Ark: A Distributed Forum on Hyperledger Fabric

Ray Huang
Department of Electrical & Computer Engineering
University of Toronto
Toronto, Canada
ruixin.huang@mail.utoronto.ca

Lichen Liu
Department of Electrical & Computer Engineering
University of Toronto
Toronto, Canada
issac.liu@mail.utoronto.ca

Sining Qin
Department of Electrical & Computer Engineering
University of Toronto
Toronto, Canada
sining.qin@mail.utoronto.ca

I. Introduction

We present Ark, a distributed forum built on top of Hyperledger Fabric. As mentioned in Section II, we are motivated by concerns about traditional online social platforms where the owning companies are capable of manipulating and exploiting user data for profits, sometimes even against users' will. The design considerations and architecture of our solution, a distributed forum that protects user data through cryptography and promotes community growth through a sustainable reward system, are detailed in Section III. We demonstrate results in Section IV, including the performance of our implementation obtained with local experiments and reward distribution obtained by simulating various scoring strategies. We analyze potential attacks to the system and discuss learnings and future work in Section V.

II. Problem Statement

The Internet has long become an important medium for social interactions. Companies provide services for social purposes through various platforms such as blogs and video sharing websites, which sustain with user-generated content.

Users have raised two major concerns about those services due to the fact that they are solely operated by the owning companies.

- *The persistency of user data*: As user accounts and data are centrally managed by companies, they play a dominant role in deciding what users to be banned and what content to be deleted. Provided pressure from organizations or individuals of power, companies may manipulate user-generated content and services available to users against users' common will, for example, blocking searches about a national scandal.
- *The collection and usage of personal information*: Knowledge about users can be easily turned into profits through means such as providing targeted advertisements or tailored services based on user preferences. Companies are thus driven to collect user data, such as personal information and tracking user activities. However, if companies are not careful enough with the security of their system, data breaches can lead to serious consequences on the user side, for example, becoming victims of fraud.

III. Solution

With the two concerns from Section II in mind, the goal of our project is to design and implement a distributed version of a particular social platform, online forum, that:

- *Protects user identity and user-generated content by cryptography*: Anyone is allowed to participate in querying content from the forum, and can publish content as long as a public-private key pair is provided. The public key is used as the identity for the user, whereas the private key proves the authenticity. In other words, the public-private key pair is the only thing required to participate in publishing content onto the forum, thus minimizing data collected from users. Further, a user signs the generated content with the private key, and the signature can be verified by other users with the corresponding public key. Paired with the functionalities provided by the forum service chaincode and mechanism in blockchain that a later data block includes a hash result generated by hashing previous blocks, users are assured that their generated content is protected against modification and deletion.
- *Encourages community growth with a sustainable reward system*: To sustain the system without needing to make profit from user data, we incorporate a reward system to incentivize authors of good posts and users who decide to acknowledge the quality through liking with reward points. More likes make a post more popular, and users who like it earlier will get more points. Similarly, users can dislike posts to demote those with inappropriate content and get rewarded for flagging it. Since initially it costs points to like or dislike a post, a rational user should wisely choose which post to perform actions on, and can rely on the likes and dislikes on a post to determine its quality and decide whether to read or ignore it.

In this section, we first present the architecture of our distributed forum built on Hyperledger Fabric, then cover the design of various functionalities including chaincodes, data storage and reward system, and finally explain the implementation.

A. Solution Architecture

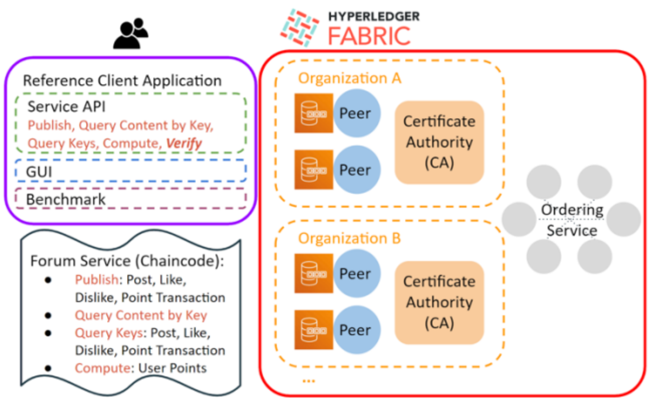

HYPERLEDGER FABRIC
Reference Client Application
Service API
Publish, Query Content by Key, Query Keys, Compute, Verify
GUI
Benchmark
Forum Service (Chaincode):
Publish: Post, Like, Dislike, Point Transaction
Query Content by Key
Query Keys: Post, Like, Dislike, Point Transaction
Compute: User Points
Organization A
Peer
Peer
Certificate Authority (CA)
Organization B
Peer
Peer
Certificate Authority (CA)
Ordering Service
...

Fig. 1. Architecture of our solution

As shown in Fig. 1, we use Fabric to construct a network of organizations to store ledgers, and a network of orderers to order transactions operated on ledgers. Each organization consists of peers each stores a copy of the ledger, and a certificate authority that issues and verifies credentials to those peers. A reference client application interacts with ledgers at peers through smart contracts, which are termed chaincode in Fabric, and are interchangeably referred to as forum services by us. For our purposes, our chaincodes include some to publish posts and likes, some to compute reward points, and some to query posts, likes and points. After a chaincode transaction is proposed by a client, it will be executed by the receiving peer. The results will then be sent to the ordering service for ordering, followed by being written to the ledger.

B. *Functionality Design*

1) *Forum Service*

The forum is hosted on the blockchain framework, and mainly serves as a data storage service with interfaces being defined in the smart contract or chaincode. The forum operates around three important features: posts, likes and dislikes. The forum provides services to allow client applications to submit and query on these features.

In addition, the forum provides all necessary metadata for the client applications to be able to verify that the content is indeed authenticated by the use of cryptographic signatures. However, the forum service that is hosted on the chaincode is not responsible for verifying these signatures. The client application may verify the validity and signatures of all data returned by the forum service by using the corresponding metadata if necessary. It is worth noting that the forum also does not provide any services for modifying or deleting existing posts, likes or dislikes. These measures are taken to comply with the functionality goal of keeping user contents immutable.

While posts constitute the fundamental features that this forum provides, likes and dislikes are also crucial in promoting community growth accompanied by a well-designed reward point system. At a high level, authors of posts are rewarded by likes and punished by dislikes. The reward points may be used by the client application to determine the quality of one's published contents. More details on the rewarding system will be discussed later in 3) reward system in this section.

