OpenCollab:
A Blockchain Based Protocol to Incentivize Open Source Software Development

Yondon Fu

May 27, 2017

Abstract

Open source software is one of the fundamental building blocks of today's technology dependent society and is relied upon by parties ranging from large technology corporations to individual hobbyist developers. The open question left for technologists is how to make open source software projects more sustainable. Attempts to provide more support for open source software projects such as bounty systems fall short of aligning the incentives of all parties involved.

The rise of decentralized networks of self-organizing, self-coordinating users incentivized by valuable cryptographic tokens enabled by Ethereum smart contracts creates the possibility of a system with embedded economics for open source software development that aligns the incentives of all parties. We present two contributions that can serve as building blocks for a better solution to open source software sustainability: a command line tool that enables a decentralized Git workflow without the need for a centralized service like Github and a proof-of-concept blockchain based protocol for incentivizing open source software development using a cryptographic token. Both contributions are implemented using Ethereum smart contracts.

1 Introduction

In today's software dependent society, open source software is everywhere. Parties ranging from large technology corporations like Google to individual hobbyist developers use open source software as the building blocks of their own projects. Tools that a developer once had to build from scratch are now widely available for anyone to use for free on websites like Github. Not only can anyone easily use these software packages, but anyone can also freely access, inspect and alter the source code, tailoring it for his or her own specialized needs.

The implications of democratized access to quality software is wide ranging. The open source web framework Ruby on Rails not only powers popular applications such as Twitter and Github that millions of people rely on everyday, but it also made web application development accessible to a broader audience by abstracting away the details of composing together components such as HTTP request handling, database querying and templating. Given the importance of open source software, the task at hand for technologists

is to figure out how to make open source software projects sustainable such that organizations and individuals in the future can continue to rely on them in the future.

The sustainability of an open source software project is tied to project health and support. Project health is determined by how actively and adequately project developers communicate with users such that the project addresses the needs of the community. Project support is determined by the availability of financial and technical resources to develop a project[16]. A sustainable open source software project needs to be both healthy and supported.

A healthy and supported project optimizes the use of developer time and attention, the scarcest resources in open source software projects. Communication between developers and users in a healthy project informs developers of community needs and issues to potentially focus their time and attention on. Availability of financial and technical resources in a supported project ensures developers are free to allocate their all of their time and attention on project issues. Consequently, in a healthy and supported project, developers can properly allocate their time and attention according to community needs.

If project developers fail to properly allocate their time and attention, users might leave a project in search of alternatives that better suit their needs. Canonical, the company behind the Linux operating system Ubuntu, created fragmentation in the Linux community when it shipped a new version of Ubuntu with the Unity interface rather than the standard GNOME interface[9]. Canonical's failure to properly poll for user opinion and allocate developer time and attention accordingly ultimately hurt the Ubuntu project.

1.1 Models for Sustaining Open Source Software

A proposed solution to open source software sustainability is a bounty system for project issues such as the one operated by the website Bountysource[7]. In these systems, users can attach monetary bounties to project issues that are rewarded to contributors that successfully resolve issues. While these bounty systems may help projects attract more contributors, they do not take into account the incentives of maintainers. The work done by maintainers to review and merge in code is just as crucial as the work done by contributors. Consequently, bounty systems only partially help with project support.

Furthermore, bounty systems do not necessarily help with project health.

Although in some cases, multiple users placing bounties on an issue might signal the importance the community places on that particular issue, it is also possible for malicious actors to place large bounties on issues that would negatively impact project quality. Such a possibility can place a burden on developers of filtering signal from noise and also introduces the possibility of collusion - a contributor might share a large bounty if a maintainer agrees to merge it into the codebase even if the contributed code is of poor quality. As a result, bounty systems can actualy hamper communication between developers and users leading to unmet community needs. Adding any form of financial compensation to open source software projects needs to align the incentives of all parties involved or else perverse incentives might arise leading to malicious behavior that harms the quality of the project.

Lastly, bounty systems operated by websites like Bountysource rely on a centralized entity to facilitate transactions. This reliance on a centralized entity not only results in a central point of failure, but can actually be more costly for users. For example, although users can freely transact within the bounty system, Bountysource charges a 10% withdrawal fee if a user wants to cash out. As a result, users choose between giving up a portion of their monetary rewards and giving up the numerous opportunies to use their monetary rewards for their own benefit outside the bounty system. This withdrawal free discourages users from leaving the system which benefits Bountysource, but harms users.

1.2 Cryptocurrencies and Blockchain Systems

The advent of cryptocurrencies and blockchains introduce a new decentralized paradigm for social systems. Cryptocurrencies are digital assets that rely on cryptography to secure transactions. In general, when we use the term cryptocurrencies, we describe decentralized cryptocurrencies managed by a distributed network of computers as opposed to fiat currencies that are managed by a central bank. Blockchains are the underlying technology that make these cryptocurrencies possible. These data structures establish the state of a system, whether it be a currency system or otherwise, without placing trust in a single entity. We will discuss blockchains in more detail in Section 5.4.

Blockchains can not only enable the creation of a bounty system that is not managed by a single entity, but also allows for the creation of a system with more complex rules that potentially offer economic support in a way that

aligns the incentives of all parties involved. With blockchains, developers can embed a set of rules for updating system state directly into software. Instead of trusting a centralized entity to enforce the rules, users know that the software will enforce the rules since it is programmed to determinstically execute and respond to a predetermined set of instructions. In a centralized paradigm, systems rely on central entities for coordination and organization. In a decentralized paradigm, systems are formed by a distributed network of self-coordinating and self-organizing users that follow a common software protocol powered by a blockchain. Protocols can economically incentivize certain actions by rewarding users with protocol native cryptocurrencies if conditions established in the protocol rule set are fulfilled.

Communities around open source software projects currently rely on centralized services to coordinate and collaborate. Adding monetary rewards to a project in a way that aligns interests of all parties is not only difficult, but also adds middlemen and transaction costs to the system. Cryptocurrencies and blockchains can be the building blocks for a potentially better system.

1.3 Contributions

The primary contributions of this thesis are the following:

- A command line tool that enables a decentralized Git workflow for developing open source software without relying on a centralized service like Github as described in Section 3. The code is available open source at https://github.com/yondonfu/opencollab.
- A proof-of-concept blockchain based protocol to incentivize open source software development as described in Section 4. The code for the set of smart contracts implementing the protocol is available open source at https://github.com/yondonfu/opencollab-contracts.

The motivation behind these contributions is to push the discussion on how to improve open source software sustainability. In particular, these contributions are attempts to answer the following questions relating to open source software sustainability.

- How can developers poll for user opinion on issue priorization for a project?

- How can a project attract regular contributors?
- How can maintainers be incentivized to carefully review and merge pull requests such that the quality of a project is upheld?

