School of Computer Science and Engineering

Faculty of Engineering

The University of New South Wales

Extending JMeter for Blockchain Performance Testing

by

Zhenqi Wang

Thesis submitted as a requirement for the degree of

Bachelor of Engineering in Software Engineering

Submitted: April 2021 Student ID: z5141545

Supervisor: Dr H.M.N. Dilum Bandara Assessor: Dr Helen Paik

Abstract

While introduced as the underlying technology for the cryptocurrency Bitcoin, blockchain is expected to transform enterprise applications across many domains. To ensure the quality of service, performance of enterprise applications are rigorously tested before being released. However, it is nontrivial to test the performance of blockchain-based applications due to the decoupling of request and response messages, high latencies, and the digital signing process. While tools like Apache JMeter has been considered an industry-standard tool for load testing, they do not support blockchain DApps performance testing. Hence, performance testing of blockchain-based Decentralised Applications (DApps) mostly involves custom scripts. Thus, it is imperative to add blockchain performance testing capabilities to JMeter, as it enables the developers to use familiar toolset to test the performance of blockchains and DApps. This project developed a plugin for JMeter to enable performance testing of Hyperledger Fabric blockchain and blockchain-based applications with multiple API calls and longer timeouts. The plugin also supports asynchronous API calls and blockchain wallet feature for transaction signing and inbuilt ERC-20 token smart contract support. A smart contract interface was implemented so that other smart contracts deployed on Hyperledger Fabric can be connected to the plugin, and performance evaluation can then be performed on blockchains or DApps based on those smart contracts. The utility of the plugin was demonstrated by performance testing blockchains and a DApps deployed on Hyperledger Fabric.

Acknowledgements

This work has been inspired by the labours of Dr Dilum Bandara at CSIRO's data61 and various academics in the field of blockchain researches.

Abbreviations

API	Application Programming Interface
BC	Blockchain
BFT	Byzantine fault Tolerance
DApps	Blockchain-based Decentralized Applications
ESCC	The Endorsement System Chaincode
MVCC	The Multi-Version Concurrency Control
PoW	Proof-of-Work
QoS	Quality of Service
VSCC	Validation System Chaincode

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Blockchains are essentially distributed databases for peer-to-peer transactions, shared across a public or private network. It is pointed out in a recent research that blockchains may have predominate short-term value in cost-reducing and acting as permissioned commercial applications (Carson et al., 2018). As a result, there has been significant growth in blockchain-based Decentralised Applications (DApps) over the recent years. Apart from the original Bitcoin blockchains, several private and consortium blockchain platforms have been developed and tuned for enterprise uses (Dinh, et al., 2017) to meet the industrial needs for DApps and to fully explore the business potential of the blockchain networks.

Performance evaluation is essential for most applications, as applications cannot be deployed without ensuring its Quality of Service (QoS). It would be essential for clients to understand the limitations of certain platforms and select the appropriate platforms for designated applications (Pongnumkul et al., 2017). DApps also require performance evaluation to ensure their ability to operate reliably and meet business demands. Therefore, there is great need of independent and hand-on performance evaluation of blockchains and related applications. Despite a number of researches on the performance testing of blockchains, most blockchain performance evaluation are still conducted semi-manually, by using ad-hoc tools and custom scripts to adjust input parameters and workloads of blockchain platforms and observe and aggregate the results (Thakkar, et al., 2018). In consequence, the process heavily involves manpower and efficiency is low. Current performance testing tools also lacks support for blockchains, large timeouts and multiple async calls. Therefore, an application that can apply automated testing and generation of large workloads to blockchain performance testing may reduce the workload significantly.

In this thesis, I developed a Apache JMeter plugin to add Hyperledger Fabric blockchain and DApp performance testing capabilities and allow for multiple async responses and large timeouts. The plugin is implemented with Java and Hyperledger Fabric gateway, with custom input parameters and interfaces for smart contract extensions. Implementation details and design are discussed in detail in this thesis, along with use cases of the plugin. Various evaluations have also been completed on the plugin, including functional correctness, performance overhead measurements of the plugin and the effects of different workload on performance. Overall, the plugin demonstrated robustness, better flow control and lower error rates compared to pre-existing performance evaluation samplers.

Chapter 2 and 3 of this thesis provide background information of the project, and current research outcomes related the fields of the project. The implementation and design of the project are discussed in Chapter 4, while the evaluation is in Chapter 5. Finally, conclusion and future work plans are provided in Chapter 6.

Chapter 2

Background

With the increasing usage of blockchain in enterprise and mission-critical applications, there is a need for performance assessment of private blockchains. Currently, performance testing relies on semi-manual testing scripts, which requires a lot of time and effort. Meanwhile, there are a few mature automatic web application performance testing tools, such as JMeter, which can generate large testing loads and save such efforts. This chapter provides an overview of blockchains and performance testing to provide detailed insight into these aspects.

2.1 Blockchains

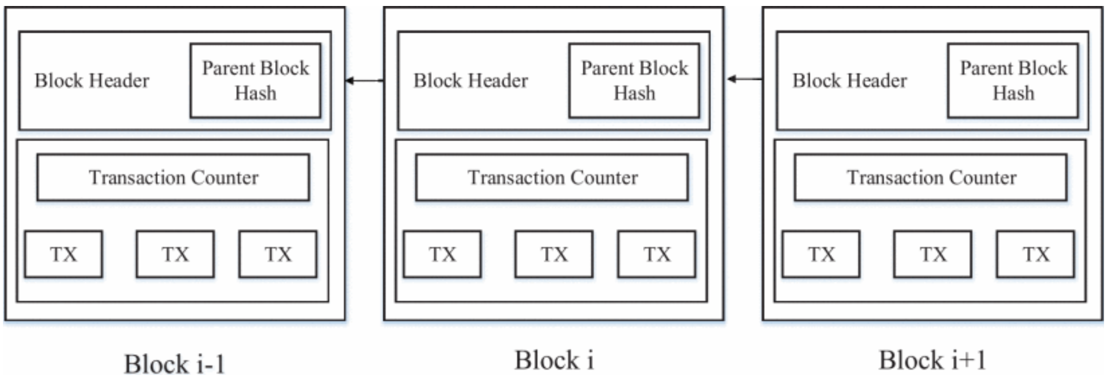


Figure 2.1 An example of blockchain which consists of a continuous sequence of blocks (Zheng, et al., 2017).

Blockchain is a sequence of data blocks holding transaction records, which is essentially a distributed database for peer-to-peer transactions, shared across a public or private network. As illustrated in Figure 2.1, each block in the blockchain has one parent block, and the hash of the parent block is stored in the header of the child block to ensure security. In the body of each block, there is a transaction counter and the transaction records. A blockchain network is usually made up of several nodes, each holding a copy of the entire blockchain. Consequently, it is impossible to have single node of failure in the network, which makes it secure, transparent, and immutable (Carson, et al., 2018).

2.1.1 Properties of Blockchains

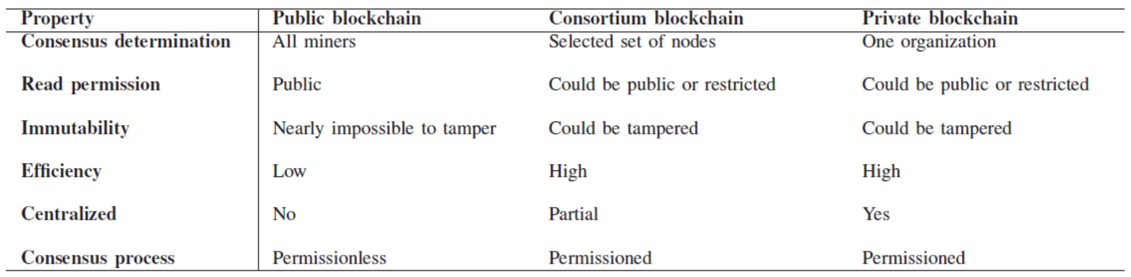
Following are some of the key properties of blockchains (Zheng, et al., 2017):

- *Immutability* – Data cannot be modified once it is published. Blockchains uses distinctive hashes to verify the blocks, which are calculated based on hash of the previous block, and the data on the current block. As a result, modified data will result in a different hash value, and will disrupt the chain.
- *Transparency* – In a public blockchain, each node can read and write to the blockchain, and all transactions can be verified publicly.
- *Distributed* – Each node in the network keeps a copy of the entire blockchain, which prevents the existence of a single point of failure. Multiple local copies of data can increase transaction speed, and it also maintains the integrity and prevents malicious tampering of data.
- *Consistency* – The copy of the blockchain at each node is always consistent. If a new block is added to the blockchain, all nodes in the network would be updated of this block.
- *Availability* – Each node in the network keeps a copy of the entire blockchain, which means in the case of one node failure, the other nodes in the network would still be able to function, making the network always available to users.

2.1.2 Different Types of Blockchains

As shown in Table 2.2 there are three major types of blockchains, namely public, consortium, and private blockchains (Zheng, et al., 2017).

Table 2.2 Comparison of different type of blockchains. (Zheng, et al., 2017)



Property	Public blockchain	Consortium blockchain	Private blockchain
Consensus determination	All miners	Selected set of nodes	One organization
Read permission	Public	Could be public or restricted	Could be public or restricted
Immutability	Nearly impossible to tamper	Could be tampered	Could be tampered
Efficiency	Low	High	High
Centralized	No	Partial	Yes
Consensus process	Permissionless	Permissioned	Permissioned

Public blockchain is the original type of blockchain. As indicated in its name, public blockchain can be accessed by the public, and its consensus determination is also determined by all users (miners). It usually uses complicated Proof-of-Work (PoW) policy as consensus strategy. In PoW, each node works to calculate a hash value for the header that is smaller or equal to a target value. The first node to achieve a hash value meeting the requirements would broadcast its results to the network for validation, followed by appending the new block to the chain. This process is called "mining" in bitcoins. However, this process can be very inefficient and waste too much resource (Zheng, et al., 2017) as the given value is usually hard to achieve. Due to the larger number of users involved invalidation and propagation, transaction throughput is also limited, and the latency is high (Zheng, et al., 2017). However, as the blockchain is fully public and stored on many nodes, it is immutable and almost impossible to be tampered with.

Consortium blockchain is different from public blockchain in that only a selected group of nodes would take part in the validation process, and it can be constructed by several organisations. As a result, it is considered partially centralised, as it is controlled by a small group of nodes. The transaction on the blockchain may also not be fully public, which makes the blockchain vulnerable to malicious tampering. However, since the validation process is limited to a much smaller number of nodes, it is more efficient than public blockchain. Consortium blockchains also usually uses Byzantine fault tolerance protocols such as Practical Byzantine Fault Tolerance that uses replication to avoid faults and can secure the network.

Private blockchain is very similar to consortium blockchain; however, it is different from consortium blockchain in that its consensus and validation is fully controlled by a certain organisation, which makes it centralised. Its transactions may also not be accessible by the public and can easily be tampered due to the small number of participants (Zheng, et al., 2017). The efficiency of private blockchain is also considered to be very high due to the small number of participants involved in validation. Unlike public blockchains which use PoW as a consensus strategy, private blockchains prefer protocols based on Byzantine Fault Tolerance (BFT).

2.1.3 Private Blockchains

As discussed in Section 2.1.2, while blockchains are originally public and transparent, it has been derived into multiple variations over the years. As indicated by Carson et al., 2018, blockchains may not have to be used to generate value, they can also be used as permissioned commercial applications to handle smart contracts, which can be used to automate insurance-claim payouts. One of the users of private blockchains is the Australian Stock Exchange (ASX), which announced its plan to use distributed ledger technology to replace the CHESS system used currently (ASX, 2017).

Currently, some of the well-established private blockchain platforms are Hyperledger Fabric, Ethereum and R3Corda.

Hyperledger Fabric is a more industrial-intended platform which uses docker containers to enable smart contracts named "Chaincode". The platform is designed to be modular and allows different industrial features. Overall, it caters to large amount of test data better and performs better than Ethereum, with faster execution, larger throughput and less latency (Pongnumkul et al., 2017).

Ethereum is an open-source and public blockchain platform based on bitcoin blockchains. It extends bitcoin technologies to allow custom business logic. Even though it is meant to be a public blockchain with its own cryptocurrency Ether, it can be modified into a private blockchain network. However, the performance of Ethereum is not as good as Hyperledger (Pongnumkul et al., 2017).

