NewsChain: News Co-authoring Platform Powered by Ethereum

by

PAN Zhiqi
(14252023)

A thesis submitted in partial fulfillment of the requirements for the degree

of

Bachelor of Science (Honors)

in

Computer Science

at

Hong Kong Baptist University

April 2019

Declaration

I hereby declare that all the work done in this Final Year Project is of my independent efforts. I also clarify that I have never submitted the idea and product of this Final Year Project for academic or employment credits.

_____

PAN Zhiqi

Date: _____

Hong Kong Baptist University

Department of Computer Science

We hereby recommend that the final year project submitted by PAN Zhiqi entitled "NewsChain: News Co-authoring Platform Powered by Ethereum" be accepted in partial fulfillment of requirements for the degree of Bachelor of Science (Honors) in Computer Science.

_____	_____
Prof. XU Jianliang	Dr. TAM, Hon Wah
Supervisor	Observer
Date: _____	Date: _____

Table of Contents

NewsChain: News Co-authoring Platform Powered by Ethereum

by

PAN Zhiqi

Department of Computer Science

Hong Kong Baptist University

ABSTRACT

A new era of blockchain technology has arrived and such booming technology has a wide usage across multiple areas. Ethereum is one of the most popular open platforms built on top of a blockchain, and this project, NewsChain, is a news co-authoring platform based on Ethereum. Citizen journalism plays an increasingly important role in news media, and in the belief of the wisdom of crowds, NewsChain aims to empower the general public to co-author their news together and brings the transparency back to the journalism industry. NewsChain by design makes the best out of the decentralization, immutability, transparency and traceability nature of Ethereum.

Introduction

Background of Blockchain and Ethereum

Blockchain is a new blooming technology in recent years, and has already been applied in several fields. Blockchain is ledger with a growing list of blocks containing transactional records, and it runs in a decentralized network where each node may hold a replica of all the blocks. To ensure data integrity and transparency, each block is chained to the previous one via a cryptographic hash and a consensus algorithm is run in the distributed network. Each node in the network can either be a light node or a full node, and can choose whether or not to be a mining node. Full node will contain all the blocks and transactions while light node only contains the partial information for a period of time or even potentially less. And a mining node is a full node that validates transactions using advanced computational power and gets rewarded once a block is mined successfully in the blockchain. Such computational power used in mining process also guarantees that transactions are irreversible once they are confirmed on the chain, and the consensus algorithm (e.g. proof of work) makes blockchain a ledger without the need of trust. Notice that almost all of the current financial services or journalism institutions nowadays are based on a trust model, which means that there are always a third-party to guarantee the transaction and give credibility and endorsement to those who involve in the transaction. However, blockchain technology is a game changer as it requires no

trust in the network as mentioned. Each node in the blockchain always considers the longest chain contained among all the nodes as the single source of truth and this avoids information being tampered as long as the majority of the nodes are honest. This way, in other words, unless more than half of the nodes are malicious or the computational power of the malicious nodes collectively outpaces the honest nodes, it is impossible to change the alter the records in blockchain.

Ethereum, an open decentralized software platform running on blockchain, is one of the most popular blockchain applications nowadays. Ethereum not only supports digital currency transaction, but more importantly, it facilitates smart contracts and related peer-to-peer applications. Our platform for co-authoring news indeed implements and applies a plenty of smart contracts in Ethereum to fulfill several innovative features that traditional journalism does not have.

Background of Journalism Industry

On the journalism side, in current world, information fed to the users is getting more and more biased due to an advertisement-driven nature of social media and news platforms. It becomes increasingly harder for ordinary users to see the whole picture of an event (e.g. social movement) because existing news providers keep feeding people what they want to hear instead of what they need to hear. As powered by

latest customization and personalization technology, current journalism industry becomes more and more attention-based, and this hence further leads to the problem of fake news, biased information and sensationalism. News industry becomes corrupted and is gradually diverging from the way it should be. Both news on social media platforms and traditional media are centralized and fully controlled by one or more authorities subject to censorship and selective bias. From agenda setting to priming, those authorities do not give a chance for the general public to voice for themselves.

Objective

We aim to break through in journalism industry by empowering the general public themselves with a news co-authoring platform powered by Ethereum network. The voice of the general public matters, and the power of citizen journalism should prevail. This project takes full advantages of Ethereum and introduces a platform where everyone is capable of (co-)authoring and a piece of news, and in such platform all pieces of news are traceable, transparent and cannot be tampered with. Smart contracts on Ethereum are the core part on this project, which transparently controls the way an article (or a piece of news) is created, modified, cited and rewarded. We break down the centralized authorities and censors who used to track and examine the news and build up a decentralized platform with open-source smart

contracts running on Ethereum, and finally an era of information co-authoring becomes inevitable. We have firm belief in the wisdom of crowds, i.e. the general public, and hence we are devoted into doing the right thing to eliminate the existing attention-based and advertisement-driven journalism model.

Technical Stack Overview

This chapter provides the audience with an overview of all the technology applied in this project with brief descriptions of corresponding use cases in order to ensure the audience can have a big picture of how this platform works before moving forward to later chapters.

Generally, the technical stack could be classified into three different layers, from the top to the bottom – the server-side technology, web technology and Ethereum and Swarm technology, as shown in Figure 1 below. We will go through them one by one in this chapter.

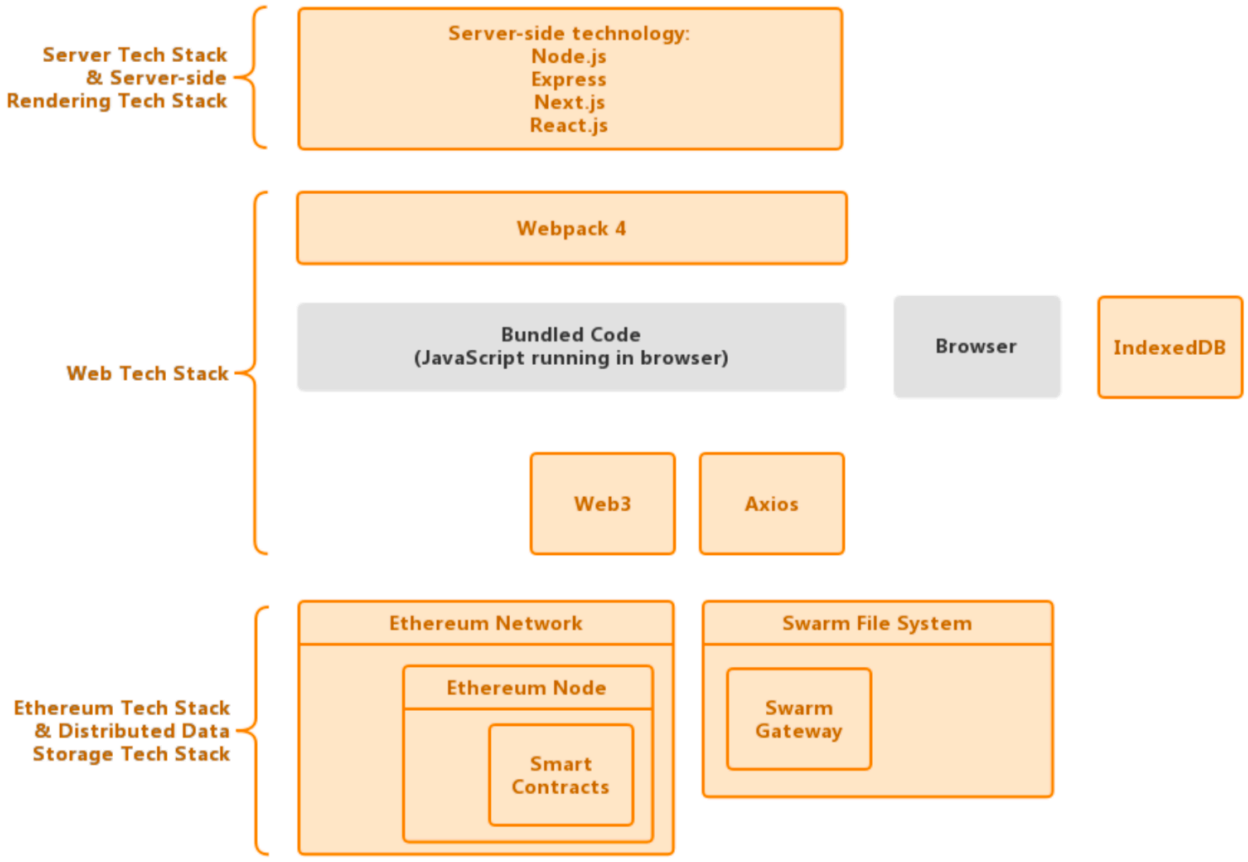


Figure 1. General tech stack in three layers

Server Technology Stack & Server-side Rendering Technology Stack

The top layer, i.e. the server technology stack and server-side rendering technology stack, includes the technology applied in the server and webpage rendering on server side. The whole layer is running in ***Node.js***, a cross-platform JavaScript run-time environment. In this layer, application programming interfaces are exposed using ***Express*** framework, providing all the server-side logic to the clients. For the webpage rendering, or precisely the pre-rendering of webpage before the final

rendering in user agent (e.g. browser), is done by ***Next.js*** and ***React.js***. We will talk more about server-side webpage pre-rendering later. Note that some other Node.js server technology or libraries used are omitted for simplicity, we will cover them later in System Design and Implementations chapter.

Web Technology Layer

The middle layer, i.e. the web technology stack, deals with all the web-related technology including code bundling and delivering, web browser caching and web browser communication. Note that we the bundled code and browser are drawn in grey color because they are not a technology applied but just added for better illustration. ***Webpack 4*** is a web resource bundler used for combining different resources into a proper single deployable (runnable) bundle in modern browser. For example, multiple ES6 JavaScript files (usually classes or components) are transpiled, bundled and minimized into one single file (technically there could be more than one if lazy loading or dynamic loading is enabled, we will cover this later). ***IndexedDB*** is a standard NoSQL database dedicated for caching on browser side, and it is applied in this project for the sake of performance. At the bottom part this layer, there are two communication agents – ***Web3*** for communication with an Ethereum node for data viewing and transaction and ***Axios*** for communicating with a Swarm gateway.

Ethereum Technology Stack & Distributed Data Storage Technology Stack

The bottom layer, i.e. the Ethereum and distributed file storage technology stack, consists of two components, one being the Ethereum network related technology (e.g. ***smart contracts***, ***solidity language*** and etc.), and the other one being the ***Swarm*** related technology. This layer applied decentralized technology replaces the traditional centralized database and object storage service (OSS) involved in traditional system. In particular, Ethereum replaces the database for storing records of pieces of news, and Swarm replaces the OSS for storing large data object like images and videos inside a news article.

All in all, this section covers the technology stack applied in this project from a general perspective to show a big picture to the audience. For simplicity, we do not cover everything in detail here, and many other necessary technical frameworks or libraries applied in this project such as data processing and visualization, unit testing and so on will be mentioned later.

Platform Features and Implementations

In this chapter, we will go through all the features this news platform has and will see how Ethereum becomes a game changer in news industry. Note that all transactions demonstrated in this chapter are done in Ethereum Rinkeby Network instead of the Main Network.

Article Creation and Persistence

On this platform, user who has an Ethereum account can create / publish articles and voluntarily provide his/her name. Different from traditional social media or news platform, all the published articles can no longer be deleted and all changes will be tracked. Article publishing is a transaction done by the interaction with a smart contract named *Article Factory*, which in turn instantiates an *Article* contract and creates one in the Ethereum network. Note that all metadata about the article itself is stored in storage variables in the *Article* contract, while the content of the article is stored in Swarm instead. Swarm is a distributed file storage system in the ecosystem of Ethereum distributed applications to complement the power of Ethereum Network [1]. Figure 2 below shows the position of Swarm and its relationship to Ethereum.

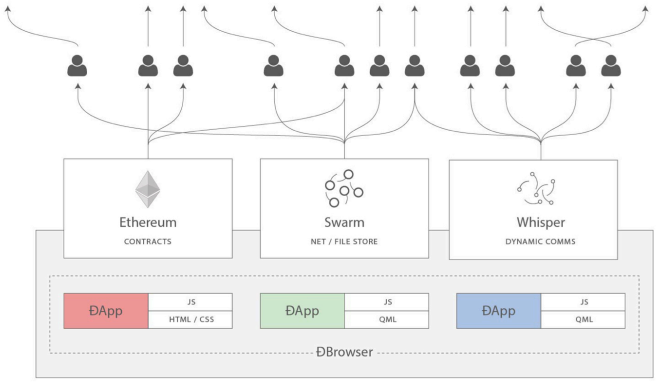


Figure 2. The distributed application supported by Ethereum, Swarm and Whisper

Ethereum itself focuses on smart contracts while storing massive data in it would be ridiculously expensive (in terms of gas), while Swarm aims to solve such issue by providing a distributed peer-to-peer network to persistently to store all the data objects. In Swarm, each data object (e.g. JSON data, image or document) is referenced by a unique 64-digit hexadecimal hash value, and by cryptographical hashing the integrity of the file content is guaranteed. Besides, the file objects on Swarm cannot be deleted which means Swarm shares the same level of capability of data tracking and transparency as Ethereum. Hence, Swarm becomes a reasonable choice to store our article content.

Below (Figure 3) is the workflow of publishing an article, where as you can see, it goes through both Swarm and Ethereum in the process.

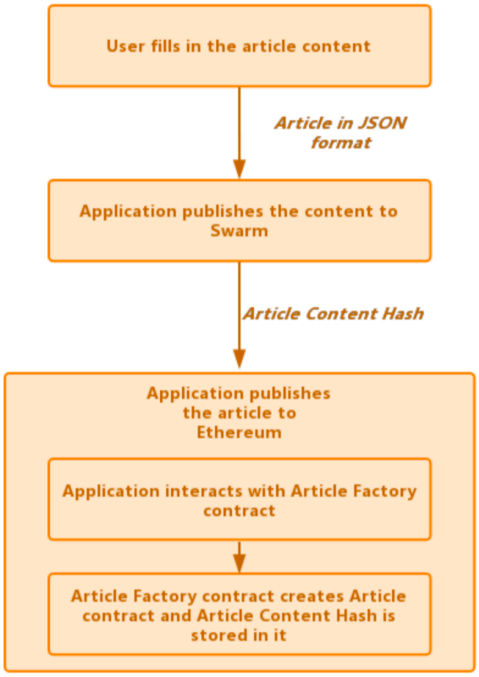


Figure 3. Application logic / workflow for article creation

Below (Figure 4) is a diagram that helps the audience better understand how Ethereum and Swarm play their role to replace traditional database and OSS. The blue side shows the traditional way of keeping articles and related data in database in OSS, and the orange side shows the brand-new way of storing articles in Ethereum and Swarm. It's remarkable that with the markdown language support in article body, image and video data are indeed compressed and encoded into the Swarm object (referenced by hash) together with the article content. For example, image is encoded into base64 string and becomes a *"img"* HTML tag in the Swarm object.

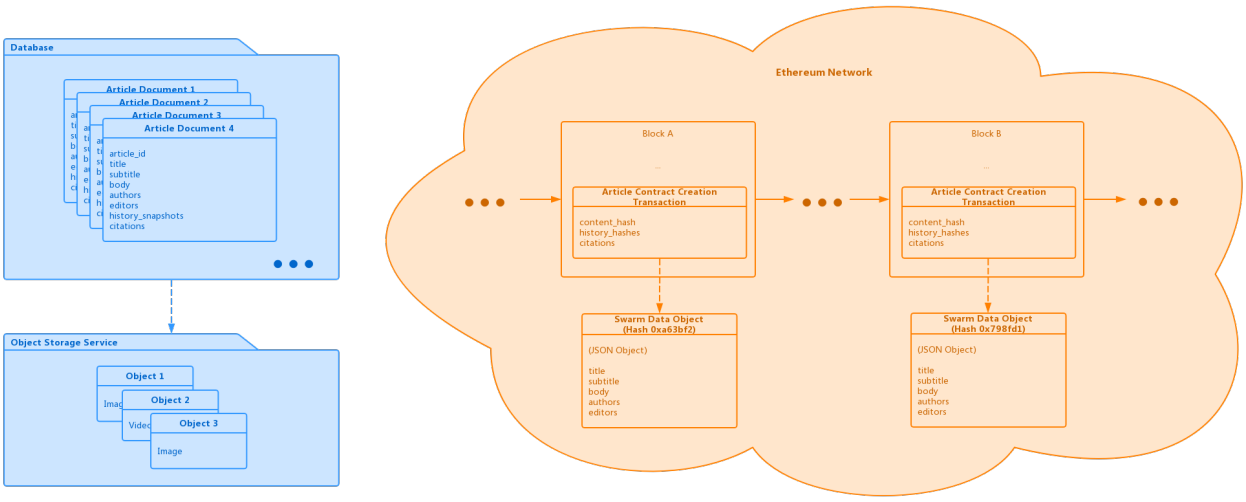


Figure 4. Traditional way and the new way of persisting articles data

After having the understanding of the implementations of article creating and persisting, now let's look at the graphic interface of such feature.

Below in Figure 5.1, as you can see it's a standard HTML form requesting users to enter content, and for the body part of the article is edited in an editor with markdown language support.