A detailed description of these contributions can be found in Sections 3 and 4.

2 Background

2.1 Ethereum

Although security focused members of the computer science community have expressed a fair amount of concern about the viability of Ethereum as a blockchain used for smart contracts due to the large attack surface presented by its Turing complete programming language, the stark reality is that the Ethereum developer ecosystem is the most active of any other blockchain ecosystem. Furthermore, various members of the community are actively researching methods to better secure the Ethereum network including formal verification, proof-of-stake as an alternative consensus algorithm to proof-of-work and smart contract programming languages with stronger security guarantees. Consequently, with these points in mind we decided to build the OpenCollab protocol on Ethereum.

2.2 Cryptographic Tokens

3 Decentralized Git Workflow

Git is a distributed version control system that is commonly used in a centralized workflow. Developers often work using a local Git repository and coordinate with other developers by pushing their changes to a remote Git repository that is either hosted by a trusted third party or self-hosted. While developers are free to host their own remote Git repositories, it is far more common to rely on a web service such as Github to handle hosting. Github abstracts away the complexities of repository hosting and also offers developers additional featues that are not native to Git such as access control, issue tracking and a pull request system. Developers use a remote protocol such as HTTP, SSH or Git (packaged with the VCS) to communicate with and transfer data to the remote repository hosted on Github[19]. Ease of use and collaboration features have solidified Github as a must have developer productivity tool. However, a centralized workflow centered around Github also has a number of downsides. If a user relies on Github, the user also relies on all of Github's dependencies. The failure of a dependency resulting from a cyberattack can render Github's services and any hosted code unavailable for a period of time[17]. Furthermore, any additional repository features such as payments and governance mechanisms around project direction cannot be directly integrated into a repository because Github ultimately controls the remote repository. As long as Github is used as a centralized service to coordinate and collaborate on a project, these features can only be integrated into a project if Github chooses to implement them. Consequently, there is potential value in a decentralized Git workflow that does not rely on a centralized service like Github to coordinate and collaborate on projects.

3.1 Mango

Mango is a remote protocol for Git that uses Ethereum smart contracts for remote repository access control and stores Git objects in decentralized content addressable storage networks[5]. The smart contract associated with a repository maintains a whitelisted set of Ethereum addresses that can push changes to the repository. Although Mango is storage solution agonistic, it is best served by a decentralized content addressable solution and the initial implementation uses the Interplanetary File System (IPFS), a peer-to-peer distributed file system[4]. Git objects are serialized and uploaded to IPFS which returns the content hash for the objects. Since Git uses the

SHA1 hash algorithm, while IPFS is hash algorithm agnostic, a snapshot mapping of Git SHA1 object hashes to IPFS object hashes is also uploaded to IPFS[6]. These IPFS snapshot hashes and Git references are then stored in an Ethereum smart contract. The smart contract address is used as an identifier for the repository. Using a command line tool, users can push to and pull from a remote repository using the smart contract address associated with a repository.

3.2 OpenCollab-CLI and Extensions to Mango

In order for the Mango protocol to support a decentralized Git workflow that is comparable in productivity capabilities to a centralized Git workflow using Github, it needs to offer issue tracking and a pull request system. As a part of our contributions, we extended the original Mango protocol implementation by modifying the MangoRepo smart contract to support CRUD operations for issues and pull requests. The OpenCollab-CLI command line tool allows users to manage issues and pull requests for a repository in a terminal. The command line tool uses Swarm, a distributed storage platform and Ethereum ecosystem service, rather than IPFS as a decentralized storage solution[27]. However, similar to the original Mango protocol, our protocol extensions are also storage solution agnostic.

Issue contents are uploaded to Swarm which returns the hash for the contents. The Swarm hash is then stored in the smart contract for the repository and mapped to an integer identifier. When using the command line tool, retrieving the contents of an issue consists of querying the smart contract with the issue identifier and then querying Swarm using the Swarm hash corresponding to the issue identifier.

Users can create a pull request with a two step process. First, they need to fork the project. The command line tool can be used to fork a project locally. In the project fork, users can initialize a new Mango repository, make relevant changes and push to the smart contract address associated with the repository fork. After pushing a project fork, users can use the command line tool to create a new pull request that references an issue identifier and the contract address for the repository fork.

A project maintainer can merge a pull request with a two step process. Maintainers can locally clone a repository fork using the contract address referenced in a pull request. The command line tool can be used to locally merge a repository fork into the main repository. Then, maintainers can

push the merged changes to the contract address associated with the main repository as long as their Ethereum address is whitelisted by the repository contract.

3.3 Future Work

The OpenCollab-CLI command line tool along with the Mango protocol extensions enable a decentralized Git workflow that not only obviates the need for a centralized service like Github to coordinate and collaborate for projects, but also creates the possibility of directly integrating features such as payments and governance mechanisms directly into a repository at the protocol level. These contributions serve as neccessary foundation for the OpenCollab blockchain based protocol which is described in the next section. At the moment, the OpenCollab-CLI command line tool and the OpenCollab protocol are separate. Future work would include upgrading the OpenCollab-CLI command line to be compatible with the OpenCollab protocol.

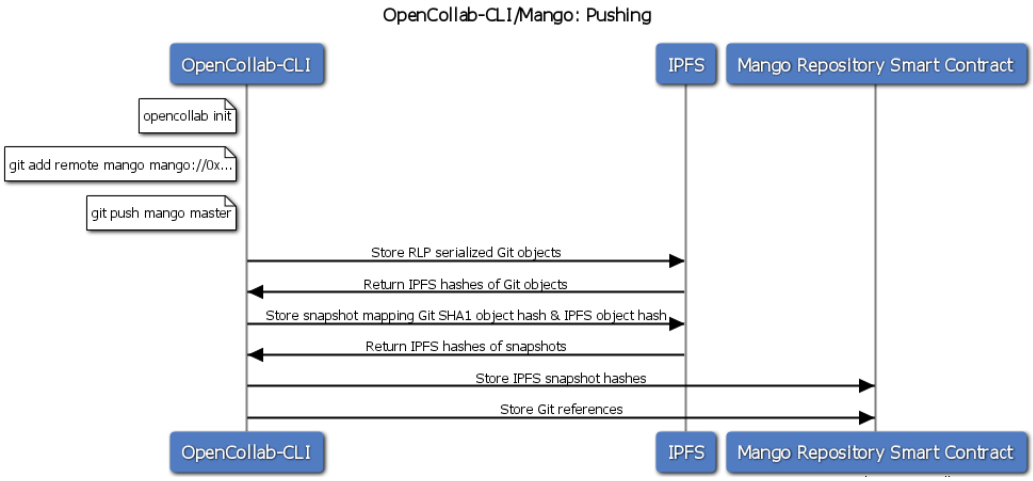

Figure 1: Pushing to a remote repository using the Mango remote protocol and OpenCollab-CLI

