Aplos: Smart Contracts Made Smart

Eranga Bandara[1], Wee Keong Ng[1], Nalin Ranasinghe[2], and Kasun De Zoysa[2]

[1] School of Computer Science and Engineering
Nanyang Technological University, Singapore
{eranga,awkng}@ntu.edu.sg
[2] University of Colombo School of Computing, Sri Lanka
{dnr,kasun}@ucsc.cmb.ac.lk

Abstract. Smart contract is a programming interface to interact with the underlying blockchain storage models. It is a database abstraction layer for blockchain. Existing smart contract platforms follow the imperative style programming model since states are shared. As a result, there is no concurrency control mechanism when executing transactions, resulting in considerable latency and hindering scalability. To address performance and scalability issues of existing smart contract platforms, we design a new smart contract platform called "Aplos" based on the Scala functional programming language and Akka actors. In Aplos, all blockchain-related smart contract functions are implemented with Akka actors. The Aplos platform is built over Mystiko—a highly scalable blockchain storage for big data. Mystiko supports concurrent transactions, high transaction throughput, data analytics and machine learning. With Aplos smart contracts over Mystiko, we have developed a blockchain for highly scalable storage that aligns with big data requirements.

Keywords: Blockchain ; Smart Contract ; Functional Programming ; Actor Model ; Big Data ; Scala ; Akka

1 Introduction

1.1 Blockchian

Blockchain stores chronological sequence of transactions in a tamper-evident manner. Each node in the blockchain has the exact same order of data. Since blockchain is like a distributed storage, it uses a consensus algorithm to order and maintain data consistency among nodes. Due to the decentralized trust ecosystem in blockchain, various industries have adopted blockchain for their applications.

Currently, there are various blockchain platforms in the market: Bitcoin [26], Ethereum [7], Bigchaindb [24], Hyperledger [4] are some examples. Ethereum and Hyperledger went beyond crypto-currencies to support different kind of asset storage models that relate to various forms of business or e-commerce activities. They introduced a new concept to blockchain called smart contracts.

Fig. 1: Smart contract overview.

1.2 Smart Contract

Smart contract imposes an additional software layer between clients and blockchain storage (Figure 1). Client requests are directed to scripts (smart contracts) that perform the logic needed to provide a complex service, such as managing state, enforcing governance, or checking credentials. Using smart contracts, users do not need to execute queries to save or retrieve data from blockchain storage. Instead smart contracts provide a programming interface to interact with the underlying blockchain storage models.

Smart contact is like a database abstraction layer for blockchain. It is similar to the Object Relational Mapping (ORM) tools in traditional programming frameworks. Unlike traditional ORMs, smart contract is capable of defining the business logic of an application. With smart contracts, business logic on the application layer can be moved to the blockchain layer. There are various smart contact platforms: Ethereum has the Solidity [34], Hyperledger fabric has the Chaincode [4], Kadena has Pact [29], RChain has Rholang [12], etc. Most of these platforms follow the imperative programming style with a shared memory model. There is no concurrency-control mechanism; they do not support concurrent execution of transactions. As a result, there is considerable latency and scalability suffers.

1.3 Aplos Smart Contract

To address issues on imperative style smart contract, we introduce the Aplos smart contract platform. This smart contract platform is built on the Mystiko blockchain, which is a highly Scalable blockchain targeted for big data [6]. Smart contract on the Aplos platform is written in the Scala functional programming [28,32] language based on Akka actors [2], so we have introduced this smart contract platform as a smart actor platform. The Actor model comes with message passing concurrency control [14, 15]. We have introduced a functional programming model instead of an imperative programming. All transactions in Mystiko are executed using Akka actors. With Akka actors and functional programming-based concurrency control, the Aplos platform enables concurrent transaction execution, which leads to high transaction throughput for blockchain.

1.4 Paper Outline

The rest of the paper is organized as follows. Section 2 discusses the Mystiko blockchain, features, characteristics and architecture. Section 3 discusses the architecture of the Aplos smart actor platform on Mystiko blockchain. Section 4 discusses examples using the Aplos smart actors in a banking application built on top of Mystiko. Section 5 performs evaluation of the Aplos smart actor platform, in comparison to the Hyperledger fabric. Section 6 surveys related work. Section 7 concludes the Aplos platform with suggestions for future work.