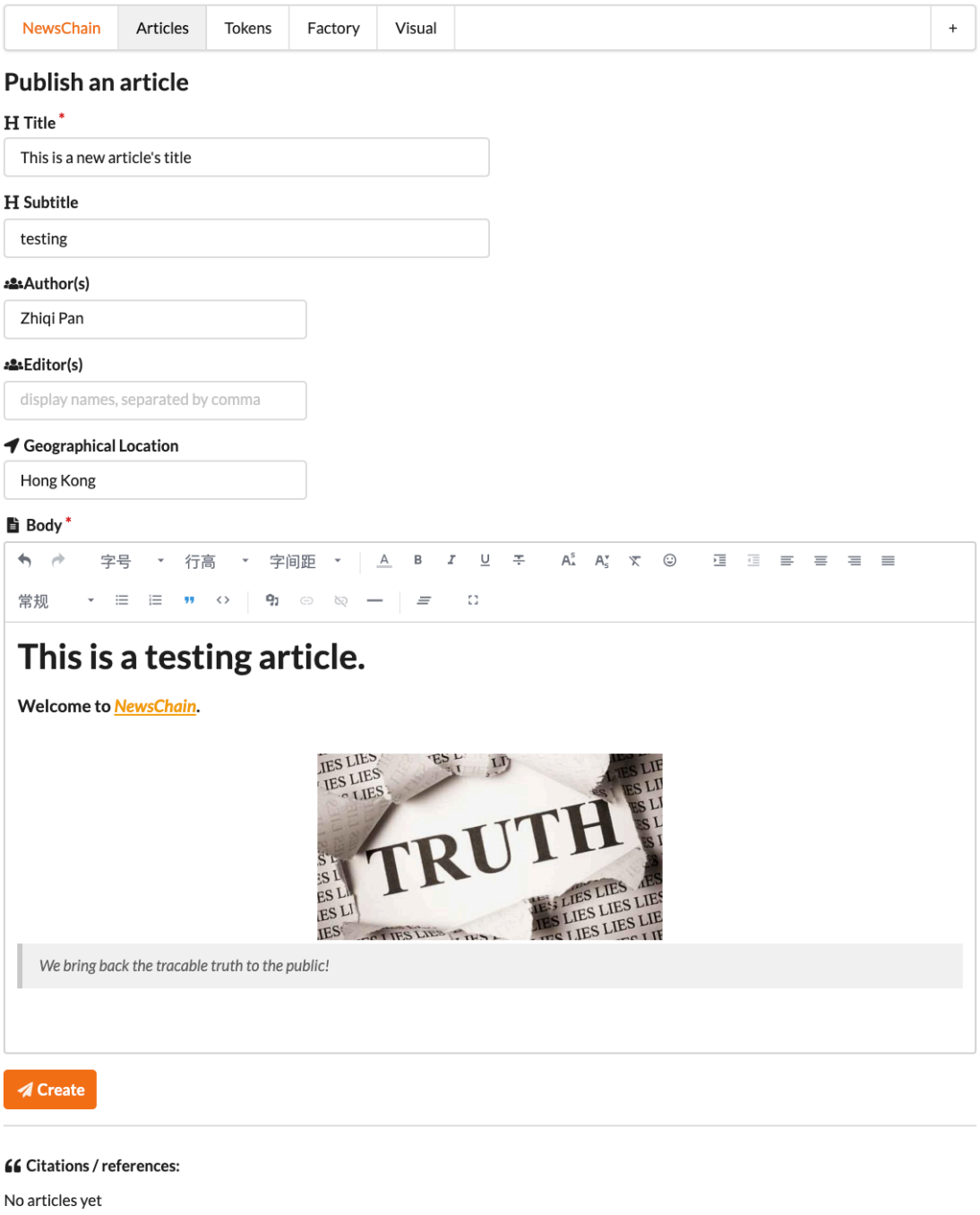


Figure 5.1. UI of creating a new article

Note that there's no citations for this article for demonstration, we will cover citations later in this chapter.

The article for demonstration in Figure 5.1 is published and is permanently traceable on Ethereum on block number *4277220* on blockchain with transaction:

0xa3e16b76b40a7bee3851835b3aa47c88cf543c94bc601d62766f3432eadc1004

And for the article content, it's available at Swarm hash:

9a0ab30e587c19b112aa892bb30b684b49614009ff32e087d0e3f6d7c7565505

The *Article* contract address is: *0xc66c56331291B75033Ecc5B7792Fb6006E730592*

With the above information, audience will be able to check this article on Ethereum and Swarm.

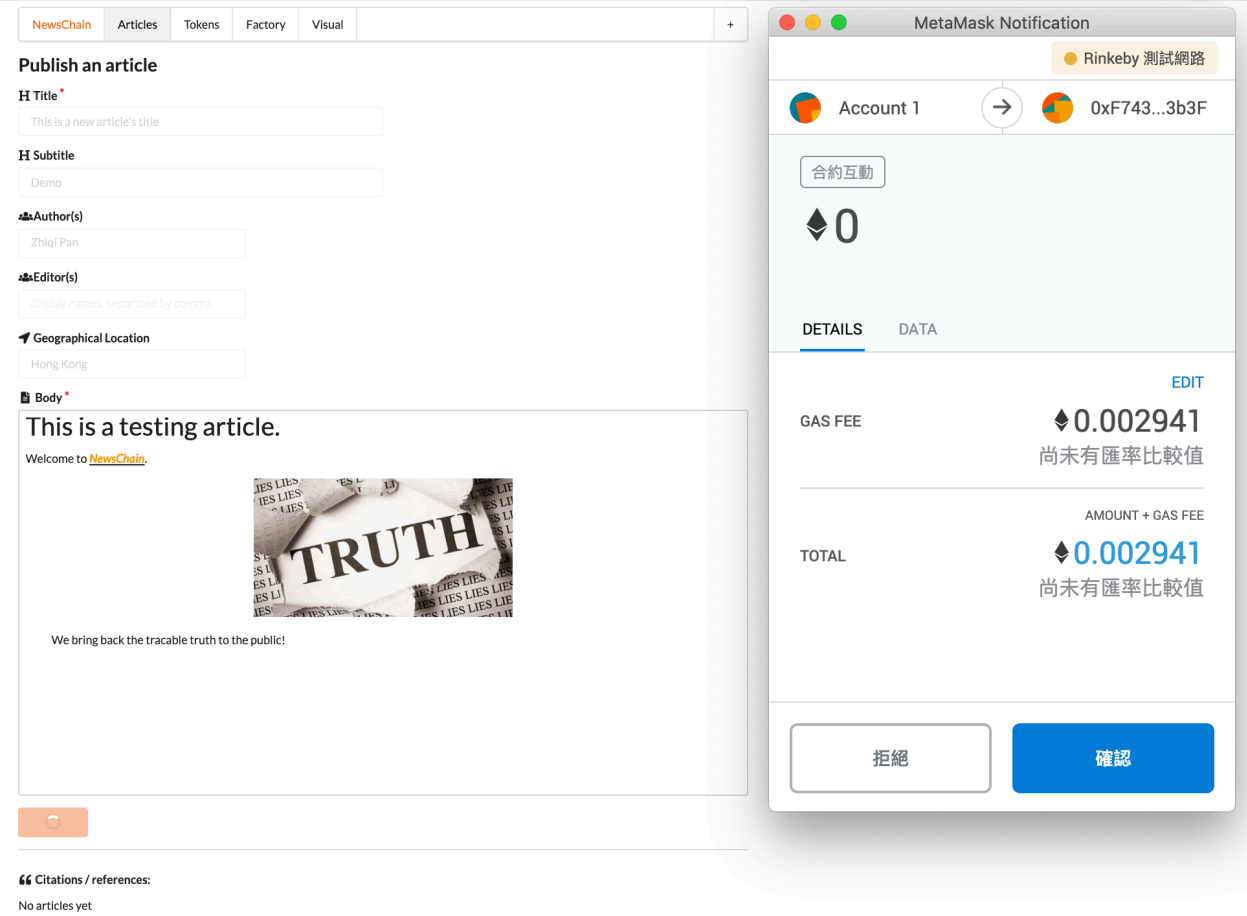


Figure 5.2. Confirming transaction of article publishing with MetaMask

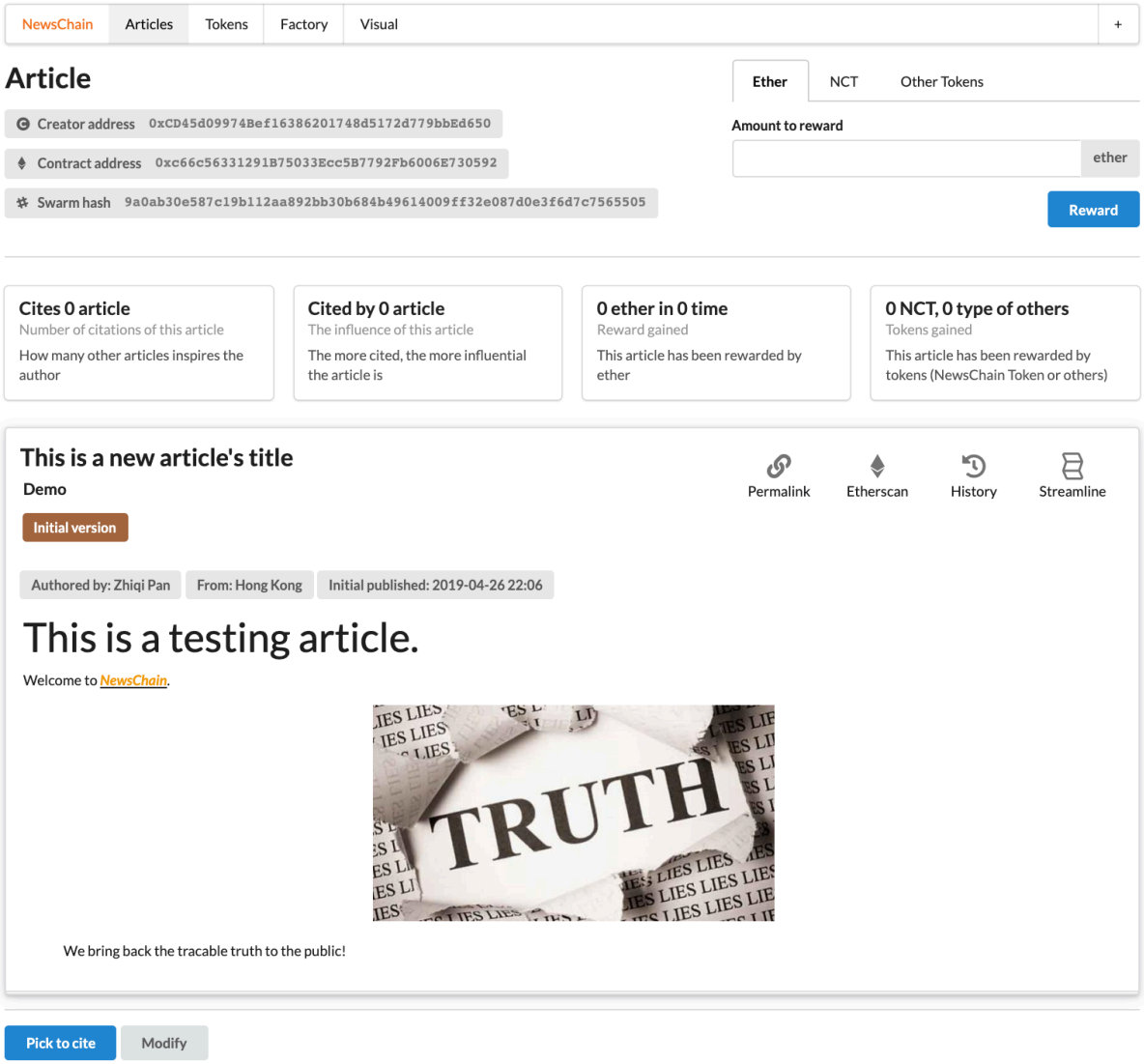


Figure 5.3. Published article shown in the NewsChain platform

Note that the brown badge in Figure 5.3 shows that this is the initial version of the article and the initial publishing time is also shown below.

Article Modification and History Tracking

It's noticeable that data on Ethereum network is not changeable, while this does not mean that our published articles cannot be modified in application level. NewsChain platform takes advantage of the un-tamperable nature of Ethereum to keep track of article versions while it uses Swarm to store multiple versions of article content. In other words, remember that our *Article* contract only tracks the article content hash but not the whole content itself, this means that the contract indeed is storing a pointer pointing to a Swarm data object. Hence, if we want to modify the article content, we only have to alter such pointer and keep the historical hashes stored in the contract. This way, multiple versions of article co-exists in the Swarm storage while the contract's *content hash field* points to the latest one, and its *history hashes field* (an array) points to historical versions of the article. Such design inherently solves the problem of recreating an *Article* contract for each time of modification. Ensuring that *Article* contract and the article itself have a one-to-one relationship is critical for easy tracking and tracing of the article itself and its citation relationship with other articles. Figure 6.1 below shows the UI for modifying an existing article on Ethereum, it looks similar to the one for creating but it does display the article address (*Article* contract address precisely) on Ethereum at the top. After modification, the article has one more version and the latest one (i.e. the second version is displayed by default in the view page), as shown in Figure 6.2.

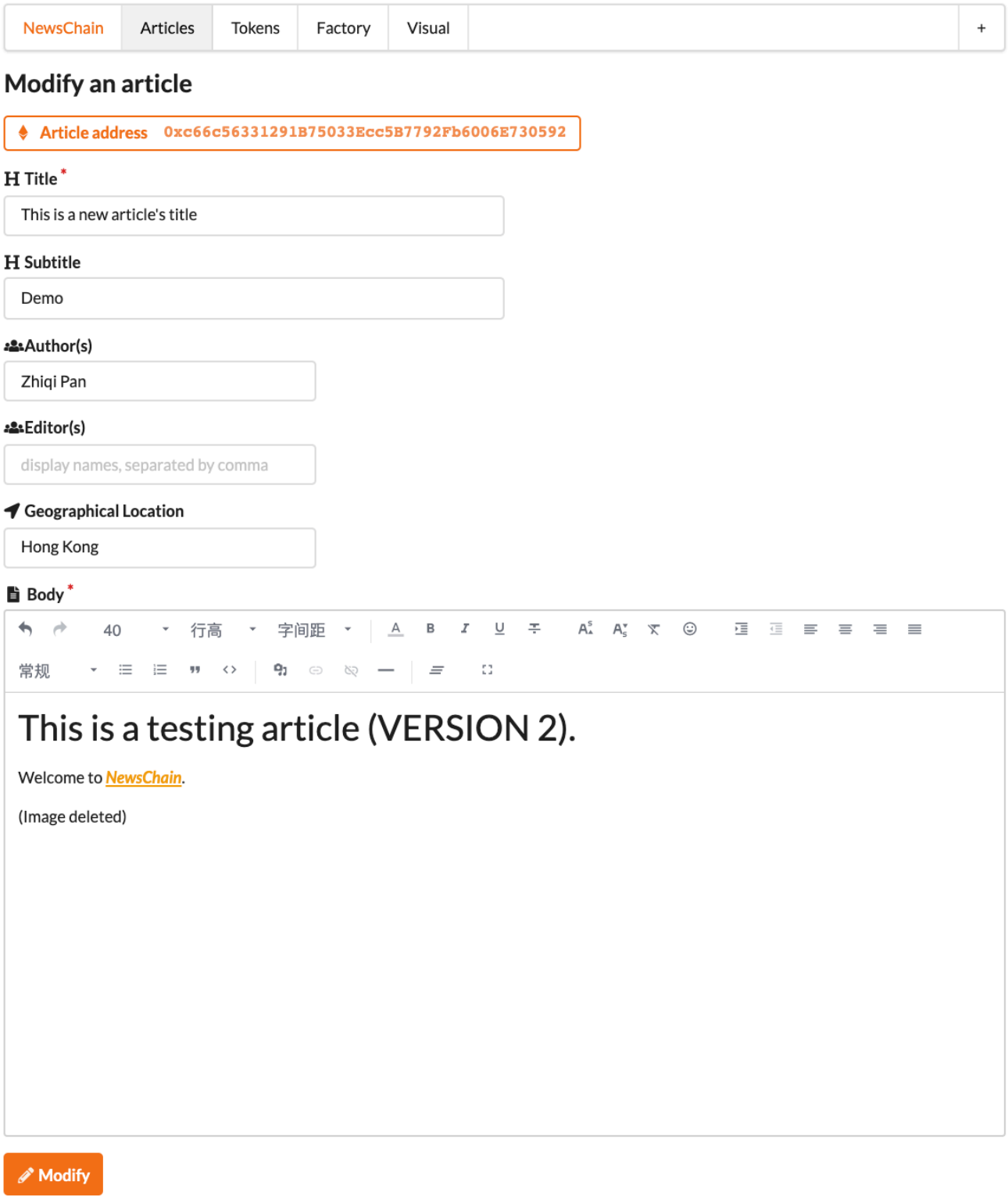


Figure 6.1. UI of modifying an existing article

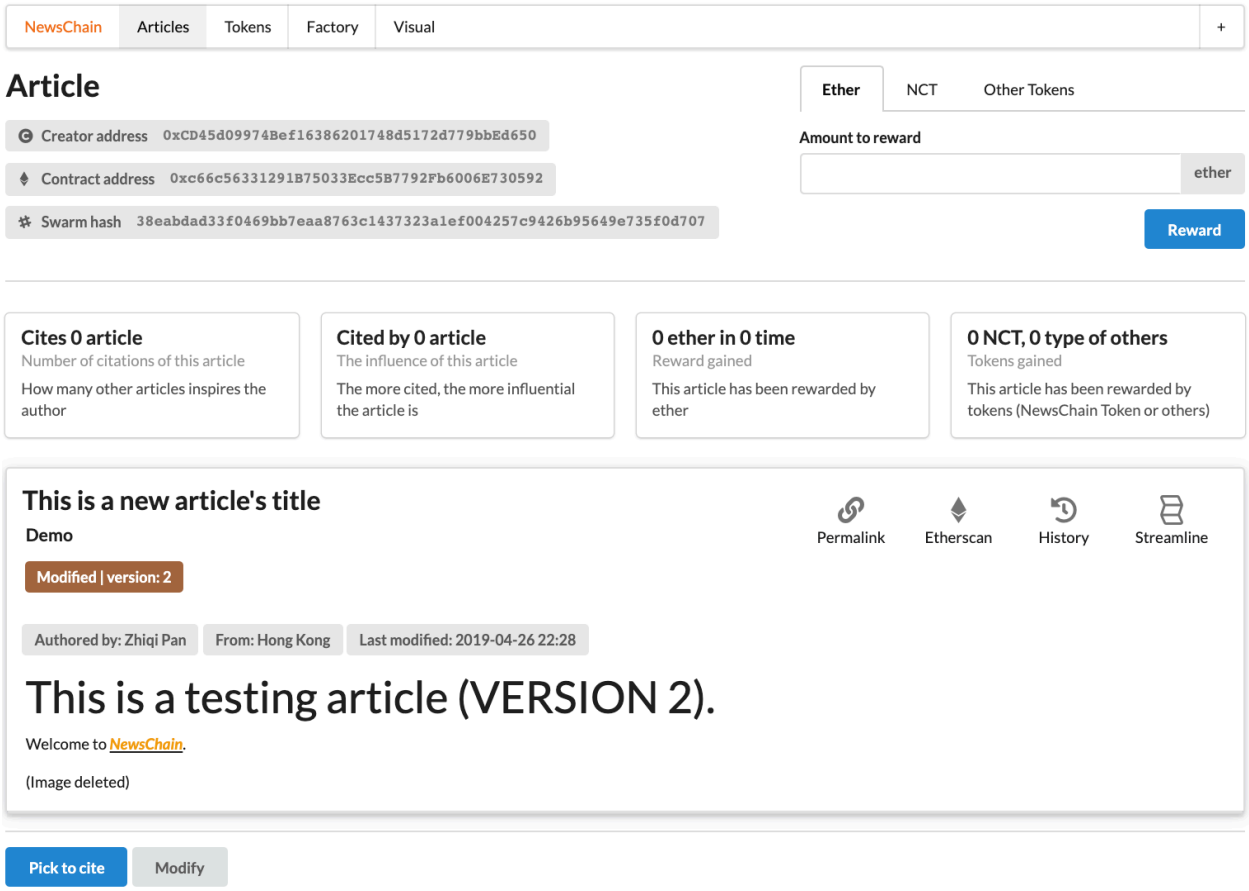


Figure 6.2. Showing the second version of the article

Note that Figure 6.2 compared to Figure 5.3, the brown badge of the article displays *"Modified | version: 2"* indicating that this is the second version, while Figure 5.3 shows that it's the initial version. Also look below, the audience shall see that the last modified time of the article.