R3 Corda is an open-source blockchain platform intended for enterprise use. Corda emphasises on user data privacy, and only shares transactions by a need-to-know rule, which suits the need of enterprise applications. As a result, no nodes in a Corda network can have the complete set of data (Hamilton, 2019). Although this would accelerate the execution by providing linear horizontal scalability, there may also be security problems as the network is no longer transparent to all users.

2.2 Web Application Performance Testing

Performance evaluation is the process of measuring the performance of a system under test (Performance, et al., 2018). The goal of performance evaluation is to observe and understand the performance of a given system. It usually consists of system-wide measurements of throughput, latency, or the time to write a block to a physical storage.

Benchmarking is the process of recording and comparing standard measurements to previous measurements or new measurements of other systems (Performance, 2018). It is essentially a controlled performance evaluation, with other platforms or previous measurements as controlled groups.

2.2.1 Performance Evaluation Metrics

There are usually two major type of metrics in Performance Benchmarking. *Throughput* measures the rate of transactions being appended to blockchains (Thakkar, et al., 2018). It is sometimes further divided into read throughput and transaction throughput (Performance, et al., 2018). *Latency* is the time between sending of the transaction proposal and the transaction being committed (Thakkar, et al., 2018), and it consists of read and transaction latency (Performance, et al., 2018). When testing a specific platform, there may be more specific types of latency. For example, when testing Hyperledger Fabric, there are four major types of latency, namely Endorsement Latency, Broadcast Latency, Commit Latency and Ordering Latency (Thakkar, et al., 2018).

2.2.2 Performance Testing tools

Blockbench is a blockchain-specified tool for performance evaluation which supports multiple blockchain platforms, such as Ethereum, Hyperledger and Parity. Blockbench provides several workloads, which are transaction-oriented currently and is designed to benchmark and micro-benchmark blockchains and related applications. It is flexible and extensible (Dinh, et al., 2017).

Apache JMeter is a Java-based tool that can be used to conduct automate performance testing on web applications or services using HTTP or FTP servers. It is considered highly extensible through a provided API (Nemerov, 2006). It is flexible and can be used to generate a heavy workload on a server, group of servers to analyse the performance of an application over different workloads. It was originally intended for web applications but has now been extended to multiple platforms (Apache JMeter documentation, 2020).

HP LoadRunner is a software testing tool used for load testing of applications and system behaviour measurement. It is considered very powerful and can simulate large concurrent load of thousands of users (Khan, et al., 2016). It simulates user activity by generating messages between different components of an application, rather than simulating human-computer interactions such as mouse activities.

2.2.3 Blockchain Performance Testing

Figure 2.3 shows a typical configuration of blockchain performance evaluation. The Test Harness mentioned above is the hardware and software used to execute the tests, and the client is the entity that injects workload into the system to be tested. The System Under Test (SUT) is considered here as the “environment” to run the blockchain system, which includes the hardware, software, network and other elements required (Performance, et al., 2018).

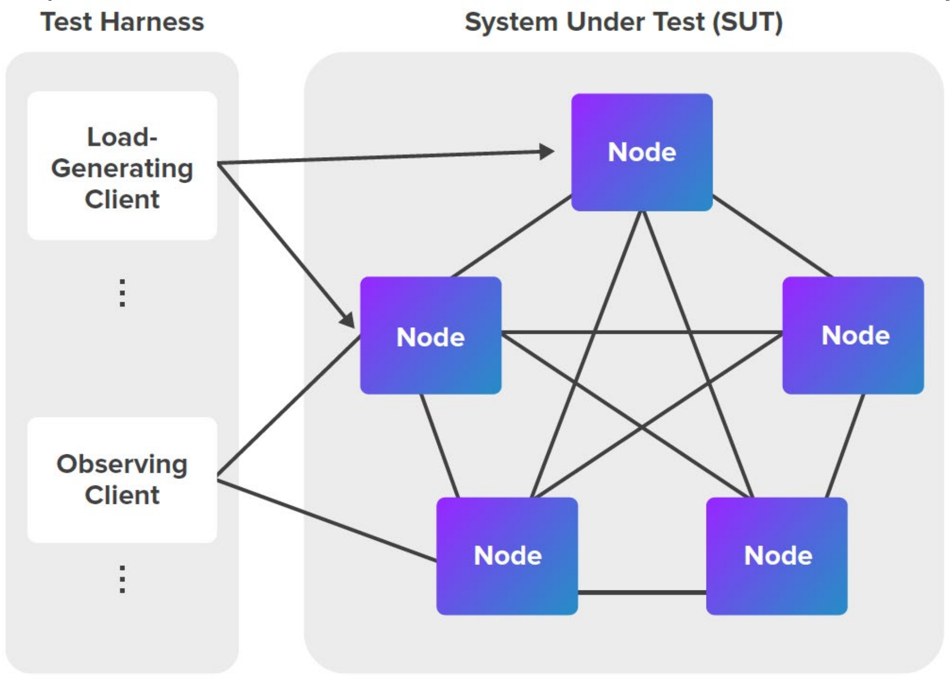


Figure 2.3 A typical configuration for a blockchain performance evaluation (Performance, et al., 2018)

Chapter 3

Previous Work

Many researches have been carried out worldwide on blockchains and blockchain performance assessment, this chapter will focus on some of the researches relevant to the project to be proposed.

3.1 Hyperledger Fabric

Hyperledger Fabric is a distributed operating system for permissioned blockchains network. It can be used to execute decentralised applications with most modern languages. It provides secure storage of its execution history with an append-only replicated ledger data structure (Androulaki, et al., 2018).

3.1.1 Architecture

The execute-order-validate blockchain architecture is introduced by Hyperledger Fabric, as shown in Figure 3.1, which does not follow the standard order-execute design (Androulaki, et al, 2018). There are four major components in this architecture.

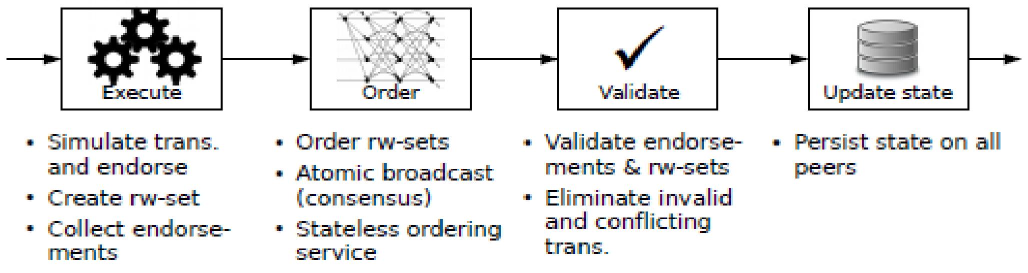

Execute
Order
Validate
Update state
Simulate trans. and endorse
Create rw-set
Collect endorse-ments
Order rw-sets
Atomic broadcast (consensus)
Stateless ordering service
Validate endorse-ments & rw-sets
Eliminate invalid and conflicting trans.
Persist state on all peers

Figure 3.1 Execute-Order-Validate architecture in Hyperledger.

(Androulaki, et al, 2018)

Peer Nodes are used to execute chaincodes, create smart contracts and maintain the system ledger in Hyperledger. There are two types of peer nodes, Endorsement Peers and Untrusted

Committing Peers. Endorsement peers are specified by the system for validation purposes and hold Chaincode logics, while the committing peers does not hold Chaincode Logics (Thakkar, et al., 2018). A copy of the system ledger is kept at both type of nodes.

Client Nodes are used to put up smart contract proposals and submit the proposal for endorsement. It will then broadcast the proposal to peer nodes, allowing the proposal to be added to a block and validated by committing nodes.

Chaincode is the smart contract used by Hyperledger Fabric. It implements the operational logic of the application and is executed during the execution phase. There are two types of Chaincode, User Chaincode and System Chaincode. User Chaincode can be written and executed on untrusted peers. As a result, it poses problems of non-deterministic execution and whether to trust the results from any given peer (Thakkar, et al., 2018). System Chaincode has the same programming model as User Chaincode, however, it is built into system execution and cannot be modified. Overall, Chaincode is considered the most important part of a decentralised application in Hyperledger (Androulaki, et al., 2018).

Endorsement Policy is responsible for the validation of transactions in Hyperledger Fabrics. The endorsement policy aims to address the problems posed by Chaincode's untrusted nature. Endorsement policies are usually specified as Boolean expressions over designated administrators (trusted users) in the network and can only be parametrised by these administrators (Thakkar, et al., 2018).

3.1.2 Transaction Flow

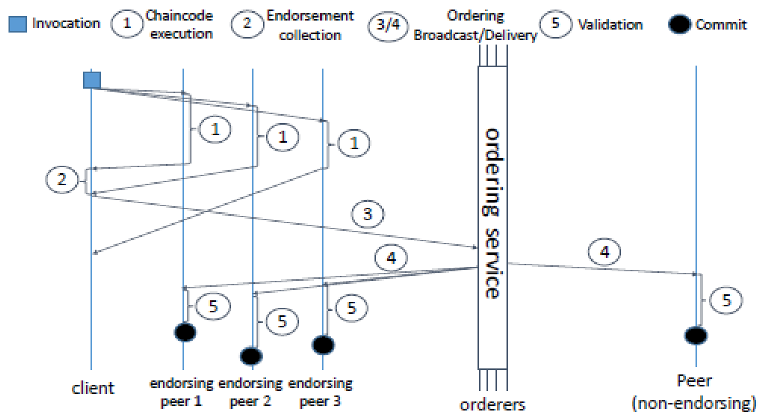


Figure 3.2 A sample transaction flow in Hyperledger.

(Androulaki, et al., 2018)

Execution Phase / Endorsement Phase is when a chaincode function is implemented by the client and sent to multiple endorsement nodes for simulation, as shown as step 1 in Figure 3.2. The transaction will be executed locally on peer and changes are written to a private temporary workplace. The final decision will then be sent back to the client by calling the ESCC system chaincode to sign the response with identifications of endorsement peers. The response will be collected by the client and verified in preparation for next phase, which is displayed as step 2 in the figure.

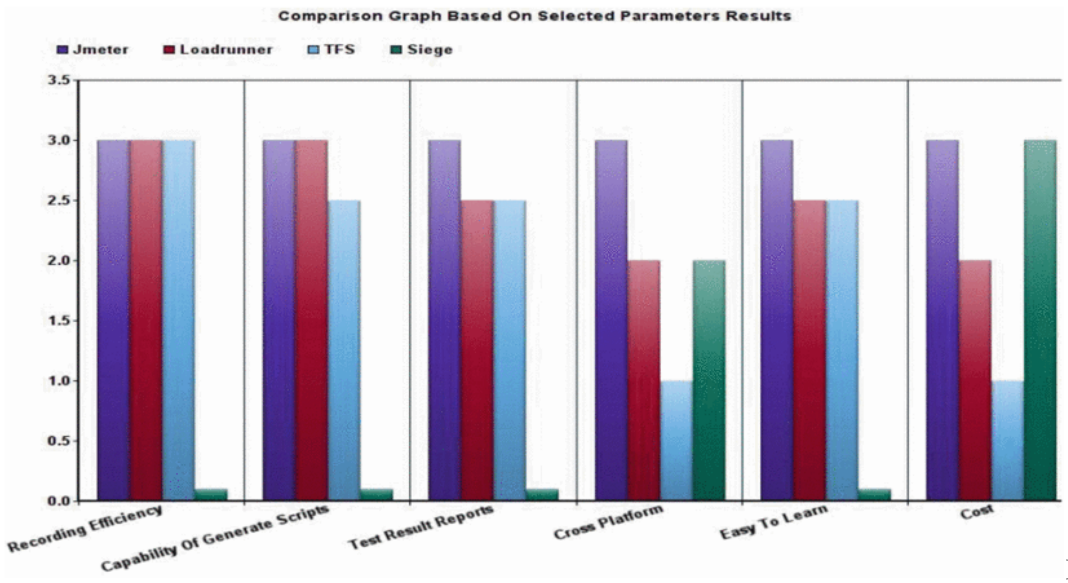
Ordering Phase is when a certain number of endorsement response is received by the client, a transaction with payload, metadata and a set of endorsement signatures would be constructed and sent to the Ordering Service, this step in shown in Figure 3.2 as step 3. The Ordering Service will not verify the transaction but join it into a block of transactions per-channel, sign a block and broadcast the block to a set of endorsing and committing peers (step 4 in Figure 3.2).

Validation Phase is when the transaction block is received by different nodes, it will be verified at the peers. The block will be first checked of the Ordering Service's signature, then it is decoded into separate transactions. The VSCC system chaincode is then called to evaluate whether the endorsement policy in the transaction matches the system endorsement policy, followed by the MVCC validation to ensure the key used in the transaction matches that in the system ledger (step 5 in Figure 3.2).

