Blockchains from a Distributed Computing Perspective

MAURICE HERLIHY, Brown University

ACM Reference Format:
Maurice Herlihy. 2018. Blockchains from a Distributed Computing Perspective. 1, 1 (January 2018), 6 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Bitcoin first appeared in a 2008 white paper authored by someone called Satoshi Nakamoto [15], the mysterious *deus abscondidus* of the blockchain world. Today, cryptocurrencies and blockchains are very much in the news. Much of this coverage is lurid, sensationalistic, and irresistible: roller-coaster prices and instant riches, vast sums of money stolen or inexplicably lost, underground markets for drugs and weapons, and promises of libertarian utopias just around the corner.

This article is a tutorial on the basic notions and mechanisms underlying blockchains, colored by the perspective that much of the blockchain world is a disguised, sometimes distorted, mirror-image of the distributed computing world.

This article is not a technical manual, nor is it a broad survey of the literature (both widely available elsewhere). Instead, it attempts to explain blockchain research in terms of the many similarities, parallels, semi-reinventions, and lessons not learned from distributed computing. This article is intended mostly to appeal to blockchain novices, but perhaps it will provide some insights to those familiar with blockchain research but less familiar with its precursors.

2 THE LEDGER ABSTRACTION

The abstraction at the heart of blockchain systems is the notion of a *ledger*, an invention of the Italian Renaissance originally developed to support double-entry bookkeeping, a distant precursor of modern cryptocurrencies. For our purposes, a ledger is just an indelible, append-only log of *transactions* that take place between various *parties*. A ledger establishes which transactions happened (*"Alice transferred 10 coins to Bob"*), and the order in which those transactions happened (*"Alice transferred 10 coins to Bob, and then Bob transferred title to his car to Alice"*). Ledgers are *public*, accessible to all parties, and they must be tamper-proof: no party can add, delete, or modify ledger entries once they have been recorded. In short, the algorithms that maintain ledgers must be *fault-tolerant*, ensuring the ledger remains secure even if some parties misbehave, whether accidentally or maliciously .

2.1 Blockchain Ledger Precursors

It is helpful to start by reviewing a blockchain precursor, the so-called *universal construction* for lock-free data structures [12].

Alice runs an on-line news service. Articles that arrive concurrently on multiple channels are placed in an in-memory table where they are indexed for retrieval. At first, Alice used a lock to synchronize concurrent access to the table, but every now and then, the thread holding the lock would take a page fault or a scheduling interrupt, leaving the articles inaccessible for too long. Despite the availability of excellent textbooks on the subject [13], Alice was uninterested in customized lock-free algorithms, so she was in need of a simple way to eliminate lock-based vulnerabilities.

She decided to implement her data structure in two parts. To record articles as they arrive, she created a *ledger* implemented as a simple linked list, where each list entry includes the article and a link to the entry before it. When an article arrives, it is placed in a shared pool, and a set of dedicated threads, called *miners* (for reasons to be explained later), collectively run a repeated protocol, called *consensus*, to select which article to append to the ledger. Here, Alice's consensus protocol can be simple: each thread creates a list entry, then calls a compare-and-swap instruction to attempt to make that entry the new head of the list.

Glossing over some technical details, to query for a recent article, a thread scans the linked-list ledger. To add a new article, a thread adds the article to the pool, and waits for for a miner to append it to the ledger.

This structure may seem cumbersome, but it has two compelling advantages. First, it is *universal*: it can implement any type of data structure, no matter how complex. Second, all questions of concurrency and fault-tolerance are compartmentalized in the consensus protocol.

A *consensus protocol* involves a collection of *parties*, some of whom are *honest*, and follow the protocol, and some of whom are *dishonest*, and may depart from the protocol for any reason. Consensus is a notion that applies to a broad range of computational models. In some contexts, dishonest parties might simply halt arbitrarily (so-called *crash* failures), while in other contexts, they may behave maliciously (so-called *Byzantine* failures). In some contexts, parties communicate through objects in a shared memory, and in others, they exchange messages. Some contexts restrict how many parties may be dishonest, some do not.

In consensus, each party proposes a transaction to append to the ledger, and one of these proposed transaction is chosen. Consensus ensures: (1) *agreement:* all honest parties agree on which transaction was selected„ (2) *termination:* all honest parties eventually learn the selected transaction, and (3) *validity:* the selected transaction was actually proposed by some party.