The second version of the article content is available at Swarm with hash:

38eabdad33f0469bb7eaa8763c1437323a1ef004257c9426b95649e735f0d707

And it's confirmed on block *4277306* on Ethereum with transaction:

0x60265614c782d6db24894e755f418c1ef40e030a1a598b148440a811808edd63

Again, everything is permanently traceable and transparent to the public.

Then we click the History button at the top right corner of the article segment, and goes to the interface of tracking the history of the article, as shown in Figure 6.3.

Figure 6.3. History view of the article

As shown on the pagination bar at the bottom of Figure 6.3, there are two versions in total, and by default the viewer views the latest version (on the second page). By clicking page 1, the initial version of such article will be shown, as presented in Figure 6.4.

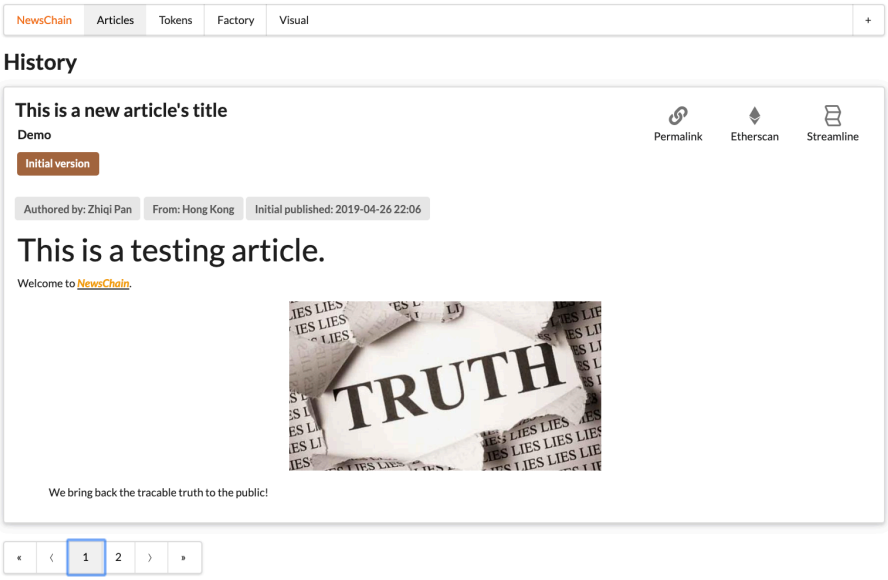


Figure 6.4. Navigating back to page 1 to see the initial version

Note that the modification can only be done by the original publisher of the article, or in other words, the address that published the article, and this authenticity will be checked in the smart contract. We will talk about such authenticity checking later when we talk about the system internals. Note also that for each article, there is no limitations on how many times it can be modified by the author, and all historical records will be persisted in the corresponding *Article* contract.

Article Citation

This section will talk about the citation of articles, which is one of the core features of NewsChain platform. To pick an article, user simply clicks the *"pick to sign"* button on article view page and then goes to create a new article. Note that user can

pick multiple articles to cite if necessary, as there's no limitation on the number of citations an article could have.

Citation is tracked within the *Article* contract, and actually for the *Article* contract there is a field named *citations* of array type that stores all the addresses (contract addresses precisely) of other articles the article cites. For easier understanding, you can simply consider it a linked list for now, and such linked list records the citation chain of multiple articles.

Figure 7.1 below demonstrates the interface of citing an article when the user is authoring a new one.

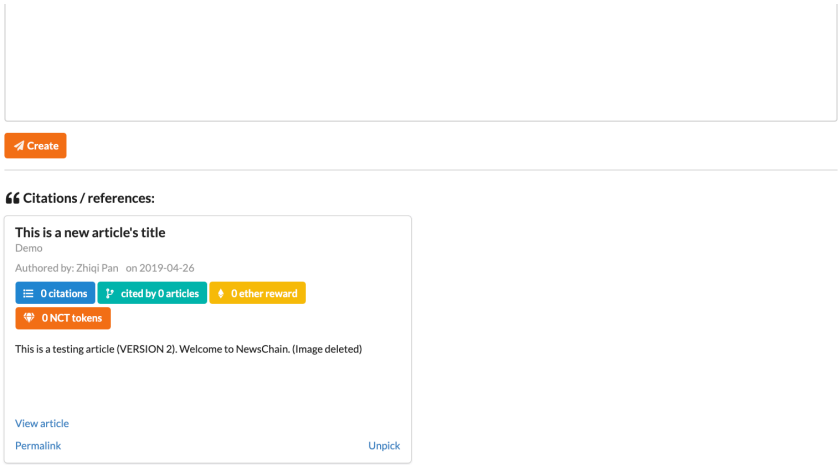


Figure 7.1. Creating a new article citing another one as reference

Figure 7.2 below shows the interface of viewing a newly published article which cites another article we published earlier. In the figure, it shows the summary of the article it indicates that the article cites 1 other article.

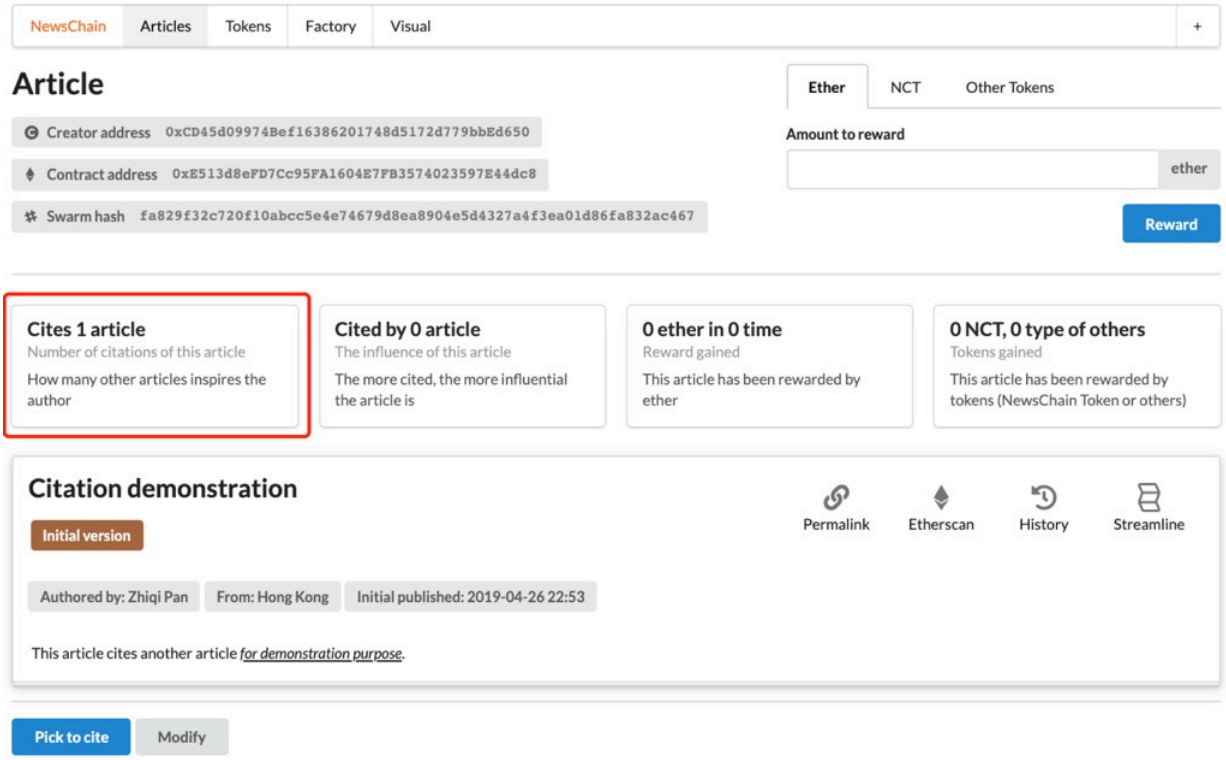


Figure 7.2. View the new article which cites another one

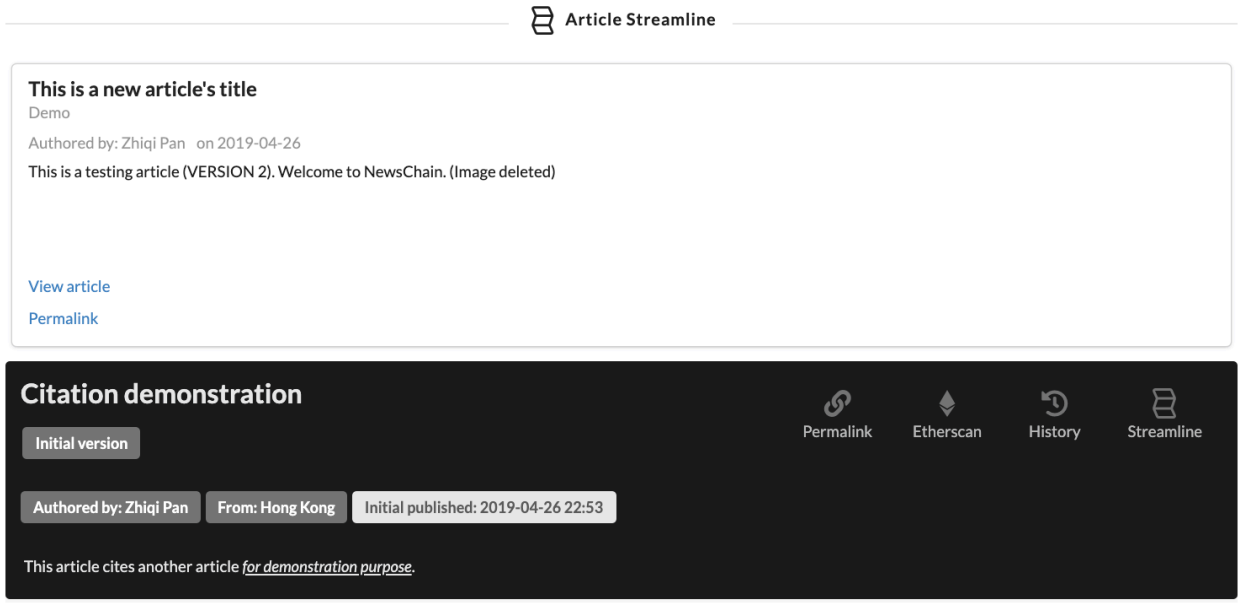


Figure 7.3. View the citation in a streamline view

Figure 7.3 above shows the streamline view of the new article, where on the top of it is the cited articles of itself.

For reference again, the new article is available on Swarm with hash:

fa829f32c720f10abcc5e4e74679d8ea8904e5d4327a4f3ea01d86fa832ac467

And it's confirmed on block *4277403* on Ethereum with transaction:

0x48587ee3a8fed2dc0fb8ae31ba2752b4a7d198d11c22b321c071fd7aab24e57f

Interested audience are encouraged to check the Ethereum Network and Swarm storage with the provided address and hash for a deeper and better understanding.

Article Rewarding by Ether

Another interesting feature NewsChain offers while the traditional platform does not is the rewarding feature. Any viewer with an Ethereum account can reward the article author with Ether. This feature is fully implemented within Ethereum and it by design inherently ensures anonymity and convenience [2]. Figure 8 below shows the workflow of rewarding an article with 1 ether.

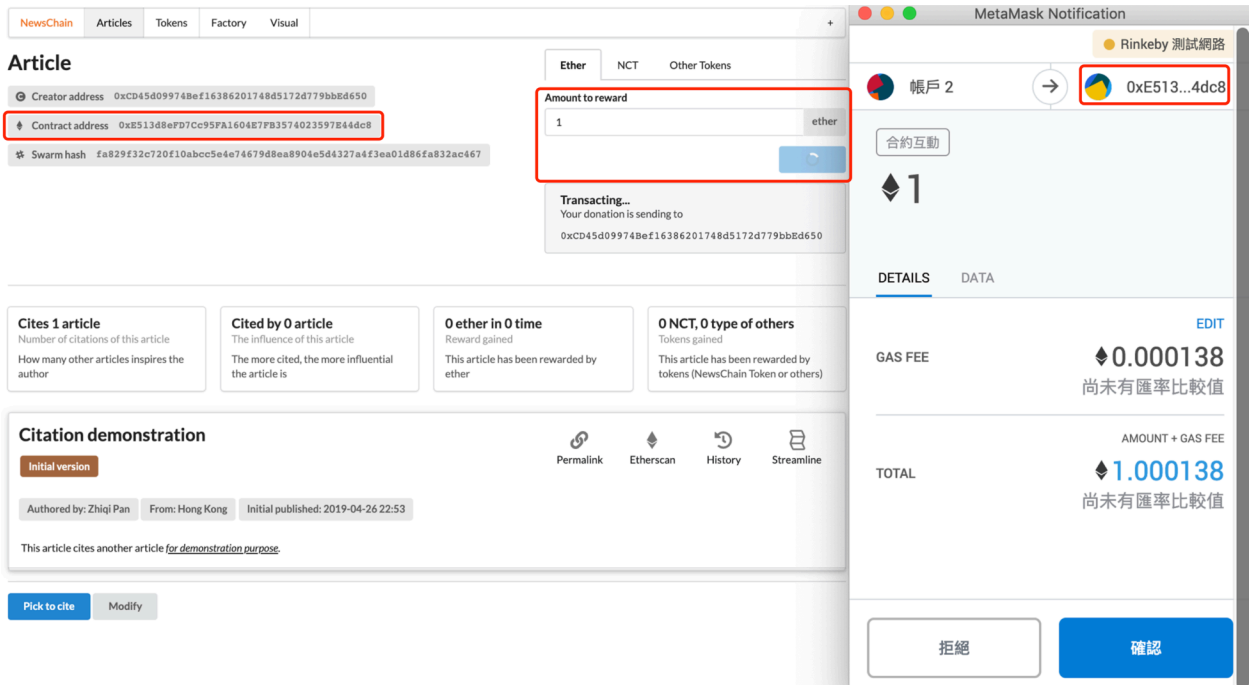


Figure 8.1. Rewarding one ether to the article author

As highlighted in red rectangle in Figure 8, it's noticeable that ether rewarding (or transferring) is done via the smart contract instead of directly being transferred to the recipient's account, this is implemented on purpose to ensure to the transparency and traceability of such transaction [2]. Once the contract receives the ether, it then initiates an internal transaction to send all the received ether to the dedicated recipient (usually the author, but this could be altered by the author itself). This way, for one thing, the *Article* contract itself can check how much ether is received from the others, and for another thing, this protect the ether sender from sending the ether to a wrong person accidentally.