2 Mystiko

2.1 Mystiko Overview

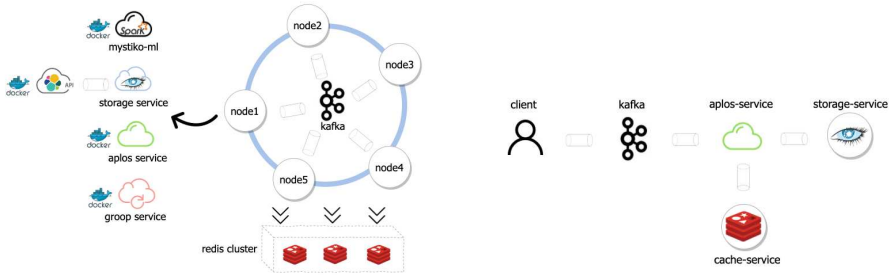
Mystiko is a highly Scalable blockchain system that utilizes the Apache Cassandra [20] distributed database (with Paxos consensus [21] as the underlying consensus platform). Mystiko uses the Apache Kafka and Akka streams [3] to handle back-pressure operations on big data. To facilitate full text search on blockchain data, Mystiko utilizes the Apache Lucene [22] based Elasticsearch [11]. It integrated with Apache Spark [25] based Mystiko-ML service to facilitate data analytics and machine learning. Mystiko addressed three main performance bottlenecks on existing blockchain platforms, namely, the Order-Execute architecture, full node data replication and imperative style smart contracts.

To address issues on the traditional Order-Execute blockchain architecture [4], Mystiko provides Validate-Execute-Group architecture [6]. This architecture allows one to validate and execute transactions whenever a client submits a transaction to the network. The client does not need to wait until a block has been created to commit the transaction. This new architecture provides high scalability and high transaction throughput.

All blocks, transactions, and asset information are stored in Cassandra database tables in Mystiko. Since it uses Cassandra for asset storage, Mystiko can stores larger data payloads with the assets. In Mystiko, every blockchain peer comes with a Cassandra node; these nodes are connected to one another in a ring cluster architecture. After executing a transaction, state update in a peer is distributed and replicated using sharding with Cassandra's Paxos consensus algorithm. In this way, Mystiko avoids the full node replication issue in conventional blockchains [35].

2.2 Mystiko Architecture

Most current blockchain systems are built as monolithic systems. A single program/service on the blockchain handles all the features in the blockchain. This includes handling consensus, maintaining the decentralized ledger, broadcasting transactions, checking double spends [26], etc. It is not an ideal design for a distributed system environment. In a monolithic system approach, one needs to build everything using a single programming language. When the codebase



(a) Mystiko microservices architecture. (b) Mystiko aplos service architecture.

Fig. 2: Mystiko blockchain architecture.

grows, it becomes unwieldy. Since only one service is available, it is not possible to scale. As such, Mystiko built using a microservice-based distributed architecture [31], solving all the aforementioned problems (Figure 2(a)). In Mystiko, all the functionalities are implemented as small services (microservices). Different programming languages and enterprise-level distributed tools are used to build the services of Mystiko. These services are dockerized [10] and available for deployment using Kubernetes [18]. Figure 2(a) shows the architecture of Mystiko. It contains the following services/components:

1. Storage service (Cassandra-based block, transaction and asset storage)
2. Cache service (Redis [30] based cache service)
3. Aplos service (Smart actor service implemented using Scala and Akka)
4. Grooper service (Block creating service implemented using Scala and Akka)
5. Kafka (Message broker)
6. Mystiko-ML (Apache spark based machine learning service)

Storage is the place where the blocks, transactions and assets are stored in Mystiko. The order of the data will be decided by Cassandra's Paxos consensus algorithm. By default Cassandra does not provide serial consistency, it only provides eventual consistency. Cassandra introduced LWT [8] to achieve serial consistency, but it consumes lots of resources and time [19]. As an alternative to Cassandra LWT, Mystiko build a Redis cache-based system to achieve serial consistency. Aplos is the smart contract service on Mystiko blockchain. It is the core part that provides high scalability and transaction throughput for Mystiko blockchain. Grooper service is responsible for creating blocks. When creating a new block, it generates a block hash from the Merkle root hash and the previous block hash. Apache Kafka [17] is used as the message broker of the Mystiko blockchain. There are two main communication use cases of Kafka in Mystiko. First, by using Kafka for client-to-blockchain communication, Mystiko is able to handle back pressure operation in big data environments. It handles all the transaction messages that come to Kafka using Akka streams. Second, when generating blocks, Groopers communicates with one another to validate and

approve the blocks. Each peer in the blockchain network has their own storage, Aplos and Grooper services.