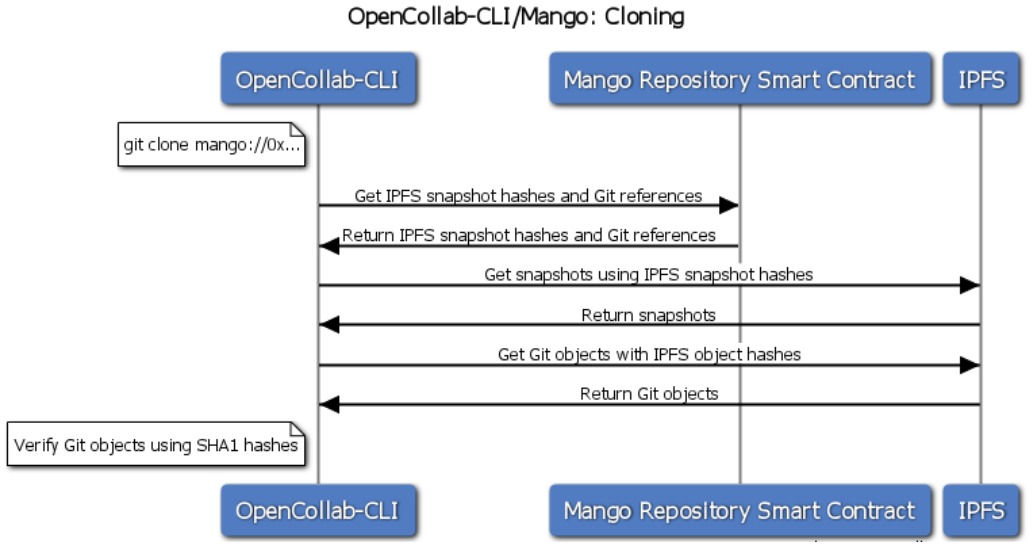

Figure 2: Pulling from a remote repository using the Mango remote protocol and OpenCollab-CLI

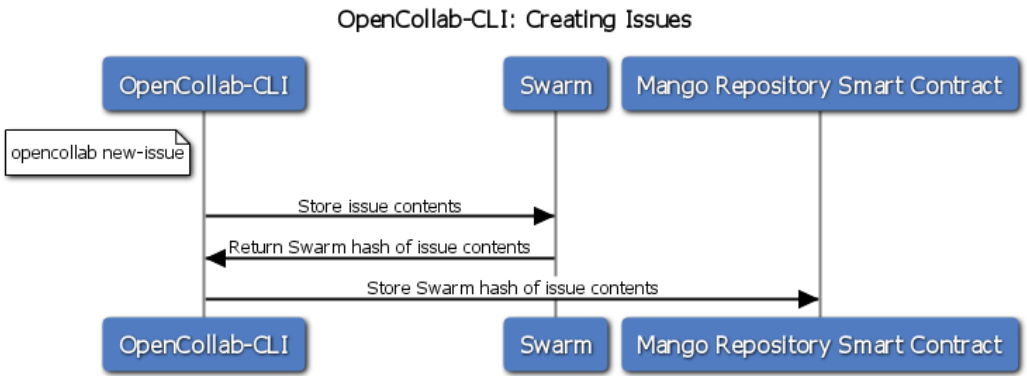

Figure 3: Creating issues using OpenCollab-CLI

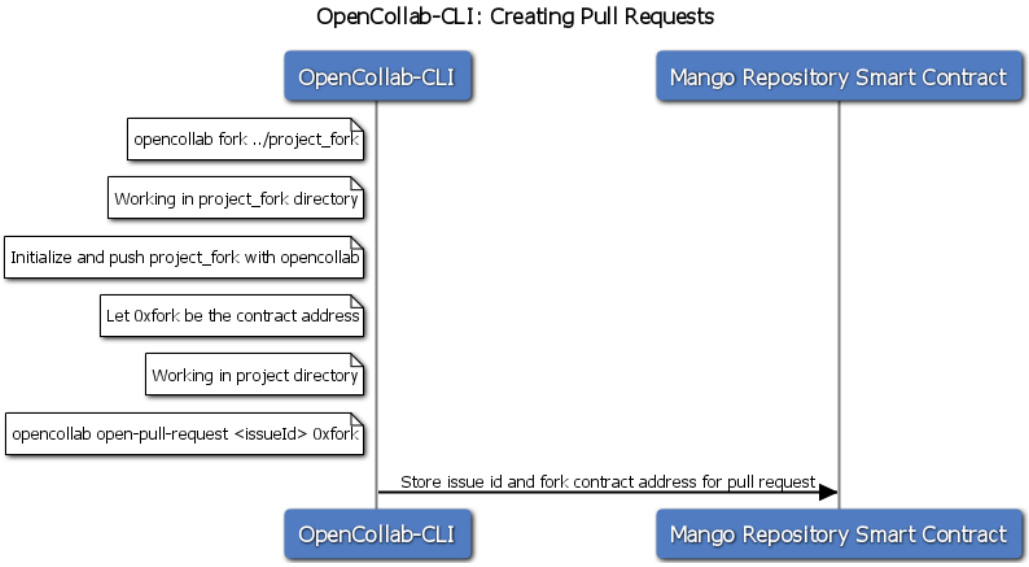

Figure 4: Creating pull requests using OpenCollab-CLI

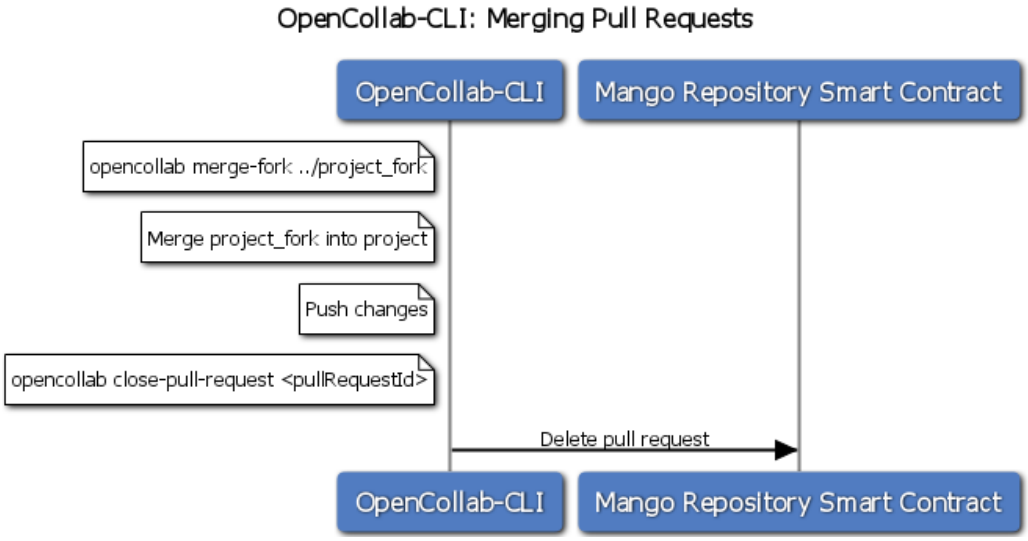

Figure 5: Merging pull requests using OpenCollab-CLI

4 OpenCollab Protocol

The OpenCollab protocol is implemented by a set of Ethereum smart contracts. By building on the Ethereum platform, we can rely on security properties of the underlying Ethereum blockchain. With the details of consensus and security abstracted away, the OpenCollab protocol focuses on defining secure economic incentives and rules that encourage open source software sustainability.