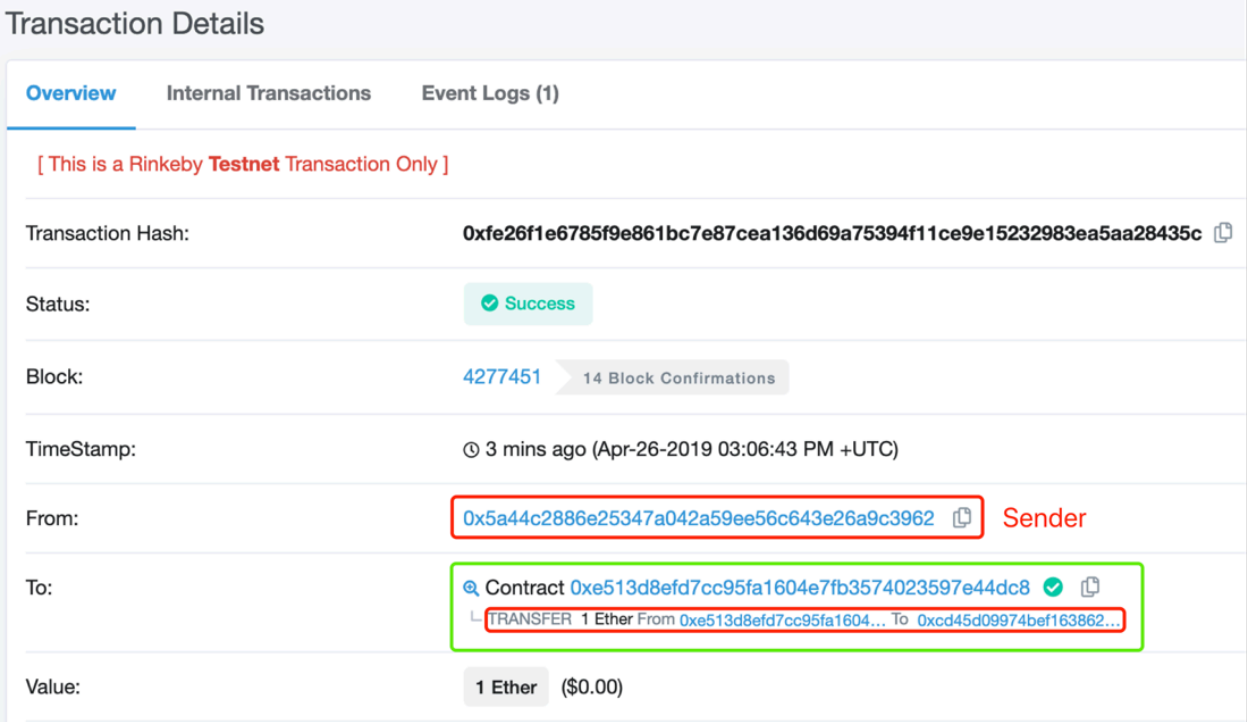


Figure 9.1. Internal contract transaction for sending the received ether to the reward recipient

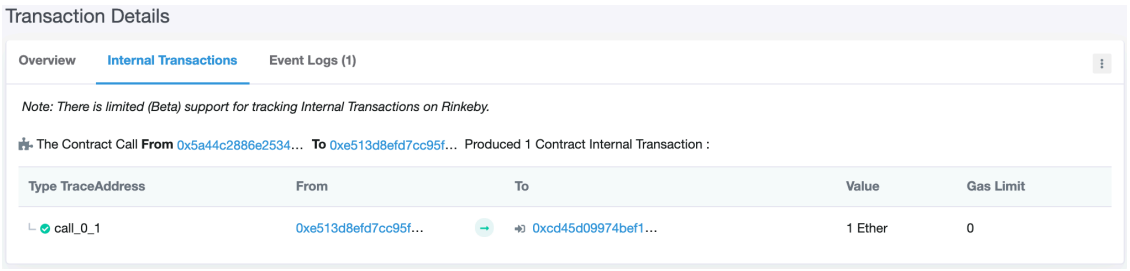


Figure 9.2. The detailed look at the internal transaction for transferring ether

NewsChain Token and Article Rewarding by ERC20 Tokens

Besides rewarding an article by ether, NewsChain also gives more flexibility to the users by support rewarding articles by ERC20 tokens [3]. NewsChain also offers its

own token named NewsChain Token (NCT), which has full compliance to ERC20 token interface and supports all the basic and advanced features of Ethereum standard token. This makes NCT tradable with other ERC20 tokens on the market on Ethereum. NCT is run on *NcToken* smart contract.

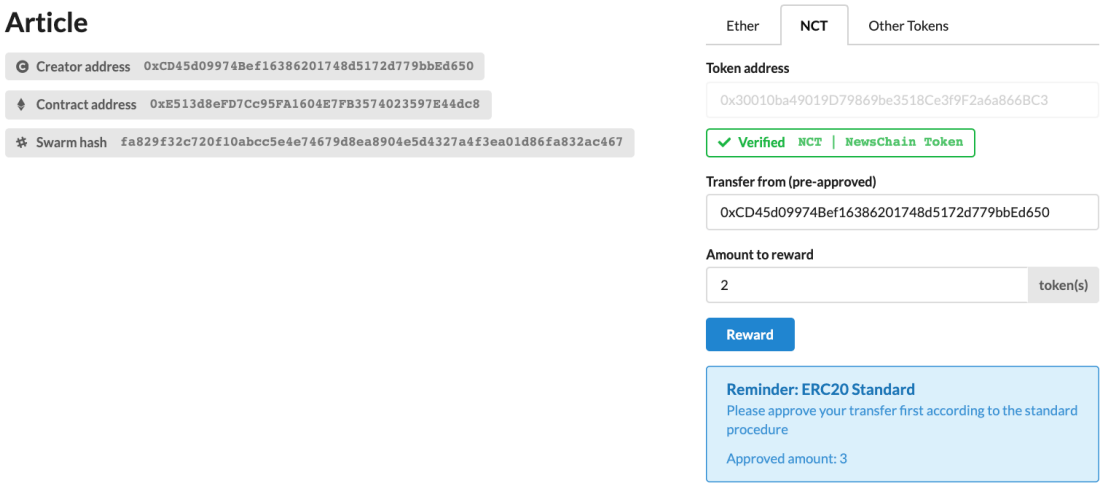


Figure 10. Rewarding an article with 2 NCT token (already approved)

Figure 10 above shows how we could reward an article with 2 NCT tokens. However, rewarding is not always this simple by design, as stated in the ERC20 standard, tokens can only be sent to the recipient with an amount under the pre-approved amount. This specification is necessary for token trading and exchanging for goods in the market, and NCT implements accordingly. Let's now look at a case where we have to approve the reward first according to the ERC20 standard before rewarding, as presented in Figure 11 below.

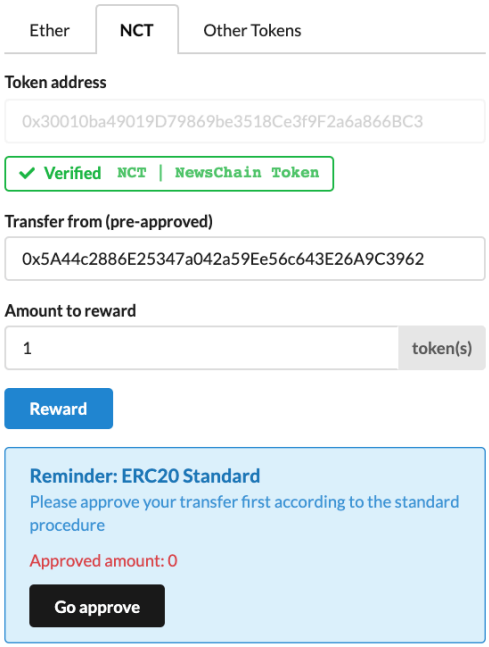


Figure 11.1. Rewarding could fail since the approved amount is zero

Note that the user could still proceed to give reward to the article regardless the warning, while this could fail and the gas used for such transaction would be refunded. To facilitate the user experience, we do a prerequisite checkup before user actually sending out reward to prevent unnecessary waste of gas.

Figure 11.2 below shows the approval procedure for the token rewarding, Note that the balance of the user who wants to send reward is 5 NCT, while the user has not yet approved for any amount to be transferred to the article reward recipient. We have input a value of 3 to be approved and will proceed, and the result of such approval will be presented in Figure 11.3 as following.

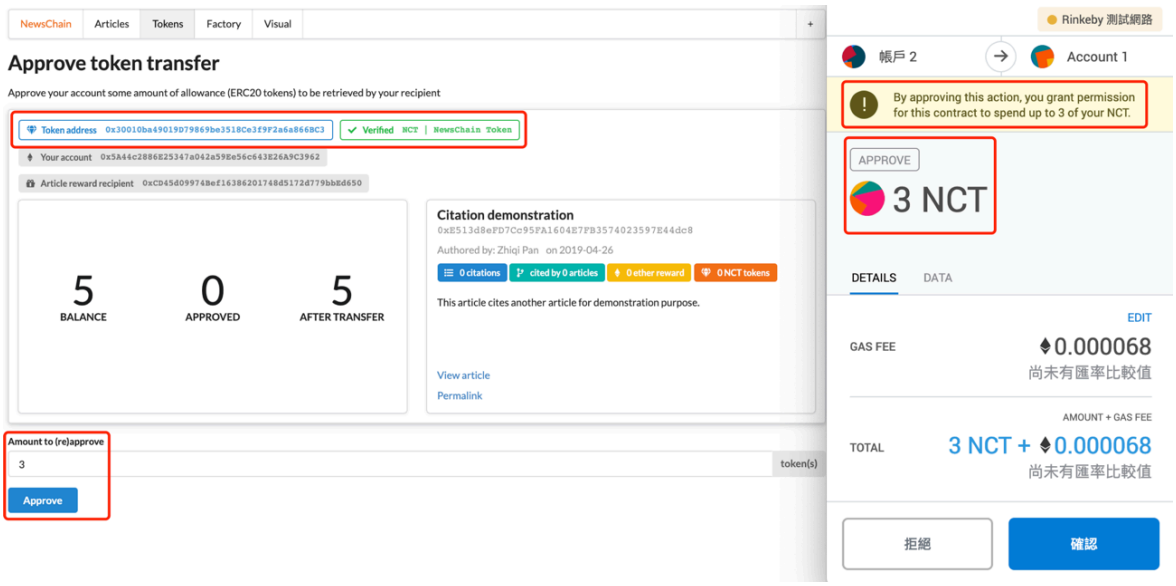


Figure 11.2. Approving for 3 NCT for the article reward recipient

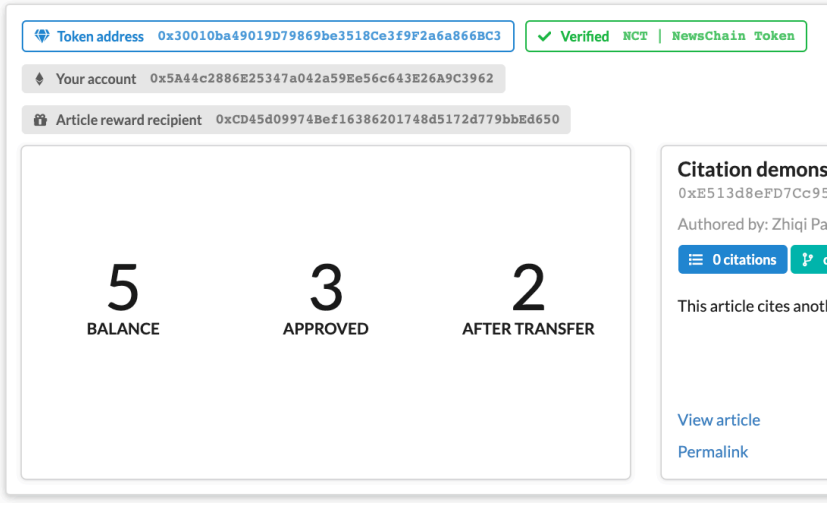


Figure 11.3. Approved 3 NCT for the article reward recipient

After approving 3 NCT tokens, we now should be able to reward the article recipient with our current account, and the NCT token rewarding procedure is done in Figure 11.4 below.

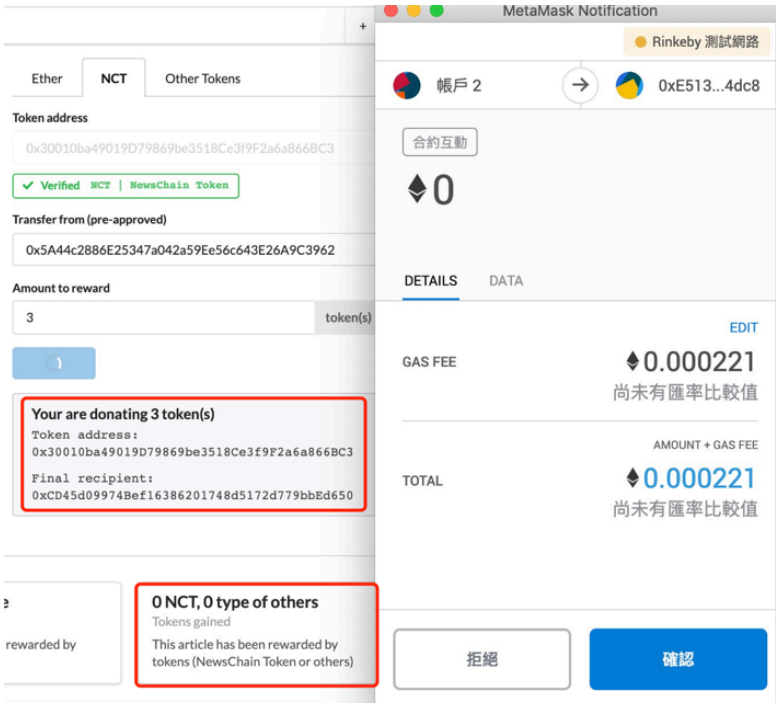


Figure 11.4. Rewarding 3 NCT to the article after sender's approval

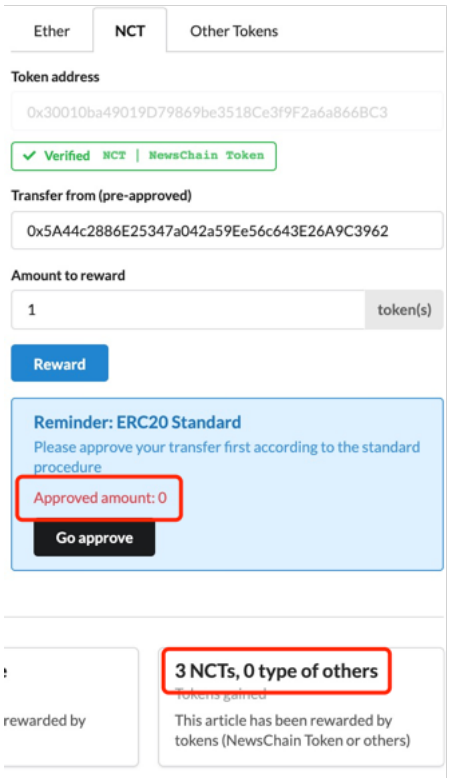


Figure 11.5 After rewarding completed

As you can see in Figure 11.4 and Figure 11.5, once the transaction is completed the approved amount becomes back to zero as all quota of approved amount is used, and the *"tokens gained"* display for the article changes from 0 to 3 NCT tokens.

Similar to rewarding with ether, such rewarding goes through the *Article* contract instead of going directly to the recipient for the same reasons aforementioned.

Besides, for the convenience sake, we also implement a page (shown in Figure 12 below) for transferring, approving and spending any tokens that comply to ERC20 standard. This means that not only NCT could be dealt with, but any other ERC20 tokens will also be handled properly and elegantly.

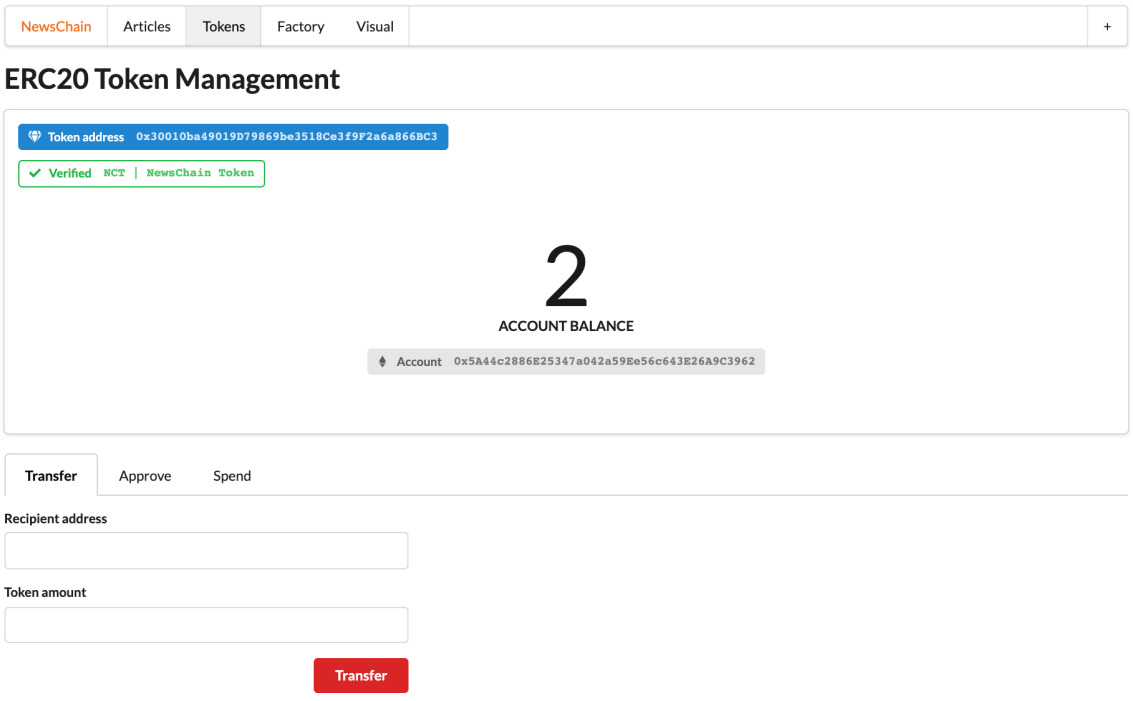


Figure 12. Interface for dealing with any types of tokens that are ERC20-compliance

Citation Auto Reward and NCT Token Initial Coin Offering (ICO)

For all the ERC20 tokens, they have to deal with Initial Coin Offering (ICO) procedure for one or multiple times, and this holds true as well for our NewsChain Token (a.k.a. NCT). However, while normally ICO are done by selling some number of tokens at a soft cap at some price of ether, our platform invents a new mechanism, citation auto reward, to offers our initial coins (tokens). Traditional ICO is transparently controlled by the related token ICO contract, while the mechanism NewsChain invented gives the power of the system manager (i.e. the creator of the *Article Factory* contract) to control the process of ICO.

