SaaS - Microservices-based Scalable Smart Contract Architecture

Eranga Bandara[1], Xueping Liang[2], Peter Foytik[1], Sachin Shetty[1], Nalin Ranasinghe[3], Kasun De Zoysa[3], and Wee Keong Ng[4]

[1] Old Dominion University, Virginia, USA
{cmedawer, pfoytik, sshetty}@odu.edu
[2] University of North Carolina at Greensboro, North Carolina, USA
{x_liang}@uncg.edu
[3] University of Colombo School of Computing, Sri Lanka
{dnr, kasun}@ucsc.cmb.ac.lk
[4] School of Computer Science and Engineering
Nanyang Technological University, Singapore
{awkng}@ntu.edu.sg

Abstract. Existing blockchain smart contract platforms are designed as monolithic architectures. Even though there are multiple smart contracts with fully independent business logic, they run on a single monolithic container. This dependence on a monolithic container can be a performance bottleneck during the processing of a large number of transactions. To address this challenge, microservice-based architecture is adopted in the blockchain smart contracts by introducing a novel architecture to run independently on separate microservices. The new smart contract architecture is built on top of Mystiko blockchain, a functional programming and actor-based "Aplos" concurrent smart contract platform. Aplos is identified as a "Smart Actor" platform since it is built using Actor-based concurrency handling. Based on the philosophy of microservices, the Aplos Smart Actor platform on Mystiko blockchain is redesigned. This architecture is introduced as "SaaS - Smart actors as a service". With SaaS, different Aplos smart actors in the blockchain are deployed as separate independent services(e.g docker containers) instead of a single monolith service. This ensures different smart actors can execute transactions independently. An additional benefits to SaaS is that the architecture increases the scalability by guaranteeing concurrent execution of transactions, producing high transaction throughput on the blockchain.

Keywords: Blockchain; Microservices; Actors ; Smart Contract; Functional Programming

1 Introduction

Most blockchain platforms introduce the function of "smart contracts" to interact with the blockchain ledger through scripting and programming. Blockchain software programs and platforms are written with these smart contracts in a

way that users can interact with them. Most of the existing blockchain smart contracts platforms are designed as a Monolithic architecture [34]. Though there are multiple smart contracts with fully independent business logic, they run on a single monolithic container. Because of this design architecture, it's not possible to run different smart contracts independently even though smart contracts have no dependency between them. This produces less transaction throughput on blockchain and scalability suffers. To address this concern we have adapted a microservices philosophy into the smart contract design.

Microservice [34] is a new architecture that is widely used to design highly scalable distributed software systems. Instead of building systems as single monolithic systems, microservices builds them as multiple small services which are introduced as microservices. These services, built based on Unix Philosophy, are built to achieve one task perfectly. To work on different tasks, the microservices communicate with other services. Communication between services is managed using message brokers like Kafka [23], AMQP or REST [16] APIs based on a reactive programming philosophy. Existing blockchain system architectures are built as monolithic systems where consensus handling, block creating, and sharing happens as a single monolithic service. This monolithic architecture produces considerable challenges when the blockchain system is scaled up. To address this concern Mystiko blockchain platform is restructured with a microservices architecture designed in its blockchain platform. The consensus handling, storage, block generation and smart contracts functions are handled on different small microservices within Mystiko. It leads to a higher scalable blockchain based system producing more throughput and allowing more transactions to occur.

In this research, we have adapted a microservices philosophy into blockchain smart contracts and built scalable smart contract architecture for real-time applications that demand high throughput transactions. Instead of building software systems as single monolithic systems, microservices are built as multiple small services. These services are built based on Unix Philosophy, "Do only one thing well". Microservices communicate with other collectively to perform complex operations. To communicate between the services, message brokers like Kafka, AMQP or REST APIs based on reactive programming philosophy are used. Based on the microservice philosophy Aplos smart actors platform on Mystiko blockchain is redesigned. Instead of having a single monolithic service to run smart actors, independent smart actors run on separate services as docker containers. This architecture, "SaaS - Smart actors as a service", allows different smart actors to execute transactions independently increasing the scalability and transaction throughput on the blockchain.

1.1 Outline

The structure of the paper has been organized in the following. Section II discusses the design of the Mystiko blockchain architecture. A detailed description of SaaS architecture is presented in Section III. Section IV performance evaluation. Section V Related works. Section VI conclusion and future works.