Mystiko-ML is the Apache Spark based machine learning and analytic service on Mystiko blockchain. It supports to do analytic on both on-chain, off-chain storage and build the supervised or unsupervised machine learning models. These models can be used to do the predictions of real-time data.

3 Aplos Smart Actors

3.1 Overview

Aplos is the smart contract platform in Mystiko blockchain. All the peers in the network have their own Aplos service which run as a docker container. The business logic of blockchain applications (e.g., asset creation, validation, authorization) are written using the Scala functional programming based Akka actors. All transactions on the Mystiko blockchain executed using these smart actors. Actors consume messages. Based on the message they execute various business logic. By using Akka actors based smart contract, Mystiko supports concurrent transaction execution.

To create/update/search assets on the blockchain, clients needs to send transaction messages to the Aplos service with actor name, message type and the transaction attributes. Based on the actor name, the Aplos service finds the smart actor that needs to be invoked. It then passes the message to that actor. When transaction message arrives, the actor validates (check double spend and digital signature) and executes the transaction. Based on the execution outcome, it inserts transaction record into Mystiko storage transaction table and updates asset status in the corresponding asset table on Mystiko storage. The Aplos service interacts with Redis cache to validate the transaction and with Storage service to update asset status (Figure 2(b)).

3.2 Transaction Messages

Aplos service consumes transaction messages from Apache Kafka(Figure 2(b)). There is a Kafka topic which the service listens to. When submitting transaction to the Aplos service, the client publishes JSON encoded messages to the Kafka topic. Figure 6(b) shows an example of a transaction message in the following mentioned (Section 4) Promize application money transfer scenario. It defines the actor name, message type and other transaction attributes (account information, amount). When a message is received by the Aplos service, it creates a scala case class [28] based on the message type and delegates it to the corresponding actor. Based on transaction parameters, validation phase and execution phases are performed by the actor. Finally, the transaction response will be returned to the client.

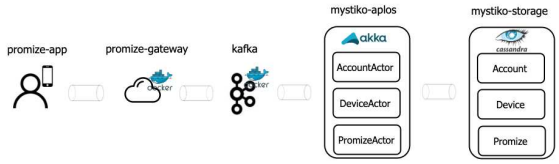


Fig. 3: Promize application architecture.

3.3 Concurrent Transactions

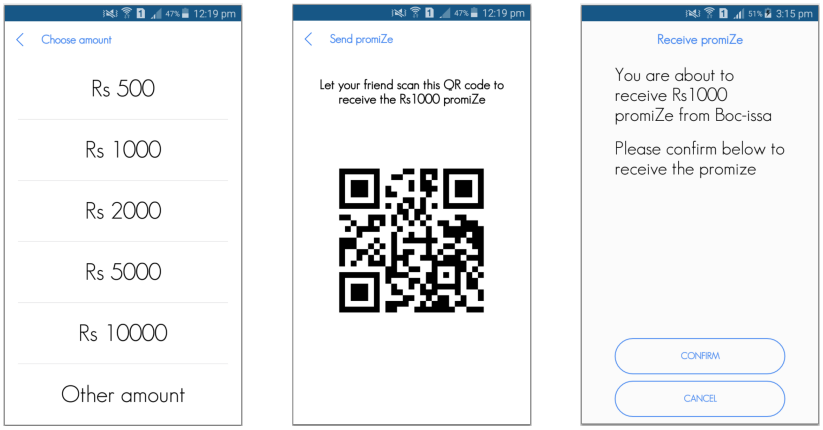
Smart actors communicate with one another using message passing. Since there is no shared state among actors, they are able to run concurrently. Akka actors come with two communication patterns `Ask` and `Tell` [2]. `Ask` is a blocking operation: When an actor A sends message to B actor by using `Ask`, actor A waits until actor B responds to the message. `Tell` is a non-blocking (fire and forget). When an actor A sends a `Tell` message to actor B, A does not need to wait until B responds to the message; actor A continues its future operations. In Aplos, we use the non-blocking `Tell` message pattern. When communicating between actors, they are message passing each other and doing the operations.

4 Promize Application

4.1 Promize Overview

We have built Promize, a peer-to-peer money transfer application for MBSL Bank, Sri Lanka [23] using the Mystiko blockchain with Aplos smart actor platform. We are introducing this application as an alternative to traditional ATMs (Mobile ATM [16]). With the Promize application, users may take money from registered authorities or their friends without going to an ATM. The architecture of the Promize service is described in Figure 3. Users are given a Promize mobile application to do money transfer transactions. The mobile app sends requests to the smart actors on Mystiko blockchain when doing Promize transactions.