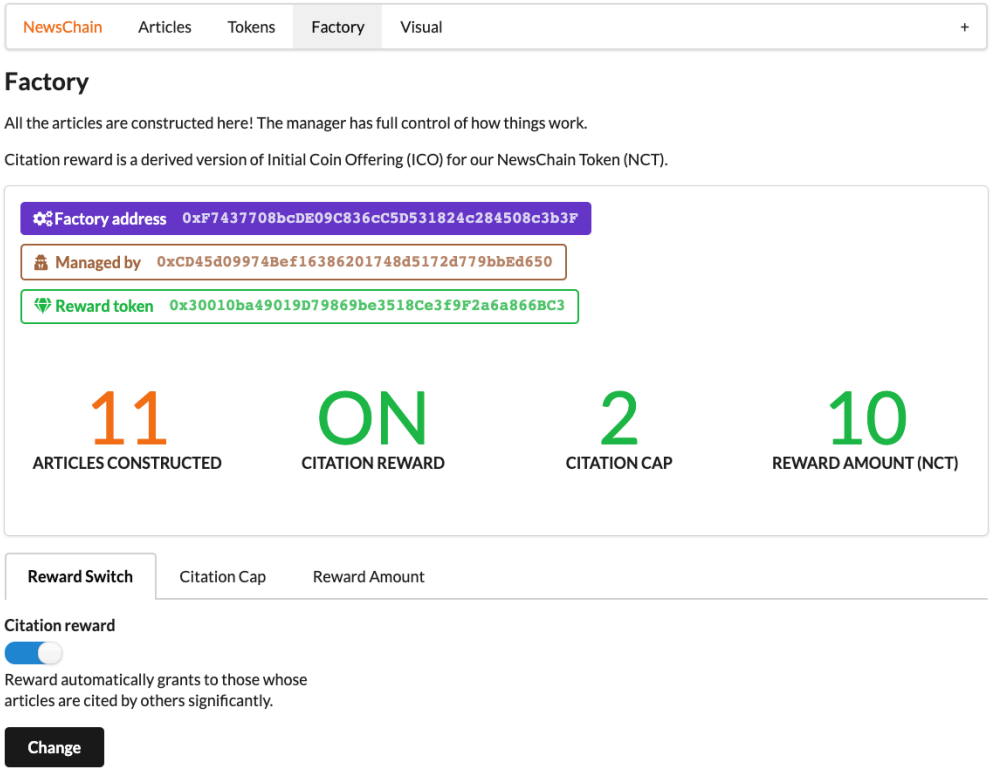


Figure 13.1 Managing citation auto reward (NewsChain Token ICO) with a reward switch controlling the on and off ICO

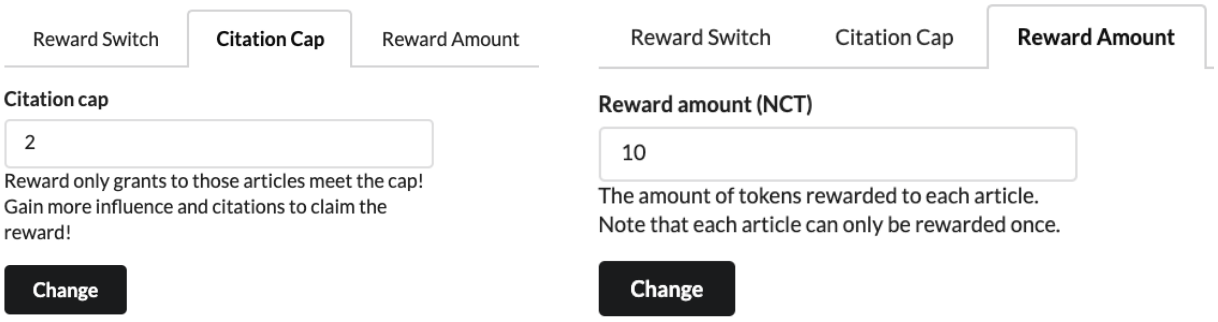


Figure 13.2 Managing the citation cap and reward amount

Figure 13.1 and Figure 13.2 above demonstrate how a manager can control the citation auto reward process, or namely the ICO process.

Citation auto reward is a smart contract behavior that when an article is first time cited by a number of other articles, it receives an amount of NCT automatically issued by the *NcToken* contract as reward. The number of citations an article should gain before getting offered NCT tokens is defined by a parameter called Citation Cap, and the amount of NCT tokens rewarded is defined by another parameter called Reward Amount. And similar to the fact that there's a limited coin offering period in traditional ICO process, NCT's ICO can be turned on and off by the manager. And the ICO is only effective when the switch called Reward Switch is turned on.

To sum up a bit of what we just talked about formally, we list out the three ICO parameters we defined as follows:

1. Reward Switch;

2. Citation Cap; and
3. Reward Amount

Besides, when we design the system and ICO process, security is always kept in mind. To ensure that only the system manager, or Article Factory contract creator itself can modify these ICO-related parameters. Our smart contract uses corresponding method modifier to check the identity of the transaction sender, and the modification of these parameters will be proceeded if and only if the identity matches the creator's.

To demonstrate the authenticity check, Figure 14.1 shows a failed transaction in which a non-manager account is trying to modify the ICO parameters.

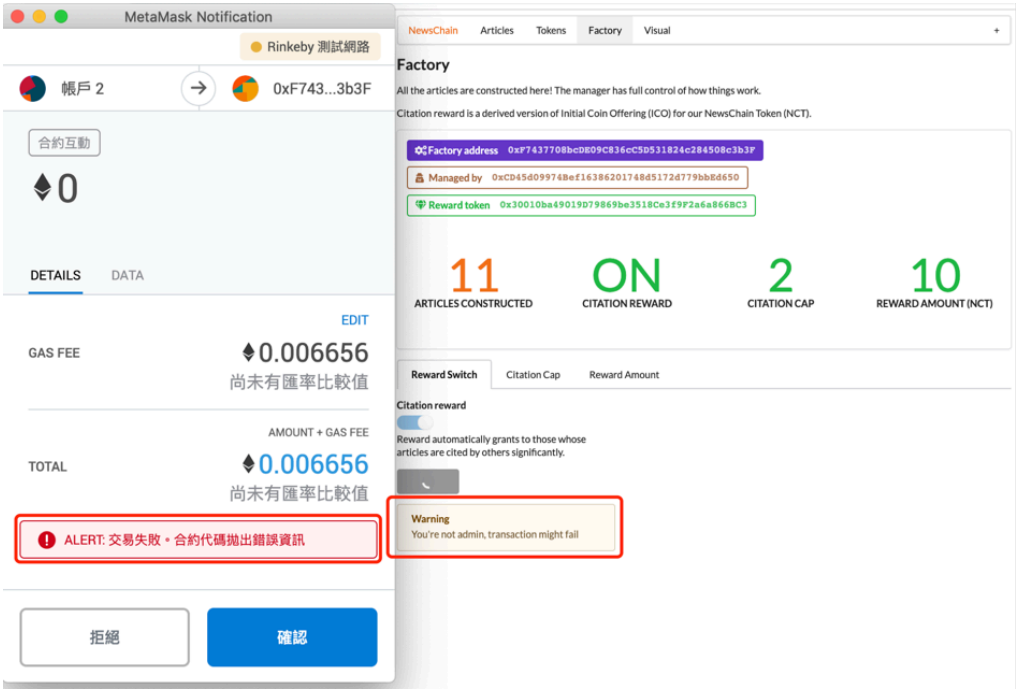


Figure 14.1. Alerting and warning about a potential failing transaction of modifying an NCT's ICO parameter Reward Switch

Note that even though warning and alert are displayed to the user, the user can still proceed the transaction regardless of them as they are only a frontend checking. The real checking is on the smart contract itself, and the failed transaction will finally occur if the user insists to proceed, as shown in Figure 14.2 below (such failed transaction does not refund all the consumed gas so it's unworthy for a user to proceed such impossible modification).

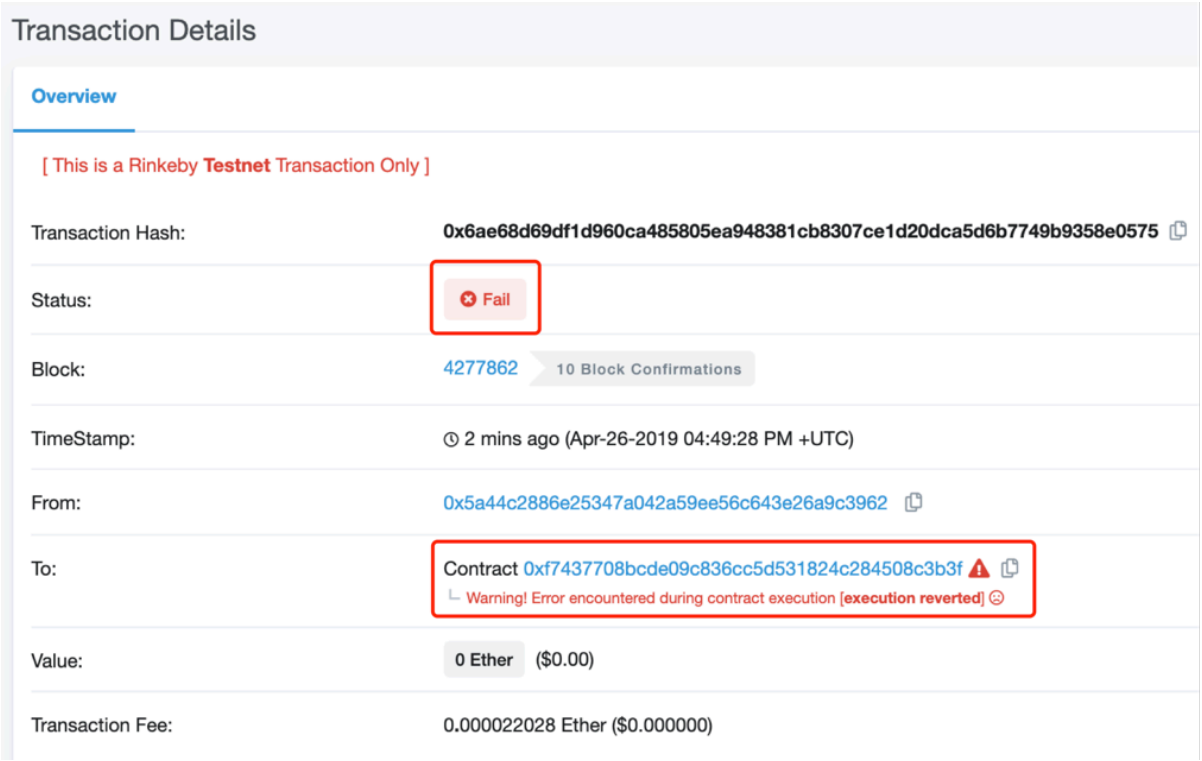


Figure 14.2. Transaction failed and execution reverted

On the other hand, with the manager account, the ICO parameters could be changed properly, and this is presented in Figure 14.3 below. Once the citation auto reward

mechanism is off, no more coin will be offered to any articles, no matter how many other articles cite an article.

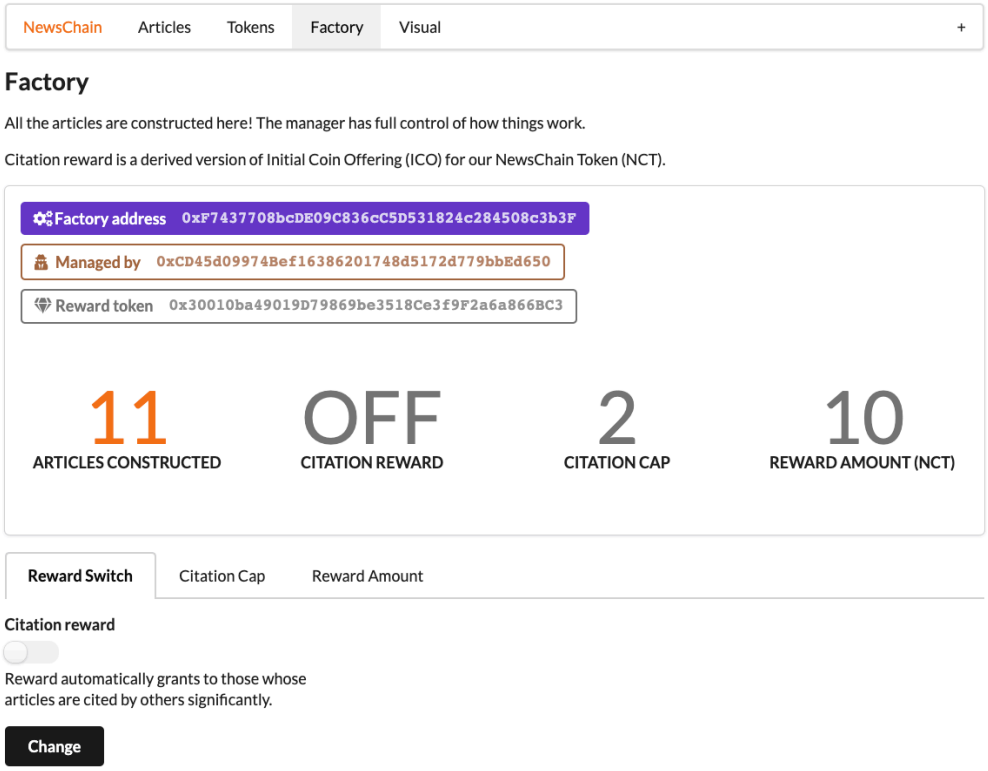


Figure 14.3. Manager successfully switched citation auto reward off

Citation Relationship Visualization

Last but not least, another major feature of the platform is citation relationship visualization. As this platform emphasizes on news co-authoring, citation makes significant contribution to co-authoring so we believe a user-friendly visualization of citation relationship would be necessary.

Our data processing and visualization both support bi-directional relationship, and this is natively supported by our *Article* contract with a *citedBy* field. This could be considered as maintaining a bi-directional linked list of citing and being cited by. The citedBy field has access control and can only be invocated by other *Article* contracts while direct invocation from human users or the system manager would be denied. This access control ensures the integrity of the bi-directional linked list and prevents it from corruption. We will cover about this in detail when we talk about contract code in System Internals section later.

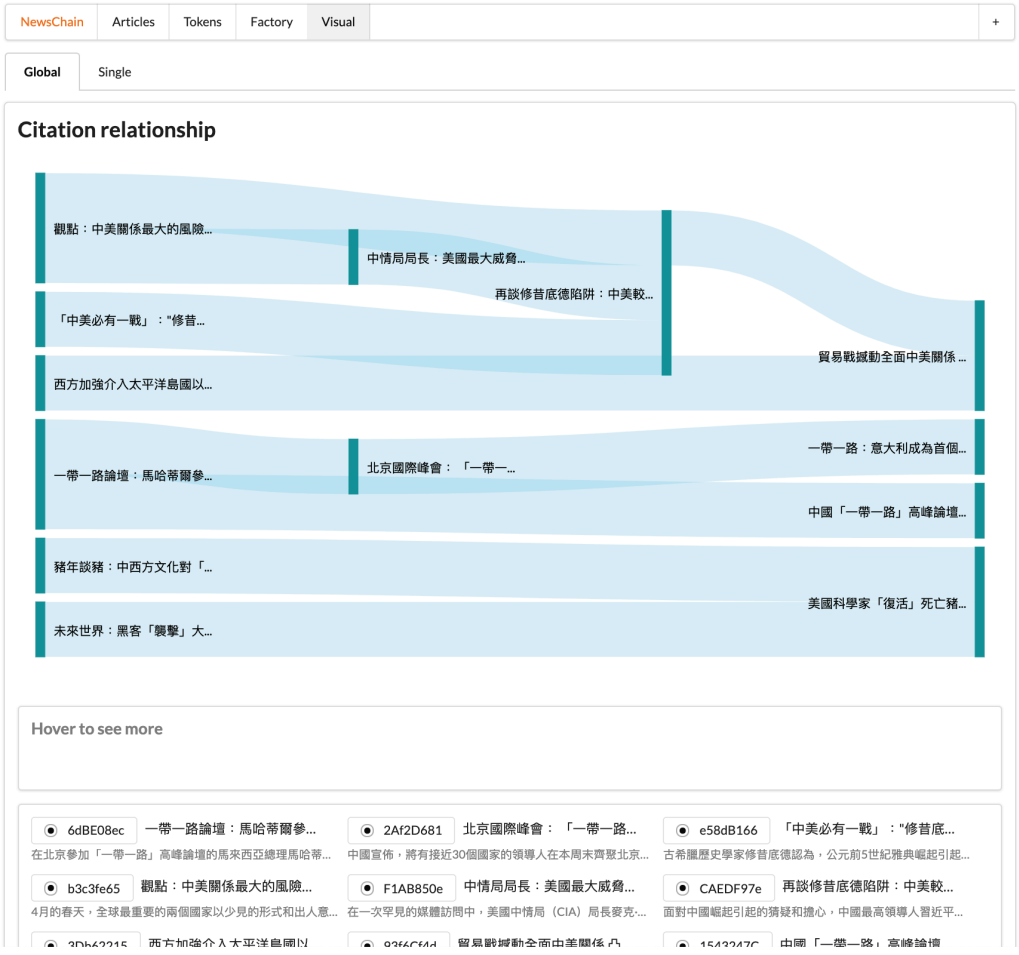


Figure 15.1. Visualization of citation relationship among all articles

Figure 15.1 above shows the visualization of citation relationship among all articles. We have a control segment for selecting which articles should be involved in the citation relationship diagram below, as shown in Figure 15.2.

