Enabling Enterprise Blockchain AppDev Teams

Prateek Reddy Yammanuru, Ayush Jain, and Harihara Vinayakaram
Wipro Technologies

***Abstract*—Blockchain is the latest buzzword in the FinTech scene and all companies big and small are vying to launch blockchain enabled products. At the basic technology level Blockchain is a distributed technology application. The challenges of operating such an application are known [1]. But the techniques of developing distributed applications by large enterprise teams, in a typical SDLC lifecycle (Develop, Test, Deploy and Upgrade) is not well known. Without proper methodologies / Formal Tools as is the case with most blockchain systems, bugs slip in easily. Studies on failures point to developers missing low handing bugs as most of the errors are simulated with 3 nodes or less [2]. The developer ecosystem is fast changing with technologies like containers and the emerging Micro Services architectures and Cloud Native Computing. The decisions on setup, build, CI/CD, Automated Testing are not taken at the beginning and as pointed out by [3] affect the entire project. The good news is that there are lot of tools available in the Open source domain that addresses the needs. The bad news is that picking the right combination to work in team sizes of 5 or more is not straight forward. This paper details our journey and lessons learnt on setting up Application Development Teams for Rapid Development in Blockchain using multiple blockchain tools like Ethereum and the HyperLedger Fabric. It details both our application architecture and the modifications needed to enable a Cloud Native architecture and the build/ deploy/ testing frameworks that we used.**

***Index Terms*—Bitcoin, Blockchain, Micro Services, Cucumber, Gherkin, HyperLedger, Scaling Testbed, Development, Chaos Testing.**

I. Introduction

BLOCKCHAIN is a technology that is gaining traction rapidly due to its scope and ability to transform processes across industries. Many companies are trying to develop products using blockchain. But there is lack of understanding about how the network behaves in certain conditions. This is not only because it is a relatively new and complex technology, but also because Blockchain being a distributed system has a lot of challenges like data consistency, handling node failures, network conditions, availability of other nodes in the network, a distributed consensus process and so on. Many of the above situations are handled by the blockchain platform itself, but these factors affect the behavior of the application. (e.g.) Transactions in a bitcoin network are not considered complete until 7 blocks are generated. (e.g.) lack of available members for quorum in a private network. Application Developers are interested in finding out how their end users would be affected. But there is a lack of tooling around how to simulate these possible factors that affect an application built on top of blockchain. There is a need for a tool which can act as a testbed for simulating various network conditions, failure scenarios and attack modes on any blockchain platform to get a deeper understanding of the platform behavior as well as potential issues.

Chaos Testing is an important aspect in distributed systems where failures are intentionally introduced into the application to check the resiliency of the application when common failures happen. Blockchain being a distributed system is designed for common failures like node failures and adverse network conditions, but a new class of distributed failures are introduced into Blockchain as the nodes have to consensus on the ledger state, called Byzantine Faults [4] in which nodes can act erratically without node failure. The above-mentioned tool should be able to help in performing chaos tests by introducing general failures like node failures, network failures and blockchain specific Byzantine Faults for any blockchain platform to test the resiliency of the platform by simulating adverse conditions. Such a tool can also act as a testbed to see how an application behaves on different types of blockchain network (e.g.) Private vs Public.

To illustrate what are the features such a tool must provide, let us take an example of a 51% attack scenario [5] in case of Proof of Work in Ethereum with 100 nodes. To simulate this scenario, we need to setup an Ethereum network with 100 nodes. We also need a way to check that all the nodes in the network are in sync. We also need a way to generate transactions (normal and Byzantine fault – wrong transactions) that can be injected into some % of the nodes. We can now observe how the malicious attack works and how long it takes for the malicious nodes to take control of the blockchain and so on. The platform should be able to provide all these features and an interface to specify the scenario to run.

We will need the following 3 requirements for the platform: Ability to setup blockchain network easily, an interface to communicate with blockchain and an interface to measure changes happening in the blockchain network. These requirements can be transformed into the following design goals:

- *Flexibility*: Flexibility to use ***any*** blockchain platform on ***any*** infrastructure platform. (Docker, VM, Bare Metal, Cloud Provider)
- *Reactive/Event Driven API*: In blockchain any changes done will only be reflected after consensus on the change by the concerned parties in the network. An API that can provide messages about what is happening inside blockchain is necessary so that we can react to them.
- *Observability*: As we are analyzing the blockchain network in various scenarios, we need metrics and other tools that can completely capture what is happening

inside the network. We also need to be able to monitor the network, while the experiments are running.