First, users register with the Promize service via the Promize mobile application. When a user registers, the mobile application sends a request to `AccountActor`(Figure 5(a)) on Mystiko blockchain with credentials/account information. Upon account create request, `AccountActor` validates the user credential/account information and creates an account for the user in the Mystiko blockchain. Assume that two users (A and B) are registered on the Promize application. User A wants to take 1,000 rupees. User A asks User B (who already have Promize app installed on his phone) for 1,000 Rupees. User B starts the Promize app and chooses the transferring amount (Figure 4(a)). This generates a QR code with embedding B's account number and transferring amount(Figure 4(b)). This QR code will be scanned by user A and submit the Promize transaction. (Figure 4(c)).



(a) Choose promize amount. (b) Promize QR code. (c) Scan promize QR code.

Fig. 4: Promize application.

When submitting the transaction, it sends Promize initializing request to `PromizeActor`(Figure 5(b)) on the Mystiko blockchain, which checks the validity of the transaction and user accounts. If transaction is valid, a confirmation request with random number(transaction salt) will be sent to user B's Promize mobile application via push notification. Push notifications will be sent by `DeviceActor` on Mystiko blockchain. Then user B confirms this request in order to approve the Promize transaction. When confirming, Promize approve request will be sent with received random number to `PromizeActor` on the Mystiko blockchain, which checks the validity of the transaction parameters (transaction salt and accounts). If transaction parameters are valid, `PromizeActor` transfers requested money(Rs 1000) from user A's account to B's account. Finally transaction confirmation status will be sent to both User A and B. Then User B physically gives 1,000 rupees to user B.

The idea here is user A physically takes money from user B which the bank electronically transfers money from A's account to B's account via Promize transactions. The Promize service guarantees Non-repudiation, Confidentiality, Integrity, Authenticity and Availability of electronic transactions [16] using the blockchain.

4.2 Smart Actors

There are three main smart actors in this system: `AccountActor`, `PromizeActor`, and `DeviceActor`. The `AccountActor` corresponds to account creation and activation function. It consumes `Create` and `Activate` messages (Figure 5(a)). The `DeviceActor` handles push notification sending functions. It consumes `Create` and `Notify` messages. The `PromizeActor` handles Promize transactions. It con-

```
import akka.actor.{Actor, Props}
import com.score.aplos.paper.AccountActor.{Activate, Create}

object AccountActor {
  case class Create(messageType: String, execer: String, id: String,
                    accountId: String, accountPassword: String,
                    accountPhone: String, accountEmail: String,
                    deviceToken: String, deviceType: String)

  case class Activate(messageType: String, execer: String, id: String,
                      accountId: String, accountSalt: String)

  def props() = Props(new AccountActor)
}

class AccountActor extends Actor {
  override def receive: Receive = {
    case create: Create =>
      // verify digital signature
      // check double spend
      // create account on ledger

      // send message to device contract to create device
      val msg = DeviceActor.Create("create", create.execer, create.id,
        create.accountId, create.deviceToken, create.deviceType,
        create.accountId)
      context.actorOf(DeviceActor.props()) ! msg

      // send status back
    case Activate =>
      // verify digital signature
      // check double spend
      // update account on ledger
      // send status back
  }
}
```

(a) Account smart actor.

```
import akka.actor.{Actor, Props}
import com.score.aplos.paper.DeviceActor.Notify
import com.score.aplos.paper.PromizeContract.{Approve, Create}

object PromizeContract {
  case class Create(messageType: String, execer: String, id: String,
                    promizeId: String, promizeFrom: String,
                    promizeTo: String, promizeAmount: String)

  case class Approve(messageType: String, execer: String, id: String,
                     promizeId: String, promizeFrom: String,
                     promizeTo: String, promizeSalt: String)

  def props() = Props(new PromizeContract)
}

class PromizeContract extends Actor {
  override def receive: Receive = {
    case create: Create =>
      // verify digital signature
      // check double spend
      // create promize on ledger

      // send message to device contract to notify from account
      val salt = "<transaction salt>"
      val msg = Notify("notify", create.execer, create.id,
        create.promizeFrom, salt)
      context.actorOf(DeviceActor.props()) ! msg

      // send status back
    case Approve =>
      // verify digital signature
      // check double spend
      // update promize
      // send status back
  }
}
```

(b) Promize smart actor.