The reward point system in the forum service is backed by point transactions. Each point transaction keeps track of the sources of reward points (payers, where the points are from) and destinations (payees, where the points are to). Multiple sources and multiple destinations are supported in a single point transaction to allow much more complicated use cases. A point transaction also keeps the issuer that initiated the transaction to make the transaction verifiable. The signature of the point transaction only comprises the payers but not payees, so that it allows the forum service to compute the reward points on behalf of the issuer of a like or dislike. Every submission of like or dislike automatically generates a point transaction, and they are cross-referenced for possible verifications.

Again, the entire forum service provides all tools, data and metadata for client applications to upload, fetch, and verify contents. The client application is free to publish any contents by invoking the corresponding chaincode. They are free to verify, filter, rank and display any data fetched from the forum service to their end users.

2) *Data Storage*

The entire forum uses four types of data to represent all fundamental features, namely: post, like, dislike and point transaction. The data are very general and have no assumptions on the underlying blockchain interfaces. Post keeps track of the author and the content of a post; Like and dislike store the user initiating the action and the targeting post; while point transaction traces all transaction flows of the reward points. Common to all data types, there are a few metadata fields, such as the timestamp.

In addition, these data structures are designed to be immutable, both logically and physically (guaranteed by the property of blockchain), after being appended onto the blockchain. Thus, they all have a signature field which is computed by taking the hash of all relevant information and encrypted by the private key of the submitting user. This provides tools for the application users to verify the authenticity of the content of all such data submissions.

As the system keeps an entire history of all point transactions rather than simply updating the latest point balance of any users, the point transaction data structure keeps a few "pointers" to the previous outgoing point transaction for all payers of the current point transaction. It also captures all incoming point transactions for these payers between the previous outgoing point transactions and the current transaction. This linked-list kind of tracking structure allows fast traversal between all point transactions that involve a particular user.

3) *Reward System*

The reward system provides a mechanism to encourage community growth and demoting low quality contents. The author of a post gains reward points when his posts are liked by other users, and loses reward points when the posts are disliked by others. Users consume their reward points to publish their likes/dislikes but are rewarded for sharing insights on the content qualities with the community based on the rank/position of likes and dislikes within that post. The reward point system is encouraged, but not required, to be used by the client applications for filtering, ranking, and promoting posts. For instance, a client application may use the authors' total reward point to hide all posts from authors with very low total reward points, because it indicates that contents (post, likes and dislikes) published by these authors have low-quality.

Due to the characteristic of chaincode invocation, the reward calculation cannot be implemented as a standalone service that must be invoked regularly every interval. Therefore, our reward system is designed in such a way that the reward is calculated at like/dislike submission.

Likes are used to reward the post author for the good content quality. The like rewarding should have a model that provides the following behaviours: a) The liker should consume reward points to avoid abuse; b) The author of the post should be rewarded and should be rewarded with more points when there are more likes; c) Previous likers of the post should be rewarded as well whenever a new like was applied to the same post for praising their likes; d) Be able to correctly handle the scenario of a self-liker. In the current implementation, every submission of a like will grow the world economy. Every like is linked to a point transaction. Whenever a like is submitted, a point transaction is created, with the liker being the payer, and author and previous likers being the payees.

Similarly, dislikes are used to penalize the post author for bad content quality. The dislike should have the following behaviours: a) The disliker should consume reward points to avoid abuse; b) The author of the post should be penalized and should be penalized with more points when there are more dislikes; c) The total penalty originated from a single post should be bounded or slow-growing, exhibiting a property of limited liability; d) Previous dislikers of the post should be rewarded whenever a new dislike was applied to the same post for praising their dislikes; e) Be able to correctly handle the scenario of a user spamming dislikes on a single post. Currently, every submission of a dislike will decrease the world economy. The point transaction for each dislike has the disliker (also the issuer) and the author being the payers, and the author and previous dislikers being the payees. The double presence of the author provides the flexibility for the forum service to compute the exact penalty on reward points of the author since the issuing disliker only needs to provide a list of easily determinable payers and point amounts.

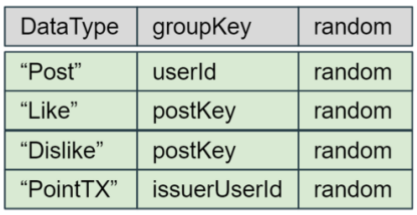
The reward system is designed to have an open economy structure, that is, the total number of reward points across all users may change over time. Compared to an economic structure with a constant total number of reward points, the open structure can more accurately represent the overall content quality of the entire community, such as the total number of likes and dislikes in the forum. Furthermore, it is allowed for a user to have a negative number of reward points, as it is the relative order between users' reward points that matter the most.

C. *Implementation*

The forum service and reference application are implemented in approximately 8000 lines of Java 11 code.

1) Forum Service on Chaincode

The forum service is hosted on Hyperledger Fabric chaincode. It provides a defined interface for client applications to publish and fetch data onto and from the ledger. All functionalities and data described in B. Functionality Design in this section are implemented into the chaincode in Java.



DataType	groupKey	random
"Post"	userId	random
"Like"	postKey	random
"Dislike"	postKey	random
"PointTX"	issuerUserId	random

Fig. 2. Composite keys for different data types

The data are written into the blockchain ledger using a key-value database interface provided by Hyperledger Fabric chaincode. The different types of data are serialized into bytes and are stored heterogeneously. However, a few features on keys provided by the key-value database interface are used to help to perform some basic queries. A key is a string that can be combined by multiple sections of subkeys, also known as a composite key. We designate the first section of the key to store the type of the data, using the second section as a group key, and the last section to be some pseudo-random values for ensuring uniqueness. Since the database interface supports partial composite key lookups, fetching all keys for a specific data type and group key becomes possible. Fig. 2 shows the design of the composite key for different data types.

The chaincodes implement all functionalities of the forum service into the following categories:

a) Publishing: Client applications can publish posts, likes, dislikes and point transactions;

b) Fetching data by key: Client applications can fetch data of posts, likes, dislikes and point transactions by keys;

c) Querying keys: Client applications can query all keys for a data type, and all keys that match with the group key. For example, query all post keys and all post keys with a specified author;

d) Computing point balance: Client applications can query all point transactions relating to a user, and the point balance of a user.

2) Reference Client Application

As mentioned in previous sections, the forum service can be freely invoked and used by the client application. We provide a reference client application demonstrating how the two goals of content protection and content quality can be achieved. This reference application has 3 parts: service API, GUI, and benchmarking.