In addition to these goals we also need a simple and clear interface to provide the specification of the scenarios to be run using the platform. With these goals in mind we have built a platform which can run the scenarios and as most of these services are needed for application development also, we have built the experiments as an application over a Framework which can be also be extended to build any application on any blockchain Platform and infrastructure. So, we added the following design goals.

- *Independent services*: As all the services for communicating with Blockchain or listeners to blockchain will be part of the application itself, the framework can be a set of services among which the application developer can choose what to use. So, services should be independent of each other. Also, independent services will ease the development of applications and replace implementation of existing services with custom implementations. What this means in practice is that it should be easy to replace Blockchain X with Blockchain implementation Y.
- *Extensibility*: As the framework is a set of services on which applications can build on. It should be easy to use the services and extend the framework for building applications over it

With these design goals, we built a framework which would help application developers answer the question how will my application behave in a given scenario. This paper details our learnings and points to the challenges faced. A lot of practices in this paper are widely known (e.g. Micro Services, Event Driven Architecture). This can also be seen as a practitioner's attempt to bring theory to practice. This can help application developers in building and testing applications on the various blockchain platforms. The framework also helps the development teams by providing the building blocks of a modern software architecture development platform namely, event driven asynchronous communication, Micro Service Infrastructure and a CI/CD process to help with rapid application development.

II. TESTBED DEVELOPMENT

We need services to be independent of each other. To achieve this goal, we chose the Framework to be a micro services system where each service is independent of each other and perform its own tasks and in view of our goal to have a Reactive API and extensibility we have used an event bus to communicate among the services so that if the framework is extended for developing some application the developers can easily extend the events from the framework to do custom logic as part of the application. Keeping in mind our goal to have a clear and simple interface, we have used a tool called swagger [6] to have a clear specification and documentation of the API for all the services. We have used Dependency Injection [7] to make the services independent of each other while development and testing in view of our goal to have independent services. So, Framework is a micro service based system where various services communicate with each other using a publish subscribe (pubsub) messaging [8] platform. Framework is a set of services that can help application development teams by acting as a bootstrap for all the blockchain related setup, interfaces to blockchain and to run sample programs on blockchain. Framework provides following 7 services

- Infrastructure service – This service helps in creating the necessary infrastructure to setup the blockchain network. Infrastructure service's service boundary extends into virtual machines created using the service, in the form of *blockchainAgent* installed by the cloud init script when starting the virtual machine which acts like a proxy to the InfraStructure service to manage the lifecycle of the blockchain nodes on that particular virtual machine. This blockchainAgent is actually responsible for creating the type of infrastructure needed like creating nodes in docker containers.
- Blockchain service – This service provides an interface for applications to communicate with Blockchain. This service provides a common REST interface for interacting with all blockchains. This service also acts as a checkpoint to control which API is to be exposed for the applications to use. This service has 2 components, first is the common interface for all Blockchain platforms which helps with common functionalities between various blockchains like reading a block or sending a transaction or creating an account to have the same interface throughout and the second part of the service is the connector which connects the interface for applications to the actual blockchain interface and also helps in listening to the blockchain and emit events when something interesting happens inside blockchain.
- Bootstrap service – This service provides the required data for a blockchain node to startup like connection details for other nodes in network and other artifacts needed for the blockchain network to start depending on the Blockchain platform selected.
- Failure Injection service – This service helps to inject failures like network failures or latencies, node failures or Byzantine failures like data corruption or dishonest node sending wrong data or malicious collusion between set of nodes.
- Dashboard service – This service takes all the data from monitoring tools, blockchain listeners and metrics and displays the data necessary to know the status of network.
- Pub-Sub service – This service acts as the event hub used for communication between various services of the framework. This also acts as an interface for using the framework with events for any application.
- Proxy service – This service provides the discovery service and API gateway service needed to scale individual services of the framework separately.

Any blockchain based applications can use the above services and extend the framework for their applications. Access to both HTTP interface and Event based interface is provided for easily integrating with the services specific to the application. We have used techniques like dependency injection to ease the use of various interfaces and

adding a new interface.

We used this Framework and built an application over it, which can simulate scenarios in blockchain network called experiment service which provides an interface to run experiments on the blockchain platform or the application like recording the behavior by sending large number transactions or by injecting a node failure or simulating network conditions or injecting Byzantine failure into the network. This service provides an interface for some predefined experiments or a cucumber based API for writing custom scenarios for running experiments by providing a set of step definitions.