Fig. 5: Smart actors.

sumes `Create` and `Approve` messages, corresponding to create and approve Promize transactions (Figure 5(b)).

4.3 Transaction Messages

When performing transactions, a client sends transaction messages to smart actors on the Mystiko blockchain. The message contains JSON encoded string with smart actor name, message type and message parameters. Figure 6(a) shows the message sent by user when creating accounts. It contains account information. Figure 6(b) shows the message that corresponds to Promize transaction. It contains accounts and amount information.

When this messages arrives at the Aplos service, it first identifies the actor name and the message type. Then it creates a scala case class message with parameters and passes it to the corresponding actor. When actor receives this message, it validates the message (check and digital signature and message attributes) and performs the execute operation (e.g., create account and transfer money). When executing a message, it first checks for double spending. If there is no double spending, it creates a transaction and executes the transaction. When executing transaction, it create/update assets on the blockchain.

4.4 Concurrent Transaction

As mentioned earlier the Aplos platform supports concurrent transaction execution. All messages come to the Aplos service via Kafka. These messages are del-

```
{
  "id": "<transaction id>",
  "execer": "<transaction executing user>",
  "messageType": "<message type>",
  "accountId": "<account id>",
  "accountPassword": "<account password>",
  "accountPhone": "<account phone no>",
  "accountEmail": "<account email>",
  "deviceToken": "<notification token>",
  "deviceType": "<mobile device type>",
  "digsig": "<digital signature>"
}
```

(a) Account create message.

```
{
  "id": "<transaction id>",
  "execer": "<transaction executing user>",
  "messageType": "<message type>",
  "promizeId": "<promize id>",
  "promizeFrom": "<promize send account>",
  "promizeTo": "<promize receive account>",
  "promizeAmount": "<promize amount>",
  "digsig": "<digital signature>"
}
```

(b) Promize create message.

Fig. 6: Smart actor messages.

egated to the corresponding actors in an asynchronous manner (non-blocking). Aplos smart actors communicate with one another via message passing. In account creation, when `Create` message comes to `AccountActor`, it first creates account by executing create function on Account actor. Then it pass a `Create` message to `DeviceActor` to create a device. `DeviceActor` executes that message and creates the device(Figure 5(a)).

When Promize message is received by `PromizeActor`, it first creates Promize by executing `Create` on `PromizeActor`. Then it pass `Notify` message to `DeviceActor` to send push notification (Figure 5(b)). `DeviceActor` executes `Notify` message and sends push notification to corresponding user.

5 Performance Evaluation

We have done a performance evaluation of the Mystiko Aplos smart actor platform. The evaluation results are obtained for the following metrics:

1. Invoke transaction throughput
2. Query transaction throughput
3. Transaction scalability
4. Transaction latency
5. Search performance

To obtain the results, we deployed multi-peer Mystiko cluster with Aplos smart actor service and Hyperledger Fabric cluster in separate AWS 2xlarge instances (16GB RAM and 8 CPUs).

5.1 Invoke Transaction Throughput

For this evaluation, we recorded the number of invoke transactions that can be executed in each Mystiko blockchain peer. Invoke transaction creates transaction in the ledger and updates the status of the assets. We flooded invoke transactions for each blockchain peer and recorded the number of executed transactions. As shown in Figure 7, we compared the transaction throughput of Mystiko with

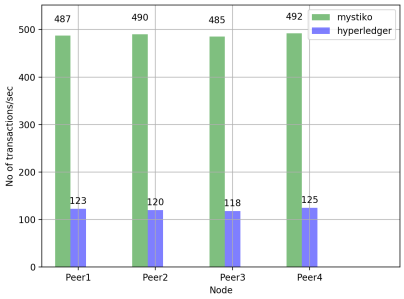


Fig. 7: Invoke transaction throughput of Mystiko and Hyperledger Fabric.

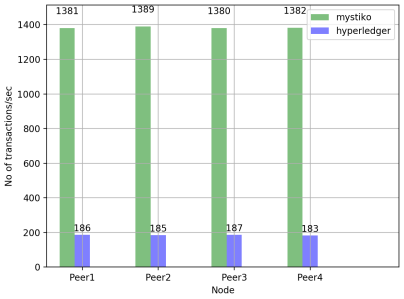


Fig. 8: Query transaction throughput of Mystiko and Hyperledger Fabric.

