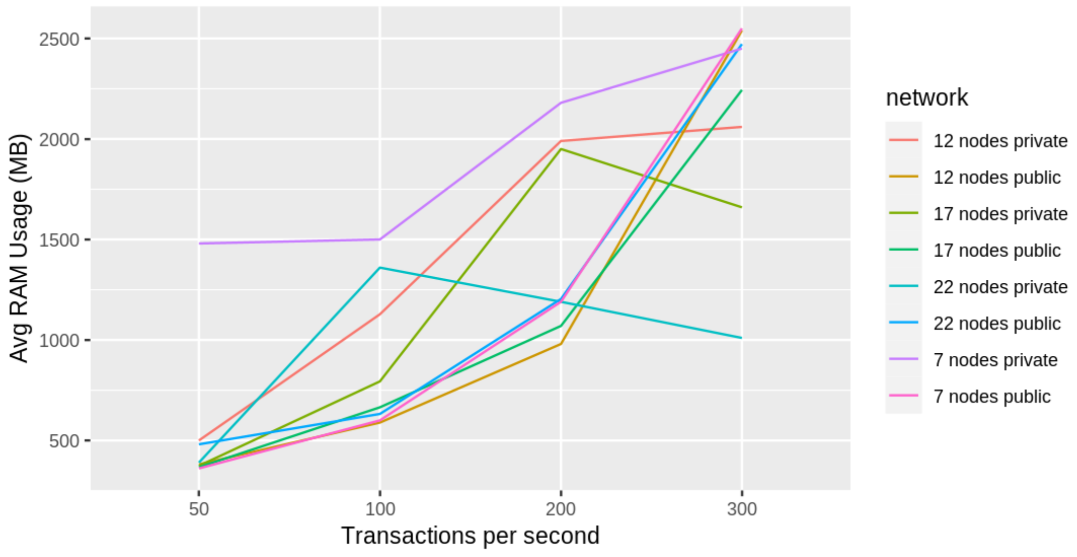


Figure 5.22: Clique *open* RAM usage

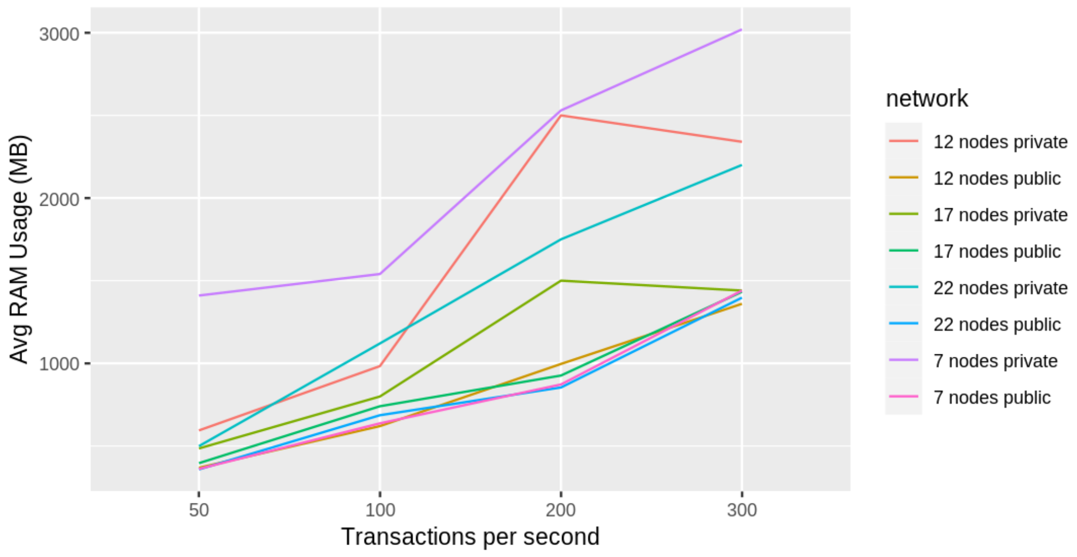


Figure 5.23: Clique *query* RAM usage

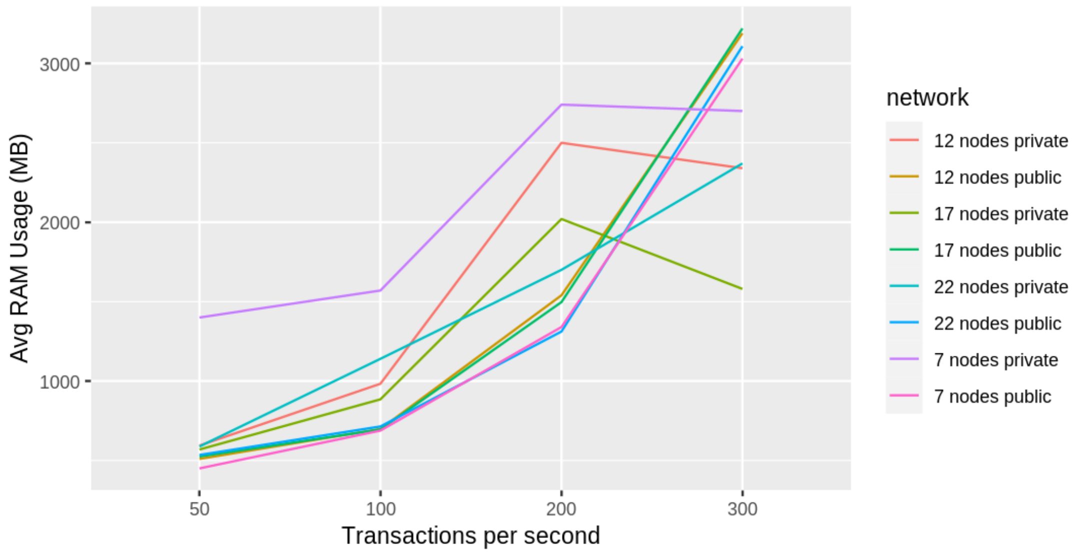


Figure 5.24: Clique *transfer* RAM usage

In all operations, once again RAM usage increases with the number of transactions per second. For the 17 and 22-node private networks, the jump in RAM usage at 200 and 300 TPS is particularly high, jumping from under 1Gb of RAM to more than double. All other scenarios saw incremental increases in RAM usage but not as steep. The 7-node network had consistently higher RAM usage until 200 TPS.

RAM usage was roughly the same across all operations with private transactions being used. On public ones, however, the *query* operation saw less RAM usage than the other two.

Fault-tolerance

Without introducing validator failures with pumba, Clique never had failures on public transactions.

With with private transaction, failures were found in the scenarios shown in Table 5.3.

Table 5.3: Clique transaction errors without chaos testing

Scenario	**Number of failures out of 5000**
17-node, 200 TPS, *open*	638
17-node, 300 TPS, *open*	269
22-node, 100 TPS, *open*	2
22-node, 200 TPS, *open*	2956
22-node, 200 TPS, *query*	669
22-node, 200 TPS, *transfer*	146
22-node, 300 TPS, *open*	3003
22-node, 300 TPS, *query*	742
22-node, 300 TPS, *transfer*	142

Using pumba, once again there were only failures with private transactions, as shown in Table5.4.

Table 5.4: Clique transaction errors with chaos testing

Scenario	Number of failures out of 5000
17-node, 200 TPS, *open*	847
17-node, 300 TPS, *open*	808
22-node, 100 TPS, *open*	16
22-node, 200 TPS, *open*	2884
22-node, 200 TPS, *query*	1
22-node, 200 TPS, *transfer*	505
22-node, 300 TPS, *open*	2843
22-node, 300 TPS, *query*	324
22-node, 300 TPS, *transfer*	912

There is no noticeable difference in the number of transaction failures, and throughput and latency were also not affected by this change in any benchmarked scenario. As such, introducing chaos testing did not affect the error rate of transactions nor the network's performance.

5.3.3 IBFT

This subsection will present the results obtained for the IBFT 1.0 consensus algorithm for both performance and fault-tolerance.

Performance

Figures 5.25, 5.26, and 5.27 show the throughput results for IBFT, for all numbers of nodes with public and private transactions. The results are divided into the different operations benchmarked.

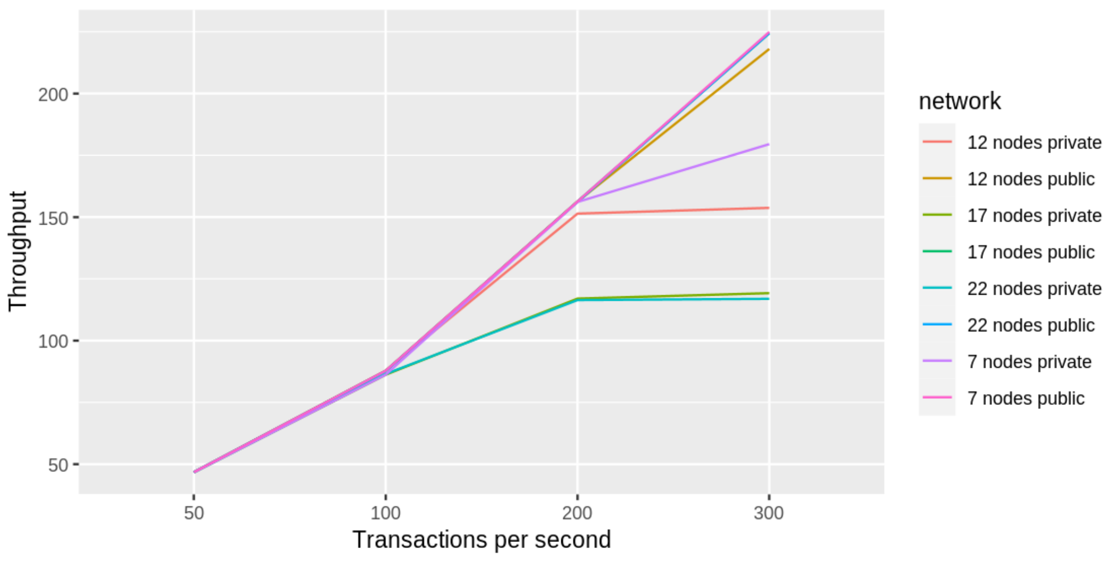


Figure 5.25: IBFT open throughput

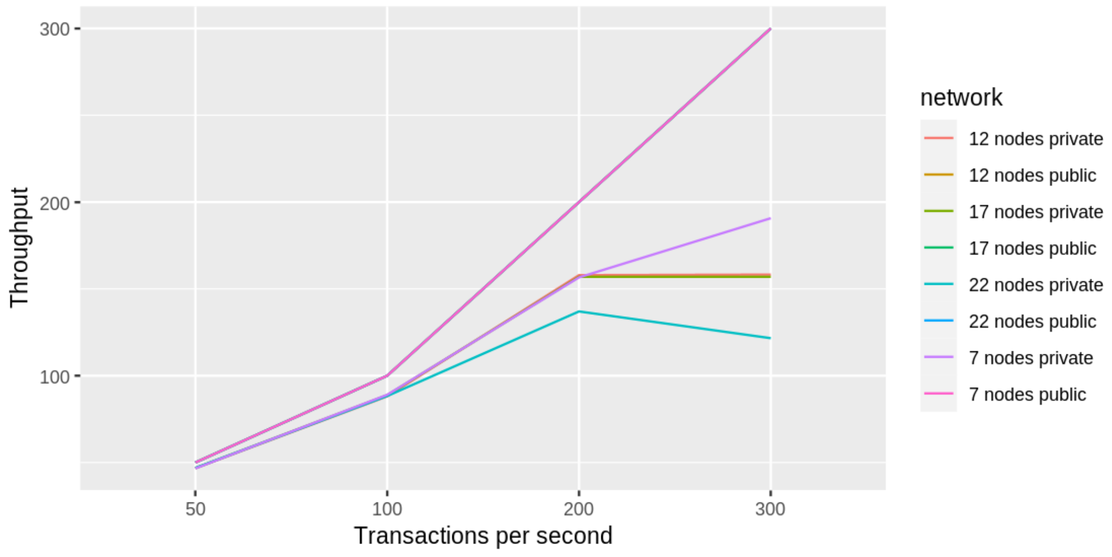


Figure 5.26: IBFT query throughput

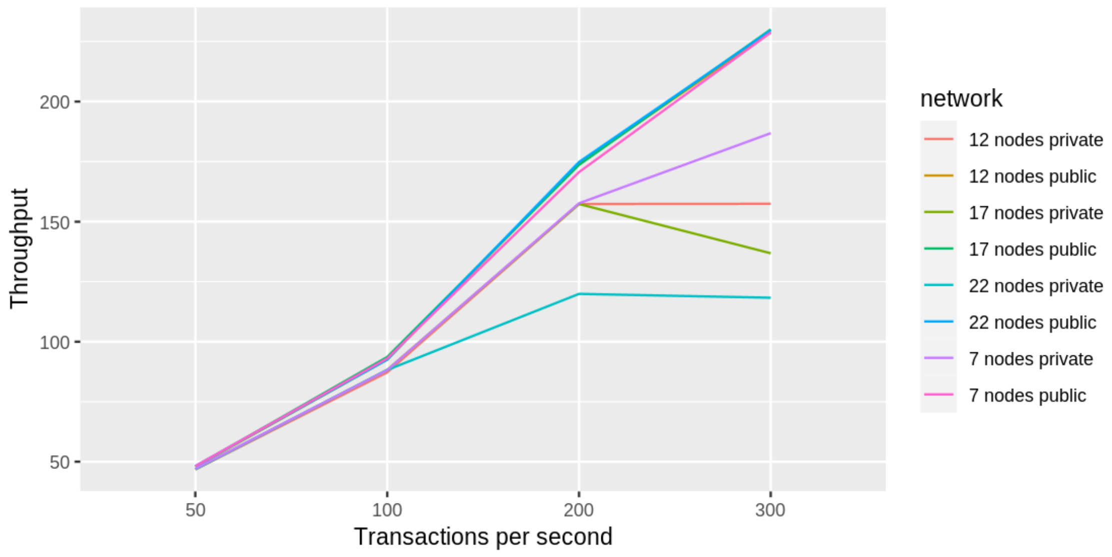


Figure 5.27: IBFT transfer throughput

In all operations, public transactions achieve a higher throughput than private ones, with results for public transactions showing similar throughput for all numbers of nodes.

With private transactions, in all operations, all networks achieve the same throughput until 100 TPS. Past that, results vary and in every scenario, networks with fewer nodes achieved a higher throughput than the rest.

Results for latency can be seen in Figures 5.28, 5.29, and 5.30.