2 Mystiko Blockchain

2.1 Mystiko Overview

Mystiko is an enterprise blockchain platform which targeted for highly scalable, concurrent applications such as big data, IoT, smart cities etc. It designed with using Apache Kafka-based [23] federated consensus [25]. Mystiko uses Apache Cassandra [26] as it's asset storage and facilitate the full-text search on Blockchain data with using Lucene index [9] based API [17]. The federated learning services in the Mystiko blockchain capable of performing data analytics and machine learning functions with the blockchain data in a privacy-preserving manner. Three main performance bottlenecks on existing Blockchain platforms are investigated and addressed, namely order-execute architecture, full node data replication, and imperative style smart contracts.

To address the issues in the "Order-Execute" architecture and support real-time transactions, "Validate-Execute-Group" blockchain architecture is proposed. This architecture supported to validate and execute transactions concurrently when clients submit them to the network [7]. The new architecture provides high transaction throughput, high scalability and lightweight consensus [25] in the Mystiko blockchain. This architecture first validates and check the double spend [29] when clients submit transactions to the blockchain. Then execute the transaction with the smart contract and replicate state updates on the ledger. Finally, create blocks based on the executed transactions. To achieve real-time transactions in this model, it required strong consistent storage with linearizable consistency [35]. Mystiko used eventually consistent distributed storage and built a linearizable consistency model on it [3, 24].

To address the issues with existing imperative style smart contracts and support concurrent transactions, Mystiko introduces functional programming [22] and actor [20] based Aplos smart contract platform(which introduced as Aplos smart actor platform). Blockchain programs(smart contracts) is written using actors. Different actors may interact with one another via message passing. This smart actor platform supports concurrent execution of transactions; this yields high transaction throughput and scalability. Scalable and concurrent applications always deal with back-pressure operations [14]. Mystiko uses reactive-streaming [2, 13, 23] based approach to handle the back-pressure operations. Clients can submit the transactions to the blockchain as streams. Smart contract actors in the blockchain will stream these transactions and execute them. Mystiko blockchain used an eventual consistency distributed storage as the underlying storage platform [26]. Every peer in the network will have its own storage node. These storage nodes are connected as a ring cluster. All the transaction, blocks, assets will be stored in this storage. After a node executes a transaction with smart contracts, the state updates will be replicated to other nodes via sharding [36].

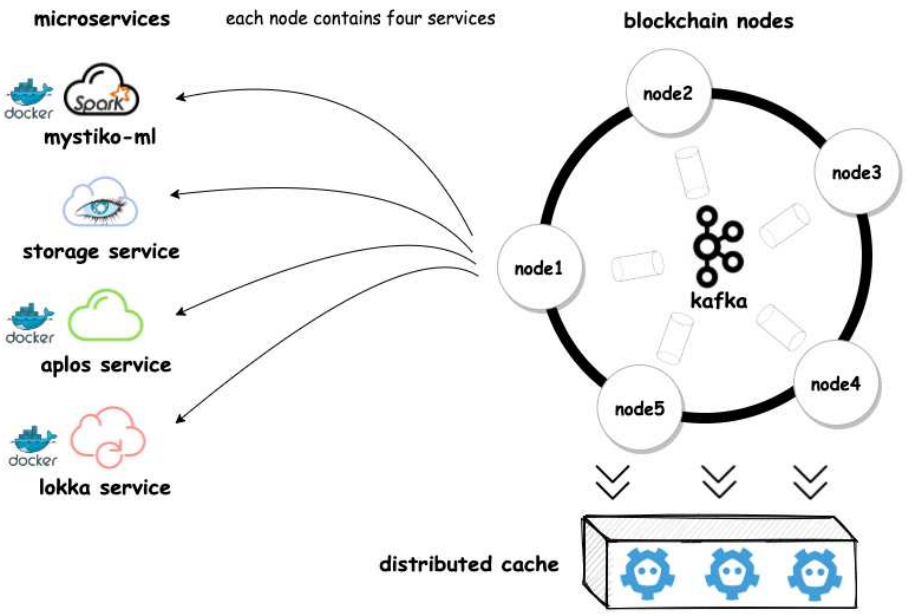

Fig. 1: Mystiko blockchain microservices-based architecture. Each blockchain node contains four services - Mystiko-ML service, Storage service, Aplos service and Lokka service.

2.2 Mystiko Architecture

Mystiko blockchain designed with using Microservices-based [5,34] architecture. The consensus handling, smart contracts, asset storage and block generation functions implemented in independent microservices services in the Mystiko blockchain. All these microservices are dockerized [28] and available for deployment using Kubernetes [10]. The architecture of the Mystiko blockchain described in Figure 1. Following are the main services/components of Mystiko blockchain:

1. Aplos service - Smart contract service.
2. Storage service - Blockchain asset storage.
3. Lokka service - Block generation service.
4. Apache Kafka - Consensus and message broker service.
5. Mystiko-Ml - Federated machine learning service.