Committing / Ledger Updating Phase occurs after the validation is completed, the transaction is committed by the peers into a new block and the system ledger is updated at each peer. The State DB which hold the keys of all transactions is also updated with the latest records.

3.2 Apache JMeter

Apache JMeter is a highly extensible Java-based tool that can be used to conduct automate performance testing on web applications or services using HTTP or FTP servers (Nemerov, 2006). JMeter is flexible and can be used to generate a heavy workload and conduct automated tests on a group of servers to analyse the performance of an application over different workloads. It was originally intended for web applications but has now been extended to multiple platforms (Apache JMeter documentation, 2020). According to Halili et al. (2008), the adoption of automated testing during software development would provide a 543% Return on Investment (ROI), while that of manual testing would be only 350%.



E

Figure 3.3 Comparison between popular performance testing tools.

(Abbas, et al., 2017)

Figure 3.3 demonstrates a comparison test of four performance testing tools - JMeter, LoadRunner, TFS and Siege by Abbas et al. in 2017. JMeter received the highest mark in all aspects, including Recording Efficiency, Capability of Generating Scripts, Test Results Reports, Cross Platform Ability, the Ease to learn and Cost.

While all four tools are considered good for test automation, it is concluded that JMeter is best option as it is open source, free and provides various options for result analysis. It can also be easily extended with a large set of plugins (Abbas, et al., 2017).

3.2.1 Architecture

A JMeter *Test Plan* usually contains three parts, which are thread groups, samplers and listeners.

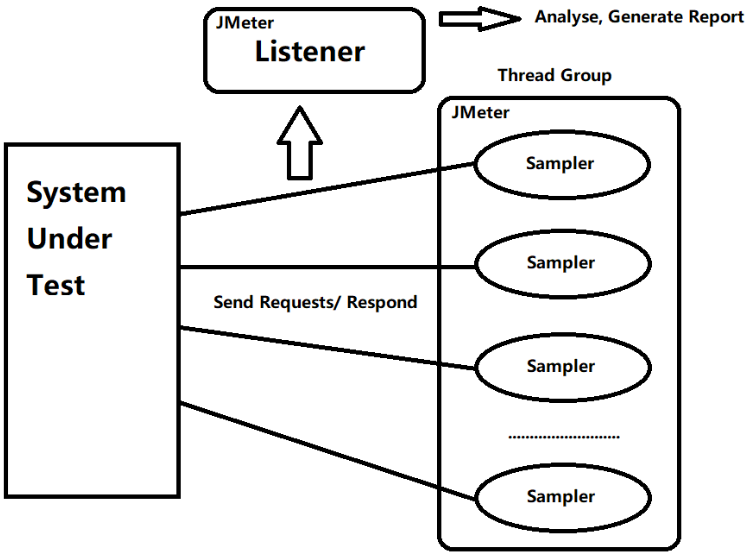


Figure 3.4 Architecture of JMeter.

As shown in Figure 3.4, a *Thread group* contains the threads to be used in the testing, each of which simulates a user. The *Samplers* consists of the methods and unit test cases selected by the user, it will also send the HTTP or FTP requests to the system under test and display simple success or failure messages. The *Listeners* can intercept the requests and responses between the sampler and the system under test and would transfer them into test scripts and records.

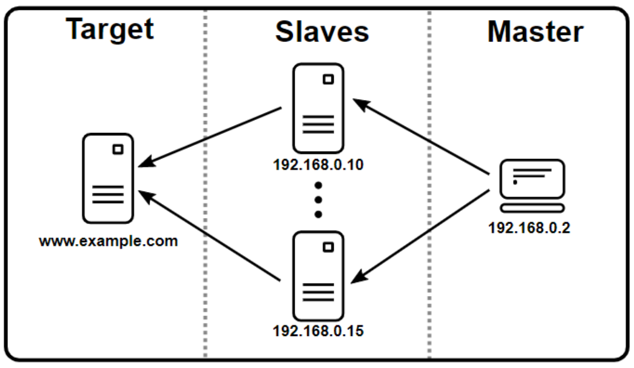


Figure 3.5 Distributed testing configuration in JMeter

(Apache JMeter Documentation, 2020)

JMeter also supports *Distributed Testing* as demonstrated in Figure 3.5. The Master node is the machine running JMeter GUI, which can remotely start and control the Slave nodes that each runs a JMeter server. The Slave nodes then send requests to the Target node, which is the system under test. The results will then be collected and post-processed by listeners.

3.3 Performance Testing of Private Blockchains

Since Hyperledger Fabric and Ethereum are currently two of the most popular platforms used in industry, we will focus on the methodology and tools for the performance evaluation of these two platforms.

3.3.1 Traditional Testing Methodology

Most researches of blockchain performance evaluation are set up on cloud machines, such as Amazon EC2. A typical software architecture would require a setup module, an evaluation and workload dispatch module and an output data collection module (Pongnumkul, et al., 2017).

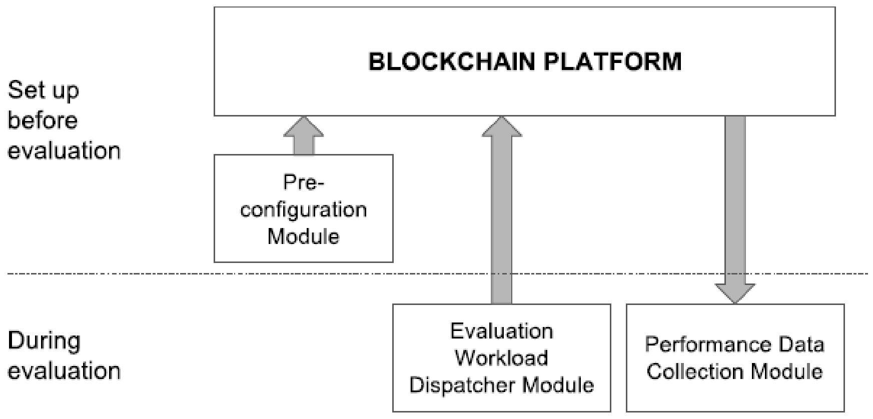


Figure 3.6 Software architecture of blockchain performance testing (Pongnumkul, et al., 2017)

The pre-configuration module is the environment setup process to configure the blockchain platform. The evaluation and workload module (Pongnumkul, et al., 2017), or load-generating client (Performance, et al., 2018), will send requests of transactions to the blockchain server asynchronously. The data collection module (Pongnumkul, et al., 2017), or observing client (Performance, et al., 2018) then collects the relevant data for processing. The two modules are referred to as the Test Harness (Performance, et al., 2018).

The *pre-configuration module* involves in application and smart contract simulation, during which a simple application with methods such as account creation, currency issuing and transaction is implemented (Pongnumkul, et al., 2017). Different blockchain platforms may require different languages and implementation for the methods. Ethereum, for example requires an account passphrase for private key decryption, while Hyperledger uses two key-value pairs to store account name and balance respectively.

```
contract TransferMoney {
  mapping (address => uint) balances;
  ...
  function sendCoin(address receiver,
  uint amount) returns(bool sufficient)
  {
    if (balances[msg.sender] < amount)
    return false;
      balances[msg.sender] -= amount;
      balances[receiver] += amount;
      return true;
  }
  ...
}
```

```
...
func (t *Chaincode) TransferMoney
    (stub shim.ChaincodeStubInterface,
    args []string) ([]byte, error) {
  Avalbytes, err := stub.GetState(A)
  Aval = strconv.Atoi(string(Avalbytes))
  ...
  Aval = Aval - amount
  Bval = Bval + amount
  ...
  stub.PutState(A,
  []byte(strconv.Itoa(Aval)))
  ...
}
...
```

Figure 3.7 Code snippets from TransferMoney function for Ethereum smart contract and Hyperledger Fabric chaincode. (Pongnumkul, et al., 2017)

Figure 3.7 demonstrates two code snippets of TransferMoney function for Ethereum and Hyperledger, written in Solidity and Golang respectively. From the snippets we can see that due to the different design of platforms, pre-configuration of the same methods can be very different, and generalisation would be difficult.

The *evaluation module* involves in experiment execution. The experiments vary in the number of HTTP requests and the different methods called, and results are averaged over several independent runs (Pongnumkul, et al., 2017).

The *data collection module* then collects the data from the experiments, such as execution time, latency and throughput.

3.3.2 Blockbench

Blockbench is a blockchain specific performance evaluation tool developed by Dinh et al. (2017). It is designed to help better understand blockchain systems and currently supports platforms including Hyperledger Fabric and Ethereum.

First, we will be looking at the architecture of Blockbench.

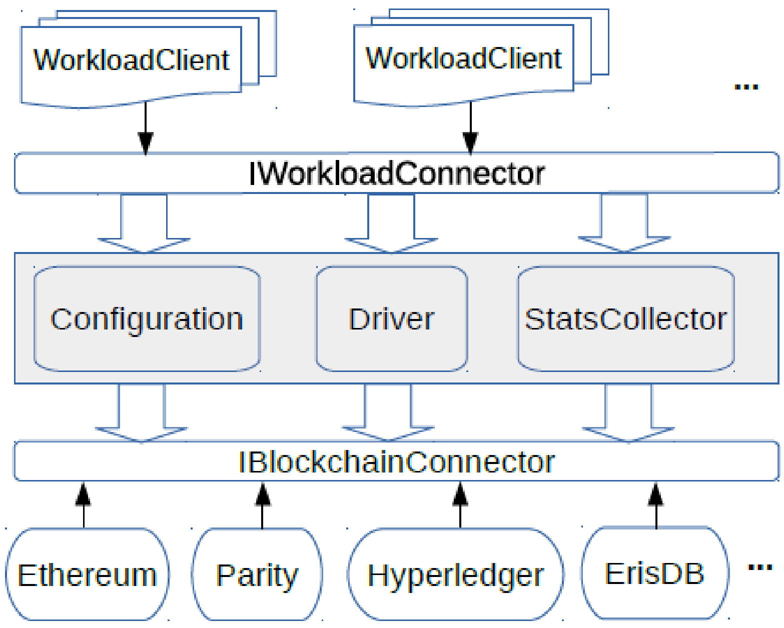


Figure 3.8 BLOCKBENCH software stack (Dinh, et al., 2017).

Software Stack Figure 3.8 demonstrates the software stack of Blockbench. Blockbench provides two connector interfaces, *IWorkloadConnector* and *IBlockchainConnector*, which allows different types of workload data and blockchain platforms to be supported. Other major internal components are Configuration, Driver and StatsCollector.

IWorkloadConnector allows for new workloads to be implemented. It essentially transforms raw workload data into transactions that can be processed by blockchains (Dinh, et al., 2017).

IBlockchainConnector provides the capability of application deployment, transaction invoking and blockchain state queries. As mentioned above, currently Parity, Hyperledger and Ethereum is supported.

Driver supports asynchronous transactions. Unlike traditional databases, blockchain transactions are not immediately processed after submission, which requires the Driver to keep track of the status of all unprocessed transactions. In order to achieve this requirement, Driver in Blockbench is designed to maintain a queue of unconfirmed transactions and update the queue frequently with worker threads (Dinh, et al., 2017).

Metrics While examining standard metrics like Latency and Throughput, Blockbench is also equipped with a *Security metric* set. In many consortium or private blockchains such as Hyperledger, Byzantine tolerant protocols such as PBFT proved to be able to secure the network. However, in public blockchains, PoW is usually applied as consensus protocol, and it can be vulnerable in cases that hackers control a certain partition of nodes in the network. In Blockbench, security is quantified as the number of nodes in blockchain forks (Dinh, et al., 2017). To examine the security of a certain network, Blockbench is designed to partition the network for certain durations to simulate the attacks.