The service API is a reference API that encapsulates the forum service chaincode invocation. In addition to those services provided by the chaincode, it also provides additional features that are not directly performed on the forum service chaincode, such as content verification. This includes signature checking and checking all the cross-references in data content of interest. As a result, the reference service API can serve as the bridge connecting the forum service chaincode and the customized client application implementation.

A reference GUI is implemented using the Java Swing framework and is built using the service API. The GUI is capable of publishing posts, likes and dislikes, and all kinds of querying on these contents as well. In addition, the GUI is capable of drawing a plot tracking a user's point balance history, as shown in Fig. 9. Upon displaying posts, likes and dislikes content, the GUI also shows a verification status, Fig.10 and Fig. 11 give an example of each.

Additionally, the benchmarking framework is also built upon the service API to measure performance and simulate reward points. This will be further discussed in Section IV.

IV. Results

In this section, we introduce the benchmarks and experiment setup and frameworks, present the performance of our system and reward distribution of sampled users obtained from simulation, and provide our analysis on those results.

A. *Benchmarks*

There are two types of experiments conducted:

- *Performance:* Only the latency was measured for analysis as the benchmarking was performed in serial.
- *Simulation:* Simulation serves for analyzing how the economic system evolves under the aforementioned reward system. Three specific scenarios are selected to

better understand how different strategies to perform likes and dislikes result in different reward distribution: all users liking a particular post, all users disliking a particular post, and a more realistic scenario where users randomly like and dislike random posts. The point balance history was plotted out for a few sampled users.

B. *Experiment Setup and Frameworks*

The benchmarking was performed on a desktop computer with a 4-core Intel i7 3770k processor operating at 4.1 GHz, with 16 GB of DDR3 RAM. The Hyperledger Fabric network was running in the docker with WSL2 as its backend on Windows 10. The network was configured to include 3 peer nodes with a total of 3 organizations in addition to 1 orderer node. Batch is disbled by setting the number of transactions per batch to 1, since a few forum services have dependencies on existing ledger states.

The performance framework started with empty ledger states. Tests to publish posts, likes and dislikes, respectively, are run, followed by those that query and fetch data.

The simulation framework aims to simulate a closed economic system under different usage scenarios. A specific simulation is structured by two parts: an initial state and a combination of actions. Currently, the simulation framework allows to set up an initial state in the following three ways: 1). Define a pool of authors that are responsible for publishing new posts; 2). Define a pool of viewers that resembles normal forum users, which are responsible for liking/disliking posts; 3). Optionally, define a pool of posts that will be liked or disliked.

Then there are three types of actions that can be triggered , which are: Post, Like and Dislike. During a Post action, an author will be drawn from the author pool, and publish a new post, then the new post will be added to a post pool. In cases of Like and Dislike, a viewer is drawn from the viewer pool and a post is drawn from the post pool, the Like/DisLike action will be done against that post by the chosen viewer.

Items can be drawn from the pools in different fashions, by Round Robin or a probability assigned to that item when it is added into the pool. With the benchmark framework, we are able to design comprehensive scenarios and analyze the economic impacts resulting from those scenarios.

C. *Performance Results*

The latency results can be found in Fig. 3, where the top graphs demonstrate latency per iteration for 3 write operations (left) and the remaining 10 read operations (right), respectively, and the bottom graph shows the average latency for all 13 operations.

Among the three write operations, publishing posts is the fastest and takes about constant time. Time spent to like or dislike a post increases as the size of the ledger becomes larger, following a quadratic trend. This is due to publishing a like and dislike require information of existing likes and

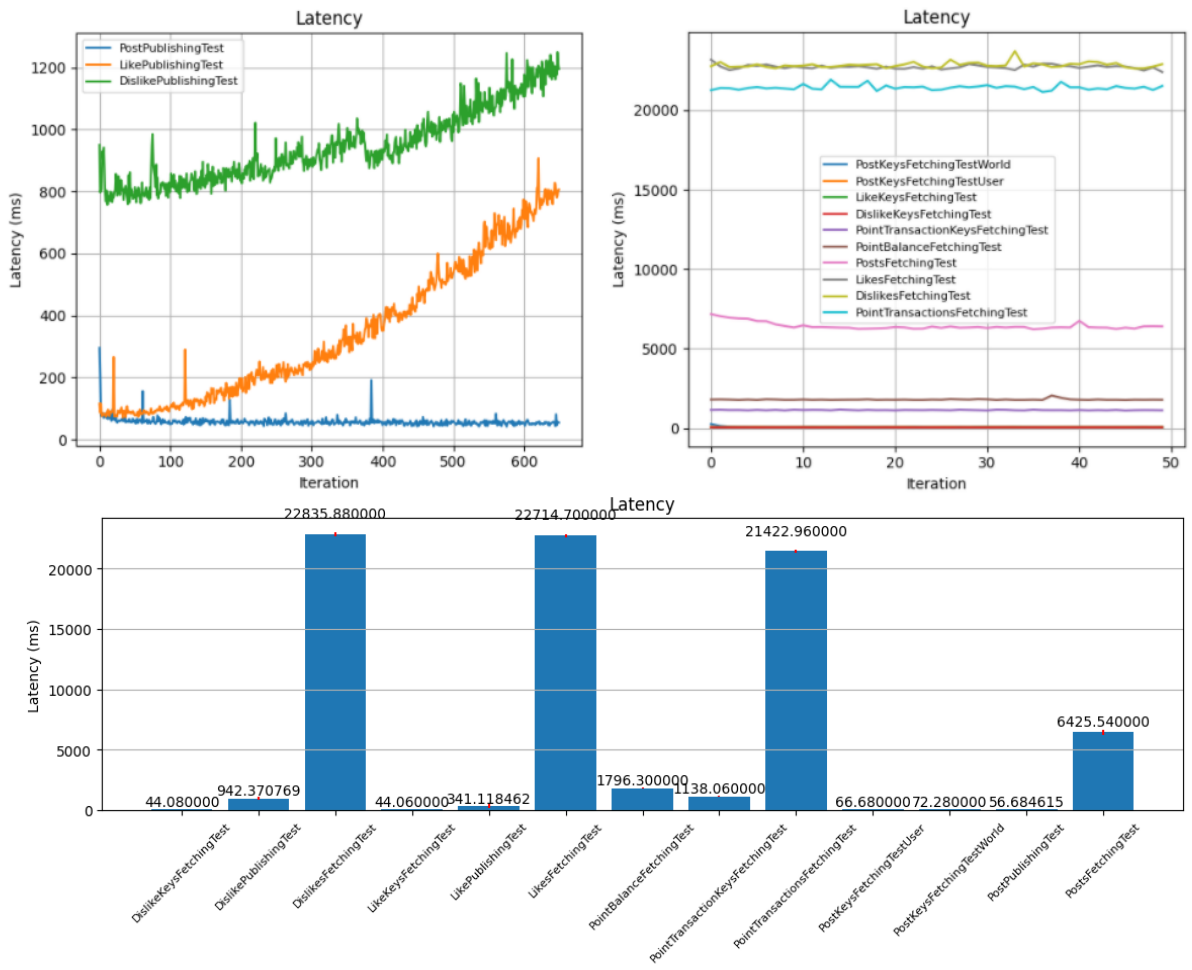


Fig. 3. Latency of various operations measured in performance experiments: top left – latency of write operations; top right – latency of read operations; bottom – average latency of all operations