Figure 15.2. Only viewing relationship among selected articles

Before visualizing, we apply graph algorithms to model the relationship. Such model helps us better generate and remove linkage between particular articles. For example, once an article is deselected from the list, several other articles related to it should be removed too if those articles become isolated in the graph. Since articles in citation relationship have complex linkage between each other, the advanced graph algorithms play a key role in not only data simplification, visualization but also in performance improvement. In the list shown in Figure 15.2, those articles checked with a black dot on the left side (selected) but without a gray rectangle around it, indicated by blue color, are isolated in the graph model, and they should no longer be shown in the relationship diagram even though they are selected by the user. Those indicated by red color are those removed (unselected) by the user itself, while those indicated by blue color are passively removed by our real-time updating graph algorithm. Note that those indicated by green color are not unselected by the user and not removed by the graph algorithm either, so they will be part of the constructed linkage graph of citation relationship.

The following code snippet is the pseudocode in JavaScript for computing the initial graph of the articles according to their citation relationship.

```
function computeInitialGraph(articles) {
  const inputAddresses = articles.map(a => a.address);

  function hasCommon(array) {
    return array.filter(addr => inputAddresses.includes(addr)).length > 0;
  }
```

```
  // calculate isolated articles in the graph
  const isolated = articles.filter(a => !hasCommon(a.citedBy)
&& !hasCommon(a.citations));
  const articles = articles.filter(a => hasCommon(a.citedBy) ||
hasCommon(a.citations));
  const articleMap = {};

  // build the article map from article array
  articles.forEach(article => {
    const address = article.address
    articleMap[address] = article
  });

  // build nodes in the graph
  const nodes = Object.keys(articleMap);

  // build links in the graph
  const links = [];
  Object.keys(articleMap).forEach((address, i) => {
    articleMap[address].citedBy.forEach(other => {
      if (articleMap[other]) {
        links.push({
          source: i,
          target: articles.map(a => a.address).indexOf(other),
        });
      }
    });
  });

  return { nodes, links, articleMap, isolated };
}
```

We call the citation relationship among multiple or all articles as global view. And besides the global view, we also support viewing the citation relationship for a single article, which is a single view and can be considered as an edge case of the global view. Figure 16.1 and Figure 16.2 below show the citation relationship for an article (single view). In particular, Figure 16.1 shows that the selected article cites / references another article (indicated by the highlighted yellow linkage), and Figure

16.2 shows that the selected article gets cited by another article (indicated by the highlighted orange linkage).

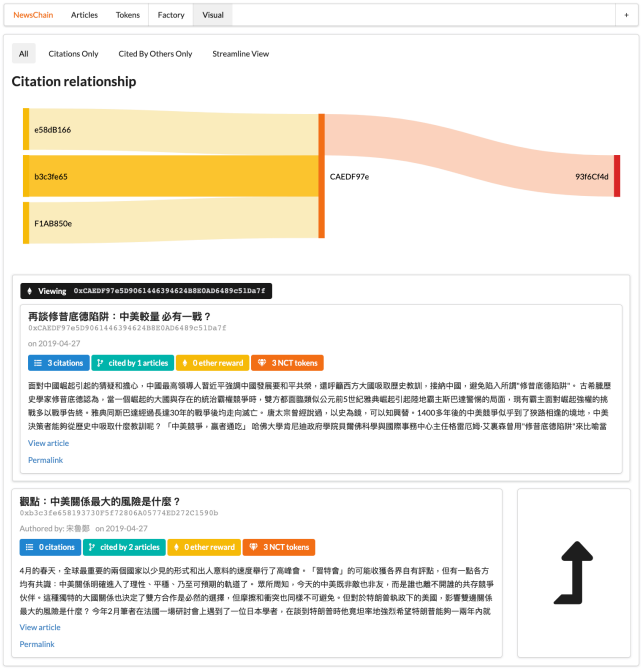


Figure 16.1. Citation relationship for the selected article (citing another one)

Figure 16.2. Citation relationship for the selected article (being cited by another)

Figure 17. Streamline view of an article with all other articles that it cites or that cite it

The single view is centered on one article as shown in Figure 16, and it helps user better investigate on the origin and derivation of a single article. For example, in the figure we can see the selected article cites (references) three other articles and is later cited by another article.

When viewing for only one article, streamline view can be triggered to have a better streamline-based (sometimes timeline-based) flow of related articles centered on the selected one. Figure 17 in the next page demonstrates the streamline view of the article shown in Figure 16. In such view, user can see all the three citations the article has clearly in one glance, and the user can also view an article that is derived from it. This page also exploits *focus mode* by showing badges and tags in grey-scale colors and remove unnecessary information, and for the selected article we want users to focus on it so dark color is used. The settings of such focus mode could be changed in setting above the visualization page, as shown in Figure 18 below.

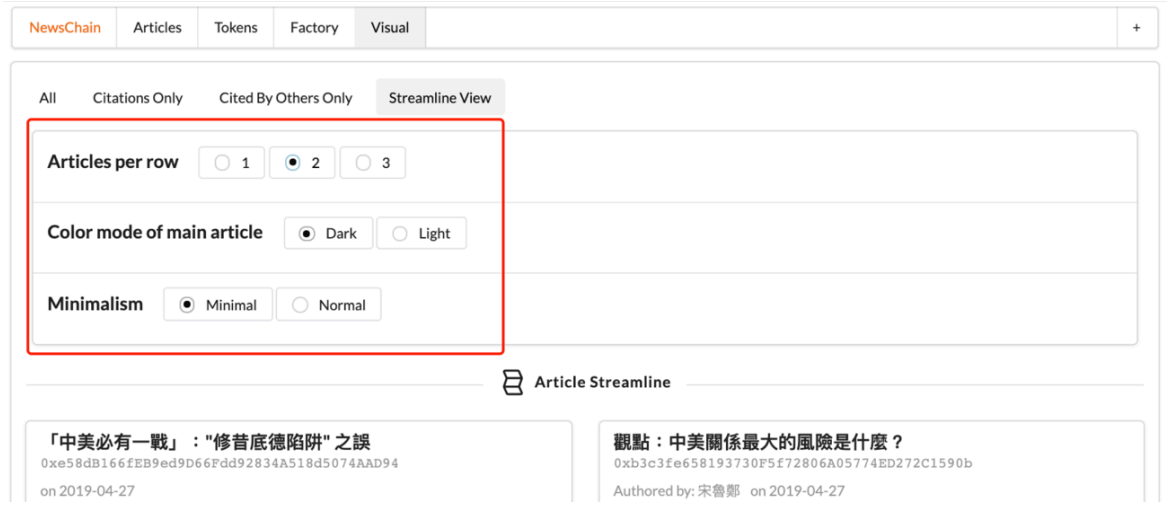


Figure 18. Control panel of the focus mode in article streamline view

Frontend Development

We have seen lots of core features of NewsChain and their implementations behind the scene, but please be noted that without proper frontend development, all things happening in the backend and service level would be visible to the clients. In this project, the author not only works on the backend development but also the frontend development as well. All the graphical interfaces you have seen in the previous chapter are built by the author on his own using ***React.js*** from scratch, and this chapter will focus on all the frontend development work in NewsChain.

Server-side Rendering

We apply the server-side rendering technology after careful considerations. The client-side rendering was tested out for a while during last year, but finally got abandoned due to performance issue when integrating Ethereum and Swarm, as well as the complexity of client-side routing issue in a progressive web app. Server-side rendering pre-fetches all the necessary data regardless of how client side interacts with the web app, then feeds all the pre-fetched data into the rendering scene (or components in React.js terminology), and returns the rendered scene back to the client side (e.g. browsers or any other user agents). After testing out and careful comparison, we believe server-side rendering gives better client-side performance as well as lighter loading on the server-side.

The main difference between server-side rendering and client-side rendering is that for server-side rendering the server's response to the client is a document (i.e. HTML) with data and is ready for displaying, while the client-side rendering gets an empty document with several links to some scripts (i.e. JavaScript files) [4]. In client-side rendering, those scripts further send requests to the server and server responds with data accordingly.

We initially implemented client-side rendering because of several concerns and potential bottleneck of server-side rendering. For example, theoretically the *time-to-first-byte (TTFB)* is usually longer in server-side rendering and ***React.js*** rendering is a synchronous call in ***Node.js*** environment and this theoretically increases the *event loop latency (ELT)*. However, according to our experiment results, due to the inherent nature of the Ethereum and Swarm, each client of NewsChain has to send multiple individual requests to get data needed to finish the browser rendering process, and we also notice that such way of requesting potentially contains multiple duplicated requests. According to lots of rounds of testing back to last year, whether or not duplicated requests could occur depends both on the internal design of the software and on how the client interacts with the app. And it's also found that it's hard to predict how clients are interacting and behaving and the improvement on software internal design has limited contribution to solving the issue. Hence, all things considered, we decide that in the Ethereum- and Swarm-powered web app, it

would be better to abandon client-side rendering totally in app based on Ethereum and Swarm but use server-side rendering instead. Moreover, we also found that server-side rendering makes the application more scalable when there are more and more users using the platform in the future.

The follow Figure 19 shows the ELT of Node.js running environment on the server (deployed on Heroku), and we can see that in practice, the ELT is still acceptable and under control even in server-side rendering. Hence, we believe that it's safe but also suitable to apply server-side rendering technology for our app.

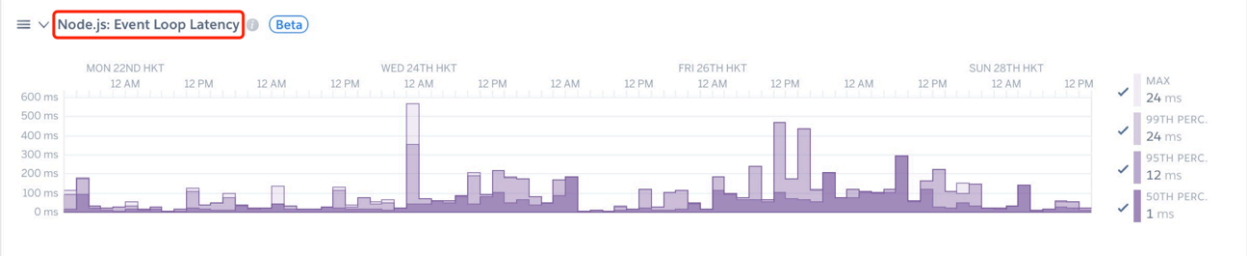


Figure 19. Node.js event loop latency monitoring diagram in a 24-hour period

Even though TTFB metrics has less priority compared to ELT, but it's still tested and we found that TTFB does not get too much longer than before after changing to server-side-rendering as long as we use lazy loading (or dynamic loading) properly for some particular React components.

State Synchronization among React Components

When the web app (built on ***React.js***) scales, it's unavoidable to split each page into several different React components for better reusability, maintainability, scalability and code readability. However, once split, the problem of state synchronization amount multiple React components arises. To solve such problem, we tried several different techniques during the whole development process, and these techniques include ***Redux***, ***MobX*** and ***React Context***. After some preliminary studies and trials in last year and development process later on, React Context are finally chosen to approach the issue.

We did use Redux for a while earlier this year, but however we abandon its due to the complexity and redundancy it brings to our code. Redux applies unidirectional data flow by creating a global store to ensure that state is synchronized by calling actions and reducing states from reducers [5], as demonstrated in Figure 20. Although it looks simple and clear, it actually requires a bunch of setup and it requires all the React components that need to have data synchronized to be connected to the global store, and it adds a lot of redundant code. As React Context technique is getting mature, we finally migrated our implementation from Redux to React Context one or two months ago.

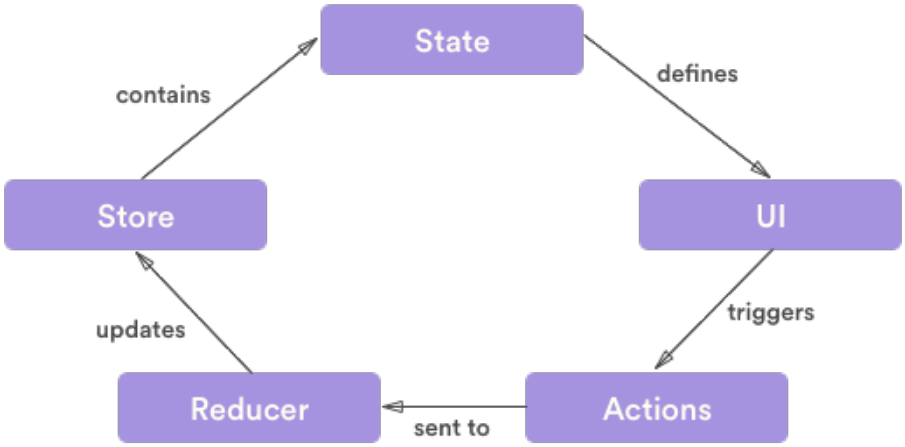


Figure 20. Unidirectional data flow in Redux

React Context is natively supported in React.js and is our final choice for state synchronization. In the implementation migrated from Redux, we create global context as the single source of truth to provide all the data needed to be shared and bind it to the most outer DOM (a.k.a. document object model) component. Then, the global context we create and bound gets consumed by all the child DOMs that use the shared data in it. Compared to Redux, React Context adds no redundant code and makes everything simple, clear and easy to maintain [6].

As a side note, MobX is based on observer pattern in its lower level [7] and after some preliminary research, we decided not to use MobX at the beginning of this project since it was too new at that moment, lacked community support and was not widely used in existing large-scale React progressive web apps.

Client-side Caching

Remember we talk about Server-side rendering in a previous section in this chapter, it helps us both increase client-side performance and decrease service-side loading. And this section will talk about another technique, namely client-side caching, that we applied in this project for the same purpose. With client-side caching, we bring the performance for end users the next level and meanwhile further lighten the burden of our server.

We have chosen and implemented ***IndexedDB*** to serve the needs of client-side caching. IndexedDB is an object-based NoSQL database supported by all the modern browsers, and it has become a standard in browser implementation. Besides, IndexedDB does not have a storage limitation which means we can store as much data as we want. This prevails another technique named Local Storage in browsers, as it limits the data size up to 5 megabytes while we have tons of articles to be stored containing rich text, images and potentially video clips. Hence, it becomes a suitable choice for us to ensure that the client-side caching works across multiple browsers for various end users. IndexedDB also natively supports database versioning so compatibility issue could be well handled when our application evolves and database schema changes. In implementation, we create a database named "NewsChain" and starts with version 1 (we are still using version 1 in our latest application, we have not made any changes that make it necessary to jump to the next version yet), and

we create an object store named “articles” to store all the article information in it. Note that we cache both data stored in *Article* smart contracts and data in Swarm in IndexedDB to facilitate the caching and increase loading speed as much as we could. Note that IndexedDB does have *data eviction protocol* [8] so we do not totally count on the integrity and consistency of the data in it. We check and renew the cache very often to prevent such issue. Note also that IndexedDB strictly follows *single origin policy* so our data pollution across multiple domains would not ever happen and this provides us a good way to separate multiple development and testing environments as well.

Data Visualization

Even though we have covered data visualization feature as one of our core features aforementioned in previous chapter, this section particularly focuses its frontend development side to help the audience understand what happens behind the scene more completely. Before going further in this section, the audience is encouraged to read through the related section in last chapter first for the data pre-processing, data real-time updating and graph algorithms.

To visualize our data, we make use of React-vis library by feeding it the graph data we generated by our algorithm, and in implementation we also ensure the diagram drew by the library will be updated in real-time manner. And in practice, when there

are more and more articles involved, we take advantages of lazy loading and caching to guarantee performance.

Regarding lazy loading, we ensure that articles not selected will be filtered out at the initial iteration of our graph construction procedure. And once an article gets selected, the real-time updating procedure will ensure that it's added back to the graph after one iteration of computing. In other words, an article is only fed to the algorithm that constructs the actual visible diagram until it is selected.

Regarding caching, it can be divided into two parts. The first part is the client-side caching, which is done by using IndexedDB as mentioned in the previous section. Client-side caching in data visualization implementation contributes to saving the time for loading article from the network. The other part is the caching or parsing result of article content, and by parsing we refer to the procedure that the article body is parsed from rich text to plaintext for preview, and that the multi-media data (e.g. images) in the article is decoded from corresponding format (e.g. base64 string format) into its display-able format (e.g. jpeg image format). Such parsing is done only once for visualization and the parsing result is cached in the memory of the web app and will be further reuse later so as to save the time from repeated parsing. After experiment and comparison, these two parts of caching start to have a significant amount of performance increasing when there are more than around 18 articles.

Hence, it's obvious that such caching procedures are necessary if we want to put NewsChain platform into production.

Client-side Article Searching

For better user experience, we implement client-side article searching function in our platform. When there are more and more articles published, searching for one or more articles would become a necessary function for our users. This section will cover the implementation of such searching function.

First of all, the searching is not done on the server side but on the frontend side and the rationale is because the server does not own the article data and needs to fetch it first from Ethereum and Swarm before performing any search. This way, there is no value-add to implement the searching on the server side, but however implementing it on the client side could make our server lightweight and less loaded. Besides, as data is stored in Ethereum and Swarm, we do not want the server becomes the center of such searching function or otherwise server will potentially become a single point of failure, as you can see in Figure 21.1 below. In such architect, the purpose of decentralization is totally defeated because we create another center – centering the backend server for searching (indicated by red color). However, Figure 21.2 shows the current architect that is inherently consistent to the decentralization nature of Ethereum and Swarm. We distribute the work of searching to all the clients

themselves and hence the server is totally transparent to the searching procedure, which means that no more potential single point of failure could happen.