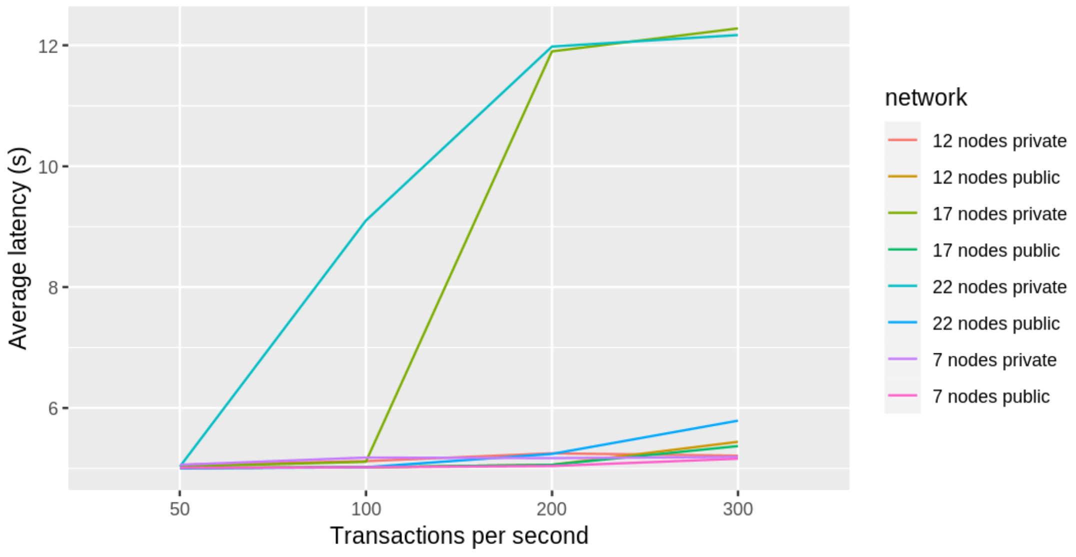


Figure 5.28: IBFT *open* latency

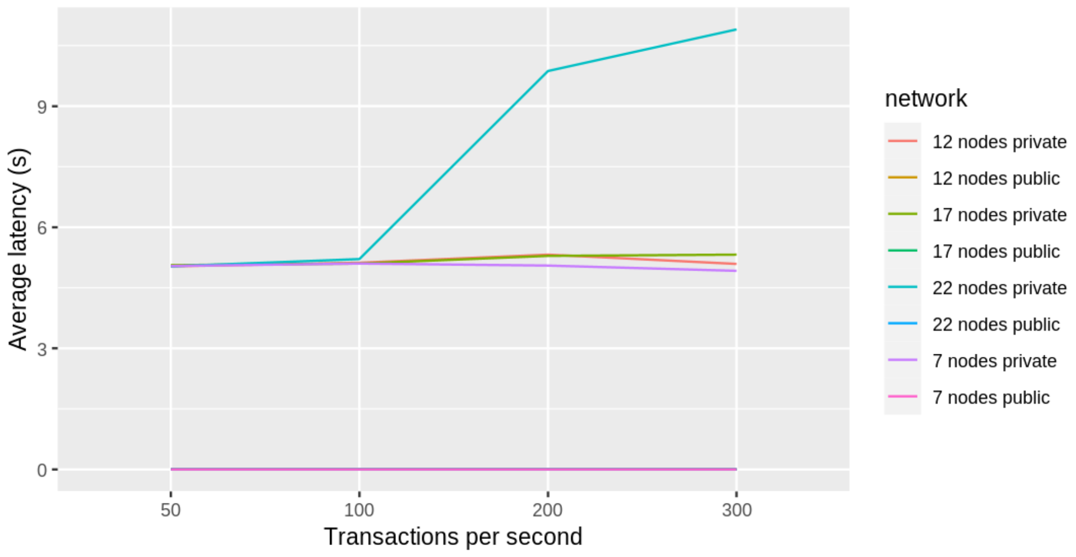


Figure 5.29: IBFT *query* latency

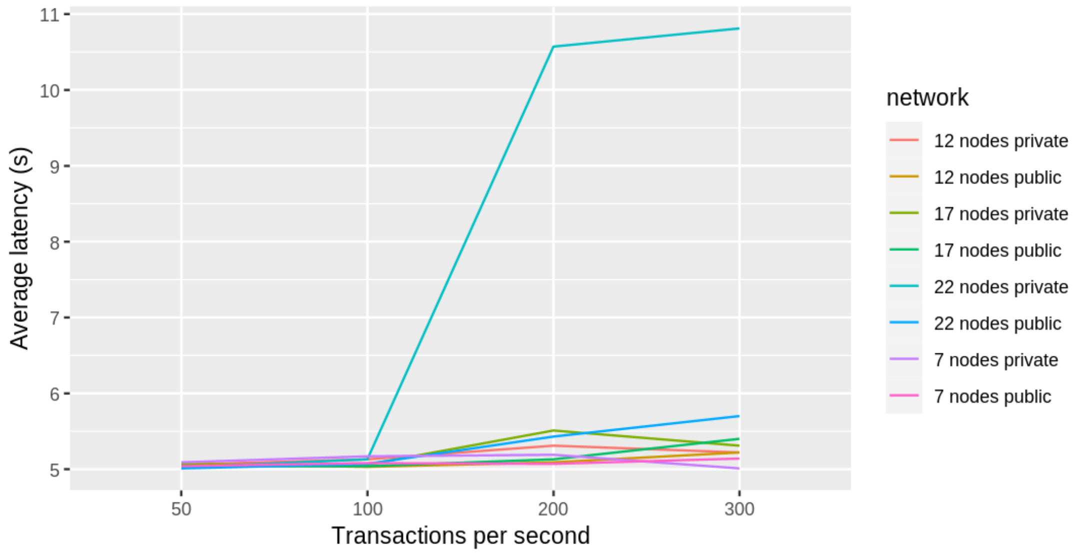


Figure 5.30: IBFT *transfer* latency

For *open*, latency is relatively similar for all network configurations, varying between 5 and 5.8 seconds. The exceptions are 17 and 22-node private networks, where the former jumps to 11.9 and 12.28 seconds past 200 TPS, and the latter to 9.10, 11.98, and 12.17 at 100, 200, and 300 TPS, respectively.

For *query*, all public networks have no latency. Private ones all have similar latency, ranging between 5 and 5.3 seconds, with the exception of the 22-node network, which jumps to 9.87 and 10.90 seconds at 200 and 300 TPS, respectively.

For *transfer*, all network configurations had latency ranging between 5 and 5.32 seconds. The exception, once again, was the 22-node network, which benchmarked 9.87 seconds at 200 TPS and 10.90 seconds at 300 TPS.

Figures 5.31, 5.32, and 5.33 show the CPU usage for IBFT across all operations.

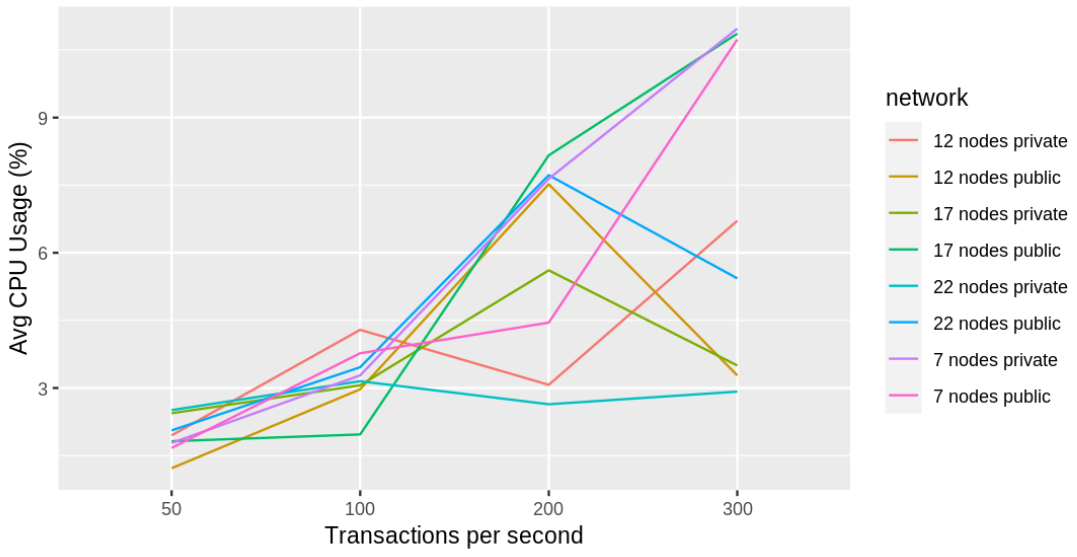


Figure 5.31: IBFT *open* CPU usage

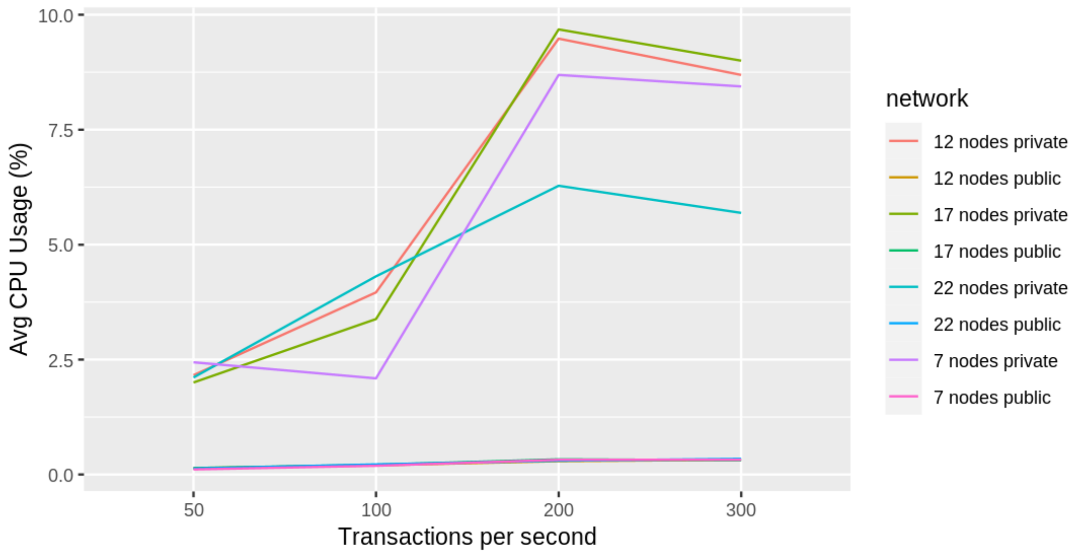


Figure 5.32: IBFT *query* CPU usage

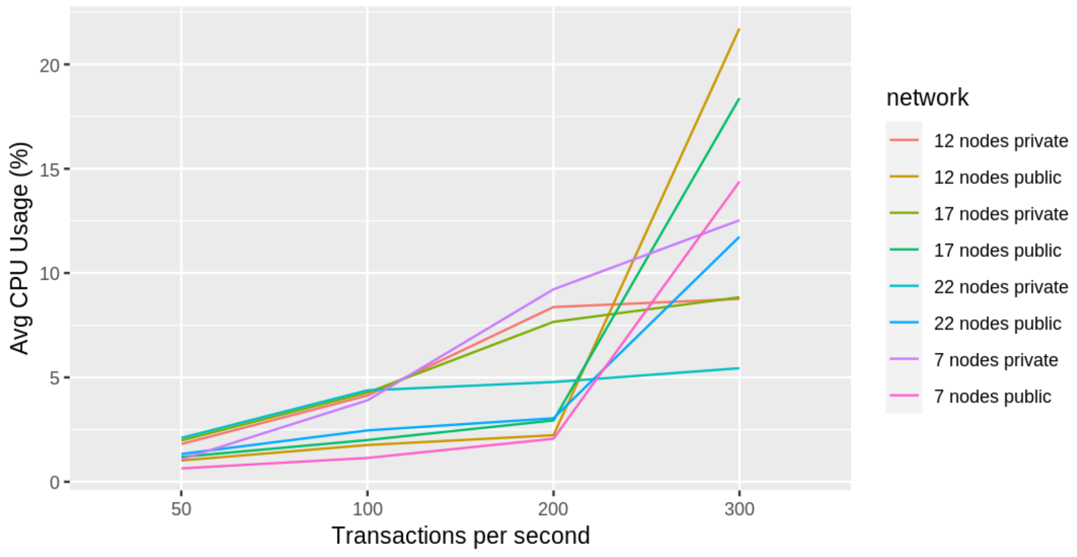


Figure 5.33: IBFT *transfer* CPU usage

For *open*, under 100 TPS all networks have roughly the same CPU usage, ranging between 1 and 4.3%. After that, although results varied across configurations, there was no clear advantage for either privacy setting, nor for any configuration of number-of-nodes. CPU usage was at maximum 11%, and it generally increased with the number of TPS.

For *query*, CPU usage was very low for public transactions, never passing 0.35%. Using private transactions, there was also no observable relationship between CPU usage and the number of nodes in the network, and CPU usage never surpassed 10%. RAM usage increased with the number of TPS, but seemed to stabilize past 200 TPS.

In *transfer* it was verified that CPU usage increased with the number of TPS for all network types, except for the 22-node private network, which always stayed around 5% past 100 TPS. Public networks saw lower CPU usage until 200 TPS, but all peaked to values higher than their private counterparts at 300 TPS. There was no observable relationship between the number of nodes in the network and the CPU usage.

Results for RAM usage can be found in Figures 5.34, 5.35, and 5.36.