4.1 Protocol Roles

- **Voters**: participate in governance voting by depositing tokens.
- **Curators**: curate project issues by staking tokens.
- **Contributors**: open pull requests to resolve issues by staking tokens.
- **Maintainers**: review and merge pull requests for issues by staking tokens.

4.2 OpenCollab Token

The OpenCollab token (OCT) powers the OpenCollab protocol. The value offered by the token is influence over an open source software project. Furthermore, the token serves the following purposes in the protocol:

- Used in deposits for token holders that choose to participate as voters in protocol governance.
- Used in a staking mechanism for issue curation. Curators stake tokens to signal the importance they place on an issue.
- Used in a staking mechanism for opening pull requests. Contributors stake a certain number of tokens when opening a pull request. If a contributor's pull request is closed without being merged in to the project, the contributor's staked tokens are destroyed. The possibility of losing staked tokens discourages contributors from opening pull requests unless they are confident about the quality of their contributions.

- Used in a staking mechanism for merging pull requests. Maintainers stake a certain number of tokens when they initiate a merge. Before a merge is finalized, a token holder can challenge a maintainer's merge to start a voting round. If token holders decide to veto a maintainer's merge, the maintainer's staked tokens are destroyed. The possibility of losing staked tokens discrourages maintainers from merging pull requests that do not benefit a project. The challenge and voting process for a merge is described in more detail in Section 4.6.

An initial allocation of tokens will be distributed so that the various protocol roles can be fulfilled by token holders. A project creator can initialize an OpenCollab repository and mint a certain amount of tokens for the initial allocation. The initial allocation might be done using a token crowdsale or by disbursement at the discretion of the project creator.

OCT is an ERC20 compliant token[31] and is divisble by 10^{18}. When a repository using the OpenCollab protocol is created, a smart contract is created governing a OCT that is specific to that particular repository. As a result, there can be many repository specific versions of OCT.

4.3 Governance Voting

Token holders can elect to participate in governance voting by depositing voterDeposit tokens. At the moment governance voting only takes place for challenged pull request merges which is described in Section 4.6, but in the future it can be used for other protocol decisions. One use case might be to vote on adding maintainers to a project. Token holders have an incentive to be voters because they stand to earn rewards if they vote well. Voters on the winning side of a vote increase their deposits by voterRewardPercentage. At the same time, voters on the losing side of a vote decrease their deposits by voterPenaltyPercentage.

4.4 Curating Issues

Curators stake a number of tokens to an issue to signal the importance that they place on the issue. Since curators lock up their tokens for a period of time when they stake tokens to an issue, they have limited curation power. Curators either exchange funds for tokens by purchasing them on the secondary market or work for tokens by being a contributor or maintainer. Fur-

thermore, since curators take on the risk of a fall in token value, they have skin in the game[29]. If curators signal importance for bad issues, developers poorly allocate their time and attention. If developers properly allocate their time and attention on resolving issues that would increase the quality of a project, the community might lose interest and less people would desire influence over the project leading to a fall in token value. Consequently, curators have an incentive to signal importance for issues that accurately reflect the needs of the community. As well curated issues are resolved, the value of the token would increase thereby benefiting curators.

4.5 Opening Pull Requests

Contributors open pull requests by staking CONTRIBUTORSTAKE tokens, where CONTRIBUTORSTAKE is a repository parameter.

If a contributor's pull request is successfully merged by a maintainer, the contributor receives a portion of the issue's token reward. The portion can be calculated as REWARD - (REWARD * MAINTAINERPERCENTAGE).

If a contributor's pull request is closed without being merged into the project, the contributor's CONTRIBUTORSTAKE staked tokens are destroyed.

4.6 Merging Pull Requests

Maintainers merge pull requests by staking MAINTAINERSTAKE tokens, where MAINTAINERSTAKE is a repository parameter.

If a maintainer wants to merge a pull request, it calls INITMERGEPULLREQUEST(ID) to signal an intent to merge a particular pull request and starts a challenge period. During this period, any token holder can challenge the maintainer by calling CHALLENGE() and staking CHALLENGERSTAKE tokens.

If a maintainer is not challenged during the challenge period, he can call MERGEPULLREQUEST(ID). The maintainer receives a portion of the issues's token reward which can be calculated as REWARD * MAINTAINERPERCENTAGE.

If a maintainer is challenged during the challenge period, a voting period begins. Voting takes place using a two step commit and reveal protocol first formalized by Brassard, Chaum and Crepeau[8]. During the commit step, voters with a minimum VOTERDEPOSIT deposit in the smart contract vote to uphold or veto a maintainer's merge by calling COMMITVOTE(HASH) with

the cryptographic hash of their vote and a secret phrase. A vote to uphold is a 1 and a vote to veto is a 2. The secret phrase can be any random string only known to the voter. The value of the vote is secure from an attacker as long as only the voter knows the secret phrase used when generating the hash. We use the SHA3 KECCAK256 hash function since it is used internally by Ethereum.

During the reveal step, voters reveal the values of their votes by submitting the concatenation of their vote and secret phrase used in the commit step by calling REVEALVOTE(VOTE). The smart contract verifies that the submitted vote corresponds with the committed hash and tallies up votes as voters reveal them. Finally, anyone can call VOTERESULT() which compares the number of uphold and veto votes. The value that receives the majority of vote ($\geq 50\%$) wins. Voters on the losing side of the vote are penalized such that VOTERPENALTYPERCENTAGE is deducted from their deposits. Voters on the winning side of the vote are rewarded such that VOTERREWARDPERCENTAGE is added to their deposits.

If a maintainer's merge decision is upheld, the maintainer is able to call MERGEPULLREQUEST(ID) to finalize the merge. The challenger's CHALLENGERSTAKE staked tokens are destroyed and the maintainer can claim his portion of the issue token reward.