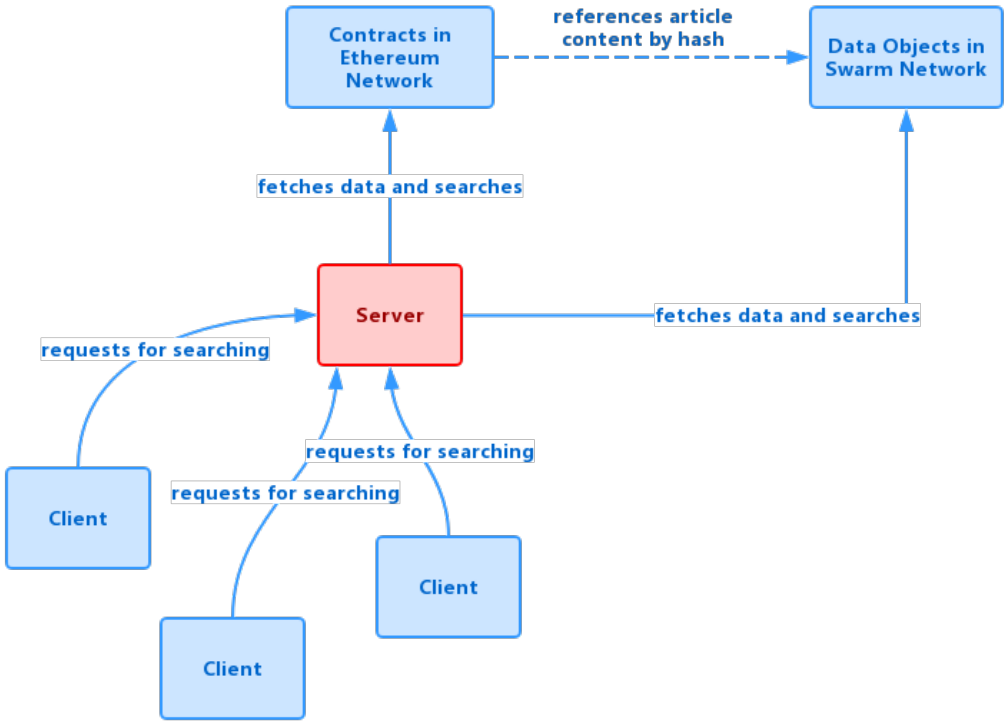


Figure 21.1 Architect of putting searching function on server (centralized)

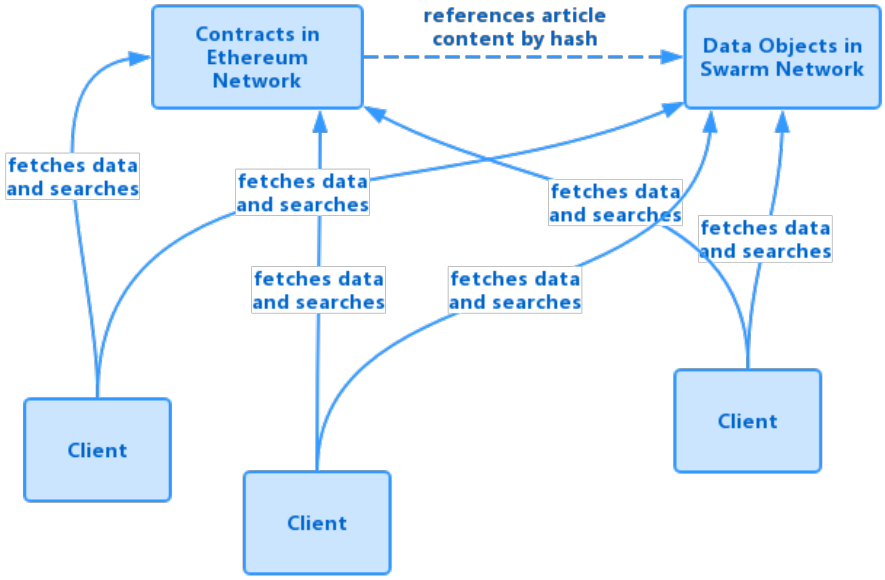


Figure 21.2 Architect of putting searching function on every client (decentralized)

There could be arguments saying that the server could index the content in Ethereum and Swarm first and the search could be done via a technique called inverted index, but however this will introduce lots of complexity to maintain an updated index table and break the single-source-of-truth principle, and more importantly the data source becomes centralized again on server instead of decentralized on Ethereum and Swarm. Hence, all things considered, it would be significantly better and more reasonable to implement the searching function on the client side and let each client to do the work itself.

The searching function supports keywords searching, key phrases searching, timestamp searching and address matching. The logical *OR* is applied among multiple keywords and key phrases by default (indicated by a space in user input). We have a brief diagram showing the workflow of parsing (by *parsing tree*) in Figure 22 below. Note that from implementation perspective, we use *regular expressions* to parse user input (e.g. searching keywords and key phrases quoted in double quotes) as well as to do searching and matching in article data. All the regular expressions are well designed to ensure searching and matching performance, and as we are running search on the client side (i.e. running in browsers), we avoid some regular expression syntaxes such as *look-behinds* that are not supported in all the browsers.

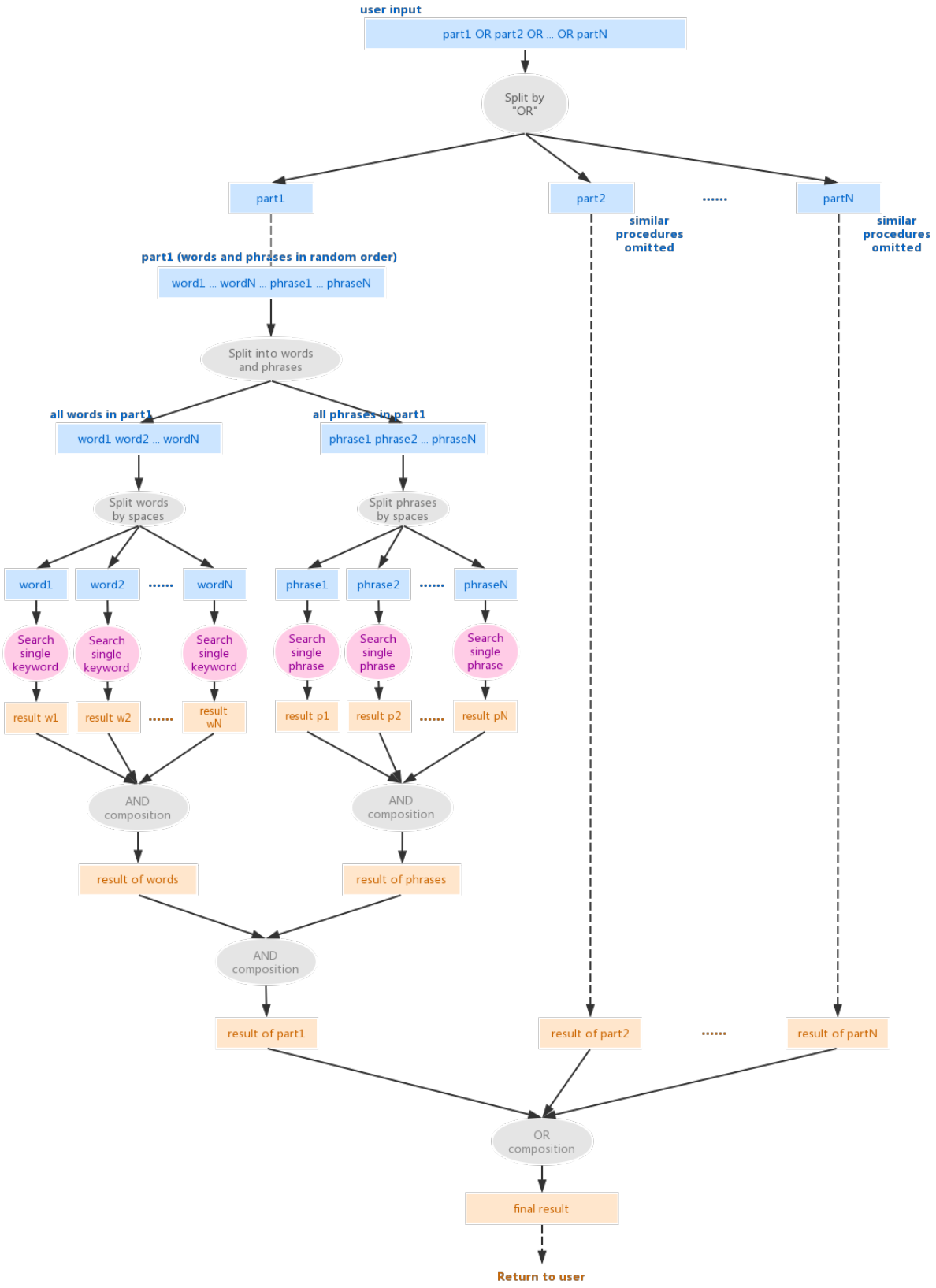


Figure 22. Parsing tree of the search function

Contract Design and Contract Interaction

In this chapter, we will delve into the contract design and contract interaction to see how our contracts are designed specifically to fit the need of NewsChain. We just cover all the core features and corresponding implementation details previously, while we still think that this separate chapter would be necessary for deeper understanding of the system. In this chapter, the audience will see how a new article gets created and later modifiable only by the author, how an article gets rewarded by ether or token and the specified recipient receives the reward immediately and automatically, how citation auto reward is triggered by citation and becomes an initial coin offering (ICO) procedure, and how the system manager reserves the right to fully control the ICO procedure by changing ICO parameters.

Contract Design

a) Article Factory Contract

Article Factory is the contract that fully controls article creation, initial citation record of a new article constructed later as well as the ICO procedure. It is deployed to the Ethereum by the system manager and the account address of the deploy-er will be recorded in the *admin* field automatically (retrieved from *msg.sender*). For ICO parameters *Reward Switch*, *Citation Cap* and *Reward Amount* aforementioned in System Features and Implementations chapter, they are *enableAutoTokenReward*

field, *autoTokenRewardCitationCap* field, and *autoTokenRewardAmount* field respectively as shown in the code snippet below. And *autoTokenRewardFrom* field indicates the NewsChain Token (NCT) address. The automatically rewarded token is also the initial coin offered by the system and is from the specific token contract indicated by the *autoTokenRewardFrom* field. A function modifier named *adminOnly* is added for checking authenticity for modifying ICO parameters. The *createArticle* function is the one used to instantiate and deploy an article contract in the Ethereum network, and the instantiation and deployment are totally done by the *Article Factory* contract itself without any manual interference. Note that an event *ArticleCreated* is defined in this contract and this event will be triggered whenever an article is created. This ensures that the web app gets notified (via ***Web3***) by such situation and reacts accordingly (e.g. request to load the latest article and re-render the page for it). The event also becomes an auditing log in the Ethereum and can be viewed by the public, so it also enhances the transparency of user contract interactions.

```
// ... some setup and configurations omitted

contract ArticleFactory {
  address public admin;
  address[] public deployedArticles;

  bool public enableAutoTokenReward;
  uint public autoTokenRewardCitationCap;
  uint public autoTokenRewardAmount;
  address public autoTokenRewardFrom; // references to NCT address

  constructor(bool enableReward, uint citationCap, uint amount, address rewardFrom)
public {
    admin = msg.sender;
```

```
    enableAutoTokenReward = enableReward;
    autoTokenRewardCitationCap = citationCap;
    autoTokenRewardAmount = amount;
    autoTokenRewardFrom = rewardFrom;
  }

  function createArticle(bytes32 _contentHash, address payable _rewardRecipient,
address[] memory _citations) public {
    address newArticle = address(new Article(_contentHash, msg.sender,
_rewardRecipient, _citations));
    deployedArticles.push(newArticle);
    for (uint i = 0; i < _citations.length; i++) {
      Article a = Article(_citations[i]);
      a.addCitedBy(newArticle);
    }
    emit ArticleCreated(newArticle);
  }

  modifier adminOnly() {
    require(msg.sender == admin);
    _;
  }

  event ArticleCreated(address newArticle);

  // ... other functions omitted
}
```

Please note that the above code snippet for Article Contract is incomplete, we omit several setup and configurations code at the beginning, and also omit some functions in the contract for simplicity. For the complete code, the audience may refer to the source code.

b) Article Contract

Article contract stores all the article metadata and Swarm hash pointer pointing to the article content stored in the Swarm network, and such pointer is stored in the field named *contentHash*. As the hash is hexadecimal, we store it in byte array

instead of string to make it more efficient and gas-saving. Note that this way, each hash value becomes one array of 32 bytes.

For all the historical versions of the article (due to modifications) are stored in an array named *history*, which is of type mapping (mapping from an integer to the hash). To modify an existing article (i.e. the already deployed *Article* contract), the *modify* function is called, and authenticity will be checked by the *creatorOnly* function modifier to make sure only the article author can make modification. Note that once modification occurs, the current version of the *contentHash* is pushed into the *history* array, and then the *contentHash* will be assigned to be the new hash of the modified article content that is just uploaded to Swarm.

The ICO procedure related code is in the function named *checkAndGenerateAutoReward*, but here we omit all the implementation details of it in the following code snippet as it's too complicated to show here. Basically, *Article* contract interacts with the *Article Factory* contract and *NewsChain Token* contract to handle the ICO procedure according to the preset ICO parameters and perform transaction accordingly. We will show the detailed relationship of these three contracts in later section when we talk about contract interaction. For those who are interested in the detail implementation here, please refer to the source code. Also, multiple events are defined and emitted properly to make sure web3 listeners

on the client side get notification about the latest updates and this also benefits our auditing log and makes user contract interaction more traceable.

For citation tracking, the bi-directional linked-list we mentioned in chapter System Features and Implementations are implemented by both *citations* array and *citedBy* array as shown in the code snippet. These two contract storage variables are maintained carefully to ensure consistency of the bi-directional linked-list. There are two *addCitedBy* functions with difference function signatures due to function overloading, and one with no parameter is called by other *Article* contracts and the other one with one parameter of type address is called by the *Article Factory* contract. Both *addCitedBy* functions have authentication check for caller *Article* and *Article Factory* respectively. Such authentication check ensures no one can cheat on the citation record (the bi-directional linked-list), and we emphasize the importance of the authenticity and integrity because remember that the ICO procedure in our design is closely related to citation relationship among articles.

```
// ... Setup and configurations omitted

contract Article {
  bytes32 public contentHash; // refers to the content in swarm
  uint public version;
  mapping(uint => bytes32) public history;

  address public creator;
  address payable public rewardRecipient; // rewardRecipient must be an EOA

  uint public rewardValue;
  uint public rewardTimes;
  mapping(address => uint) public rewardDonatorRecord;

  address[] public tokenTypes;
  mapping(address => uint) public tokenRewardTimes;
```

```
  mapping(address => uint) public tokenRewardValue;
  mapping(address => mapping(address => uint)) public tokenDonatorRecord;

  address[] public citations;
  address[] public citedBy;

  address factoryAddress;
  bool public autoTokenRewarded;

  constructor(bytes32 _contentHash, address _creator, address payable
_rewardRecipient, address[] memory _citations) public {
    contentHash = _contentHash;
    version = 0;
    creator = _creator;
    rewardRecipient = _rewardRecipient;
    factoryAddress = msg.sender;
    for (uint i = 0; i < _citations.length; i++) {
      citations.push(_citations[i]);
    }
  }

  function modify(bytes32 newHash) public creatorOnly {
    history[version] = contentHash;
    contentHash = newHash;
    version += 1;
    emit Modified(newHash, version);
  }

  function reward() public payable {
    require(msg.value > 0);
    rewardDonatorRecord[msg.sender] += msg.value;
    rewardValue += msg.value;
    rewardTimes += 1;
    rewardRecipient.transfer(msg.value);
    emit Rewarded(msg.sender, msg.value);
  }

  function rewardToken(address tokenAddress, address from, uint tokens) public {
    if (tokenRewardTimes[tokenAddress] == 0) {
      tokenTypes.push(tokenAddress); // add to tokenTypes if first time to donate
such token
    }
    tokenDonatorRecord[msg.sender][tokenAddress] += tokens;
    tokenRewardTimes[tokenAddress] += 1;
    tokenRewardValue[tokenAddress] += tokens;
    ERC20Interface(tokenAddress).transferFrom(from, rewardRecipient, tokens); //
should be approved first
    emit TokenRewarded(from, tokenAddress, tokens);
  }

  function addCitedBy() external {
    Article article = Article(msg.sender);
    require(article.isArticle()); // should be sent from an article
    citedBy.push(msg.sender);
```

```
    emit Cited(msg.sender);
    checkAndGenerateAutoReward();
  }

  function addCitedBy(address other) external {
    require(msg.sender == factoryAddress);
    citedBy.push(other);

    emit Cited(other);
    checkAndGenerateAutoReward();
  }

  function checkAndGenerateAutoReward() private {
    // ... ICO related implementation omitted
  }

  modifier creatorOnly() {
    require(msg.sender == creator);
    _;
  }

  // ... Many functions omitted


  event Cited(address otherArticle);
  event Modified(bytes32 newContentHash, uint newVersion);
  event Rewarded(address donatorAddress, uint amount);
  event TokenRewarded(address donatorAddress, address tokenAddress, uint
tokenAmount);
}
```

Note that same as before, the code snippet is incomplete and lots of functions and configurations are omitted. For the complete version, we request the audience to refer to the source code.

c) ERC20 Token Interface

This is the interface contract according to ERC20 token standard [3] and we will totally comply to it when we implement our NewsChain token. Besides the six functions and two events defined in the standard interface [3], we also add some

other functions to fit our specific need in NewsChain platform, including the citation auto reward and ICO procedure.