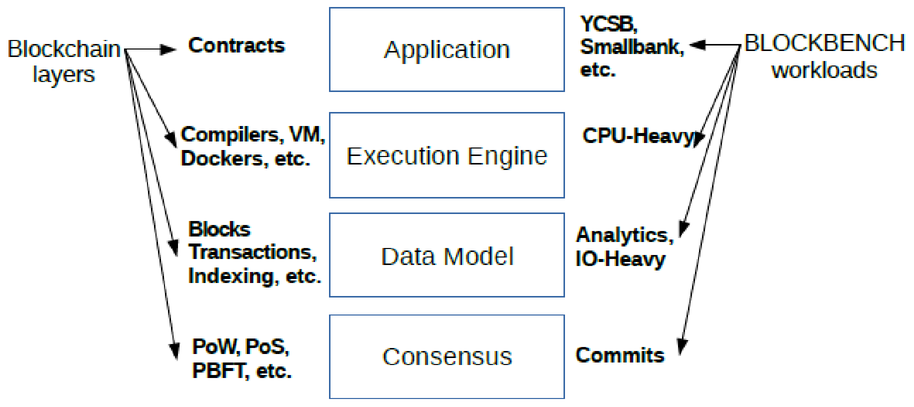


Figure 3.9 Abstraction layers in blockchain, and the corresponding workloads in BLOCKBENCH. (Dinh, et al., 2017)

Blockbench separates a traditional blockchain network into four abstract layers, each being Application Layer, Execution Layer, Data Layer and Consensus Layer, and implements four different types of workload for each layer. Generally, *Workload* is divided into two types, *Macro workload* for Application layer, and *Micro workload* for other layers (Dinh, et al., 2017). Among all the workloads implemented by Blockbench, YCSB, Smallbank, EtherId, Doubler and WavesPresale are Macro workload, while the others being Micro workload.

Chapter 4

Solution and Implementation

Over recent years, decentralised applications with blockchain backends have proven to be of great business potential. Therefore, it would be valuable to develop an application to provide a simple and automated evaluation of the performance of blockchains and decentralised blockchain applications so that businesses can select the appropriate platforms for their needs based on related performance evaluation. Furthermore, the performance evaluation of blockchains would require the blockchain wallet to be integrated into the plugin, such that it could directly communicate and append to the blockchain network. Consequently, the problem and relevant research on the topic can be summarised into a single problem statement:

How to extend Apache JMeter to support blockchain and blockchain-based decentralised application testing by supporting multiple asynchronous responses and timeouts longer than 2 minutes, as well as supporting blockchain wallet integration?

A JMeter plugin was implemented in this project to act as a link between JMeter and Blockchain DApps so that it combines the convenience of large workload generation with the ability to handle multiple responses and large timeouts. The plugin also comes with in-built support for ERC-20 token and Fabcar contracts provided in official Hyperledger Fabric samples. To ensure the flexibility and usability of the plugin, the implementation supports custom extensions so that potentially all Hyperledger Fabric smart contracts can be supported by this plugin. The final architecture and design of the project are discussed below in Section 4.1.

4.1 Overall Architecture and Core Idea

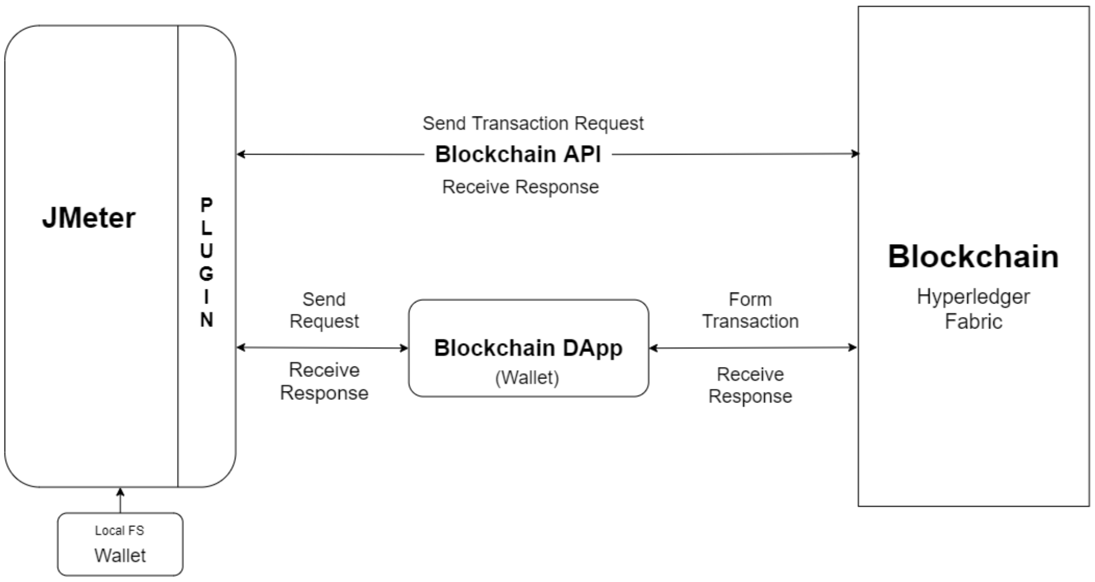


Figure 4.1 Application architecture

Figure 4.1 shows the overall structure of the project. While the project aims at the performance evaluation of blockchain DApps, the process would often involve the performance testing of blockchains as well, to analyse the performance overhead caused by the DApp itself. It would also be helpful to make sure a newly deployed blockchain reaches the desired performance metrics. Therefore, a mode selection option is provided in this plugin for users to choose whether the performance test would be performed on a DApp or a blockchain. The application would be connected to both JMeter and the blockchain or a DApp, depending on the connection mode that can be set through a parameter. In the first connection mode, which is meant for the performance testing of blockchain backends, the plugin would retrieve the blockchain wallets from local file system, and establish a direct connection from JMeter to the blockchain with parameters and transaction types supplied as JMeter parameters. In the second connection mode, the plugin would establish an HTTP connection to a DApp specified in JMeter parameters, which includes relevant transaction details and methods. The DApp would then communicate with the blockchain and append the transaction to it. This DApp connection mode is intended for the performance testing of blockchain DApps.

4.2 Technology Overview

This section provides a brief overview of the technical details involved in the development of this project. The two major platforms involved in this project are *JMeter* and *Hyperledger Fabric*.

JMeter was the base platform of the plugin. In this project, JMeter 5.3 was used and set up on Windows 10. Primarily, JMeter was studied and tested, which revealed that it can support async HTTP calls and timeouts with *Duration Assertions*. The plugin was then implemented to handle connections to and from Hyperledger Fabric blockchains and DApps.

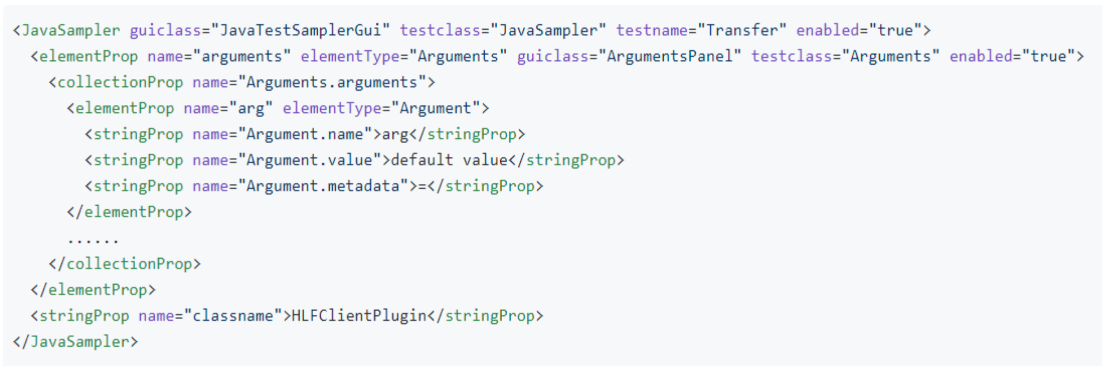
Hyperledger Fabric was used as the blockchain platform for testing in this project. It will also act as the backend of the blockchain DApps that would be tested by the plugin. The execution environment for Hyperledger Fabric will be a lightweight *mini fabric* platform provided by IBM.

Java is the main language used in the implementation in this project. As JMeter is implemented in Java, JMeter plugins have to be implemented in Java. Hyperledger Fabric gateway is also supported in Java, which was used in this project for the plugin implementation and the DApp implementation. In the development of this plugin, JDK 1.8 was used as the project interpreter, and IntelliJ IDEA was used as the IDE for this project.

4.3 Use of the plugin

After the plugin is configured to work with the local environment and the specified smart contract, it can be compiled and run with standard Java compilation in the project folder for debugging and testing before it is deployed in JMeter.

After the plugin is configured and tested properly, it can be deployed in JMeter for performance testing. An executable JAR file with dependencies can be generated by running *mvn assembly:assembly -DskipTests* command in the terminal. The compiled JAR file should be placed in JMeter external library folder. The plugin then should be configured in a test plan file as a *JavaSampler* sampler, as displayed below in Figure 4.2.



```
<JavaSampler guiclass="JavaTestSamplerGui" testclass="JavaSampler" testname="Transfer" enabled="true">
  <elementProp name="arguments" elementType="Arguments" guiclass="ArgumentsPanel" testclass="Arguments" enabled="true">
    <collectionProp name="Arguments.arguments">
      <elementProp name="arg" elementType="Argument">
        <stringProp name="Argument.name">arg</stringProp>
        <stringProp name="Argument.value">default value</stringProp>
        <stringProp name="Argument.metadata">=</stringProp>
      </elementProp>
      ......
    </collectionProp>
  </elementProp>
  <stringProp name="classname">HLFClientPlugin</stringProp>
</JavaSampler>
```