Consensus protocols have been the focus of decades of research in the distributed computing community. The literature contains many algorithms and impossibility results for many different models of computation (see surveys in [1, 13]).

Author's address: Maurice HerlihyBrown University, maurice.herlihy@gmail.com.

XXXX-XXXX/2018/1-ART $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

Because ledgers are long-lived, they require the ability to do *repeated* consensus to append a stream of transactions to the ledger. Usually, consensus is organized in discrete rounds, where parties start round $r + 1$ after round r is complete.

Of course, this shared-memory universal construction is not yet a blockchain, because although it is concurrent, it is not distributed. Moreover, it does not tolerate truly malicious behavior (only crashes). Nevertheless, we have already introduced the key concepts underlying blockchains.

2.2 Private Blockchain Ledgers

Alice also owns a frozen yogurt parlor, and her business is in trouble. Several recent shipments of frozen yogurt have been spoiled, and Bob, her supplier, denies responsibility. When she sued, Bob's lawyers successfully pleaded that not only had Bob never handled those shipments, but they were spoiled when they were picked up at the yogurt factory, and they were in excellent condition when delivered to Alice's emporium.

Alice decides it is time to blockchain her supply chain. She rents some cloud storage to hold the ledger, and installs internet-enabled temperature sensors in each frozen yogurt container. She is concerned that sensors are not always reliable (and that Bob may have tampered with some), so she wires the sensors to conduct a *Byzantine fault-tolerant* consensus protocol [4], which uses several rounds of voting to ensure that temperature readings cannot be distorted by a small number of of faulty or corrupted sensors. At regular intervals, the sensors reach consensus on the current temperature. They timestamp the temperature record, and add a hash of the prior record, so that any attempt to tamper with earlier records will be detected when the hashes do not match. They sign the record to establish authenticity, and then append the record to the cloud storage's list of records.

Each time a frozen yogurt barrel is transferred from Carol's factory to Bob's truck, Bob and Carol sign a certificate agreeing on the change of custody. (Alice and Bob do the same when the barrel is delivered to Alice.) At each such transfer, the signed change-of-custody certificate is timestamped, the prior record is hashed, the current record is appended to the cloud storage's list.

Alice is happy because she can now pinpoint when a yogurt shipment melted, and who had custody at the time. Bob is happy because he cannot be blamed if the shipment had melted before he picked it up at the factory, and Carol is similarly protected.

Here is a point that will become important later. At every stage, Alice's supply-chain blockchain includes identities and access control. The temperature sensors sign their votes, so voter fraud is impossible. Only Alice, Bob, and Carol (and the sensors) have permission to write to the cloud storage, so it is possible to hold parties accountable if someone tries to tamper with the ledger.

In the shared-memory universal construction, a linked list served as a ledger, and an atomic memory operation served as consensus. Here, a list kept in cloud storage serves as a ledger, and a combination of Byzantine fault-tolerant voting and human signatures serves as consensus. Although the circumstances are quite different, the "ledger plus consensus" structure is the same.

3 PUBLIC BLOCKCHAIN LEDGERS

Alice sells her frozen yogurt business and decides to open a restaurant. Because rents are high and venture capitalists rapacious, she decides to raise her own capital via an *intriguing coupon offering* (ICO): she sells digital certificates redeemable for discount meals when the restaurant opens. Alice hopes that her ICO will go viral, and soon people all over the world will be clamoring to buy Alice's Restaurant's coupons (many with the intention of reselling them at a markup).

Alice is media-savvy, and she decides that her coupons will be more attractive if she issues them as *cryptocoupons* on a blockchain. Alice's cryptocoupons have three components: a *private key*, a *public key*, and a *ledger entry* (see sidebar). Knowledge of the private key confers *ownership*: anyone who knows that private key can transfer ownership of ("spend") the coupon. The public key enables *proof of ownership*: anyone can verify that a message encrypted with the private key came from the coupon's owner. The ledger conveys *value*: it establishes the link between the public key and the coupon with an entry saying: *"Anyone who knows the secret key matching the following public key owns one cryptocoupon"*.

Suppose Bob owns a coupon, and decides to transfer half of it to Carol, and keep the other half for himself. Bob and Carol each generates a pair of private and public keys. Bob creates a new ledger entry with his current public key, his new public key, and Carol's public key, saying: *"I, the owner of the private key matching the first public key, do hereby transfer ownership of the corresponding coupon to the owners of the private keys matching the next two public keys"*. Spending one of Alice's cryptocoupons is like breaking a \$20-dollar bill into two \$10-dollar bills: the old coupon is consumed and replaced by two distinct coupons of smaller value. (This structure is called the *unspent transaction output* (UTXO) model in the literature.)