2.3 Aplos Smart Actors

Mystiko blockchain introduced Scala functional programming language [4,22,31] and Akka actor [1,18] based Aplos smart contract platform [8]. Aplos introduced

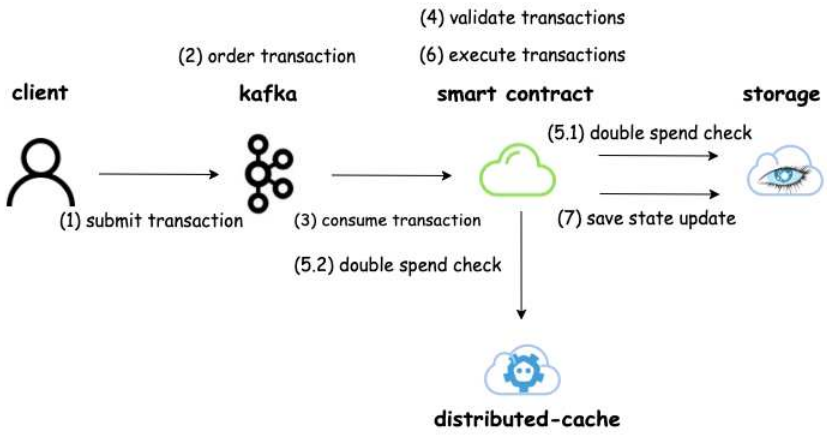

Fig. 2: Mystiko blockchain transaction flow. Two step transaction validation happens with asset storage and distributed cache.

as a Smart Actor platform since it built using Actor-based concurrency handling [20,21]. It enables concurrent transaction execution on Mystiko blockchain and produces high transaction throughput [8]. The transaction flow of the Mystiko blockchain with Aplos smart actors is described in Figure 2. All blockchain-based software programs and the messages that pass between them are written as Akka actors [1,18] and saved in the Aplos service. Each blockchain node in the cluster has Aplos service. Theses services consume transactions from Kafka message broker. Each Aplos service is connected to separate Kafka topic and consume transactions parallelly. Clients publish transactions to these Kafka topics. A transaction message contains a JSON encoded object with smart actor name and transaction parameters. Aplos service streams these transaction messages from Kafka and delegates them to corresponding smart actors based on the actor name in the message. Once a transaction message receives, smart actor validates and executes the transaction and generates the asset update. Finally, the asset update saved in the storage service and replicated with other blockchain nodes in the network as shown in Figure 2.

3 SaaS

3.1 Overview

SaaS "Smart actors as a service" is a new architecture that we are proposing to run smart contracts on top of the blockchain network. In a blockchain environment, there will be multiple smart contracts. Traditionally, all of these smart contracts run on a single monolithic service even though there is no interconnection between them. We have been identified that this would lead to major

performance bottleneck on the blockchain architecture. With SaaS architecture, instead of having a single monolithic smart contract service, independent smart contracts run on separate services(microservices) as actors. The SaaS smart contract architecture built on top of Mystiko blockchain Aplos smart actor platform. With SaaS, microservices architecture is introduced into Aplos smart actors on Mystiko. In this way, different smart actors can execute transactions independently. It will increase the scalability and produce high transaction throughput on the blockchain.

3.2 SaaS Scenario

As a use case of SaaS, we have built a document approving application on top of Mystiko blockchain using SaaS architecture. This application can be used to automate the document approval process on an organization(or cross-organization). Current document approval in an organization happens as a manual process. When multiple persons(e.g. multiple managers) need to sign a document, an employee of the company takes the document from one manager to another by hand and gets the document approved(e.g. signatures). We automate this process by using the Lekana platform. An administrator can upload documents and define the signatories of the document. It can define signing flow, assume A, B, C managers need to sign and the flow would be first A, second B, and third C. Once this document is uploaded to the Lekana, it will be notified to the first manager that needs to be signed via Lekana mobile application. When the notification is received, the manager pulls the document to the mobile application and approves or rejects it. When the document is approved, it digitally signs the document as well as adds the physical signature to the document by using PDF annotations. Once the first manager signs, the second manager will be notified. All document creating, notifying, user management functions are handled by smart actors on Mystiko blockchain.