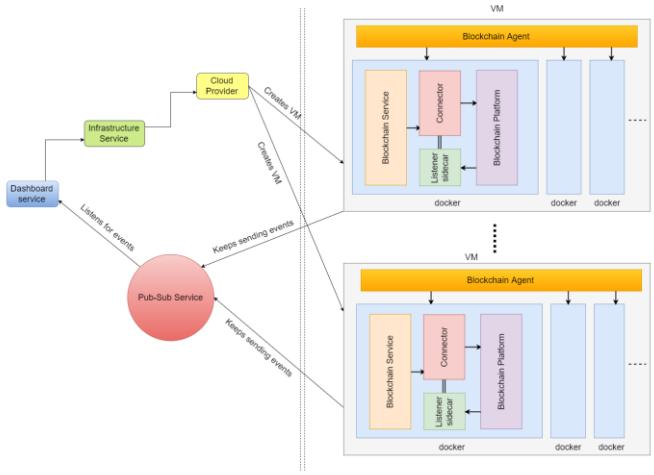

Figure 1: Architecture of the Framework

Framework for running scenarios

In an enterprise setup, software development is managed by both business and technical teams, which leads to a new type of agile development process called Behavior Driven Development [9] and there are many tools built around it and one such tool is cucumber [10]. Cucumber provides an interface that can enable business teams to write specifications in Gherkin which is a language similar to natural language (English). To achieve the goal of creating simple interface and to enable the business teams also to use framework we have used cucumber tool to creating an interface using Gherkin for running scenarios with Framework (see Appendix A) by providing a set of step definitions as building blocks to write complex scenarios. Framework can be used to test Blockchain Platforms and test their consensus models by introducing Byzantine faults like data corruption or data changed by dishonest nodes or collusion between some dishonest nodes. These tests help understand the behavior of blockchain network under specified conditions. Let us take the same example we used before to simulate 51% attack scenario in case of Proof of Work in Ethereum with 100 nodes, framework provides a cloud service which can create the 100 node Ethereum network, once this is done we have experiment service to communicate to the blockchain node, so that we can check that all the nodes are connected and in sync. Now we want to inject failure into 51% of the nodes, so we use the failure injection service to inject byzantine failures into 51 nodes. Now when the malicious nodes come back into the network we can analyze what is happening in the network using the events emitted by the listeners in blockchain service. This way Framework can be used to simulate attack modes and test the blockchain platforms against any attacks.

Framework currently supports experiments on Ethereum and support for Hyperledger fabric is in progress. An example scenario to be simulated by Hyperledger Fabric is in Appendix B.

Observability of Framework

Observability is an important property of the framework as it is one of our design goals and the goal of the testbed is to simulate conditions for the blockchain platforms and observe the behavior of the network. We have three ways in the framework which help us achieve a good observability of what is happening inside the network, one is the listeners attached to every blockchain node as part of the blockchain service which emits event whenever anything of interest happens on the network like transaction submission, block generation and so on, the second way that provides data about what is happening inside the network and the state of the blockchain nodes are the metrics that are implemented as part of the framework. The third way is by integrating a monitoring tool to know the status of various components of the network. Data from all these tools are collected and consolidated by Dashboard service which gives an overview of the status of network.

DevOps for the framework

Framework also provides a CI/CD pipeline for the applications built using framework. It is a docker based pipeline where the code is pulled from the git repository by a Jenkins Job and an image is created out of the repository by parsing a build configuration file. One the image is created tests are run against the image and if passed the image is versioned and pushed to the docker registry. Similarly, when deploying images are pulled from the docker registry. This CI/CD pipeline based on Jenkins and docker registry is converted to a HyperV image using Packer tool which can help in setting up the pipeline or replicating the pipeline easily.

Lessons learnt from Framework development

Framework in its current state supports Azure cloud service for Infrastructure creation and the blockchain platforms supported is Ethereum. Support for Hyperledger Fabric is in progress.

Azure templates are being used in the framework for the programmatic creation of infrastructure in declarative format along with the cloud init script using the Azure Node SDK which helps in creation of the infrastructure and starting an agent in the created virtual machine.

One of the challenges while implementing the Framework is the ability of the framework to support multiple blockchain platforms. Even though the concept of blockchain is same across various implementations of Blockchains, there is drastic differences in the implementations of various blockchain platforms which poses a challenge when building a framework that should support multiple blockchain platforms. To overcome this challenge and for ease of setting up the platform, the services have been dockerized and the images are stored in a docker registry which enables creation of blockchain containers on the fly and simplifies the switching of the blockchain platform in runtime.