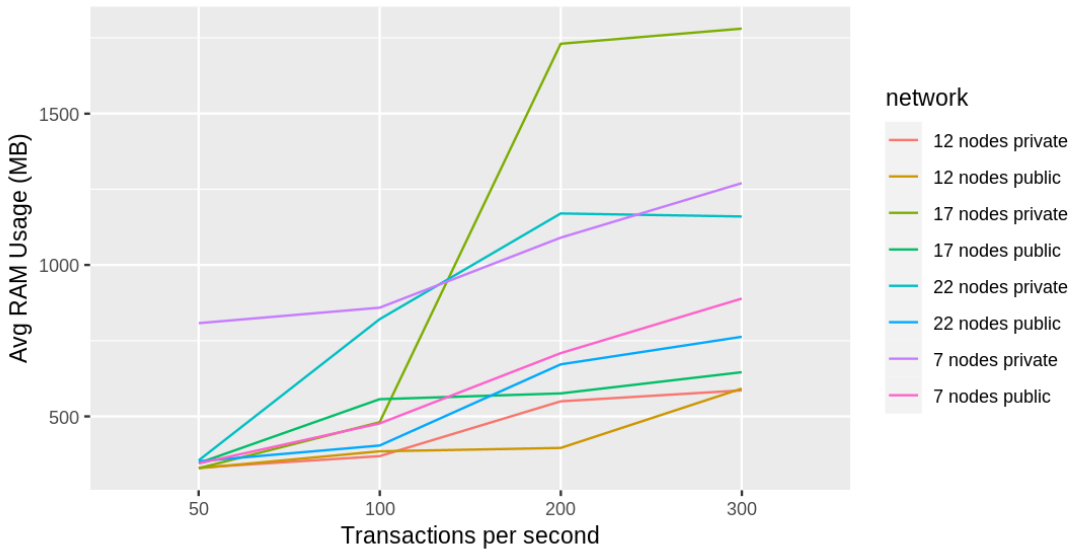


Figure 5.34: IBFT *open* RAM usage

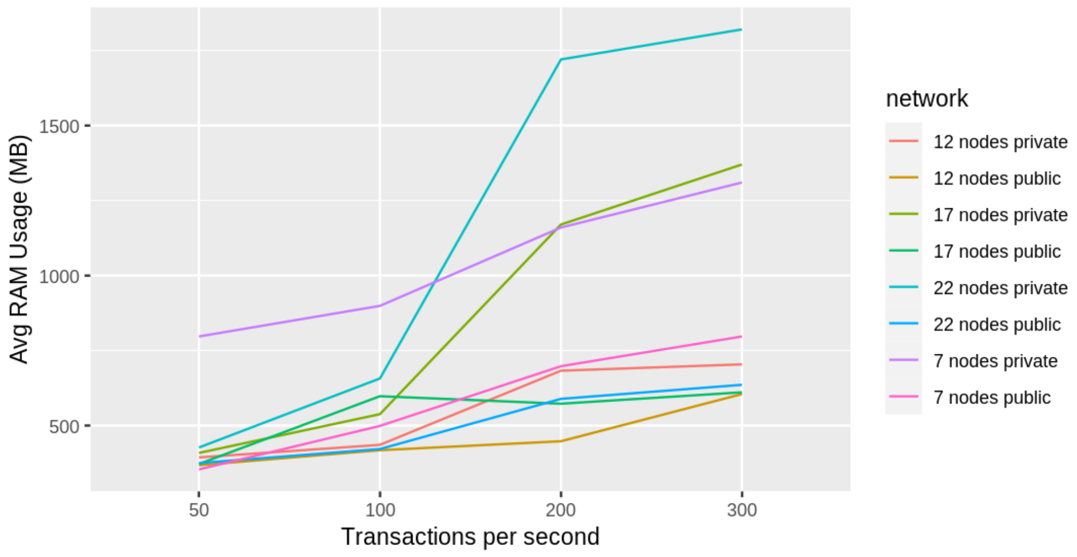


Figure 5.35: IBFT *query* RAM usage

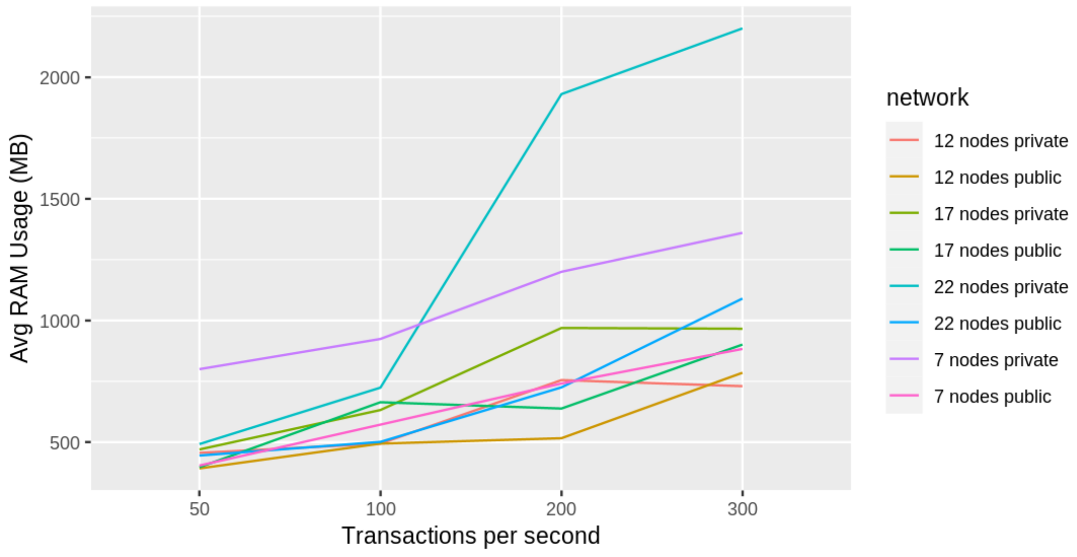


Figure 5.36: IBFT *transfer* RAM usage

All operations saw roughly the same RAM usage.

Private transactions had a higher RAM consumption than the public ones on all operations, as the latter never passed 1Gb, whereas the former reached past that on all operations and network configurations.

Whereas on public transactions networks with more nodes used less RAM per node overall. On private networks this was not the case, as different network configurations had higher or less RAM usages on different TPS and operations.

Fault-tolerance

Without introducing validator failures with pumba, IBFT also never had failures on public transactions. Failures using private transaction can be found in Table 5.5.

Table 5.5: IBFT transaction errors without chaos testing

Scenario	**Number of failures out of 5000**
17-node, 200 TPS, *open*	824
17-node, 300 TPS, *open*	866
22-node, 100 TPS, *open*	234
22-node, 200 TPS, *open*	3083
22-node, 200 TPS, *query*	491
22-node, 200 TPS, *transfer*	697
22-node, 300 TPS, *open*	3142
22-node, 300 TPS, *query*	663
22-node, 300 TPS, *transfer*	718

Using pumba, no differences were found for public networks, and the failures for private ones can be found in Table 5.6.

Table 5.6: IBFT transaction errors with chaos testing

Scenario	**Number of failures out of 5000**
17-node, 200 TPS, *open*	919
17-node, 300 TPS, *open*	808
22-node, 100 TPS, *open*	330
22-node, 200 TPS, *open*	3092
22-node, 200 TPS, *query*	190
22-node, 200 TPS, *transfer*	863
22-node, 300 TPS, *open*	3136
22-node, 300 TPS, *query*	727
22-node, 300 TPS, *transfer*	1034

Once again, there is no noticeable difference in the number transaction failures, and throughput and latency were also not affected by this change in any benchmarked scenario. As such, the chaos testing approach did not affect the error rate of transactions nor the general performance in networks using IBFT.

5.3.4 QBFT

This subsection will present the results obtained for the QBFT consensus algorithm for both performance and fault-tolerance.

Performance

Figures 5.37, 5.38, and 5.39 show the throughput results for QBFT, for all numbers of nodes with public and private transactions. The results are divided into the different operations benchmarked.

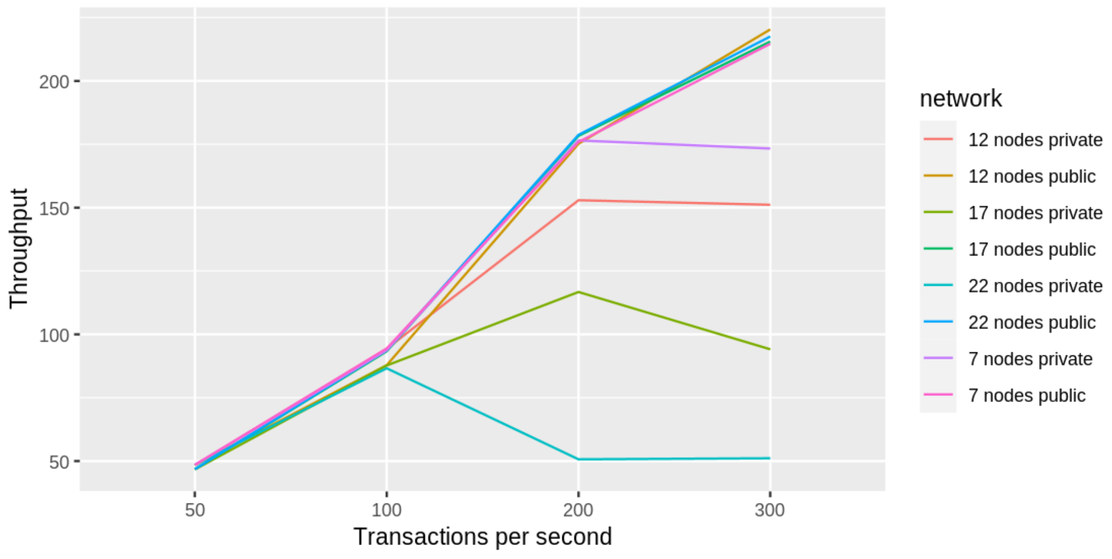


Figure 5.37: QBFT open throughput

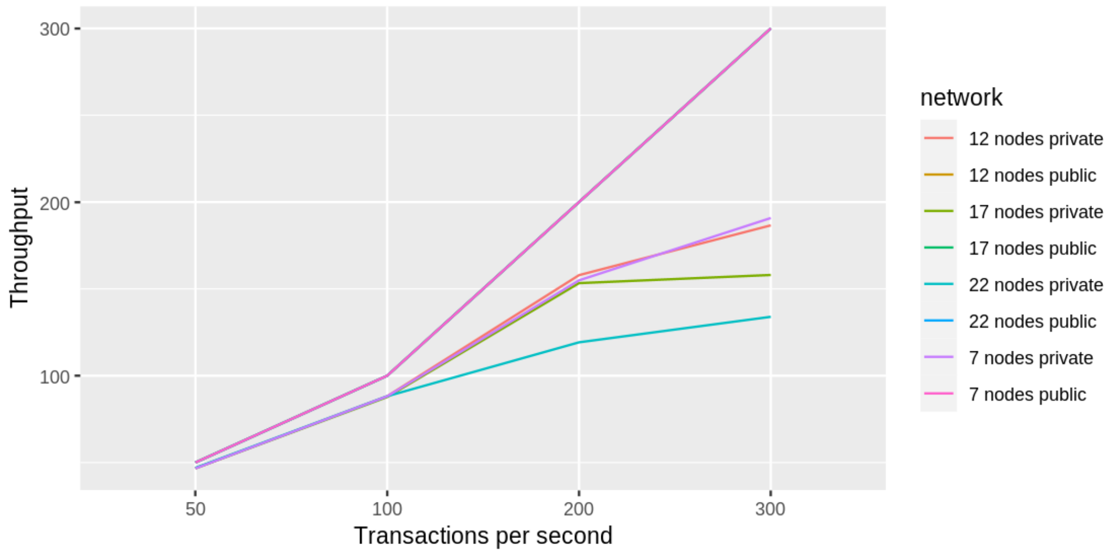


Figure 5.38: QBFT query throughput

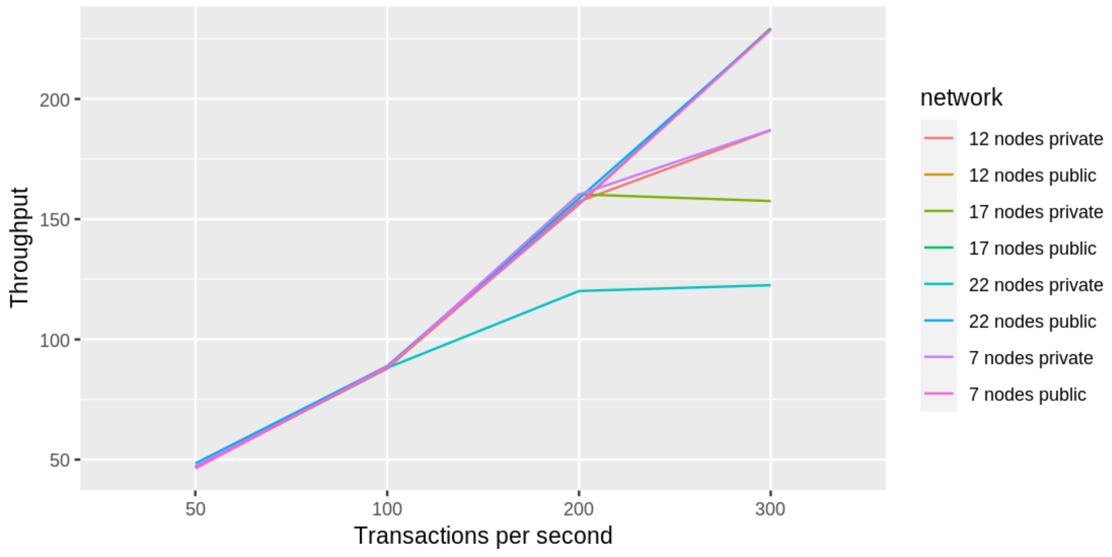


Figure 5.39: QBFT transfer throughput

Public transactions also achieve a higher throughput than private ones in all operations with QBFT. All public network configurations achieve similar throughput for all numbers of nodes, peaking at around 220 for *open*, 300 for *query*, and 230 for *transfer*.

With private transactions, in all operations, all networks achieve the same throughput until 100 TPS. Past that, results vary and in every scenario, networks with fewer nodes achieved a higher throughput than the rest.

The outlier here is the 22-node network, whose throughput decreased to just 50 on 200 and 300 TPS.

Results for latency can be seen in Figures 5.40, 5.41, and 5.42.

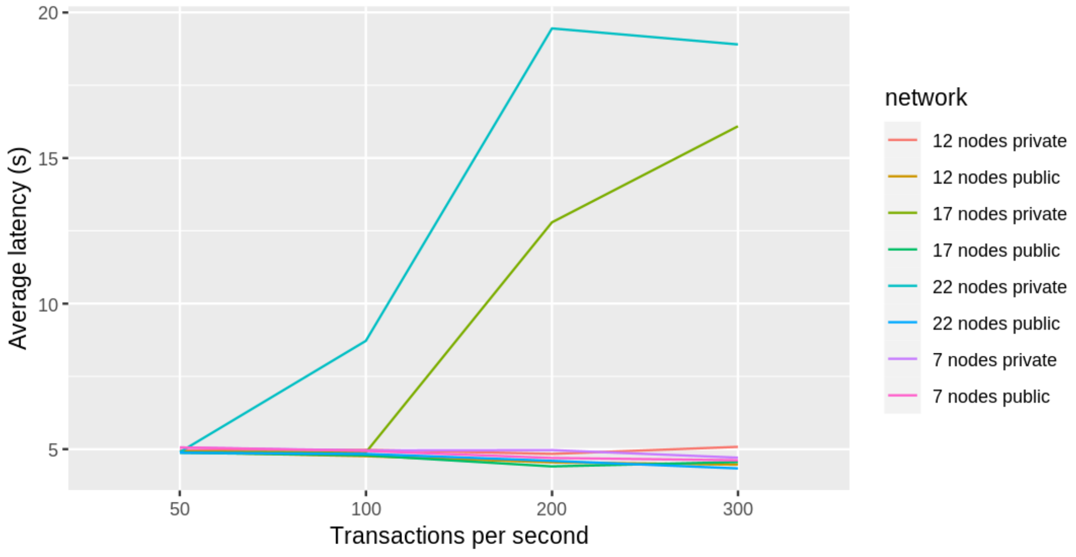


Figure 5.40: QBFT *open* latency

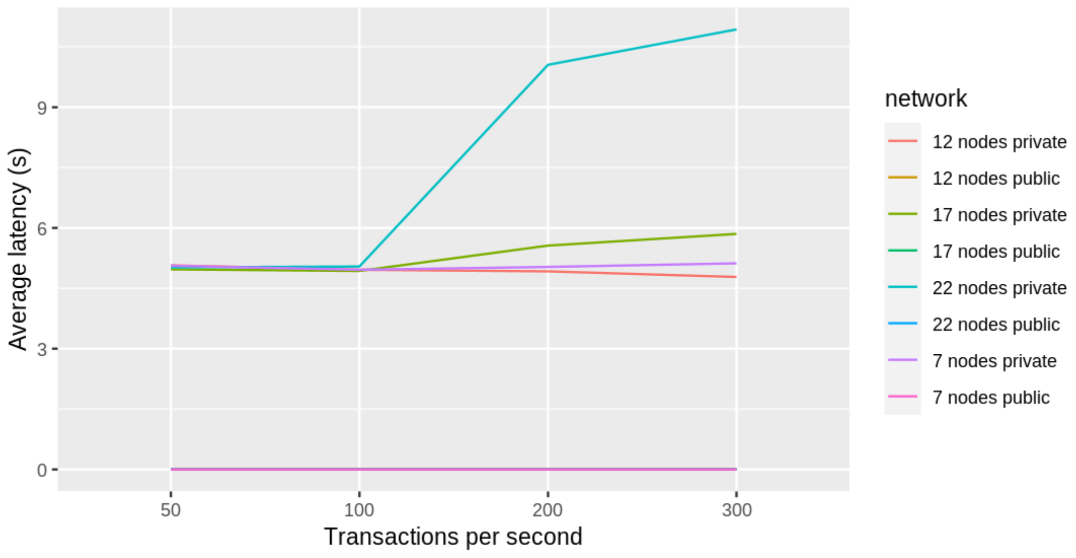


Figure 5.41: QBFT *query* latency

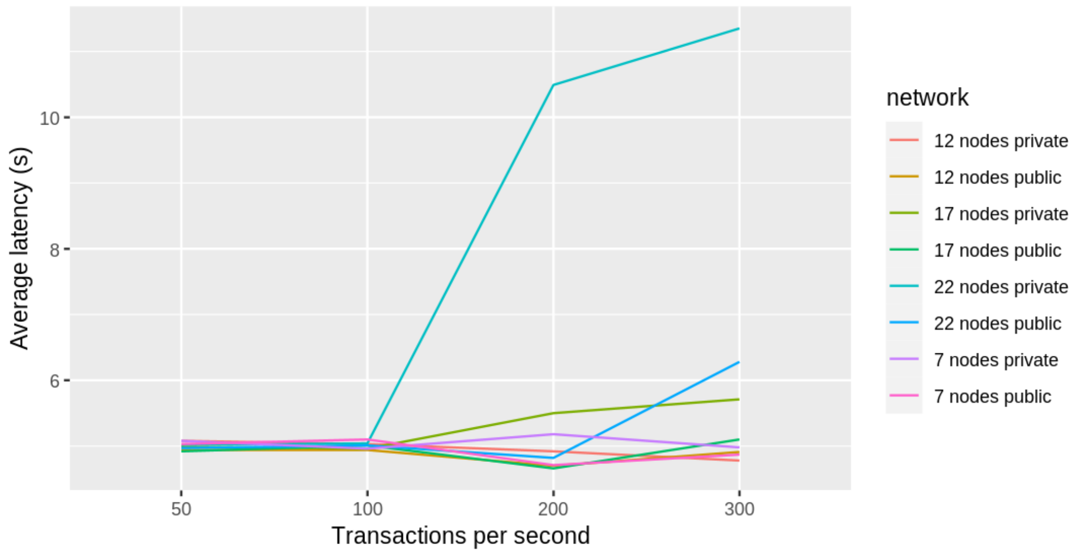


Figure 5.42: QBFT *transfer* latency

For *open*, latency is relatively similar for all network configurations, varying between 4.3 and 5 seconds. The exceptions are 17 and 22-node private networks, where the former jumps to 12.79 and 16.09 seconds past 200 TPS, and the latter to 8.72, 19.45, and 18.90 seconds at 100, 200, and 300 TPS, respectively.