Next, Alice must decide how to manage her blockchain. Alice does not want to do it herself, because she knows that potential customers might not trust her. She has a clever idea: she will crowd-source blockchain management by offering additional coupons as a fee to anyone who volunteers to be a *miner*, that is, to do the work of running a consensus protocol. She sets up a shared bulletin board (sometimes called a *peer-to-peer* network) to allow coupon aficionados to share data. Customers wishing to buy or sell coupons post their transactions to this bulletin board. A group of volunteer *miners* pick up these transactions, batch them into *blocks* for efficiency, and collectively execute repeated consensus protocols to append these blocks to the shared ledger, which is itself broadcast over the bulletin board. Every miner, and everyone else who cares, keeps a local copy of the ledger, kept more-or-less up-to-date over the peer-to-peer bulletin board.

Alice is still worried that crooked miners could cheat her customers. Most miners are probably honest, content to collect their fees, but there is still a threat that even a small number of dishonest miners might collude with one another to cheat Alice's investors. Alice's first idea is have miners, identified by their IP addresses, vote via the *Byzantine fault-tolerant* consensus algorithm [4] used in the frozen yogurt example.

Alice quickly realizes this is a bad idea. Alice has a nemesis, Sybil, who is skilled in the art of manufacturing fake IP addresses. Sybil could easily overwhelm any voting scheme simply by flooding the protocol with "sock-puppet" miners who appear to be independent, but are actually under Sybil's control.

We noted earlier that the frozen yogurt supply chain blockchain was not vulnerable to this kind of "Sybil attack" because parties had reliable *idenities*: only Alice, Bob, and Carol were allowed to participate, and even though they did not trust one another, each one knew they would be held accountable if caught cheating. By contrast, Alice's Restaurant's cryptocoupon miners do not have reliable identities, since IP addresses are easily forged, and a victim would have no recourse if Sybil were to steal his coupons.

Essentially the same problem arises when organizing a street gang: how to ensure that someone who wants to join the gang is not a plain-clothes police officer, newspaper reporter, or just a freeloader? One approach is what sociologists call *costly signaling* [21]: the candidate is required to do something expensive and hard to fake, like robbing a store, or getting a gang symbol tattoo.

In the public blockchain world, the most common form of costly signaling is called *proof of work* (PoW). In PoW, consensus is reached by holding a *lottery* to decide which transaction is appended next to the ledger. Here is the clever part: buying a lottery ticket is a form of costly signaling because it is *costly*: expensive in terms of time wasted and electricity bills. Sybil's talent for impersonation is useless to her if each of her sock puppet miners must buy an expensive, long-shot lottery ticket.

Specifically, in the PoW lottery, miners compete to solve a useless puzzle, where solving the puzzle is hard, but proving one has solved the puzzle is easy (see sidebar). Simplifying things for a moment, the first miner to solve the puzzle wins the consensus, and gets to choose the next block to append to the ledger. That miner also receives a fee (another coupon), but the other miners receive nothing, and must start over on a new puzzle.

As hinted, the previous paragraph was an oversimplification. In fact, PoW consensus is not really consensus. If two miners both solve the puzzle at about the same time, they could append blocks to the blockchain in parallel, so that neither block precedes the other in the chain. When this happens, the blockchain is said to *fork*. Which block should subsequent miners build on? The usual answer is to build on the block whose chain is longest, although other approaches have been suggested [19].

As a result, there is always some uncertainty whether a transaction on the blockchain is permanent, although the probability that a block, once on the blockchain, will be replaced decreases exponentially with the number of blocks that follow it [8]. If Bob uses Alice's cryptocoupons to buy a car from Carol, Carol would be prudent to wait until Bob's transaction is fairly deep in the blockchain to minimize the chances that it will be displaced by a fork.

Although PoW is currently the basis for the most popular cryptocurrencies, it is not the only game in town. There are multiple proposals where cryptocurrency ownership assumes the role of costly signaling, such as Ethereum's Casper [2] or Algorand [9]. Cachin and Vukolic [3] give a comprehensive survey of blockchain consensus protocols.