Another Challenge that we faced is that the process of initialization of the blockchain network changes widely based on the blockchain platform chosen. For example, comparing the platform Ethereum and Hyperledger Fabric, Ethereum is initialized using the genesis block which will contain the information needed to start the node. Hyperledger Fabric on the other hand needs to create certificates for the organizational structure in order to start the network. Keeping these challenges in mind we have added a service called bootstrap service which provides the necessary data for blockchain nodes to start and become part of the network.

III. DESIGN RATIONALE

As discussed above, Flexibility, Reactive API, Observability,

Independent services and Extensibility are the design goals that guided the development of the framework. The following are the various design patterns used as part of the Framework.

Microservices

Though we started off with a monolithic architecture while development of the proof of concept for the framework, we quickly realized that Monolithic architecture will not satisfy our design goals due to the following reasons

- It is difficult to maintain the codebase in an environment where multiple developers work on the same codebase.
- It adds more coupling between services which makes it hard to make major changes to one service without affecting others.
- It is very difficult for a new team member to start working on the implementation as it needs knowledge of all other parts of the monolith.

Due to these reasons, we implemented the framework as a set of microservices which reduces coupling and make it easier to implement changes or work on a service without worrying about the affect these changes can have on other services. Microservices also help in individual scaling of services.

Publish Subscribe Messaging

Communication pattern is one of the important parts of a microservices design and we chose Publish Subscribe (PubSub) messaging pattern over a HTTP based communication because PubSub pattern provides more decoupling between services and it also helps in extending the capabilities of framework without changing other services. PubSub pattern also suits the way blockchain is designed as any transactions we post are only processed when a block is generated which is not an instantaneous process due to the consensus involved, so we need an API which can notify any interesting event happening inside the blockchain network. This was done in view of the design goals of Extensibility and Reactive API. Framework also provides HTTP API to all the services which gives developer flexibility to help extend the framework.

Service Deployment Model

Deployment patterns for the services in Framework are based on the scale at which the Framework has to be deployed but Blockchain service is deployed in a Service instance per container deployment model. As blockchain service is the interface to communicate with blockchain, large number of transactions are sent to the blockchain service during experiment, which might increase the load on the blockchain service. Because of this increased load effects like backpressure can come into play and distort the results of experiment. Due to these reasons Service instance per container deployment model for Blockchain service helps to avoid distortions to the data received while running experiments.

API Gateway and Discovery

As framework is to be used to run and compare experiments simulating multiple scenarios and changing multiple parameters, framework includes provisions like API gateway and Server side Discovery that can help in scaling services separately. Individual services can scale based on the experiments to be run and if scaling is necessary. Discovery service keeps checking for health of the services and updates accordingly so that requests can be routed to the service instances that are heathy.

Data Management

Data in the framework is managed by the service that creates that data. We are following a database per service pattern so that each data that is generated by the service is private to that service and can only be accessed by API provided by that service which helps to keep the data consistent in the framework as that data can only be accessed from one source.

IV. RESULTS AND DISCUSSION

The framework described above was used to launch experiments on Ethereum by simulating a scenario in which few Ethereum nodes and miners are part of a network. Some number of transactions are sent to this Ethereum network.

The experiments were conducted with Geth client version 1.3.6 on Azure vm of type Standard_D4 with 8 vCores and 28GB memory. The experiment has 2 stages:

- Initialization: In this stage the network is created and bootstrapped so that the experiment can be conducted. Blockchain network was initialized with genesis block with low difficulty to speed up the mining process and an account pre-allocated with large amount of ether. Once the network is created few accounts were created randomly among the nodes. A transaction allocated some ether was made from the account bootstrapped with ether in the genesis block to the newly created accounts, thus allocating the required ether to each account for running the experiment.
- Experiment: In this stage, large number of transactions were sent from and to a random account among the accounts created in the initialization stage. When these transactions are processed and blocks are generated, the listeners that are part of the blockchain service emits events to the framework when the blocks are generated. These events of block creation are subscribed by the dashboard service and the following plots were obtained.