For the *query* operation, all public networks have no latency. Private ones all have similar latency, ranging between 4.78 and 5.85 seconds, with the exception of the 22-node network, which jumps to 10.05 and 10.93 seconds at 200 and 300 TPS, respectively.

For *transfer*, all network configurations had similar latency across all TPS, ranging between 4.70 and 6.28 seconds. The exception was again the 22-node network, which had 10.49 seconds at 200 TPS and 11.35 seconds at 300 TPS.

Figures 5.43, 5.44, and 5.45 show the CPU usage for QBFT across all operations.

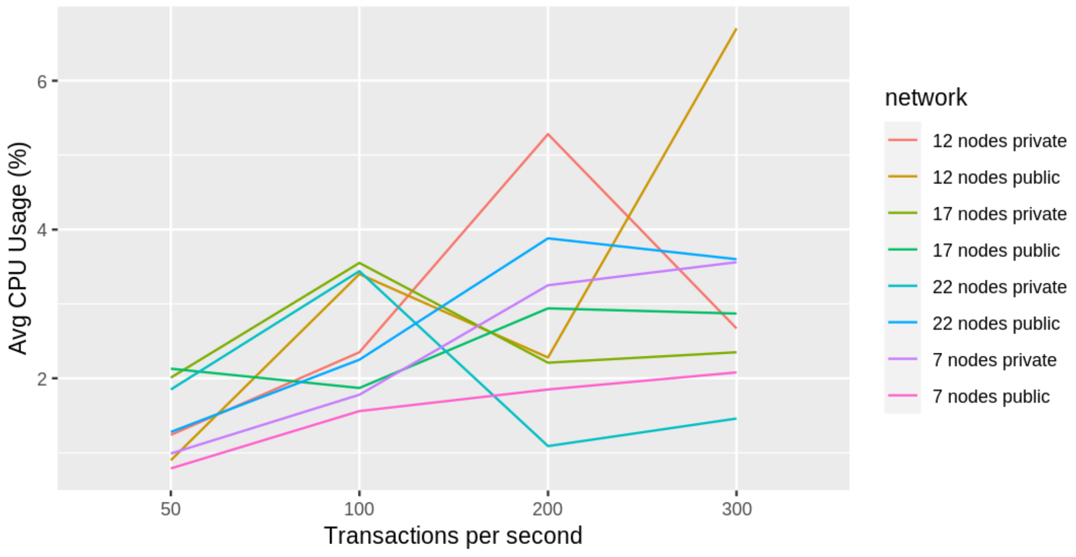


Figure 5.43: QBFT *open* CPU usage

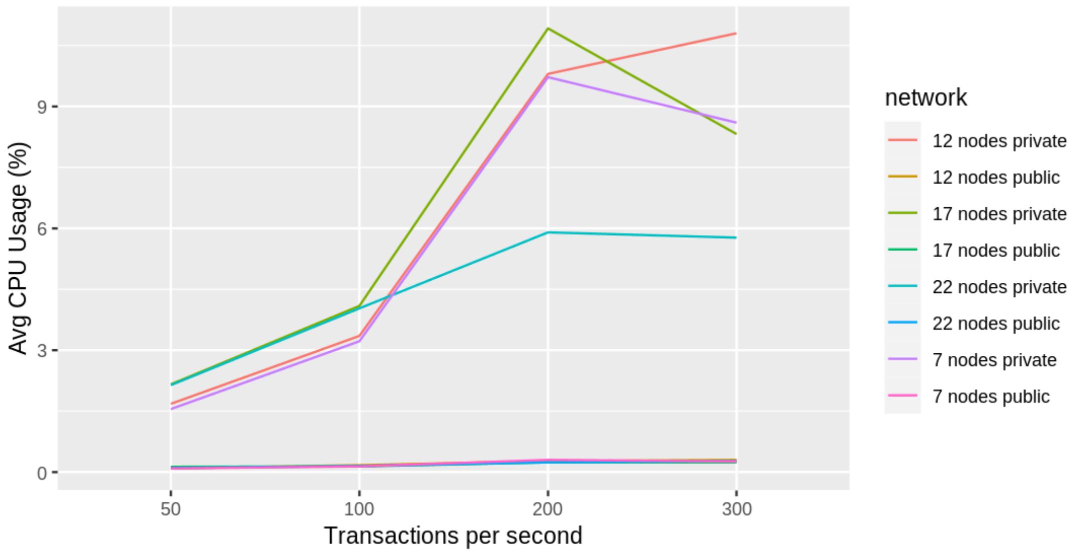


Figure 5.44: QBFT *query* CPU usage

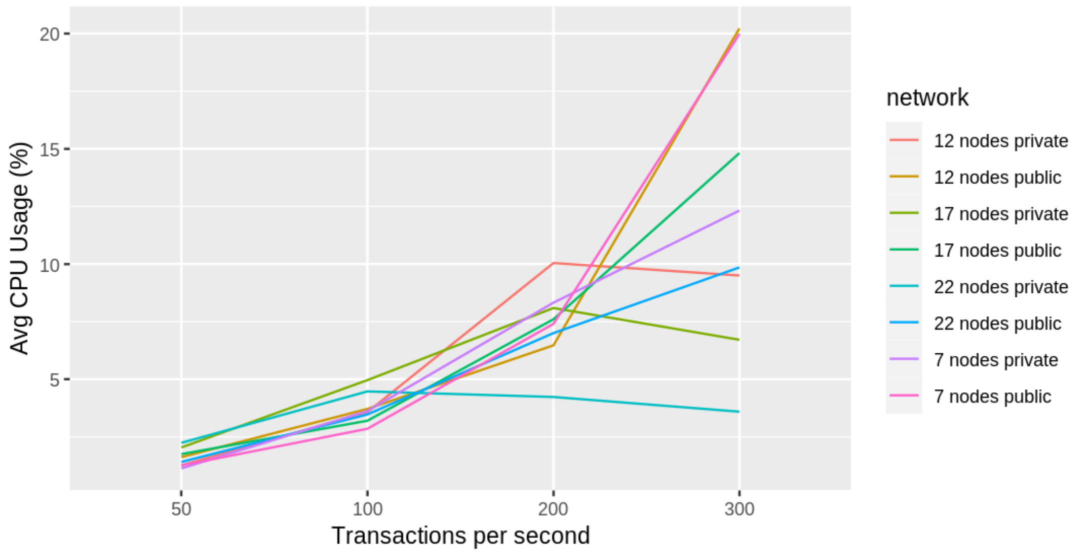


Figure 5.45: QBFT *transfer* CPU usage

For *open*, CPU usage generally increased as the number of TPS increased. There was no clear advantage of public or private transactions here, nor the number of nodes in the network. CPU usage was at maximum 6.7%.

For *query*, CPU usage was very low for public transactions, never passing 0.3%. With private transactions, there was no observable relationship between CPU usage and the number of nodes in the network, and CPU usage never passed 11%.

In *transfer*, it was verified that CPU usage increased with the number of TPS for all network types, except for the 22-node private network, which always stayed under 5%. This, however, can be explained by its very high latency in this operation, as previously observed. CPU usage was higher in this operation, reaching 20% in the 7 and 12-node private networks.

Results for RAM usage can be found in Figures 5.46, 5.47, and 5.48.

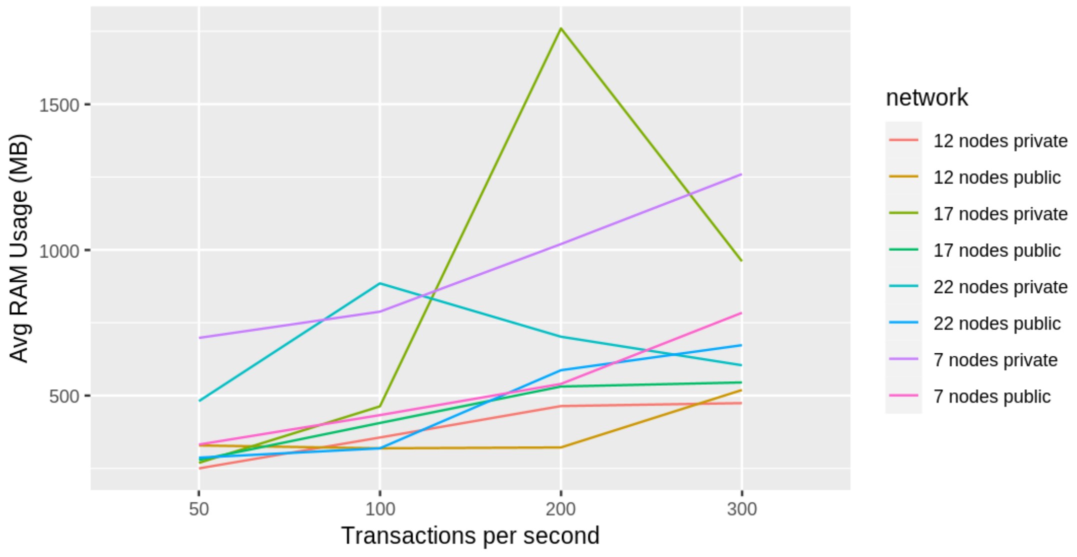


Figure 5.46: QBFT *open* RAM usage

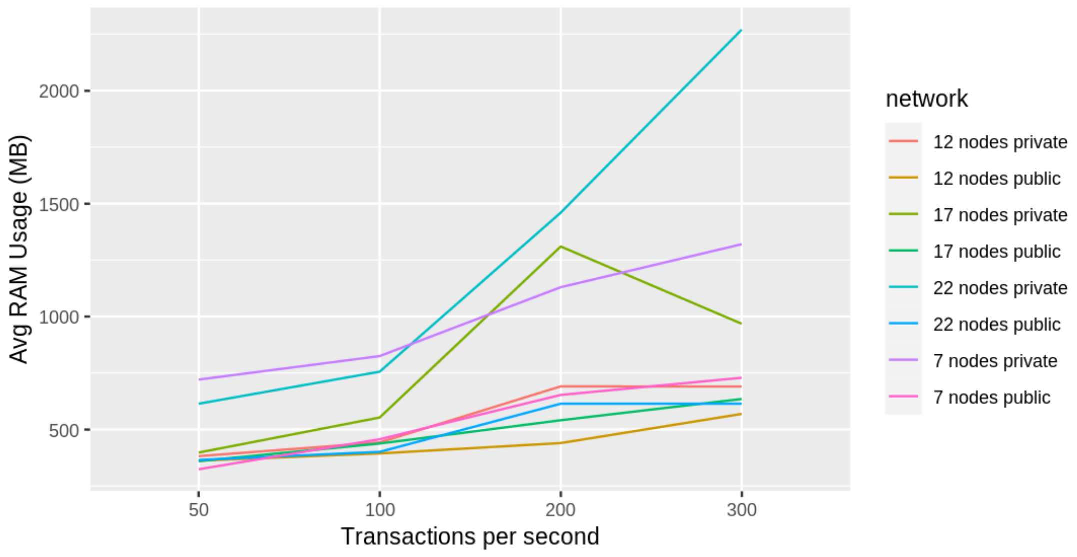


Figure 5.47: QBFT *query* RAM usage

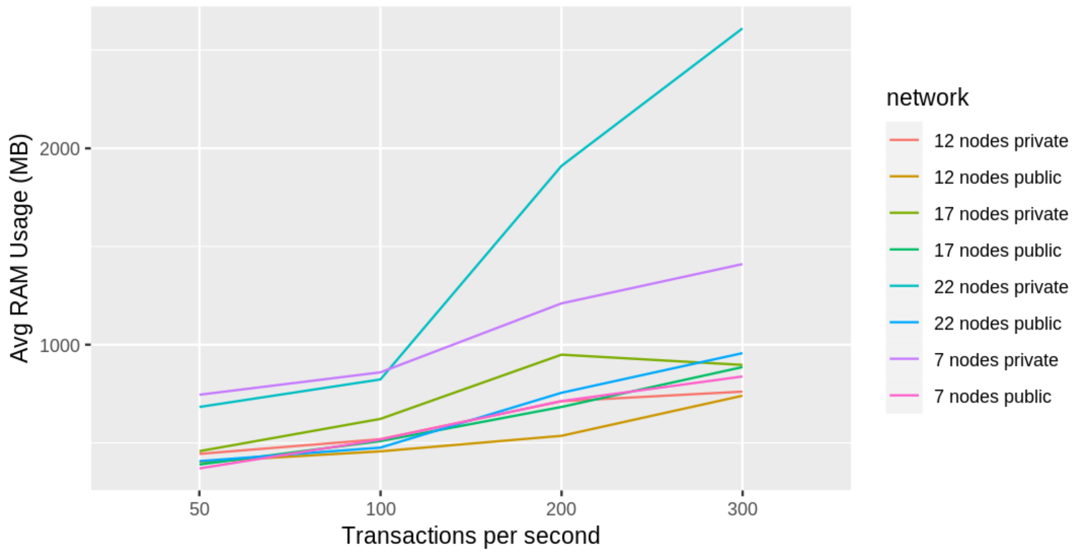


Figure 5.48: QBFT *transfer* RAM usage

For the *open* operation, RAM usage for all network configurations ranged between 279MB and 784Mb, increasing with the number of TPS. The outliers are the 7 and 17-node private networks, where the former saw consistently higher RAM usage across all TPS, and the latter had a peak of 1260MB at 200 TPS.

For *query*, RAM usage again grew with the number of TPS. Public transactions saw less RAM usage than private ones across all configurations, never reaching past 720MB. With private transactions, the 22-node private network saw by far the highest peak, using more than 2GB of RAM at 300 TPS.

Fault-tolerance

Without introducing validator failures with pumba, QBFT never had failures on public transactions, and only saw failures with private transactions in the scenarios found in Table 5.7.