If a maintainer's merge decision is vetoed, the challenger's staked tokens are returned and the maintainer's MAINTAINERSTAKE tokens are destroyed and is removed from the maintainer set for the repository. Consequently, the former maintainer would not only lose the staked tokens, but also the economic value of future issue token rewards. The possibility of losing tokens and maintainer status serves to encourage maintainer to only merge pull requests that ensure the quality of the project.

4.7 Amortisation of Work

Each computation step taken by an Ethereum smart contract costs a certain amount of gas, an internal accounting unit. As a result, when writing smart contracts, it is important to keep in mind gas costs. One of the most costly programming constructs that can be included in a smart contract is a loop over a large array of elements. If the array of elements can become arbitrarily large, the gas cost of iterating over the array can also become arbitrarily large.

In the OpenCollab protocol, voters are penalized or rewarded after a voting round depending on whether they were on the winning or losing side

of the vote. A simple and naive way of performing the accounting for these penalties and rewards would be to loop through all voters and check if they were on the winning or losing side of the most recent vote. This accounting can be done at the end of the VOTERESULT() function.

However, the size of the voters array can become arbitrarily large as more token holders put down despoits to participate in governance voting. As the voters array grows in size, the gas cost of performing the accounting for penatlies and rewards after a vote will grow as well. Accounting for penatlies and rewards in such a way will eventually become too expensive.

A common design pattern used in smart contracts to avoid massive gas costs in single function calls is amortisation of work[22]. We can break up the work being done over other operations. In the OpenCollab protocol, rather than updating voter deposits with penalties and rewards after every vote, we introduce a VOTERCHECKIN() function. The function updates a voter deposit with penalties and rewards for all voting rounds that occured since the last voting round that the voter checked in to.

Voters can only withdraw their deposits if they have checked in to the latest voting round. Voters are incentivized to call VOTERCHECKIN() frequently because otherwise the gas cost of the function increases with the number of voting rounds that the voter has not checked in. As a result, the work of calculating penalties and rewards for votes is distributed across all voters and we avoid large gas costs associated with single function calls.