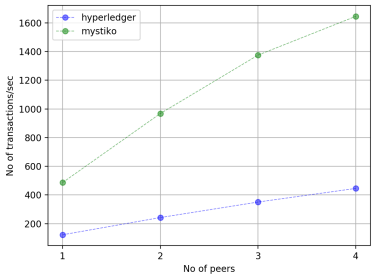


Fig. 9: Transaction scalability of Mystiko and Hyperledger Fabric.

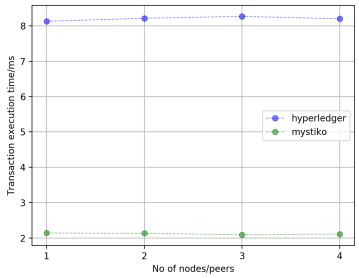


Fig. 10: Transaction latency of Mystiko and Hyperledger Fabric.

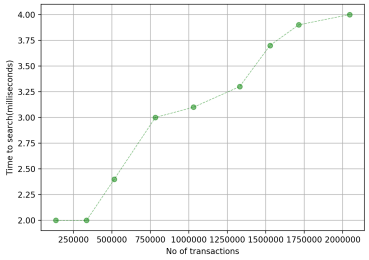


Fig. 11: Search performance of Mystiko.

Hyperledger Fabric. Functional programming-based smart actor platform and Akka streams based back pressure handling of Aplos service provides the high invoke transaction throughput for Mystiko blockchain.

5.2 Query Transaction Throughput

For this evaluation, we recorded the number of query transactions that can be executed in each Mystiko blockchain peer. Query transaction queries the status of the ledger. They neither create transaction in the ledger nor update the asset status. We flooded query transactions for each blockchain peer and recorded the number of completed transactions. As shown in Figure 8, we compared the transaction throughput of Mystiko with Hyperledger Fabric. Since query transactions are not updating the ledger status, it has high throughput compared to invoke transactions. Likewise the query transaction throughput of Mystiko is higher than Hyperledger fabric. Functional programming-based smart actor platform, Akka streams based back pressure handling, Elasticsearch based search API are the main reasons for this result [6].

5.3 Transaction Scalability

For this evaluation, we recorded the number of invoke transactions (per second) over the number of blockchain peers in the network. We flooded concurrent transactions in each blockchain peer and recorded the number of executed transactions. Figure 9 shows transaction scalability comparison of Mystiko blockchain with the Hyperledger Fabric blockchain. The main reason for high scalability in Mystiko is the underlying Paxos-based Cassandra's master-less ring architecture [20]. In Cassandra, all blockchain nodes have write capability. So when adding a node to the cluster, it linearly increases the transaction throughput. The other reason is asynchronous and concurrent transaction handling on Aplos smart actor service.

5.4 Transaction Latency

Next, we evaluate transaction latency in Mystiko. We flooded concurrent transactions in each blockchain peer and calculated the average transaction latency. Figure 10 shows the transition latency comparison of the Mystiko blockchain with Hyperledger Fabric blockchain. Cassandra's high write throughput, Redis cache-based Validate-Execute-Group architecture and functional programming-based smart actors produce less transaction latency in Mystiko [6].

5.5 Search Performance

Mystiko allows one to search data in the transaction/block/asset tables using Elasticsearch. For this evaluation, we issued concurrent transaction search queries to Mystiko and compute the search time. As shown in Figure 11, we achieved super fast search performance (few milliseconds time). The Apache Lucene index-based Elasticsearch storage and concurrent transaction execution of the Aplos service are the main reasons yielding super fast search in Mystiko.

Table 1: Smart contract platform comparison.

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

6 Related Work

Much research has been conducted to improve the performance and address the issues of smart contracts [1, 13]. Most of them followed imperative programming style and shared memory concurrency model. In this section, we outline the main features and architecture of these research projects.

Solidity (Ethereum) [34] is the most popular smart contract language today. It is a Turing complete language and resembles Javascript. As it supports Turing complete smart contracts, very complicated logic can be implemented in smart contracts, at the same opening it to vulnerabilities. To prevent infinite loops, execution is limited by a counter called "gas", which is paid for in Ethereum's unit of account, Ether, to the miner of the block containing the transaction. When a program runs out of gas, the transaction is nullified but the gas is still paid to the miner to ensure they are compensated for their computation efforts. Ethereum solidity smart contracts follows imperative style programming while Aplos contracts follows functional style. Also Solidity uses shared status and does not support concurrent transaction execution. But Aplos supports concurrent transaction executions without sharing status among Actors.