Table 5.7: QBFT transaction errors without chaos testing

Scenario	**Number of failures out of 5000**
17-node, 200 TPS, *open*	972
17-node, 300 TPS, *open*	1109
22-node, 100 TPS, *open*	226
22-node, 200 TPS, *open*	2920
22-node, 200 TPS, *query*	494
22-node, 200 TPS, *transfer*	751
22-node, 300 TPS, *open*	3061
22-node, 300 TPS, *query*	878
22-node, 300 TPS, *transfer*	760

Using chaos testing, there were no changes in public transactions, and the results for private ones can be seen in Table 5.8.

Table 5.8: QBFT transaction errors with chaos testing

Scenario	**Number of failures out of 5000**
17-node, 200 TPS, *open*	771
17-node, 300 TPS, *open*	807
22-node, 100 TPS, *open*	37
22-node, 200 TPS, *open*	3070
22-node, 200 TPS, *query*	782
22-node, 200 TPS, *transfer*	667
22-node, 300 TPS, *open*	2998
22-node, 300 TPS, *query*	1022
22-node, 300 TPS, *transfer*	1218

As can be seen there is no noticeable difference in the number transaction failures. Performance was also not affected, as throughput and latency remained consistent with the other results.

5.3.5 Comparison

With the results collected above, the various consensus algorithms can be compared to each other on the different scenarios benchmarked.

For public transactions, Raft was the clear winner across all performance metrics, having reached near perfect throughput for all node-number configurations and operations. Figure 5.49 shows the comparison of throughput for *open*, which was the less performant operation for all consensus algorithms.

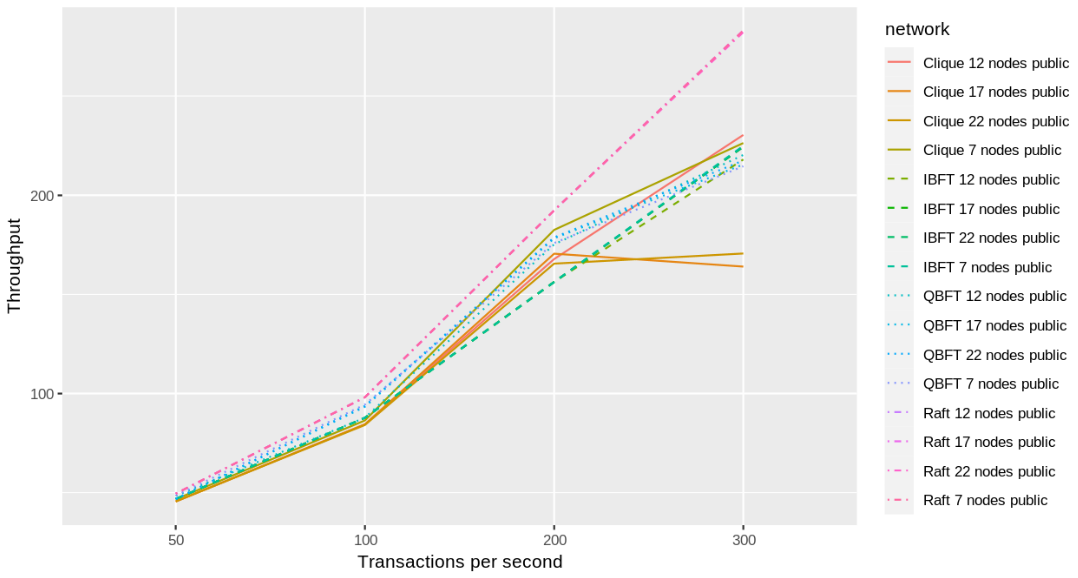


Figure 5.49: Public transaction throughput comparison for *open*

Raft also achieved the lowest latency in these transactions, never passing 1.1 seconds. Figure 5.50 shows the comparison of latency for *open* to illustrate this.

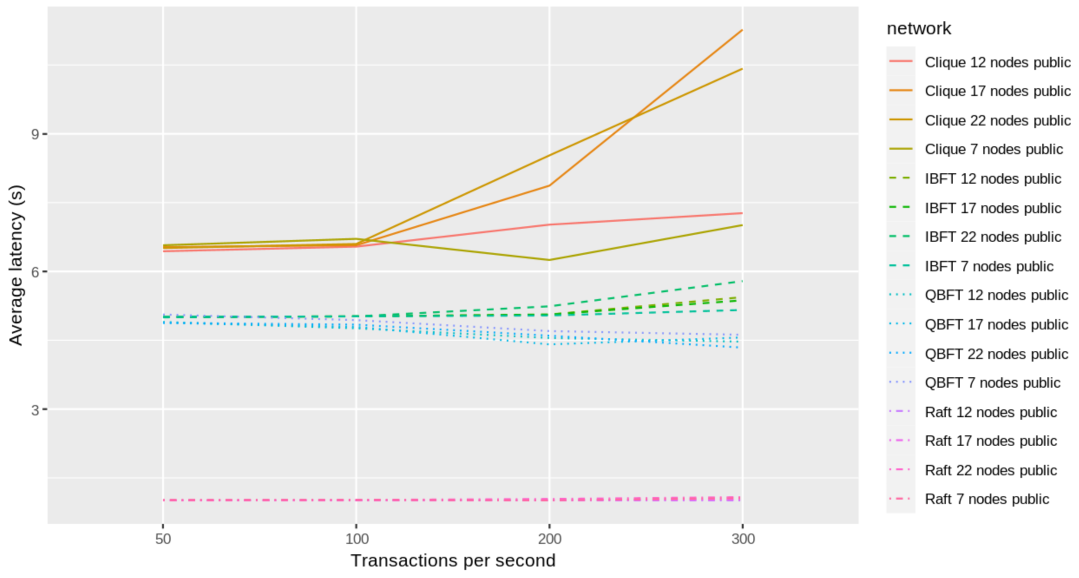


Figure 5.50: Public transaction latency comparison for *open*

Resource usage was also the lowest for this algorithm, with the average CPU usage never passing 5% and average RAM usage never passing 681MB.

Clique was the worst performer, having had the highest resource usage, while also having the lowest throughput.

IBFT and QBFT essentially tied for performance in all benchmarks for public transactions.

Regarding private transactions, which is where most of Quorum's added functionality is, and thus better representative of Quorum's real-world use cases, Raft was again the winner in performance, having also beaten the rest in all benchmarks. To illustrate this for both write and read operations, Figure 5.51 shows the comparison of throughput for *open* and Figure 5.52 shows the comparison of throughput for *query*.

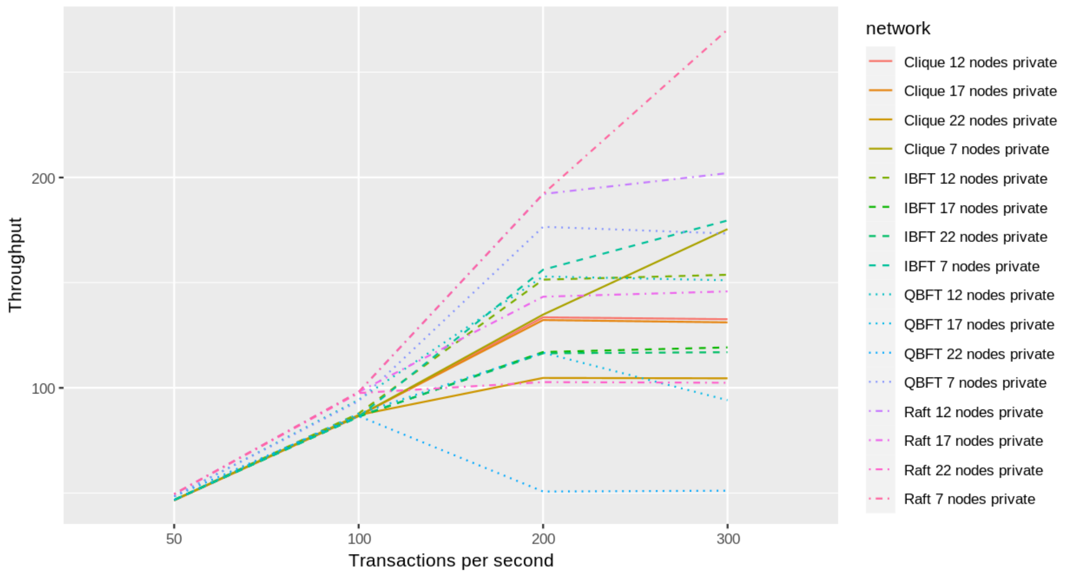


Figure 5.51: Private transaction throughput comparison for *open*

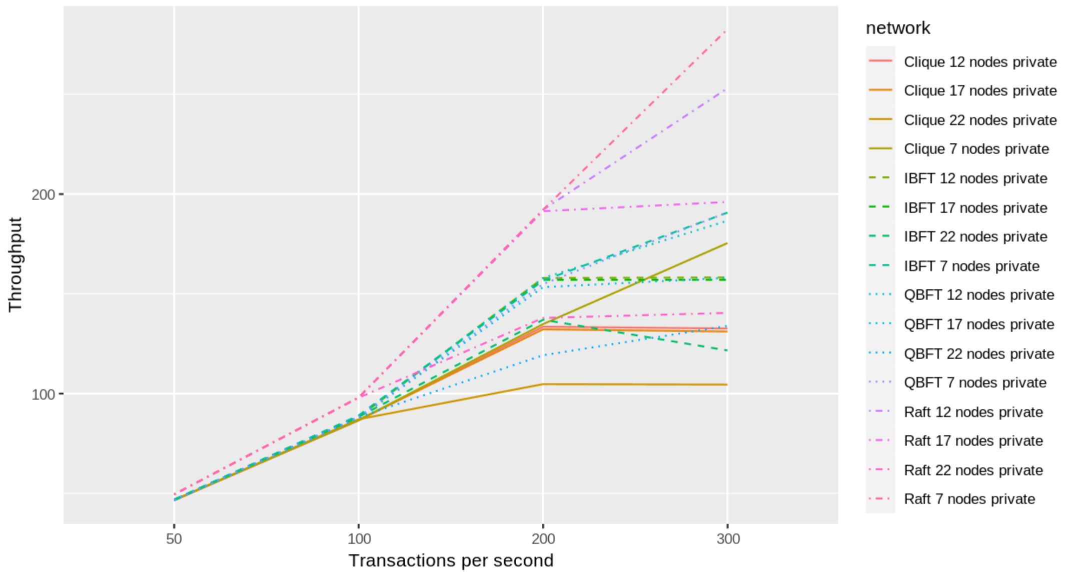


Figure 5.52: Private transaction throughput comparison for *query*

Clique was once again the worst performer despite having a throughput and latency comparable to IBFT and QBFT, as it used considerably more resources than the rest. Table 5.9 shows the average CPU usage for Clique, IBFT and QBFT, and Table 5.10 shows the same data for RAM usage.

Table 5.9: Average CPU usage for Clique, IBFT, and QBFT

Algorithm	**Operation**	**Avg CPU usage across all private transaction runs**
Clique	*open*	3.8%
IBFT	*open*	4.8%
QBFT	*open*	2.5%
Clique	*query*	7.27%
IBFT	*query*	5.52%
QBFT	*query*	5.75%
Clique	*transfer*	8.27%
IBFT	*transfer*	5.58%
QBFT	*transfer*	5.38%

Table 5.10: Average RAM usage for Clique, IBFT, and QBFT

Algorithm	**Operation**	**Avg RAM usage across all private transaction runs**
Clique	*open*	1376MB
IBFT	*open*	856MB
QBFT	*open*	715MB
Clique	*query*	1544MB
IBFT	*query*	905MB
QBFT	*query*	908MB
Clique	*transfer*	1604MB
IBFT	*transfer*	943MB
QBFT	*transfer*	975MB

As it can be seen, despite Clique having used less CPU on average than IBFT for the *open* operation, it registered a higher CPU usage on all other operations and it required considerably more RAM across all operations.

Between IBFT and QBFT, once again these had very similar performances across all benchmarks. QBFT had worse throughput in write operations on the 17 and 22-node networks, but given the number of transaction failures in these scenarios, their significance is decreased. The algorithms also had very similar resource usage across all scenarios.

For fault tolerance, with private transactions, as observed, all consensus algorithms fail under the same scenarios, being unable to scale past 17-nodes without having a very high number of transaction failures, making them not viable at this scale. Introducing chaos testing had no effect on either of the consensus algorithms, and the number of failures was consistent across all of them, with none having a clear advantage over the rest based on the benchmarks done.

Based on the results found, for Q1, the hypothesis 1 (HA1) was verified, as the choice of consensus algorithm dramatically affected performance in networks for all scenarios. For Q2, under the scenarios tested, no difference was found between consensus algorithms for fault-tolerance, which negates hypothesis 1 (HB1) and consequently confirms the null hypothesis (HB0).

Chapter 6

Conclusion

This chapter presents a summary of the work done, analyzing the results achieved, shortcomings and limitations, as well as future improvements. Finally, a personal appreciation of the project is done.

6.1 Overview

Based on the results obtained, Raft is the most performant consensus algorithm. This result is not surprising, as Raft does not offer byzantine fault tolerance and thus can focus solely on achieving higher throughput while offering only crash fault tolerance and immediate finality.

Regarding fault-tolerance, with the scenarios tested, no difference was found between consensus algorithms. However, since IBFT and QBFT provide theoretical byzantine fault-tolerance, and QBFT was introduced as a replacement for IBFT in order to fix previously identified issues with the latter that caused network deadlocks, QBFT should still be the better choice in this metric, although these properties were not tested in this body of work.

Since QBFT achieved the same performance as IBFT across all metrics, there is no reason to use IBFT over QBFT, as the new algorithm fixes the aforementioned issues without compromising performance.

None of the consensus algorithms in GoQuorum scaled past 17 nodes under private transactions. However, benchmarks were for the worst possible scenario, and thus implementations where most transactions are only shared across a few members might be able to scale past this number.

Analyzing use cases for each consensus algorithm, Raft is more appropriate for scenarios where performance is important and the network administrators can trust every party in the network to not be compromised. For scenarios where it can not be guaranteed that validators will always act benevolently, or where security is more important than raw performance, QBFT should be the algorithm chosen, as it provides a good compromise between performance and byzantine fault-tolerance.

For development purposes, both QBFT and Raft perform fast enough with a low number of nodes with relatively low resource usage for a development machine, and thus whichever algorithm is chosen for production should be the one chosen for development and testing. This will also ensure that quirks in the configurations of each algorithm are caught before reaching production.

6.2 Goals achieved

The goal of obtaining empirical data on how each GoQuorum's consensus algorithm performs under the same scenarios was achieved. The strategy chosen to test fault-tolerance did not find differences across consensus algorithms, as the chaos testing solution used was not enough to affect the networks. Nevertheless, it was proven that when high availability is used for validators, GoQuorum networks can handle validator crashes without compromising performance, regardless of the consensus algorithm employed.

As such, this work has provided concrete data on how each consensus algorithm compares under different scenarios and for different operations, which can help interested parties inform their decision when deciding whether to adopt these technologies.

6.3 Threats to validity

Some issues and threats have been identified in the development of this project that could impact the results obtained.

First, the benchmarks were run in a single machine, which is the best case scenario for communication between distributed nodes. Results then might not fully apply and vary a lot in a scenario where nodes are distributed across different machines, particularly in different regions of the globe, which can be a real use case for big enterprises.

Finally, all metrics collected came from the version 0.5 of Hyperledger Caliper, and thus results have a hard dependency on this implementation. If this tool is found to have bugs in its data collection logic, then the results obtained will be compromised.

6.4 Future work

As mentioned in the introductory chapter, this work does not explore the security implication of each consensus algorithm outside their theoretical byzantine fault tolerance. Since this metric is increasingly important, especially for enterprises, a body of work exploring this subject is also in the community's interest.