Figure 4.2 Configure the plugin in JMeter test plan

4.4 Implementation Details

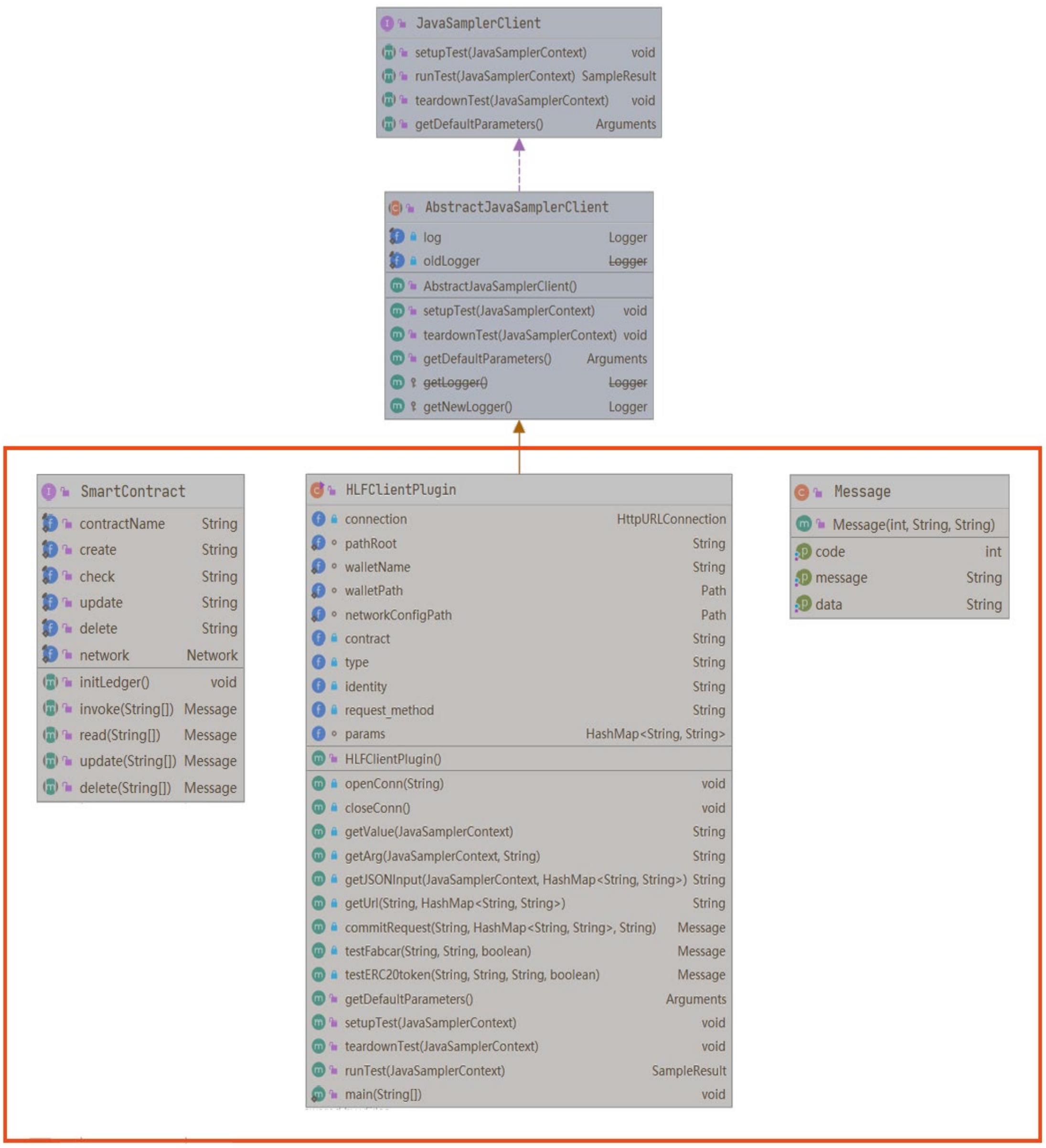

JavaSamplerClient
setupTest(JavaSamplerContext) void
runTest(JavaSamplerContext) SampleResult
teardownTest(JavaSamplerContext) void
getDefaultParameters() Arguments
AbstractJavaSamplerClient
log Logger
oldLogger ~~Logger~~
AbstractJavaSamplerClient()
setupTest(JavaSamplerContext) void
teardownTest(JavaSamplerContext) void
getDefaultParameters() Arguments
~~getLogger()~~ ~~Logger~~
getNewLogger() Logger
SmartContract
contractName String
create String
check String
update String
delete String
network Network
initLedger() void
invoke(String[]) Message
read(String[]) Message
update(String[]) Message
delete(String[]) Message
HLFClientPlugin
connection HttpURLConnection
pathRoot String
walletName String
walletPath Path
networkConfigPath Path
contract String
type String
identity String
request_method String
params HashMap<String, String>
HLFClientPlugin()
openConn(String) void
closeConn() void
getValue(JavaSamplerContext) String
getArg(JavaSamplerContext, String) String
getJSONInput(JavaSamplerContext, HashMap<String, String>) String
getUrl(String, HashMap<String, String>) String
commitRequest(String, HashMap<String, String>, String) Message
testFabcar(String, String, boolean) Message
testERC20token(String, String, String, boolean) Message
getDefaultParameters() Arguments
setupTest(JavaSamplerContext) void
teardownTest(JavaSamplerContext) void
runTest(JavaSamplerContext) SampleResult
main(String[]) void
Message
Message(int, String, String)
code int
message String
data String

Figure 4.3 UML Diagram of the final product

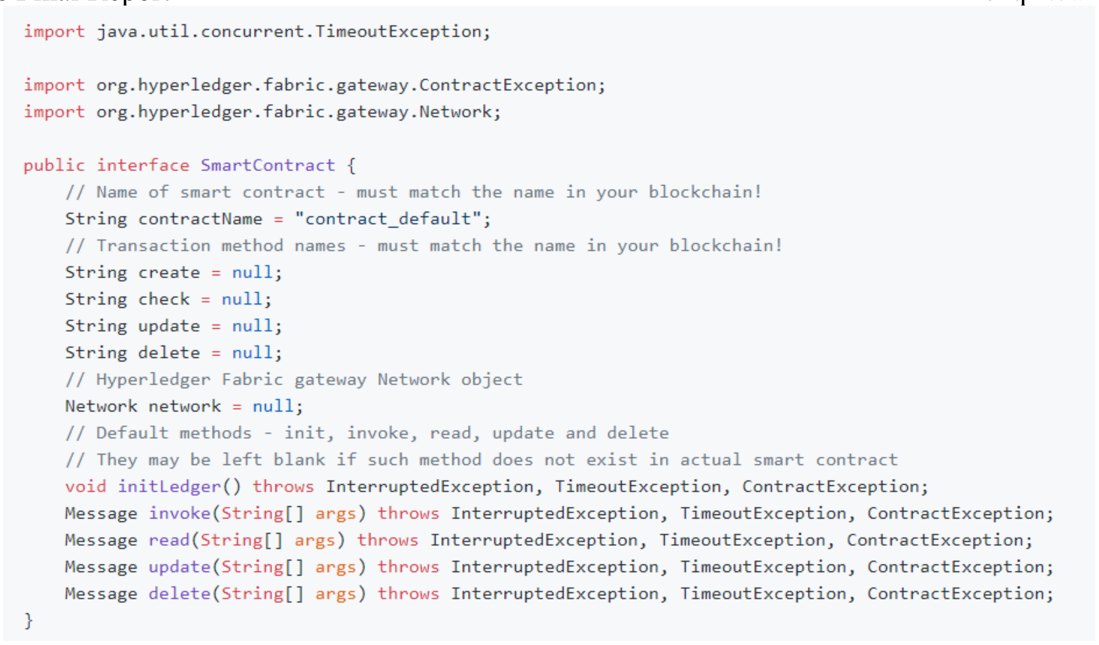
Figure 4.2 shows a UML class diagram of the final implementation of this application. The final plugin has 3 base classes, *HLFClientPlugin*, *SmartContract* and *Message*. The plugin also contains in-built support for an ERC-20 token contract, which is a popular cryptocurrency contract used by Ethereum.

The *HLFClientPlugin* class inherits the *AbstractJavaSampler* class provided by JMeter and acts as the main class for this plugin. This class controls the process of the performance testing, which includes storing the blockchain and DApp connection details, connection control, arguments handling, test running and result formatting. The plugin is also flexible and can be modified and extended to cater for potentially all Hyperledger Fabric smart contracts.

Table 4.1 List of Default Methods in HLFClientPlugin class

Method	***Value***
openConn	Open an *HTTPURLConnection* to designated address.
closeConn	Close the *HTTPURLConnection* created above.
getValue	Generate a random value given lower bound and upper bound for a transaction. If random_value option not set, use the upper bound as the return value.
getArg	Fetch an argument from JMeter input context.
defaultArgs	Get the default arguments for specified contract, if not given or executed in raw test mode, gives default value.
getJSONInput	Form a JSON body for POST requests given arguments and parameters.
getUrl	Form end URLs to DApp API endpoints given transaction type & parameters.
commitRequest	Form and commit an HTTP request given parameters and operation types, form responses into a Message object.
generateSC	Generate a SmartContract object given the smart contract to be used.
defaultTest	Test the default methods as given in the SmartContract Interface. Extra methods can be tested via implementing a extended test method and append to related section in this method.
getDefaultParameters	Inherited method. Set default parameters and default values.
setupTest	Inherited method. Setup test and argument handling.
teardownTest	Inherited method. Teardown the test when it is completed.
runTest	Inherited method. Conduct tests as requested and format results to JMeter format.

The SmartContract interface provides a unified interface for clients to add different smart contract test capabilities to the plugin. As shown in Figure 4.3, six attributes and five default methods consisting of basic read and write transactions are included in this interface, providing the basic functionalities of a Hyperledger Fabric smart contract. Custom methods corresponding to other specific smart contract functionalities may be added when a new *SmartContract* object is implemented for a specific smart contract. By implementing this class, this plugin can be used for the testing of different Hyperledger Fabric smart contracts.



```java
import java.util.concurrent.TimeoutException;

import org.hyperledger.fabric.gateway.ContractException;
import org.hyperledger.fabric.gateway.Network;

public interface SmartContract {
    // Name of smart contract - must match the name in your blockchain!
    String contractName = "contract_default";
    // Transaction method names - must match the name in your blockchain!
    String create = null;
    String check = null;
    String update = null;
    String delete = null;
    // Hyperledger Fabric gateway Network object
    Network network = null;
    // Default methods - init, invoke, read, update and delete
    // They may be left blank if such method does not exist in actual smart contract
    void initLedger() throws InterruptedException, TimeoutException, ContractException;
    Message invoke(String[] args) throws InterruptedException, TimeoutException, ContractException;
    Message read(String[] args) throws InterruptedException, TimeoutException, ContractException;
    Message update(String[] args) throws InterruptedException, TimeoutException, ContractException;
    Message delete(String[] args) throws InterruptedException, TimeoutException, ContractException;
}
```

Figure 4.4 SmartContract class implementation details

The *Message* class handles responses from transaction requests, and stores them in a format supported by JMeter so that test results can be formed into a JMeter *SampleResult* instance for further post-processing and summary reports when a test thread is completed. It has three attributes, *code*, *message* and *data*. The *code* field stores the HTTP response code of the request, while *message* and *data* stores response message and response data from the transaction requests.

4.5 Plugin Setup and Input Parameters

The Hyperledger JMeter plugin supports a wide range of custom parameters in order to support different Hyperledger Fabric smart contracts. The plugin comes with a list of pre-defined generic parameters, as displayed below in Table 4.1. The pre-defined parameters include compulsory parameters such as mode selection and transaction type selection, as well as generic parameters such as user identity and transaction value details to be used in transaction requests. The pre-defined parameters also included support for Fabcar and ERC-20 token contracts, which will be discussed below as a use case. More parameter can also be added to cater for different smart contracts. As the JMeter GUI currently does not support addition of parameters, parameters must be set through command line (if test plan executed in command line) or declared in *getDefaultParameters()* method in the *HLFClientPlugin* class.

Table 4.2 List of Parameters

Parameter	***Value***
type	The type of transaction to be executed. Expected input values and formats can be defined per user request.
uid1	User id 1 to be used in transaction.
uid2	User id 2 to be used in transaction.
random_value	*True* or *False*, decides whether random value generation is used in transactions.
minval	When using random value generation, the lower bound of generated value.
maxval	When using random value generation, the upper bound of generated value. If random value is not used, this value will be used as parameter in transactions.
direct	*True* or *False*, decides connection modes. When set as *True*, the plugin will attempt to establish direct connection to the blockchain, otherwise it will try to connect to a DApp with the URL value parameter given below.
organisation	The organization to use when connecting directly to the blockchain.
identity	The identity to use when connecting directly to the blockchain.
init	*True* or *False*, decides whether to initialise the blockchain. When set as *True*, the plugin will call related method to create user accounts and initialise the ledger with related commodity object, e.g., ERC-20 tokens.
contract	Name of smart contract to be used. Expected input values and formats can be defined per user request.
http_method	The http method to be used in DApp mode. Must be capitalized letters, e.g., *GET, POST*.
url	The root URL of the DApp to be tested. e.g. *http://localhost:5050/org-1-blockchain-functions/*

More custom parameters can be added by modifying the getDefaultParameters() method or through JMeter command line to cater for extensions to other smart contracts or generation of workloads with higher varieties.

4.6 Extension to Different Smart Contracts

To support different Hyperledger Fabric smart contracts such that the plugin can be applied to different potential test scenarios, I designed the plugin to provide the ability to extend to different smart contracts. The extension to different smart contracts requires the implementation of a *SmartContract* class object, and a few changes to the main class *HLFClientPlugin*. The following section provides a concrete example of smart contract extension based on the inbuilt ERC-20 token use case.

The ERC-20 token smart contract is a cryptocurrency smart contract provided by Ethereum and a widely-used and more practical smart contract example. In order to add ERC-20 support to our plugin, a SmartContract object class containing all necessary methods involved in an ERC-20 contract need to be implemented. A list of the methods implemented for this particular ERC-20 contract is displayed below in Table 4.3.

Table 4.3 List of Methods for ERC-20 Token Contract

Method	***Usage***
initLedger	Initialize the blockchain ledger with a few users and some currency supply.
invoke	Transfer some currency from minter to user.
read	Return the balance of a user given user ID.
update	Transfer between two users with a given amount.
delete	Not used in this contract, throws a HTTP 404 error/
sumAssertion	Check whether the value after a write operation falls within a range with a threshold to handle concurrent operations.
clientAccountID	Returns account ID of current user.
mint	Issue new currency into the ledger, requires minter identity.
allowance	Returns the allowance of a user given user ID.
approve	Ask for approval of an amount to spend for a user.
clientAccountBalance	Returns the balance of the current user.
totalSupply	Returns total supply of currency in the ledger.
generate_JSON	Generates a JSON string for DApp testings.

Some other changes are then required to integrate the new data type into the plugin, which includes a few appending to the *HLFClientPlugin* class. After all required changes are made, the plugin should now be able to handle transactions related to the new smart contract. The following Table 4.4 briefly explains the details of each appending.

Table 4.4 List of Appending for Smart Contract Extension

Method	***Details of Appending***
Extended test method	Implement an extended test method to test the extra functionalities provided in a smart contract, e.g., testErc20() method provided.
generateSC	Append new data type to be returned from this switch.
defaultTest	Append the extended test method to be executed in this method.
getDefaultParameters	Add new parameters if required by the contract.
defaultArgs	Add new argument handling with default values if needed.
setupTest, runTest	Add parameter retrieving or handling if needed.
JSON class	Implement a JSON class for HTTP requests.
getJSONInput	Append new JSON generation method to this method.
getUrl	Append DApp endpoint configuration for new DApp.

Chapter 5

Project Evaluation

This chapter mainly discusses evaluation of the plugin implemented in this project.