4.8 Future Work

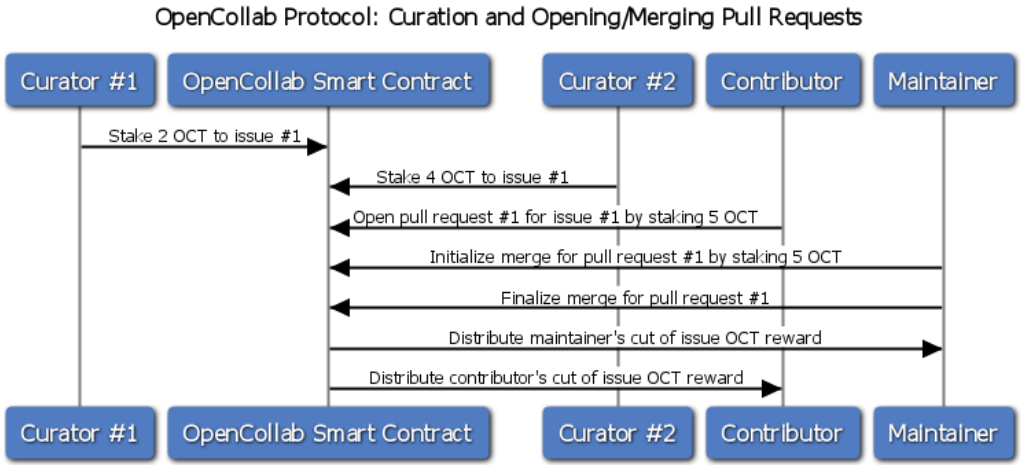

Figure 6: Curating, opening pull requests and merging unchallenged pull requests

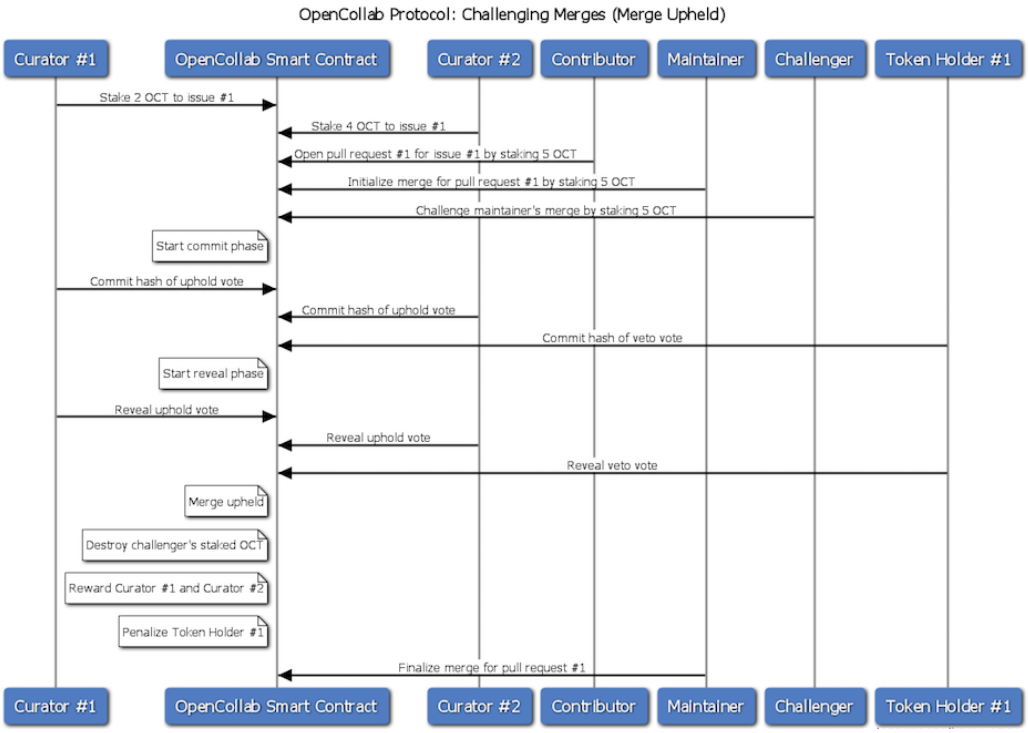

Figure 7: Upholding a challenged merge

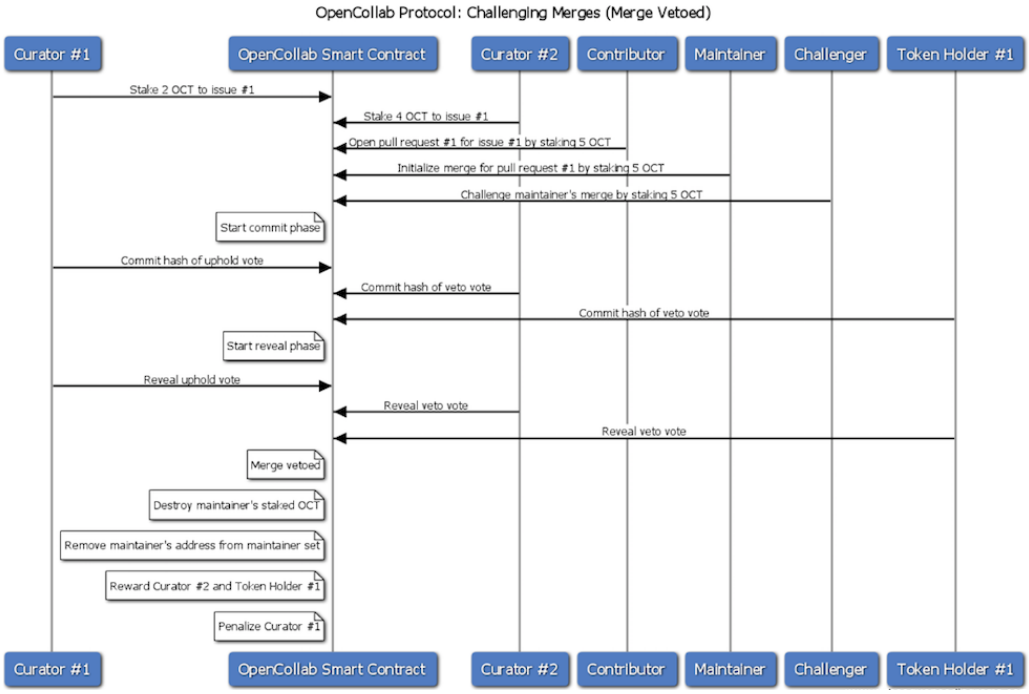

Figure 8: Vetoing a challenged merge

```
// vr = latest voting round
for (uint256 i = 0; i < voters.length; i++) {
  if(vr.votes[voters[i]].voteValue == VoteValue.None) {
    // Voter abstained
    // Penalize voter
  } else {
    if (vr.result == vr.votes[voters[i]].voteValue) {
      // Voter on winning side
      // Reward voter
    } else {
      // Voter on losing side
      // Penalize voter
    }
  }
```

Figure 9: Calculating penalties and rewards for all voters in VOTERESULT()

```
        for (uint i = startRound; i < rounds.length; i++) {
          if (rounds[i].votes[msg.sender].voteValue == VoteValue.None)
            // Voter abstained
            // Penalize voter
          } else {
            if (rounds[i].result == rounds[i].votes[msg.sender].voteVa
              // Voter on winning side
              // Reward voter
            } else {
              // Voter on losing side
              // Penalize voter
            }
          }
        }
```

Figure 10: Amorisation of work - voters calculate their own penalties and rewards in VOTERCHECKIN()

5 Related Work

Although cryptocurrencies and blockchains have only recently captured the attention of academics and business people, the technical foundations of these systems actually have a longer history. Cryptocurrencies and blockchains use ideas from past work by computer science researchers in the areas of distributed consensus, electronic money and digital time-stamping. Additionally, as types of distributed systems, the deployment of cryptocurrencies and blockchains in a public setting and on a large scale provide interesting insights particularly when viewed in the context of past distributed consensus research.

5.1 Distributed Consensus

The reliability of a distributed system depends on system proccesses to reach consensus on particular values. A distributed system can be viewed as a replicated state machine consisting of a state machine replicated across multiple processes. The replication of state across many processes creates a level of redundancy that can protect a system against failure due to a single faulty individual component. The replicated state machine uses a deterministic state transition function to map a set of inputs and the current state to a new state[26]. State transitions are atomic such that they either occur completely or do not occur at all, consistent such that they must be valid mappings of inputs and the current state to a new state and durable such that once they occur, state is permanently updated. Thus, we can consider state transition function inputs as transactions[20].

Although a distributed system can protect against failure due to a single faulty individual component, it remains vulnerable to multiple process faults that prevent the system from achieving consensus on a particular value due to incorrect process behavior. Process faults come in a different varieties, but two common types are fail-stop faults, where processes crash and other proccesses can detect the failure, and Byzantine faults, where processes exhibit arbitrary and potentially malicious behavior[26]. In comparison to solutions for fail-stop faults, solutions for Byzantine faults require additional complexity because faulty Byzantine processes can transmit conflicting information to other processes that might not be immediately detected. Lamport et al. initially presented Byzantine fault tolerance using the Byzantine Generals Problem, in which a number of Byzantine generals attempt to coordinate

an attack on an enemy city in the presence of potentially traitorous generals. The traitorous generals can be characterized as Byzantine processes in a computing context.[23].

Lamport et al. offered a few solutions to the Byzantine Generals Problem, one involving oral messages requiring fewer than f traitorous generals given $3f + 1$ generals, and another involving messages with unforgeably signatures that allows an arbitrary number of traitorous generals. However, due to an assumption that the absence of messages can always be detected, these solutions are only applicable to synchronous environments. In synchronous environments, system designers can make assumptions about the maximum delay of network messages, but in asynchronous environments, system designers cannot make any assumptions about network delays[10]. In asynchronous environments, an adversary might have the power to schedule network messages. Consequently, the solutions of Lamport et al. would not be sufficient in an asynchronous environment because there is no guarantee of the detecting the absence of messages.

The FLP impossibility proof states that distributed consensus in an asynchronous environment with deterministic processes is impossible if a single process can crash[18]. However, the proof does not preclude distributed consensus in asynchronous environments with certain weak synchronous assumptions or with randomization. One example of a Byzantine fault tolerant consensus algorithm that uses weak synchronous assumptions by tweaking a timeout parameter is Practical Byzantine Fault Tolerance (PBFT) which offers system resilience as long as there are fewer than f Byzantine processes given $3f+1$ total processes at any given point in time[13]. A number of computer science researchers have also explored Byzantine fault tolerant consensus algorithms using randomization. Ben-Or offered one such algorithm that involves processes using a register that is probabilistically set to either 0 or 1 to decide on a binary value. The algorithm works with probability eventually reaching 1, but also requires fewer than f Byzantine processes given $5f + 1$ total processes[3]. We will observe in Section 5.4 that certain cryptocurrency systems and blockchains use a combination of weak synchronous assumptions and randomization to achieve Byzantine fault tolerant distributed consensus in asynchronous environments.

5.2 Electronic Money

The advent of the Internet and the mainstream adoption of computing devices encouraged the development of many forms of electronic money by recording balances electronically on devices.

In 1983, David Chaum introduced a cryptographic protocol for anonymous payments using blind signatures. In Chaum's protocol, a bank uses its private key to sign blinded tokens and payees accept signed tokens by clearing a signed token with the bank[14]. The authenticity of tokens can be guaranteed by verifying the bank's signature on tokens with the bank's public key. The bank also does not know the identity of a payer when clearing a signed token sent by a payee because the token was blinded to obfuscate the amount and sender the bank originally signed the token. Chaum applied this protocol in his DigiCash project.