Regarding fault-tolerance, the strategy employed did not trigger failures in the network due to the high number of validators used, and it did not test how quickly these can recover after crashing on each consensus algorithm. As such, it can be worthwhile to further explore this topic by testing different scenarios where validator nodes are crashed but allowed to instantly recover. Another evolution on this topic would be to evaluate exactly when and how each of these consensus algorithms can fail, in order to understand the true minimum resource requirements for each.

Additionally, the results showed that the consensus algorithms failed to scale past 17 nodes at 200 TPS or higher. However, it did not test the maximum number of transactions per second that the 7 or 12-node networks can handle before failing, and thus it did not fully cover the scalability of GoQuorum. With that said, this would be a very extensive body of work and its eventual benefits are unclear, since real-world scenarios can have very different transactions which can have different benchmark results, as it was observed in the difference between the *open* and *query* operations.

6.5 Personal appreciation

The development of this work has been a very compelling and worthwhile experience for the author, as it granted the opportunity to learn about and explore the state of the art in blockchain technology. It has, then, been a very enriching personal project, full of learning and exploration of new and interesting topics.

Bibliography

[1] *Deloitte's 2021 Global Blockchain Survey*, [Online; accessed 19. Feb. 2023], Feb. 2023. [Online]. Available: https://www2.deloitte.com/us/en/insights/topics/understanding-blockchain-potential/global-blockchain-survey.html.

[2] S. Nakamoto, *Bitcoin: A Peer-to-Peer Electronic Cash System*, [Online; accessed 29. Apr. 2023], Mar. 2008. [Online]. Available: https://bitcoin.org/bitcoin.pdf.

[3] *Key Challenges to the Enterprise Blockchain Technology Market (and Solutions) | LinkedIn*, [Online; accessed 21. May 2023], May 2023. [Online]. Available: https://www.linkedin.com/pulse/key-challenges-enterprise-blockchain-technology-market-singh.

[4] V. Buterin, "Ethereum: A Next-Generation Smart Contract and Decentralized Application Platform.," en, 2014.

[5] A. Baliga, I. Subhod, P. Kamat, and S. Chatterjee, *Performance Evaluation of the Quorum Blockchain Platform*, en, arXiv:1809.03421 [cs], Jul. 2018. [Online]. Available: http://arxiv.org/abs/1809.03421 (visited on 02/04/2023).

[6] B. Dimitrov, "How Strong Enterprise Interest In 2020 Is Pushing Blockchain Technology Further," *Forbes*, Feb. 2020. [Online]. Available: https://www.forbes.com/sites/biserdimitrov/2020/02/03/how-is-strong-enterprise-interest-in-2020-pushing-blockchain-technology-further/?sh=1c360798139d.

[7] *Enterprise Blockchain Platforms - Current State and Development.* [Online; accessed 19. Feb. 2023], Nov. 2022. [Online]. Available: https://limechain.tech/blog/enterprise-blockchain-platforms.

[8] A. Garcia, "IBM is betting big on blockchain technology. Is it worth the risk?" *CNNMoney*, Sep. 2018. [Online]. Available: https://money.cnn.com/2018/09/06/technology/ibm-blockchain-gamble/index.html.

[9] W. Henry and L. Pawczuk, "Blockchain: Ready for business," *Deloitte Insights*, Jan. 2023. [Online]. Available: https://www2.deloitte.com/us/en/insights/focus/tech-trends/2022/blockchain-trends.html.

[10] J. Swift, "The 2023 Forbes Blockchain 50 Reveals Top Enterprises Continuing To Invest In Future Blockchain Innovations," *Forbes*, Feb. 2023. [Online]. Available: https://www.forbes.com/sites/forbes-spotlights/2023/02/07/the-2023-forbes-blockchain-50-reveals-top-enterprises-continuing-to-invest-in-future-blockchain-innovations/?sh=54c68620477e.

[11] D. A. Disparte, "Why Enterprise Blockchain Projects Fail," *Forbes*, May 2019. [Online]. Available: https://www.forbes.com/sites/dantedisparte/2019/05/20/why-enterprise-blockchain-projects-fail/?sh=d85d5e04b96b.

[12] *Corda*, [Online; accessed 29. Apr. 2023], Apr. 2023. [Online]. Available: https://r3.com/products/corda.

[13] Hyperledger, *Fabric*, [Online; accessed 29. Apr. 2023], Apr. 2023. [Online]. Available: https://github.com/hyperledger/fabric.

[14] ConsenSys, *ConsenSys Quorum | ConsenSys*, [Online; accessed 29. Apr. 2023], Apr. 2023. [Online]. Available: https://consensys.net/quorum.

[15] 101 Blockchains, "Top Blockchain Platforms and Enterprise Solutions to Choose From," *101 Blockchains*, Aug. 2022. [Online]. Available: https://101blockchains.com/blockchain-platforms.

[16] J. Polge, J. Robert, and Y. Le Traon, "Permissioned blockchain frameworks in the industry: A comparison," en, *ICT Express*, vol. 7, no. 2, pp. 229–233, Jun. 2021, issn: 24059595. doi: 10.1016/j.icte.2020.09.002. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S2405959520301909 (visited on 10/05/2022).

[17] Hyperledger, *The Ordering Service — hyperledger-fabricdocs main documentation*, [Online; accessed 15. Apr. 2023], Apr. 2023. [Online]. Available: https://hyperledger-fabric.readthedocs.io/en/release-2.5/orderer/ordering_service.html.

[18] R3, *Consensus*, [Online; accessed 15. Apr. 2023], Jan. 2023. [Online]. Available: https://docs.r3.com/en/platform/corda/4.10/enterprise/key-concepts-consensus.html#summary.

[19] *Configuring a JPA notary backend*, [Online; accessed 15. Apr. 2023], Apr. 2020. [Online]. Available: https://docs.r3.com/en/platform/corda/4.8/enterprise/notary/installing-jpa.html.

[20] *Setting up a notary service*, [Online; accessed 15. Apr. 2023], Apr. 2020. [Online]. Available: https://docs.r3.com/en/platform/corda/4.10/enterprise/notary/running-a-notary.html#crash-fault-tolerant-experimental.

[21] *Comparing PoA consensus protocols - GoQuorum - latest*. [Online]. Available: https://consensys.net/docs/goquorum/en/latest/concepts/consensus/comparing-poa/ (visited on 11/01/2022).

[22] M. Nelson, *What is ConsenSys Quorum? | ConsenSys*, [Online; accessed 26. Feb. 2023], Jun. 2021. [Online]. Available: https://consensys.net/blog/quorum/what-is-consensys-quorum.

[23] M. Leising, *JPMorgan Sells Quorum Blockchain Unit, Takes Stake in ConsenSys*, [Online; accessed 26. Feb. 2023], Aug. 2020. [Online]. Available: https://news.bloomberglaw.com/payroll/jpmorgan-sells-quorum-blockchain-unit-takes-stake-in-consensys?context=article-related.

[24] ConsenSys, *ConsenSys/quorum*, original-date: 2016-11-14T05:42:57Z, Feb. 2023. [Online]. Available: https://github.com/ConsenSys/quorum (visited on 02/15/2023).

[25] *Introduction to smart contracts*, en. [Online]. Available: https://ethereum.org (visited on 10/30/2022).

[26] *Proof-of-work (PoW)*, en. [Online]. Available: https://ethereum.org (visited on 11/01/2022).

[27] D. Li, W. E. Wong, and J. Guo, "A Survey on Blockchain for Enterprise Using Hyperledger Fabric and Composer," in *2019 6th International Conference on Dependable Systems and Their Applications (DSA)*, Jan. 2020, pp. 71–80. doi: 10.1109/DSA.2019.00017.

[28] *Proof-of-stake (PoS)*, en. [Online]. Available: https://ethereum.org (visited on 02/05/2023).

[29] *Architecture - GoQuorum - latest*. [Online]. Available: https://consensys.net/docs/goquorum/en/latest/concepts/architecture/ (visited on 11/01/2022).

[30] *Patricia Merkle Trees*, en. [Online]. Available: https://ethereum.org (visited on 11/01/2022).

[31] ConsenSys, *quorum*, [Online; accessed 29. Apr. 2023], Apr. 2023. [Online]. Available: https://github.com/ConsenSys/quorum.

[32] Hyperledger, *besu*, [Online; accessed 29. Apr. 2023], Apr. 2023. [Online]. Available: https://github.com/hyperledger/besu.

[33] *Privacy - GoQuorum - latest*. [Online]. Available: https://consensys.net/docs/goquorum//en/latest/concepts/privacy/#enclave (visited on 01/29/2023).

[34] *Tessera Private Transaction Manager*. [Online]. Available: https://docs.tessera.consensys.net/en/stable/ (visited on 01/29/2023).

[35] *Private transaction lifecycle - GoQuorum - latest*. [Online]. Available: https://consensys.net/docs/goquorum/en/latest/concepts/privacy/private-transaction-lifecycle/ (visited on 01/29/2023).

[36] *Permissioning - GoQuorum - latest*. [Online]. Available: https://consensys.net/docs/goquorum//en/latest/concepts/permissions-overview/ (visited on 02/04/2023).

[37] *Basic permissioning - GoQuorum - latest*. [Online]. Available: https://consensys.net/docs/goquorum//en/latest/configure-and-manage/configure/permissioning/basic-permissions/ (visited on 02/04/2023).

[38] *Enhanced permissions | ConsenSys GoQuorum*, en, Feb. 2023. [Online]. Available: https://docs.goquorum.consensys.net/configure-and-manage/manage/enhanced-permissions/ (visited on 02/06/2023).

[39] *What is Proof of Authority?* [Online; accessed 9. Jun. 2023], Dec. 2019. [Online]. Available: https://www.coinhouse.com/what-is-proof-of-authority.

[40] D. Ongaro and J. Ousterhout, "In search of an understandable consensus algorithm," en, p. 18, 2014. [Online]. Available: https://raft.github.io/raft.pdf.

[41] J. Clow and Z. Jiang, "A byzantine fault tolerant raft," en, 2017.

[42] Péter Szilágyi, *EIP-225: Clique proof-of-authority consensus protocol*, en, Mar. 2017. [Online]. Available: https://eips.ethereum.org/EIPS/eip-225 (visited on 02/05/2023).

[43] *Clique - Hyperledger Besu*. [Online]. Available: https://besu.hyperledger.org/en/stable/private-networks/how-to/configure/consensus/clique/#extra-data (visited on 02/07/2023).

[44] *Scaling Consensus for Enterprise: Explaining the IBFT Algorithm*, en. [Online]. Available: https://consensys.net/blog/enterprise-blockchain/scaling-consensus-for-enterprise-explaining-the-ibft-algorithm/ (visited on 02/07/2023).

[45] ConsenSys, *Another day, another consensus algorithm. why ibft 2.0?* en, Feb. 2019. [Online]. Available: https://consensys.net/blog/news/another-day-another-consensus-algorithm-why-ibft-2-0/ (visited on 02/09/2023).

[46] H. Moniz, *The Istanbul BFT Consensus Algorithm*, en, arXiv:2002.03613 [cs], May 2020. [Online]. Available: http://arxiv.org/abs/2002.03613 (visited on 02/08/2023).

[47] R. Saltini and D. Hyland-Wood, *Correctness Analysis of IBFT*, en, arXiv:1901.07160 [cs], Aug. 2019. [Online]. Available: http://arxiv.org/abs/1901.07160 (visited on 02/08/2023).

[48] *Qbft consensys goquorum*, en, Feb. 2023. [Online]. Available: https://docs.goquorum.consensys.net/configure-and-manage/configure/consensus-protocols/qbft/ (visited on 02/10/2023).

[49] A. Garreta, *How to reach consensus with strangers*, en, Nov. 2022. [Online]. Available: https://medium.com/nethermind-eth/how-to-reach-consensus-with-strangers-9b57264afd65 (visited on 02/10/2023).

[50] K. Petersen, R. Feldt, S. Mujtaba, and M. Mattsson, "Systematic Mapping Studies in Software Engineering," *Proceedings of the 12th International Conference on Evaluation and Assessment in Software Engineering*, vol. 17, Jun. 2008.

[51] *Google Trends*, [Online; accessed 17. May 2023], May 2023. [Online]. Available: https://trends.google.com/trends/explore?date=2015-04-17%202023-05-17&q=permissioned,private%20blockchain&hl=en-GB.

[52] Aaron Blankstein, *Blockchains and Consensus Protocols: Liveness and Correctness are Inseparable*, [Online; accessed 14. Jun. 2023], 2018. [Online]. Available: https://blog.blockstack.org/blockchains-and-consensus-protocols-liveness-and-correctness-are-inseparable.

[53] T. T. A. Dinh, J. Wang, G. Chen, R. Liu, B. C. Ooi, and K.-L. Tan, *Blockbench: A framework for analyzing private blockchains*, Mar. 2017. arXiv: 1703.04057v1 [cs.DB]. [Online]. Available: http://arxiv.org/abs/1703.04057v1;%20http://arxiv.org/pdf/1703.04057v1.

[54] *Hyperledger caliper architecture*, [Online; accessed 17. Feb. 2023], Aug. 2022. [Online]. Available: https://hyperledger.github.io/caliper/v0.5.0/architecture/#birds-eye-view.

[55] *Installing and Running Caliper*, [Online; accessed 17. Feb. 2023], Aug. 2022. [Online]. Available: https://hyperledger.github.io/caliper/v0.5.0/installing-caliper.

[56] M. Mazzoni, *Caliper*, [Online; accessed 17. Feb. 2023], Feb. 2023. [Online]. Available: https://github.com/MarcoMazzoni/caliper.

[57] Andreas Krueger, *chainhammer*, [Online; accessed 11. Jun. 2023], Jun. 2023. [Online]. Available: https://github.com/drandreaskrueger/chainhammer.

[58] B. Nasrulin, M. De Vos, G. Ishmaev, and J. Pouwelse, "Gromit: Benchmarking the Performance and Scalability of Blockchain Systems," *arXiv*, Aug. 2022. doi: 10.48550/arXiv.2208.11254. eprint: 2208.11254.

[59] D. Saingre, T. Ledoux, and J.-M. Menaud, "BCTMark: a framework for benchmarking blockchain technologies," *IEEE*, pp. 1–8, Nov. 2020. doi: 10.1109/AICCSA50499.2020.9316536.

[60] *Diablo Blockchain Benchmark Suite*, [Online; accessed 11. Jun. 2023], Oct. 2022. [Online]. Available: https://diablobench.github.io.

[61] *Writing Connectors*, [Online; accessed 24. May 2023], Aug. 2022. [Online]. Available: https://hyperledger.github.io/caliper/v0.5.0/writing-connectors.

[62] V. R. Basili and D. M. Weiss, "A methodology for collecting valid software engineering data," *IEEE Transactions on Software Engineering*, vol. SE-10, no. 6, pp. 728–738, 1984. doi: 10.1109/TSE.1984.5010301.

[63] Hyperledger, *Hyperledger Blockchain Performance Metrics White Paper Hyperledger Foundation*, [Online; accessed 27. May 2023], May 2023. [Online]. Available: https://www.hyperledger.org/learn/publications/blockchain-performance-metrics.

[64] T. Mull, "Measuring the Performance Impact of TLS Encryption Using TPC-C," *Yugabyte*, Feb. 2021. [Online]. Available: https://www.yugabyte.com/blog/measuring-the-performance-impact-of-tls-encryption-using-tpcc.

[65] *Ethereum*, [Online; accessed 30. May 2023], Aug. 2022. [Online]. Available: https://hyperledger.github.io/caliper/v0.5.0/ethereum-config.

[66] *caliper-benchmarks*, [Online; accessed 31. May 2023], May 2023. [Online]. Available: https://github.com/hyperledger/caliper-benchmarks.