3.1 Discussion

The distinction between *private* (or *permissioned*) blockchain systems, where parties have reliable identities, and only vetted parties can participate, and *public* (or *permissionless*) blockchain systems, where parties cannot be reliably identified, and anyone can participate, is critical for making sense of the blockchain landscape.

Private blockchains are better suited for business applications, particularly in regulated industries, like finance, subject to know-your-customer and anti-money-laundering regulations. Private blockchains also tend to be better at governance, for example, by providing orderly procedures for updating the ledger protocol [11]. Most prior work on distributed algorithms has focused on systems where participants have reliable identities.

Public blockchains are appealing for applications such as Bitcoin, which seek to ensure that nobody can control who can participate, and participants may not be eager to have their identities known. Although PoW was invented by Dwork and Naor [6] as a way to control spam, Nakamoto's application of PoW to large-scale consensus was a genuine innovation, one that launched the entire blockchain field.

4 SMART CONTRACTS

Most blockchain systems also provide some form of scripting language to make it easier to add functionality to ledgers. Bitcoin provides a rudimentary scripting language, while Ethereum [7] provides a Turing-complete scripting language. Such programs are often called *smart contracts* (or *contracts*) (though they are arguably neither smart nor contracts).

Here are some examples of simple contract functionality. A *hashlock* h prevents an asset from being transferred until the contract receives a matching *secret* s, where $h = H(s)$, for H a cryptographic hash function (see sidebar). Similarly, a *timelock* t prevents an asset from being transferred until a specified future time t.

Suppose Alice wants to trade some of her coupons to Bob in return for some bitcoins. Alice's coupons live on one blockchain, and Bob's bitcoin live on another, so they need to devise an *atomic cross-chain swap* protocol to consummate their deal. Naturally, neither one trusts the other.

Here is a simple protocol. Let us generously assume 24 hours is enough time for anyone to publish a smart contract on either blockchain, and for the other party to detect that that contract has been published.

- Alice creates a secret s, $h = H(s)$, and publishes a contract on the coupon blockchain with hashlock h and timelock 48 hours in the future, to transfer ownership of some coupons to Bob.
- When Bob confirms that Alice's contract has been published on the coupon blockchain, he publishes a contract on the Bitcoin blockchain with the same hashlock h but with timelock 24 hours in the future, to transfer his bitcoins to Alice.
- When Alice confirms that Bob's contract has been published on the Bitcoin blockchain, she sends the secret s to Bob's contract, taking possession of the bitcoins, and revealing s to Bob.

```
1 function withdraw(uint amount) {
2    client = msg.sender;
3    if (balance[ client ] >= amount} {
4       if ( client . call .sendMoney(amount)) {
5          balance[ client ] −= amount;
6       }}}
```

Fig. 1. Pseudocode for DAO-like contract

```
1 function sendMoney(unit amount) {
2    victim = msg.sender;
3    balance += amount;
4    victim .withdraw(amount)
5 }
```

Fig. 2. Pseudocode for DAO-like exploit

- Bob sends s to Alice's contract, acquiring the coupons and completing the swap.

If Alice or Bob crashes during steps one or two, then the contracts time out and refund their assets to the original owners. If either crashes during steps three and four, then only the party who crashes ends up worse off. If either party tries to cheat, for example, by publishing an incorrect contract, then the other party can simply halt and its asset will be refunded. Alice's contract needs a 48-hour timelock to give Bob enough time to react when she releases her secret before her 24 hours are up.

This example illustrates the power of smart contracts. There are many other uses for smart contracts, including *off-chain* transactions [16], where assets are transferred back and forth off the blockchain for efficiency, using the blockchain only to settle balances at infrequent intervals.

4.1 Smart Contracts as Objects

A smart contract resembles an object in an object-oriented programming language. A contract encapsulates *long-lived state*, a *constructor* to initialize that state, and one or more *functions* (methods) to manage that state. Contracts can call one another's functions.

In Ethereum, all contracts are recorded on the blockchain, and the ledger includes those contracts' current states. When a miner constructs a block, if fills that block with smart contracts and executes them one-by-one, where each contract's final state is the next contract's initial state. These contract executions occur in order, so it would appear that there is no need to worry about concurrency.

4.2 Smart Contracts as Monitors

The *Decentralized Autonomous Organization* (DAO) was an investment fund set up in 2016 to be managed entirely by smart contracts, with no direct human administration. Investors could vote on how the fund's funds would be invested. At the time, there were breathless journalistic accounts explaining how the DAO wold change forever the shape of investing [17, 20].