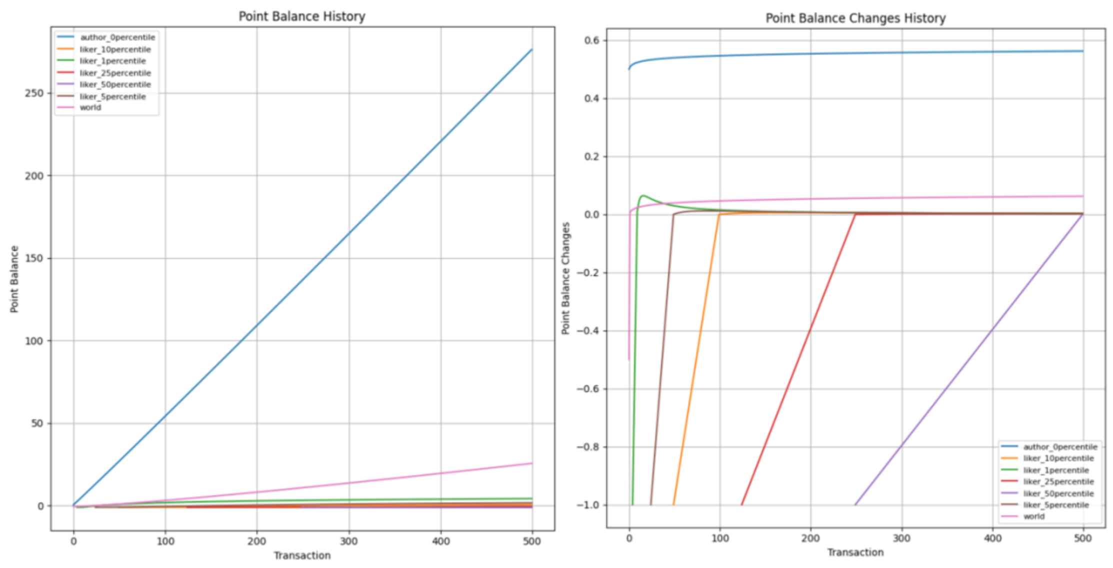


Fig. 4. Point balance history and change history of the author of a post and the 1st, 5th, 10th, 25th and 50th percentile of the users who liked it

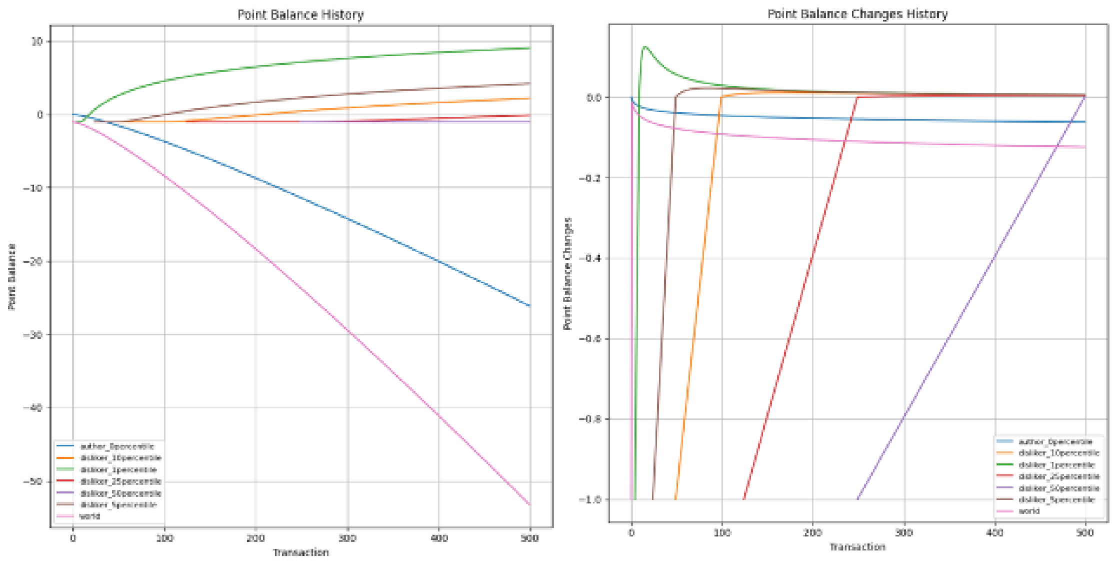


Fig. 5. Point balance history and change history of the author of a post and the 1st, 5th, 10th, 25th and 50th percentile of the users who disliked it

dislikes on a post to calculate a reward or penalty that previous participants, as well as the author, will receive. As the chain grows longer, and the post receives more likes and dislikes, liking and disliking will involve significantly more data fetching, therefore resulting in higher latency. In addition, this is also due to likes and dislikes are all backed by point transactions which introduces another level of dependency.

The latency of each read operation stays about the same throughout the experiment. Firstly, the few fetching tests that have the largest latencies include fetching and verifying the data, such as fetching likes, dislikes, point transactions and posts. The large latency is due to the content verification involved, which is performed in the reference service API in the reference client application. As the verification is not part of the forum service API, there are little performance implications to other client applications who do not verify contents. Post fetching is faster than the other 3 since there is no need to cross-check point transactions. The various key fetching can be found at the bottom of the graph, which indicates that they are very fast since they are directly supported by the forum service chaincode.