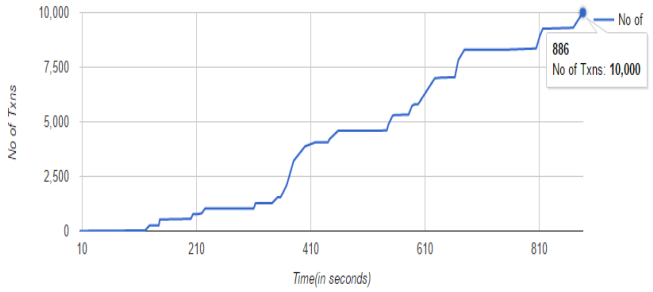

Figure 2: Time vs Number of transactions processed graph when 10000 transactions are sent to 10 nodes and 10 miners.

From the above plot, there are no transactions processed in the first 120 seconds, this is because when 10000 transactions were sent to randomly to the 20 go-ethereum (Geth) clients. The Geth clients were unable to process the transactions in the first 120 seconds and there was no transaction hash received immediately for the transactions sent. But after around 120 seconds, the Geth client processed the transactions. There were many interesting events happening when lot of transactions were sent to the network like network started mining stale blocks and uncle blocks. There was also an interesting behavior that generally do not happen in test environments which was as the network now had many miners mining at low difficulty, many miners simultaneously mined blocks which resulted in blocks on some nodes getting overridden after being accepted, as the new block was of higher difficulty. This minor chain reorganization created some issues as the listeners that sent out events when a block is generated in the network, sent out more than one events for the same block due to chain reorganization. This is one of the issues that shows the need

for such a framework to be used for testing the application in various scenarios before deploying on the live blockchain network.

Findings from Experiments

We have learnt some interesting ways in which Ethereum works when running the experiments. One such thing is that the order in which transactions are included in a block is not as per the order in which you send the transactions. For example, consider 3 accounts A, B, C where each account contains 10 ether, then if we send a transaction of 6 ether from A to B and then transactions of 4 ether from B to C. Then if we send a transaction of 7 ether from B to A and from B to C. If the transactions are processed in the same order in which transactions are sent, then all the transaction would have succeeded but that is not the case when we actually perform these transactions because the transactions may not be ordered in the block in the same way as they were sent.

There is another issue that we found to be interesting is the security issue in the geth version that we used, which makes it possible to send multiple transactions by unlocking the account once. This gives the attacker a chance to send a malicious transaction during the time in which account is unlocked, if the RPC endpoint for sending transaction is enabled. This issue was also mentioned by Parity [11].

V. CHALLENGES AND FUTURE WORK

We faced some challenges while developing the framework which we are mentioning here and possible solutions to the problems. Simulating a Byzantine Fault in a live network was an issue. Sometimes the results were delayed due to network delays and the test results were skewed. The blockchain distributions typically involve more than one programming language in most of the cases (Javascript and GoLang). This meant that familiarity with both the languages which is a difficult proposition to master. We are also exploring the Jepsen framework which is used to test NoSQL database reliability for the framework. The constant churn in Blockchain platforms also makes for a lot of challenges. We are also exploring the use of Formal methods to prove the Smart Contract. This seems possible in a Replicated State Machine with a limited set of instructions like Solidity but sounds difficult in a generic programming language like Chaincode.

Moving forward, we are working on improving the platform to support complex Byzantine faults scenarios readily. Along with developing the framework to be a blockchain verification and test platform with fault injection, we want to simulate scenarios for real life applications which can be directly integrated into the framework and tested easily. We are exploring the Service Mesh pattern [12] to have a clear control and separation over the activities happening in the control plane. We are also exploring the possibility of implementing GRPC API for communicating with various service of the Framework.

VI. RELATED WORK

There has been some similar work around the blockchain space like Ethereum's Hive [13] is a project similar to our work which does similar black box testing of the Ethereum platform as well as various implementations of clients. There has been work similar to the chaos experiments part of the framework for distributed databases called Jepsen [14] which also injects failures and tests to check the reliability of the platform.

VII. CONCLUSION

In this paper, we have introduced Framework, a set of microservices that can help development teams to speed up Blockchain development and a platform that provides the tools necessary to test blockchain platforms by creating custom scenarios. We believe that this Framework is a step towards improving the tooling around Blockchain platform testing and rapid development of blockchain based applications. The Framework provides a unique feature of testing blockchain platforms by simulating failure scenarios and attack modes based on a simple natural language like Gherkin, based interface which can be used by business teams also.