Figure 1 shows a fragment of a DAO-like contract, illustrating a function that allows an investor to withdraw funds. First, the function extracts the client's address (Line 2), then checks whether the client has enough funds to cover the withdrawal (Line 3). If so, the funds are sent to the client through an external function call (Line 4), and if the transfer is successful, the client's balance is decremented (Line 5).

This code is fatally flawed. In June 2016, someone exploited this function to steal about $50 million funds from the DAO. As noted, the expression in Line 3 is a call to a function in the client's contract. Figure 2 shows the client's code. The client's contract immediately calls withdraw() again (Line 4). This *re-entrant* call again tests whether the client has enough funds to cover the withdrawal (Line 3), and because withdraw() decrements the balance only *after* the nested call is complete, the test erroneously passes, and the funds are transferred a second time, then a third, and so on, stopping only when the call stack overflows.

This kind of *re-entrancy attack* may at first glance seem like an exotic hazard introduced by a radically new style of programming, but if we change our perspective slightly, we can recognize a pitfall familiar to any undergraduate who has taken a concurrent programming course.

First, some background. A *monitor* is a concurrent programming language construct invented by Hoare [14] and Brinch Hansen [10]. A monitor is an object with a built-in mutex lock, which is acquired automatically when a method is called and released when the method returns. (Such methods are called **synchronized** methods in Java.) Monitors also provide a wait () call that allows a thread to release the monitor lock, suspend, eventually awaken, and reacquire the lock. For example, a thread attempting to consume an item from an empty buffer could call wait () to suspend until there was an item to consume.

The principal tool for reasoning about the correctness of a monitor implementation is the *monitor invariant*, an assertion which holds whenever no thread is executing in the monitor. The invariant can be violated while a thread is holding the monitor lock, but it must be restored when the thread release the lock, either by returning from a method, or by suspending via wait () .

If we view smart contracts through the lens of monitors and monitor invariants, then the re-entrancy vulnerability looks very familiar. An external call is like a suspension, because even though there is no explicit lock, the call makes it possible for a second program counter to execute that contract's code concurrently with the first program counter. The DAO-like contract shown here implicitly assumed the invariant that each client's entry in the balance table reflects its actual balance. The error occurred when the invariant, which was temporarily violated, was not restored before giving up the (virtual) monitor lock by making an external call.

Here is why the distributed computing perspective is valuable. When explained in terms of monitors and monitor invariants, the reentrancy vulnerability is a familiar, classic concurrency bug, but when expressed in terms of smart contracts, it took respected, expert programmers by surprise, resulting in substantial disruption and embarrassment for the DAO investors, and required essentially rebooting the Ethereum currency itself [5].

4.3 Smart Contracts as Read-Modify-Write Operations

The *ERC20* token standard is the basis for many recent *initial coin offerings* (ICOs), a popular way to raise capital for an undertaking without actually selling ownership. The issuer of an ERC20 token controls token creation. Tokens can be traded or sold, much like Alice's Restaurant's coupons discussed earlier. ERC20 is a standard, like a Java interface, not a particular implementation.

An ERC20 token contract keeps track of how many tokens each account owns (the balances mapping at Line 3), and also how many tokens each account will allow to be transferred to each other account (the allowed mapping at Line 5). The approve() function (Lines 9-13) adjusts the limit on how many tokens can be transferred at one time to another account. It updates the allowed table (Line 10), and generates a blockchain event to make these changes easier to track (Line 11). The allowance() function queries this allowance (Lines 14-16).

The transferFrom function (Lines 17-23) transfers tokens from one account to another, and decreases the allowance by a corresponding amount. This function assumes the recipient has sufficient allowance for the transfer to occur.

Here is how this specification can lead to undesired behavior. Alice calls approve() to authorize Bob to transfer as many as 1000 tokens from her account to his. Alice has a change of heart, and issues a transaction to reduce Bob's allowance to a mere 100 tokens. Bob learns of this change, and before Alice's transaction makes it onto the blockchain, Bob issues a transferFrom() call for 1000 tokens to a friendly miner, who makes sure that Bob's transaction precedes Alice's in the next block. In this way, Bob successfully withdraws his old allowance of 1000 tokens, setting his authorization to zero, and then, just to spite Alice, he withdraws his new allowance of 100 tokens. In the end, Alice's attempt to reduce Bob's allowance from 1000 to 10 made it possible for Bob to withdraw 1100 tokens, which was not her intent.