D. *Reward Distribution Results from Simulation*

As mentioned previously, the model we used to reward users based on likes and dislikes contains various tunable parameters, such as the ratio of points going to the author vs. to those who liked a post and inflation rate, and interchangeable formulas, such as beta distribution and logarithm. Thus, the results presented here are those produced with one particular set of parameters and functions. The system can be further tuned to generate desired reward distribution for more scenarios. We leave more exploration on this for future work.

Fig. 4 shows the reward history of the author of a post and the 1st, 5th, 10th, 25th and 50th percentile of the users who liked this post. With the current set of parameters, most of the points spent to like the post went into the author's pocket. Sampled non-author users are those who liked the post relatively early, thus as desired, did not lose points and gained more if the like happened earlier.

Fig. 5 corresponds to the scenario where a post is disliked by many users. With the current set of parameters, the author of the bad quality is correctly punished with loss of points, and users who contributed to the identification of bad content early are correctly rewarded. However, the loss of points at the author caused by dislikes is much smaller compared to the gain obtained from likes. This means currently dislikes cannot offset likes effectively, making it more difficult to rank bad-quality content to lower accordingly.

Results of a more realistic setup can be found in Fig. 6. In this case, users are posting new posts, liking and disliking posts during the simulation. Specifically, all authors have an equal chance to publish a new post. All published posts have a chance of being liked or disliked following a normal distribution. When triggering a like or dislike action, all viewers also have an equal chance to like/dislike a post. The proportions of actions triggered as post, like and dislike roughly follow the 1: 7: 2 ratio. As a result, authors tend to benefit from such assumed scenarios, for their fortune consistently grows throughout the simulation. On the other hand, the wealth growth of viewers seems less predictable, as it depends on if they successfully “bet” on any popular published posts.

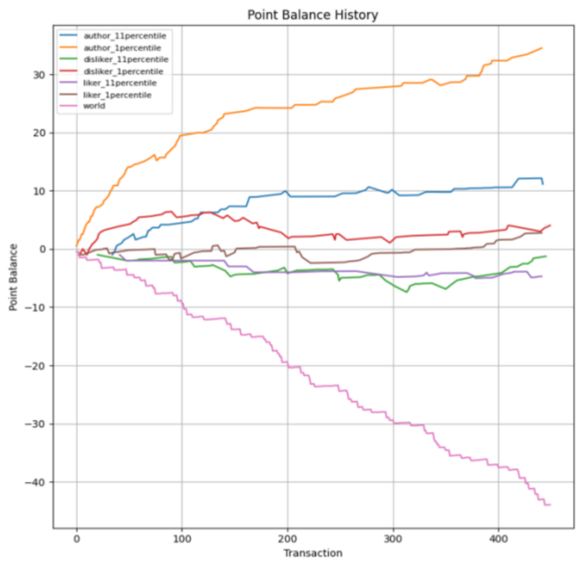


Fig. 6. Point balance history of a more realistic scenario

V. Discussion

We briefly discuss lessons learned while implementing our solution using Hyperledger Fabric, analyze potential attacks to the system, and identify some future work for improvement.

A. Lessons Learned

We chose to adopt Hyperledger Fabric because it is open-source thus free to use, popular thus has proper documentation and is supported by Amazon Web Services (AWS) thus easy to deploy. However, Fabric only supports permissioned blockchain, while our system is better suited to be built over a permissionless one. This mismatch led to unnecessary complications in the implementation.

A better alternative we later became aware of is Hyperledger Sawtooth, which supports both permissioned and permissionless blockchains.

B. Potential Attacks

1) Invalid content and signature

As mentioned before, publishing a new post requires the publisher to use its private key to generate a signature based on the hash of the post content, timestamp and the publisher’s public key. These information are all publicly viewable when a user views a post, thus a viewer can generate the same hash used when the publisher published the post. Therefore if a post content was modified by an attacker, it will be caught by decrypting the signature using the publisher's public key and compare the result with the hash to check if they are equal.

2) Self-Likers

It is possible for a user to gain reward points by excessively liking a post that was authored by him. An ideal reward model for likes should be prone to such use cases. However, simulation based on the current reward model proves this grants a self liking author the ability to abuse the reward system, resulting in a zero-cost way for an author to massively gain wealth, as shown in Fig 7. In reality, such authors will be greatly punished if the post is disliked by a lot of other users. There are many other approaches to prevent this scenario from happening, including simply banning an author from liking his own post. Therefore, we leave it as future work to improve the mechanism of like rewarding.

3) Society Haters

It is possible for a user to abuse the dislike mechanism to attach to other authors by excessively disliking their posts. Simulation from the current implementation as shown in Fig. 8 shows that when a user (society hater) excessively dislikes a post, the author loses fewer reward points than the disliker. However, this still cannot protect the author from being penalized for the excessive amount of fake dislikes imposed by society haters, who usually do not have a good reputation. This is also left to future work.

C. Future Work

Results presented in Section IV are obtained with local runs of experiments in docker containers on a single machine. Thus, the first improvement is to deploy to AWS for evaluation. In fact, we have tried to set up AWS, but the latest Hyperledger Fabric version supported by Amazon managed blockchain uses Java 8, whereas our chaincode was written in Java 11. As a result, we were unable to install our chaincodes onto peers managed by AWS.

Another improvement is to increase throughput. Currently, in our benchmarks, every invocation is a blocking call, thus slow in execution. To support parallelism, we need to reduce dependences between functionalities, otherwise, those dependencies will prevent batching.

In this project, we have only been able to implement a naive reward model. The first aspect is that dislikes and likes have different impacts on authors’ reward points. The second issue is that the current model is not prone to self-like attacks and also not able to completely eliminate society-hater attacks. A good direction is to consider the like/dislike issuer’s point balance at the time of the submission of like/dislike when calculating the reward. So that the reputations of the likers/dislikers are considered for their likes and dislikes.