3.3 Smart Contract Services

In Aplos, the business logic of the blockchain applications is written using Akka actors. Actors consume transaction messages and execute business logic based on the message parameters. `AccountActor`, `DocumentActor`, and `DeviceActor` are the three smart actors on Lekana platform. The user account related functions(account creation, activation) handles with the `AccountActor`. The push notification related functions handle with `DeviceActor`. The `DocumentActor` handles document creation, update, approval functions. With SaaS architecture, all these actors run as separate services(microservice) on Mystiko blockchain. These actors consume messages via Apache Kafka with Akka streams. Each actor service has its Kafka topic, Figure 3. Clients submit messages to these actor services via Kafka as transaction messages. For an example when creating a document client summit transaction message shown in Figure 5. When approving a document client submits the transaction message shown in Figure 4. These messages first go to Gateway service in the Mystiko. Gateway service identifies

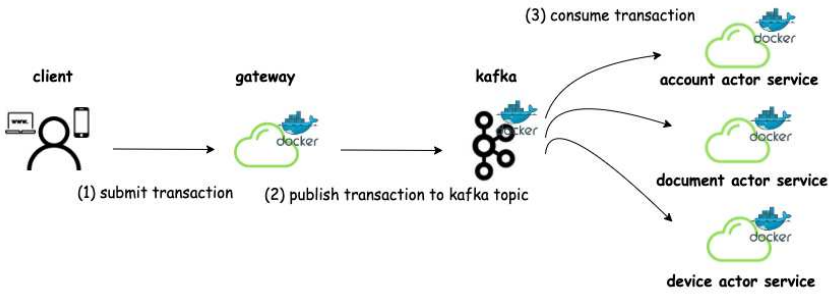

Fig. 3: Smart actor services in Lekana application. Different smart actors run independently on top of Docker containers.

```
{
   "id": "<transaction id>",
   "execer": "<transaction executing user>",
   "messageType": "sign",
   "documentId": "<document id>",
   "documentSigner": "<document signing user>",
   "documentSignature": "<document digital signature>",
   "documentBlob": "<base64 encoded document payload>",
   "documentStatus": "<approve/reject status>",
   "digsig": "<digital signature of transaction>"
}
```

Fig. 4: Document approve message in Lekana application

the message type based on the `messageType` field and routes the message to corresponding smart contract service via Kafka. Then smart contract consumes the transaction, executes it and returns the response to the client side for execution.

3.4 Smart Contract Service Communication

In some scenarios, one smart contract needs to communicate with another smart contract. For example, when document approves `DocumentActor`(resides in document service) needs to communicate with the smart actor on `DeviceActor`(resides in devices service) to send push notifications. In this case, `DocumentActor` in document service creates notification messages, shown in Figure 6 and publishes Kafka topic on device service. Then device service picks it up and sends the push notification to the given client.

3.5 Concurrent Transaction Execution

When executing transactions, traditional blockchain use order-execute architecture. They execute transactions sequentially(one after one). In Mystiko with validate-execute-group architecture, it can execute transactions concurrently. For

```
{
   "id": "<transaction id>",
   "execer": "<transaction executing user>",
   "messageType": "create",
   "documentId": "<document id>",
   "documentCreator": "<document creating user>",
   "documentName": "<document name>",
   "documentCompany": "<document own company>",
   "documentType": "<document type>",
   "documentBlob": "<base64 encoded document payload>",
   "documentSigners": "<signinig user details>",
   "digsig": "<digital signature of transaction>"
}
```

Fig. 5: Document create message in Lekana application

```
{
   "id": "<transaction id>",
   "execer": "<transaction executing user>",
   "messageType": "notify",
   "notifyDevice": "<notifying device>",
   "notifyMessage": "<notification message>",
   "digsig": "<digital signature of transaction>"
}
```

Fig. 6: Notify message in Lekana application

example, consider a scenario where concurrent transactions come to create an account(transaction A) and create a document(transaction B). These contracts are not interrelated. In the transitional blockchain, these transactions will be executed with the order they created, for example 'transaction A' after 'transaction B'. All these transactions will be executed in a single smart contract service. With Aplos SaaS architecture, these transactions will be executed concurrently in different smart contract services. 'transaction A' will be on account service and 'transaction B' will be on document service. Unlike other blockchains, Mystiko blockchain executes transactions only one time. After executing the transaction, the asset will be updated and the result will be shared on other nodes based on the process of sharding.

3.6 Scalability and Load Balancing

With SaaS, multiple replicas of smart actor services can be run in the cluster. For example, we can run multiple replicas of document actor service on the cluster as shown in Figure 7. Since Mystiko blockchain executes transactions only one-time, multiple replicas can be run parallelly. These replicas connected on a Kafka consumer group, via partitioned Kafka topic. Then Kafka handles the message partitioning and message broadcasting between the smart contract services(load balancing), guaranteeing total order(provide total order by sending a message