In practice, ERC20 token implementations often employ *ad-hoc* workarounds to avoid this vulnerability, the most common being to redefine the meaning of allow() so that it will reset an allowance from a positive value to zero, and in a later call, from zero to the new positive value, but will fail if asked to reset an allowance from one positive value to another.

The problem is that approve() blindly overwrites the old allowance with the new allowance, regardless of whether the old allowance has changed. This practice is analogous to trying to implement an atomic decrement as shown in Figure 4. Here, the decrement method reads the shared counter state into a local variable (Line 4), increments the local variable (Line 5), and stores the result back in the shared state (Line 6). It is not hard to see that this method is incorrect if it can be called by concurrent threads, because the shared state can change between when it was read at Line 4 and when it was written at Line 6. When explained in terms of elementary concurrent programming, this concurrency flaw is obvious, but when expressed in terms of smart contracts that ostensibly do not need a concurrency model, the same design flaw was immortalized in a token standard with a valuation estimated in billions of dollars.

4.4 Discussion

We have seen that the notion that smart contracts do not need a concurrency model because execution is single-threaded is a dangerous illusion. Sergey and Hobor [18] give an excellent survey of pitfalls and common bugs in smart contracts that are disguised versions of familiar concurrency pitfalls and bugs. Atzei *et al.* provide a comprehensive survey of vulnerabilities in Ethereum's smart contract design.

5 CONCLUSIONS

Radical innovation often emerges more readily from outside an established research community than from inside. Would Nakamoto's original Bitcoin paper have been accepted to one of the principal distributed conferences back in 2008? We will never know, of course, but the paper's lack of a formal model, absence of rigorous proofs, and lack of performance numbers would have been a handicap.

Today, blockchain research is one of the more vibrant areas of computer science, with the potential of revolutionizing how our society deals with trust. The observation that many blockchain constructs have underacknowledged doppelgängers (or at least, precursors) is not a criticism of either research community, but rather an appeal to each side to pay more attention to the other.

6 SIDEBAR: PUBLIC AND PRIVATE KEYS

Modern cryptography is based on the notions of matching *public* and *private* keys. Any string encrypted by one can be encrypted by the other. Encrypting a message with Alice's public key yields a message only Alice can read, and encrypting a message with Alice's private key yields a *digital signature*, a message everyone can read but only Alice could have produced.

7 SIDEBAR: CRYPTOGRAPHIC HASH FUNCTION

A *cryptographic hash function* $H(\cdot)$ has the property that for any value v, it is easy to compute $H(v)$, but it is infeasible to discover a $v' \neq v$ such that $H(v') = H(v)$.

8 SIDEBAR: PROOF OF WORK PUZZLES

Here is puzzle typical of those used in PoW implementations. Let b be the block the miner wants to append to the ledger, $H(\cdot)$ a cryptographic hash function, and "$\cdot$" concatenation of binary strings. The puzzle is to find a value c such that $H(b \cdot c) < D$, where D is a *difficulty setting* (the smaller D, the more difficult). Because H is difficult to invert, there is no way to find c substantially more efficient than exhaustive search.

REFERENCES

[1] Hagit Attiya and Jennifer Welch. 2004. *Distributed Computing: Fundamentals, Simulations and Advanced Topics.* John Wiley & Sons.

[2] Vitalik Buterin and Virgil Griffith. 2017. Casper the Friendly Finality Gadget. https://github.com/ethereum/research/commits/master/papers/casper-basics/casper_basics.pdf. (sep 2017). Accessed: 6 January 2018.

[3] Christian Cachin and Marko Vukolic. 2017. Blockchain Consensus Protocols in the Wild. *CoRR* abs/1707.01873 (2017). arXiv:1707.01873 http://arxiv.org/abs/1707.01873

[4] Miguel Castro and Barbara Liskov. 1999. Practical Byzantine Fault Tolerance. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation (OSDI '99).* USENIX Association, Berkeley, CA, USA, 173–186. http://dl.acm.org/citation.cfm?id=296806.296824

```
contract ERC20Example {
  // Balances for each account
  mapping(address => uint256) balances;
  // Owner of account approves the transfer of an amount to another account
  mapping(address => mapping (address => uint256)) allowed;
  // other fields omitted
   ...
  // Allow spender to withdraw from your account, multiple times, up to the amount.
  function approve(address spender, uint amount) public returns (bool success) {
    allowed[msg.sender][spender] = amount; // alter approval
    Approval(msg.sender, spender, amount); // blockchain event
    return true;
  }
  function allowance(address tokenOwner, address spender) public returns (uint remaining) {
    return allowed[tokenOwner][spender];
  }
  function transferFrom(address from, address to, uint tokens) public (bool success) {
    balances[from] = balances[from].sub(tokens);
    allowed[from][msg.sender] = allowed[from][msg.sender].sub(tokens);
    balances[to] = balances[to].add(tokens);
    Transfer(from, to, tokens);
    return true;
  }
   ... // other functions omitted
}
```