Appendices

A. *Code*

All code for this project can be found at: https://github.com/lichen-liu/ark2

B. *Work Allocation*

- Ray Huang: Application logic, benchmarks
- Lichen Liu: Chaincodes, GUI
- Sining Qin: Deployment, analysis

C. *Additional Graphs*

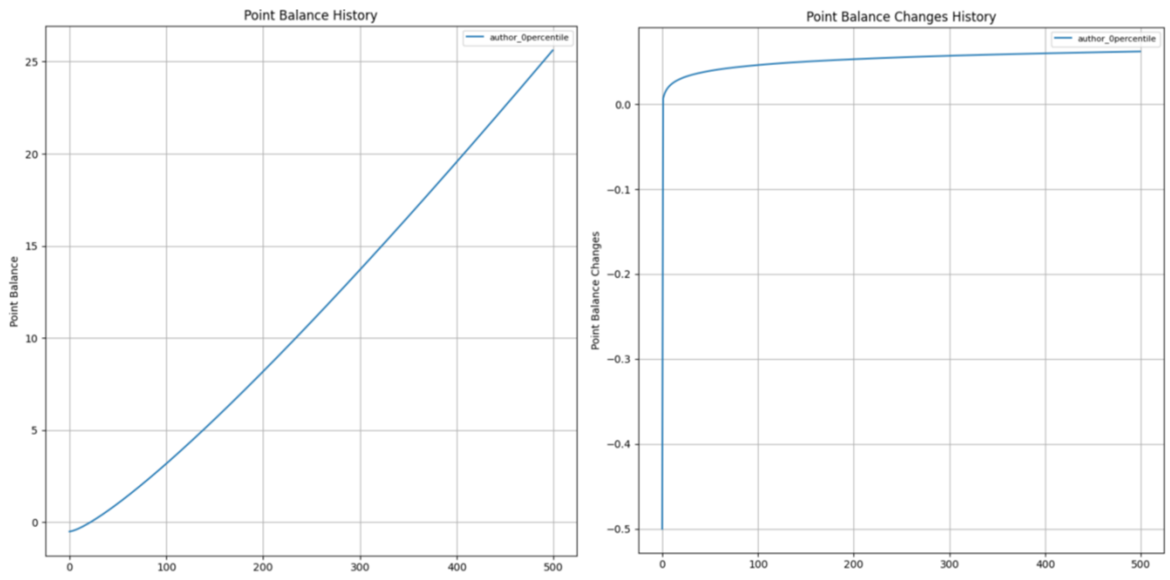


Fig. 7. Point balance history and change history of a self-liker, i.e. a user who repeatedly like his/ her own post

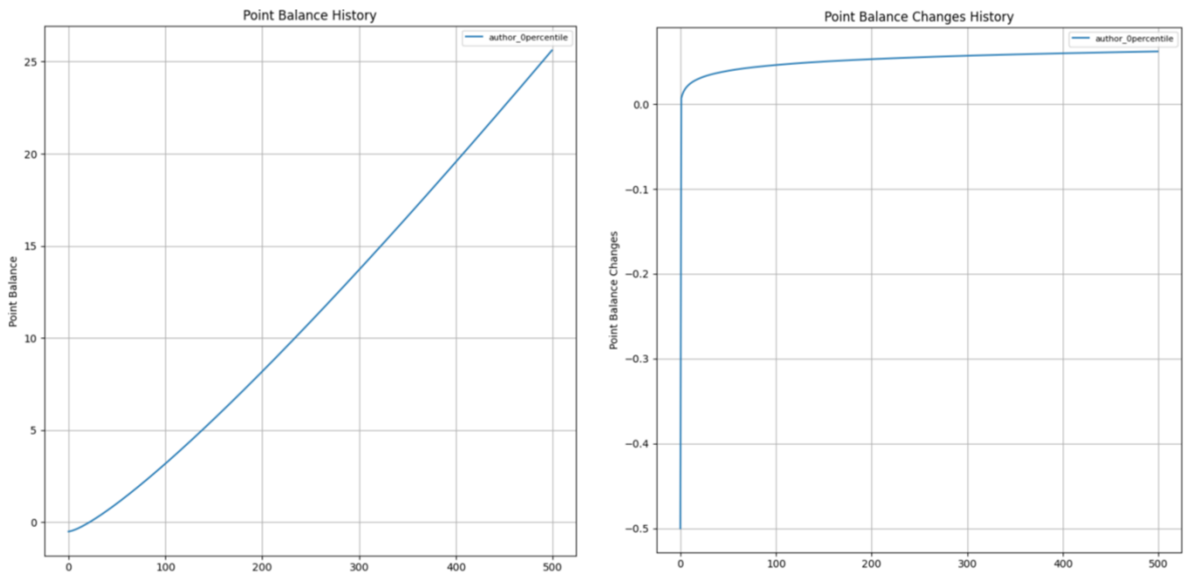
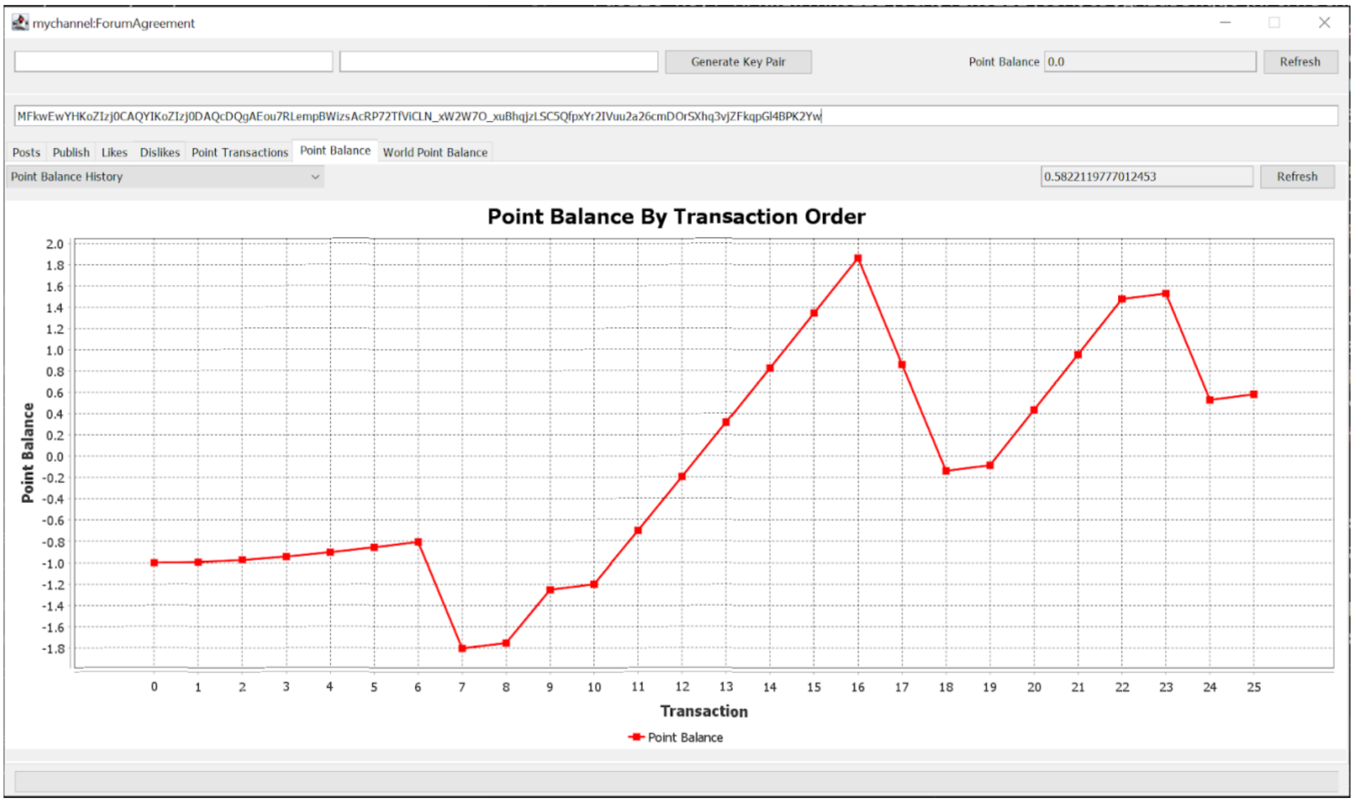
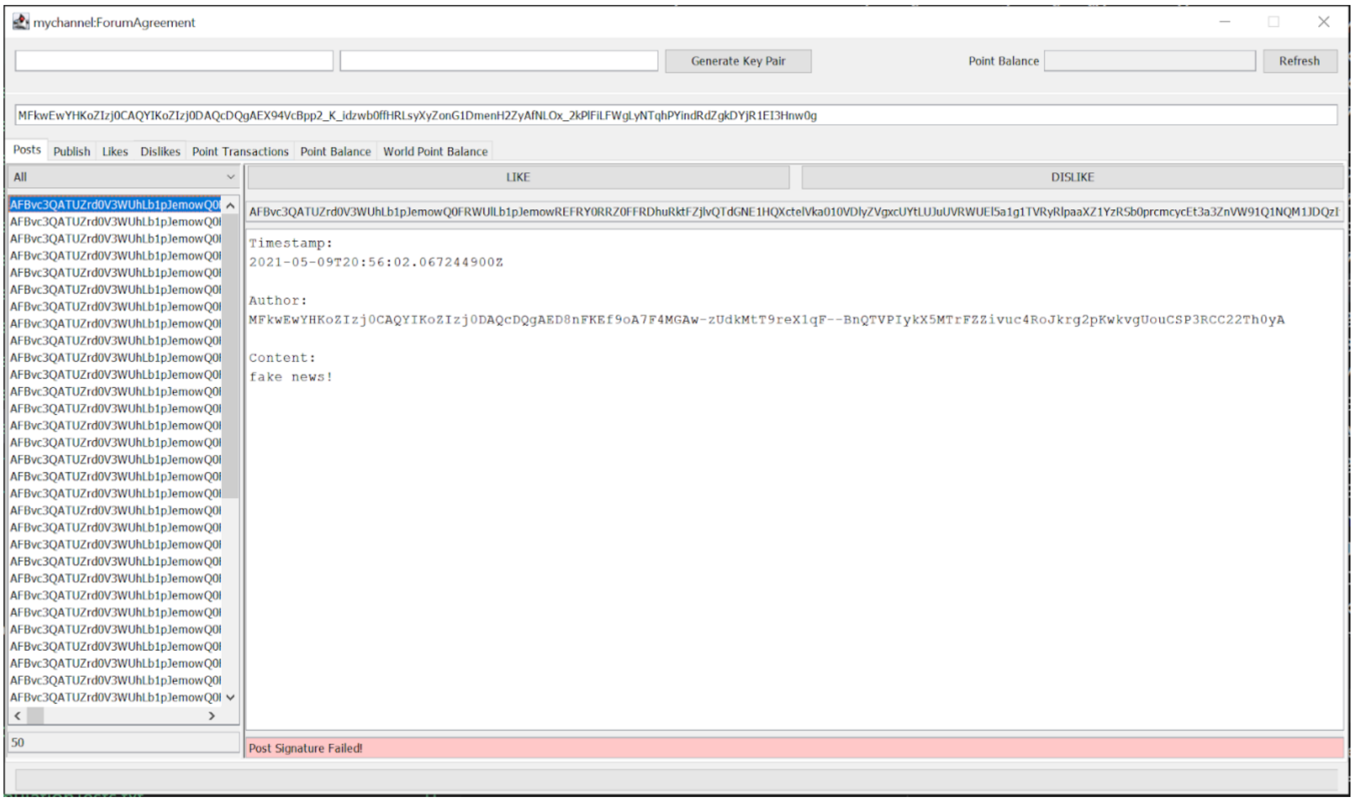