Chaincode (Hyperledger) [4] is the smart contract platform on Hyperledger Fabric which written in Golang. It defines assets on the blockchain as Golang Structs. The functions to create, update, get assets from the blockchain ledger are implemented as contract functions. Hyperledger Chaincode follows the imperative style programming model while Aplos follows the functional programming model. Both Aplos and Hyperledger Chaincode provides Turing complete smart contracts. They support loops and are vulnerable infinite looping. Both platforms not intended to be public blockchain, they are private blockchains. Smart contracts will not uploaded by any user. To prevent infinite loops and other vulnerabilities, developers and internal team must thoroughly test smart contracts before use.

Simplicity (Bitcoin) [27] is a typed, combinator-based, functional language. It is designed to work as Turing incomplete schematic without loops and recursion, to be used for crypto-currencies and blockchain applications. By using functional and Turing incomplete schematics it aims to improve upon existing crypto-currency languages, such as Bitcoin Script and Ethereum's Solidity. It avoids the shared global state, the transaction does not need to access any

information outside the transaction. It also does not support communication contracts, that means contracts do not talk to each other. The functional programming model of Aplos is similar to the Simplicity model. However, Aplos used non shared status by using Actors. Due to this reason Aplos can talk with other smart contracts in the system.

Scilla (Zilliqa) [33], is an intermediate-level smart contract language for verified smart contracts. It has been designed as a principled language with smart contract safety in mind. Scilla manages read and write to a shared memory space and is designed for an account-based model, where contracts can communicate with each other. Scilla is not fully functional smart contract language as transactions affect the external state. Scilla supports looping constructs via well-founded recursive function definitions, so their termination can be proved statically. Both Aplos and Scilla contracts are capable of interacting with other contracts. But scilla use shared memory status while Aplos uses non shared actor based model.

Pact (Kadena) [29] is new programming language which mainly targeted for private blockchain. Pack follows Turing incomplete safety oriented design. Recursion in Pact is detected and causes an immediate failure at module load. Looping is only supported using map and fold on finite list structures. A benefit of this restriction is that Pact does not need to employ any kind of cost model like Ethereum's "gas" to limit computation. Pact follows the functional programming design, supports module definitions, imports and atomic transaction executions. Pact smart contract code is stored in an unmodified, human-readable form on the ledger. Both Aplos and Pact platforms support functional programming paradigm on their smart contracts. But Aplos with actor model support non shared statuses and concurrent transactions.

Rholang (RChain) [12] is designed to be used to implement protocols and smart contracts on a general-purpose blockchain. The compiled Rholang contract is executed in a Rho virtual machine (RhoVM). It admits unbounded recursion, behaviorally typed, Turing complete concurrent programming language, with a focus on message-passing and formally modeled by the pi-calculus. The language is concurrency-oriented, with a focus on message-passing through channels. When comparing Aplos and Rholnag, one can see their main concepts are similar. Both using concurrency oriented message passing and functional programming. But in Rholang transactions that do not interact must be able to complete at the same time.

The comparison summary of these smart contract platforms and Aplos platform is presented in Table 1. It compares running blockchain platform, blockchain type (public, private), Turing completeness, loop support, concurrent contract support, functional, concurrent execution, smart contract communication and implemented language details.

7 Conclusions and Future Work

With Aplos we introduced Scala functional programming and Akka actor based smart contract platform into Mystiko blockchain. By using Akka actors to build

smart contract, Mystiko supports concurrent transaction execution. This results in high transaction throughput. With Aplos smart actors on Mystiko we have built a blockchain that has highly scalable storage aligned for big data requirements.

We have evaluated the scalability and transaction throughput with empirical evaluations. We have integrated Aplos smart actor platform with Mystiko blockchain into production grade applications in the banking and financial sectors. The deployments are votes of confidence for Aplos platform based Mystiko as an ideal blockchain system for big data and cloud storage.

Most recently we have released Mystiko version 2.0 with the Aplos actor platform. We follow the agile continuous delivery approach when building and releasing the product. The following features we will release in the future:

1. Secure multiparty computation [36] with Aplos framework.
2. Incorporate homomorphic encryption [5] to provide privacy and confidentiality features.
3. Integrate ETCD-based [9] distributed key/value pair storage to achieve fully decentralized caching.