5.1 Experimental Setup

The evaluation of the plugin will be conducted on Windows 10 with local blockchain setup. The testing equipment is a laptop with Intel i7-9700H CPU and 16GB RAM. Figure 5.1 demonstrates a sample test structure of the experimental setup.

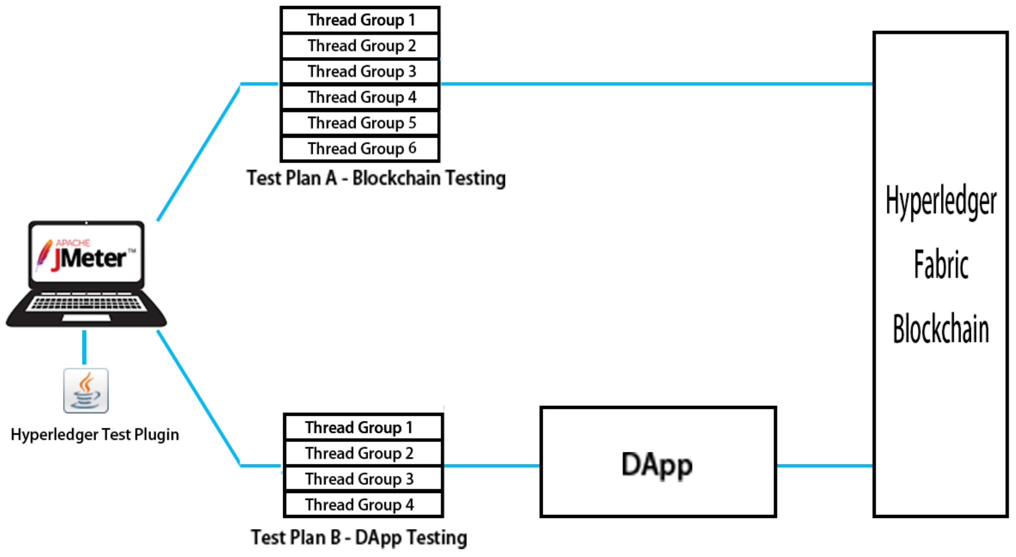


Figure 5.1 Experimental setup.

During the performance evaluation process, JMeter 5.3 and Hyperledger Fabric v2.0 will be used as the testing platform. The Hyperledger v2.0 mini fabric blockchain are setup locally with two organizations, with the help of IBM Blockchain VSCode plugin v2.0. The plugin is compiled with Java v1.8.212, with JMeter v5.3, Hyperledger gateway v2.0 and Alibaba Fast Json v1.2.7 included as dependencies. All the evaluation will be completed with the ERC-20 token smart contract provided in Hyperledger Fabric examples.

5.2 Evaluation

The evaluation of the plugin will be based on two sections, functional correctness and performance tests.

5.2.1 Functional Correctness

Functional correctness primarily evaluates the correctness of the plugin. Specifically, the functional correctness testing focused on whether the transactions are processed and appended to the blockchain correctly. As mentioned in Section 4.7, while Junit is not suitable for the functional correctness tests for this plugin, the plugin will be packed and tested in JMeter. During functional correctness tests, the plugin will execute different transactions sequentially in JMeter, with the ERC-20 token contract provided in Hyperledger Fabric samples, both for direct mode and DApp mode, where the response and transaction results will be evaluated against the blockchain output from the blockchain console. The plugin will also be tested on the accuracy of time measurements, so that we know the plugin reflects the real latency between submitting a transaction and its including in a block and avoid error in the data collected by the plugin.

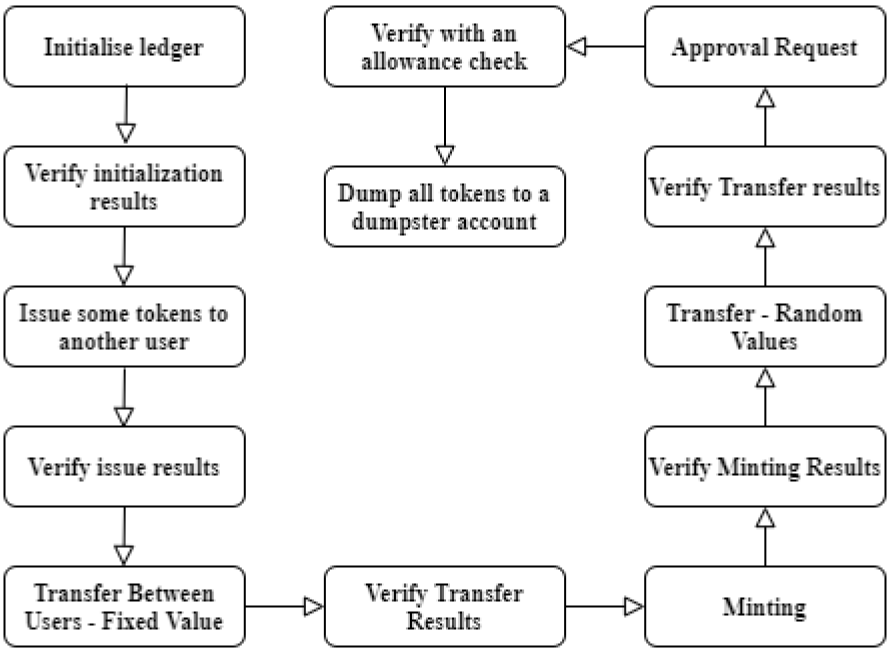
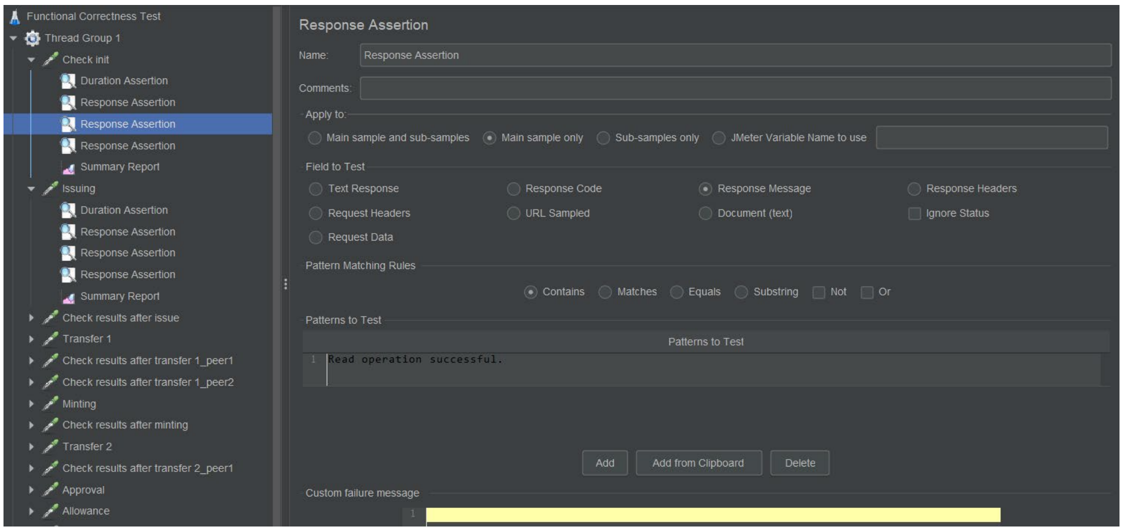

Initialise ledger
Verify initialization results
Issue some tokens to another user
Verify issue results
Transfer Between Users - Fixed Value
Verify Transfer Results
Minting
Verify Minting Results
Transfer - Random Values
Verify Transfer results
Approval Request
Verify with an allowance check
Dump all tokens to a dumpster account

Figure 5.2 Functional Correctness Test Example

As shown in Figure 5.2, a unit test plan for functional correctness test may contain a number of different blockchain transactions. For example, minting and transfer between users in ERC-20. Different combinations of transactions and execution orders were used to cover different scenarios that may occur during real-life use of the plugin. During each step, assertions were added to verify the response messages and response data given by the plugin, cross references were also performed against the blockchain logs to ensure the correctness of timing and transactions. Figure 5.3 shows an example of Java Sampler configuration to perform these tests. As displayed, multiple assertions are added to verify the response code, message and data returned by the plugin.

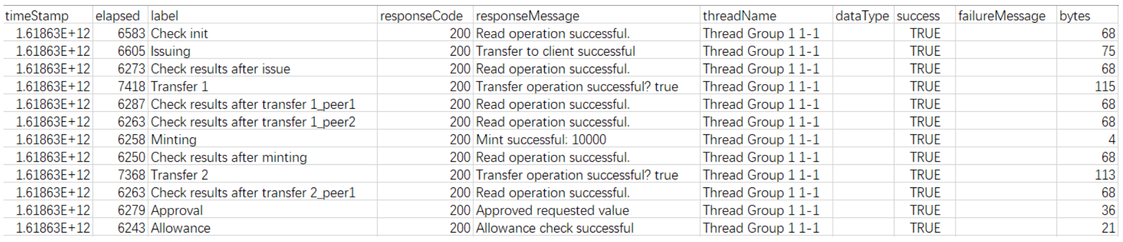


Functional Correctness Test
Thread Group 1
Check init
Duration Assertion
Response Assertion
Response Assertion
Response Assertion
Summary Report
Issuing
Duration Assertion
Response Assertion
Response Assertion
Response Assertion
Summary Report
Check results after issue
Transfer 1
Check results after transfer 1_peer1
Check results after transfer 1_peer2
Minting
Check results after minting
Transfer 2
Check results after transfer 2_peer1
Approval
Allowance

Response Assertion
Name: Response Assertion
Comments:
Apply to: Main sample and sub-samples [✓ Main sample only] Sub-samples only JMeter Variable Name to use
Field to Test: Text Response, Response Code, [✓ Response Message], Response Headers, Request Headers, URL Sampled, Document (text), Ignore Status, Request Data
Pattern Matching Rules: [✓ Contains] Matches Equals Substring Not Or
Patterns to Test:
```
Read operation successful.
```
Custom failure message

Figure 5.3 Example Configuration of Functional Correctness Test Samplers in JMeter

The results from functional correctness results were then aggregated into a summary report for analysis. Figure 5.4 shows a segment of summary report from one of the functional correctness tests. This test first initialised the ledger with 50,000 tokens, and the minter then issued 30,000 tokens to another user. A few transactions including transfer, minting, approval, and allowance were then performed to examine the correctness of the plugin. Failure and exception situations were also tested, with assertions to verify the correct error message had been thrown.



timeStamp	elapsed	label	responseCode	responseMessage	threadName	dataType	success	failureMessage	bytes
1.61863E+12	6583	Check init	200	Read operation successful.	Thread Group 1 1-1		TRUE		68
1.61863E+12	6605	Issuing	200	Transfer to client successful	Thread Group 1 1-1		TRUE		75
1.61863E+12	6273	Check results after issue	200	Read operation successful.	Thread Group 1 1-1		TRUE		68
1.61863E+12	7418	Transfer 1	200	Transfer operation successful? true	Thread Group 1 1-1		TRUE		115
1.61863E+12	6287	Check results after transfer 1_peer1	200	Read operation successful.	Thread Group 1 1-1		TRUE		68
1.61863E+12	6263	Check results after transfer 1_peer2	200	Read operation successful.	Thread Group 1 1-1		TRUE		68
1.61863E+12	6258	Minting	200	Mint successful: 10000	Thread Group 1 1-1		TRUE		4
1.61863E+12	6250	Check results after minting	200	Read operation successful.	Thread Group 1 1-1		TRUE		68
1.61863E+12	7368	Transfer 2	200	Transfer operation successful? true	Thread Group 1 1-1		TRUE		113
1.61863E+12	6263	Check results after transfer 2_peer1	200	Read operation successful.	Thread Group 1 1-1		TRUE		68
1.61863E+12	6279	Approval	200	Approved requested value	Thread Group 1 1-1		TRUE		36
1.61863E+12	6243	Allowance	200	Allowance check successful	Thread Group 1 1-1		TRUE		21

Figure 5.4 Example JMeter Summary Report for Functional Correctness Tests

Of all the functional correctness tests conducted in both blockchain and DApp modes, the plugin handled all of them correctly, and the results and timestamps corresponded with those provided by the blockchain DApp and blockchain logs. As a result, this plugin is considered functionally correct.

5.2.2 Performance Evaluation

Performance tests focus on the performance overhead brought by the plugin and the wallet integration, and the effect on the performance due to different workload composition. The plugin and the wallet integration should not slow down the workload generation, and as a result, performance testing was set up to monitor the performance overhead introduced by the plugin and the wallet.