APPENDIX A. SAMPLE SCRIPT TO SPECIFY EXPERIMENTS.

```
Given Ethereum Network ABC with 4 nodes
And Ethereum Network ABC with 5 miners
When Network ABC is ready
And 10 accounts [Acc] are created
And following contracts [contract] are deployed
 | contractFile | From Account | Params | value |
 | test.sol     | $Acc[1]      | 1,2,3  | 400   |
And 1 block is mined
And following Tx are sent between Accounts
 | from    | to      | value | number |
 | $Acc[2] | $Acc[3] | 300   | 2      |
And following Tx are sent to contracts
 | from    | to           | function | params    | value |
 | $Acc[2] | $contract[3] | testFunc | 'test', 1 | 250   |
Then blocks will be generated with 3 tx
```

APPENDIX B. SAMPLE SCRIPT TO SPECIFY EXPERIMENTS ON HYPERLEDGER FABRIC.

```
Given Fabric network with name test
And with 3 peers in org1
And with 1 orderer
And 2 accounts in $peers[0] of org1
And 2 accounts in $peers[1] of org1
And 2 accounts in $peers[2] of org1
And 1 channel with orderer as $orderers[0] from peer $peers[0] of org1
And peer $peers[0] of org1 join channel $channels[0]
And peer $peers[1] of org1 join channel $channels[0]
And peer $peers[2] of org1 join channel $channels[0]
And network test is ready
When install following contract on $peers[0], $peers[1], $peers[2] of org1
    |contractFile|
    |test.go     |
And instantiate following contracts from $peers[0] of org1
    |contract| fromAccount| params|
    |$contract[0]| $acc[0]| "init", "a", "100", "b", "200"|
And 1 block is created
And following transactions are sent to contracts
    |from    | to        | function | params        | number|
    |$accPeer0|contract[0]| invoke   | "a", "b", rand(0,40)|100|
    |$accPeer1|contract[0]| invoke   | "a", "b", rand(0,40)|100|
    |$accPeer2|contract[0]| invoke   | "a", "b", rand(0,40)|100|
Then blocks will be generated with 300 transactions
```

ACKNOWLEDGMENT

.

REFERENCES

[1] Acolyer. "Internet Scale Services Checklist." Gist. Accessed August 21, 2017. https://gist.github.com/acolyer/95ef23802803cb8b4eb5

[2] Yuan, Ding, Yu Luo, Xin Zhuang, Guilherme Renna Rodrigues, Xu Zhao, Yongle Zhang, Pranay Jain, and Michael Stumm. "Simple Testing Can Prevent Most Critical Failures: An Analysis of Production Failures in Distributed Data-Intensive Systems." In *OSDI*, pp. 249-265. 2014.

[3] Nygard. *Release It!* OReilly Media, 2008.

[4] Castro, Miguel, and Barbara Liskov. "Byzantine fault tolerance." U.S. Patent 6,671,821, issued December 30, 2003.

[5] Ethereum. "Ethereum/wiki." GitHub. Accessed August 21, 2017. https://github.com/ethereum/wiki/wiki/Problems.

[6] "Swagger." Swagger. Accessed August 21, 2017. https://swagger.io/.

[7] Prasanna, Dhanji R. *Dependency injection*. Manning Publications Co., 2009.

[8] Eugster, Patrick Th, Pascal A. Felber, Rachid Guerraoui, and Anne-Marie Kermarrec. "The many faces of publish/subscribe." *ACM computing surveys (CSUR)* 35, no. 2 (2003): 114-131.

[9] Wynne, Matt, and Aslak Hellesoy. *The cucumber book: behaviour-driven development for testers and developers*. Pragmatic Bookshelf, 2012.

[10] Cucumber. "Simple, human collaboration." Cucumber. Accessed August 21, 2017. https://cucumber.io/.

[11] Konstantin. "Parity: Stepping Up the Security Model 1: A modular approach to transaction signing." Parity Technologies. May 30, 2016. Accessed August 21, 2017. https://blog.ethcore.io/parity-stepping-up-the-security-model-1-a-modular-approach-to-transaction-signing/.

[12] Calcado, Phil. Pattern: Service Mesh. August 3, 2017. Accessed August 21, 2017. http://philcalcado.com/2017/08/03/pattern_service_mesh.html.

[13] Szilágyi, Peter. "Hive: How we strived for a clean fork." Ethereum Blog. October 28, 2016. Accessed August 21, 2017. https://blog.ethereum.org/2016/07/22/hive-strived-clean-fork/.

[14] "JEPSEN." Distributed Systems Safety Research. Accessed August 21, 2017. https://jepsen.io/.