As it's a standard interface, we do not attach the code snippet of the interface contract here.

d) NewsChain Token Contract

NewsChain Token contract implements the ERC20 token interface and makes itself a valid token tradable in Ethereum. Note that the *generateAutoRewardToArticle* function is one part of the ICO procedure, an *Article* contract will call this function to generate token and this function checks that only *Article* contracts can call itself. The reward token amount is sent from the *Article* contract which is in turn set in the *Article Factory* contract. After this function's execution, the total supply of the token would be increased accordingly. If the audience feels confusing of this part, it's encouraged that the audience should look back to the chapter System Features and Implementations about how citation auto reward workflow is like and how it is related to ICO. The other part of the implementation in this token contract should be self-explanatory, and the audience can refer to the later section about contract interaction in this chapter to see how this token contract interacts and works seamlessly with other smart contracts we have.

```
// ... setup and configurations omitted

contract NcToken is ERC20Interface {
  string private _name = "NewsChain Token";
  string private _symbol = "NCT";
  uint public decimals = 0;

  uint public supply;
  address public founder;

  mapping(address => uint) public balances;
  mapping(address => mapping(address => uint)) allowed;

  event Transfer(address indexed from, address indexed to, uint tokens);
  event Approval(address indexed tokenOwner, address indexed spender, uint tokens);

  constructor() public {
    founder = msg.sender;
  }

  function name() public view returns(string memory) {
    return _name;
  }

  function symbol() public view returns(string memory) {
    return _symbol;
  }

  function allowance(address tokenOwner, address spender) view public returns(uint) {
    return allowed[tokenOwner][spender];
  }

  function approve(address spender, uint tokens) public returns(bool) {
    require(balances[msg.sender] >= tokens);
    require(tokens > 0);

    allowed[msg.sender][spender] = tokens;
    emit Approval(msg.sender, spender, tokens);
    return true;
  }

  function transferFrom(address from, address to, uint tokens) public returns(bool) {
    require(allowed[from][to] >= tokens);
    require(balances[from] >= tokens);

    balances[from] -= tokens;
    balances[to] += tokens;

    allowed[from][to] -= tokens;

    return true;
  }

  function totalSupply() public view returns (uint) {
    return supply;
```

```
  }

  function balanceOf(address tokenOwner) public view returns (uint balance) {
    return balances[tokenOwner];
  }

  function transfer(address to, uint tokens) public returns (bool success) {
    require(balances[msg.sender] >= tokens && tokens > 0);

    balances[to] += tokens;
    balances[msg.sender] -= tokens;
    emit Transfer(msg.sender, to, tokens); // emit
    return true;
  }

  // Generate token to an article for ICO, called by an Article contract
  function generateAutoRewardToArticle(uint tokens) public {
    Article article = Article(msg.sender);
    require(article.isArticle()); // transaction should be sent from an article

    balances[article.rewardRecipient()] += tokens;
    supply += tokens;
  }
}
```

Note that same as before, the code snippet above is incomplete for simplicity and better readability, and the audience should see the source code for the complete version.

Contract Interaction

After talking about the four smart contracts (or three smart contracts and one interface, note that interface is also a kind of contract by definition) above, we now will see how all of them are integrated and interact together with each other and externally owned accounts (EOAs) seamlessly. Figure 23 below shows the big picture of the interaction among all of the contracts, the interface and the EOAs.

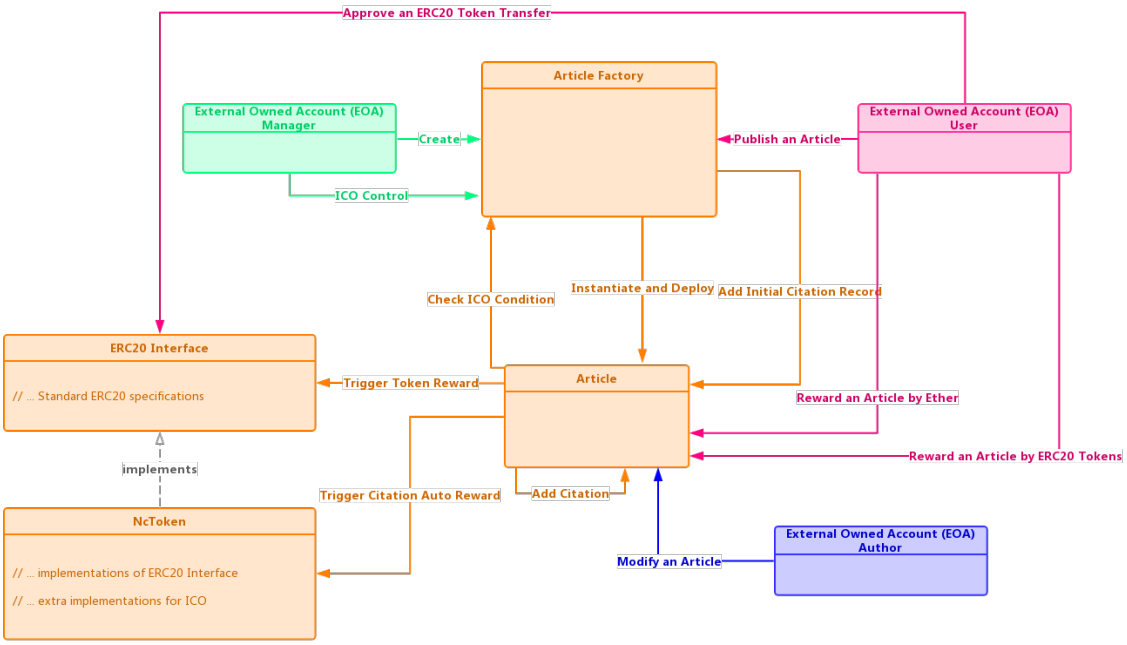


Figure 23. Interaction among contracts and EOAs

In Figure 23 above, the four orange boxes indicate the four smart contracts we have in our platform, and the orange arrows show the interaction behaviors and the data flow of such interactions among them. The green box indicates the manager account which creates and deploys the *Article Factory* contract as well as controls the citation auto reward and ICO procedure. The blue box indicates an article author account that is able to modify an article it published earlier. And last but not least, the pink box indicates a general user of the platform and the user can publish an article, reward an article by ether and reward an article by any kind of the ERC20 tokens including our NewsChain Token.

In particular, for ICO procedure, the *Article* contract checks with the *Article Factory* contract that deploys the *Article* contract itself to see whether or not the ICO is taking effect (determined by ICO control parameter *Reward Switch*), what is the number of citations it has to get in order to trigger the auto reward (determined by ICO control parameter *Citation Cap*), and how much the recipient will be offered (determined by ICO control parameter *Reward Amount*). After checking, if all the requirements of ICO are met, the *Article* contract triggers the citation auto reward by interacting with the *NewsChain Token* contract and *NewsChain Token* contract will transfer the preset number of tokens to the recipient accordingly.

Note that for simplicity, the above diagram in Figure 23 only shows one *Article* contract, while the audience should be noted that there are more than one article contracts in practice as for each published article there will be one *Article* contract corresponds to it. In other words, while there are one and only one *Article Factory* contract, *ERC20 Token Interface* contract and *NewsChain Token* contract, there will be multiple *Article* contracts depending on how many articles are published by the *Article Factory* contract.

Unit Testing with Ganache

As complex as it looks in the previous two sections, such contract design needs to be tested carefully before we start using it in production. Hence, we also apply unit

testing to automatically test out our contracts to see if there is any security flaw that allows user to cheat on ICO or bug that compromises application-level consistency and integrity. The unit test is set up on Ganache test network which is a local Ethereum running on the developer's machine with nearly zero block time. Ganache helps both separating testing environment from other environments as well as make testing efficient to confirmed transactions immediately on a block.

System Deployment

Contract Compilation and Deployment

We use Solidity 0.5.7 as the programming language for developing our smart contracts, and we have to manually deploy our *Article Factory* contract and *NewsChain Token* contract. The account address that initiates such deployment will become the system manager who will be able to fully control the ICO procedure later. The *NewsChain Token* contract should be deployed before *Article Factory* contract as *Article Factory* contract holds an immutable pointer pointing to the token contract (for citation auto rewarding). Note that as aforementioned, regarding *Article* contracts, they are only instantiated and deployed by the *Article Factory* contract so we do not deploy it manually, and regarding the *ERC20 Interface* contract, it will be implemented by other ERC20 token contracts in Ethereum and the deployment of the interface is unnecessary.

Before deployment, we should compile the smart contracts developed in Solidity first with a corresponding compiler (e.g. ***solc***, a compiler dedicated for compiling Solidity contracts). The code for compiling is attached below for the audience's reference, and some logic within it is removed for simplicity, for the complete version please refer to the source code.

```
const fs = require('fs-extra');
const solc = require('solc');

const inputPath = /*... path of the solidity file ...*/;
const outputPath = /*... path of the output json file ...*/;

const sources = {};
fs.readdirSync(inputPath).forEach(name => {
  sources[name] = { content: fs.readFileSync(path.resolve(inputPath, name),
'utf8') };
});

const { contracts } = JSON.parse(solc.compile(JSON.stringify({
  sources,
  language: 'Solidity',
  settings: {
    outputSelection: { '*': { '*': ['*'] } },
  },
})));

// ... other logic for compilation omitted
```

After compilation completes, we start our deployment process and the code snippet for deployment is attached below for reference. Note that the code is incomplete for simplicity and for the complete version please refer to the source code.

```
const fs = require('fs-extra');
const path = require('path');
const HDWalletProvider = require('truffle-hdwallet-provider');
const Web3 = require('web3');
const { mnemonic, ethereumNodeAddress } = /*... the deployment configuration ...*/
const provider = new HDWalletProvider(mnemonic, ethereumNodeAddress);
const eth = new Web3(provider).eth;

async function deploy(abi, bytecode, arguments = []) {
```

```
  const accounts = await eth.getAccounts();
  const result = await new eth.Contract(abi)
    .deploy({ data: '0x' + bytecode, arguments })
    .send({ from: accounts[0] });
  return result.options.address;
}

const compiledToken = /*... the compiled token contract ...*/
const compiledFactory = /*... the compiled article factory contract ...*/

const icoOptions = /*... Initial Coin Offering (citation auto reward) control
parameters ...*/;
const { enableAutoReward, citationCap, amount } = icoOptions;
const tokenAddress = await deploy(compiledToken.abi,
compiledToken.evm.bytecode.object);
const factoryAddress = await deploy(
  compiledFactory.abi,
  compiledFactory.evm.bytecode.object,
  [enableAutoReward, citationCap, amount, tokenAddress]);


// ... other logic for deployment omitted
```

Server Deployment

After deploying our contracts, we will now deploy our server so that it could serve our clients. Note that powered by Ethereum and Swarm, the server takes a minimum amount of responsibility, which mostly is sending back the server-side rendered document to the client, as aforementioned in previous chapter. Hence, different from traditional deployment architect, it is not of great necessity to add many redundancies for our server. This way, we could deploy it on a cloud platform with a minimal amount of setup process.

After some experiment and trial on different cloud platforms, our server is finally deployed on Heroku platform. Heroku only supports seamless deployment workflow with command line interface, but also provides elegant server metrics monitoring

services. Besides, Heroku applies container technology to have our app run in a separate virtual environment with great computational elasticity and reliability. All things considered, we believe Heroku would be the appropriate cloud service provider for this project to handle the server deployment process. We also add necessary plugins in Heroku to deal with tasks such as server logging and auditing.

Conclusion

NewsChain platform aims to solve the issue in journalism industry by empowering citizen journalism for the general public. In the belief of the wisdom of crowds and the distrust of the central authorities, plus the booming new decentralized blockchain technology, this project finally comes into being. As a DApp (decentralized app), NewsChain takes advantages of the P2P nature of Ethereum and Swarm, making news co-authoring fully decentralized, traceable and transparent. Besides, all the content in the platform is permanently stored and available on Ethereum and Swarm distributed network, making data loss impossible and eliminating the need for data backup. In such inherent nature of this platform, the server's responsibility is reduced to a minimum level – mostly only for server-side rendering.

However, we also notice some limitations in our app, an important one being the low adoption rate of blockchain technology. As we know, to send a transaction to the Ethereum Network, it is necessary to have an Ethereum wallet (e.g. MetaMask

in Chrome browser) to sign the transaction with the account private key. In other words, without the Ethereum wallet, users cannot publish, modify or reward an article as such actions are all Ethereum transactions behind the scene. There will be some amount of education cost for the traditional users to switch to our platform and install an Ethereum wallet such as MetaMask, even though it takes only 3 minutes to set up MetaMask. Nonetheless, we support the read-only mode whenever the user does not have MetaMask installed and under such circumstance all functions (e.g. reading articles and seeing citation relationship in data visualization dashboard) without the need of transaction are not affected at all.

As this is an ongoing project, in the future, we will consider adding several other features and/or making improvement on the current architect. For example, we may add features like voting (similar to the like and dislike feature on other social media platforms), so each article can be voted by the readers. Moreover, we may want to apply server-side cache layer to complement the current client-side cache, and ***Redis*** (in-memory key-value database) would be a good candidate for such layer. Also, although our server does not have much responsibility due to the powerful Ethereum and Swarm, deploying our server using serverless deployment method instead of Heroku is one consideration as well. Serverless deployment in some case could be cheaper in terms of pricing and usually of higher availability and better reliability compared to the traditional deployment method.

Last but not least, to conclude, blockchain technology makes NewsChain become possible and NewsChain empowers the society in a way traditional journalism industry cannot. It's been a fruitful year working on this project seeing it scale, grow and get more and more mature. NewsChain now demonstrates a new way of news co-authoring for the general public and there will be more effort put into the project in the future to make it more powerful.

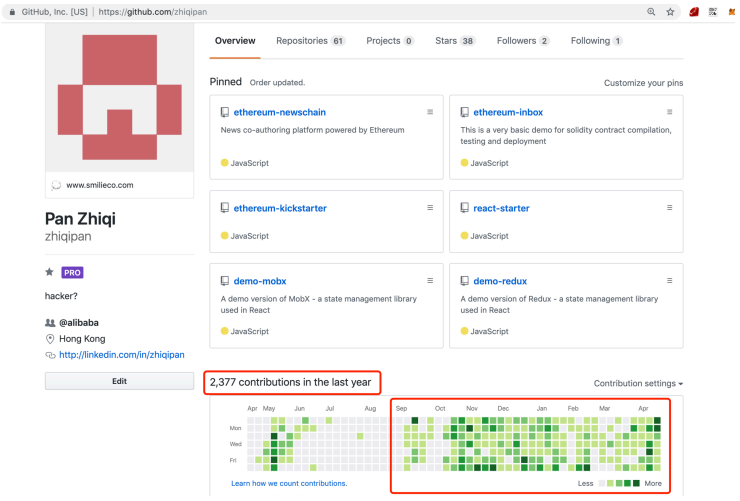
Appendix

Author and Developer

In this report, the author, *PAN Zhiqi*, may use the word "we" to refer to the team of this project, but however please note that all the technical work in this project including software architecture design, software development, software testing and software deployment are done by the author on his own. And by software, it means both frontend and backend software. This project was initiated by a team with the author's supervisor and other students from the School of Communications, while the only software developer of this project would be the author himself.

Work of This Year

It's been a great academic year working on this project, tremendously amount of effort has been put into this project and the GitHub home page of the author as well as the software developer of NewsChain is attached below for reference.



URL for Demonstration Purpose

The audience can play around with this platform which is available at this URL: http://eth-news.herokuapp.com

Disclaimer for the provided URL:

The URL is non-permanent due to the following reasons. If you find the above URL is not accessible anymore, please contact the author at *14252023@life.hkbu.edu.hk* or deploy the provided source code on your own.

1. NewsChain is an ongoing project, and there will be further development and deployment sooner or later in the future. Hence, the *URL* presented above might be changed or become unavailable when the further development is on progress; and
2. The content in the URL represents neither the latest work nor the work should be submitted for this final year project, as during the process of further development, sometimes we may add some not-so-stable features for alpha or beta testing.

Reference:

[1] Gerring, Taylor. *Building the Decentralized Web 3.0*. (2014, August 18). Retrieved April 29, 2019, from https://blog.ethereum.org/2014/08/18/building-decentralized-web

[2] *Smart Contracts—How to transfer Ether*. (2017, November 13). Retrieved April 29, 2019, from https://medium.com/coinmonks/smart-contracts-how-to-transfer-ether-ba464ec005c6

[3] *ERC20 Token Standard*. (n.d.). Retrieved April 29, 2019 from https://theethereum.wiki/w/index.php/ERC20_Token_Standard

[4] Grigoryan, Alex. *The Benefits of Server Side Rendering Over Client Side Rendering*. (2017, April 18). Retrieved April 29, 2019 from https://medium.com/walmartlabs/the-benefits-of-server-side-rendering-over-client-side-rendering-5d07ff2cefe8

[5] *Redux Core Concepts*. (n.d.). Retrieved April 29, 2019 from https://redux.js.org/introduction/core-concepts

[6] *React Context*. (n.d.). Retrieved April 29, 2019 from https://reactjs.org/docs/context.html

[7] *MobX Concept and Principles*. (n.d.). Retrieved April 29, 2019 from https://mobx.js.org/intro/concepts.html

[8] Browser Storage Limits and Eviction Criteria. (n.d.). Retrieved April 29, 2019 from https://developer.mozilla.org/en-US/docs/Web/API/IndexedDB_API/Browser_storage_limits_and_eviction_criteria