The performance test of this plugin is completed with IBM mini-fabric Hyperledger blockchain and ERC-20 token smart contract provided by Hyperledger Fabric on Windows 10 with the environment setup discussed in section 5.1.

5.2.2.1 Estimating the performance overhead of the plugin

In the first part, the plugin's performance was compared against the performance of direct HTTP interactions with the DApp. First of all, a control sample was taken with JMeter HTTP request sampler and a simple Hyperledger DApp. The workload data remained the same for all tests in this part, and were left raw. The plugin was then tested with the same blockchain back-end and DApp used above, with the same read and write composition and request numbers. The results were then compared against the control group samples to verify whether the plugin introduced a significant performance overhead or caused increased error rates.

In this part, the workloads all consisted of four thread groups, two for reading operations and two for writing operations, each contained five threads. The number of loops each thread group was executed was set to match the seven-to-three read and write proportion. The number of total requests of the workloads started at 100, and was doubled each time, until it reached 6400. Ramp-up period for all tests were set at 40 seconds.

Figure 5.5, 5.6, and 5.7 show the average response time and throughput graph from the control group, plugin direct mode and plugin DApp mode respectively. As we can see for the control sample (see Figure 5.5) the average response time increased steadily from 100 requests to 400 requests, then sharply increased as the total request number increased towards 1600, and started to decrease afterwards, while the throughput increased steadily as the number of requests increased. In the plugin's direct mode (see Figure 5.6), the average response time of reading and writing operations remained to be steady despite the increasing number of requests. The response time of writing operations are about 0.6 second longer, due to the extra balance check and assertion procedure involved to ensure the correctness of the transaction. The throughputs of both reading and writing operations increased slowly as number of requests increase, and were much more gradual than the control sample. Overall, the throughput values were significantly smaller than the control sample, but a lot steadier, indicating the system did not fail due to large number of requests, but proceeded with processing all transactions properly. In conclusion, the plugin's direct mode did not result in a significant performance overhead, and was able to maintain a steady response time and throughput as the number of requests increased. The plugin's DApp mode (see Figure 5.7) achieved similar results as the direct mode. The average response times were steady, except that the writing group average response time was significantly lower (around 6400ms) than direct mode as the DApp's write operations involved fewer assertions and read transactions to assure correctness. The DApp mode also achieved a lower but steady throughput compared to the control sample.

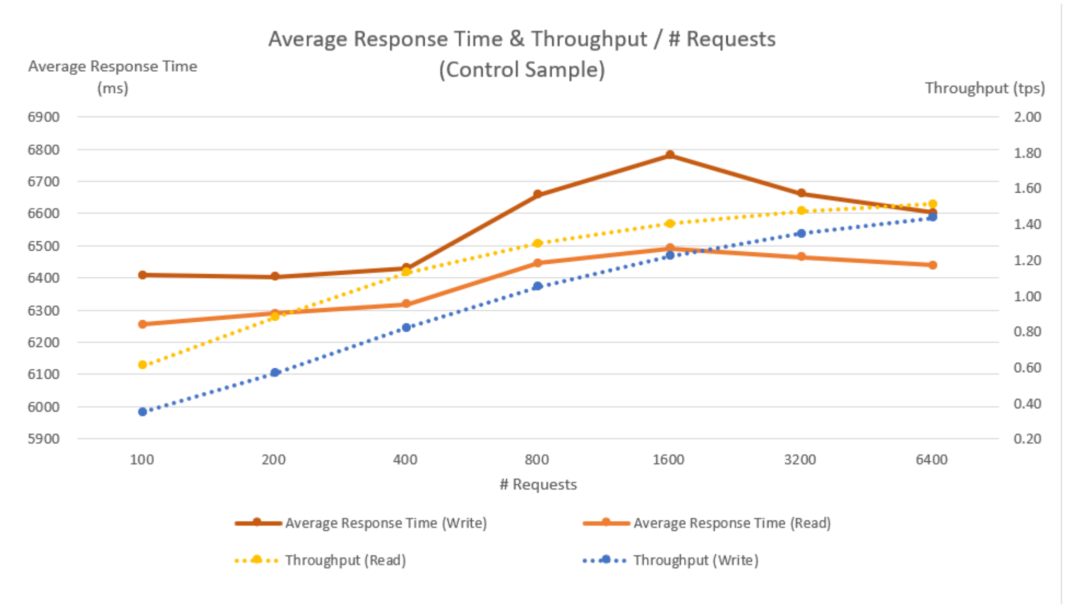


Figure 5.5 Control Group Average Response Time and Throughput Graph

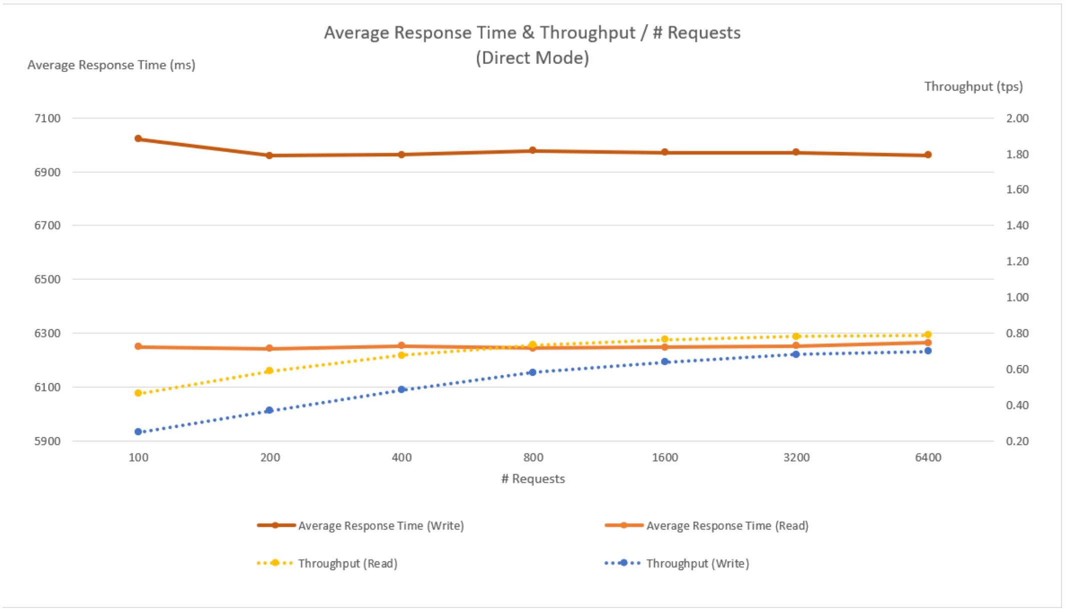


Figure 5.6 Plugin Direct Mode Average Response Time and Throughput Graph

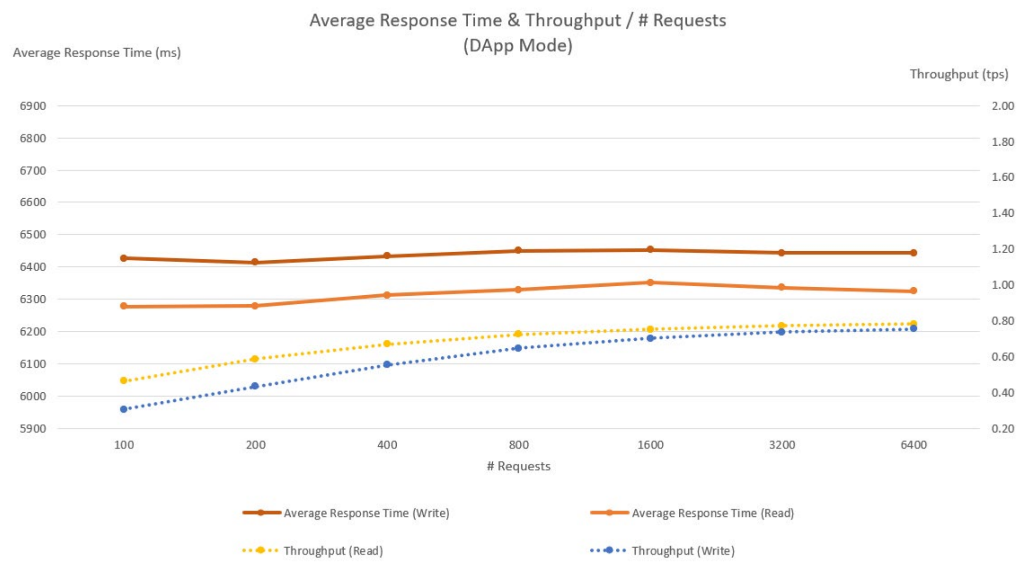


Figure 5.7 Plugin DApp Mode Average Response Time and Throughput Graph

Figure 5.8 below shows the overall error rates for the control sample, plugin direct mode and DApp mode. As can be seen in the graph, the control group had significantly higher error rates as the request number increased. Alternatively, the direct mode of the plugin was able to handle the increasing number of requests without problem and maintain a low error rate that can be up to 30% lower than the control group. The DApp mode of the plugin showed a similar trend to the control group, as the performance depends on the performance of the DApp used. However, the error rate of the plugin was 20% percent lower overall, compared to the control group, thanks to better flow control.

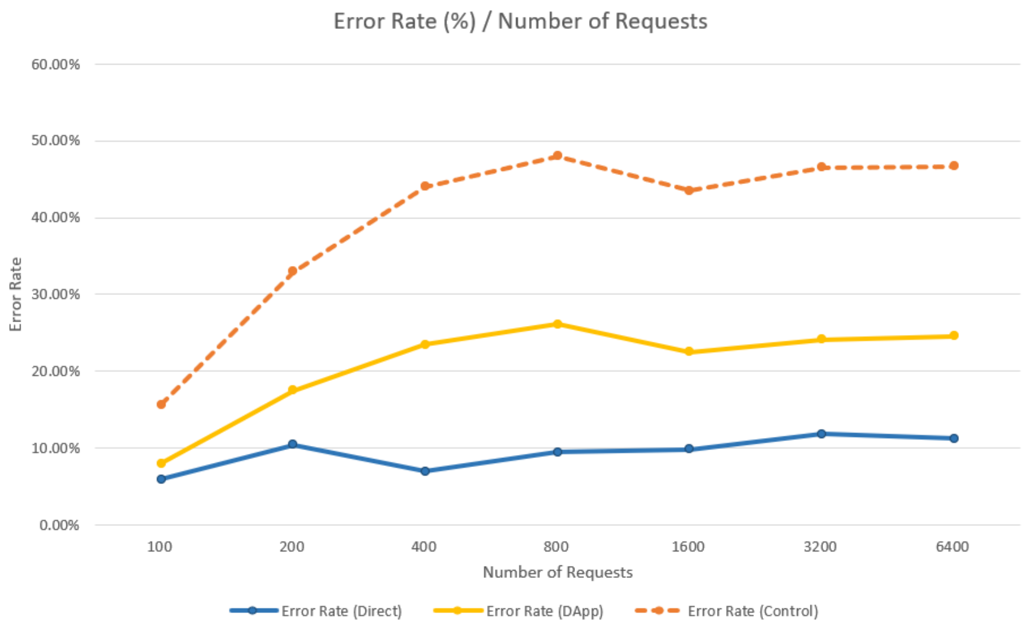


Figure 5.8 Error Rate / Number of Samples Graph for All Groups

From the evaluation results above, it can be seen that the plugin did not cause any significant performance overhead in both direct and DApp mode, despite the addition of workload variation and response handling. The plugin also achieved more steady and lower response times and significantly lower error rates compared to the control sample, proving its high efficiency and correctness.

5.2.2.2 The effects of different workload composition

In the second part, the plugin was tested with different compositions of read and write operations. A variety of workloads were introduced for the performance evaluation of the plugin. In this part, the same blockchain and DApp from the previous part were used, with ERC-20 token contract, and the total number of transactions were set at 500 requests. The compositions had six thread groups each with five threads. The workload percentage started from 10% write operations, and was increased by 10% each time until 60% write operations. In direct mode, write operations included transfer, approval and mint operations, while read operations consisted of allowance and balance checks. Three thread groups were used for write operations in direct mode. In DApp mode, two thread groups were used for write operations, one for transfer and one for mint, while the other four were used for read operations consisting of allowance, id and balance checks. Figure 5.9 illustrates two examples of workload, for direct mode and DApp mode respectively.