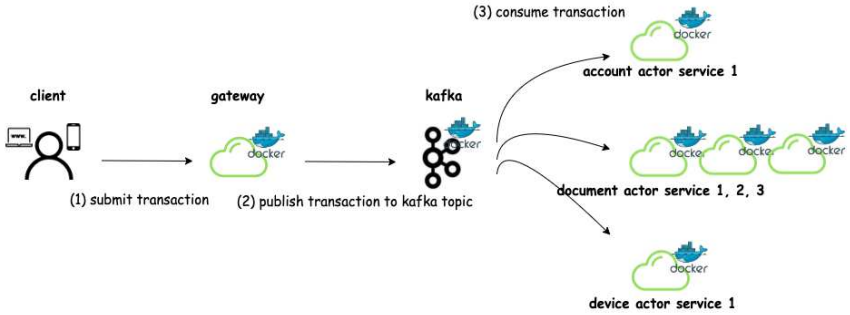

Fig. 7: Multiple replicas of Document smart actor service run parallelly in Lekana application.

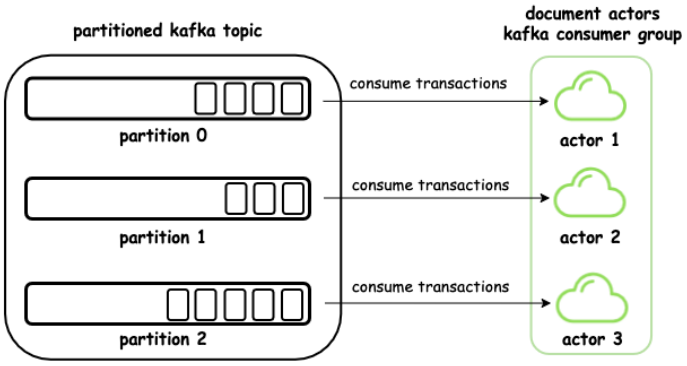

Fig. 8: Document actor connected as Kafka consumer group. Kafka will handle message broadcasting between multiple Document actor services with guaranteeing total order.

only to one consumer by topic partitioning 8. Mystiko blockchain supports the deployment of its services via Kubernetes. We can have dynamic load balancing with smart actors services via Kubernetes as well. For example, depending on the load of the system, we can dynamically increase the instance of the document service via Kubernetes. Then Routing will be handled with Kubernetes. With this approach, we have added enterprise distributed systems features into the blockchain.

4 Performance Evaluation

The evaluation of the proposed smart contract architecture has been done in the following five areas.

1. Performance of Invoke(write) transactions

2. Performance of Query(read) transactions
3. Scalability performance
4. Transaction latency
5. Transaction execution rate

These results have been obtained from the Lekana document approving application. To get the statistics, we have built two versions of the Lekana application with different smart contract architectures on Mystiko blockchain. The first version builds with default monolithic architecture which all smart contracts run on single services. The second version built with SaaS architecture where smart contracts run on different services. The performance results obtained on both these Lekana versions and evaluated.

4.1 Performance of Invoke transactions

Invoke transactions write state update to the ledger. In this evaluation, we have executed concurrent Invoke transactions in different blockchain peers and recorded all the completed transactions in each second. As shown in Figure 9 we compared the Invoke transaction on both monolithic smart contract platform and SaaS-based platform. The SaaS-based platform provides high invoke transaction throughout. In SaaS, different smart contracts can run independently/parallelly, so it increased the transaction throughput.

4.2 Performance of Query transactions

Query transactions just read the state from the ledger. In this evaluation, we have executed concurrent Query transactions in different blockchain peers and recorded the completed transactions(per second), Figure 10. As same as Invoke transactions, Query transactions throughput is high on SaaS-based implementation since it can run smart contracts parallelly. Query transactions do not update the ledger status while invoke transactions update the ledger state. Due to this reason, the throughput is higher than Invoke transaction throughput.

4.3 Scalability Performance

The scalability performance obtained against the number of executed invoke transactions in second and the total number of peers in the network. The scalability results have been recorded up to 7 blockchain peers. As shown in Figure 11 we have compared the scalability of monolithic smart contract architecture with SaaS architecture. Both architectures increase transaction throughput when increasing the number of peers in the network. But due to concurrent transaction execution on SaaS, it produces high scalability when compared with the monolithic system.

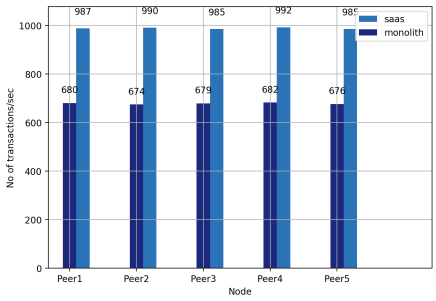

Fig. 9: Invoke transaction throughput of monolith smart contract service and SaaS based service.