[67] Prometheus, *Prometheus - Monitoring system & time series database*, [Online; accessed 17. Jun. 2023], Jun. 2023. [Online]. Available: `https://prometheus.io`.

[68] *Grafana: The open observability platform | Grafana Labs*, [Online; accessed 17. Jun. 2023], Jun. 2023. [Online]. Available: `https://grafana.com`.

[69] *Use metrics | ConsenSys GoQuorum*, [Online; accessed 1. Jun. 2023], Jun. 2023. [Online]. Available: `https://docs.goquorum.consensys.net/configure-and-manage/monitor/metrics`.

[70] *Monitors and Observers*, [Online; accessed 1. Jun. 2023], Aug. 2022. [Online]. Available: `https://hyperledger.github.io/caliper/v0.4.2/caliper-monitors/#resource`.

[71] *Tutorial: Migrate from web3js-eea - Documentation*, [Online; accessed 8. Jun. 2023], Apr. 2022. [Online]. Available: `%7Bhttps://consensys.github.io/web3js-quorum/latest/tutorial-Migrate%20from%20web3js-eea.html%7D`.

[72] *quorum-dev-quickstart*, [Online; accessed 18. Jun. 2023], Jun. 2023. [Online]. Available: `https://github.com/ConsenSys/quorum-dev-quickstart`.

[73] *Grafana Loki OSS | Log aggregation system*, [Online; accessed 18. Jun. 2023], Jun. 2023. [Online]. Available: `https://grafana.com/oss/loki`.

[74] *Promtail | Grafana Loki documentation*, [Online; accessed 18. Jun. 2023], Jun. 2023. [Online]. Available: `https://grafana.com/docs/loki/latest/clients/promtail`.

[75] *quorum-genesis-tool*, [Online; accessed 18. Jun. 2023], Jun. 2023. [Online]. Available: `https://github.com/ConsenSys/quorum-genesis-tool`.

[76] João Lopes, *quorum-benchmark*, [Online; accessed 21. Jun. 2023], Jun. 2023. [Online]. Available: `https://github.com/joao-asc-lopes/quorum-benchmark`.

[77] *hyperledger/caliper: A blockchain benchmark framework to measure performance of multiple blockchain solutions https://wiki.hyperledger.org/display/caliper*, [Online; accessed 29. Jun. 2023], Jun. 2023. [Online]. Available: `https://github.com/hyperledger/caliper/blob/main/packages/caliper-ethereum/lib/ethereum-connector.js`.

Appendix A

Project plan

To facilitate the project's management, it is important to have a good overview of the project's phases, their sequence and estimated effort. To achieve this, the Gantt chart in A.1 was created, detailing the expected project's phases and their time estimation. Highlighted tasks represent main tasks, while the non-highlighted ones represent subtasks of the first main task above them.

Table A.1: Project plan

Task	Start	End	Duration
Literature review	11/2022	02/2023	3 months
Informal reading on the topic			1 month
Systematic mapping study			2 months
Evaluation definition	11/2022	01/2023	2 months
Define research hypothesis			2 weeks
Define evaluation methodology			2 months
Value analysis	01/2022	02/2023	2 months
Investigate benchmarking frameworks for Quorum	02/2023	02/2023	2 weeks
Design methodology for algorithm benchmarking	03/2023	04/2023	1 month
Implement system to benchmark algorithms	04/2023	05/2023	1 month
Analyse results and draw conclusions from benchmarks	05/2023	06/2023	1 month
Evaluate results based on the previous definition	06/2023	06/2023	2 weeks
Dissertation writing	11/2022	06/2023	8 months

Appendix B

Hyperledger Caliper Ethereum connector

```
'use strict';

const EthereumHDKey = require('ethereumjs-wallet/hdkey');
const Web3 = require('web3');
const EEAClient = require('web3-eea');
const {ConnectorBase, CaliperUtils, ConfigUtil, TxStatus} = require('@hyperledger/caliper
    -core');

const logger = CaliperUtils.getLogger('ethereum-connector');

/**
 * @typedef {Object} EthereumInvoke
 *
 * @property {string} contract Required. The name of the smart contract
 * @property {string} verb Required. The name of the smart contract function
 * @property {string} args Required. Arguments of the smart contract function in the
    order in which they are defined
 * @property {boolean} readOnly Optional. If method to call is a view.
 */

/**
 * Extends {BlockchainConnector} for a web3 Ethereum backend.
 */
class EthereumConnector extends ConnectorBase {

    /**
     * Create a new instance of the {Ethereum} class.
     * @param {number} workerIndex The zero-based index of the worker who wants to create
     an adapter instance. -1 for the manager process.
     * @param {string} bcType The target SUT type
     */
    constructor(workerIndex, bcType) {
        super(workerIndex, bcType);

        let configPath = CaliperUtils.resolvePath(ConfigUtil.get(ConfigUtil.keys.
    NetworkConfig));
        let ethereumConfig = require(configPath).ethereum;

        // throws on configuration error
        this.checkConfig(ethereumConfig);

        this.ethereumConfig = ethereumConfig;
        this.web3 = new Web3(this.ethereumConfig.url);
        if (this.ethereumConfig.privacy) {
            this.web3eea = new EEAClient(this.web3, ethereumConfig.chainId);
        }
        this.web3.transactionConfirmationBlocks = this.ethereumConfig.transactionConfirmationBlocks;
        this.workerIndex = workerIndex;
        this.context = undefined;
    }

    /**
     * Check the ethereum networkconfig file for errors, throw if invalid
     * @param {object} ethereumConfig The ethereum networkconfig to check.
     */
    checkConfig(ethereumConfig) {
        if (!ethereumConfig.url) {
            throw new Error(
                'No URL given to access the Ethereum SUT. Please check your network configuration. ' +
                'Please see https://hyperledger.github.io/caliper/v0.3/ethereum-config/ for more info.'
            );
        }

        if (ethereumConfig.url.toLowerCase().indexOf('http') === 0) {
            throw new Error(
                'Ethereum benchmarks must not use http(s) RPC connections, as there is no way to guarantee the ' +
                'order of submitted transactions when using other transports. For more information, please see ' +
                'https://github.com/hyperledger/caliper/issues/776#issuecomment-624771622'
            );
        }

        //TODO: add validation logic for the rest of the configuration object
    }

    /**
     * Initialize the {Ethereum} object.
     * @param {boolean} workerInit Indicates whether the initialization happens in the worker process.
     * @return {object} Promise<boolean> True if the account got unlocked successful otherwise false.
     */
    init(workerInit) {
        if (this.ethereumConfig.contractDeployerAddressPrivateKey) {
            this.web3.eth.accounts.wallet.add(this.ethereumConfig.contractDeployerAddressPrivateKey);
        } else if (this.ethereumConfig.contractDeployerAddressPassword) {
            return this.web3.eth.personal.unlockAccount(this.ethereumConfig.contractDeployerAddress, this.ethereumConfig.contractDeployerAddressPassword, 1000);
        }
    }

    /**
     * Deploy smart contracts specified in the network configuration file.
     * @return {object} Promise execution for all the contract creations.
     */
    async installSmartContract() {
        let promises = [];
        let self = this;
        logger.info('Creating contracts...');
        for (const key of Object.keys(this.ethereumConfig.contracts)) {
            const contract = this.ethereumConfig.contracts[key];
            const contractData = require(CaliperUtils.resolvePath(contract.path)); //
TODO remove path property
            const contractGas = contract.gas;
            const estimateGas = contract.estimateGas;
            let privacy;
            if (this.ethereumConfig.privacy) {
                privacy = this.ethereumConfig.privacy[contract.private];
            }

            this.ethereumConfig.contracts[key].abi = contractData.abi;
            promises.push(new Promise(async function(resolve, reject) {
                let contractInstance;
                try {
                    if (privacy) {
                        contractInstance = await self.deployPrivateContract(contractData,
 privacy);
                        logger.info(`Deployed private contract ${contractData.name} at ${
contractInstance.options.address}`);
                    } else {
                        contractInstance = await self.deployContract(contractData);
                        logger.info(`Deployed contract ${contractData.name} at ${
contractInstance.options.address}`);
                    }
                } catch (err) {
                    reject(err);
                }
                self.ethereumConfig.contracts[key].address = contractInstance.options.
address;
                self.ethereumConfig.contracts[key].gas = contractGas;
                self.ethereumConfig.contracts[key].estimateGas = estimateGas;
                resolve(contractInstance);
            }));
        }
        return Promise.all(promises);
    }

    /**
     * Return the Ethereum context associated with the given callback module name.
     * @param {Number} roundIndex The zero-based round index of the test.
     * @param {object} args worker arguments.
     * @return {object} The assembled Ethereum context.
     * @async
     */
    async getContext(roundIndex, args) {
        let context = {
            chainId: 1,
            clientIndex: this.workerIndex,
            gasPrice: 0,
            contracts: {},
            nonces: {},
            web3: this.web3
        };

        context.gasPrice = this.ethereumConfig.gasPrice !== undefined
            ? this.ethereumConfig.gasPrice
            : await this.web3.eth.getGasPrice();

        context.chainId = this.ethereumConfig.chainId !== undefined
            ? this.ethereumConfig.chainId
            : await this.web3.eth.getChainId();

        for (const key of Object.keys(args.contracts)) {
            context.contracts[key] = {
                contract: new this.web3.eth.Contract(args.contracts[key].abi, args.contracts[key].address),
                gas: args.contracts[key].gas,
                estimateGas: args.contracts[key].estimateGas
            };
        }

        if (this.ethereumConfig.fromAddress) {
            context.fromAddress = this.ethereumConfig.fromAddress;
        }

        if (this.ethereumConfig.contractDeployerAddress) {
            context.contractDeployerAddress = this.ethereumConfig.contractDeployerAddress;
            context.contractDeployerAddressPrivateKey = this.ethereumConfig.contractDeployerAddressPrivateKey;
        }

        if (this.ethereumConfig.fromAddressSeed) {
            let hdwallet = EthereumHDKey.fromMasterSeed(this.ethereumConfig.fromAddressSeed);
            let wallet = hdwallet.derivePath('m/44\'/60\'/' + this.workerIndex + '\'/0/0').getWallet();
            context.fromAddress = wallet.getChecksumAddressString();
            context.nonces[context.fromAddress] = await this.web3.eth.getTransactionCount(context.fromAddress);
            this.web3.eth.accounts.wallet.add(wallet.getPrivateKeyString());
        } else if (this.ethereumConfig.fromAddressPrivateKey) {
            context.nonces[this.ethereumConfig.fromAddress] = await this.web3.eth.getTransactionCount(this.ethereumConfig.fromAddress);
            this.web3.eth.accounts.wallet.add(this.ethereumConfig.fromAddressPrivateKey);
        } else if (this.ethereumConfig.fromAddressPassword) {
            await context.web3.eth.personal.unlockAccount(this.ethereumConfig.fromAddress, this.ethereumConfig.fromAddressPassword, 1000);
        }

        if (this.ethereumConfig.privacy) {
            context.web3eea = this.web3eea;
            context.privacy = this.ethereumConfig.privacy;
        }

        this.context = context;
        return context;
    }

    /**
     * Release the given Ethereum context.
     * @async
     */
    async releaseContext() {
        // nothing to do
    }

    /**
     * Submit a transaction to the ethereum context.
     * @param {EthereumInvoke} request Methods call data.
     * @return {Promise<TxStatus>} Result and stats of the transaction invocation.
     */
    async _sendSingleRequest(request) {
        if (request.privacy) {
            return this._sendSinglePrivateRequest(request);
        }

        const context = this.context;
        let status = new TxStatus();
        let params = {from: context.fromAddress};
        if (request.hasOwnProperty('value') && request.value > 0) {
            params.value = request.value;
        }
        let contractInfo = context.contracts[request.contract];

        let receipt = null;
        let methodType = 'send';
        if (request.readOnly) {
            methodType = 'call';
        } else if (context.nonces && (typeof context.nonces[context.fromAddress] !== '
undefined')) {
            let nonce = context.nonces[context.fromAddress];
            context.nonces[context.fromAddress] = nonce + 1;
            params.nonce = nonce;

            // leaving these values unset causes web3 to fetch gasPrice and
            // chainId on the fly. This can cause transactions to be
            // reordered, which in turn causes nonce failures
            params.gasPrice = context.gasPrice;
            params.chainId = context.chainId;
        }

        const onFailure = (err) => {
            status.SetStatusFail();
            logger.error(`Failed tx on ${request.contract}; calling method: ${request.
verb}; nonce: ${params.nonce}`);
            logger.error(err);
        };

        const onSuccess = (rec) => {
            status.SetID(rec.transactionHash);
            status.SetResult(rec);
            status.SetVerification(true);
            status.SetStatusSuccess();
        };

        if (request.args) {
            if (contractInfo.gas && contractInfo.gas[request.verb]) {
                params.gas = contractInfo.gas[request.verb];
            } else if (contractInfo.estimateGas) {
                params.gas = 1000 + await contractInfo.contract.methods[request.verb](...
request.args).estimateGas();
```

```
            }

            try {
                receipt = await contractInfo.contract.methods[request.verb](...request.
    args)[methodType](params);
                onSuccess(receipt);
            } catch (err) {
                onFailure(err);
            }
        } else {
            if (contractInfo.gas && contractInfo.gas[request.verb]) {
                params.gas = contractInfo.gas[request.verb];
            } else if (contractInfo.estimateGas) {
                params.gas = 1000 + await contractInfo.contract.methods[request.verb].
    estimateGas(params);
            }

            try {
                receipt = await contractInfo.contract.methods[request.verb]()[methodType
    ](params);
                onSuccess(receipt);
            } catch (err) {
                onFailure(err);
            }
        }

        return status;
    }

    /**
     * Submit a private transaction to the ethereum context.
     * @param {EthereumInvoke} request Methods call data.
     * @return {Promise<TxStatus>} Result and stats of the transaction invocation.
     */
    async _sendSinglePrivateRequest(request) {
        const context = this.context;
        const web3eea = context.web3eea;
        const contractInfo = context.contracts[request.contract];
        const privacy = request.privacy;
        const sender = privacy.sender;

        const status = new TxStatus();

        const onFailure = (err) => {
            status.SetStatusFail();
            logger.error(`Failed private tx on ${request.contract}; calling method: ${
    request.verb}; private nonce: ` + 0);
            logger.error(err);
        };

        const onSuccess = (rec) => {
            status.SetID(rec.transactionHash);
            status.SetResult(rec);
            status.SetVerification(true);
            status.SetStatusSuccess();
        };

        let payload;
        if (request.args) {
            payload = contractInfo.contract.methods[request.verb](...request.args).
    encodeABI();
        } else {
            payload = contractInfo.contract.methods[request.verb]().encodeABI();
        }

        const transaction = {
            to: contractInfo.contract._address,
            data: payload
        };

        try {
            if (request.readOnly) {
                transaction.privacyGroupId = await this.resolvePrivacyGroup(privacy);

                const value = await web3eea.priv.call(transaction);
                onSuccess(value);
            } else {
                transaction.nonce = sender.nonce;
                transaction.privateKey = sender.privateKey.substring(2);
                this.setPrivateTransactionParticipants(transaction, privacy);

                const txHash = await web3eea.eea.sendRawTransaction(transaction);
                const rcpt = await web3eea.priv.getTransactionReceipt(txHash, transaction
    .privateFrom);
                if (rcpt.status === '0x1')  {
                    onSuccess(rcpt);
                } else {
                    onFailure(rcpt);
                }
            }
        } catch(err) {
            onFailure(err);
        }

        return status;
    }


    /**
     * Deploys a new contract using the given web3 instance
     * @param {JSON} contractData Contract data with abi, bytecode and gas properties
     * @returns {Promise<web3.eth.Contract>} The deployed contract instance
     */
    async deployContract(contractData) {
        const web3 = this.web3;
        const contractDeployerAddress = this.ethereumConfig.contractDeployerAddress;
        const contract = new web3.eth.Contract(contractData.abi);
        const contractDeploy = contract.deploy({
            data: contractData.bytecode
        });

        try {
            return contractDeploy.send({
                from: contractDeployerAddress,
                gas: contractData.gas
            });
        } catch (err) {
            throw(err);
        }
    }

    /**
     * Deploys a new contract using the given web3 instance
     * @param {JSON} contractData Contract data with abi, bytecode and gas properties
     * @param {JSON} privacy Privacy options
     * @returns {Promise<web3.eth.Contract>} The deployed contract instance
     */
    async deployPrivateContract(contractData, privacy) {
        const web3 = this.web3;
        const web3eea = this.web3eea;
        // Using randomly generated account to deploy private contract to avoid public/
    private nonce issues
        const deployerAccount =  web3.eth.accounts.create();

        const transaction = {
            data: contractData.bytecode,
            nonce: deployerAccount.nonce,
            privateKey: deployerAccount.privateKey.substring(2),    // web3js-eea doesn't
     not accept private keys prefixed by '0x'
        };

        this.setPrivateTransactionParticipants(transaction, privacy);

        try {
            const txHash = await web3eea.eea.sendRawTransaction(transaction);
            const txRcpt = await web3eea.priv.getTransactionReceipt(txHash, transaction.
    privateFrom);

            if (txRcpt.status === '0x1') {
                return new web3.eth.Contract(contractData.abi, txRcpt.contractAddress);
            } else {
                const msg = `Failed private transaction hash ${txHash}`;
                logger.error(msg);
                throw new Error(msg);
            }
        } catch (err) {
            logger.error('Error deploying private contract: ', JSON.stringify(err));
            throw(err);
        }
    }

    /**
     * It passes deployed contracts addresses to all workers (only known after deploy
    contract)
     * @param {Number} number of workers to prepare
     * @returns {Array} worker args
     * @async
     */
    async prepareWorkerArguments(number) {
        let result = [];
        for (let i = 0 ; i<= number ; i++) {
            result[i] = {contracts: this.ethereumConfig.contracts};
        }
        return result;
    }

    /**
```

```
     * Returns the privacy group id depending on the privacy mode being used
     * @param {JSON} privacy Privacy options
     * @returns {Promise<string>} The privacyGroupId
     */
    async resolvePrivacyGroup(privacy) {
        const web3eea = this.context.web3eea;

        switch(privacy.groupType) {
            case 'legacy': {
                const privGroups = await web3eea.priv.findPrivacyGroup({addresses: [
    privacy.privateFrom, ...privacy.privateFor]});
                if (privGroups.length > 0) {
                    return privGroups.filter(function(el) {
                        return el.type === 'LEGACY';
                    })[0].privacyGroupId;
                } else {
                    throw new Error('There are multiple legacy privacy groups with same
    members. Can\'t resolve privacyGroupId.');
                }
            }
            case 'pantheon':
            case 'onchain': {
                return privacy.privacyGroupId;
            } default: {
                throw new Error('Invalid privacy type');
            }
        }
    }

    /**
     * Set the participants of a privacy transaction depending on the privacy mode being
    used
     * @param {JSON} transaction Object representing the transaction fields
     * @param {JSON} privacy Privacy options
     */
    setPrivateTransactionParticipants(transaction, privacy) {
        switch(privacy.groupType) {
            case 'legacy': {
                transaction.privateFrom = privacy.privateFrom;
                transaction.privateFor = privacy.privateFor;
                break;
            }
            case 'pantheon':
            case 'onchain': {
                transaction.privateFrom = privacy.privateFrom;
                transaction.privacyGroupId = privacy.privacyGroupId;
                break;
            } default: {
                throw new Error('Invalid privacy type');
            }
        }
    }
}

module.exports = EthereumConnector;
```