One of the flaws of DigiCash is that it can only offer durable transactions in exchange for decreased anonymity. Users must present tokens to the bank for verification or else they are vulnerable to double spending attacks since electronic messages can easily be duplicated[12]. Furthermore, reliance on the bank creates a bottleneck for system throughput and a central point of failure. If a bank's private key is compromised, an attacker can use the bank's private key to create counterfeit tokens.

Easily duplicated electronic messages leave electronic money systems vulnerable to denial of service attacks. As a solution, Adam Back proposed using hashcash, a easily verifiable, but difficult to compute cost function to mint tokens[1]. The cost function or *proof-of-work* is based on finding partial hash collisions on the k-bit string 0^k, for which the fastest known algorithm is brute force meaning users must perform a certain amount of work in terms of computing cycles to mint tokens. A proof-of-work requirement discourages electronic message duplication by making message creation costly.

Nick Szabo highlighted the utility of proof-of-work for electronic money in his bit gold protocol. The protocol uses a proof of work function to compute a string of bits that is timestamped in a distributed property title registry[28]. Users can verify owernship of a string of bits in the title registry. In contrast with DigiCash, bit gold allows valuable bits to be created, transferred and stored without depending on a trusted third party. However, a system implementing such a protocol was never implemented in practice.

5.3 Digital Time-Stamping

Widespread digitization of all types of documents brought many benefits to society, but also introduced the question of how to prove the existence and time of creation or change of a digital document.

In 1991, Haber and Stornetta presented a time-stamping method for digital documents that consisted of certificates cryptographically signed by a time-stamping service. The certificates contain the hash of the document as well as linking information from a previous certificate which includes a hash of the previous certificate's linking information[21]. The result is a hash linked chain of certificates that prevents the faking of time-stamps.

Bayer, Haber and Stornetta extended this time-stamping method using merkle trees. In the original time-stamping method, verification of a document timestamp can require at most N steps by following the chain links to a time-stamp certificate that is trustworthy[2]. Instead of linking N hashes of documents, the hash values can be stored in a merkle tree. Participants can record the hashes of their own documents and the sibling hash values along the path from the document hash to the root of the merkle tree. Consequently, verification can be done in at most $\lg N$ steps by presenting the document hash and the $\lg N$ hashes on the path to the root. This modified time-stamping approach reduces storage requirements and verification time.

5.4 Blockchains

Blockchains combine learnings from past work in distributed consensus, electronic money and digital time-stamping. A blockchain is a type of distributed system with two key defining characteristics. The first characteristic is that transactions are grouped into blocks. A common optimization reminiscient of Bayer, Haber and Stronetta's digital time-stamping method using merkle trees is to store the root of a merkle tree that contains transactions in the block header. This optimization allows clients to easily verify transactions solely using the merkle roots of transactions in downloaded block headers without storing all the actual transactions. The second characteristic is that blocks are linked by cryptographic hashes. As demonstrated by Haber and Stornetta's work with hash linked digital timestamps, a hash linked chain of blocks prevents tampering of blocks unless an adversary has majority control of the system such that it can rewrite the entire hash linked chain. The result is a distributed ledger that is not controlled or managed by a central

entity powered by a network of connected computers that use a consensus mechanism to reach agreement over shared data[30].

5.5 Bitcoin

The first blockchain was the Bitcoin blockchain[25] powering the Bitcoin cryptocurrency system. Unlike traditional distributed systems, Bitcoin is deployed in a public setting without any participation permissions and publicly known process identities. These conditions leave distributed systems vulnerable to Sybil attacks consisting of an adversary using multiple identities to influence consensus decisions[15]. Additionally, as a global cryptocurrency system, Bitcoin operates over the public internet, an asynchronous environment. Consequently,

Bitcoin achieves Byzantine fault tolerant distributed consensus in an public, adversarial and asynchronous environment using randomization in the form of a proof-of-work consensus algorithm. Proof-of-work is based on solving moderately hard cryptographic puzzles in order to prevent computationally bounded adversaries from claiming many identites in the system[24]. More specifically, Bitcoin's proof-of-work algorithm is based on Adam Back's hashcash cost function. A subset of Bitcoin nodes, the processes of the larger distributed system, solve partial hash collisions to append blocks of transactions to the Bitcoin blockchain and update the state of the system. These nodes are commonly known as miners. This type of consensus algorithm has become commonly known as eventually consistent Nakamoto consensus[32]. Bitcoin can achieve consensus as long as an adversary does not control more than half of the total computing power. Phrased in terms of Byzantine fault tolerance, Bitcoin can achieve consensus as long as less than f computing power is Byzantine of $2f + 1$ total computing power.

Bitcoin also complements this consensus algorithm with economic rewards to incentivize miners to secure and maintain the Bitcoin blockchain. The first transaction in a new block mints new economically valuable tokens, the Bitcoin cryptocurrency, that is rewarded to the miner that successfully solved a partial hash collision and added the new block[25]. As a result, miners are economically motivated behave honestly. It is important to also note that a number of flaws in Bitcoin have been discovered over the years, but a discussion of these flaws is outside the scope of this thesis and will be left for outside research.

5.6 Ethereum

Vitalik Buterin developed Ethereum as a solution to leverage the distributed consensus capabilities of blockchains to create decentralized applications. Ethereum is a blockchain with built-in Turing complete programming language that allow users to write so called *smart contracts* that define arbitrary state transition functions[11]. Nodes in the Ethereum network run the Ethereum Virtual Machine (EVM). The value proposition of smart contracts is the ability to define arbitrary rules and agreements in a self-enforcing and self-executing program. As a result, participants in a protocol or network can trust the automatic enforced execution of code in the smart contract rather than trust some centralized entity.

Similar to Bitcoin, Ethereum uses a proof-of-work consensus algorithm. However, the core Ethereum developers have announced plans to move toward a proof-of-stake consensus algorithm in the future that would instead rely on cryptocurrency holders as validators to append new blocks to the Ethereum blockchain, incentivizing good behavior by rewarding honest validators and discouraging bad behavior by penalizing dishonest validators[33].

The role of Ethereum in the contributions for this thesis is discussed in more detail in Section 2.

6 Conclusion

Cryptocurrencies and blockchains provide the foundation for systems powered by self-organizing and self-coordinating networks of economically incentivized individuals. We built the OpenCollab-CLI command line tool and the OpenCollab protocol to leverage these building blocks with the goal of designing a sustainable open source software system that aligns the incentives of all parties involved. Ethereum provided a base blockchain layer such that we could take advantage of its underlying trust properties and infrastructure. Consequently, we focused on writing smart contract code that defines rules that incentivize open source software project community members to collaborate in improving project quality and sustainability.