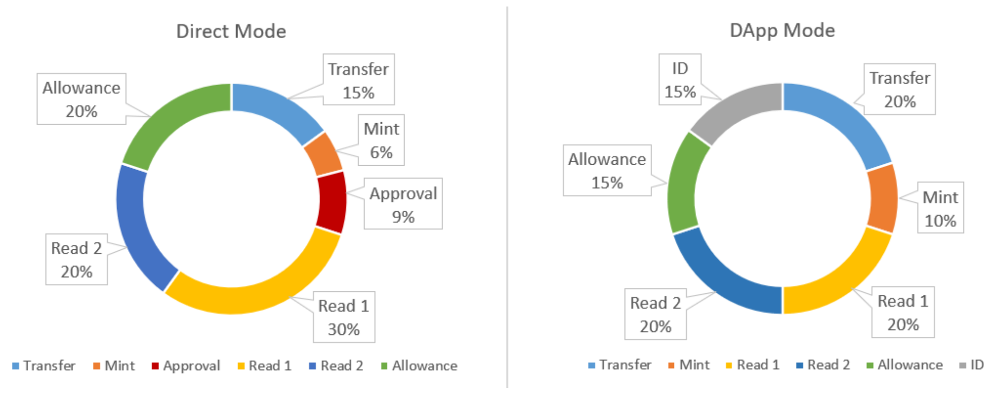


Figure 5.9 Workload Composition Examples

Figure 5.10 and 5.11 below shows the average response time and error rate of different workloads for direct mode and DApp mode respectively. As can be seen in figure 5.10, the reading and writing average response times were not affected by the change of read and write proportions in the workload and remained a consistent level. Both read and write error rates increased as the proportion of write requests increased from 10% to 40%. When the proportion of write operations exceeded 20%, the error rates were significantly higher, and remained rather steady as proportion of write operations further increased. In DApp mode (see Figure

5.11), the read and write times increased as write proportion grew to 30%, then remained constant. The overall error rate reached its highest when write proportion increased from 20% to 30%, when the write error rate spiked a 15% at the same time, then decreased steadily as write proportion further increased.

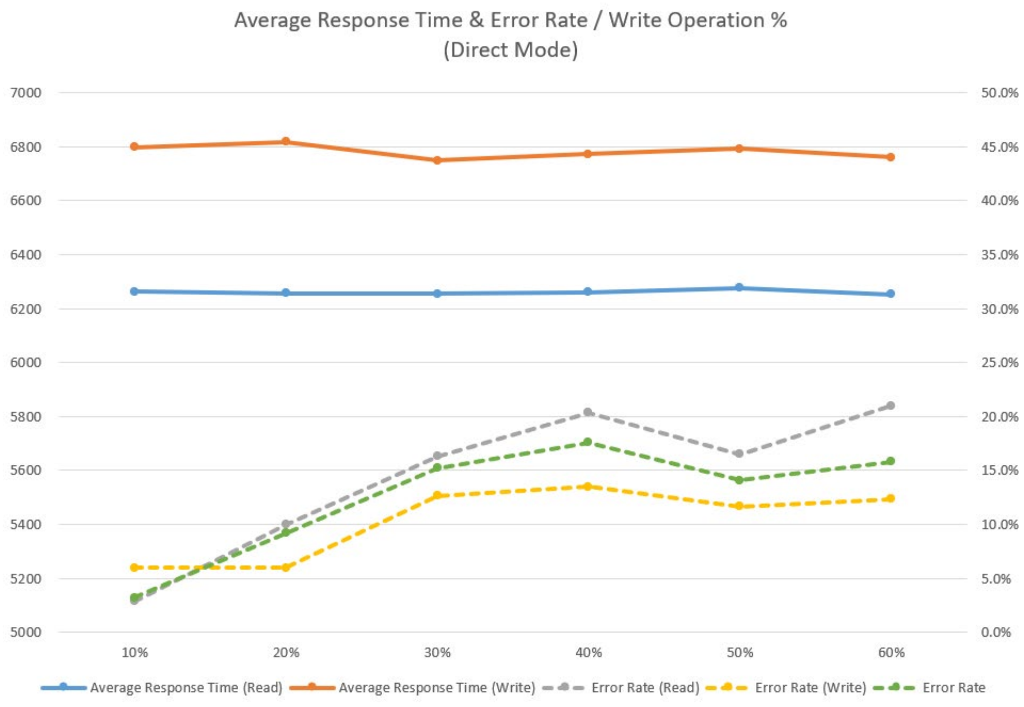


Figure 5.10 Average Response Time & Error Rate / Percentage of Write Operations in Direct Mode

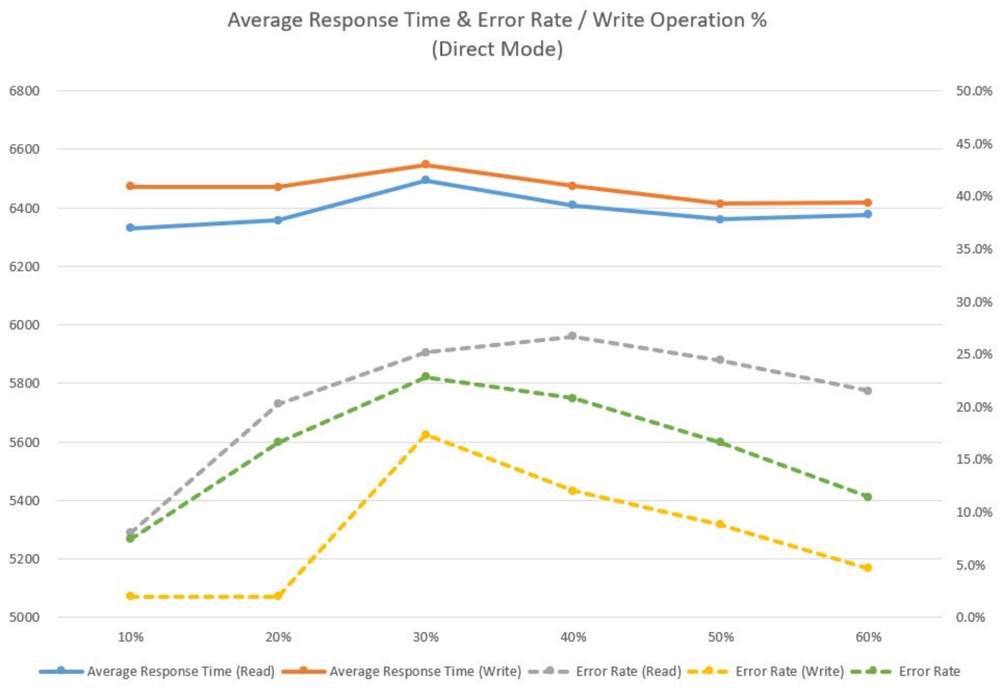


Figure 5.11 Average Response Time & Error Rate / Percentage of Write Operations in DApp Mode

In conclusion, it appeared that as the proportion of write operations increased, the average response time was not affected and remained steady with changes less than 0.1 seconds overall. On the other hand, the error rates of both read and write operations were affected by increased proportion of write operations, as both rates increased when write proportion became larger.

Chapter 6

Conclusion

6.1 Summary

This project implemented a JMeter plugin for Hyperledger Fabric to add Hyperledger blockchain and DApp testing capabilities to JMeter, handling async calls and catering for different smart contracts and large timeouts. The plugin supports customized parameters and allows the clients to customize the test workloads with different values and transaction types. Smart contract extension capability are also provided, with detailed documentations and guidelines so that different Hyperledger Fabric smart contracts can be tested for their performance with minimal changes to the plugin. The plugin was implemented in Java with three base classes, HLFClientPlugin, Message and SmartContract (Interface). The source code and template of the plugin contains around 600 lines of code, while extensions cost around 200 lines each. The plugin can both be executed with standard Java compilation without JMeter for debugging and testing, and in JMeter for performance evaluation.
During the evaluation process, functional correctness and performance of the plugin was thoroughly tested and the result indicated that the plugin is able to perform performance testings on both Hyperledger Fabric blockchain and DApps correctly and efficiently without causing a significant performance impact on the execution.

6.2 Limitations

Due to the various restrictions caused by JMeter and Hyperledger Fabric, the plugin currently has a few limitations that resulted in inconvenience when testing and reduces flexibility. Firstly, all the public methods in the plugin's main class only takes in JMeter test context, which is not customizable during raw execution. As JMeter plugins also does not allow command-line arguments during raw execution, it is difficult to conduct functional correctness testing with Junit, as parameters cannot be passed in or acquired from default during raw execution. The same issue also resulted in extra lines of code when doing an extension to other smart contracts, as extra methods are added to supply default values for arguments and support local testing. Secondly, as Hyperledger Fabric gateway does not allow non-static blockchain connection details, and JMeter arguments are only retrieved after the plugin main class has been instantiated, it is not possible to change the connection details as per given in test parameters. In consequence, currently different connection details have to be set before the plugin is packed, possibly under different names, and a switch is implemented to choose the specific connection profile to use based on client choice given in parameters.

6.3 Future Work

Future work is planned on this plugin to improve the flexibility and ease of use. As mentioned above, Hyperledger Fabric posed limitations to the implementation of the plugin. As a result, work is planned to be done on the connection profile configuration and smart contract extension part to allow for easier connection profile management and organization switches, as well as more concentrated smart contract extension, with fewer lines of codes. Additionally, we would like to extend the plugin to other private/consortium blockchain platforms such as Ethereum, and add more workload generation options to provide a better variety of workloads. Lastly, the plugin is also to be improved with source code refactoring and new public methods to pass in arguments during raw execution and simplify the process of debugging and testing prior to deployment in JMeter.

References

Abbas, R., Sultan, Z., & Bhatti, S. N. (2017, April). Comparative analysis of automated load testing tools: Apache JMeter, Microsoft Visual Studio (TFS), LoadRunner, Siege. In 2017 International Conference on Communication Technologies (ComTech) (pp. 39-44).

Androulaki, E., Barger, A., Bortnikov, V., Cachin, C., Christidis, K., De Caro, A., ... & Muralidharan, S. (2018, April). Hyperledger Fabric: A distributed operating system for permissioned blockchains. In Proceedings of the thirteenth EuroSys conference (pp. 1-15).

Apache JMeter Documentation. 25. Apache JMeter Distributed Testing Step-by-step. https://jmeter.apache.org/usermanual/jmeter_distributed_testing_step_by_step.html

"ASX selects distributed ledger technology to replace CHESS". ASX, December 2017.

Carson, B., Romanelli, G., Walsh, P., & Zhumaev, A. (2018). Blockchain beyond the hype: What is the strategic business value. McKinsey & Company, 1-13.

Hamilton, A.C. (2019, Nov). Corda v Hyperledger v Quorum v Ethereum v Bitcoin, https://medium.com/corda/corda-v-hyperledger-v-quorum-v-ethereum-v-bitcoin-58f2f0890dce/

Dinh, T. T. A., Wang, J., Chen, G., Liu, R., Ooi, B. C., & Tan, K. L. (2017, May). Blockbench: A framework for analysing private blockchains. In Proceedings of the 2017 ACM International Conference on Management of Data (pp. 1085-1100).

Halili, E. H. (2008). Apache JMeter: A practical beginner's guide to automated testing and performance measurement for your websites. Packt Publishing Ltd.

Khan, R., & Amjad, M. (2016, April). Web application's performance testing using HP LoadRunner and CA Wily introscope tools. In 2016 International Conference on Computing, Communication and Automation (ICCCA) (pp. 802-806). IEEE.

Performance, H., & Scale Working Group. (2018). Hyperledger Blockchain Performance Metrics. Whitepaper. https://www.hyperledger.org/wpcontent/uploads/2018/10/HL_Whitepaper_Metrics_PDF_V1, 1.

Pongnumkul, S., Siripanpornchana, C., & Thajchayapong, S. (2017, July). Performance analysis of private blockchain platforms in varying workloads. In 2017 26th International Conference on Computer Communication and Networks (ICCCN) (pp. 1-6). IEEE.

Nevedrov, D. (2006). Using JMeter to Performance Test Web Services. Published on dev2dev, 1-11.

Thakkar, P., Nathan, S., & Viswanathan, B. (2018, September). Performance benchmarking and optimising Hyperledger Fabric blockchain platform. In 2018 IEEE 26th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS) (pp. 264-276).

Zheng, Z., Xie, S., Dai, H., Chen, X., & Wang, H. (2017, June). An overview of blockchain technology: Architecture, consensus, and future trends. In 2017 IEEE International Congress on Big Data (BigData congress) (pp. 557-564).