Listing B.1: Caliper Ethereum connector [77]

Appendix C

Prometheus configuration

```
global:
  scrape_interval:     15s
  evaluation_interval: 15s

alerting:
rule_files:
scrape_configs:
  - job_name: validator1
    scrape_interval: 15s
    scrape_timeout: 10s
    metrics_path: /debug/metrics/prometheus
    scheme: http
    static_configs:
      - targets: [ validator1:9545 ]

  - job_name: validator2
    scrape_interval: 15s
    scrape_timeout: 10s
    metrics_path: /debug/metrics/prometheus
    scheme: http
    static_configs:
      - targets: [ validator2:9545 ]

  - job_name: validator3
    scrape_interval: 15s
    scrape_timeout: 10s
    metrics_path: /debug/metrics/prometheus
    scheme: http
    static_configs:
      - targets: [ validator3:9545 ]

  - job_name: validator4
    scrape_interval: 15s
    scrape_timeout: 10s
    metrics_path: /debug/metrics/prometheus
    scheme: http
    static_configs:
      - targets: [ validator4:9545 ]

  - job_name: rpcnode
    scrape_interval: 15s
    scrape_timeout: 10s
    metrics_path: /debug/metrics/prometheus
    scheme: http
    static_configs:
      - targets: [ rpcnode:9545 ]

  - job_name: newnode
    scrape_interval: 15s
    scrape_timeout: 10s
    metrics_path: /debug/metrics/prometheus
    scheme: http
    static_configs:
      - targets: [ newnode:9545 ]

  - job_name: member1quorum
    scrape_interval: 15s
    scrape_timeout: 10s
    metrics_path: /debug/metrics/prometheus
    scheme: http
    static_configs:
      - targets: [ member1quorum:9545 ]

  - job_name: member2quorum
    scrape_interval: 15s
    scrape_timeout: 10s
    metrics_path: /debug/metrics/prometheus
    scheme: http
    static_configs:
      - targets: [ member2quorum:9545 ]

  - job_name: member3quorum
    scrape_interval: 15s
    scrape_timeout: 10s
    metrics_path: /debug/metrics/prometheus
    scheme: http
    static_configs:
      - targets: [ member3quorum:9545 ]

  - job_name: member1tessera
    scrape_interval: 15s
    scrape_timeout: 10s
    metrics_path: /metrics
    scheme: http
    static_configs:
      - targets: [ member1tessera:9000 ]

  - job_name: member2tessera
    scrape_interval: 15s
    scrape_timeout: 10s
    metrics_path: /metrics
    scheme: http
    static_configs:
      - targets: [ member2tessera:9000 ]

  - job_name: member3tessera
    scrape_interval: 15s
    scrape_timeout: 10s
    metrics_path: /metrics
    scheme: http
    static_configs:
      - targets: [ member3tessera:9000 ]

```

Listing C.1: Prometheus configuration

Appendix D

Base docker compose file

```
---
version: '3.6'

services:
  prometheus:
    image: "prom/prometheus"
    volumes:
      - ./common/config/prometheus/prometheus.yml:/etc/prometheus/prometheus.yml
      - prometheus:/prometheus
    command:
      - --config.file=/etc/prometheus/prometheus.yml
    ports:
      - 9090:9090/tcp
    networks:
      quorum-benchmark:
        ipv4_address: 172.16.239.32

  grafana:
    image: "grafana/grafana"
    environment:
      - GF_AUTH_ANONYMOUS_ENABLED=true
      - GF_USERS_VIEWERS_CAN_EDIT=true
    volumes:
      - ./common/config/grafana/provisioning/:/etc/grafana/provisioning/
      - grafana:/var/lib/grafana
    ports:
      - 3000:3000/tcp
    networks:
      quorum-benchmark:
        ipv4_address: 172.16.239.33

  chaos-delay:
    image: gaiaadm/pumba
    volumes:
      - /var/run/docker.sock:/var/run/docker.sock
    command: "--log-level debug --interval 20s --random netem --tc-image gaiadocker/
    iproute2 --duration 10s delay re2:^network-validator"

networks:
  quorum-benchmark:
    name: quorum-benchmark
    driver: bridge
    ipam:
      driver: default
      config:
```

```
      - subnet: 172.16.239.0/24

volumes:
  prometheus:
  grafana:
```

Listing D.1: Base docker compose file

Appendix E

Seven nodes docker compose file

```
---
version: '3.6'

x-quorum-def:
  &quorum-def
  restart: "on-failure"
  build:
    context: .
    dockerfile: common/config/goquorum/Dockerfile
    args:
      QUORUM_VERSION: ${QUORUM_VERSION:-latest}
      GOQUORUM_CONS_ALGO: ${GOQUORUM_CONS_ALGO}
      NODES_NUMBER: ${NODES_NUMBER}
  expose:
    - 30303
    - 8545
    - 9545
  healthcheck:
    test: ["CMD", "wget", "--spider", "--proxy", "off", "http://localhost:8545"]
    interval: 3s
    timeout: 3s
    retries: 10
    start_period: 5s

x-tessera-def:
  &tessera-def
  build:
    context: .
    dockerfile: common/config/tessera/Dockerfile
    args:
      TESSERA_VERSION: ${TESSERA_VERSION:-latest}
      NODES_NUMBER: ${NODES_NUMBER}
  environment:
    TESSERA_MODE: tessera
  expose:
    - 9000
    - 9080
    - 9101
  restart: "no"
  healthcheck:
    test: ["CMD", "wget", "--spider", "--proxy", "off", "http://localhost:9000/upcheck"]
    interval: 3s
    timeout: 3s
    retries: 20
    start_period: 5s
```

```

services:
  explorer:
    image: consensys/quorum-explorer:${QUORUM_EXPLORER_VERSION:-latest}
    volumes:
      - ./${NODES_NUMBER}-nodes/quorum-explorer/config.json:/app/config.json
      - ./${NODES_NUMBER}-nodes/quorum-explorer/env:/app/.env.production
    depends_on:
      - rpcnode
    ports:
      - 25000:25000/tcp
    networks:
      quorum-benchmark:
        ipv4_address: 172.16.239.31

  rpcnode:
    <<: *quorum-def
    container_name: rpcnode
    ports:
      - 8545:8545/tcp
      - 8546:8546/tcp
      - 30303
      - 9545
    environment:
      - GOQUORUM_CONS_ALGO=${GOQUORUM_CONS_ALGO}
    volumes:
      - ./${NODES_NUMBER}-nodes/${GOQUORUM_CONS_ALGO}/rpcnode:/config/keys
      - ./logs/quorum:/var/log/quorum/${GOQUORUM_CONS_ALGO}/
    networks:
      quorum-benchmark:
        ipv4_address: 172.16.239.38

  validator0:
    << : *quorum-def
    ports:
      - 21001:8545/tcp
      - 30303
      - 9545
    environment:
      - GOQUORUM_CONS_ALGO=${GOQUORUM_CONS_ALGO}
    volumes:
      - ./${NODES_NUMBER}-nodes/${GOQUORUM_CONS_ALGO}/validator0:/config/keys
      - ./logs/quorum:/var/log/quorum/${GOQUORUM_CONS_ALGO}/
    networks:
      quorum-benchmark:
        ipv4_address: 172.16.239.11

  validator1:
    << : *quorum-def
    ports:
      - 21002:8545/tcp
      - 30303
      - 9545
    environment:
      - GOQUORUM_CONS_ALGO=${GOQUORUM_CONS_ALGO}
    volumes:
      - ./${NODES_NUMBER}-nodes/${GOQUORUM_CONS_ALGO}/validator1:/config/keys
      - ./logs/quorum:/var/log/quorum/${GOQUORUM_CONS_ALGO}/
    networks:
      quorum-benchmark:
        ipv4_address: 172.16.239.12

  validator2:
    << : *quorum-def
    ports:
      - 21003:8545/tcp
      - 30303
      - 9545
    environment:
      - GOQUORUM_CONS_ALGO=${GOQUORUM_CONS_ALGO}
    volumes:
      - ./${NODES_NUMBER}-nodes/${GOQUORUM_CONS_ALGO}/validator2:/config/keys
      - ./logs/quorum:/var/log/quorum/${GOQUORUM_CONS_ALGO}/
    networks:
      quorum-benchmark:
        ipv4_address: 172.16.239.13

  validator3:
    << : *quorum-def
    ports:
      - 21004:8545/tcp
      - 30303
      - 9545
    environment:
      - GOQUORUM_CONS_ALGO=${GOQUORUM_CONS_ALGO}
    volumes:
      - ./${NODES_NUMBER}-nodes/${GOQUORUM_CONS_ALGO}/validator3:/config/keys
      - ./logs/quorum:/var/log/quorum/${GOQUORUM_CONS_ALGO}/
    networks:
      quorum-benchmark:
        ipv4_address: 172.16.239.14

  member0tessera:
    << : *tessera-def
    ports:
      - 9081:9080
    volumes:
      - ./${NODES_NUMBER}-nodes/${GOQUORUM_CONS_ALGO}/member0:/config/keys
      - ./logs/tessera:/var/log/tessera/
    networks:
      quorum-benchmark:
        ipv4_address: 172.16.239.26

  member0quorum:
    << : *quorum-def
    ports:
      - 20000:8545/tcp
      - 20001:8546/tcp
      - 30303
      - 9545
    depends_on:
      - member0tessera
    environment:
      - GOQUORUM_CONS_ALGO=${GOQUORUM_CONS_ALGO}
      - QUORUM_PTM=member0tessera
    volumes:
      - ./${NODES_NUMBER}-nodes/${GOQUORUM_CONS_ALGO}/member0:/config/keys
      - ./logs/quorum:/var/log/quorum/${GOQUORUM_CONS_ALGO}/
    networks:
      quorum-benchmark:
        ipv4_address: 172.16.239.15

  member1tessera:
    << : *tessera-def
    ports:
      - 9082:9080
    volumes:
      - ./${NODES_NUMBER}-nodes/${GOQUORUM_CONS_ALGO}/member1:/config/keys
      - ./logs/tessera:/var/log/tessera/
    networks:
      quorum-benchmark:
        ipv4_address: 172.16.239.27

  member1quorum:
    << : *quorum-def
    ports:
      - 20002:8545/tcp
      - 20003:8546/tcp
      - 30303
      - 9545
    depends_on:
      - member1tessera
    environment:
      - GOQUORUM_CONS_ALGO=${GOQUORUM_CONS_ALGO}
      - QUORUM_PTM=member1tessera
    volumes:
      - ./${NODES_NUMBER}-nodes/${GOQUORUM_CONS_ALGO}/member1:/config/keys
      - ./logs/quorum:/var/log/quorum/
    networks:
      quorum-benchmark:
        ipv4_address: 172.16.239.16

  member2tessera:
    << : *tessera-def
    ports:
      - 9083:9080
    volumes:
      - ./${NODES_NUMBER}-nodes/${GOQUORUM_CONS_ALGO}/member2:/config/keys
      - ./logs/tessera:/var/log/tessera/
    networks:
      quorum-benchmark:
        ipv4_address: 172.16.239.28

  member2quorum:
    << : *quorum-def
    ports:
      - 20004:8545/tcp
      - 20005:8546/tcp
      - 30303
      - 9545
    depends_on:
      - member2tessera
    environment:
      - GOQUORUM_CONS_ALGO=${GOQUORUM_CONS_ALGO}
      - QUORUM_PTM=member2tessera
    volumes:
      - ./${NODES_NUMBER}-nodes/${GOQUORUM_CONS_ALGO}/member2:/config/keys
      - ./logs/quorum:/var/log/quorum/
    networks:
      quorum-benchmark:
        ipv4_address: 172.16.239.17
```

Listing E.1: Seven nodes docker compose file