Fig. 8. Point balance history and change history of a society hater, i.e. a user who randomly dislike many posts



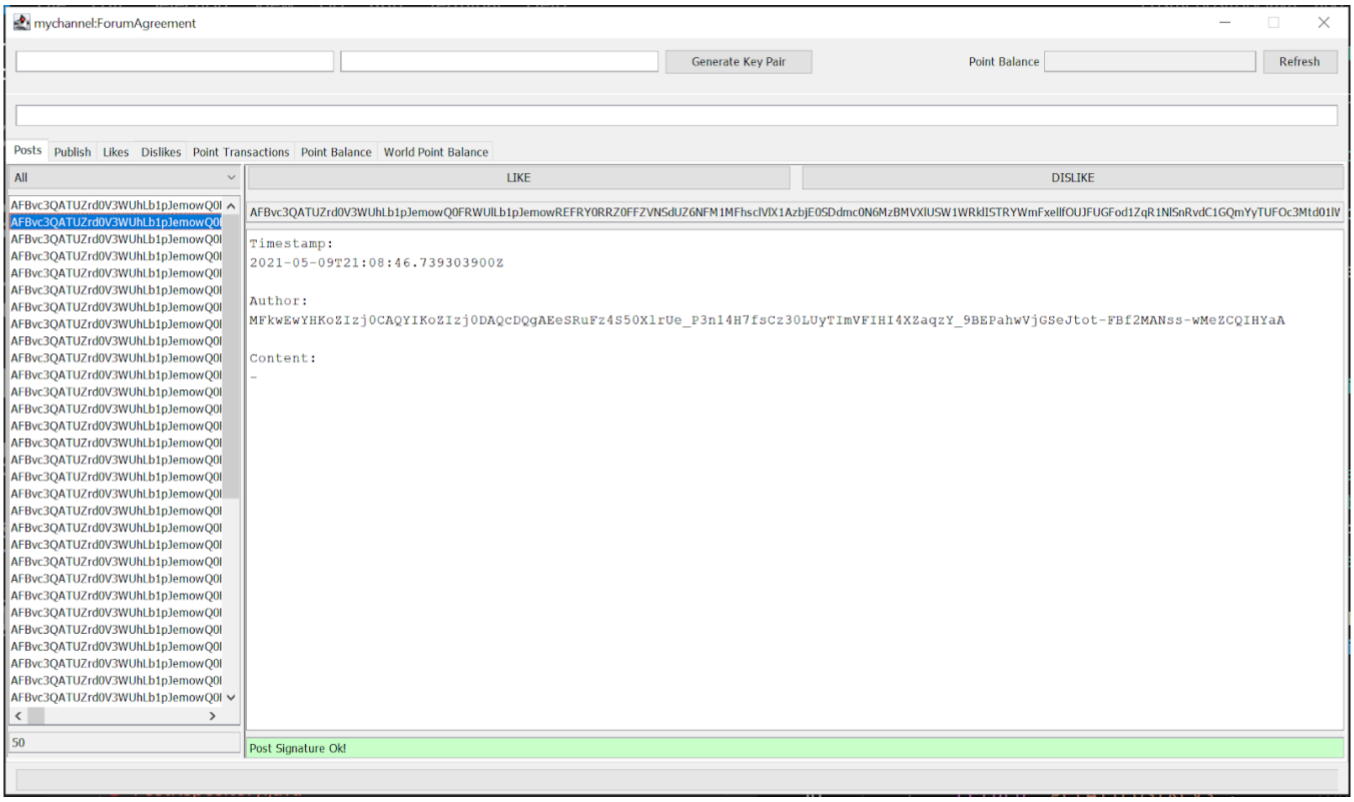
```
Point Balance 0.0
MFkwEwYHKoZIzj0CAQYIKoZIzj0DAQcDQgAEou7RLempBWizsAcRP72TfViCLN_xW2W7O_xuBhqjzLSC5QfpxYr2IVuu2a26cmDOrSXhq3vjZFkqpGl4BPK2Yw
0.5822119777012453
```

Fig. 9. An example of user’s point balance history



```
MFkwEwYHKoZIzj0CAQYIKoZIzj0DAQcDQgAEX94VcBpp2_K_idzwb0ffHRLsyXyZonG1DmenH2ZyAfNLOx_2kPlFiLFWgLyNTqhPYindRdZgkDYjR1EI3Hnw0g

AFBvc3QATUZrd0V3WUhLb1pJemowQ0FRWUlLb1pJemowREFRY0RRZ0FFRDhuRktFZjlvQTdGNE1HQXctelVka010VDlyZVgxcUYtLUJuUVRWUEl5a1g1TVRyRlpaaXZ1YzRSb0prcmcycEt3a3ZnVW91Q1NQM1JDQzI

Timestamp:
2021-05-09T20:56:02.067244900Z

Author:
MFkwEwYHKoZIzj0CAQYIKoZIzj0DAQcDQgAED8nFKEf9oA7F4MGAw-zUdkMtT9reX1qF--BnQTVPIykX5MTrFZZivuc4RoJkrg2pKwkvgUouCSP3RCC22Th0yA

Content:
fake news!

50
Post Signature Failed!
```

Fig. 10. An example of a post that fails signature verification



```
AFBvc3QATUZrd0V3WUhLb1pJemowQ0I
AFBvc3QATUZrd0V3WUhLb1pJemowQ0I
AFBvc3QATUZrd0V3WUhLb1pJemowQ0I
AFBvc3QATUZrd0V3WUhLb1pJemowQ0I
AFBvc3QATUZrd0V3WUhLb1pJemowQ0I
AFBvc3QATUZrd0V3WUhLb1pJemowQ0I
AFBvc3QATUZrd0V3WUhLb1pJemowQ0I
AFBvc3QATUZrd0V3WUhLb1pJemowQ0I
AFBvc3QATUZrd0V3WUhLb1pJemowQ0I
AFBvc3QATUZrd0V3WUhLb1pJemowQ0I
AFBvc3QATUZrd0V3WUhLb1pJemowQ0I
AFBvc3QATUZrd0V3WUhLb1pJemowQ0I
AFBvc3QATUZrd0V3WUhLb1pJemowQ0I
AFBvc3QATUZrd0V3WUhLb1pJemowQ0I
AFBvc3QATUZrd0V3WUhLb1pJemowQ0I
AFBvc3QATUZrd0V3WUhLb1pJemowQ0I
AFBvc3QATUZrd0V3WUhLb1pJemowQ0I
AFBvc3QATUZrd0V3WUhLb1pJemowQ0I
AFBvc3QATUZrd0V3WUhLb1pJemowQ0I
AFBvc3QATUZrd0V3WUhLb1pJemowQ0I
AFBvc3QATUZrd0V3WUhLb1pJemowQ0I
AFBvc3QATUZrd0V3WUhLb1pJemowQ0I
AFBvc3QATUZrd0V3WUhLb1pJemowQ0I
AFBvc3QATUZrd0V3WUhLb1pJemowQ0I
AFBvc3QATUZrd0V3WUhLb1pJemowQ0I
AFBvc3QATUZrd0V3WUhLb1pJemowQ0I
AFBvc3QATUZrd0V3WUhLb1pJemowQ0I
AFBvc3QATUZrd0V3WUhLb1pJemowQ0I
AFBvc3QATUZrd0V3WUhLb1pJemowQ0I
AFBvc3QATUZrd0V3WUhLb1pJemowQ0I
50
```

```
AFBvc3QATUZrd0V3WUhLb1pJemowQ0FRWUlLb1pJemowREFRY0RRZ0FFZVNSdUZ6NFM1MFhsclVlX1AzbjE0SDdmc0N6MzBMVXlUSW1WRklISTRYWmFxellfOUJFUGFod1ZqR1NlSnRvdC1GQmYyTUFOc3Mtd01lV

Timestamp:
2021-05-09T21:08:46.739303900Z

Author:
MFkwEwYHKoZIzj0CAQYIKoZIzj0DAQcDQgAEeSRuFz4S50XlrUe_P3nl4H7fsCz30LUyTImVFIHI4XZaqzY_9BEPahwVjGSeJtot-FBf2MANss-wMeZCQIHYaA

Content:
-

Post Signature Ok!
```

Fig. 11. An example of a post that passes signature verification
]