We hope that at the very least the contributions of this thesis can push the discussion around open source software sustainability forward and encourage other open source software community members to experiment with new approaches of maintaining healthy and supported open source software projects. The world stands to benefit tremendously if any of these experiments are successful.

References

[1] Adam Back. *Hashcash - A Denial of Service Counter-Measure*. http://www.hashcash.org/papers/hashcash.pdf. Accessed: 2017-4-29. 2002.

[2] Dave Bayer, Stuart Haber., and W. Scott Stornetta. "Improving the Efficiency and Reliability of Digital Time-Stamping". In: *Sequences II: Methods in Communication, Security and Computer Science*. 1993, pp. 329–334.

[3] Michael Ben-Or. "Another Advantage of Free Choice (Extended Abstract): Completely Asynchronous Agreement Protocols". In: *Proceedings of the Second Annual ACM Symposium on Principles of Distributed Computing*. 1983, pp. 27–30.

[4] Juan Benet. *IPFS - Content Addressed, Versioned, P2P File System (DRAFT 3)*. https://github.com/ipfs/papers/blob/master/ipfs-cap2pfs/ipfs-p2p-file-system.pdf.

[5] Alex Beregszaszi. *Mango: Git, completely decentralized*. https://github.com/axic/mango/. Accessed: 2017-04-24.

[6] Alex Beregszaszi. *Mango Specification*. https://github.com/axic/mango/blob/master/TECH.md. Accessed: 2017-05-13.

[7] *Bounty Source: Support for Open-Source Software*. https://www.bountysource.com/. Accessed: 2017-04-24.

[8] Gilles Brassard, David Chaum, and Claude Crepeau. "Minimum Disclosure Proofs of Knowledge". In: *Journal of Computer and System Sciences* 37 (1988), pp. 156–189.

[9] Jon Brodkin. *Ubuntu Unity is dead: Desktop will switch back to GNOME next year*. https://arstechnica.co.uk/information-technology/2017/04/ubuntu-unity-is-dead-back-to-gnome/. 2017.

[10] Ethan Buchman. "Tendermint: Byzantine Fault Tolerance in the Age of Blockchains". MA thesis. The University of Guelph, 2016.

[11] Vitalik Buterin. *Ethereum White Paper: A Next-Generation Smart Contract and Decentralized Application Platform*. https://github.com/ethereum/wiki/wiki/White-Paper. 2013.

[12] L. Jean Camp, Marvin Sirbu, and J.D. Tygar. "Token and Notational Money in Electronic Commerce". In: *Proceedings of the 1st USENIX Workshop on Electronic Commerce*. 1995, pp. 1–12.

[13] Miguel Castro and Barbara Liskov. "Practical Byzantine Fault Tolerance and Proactive Recovery". In: *ACM Transactions on Computer Systems* 20.4 (2002), pp. 398–461.

[14] David Chaum. "Blind Signatures for Untraceable Payments". In: *Advances in Cryptology: Proceedings of Crypto 82*. Springer US, 1983, pp. 199–203.

[15] John R. Douceur. "The Sybil Attack". In: *Proceedings of 1st International Workshop on Peer-to-Peer Systems (IPTPS)*. 2002, pp. 251–260.

[16] Nadia Eghbal. *What success really looks like in open source*. https://medium.com/@nayafia/what-success-really-looks-like-in-open-source-2dd1facaf91c. 2016.

[17] Darrell Etherington. *Large DDoS attacks cause outages at Twitter, Spotify, and other sites*. https://techcrunch.com/2016/10/21/many-sites-including-twitter-and-spotify-suffering-outage/. Accessed: 2017-05-13. 2016.

[18] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. "Impossibility of Distributed Consensus With One Faulty Process". In: *Journal of the ACM* 32.2 (1985), pp. 374–382.

[19] *Git on the Server - The Protocols*. https://git-scm.com/book/en/v2/Git-on-the-Server-The-Protocols. Accessed: 2017-04-24.

[20] Jim Gray. "The Transaction Concept: Virtues and Limitations (Invited Paper)". In: *Proceedings of the Seventh International Conference on Very Large Data Bases*. Vol. 7. 1981, pp. 144–154.

[21] Stuart Haber and W. Scott Stornetta.
"How to time-stamp a digital document".
In: *Journal of Cryptology* 3.2 (1991), pp. 9–111.

[22] Nick Johnson. *Dividend-Bearing Tokens on Ethereum*.
https://medium.com/@weka/dividend-bearing-tokens-on-ethereum-42d01c710657. Accessed: 2017-05-13.
2017.

[23] Leslie Lamport, Robert Shostak, and Marshall Pease.
"The Byzantine Generals Problem". In: *ACM Transactions on Programming Languages and Systems* 4.3 (1982), pp. 382–401.

[24] Andrew Miller and Joseph J. LaViola Jr. *Anonymous Byzantine Consensus from Moderately-Hard Puzzles: A Model for Bitcoin*.
Tech. rep. 2014.

[25] Satoshi. Nakamoto. *Bitcoin: A Peer-to-Peer Electronic Cash System*.
https://bitcoin.org/bitcoin.pdf. 2008.

[26] Fred B. Schneider. "Implementing Fault-Tolerant Services Using the State Machine Approach: A Tutorial".
In: *ACM Computing Surveys (CSUR)* 22.11 (1990), pp. 299–319.

[27] *Swarm: Serverless Hosting Incentivised Peer-To-Peer Storage and Content Distribution*.
http://swarm-gateways.net/bzz:/theswarm.eth/.
Accessed: 2017-05-13.

[28] Nick Szabo. *Bit gold*.
http://unenumerated.blogspot.com/2005/12/bit-gold.html.
Accessed: 2017-4-29. 2008.

[29] Nassim N. Taleb and Constantine Sandis. "The Skin In The Game Heuristic for Protection Against Tail Events".
In: *Review of Behavioral Economics* 1 (2014), pp. 1–21.

[30] Peter Van Valkenburgh. *What is "Blockchain" anyway?*
https://coincenter.org/entry/what-is-blockchain-anyway. 2017.

[31] Fabian Vogelsteller. *ERC: Token Standard*.
https://github.com/ethereum/EIPs/issues/20. Accessed: 2017-4-28.

[32] Dominic Williams. *Byzantine Consensus Suitable for Decentralized Networks Using Cryptographic Randomness*.
https://drive.google.com/file/d/0B9a1K-U2gZ2KN3pYSXF6bXF0QTg/view.
Accessed: 2017-5-27.

[33] Vlad Zamfir. *Introducing Casper "the Friendly Ghost"*.
https://blog.ethereum.org/2015/08/01/introducing-casper-friendly-ghost/. Accessed: 2017-5-27.
2015.