References

1. 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)
2. Akka: Akka documentation, `https://doc.akka.io/docs/akka/2.5/actors.html`
3. Akka: Akka streams documentation, `https://doc.akka.io/docs/akka/2.5/stream/`
4. 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)
5. Armknecht, F., Boyd, C., Carr, C., Gjøsteen, K., Jäschke, A., Reuter, C.A., Strand, M.: A guide to fully homomorphic encryption. IACR Cryptology ePrint Archive **2015**, 1192 (2015)
6. Bandara, E., Ng, W.K., Zoysa, K.D., Fernando, N., Tharaka, S., Maurakirinathan, P., Jayasuriya, N.: Mystiko - blockchain meets big data. In: IEEE International Conference on Big Data, Big Data 2018, Seattle, WA, USA, December 10-13, 2018. pp. 3024–3032 (2018)
7. Buterin, V., et al.: A next-generation smart contract and decentralized application platform. white paper (2014)
8. Cassandra: Cassandra lwt, `https://docs.datastax.com/en/cassandra/3.0/cassandra/dml/dmlLtwtTransactions.html`
9. Coreos: coreos/etcd (Aug 2018), `https://github.com/coreos/etcd`
10. Docker: Docker documentation (Aug 2018), `https://docs.docker.com/`
11. Elasticsearch: Elasticsearch documentation, `https://www.elastic.co/guide/index.html`
12. Eykholt, E., Meredith, G., Denman, J.: Rchain architecture documentation (2017)

13. Harz, D., Knottenbelt, W.: Towards safer smart contracts: A survey of languages and verification methods (09 2018)
14. Hewitt, C.: Actor model of computation: scalable robust information systems. arXiv preprint arXiv:1008.1459 (2010)
15. Hoare, C.A.R.: Communicating sequential processes. Communications of the ACM **21**(8), 666–677 (1978)
16. Karunanayake, A., De Zoysa, K., Muftic, S.: Mobile atm for developing countries (01 2008). https://doi.org/10.1145/1403007.1403014
17. 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)
18. Kubernetes: Kubernetes documentation, `https://kubernetes.io/docs/home/?path=users&persona=app-developer&level=foundational`
19. Kurath, A.: Analyzing Serializability of Cassandra Applications. Ph.D. thesis, Masters thesis. ETH Zürich (2017)
20. Lakshman, A., Malik, P.: Cassandra: a decentralized structured storage system. ACM SIGOPS Operating Systems Review **44**(2), 35–40 (2010)
21. Lamport, L.: The part-time parliament. ACM Transactions on Computer Systems (TOCS) **16**(2), 133–169 (1998)
22. Lucene: Apache lucene documentation, `http://lucene.apache.org/`
23. MBSL: Mbsl bank, `https://www.mbslbank.com`
24. McConaghy, T., Marques, R., Müller, A., De Jonghe, D., McConaghy, T., McMullen, G., Henderson, R., Bellemare, S., Granzotto, A.: Bigchaindb: a scalable blockchain database. white paper, BigChainDB (2016)
25. Meng, X., Bradley, J., Yavuz, B., Sparks, E., Venkataraman, S., Liu, D., Freeman, J., Tsai, D., Amde, M., Owen, S., et al.: Mllib: Machine learning in apache spark. The Journal of Machine Learning Research **17**(1), 1235–1241 (2016)
26. Nakamoto, S.: Bitcoin: A peer-to-peer electronic cash system (2008)
27. O'Connor, R.: Simplicity: A new language for blockchains. pp. 107–120 (10 2017). https://doi.org/10.1145/3139337.3139340
28. 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)
29. Popejoy, S.: The pact smart contract language. June-2017.[Online]. Available: http://kadena. io/docs/Kadena-PactWhitepaper. pdf (2016)
30. Redis: Redis documentation, `https://redis.io/documentation`
31. Richardson, C.: Microservices pattern, `http://microservices.io/patterns/microservices.html`
32. Scala: Scala documentation, `https://docs.scala-lang.org/`
33. Sergey, I., Kumar, A., Hobor, A.: Scilla: a smart contract intermediate-level language (01 2018)
34. Solidity: Solidity documentation, `https://solidity.readthedocs.io/en/develop/`
35. Zamani, M., Movahedi, M., Raykova, M.: Rapidchain: A fast blockchain protocol via full sharding. IACR Cryptology ePrint Archive **2018**, 460 (2018)
36. Zyskind, G., Nathan, O., Pentland, A.: Enigma: Decentralized computation platform with guaranteed privacy. arXiv preprint arXiv:1506.03471 (2015)