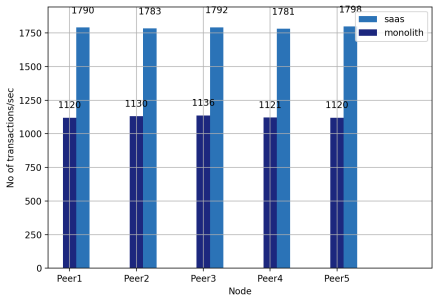

Fig. 10: Query transaction throughput of monolith smart contract service and SaaS based service.

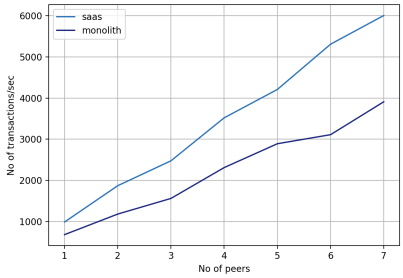

Fig. 11: Transaction scalability of monolith smart contract service and SaaS based service.

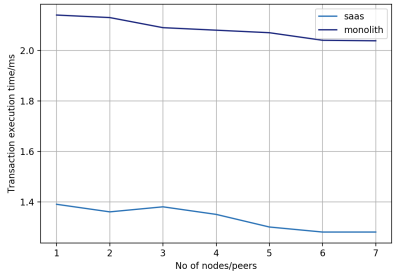

Fig. 12: Transaction latency of monolith smart contract service and SaaS based service.

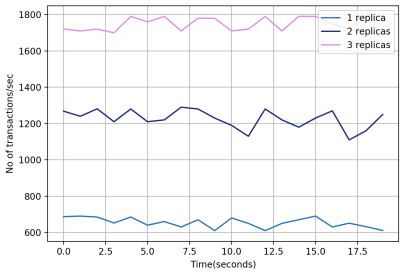

Fig. 13: Transaction execution rate comparison with number of smart contract service replicas.

4.4 Transaction Latency

The latency results obtained with the invoke transaction latency(in milliseconds) and the number of blockchain peers in the network. Concurrency invoke transac-

tions have been executed in different blockchain peers and calculated the average transaction latency. As shown in Figure 12 we have compared the latency results of monolithic smart contract services with SaaS-based smart contract service. The SaaS-based system produces less latency due to high transaction throughput on it.

4.5 Transaction Execution Rate

Finally, we evaluate the transaction execution rate with the number of smart contract service replicas. We run multiple smart contract service replicas of Document Actor service in Lekana platform and record the total number of executed transactions when recording the time. Figure 13 shows how transaction execution rate varies when having a different number of smart contract replicas. When the number of replicas increases, the rate of executed transactions is increased relatively. There is a back-pressure operation [14] between the rates of submitted transactions and executed transactions. We have used a reactive streaming-based approach with Apache Kafka to handle these back-pressure operations.

5 Related Work

Recent effort has been taken to address the issues and improve the performance of existing Smart contract platforms [6, 19]. A large portion of the current smart contracts are using the imperative style programming which has side effects. In this manner, concurrent transactions are not upheld in the blockchain. Simplicity [30], Scilla [33] and Pact [32] smart contracts are constructed dependent on functional programming semantics. Their fundamental objective is to have side-effects less blockchain functions by utilizing a functional programming based methodology. Simplicity is a composed, combinator-based, smart contract language. It is intended to function as Turing complete without the existence of loops and recursion, to be utilized for cryptographic forms of money and blockchain applications. By doing so, it intends to enhance existing cryptocurrency dialects such as the Script in Bitcoin. It keeps away from the shared global state, where the transaction doesn't have to access any data not related to the transaction. It additionally doesn't uphold communication contracts, that implies contracts don't talk with one another. Scilla [33] is a smart contract language for verified contracts which are designed mainly considering smart contract security. Scilla oversees the read and write operations to the shared address space and is intended for the implementation of the account-based model, in which smart contracts exchange messages with one another. Scilla isn't completely implemented as a functional language since the transactions could affect the outer state. Basically, Scilla utilizes recursive functions, which can be demonstrated statically. Pact [32] is another smart contract programming language which is essentially focused for private blockchain. Pact adopts the Turing incomplete semantic and the recursion causes a quick termination at each module load. But supports looping via map and fold. An advantage of this limitation is

that Pact doesn't have to utilize any sort of cost model such as the Ethereum's "gas" [27] to restrict calculation. The functional programming configuration and module definitions are adopted, as well as the atomic transaction executions. The Pack smart contracts are stored in the blockchain ledger itself as an unmodified human-readable form.

Rholang [15] is an alternate kind of smart contract platform which is mostly intended for concurrent applications. It is intended to implement smart contracts on top of general-purpose blockchain platforms. The Rho virtual machine (RhoVM) used to execute the compiled Rholang smart contract programs. It concedes unbounded recursion, which is a behaviorally typed and Turing complete programming language. Rholang smart contract language is designed to handle concurrent transactions with using message-passing through channels. This concurrent design modelled based on pi-calculus [11]-based semantic.

Solidity [12] and Chaincode [7] are two general-purpose smart contract platforms. Solidity is the most mainstream smart contract platform currently available. It is Turing complete and looks similar to Javascript, which underpins Turing schematic and the complex business logic can be actualized in smart contracts, simultaneously faced with several weaknesses. To forestall infinite loops and address the halting problem, execution is restricted by "gas", which is paid for in Ether, to the miner. At the point when the gas is run out, the exchange is invalidated however the gas is as yet paid in order to guarantee they are made up for the calculation endeavours. Solidity utilizes shared status and doesn't uphold concurrent transaction execution. Hyperledger Chaincode characterizes resources on the blockchain and the capacities to create, update, get resources from the blockchain record are actualized as contract functions, following the imperative style programming and Turing complete smart contracts. Hyperledger isn't proposed to be public blockchain, it is a private blockchain. Smart contracts won't be transferred to the blockchain by any single participant. To forestall possible weaknesses, engineers and inner groups should completely test smart contracts before use.

Table 1 compares the features of these smart contracts platforms with Aplos/SaaS smart contract platform. It discusses Turing completeness of the smart contract language, loop/recursion support of the smart contract language, functional/imperative style of the smart contract language, concurrent transaction execution support, communication contract support and smart contract implemented language details.

6 Conclusions and Future Work

SaaS introduced microservices-based highly scalable smart actor architecture for blockchain. Instead of having a single monolithic smart contract service, SaaS can run independent smart contracts on separate services as actors. These smart contracts which are introduced as smart actors can execute transactions independently. It will increase the scalability and produce high transaction throughput on the blockchain.

Table 1: Comparison of existing smart contract platforms with the Aplos/SaaS smart actor platform.

Platform	Blockchain	Public/Private	Turing complete	Loops	Functional	Concurrent Transactions	Shared State	Communication Contracts	Implemented Language
Aplos/SaaS	Mystiko	Private	Yes	Yes	Yes	Yes	No	Yes	Scala
Solidity [12]	Etherium	Both	Yes	Yes	No	No	Yes	Yes	C++/Solidity
Chaincode [7]	Hyperledger	Private	Yes	Yes	No	No	Yes	Yes	Golang
Simplicity [30]	Bitcoin	Public	No	No	Yes	No	Yes	No	Tcl/Haskell
Scilla [33]	Zilliqa	Public	No	Recursion	No	No	Yes	Yes	OCaml
Pact [32]	Kadena	Both	No	No	Yes	No	Yes	Yes	Haskell
Rholang [15]	Rchain	Both	Yes	Yes	Yes	Yes	No	Yes	Java

The evaluation has proven the scalability and transaction throughput of SaaS smart contract architecture. We have integrated SaaS-based smart contracts into production-grade blockchain applications(e.g Lekana). This deployment is a vote of confidence for SaaS as an ideal smart contract architecture to build scalable blockchain applications. Currently, we have integrated SaaS architecture into Mystiko blockchain. We are planning to incorporate secure multiparty computation [37] of Aplos framework in a future release.

References

1. Akka documentation, `https://doc.akka.io/docs/akka/2.5/actors.html`
2. Akka streams documentation, `https://doc.akka.io/docs/akka/2.5/stream/`
3. How do i accomplish lightweight transactions with linearizable consistency?, `https://docs.datastax.com/en/cassandra/3.0/cassandra/dml/dmlLtwtTransactions.html`
4. The scala programming language, `https://www.scala-lang.org/`
5. Aderaldo, C.M., Mendonça, N.C., Pahl, C., Jamshidi, P.: Benchmark requirements for microservices architecture research. In: 2017 IEEE/ACM 1st International Workshop on Establishing the Community-Wide Infrastructure for Architecture-Based Software Engineering (ECASE). pp. 8–13. IEEE (2017)
6. Adrian, O.R.: The blockchain, today and tomorrow. In: 2018 20th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC). pp. 458–462. IEEE (2018)
7. Androulaki, E., Barger, A., Bortnikov, V., Cachin, C., Christidis, K., De Caro, A., Enyeart, D., Ferris, C., Laventman, G., Manevich, Y., et al.: Hyperledger fabric: a distributed operating system for permissioned blockchains. In: Proceedings of the Thirteenth EuroSys Conference. p. 30. ACM (2018)
8. Bandara, E., NG, W.K., De Zoysa, K., Ranasinghe, N.: Aplos: Smart contracts made smart. BlockSys'2019 (2019)
9. Białecki, A., Muir, R., Ingersoll, G., Imagination, L.: Apache lucene 4. In: SIGIR 2012 workshop on open source information retrieval. p. 17 (2012)
10. Burns, B., Grant, B., Oppenheimer, D., Brewer, E., Wilkes, J.: Borg, omega, and kubernetes. Queue **14**(1), 70–93 (2016)
11. Cristescu, I.D., Krivine, J., Varacca, D.: Rigid families for ccs and the pi-calculus. In: International Colloquium on Theoretical Aspects of Computing. pp. 223–240. Springer (2015)

12. Dannen, C.: Introducing Ethereum and solidity, vol. 1. Springer (2017)
13. Davis, A.L.: Akka streams. In: Reactive Streams in Java, pp. 57–70. Springer (2019)
14. Destounis, A., Paschos, G.S., Koutsopoulos, I.: Streaming big data meets backpressure in distributed network computation. In: IEEE INFOCOM 2016-The 35th Annual IEEE International Conference on Computer Communications. pp. 1–9. IEEE (2016)
15. Eykholt, E., Meredith, L.G., Denman, J.: Rchain architecture documentation (2017)
16. Fernandes, J.L., Lopes, I.C., Rodrigues, J.J., Ullah, S.: Performance evaluation of restful web services and amqp protocol. In: 2013 Fifth International Conference on Ubiquitous and Future Networks (ICUFN). pp. 810–815. IEEE (2013)
17. Gormley, C., Tong, Z.: Elasticsearch: the definitive guide: a distributed real-time search and analytics engine. " O'Reilly Media, Inc." (2015)
18. Gupta, M.: Akka essentials. Packt Publishing Ltd (2012)
19. Harz, D., Knottenbelt, W.: Towards safer smart contracts: A survey of languages and verification methods (09 2018)
20. Hewitt, C.: Actor model of computation: scalable robust information systems. arXiv preprint arXiv:1008.1459 (2010)
21. Hoare, C.A.R.: Communicating sequential processes. Communications of the ACM **21**(8), 666–677 (1978)
22. Hughes, J.: Why functional programming matters. The computer journal **32**(2), 98–107 (1989)
23. Kreps, J., Narkhede, N., Rao, J., et al.: Kafka: A distributed messaging system for log processing. In: Proceedings of the NetDB. pp. 1–7 (2011)
24. Kurath, A.: Analyzing Serializability of Cassandra Applications. Ph.D. thesis, Master's thesis. ETH Zürich (2017)
25. Kwon, J.: Tendermint: Consensus without mining. Draft v. 0.6, fall **1**, 11 (2014)
26. Lakshman, A., Malik, P.: Cassandra: a decentralized structured storage system. ACM SIGOPS Operating Systems Review **44**(2), 35–40 (2010)
27. Marescotti, M., Blicha, M., Hyvärinen, A.E., Asadi, S., Sharygina, N.: Computing exact worst-case gas consumption for smart contracts. In: International Symposium on Leveraging Applications of Formal Methods. pp. 450–465. Springer (2018)
28. Merkel, D.: Docker: lightweight linux containers for consistent development and deployment. Linux journal **2014**(239), 2 (2014)
29. Nakamoto, S.: Bitcoin: A peer-to-peer electronic cash system (2008)
30. O'Connor, R.: Simplicity: A new language for blockchains. In: Proceedings of the 2017 Workshop on Programming Languages and Analysis for Security. pp. 107–120. ACM (2017)
31. Odersky, M., Altherr, P., Cremet, V., Emir, B., Maneth, S., Micheloud, S., Mihaylov, N., Schinz, M., Stenman, E., Zenger, M.: An overview of the scala programming language. Tech. rep. (2004)
32. Popejoy, S.: The pact smart contract language. June-2017.[Online]. Available: http://kadena. io/docs/Kadena-PactWhitepaper. pdf (2016)
33. Sergey, I., Kumar, A., Hobor, A.: Scilla: a smart contract intermediate-level language (01 2018)
34. Thönes, J.: Microservices. IEEE software **32**(1), 116–116 (2015)
35. Traiger, I.L., Gray, J., Galtieri, C.A., Lindsay, B.G.: Transactions and consistency in distributed database systems. ACM Transactions on Database Systems (TODS) **7**(3), 323–342 (1982)
36. Zamani, M., Movahedi, M., Raykova, M.: Rapidchain: A fast blockchain protocol via full sharding. IACR Cryptology ePrint Archive **2018**, 460 (2018)

37. Zyskind, G., Nathan, O., Pentland, A.: Enigma: Decentralized computation platform with guaranteed privacy. arXiv preprint arXiv:1506.03471 (2015)