Fig. 3. ERC20 Token example

```
class Counter {
  private int counter;
  public void dec() {
    int temp = counter;
    temp = temp + 1;
    counter = temp;
   }
   ...
}
```

Fig. 4. An incorrect atomic decrement operation

[5] Michael del Castillo. 2016. Ethereum Executes Blockchain Hard Fork to Return DAO Funds. https://www.coindesk.com/ethereum-executes-blockchain-hard-fork-return-dao-investor-funds/. (July 2016). Accessed: 6 Januar 2018.

[6] Cynthia Dwork and Moni Naor. 1993. *Pricing via Processing or Combatting Junk Mail*. Springer Berlin Heidelberg, Berlin, Heidelberg, 139–147. https://doi.org/10.1007/3-540-48071-4_10

[7] Ethereum. [n. d.]. https://github.com/ethereum/. ([n. d.]).

[8] Juan Garay, Aggelos Kiayias, and Nikos Leonardos. 2015. *The Bitcoin Backbone Protocol: Analysis and Applications*. Springer Berlin Heidelberg, Berlin, Heidelberg, 281–310. https://doi.org/10.1007/978-3-662-46803-6_10

[9] Yossi Gilad, Rotem Hemo, Silvio Micali, Georgios Vlachos, and Nickolai Zeldovich. 2017. Algorand: Scaling Byzantine Agreements for Cryptocurrencies. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP '17)*. ACM, New York, NY, USA, 51–68. https://doi.org/10.1145/3132747.3132757

[10] Per Brinch Hansen. 1973. *Operating System Principles*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA.

[11] Mike Hearn. [n. d.]. The resolution of the Bitcoin experiment. ([n. d.]). https://blog.plan99.net/the-resolution-of-the-bitcoin-experiment-dabb30201f7.

[12] Maurice Herlihy. 1991. Wait-free Synchronization. *ACM Trans. Program. Lang. Syst.* 13, 1 (Jan. 1991), 124–149. https://doi.org/10.1145/114005.102808

[13] Maurice Herlihy and Nir Shavit. 2008. *The Art of Multiprocessor Programming*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[14] C. A. R. Hoare. 1974. Monitors: An Operating System Structuring Concept. *Commun. ACM* 17, 10 (Oct. 1974), 549–557. https://doi.org/10.1145/355620.361161

[15] Satoshi Nakamoto. 2009. Bitcoin: A Peer-to-Peer Electronic Cash System. (May 2009). http://www.bitcoin.org/bitcoin.pdf

[16] J. Poon and T. Dryja. 2016. The Bitcoin Lightning Network: Scalable Off-Chain Instant Payments. https://lightning.network/lightning-network-paper.pdf. (Jan. 2016). Accessed: 29 December 2017.

[17] Nathaniel Popper. 2016. A venture fund with plenty of virtual capital, but no capitalist. New York Times. (man 2016). https://www.nytimes.com/2016/05/22/business/dealbook/crypto-ether-bitcoin-currency.html.

[18] Ilya Sergey and Aquinas Hobor. 2017. A Concurrent Perspective on Smart Contracts. *CoRR* abs/1702.05511 (2017). arXiv:1702.05511 http://arxiv.org/abs/1702.05511

[19] Yonatan Sompolinsky, Yoad Lewenberg, and Aviv Zohar. 2016. SPECTRE: A Fast and Scalable Cryptocurrency Protocol. Cryptology ePrint Archive, Report 2016/1159. (2016). http://eprint.iacr.org/2016/1159.pdf Accessed: 2017-02-20.

[20] Paul Vigna. 2016. hiefless Company Rakes In More Than $100 Million. Wall Street Journal. (may 2016). https://www.wsj.com/articles/chiefless-company-rakes-in-more-than-100-million-1463399393.

[21] Wikipedia. [n. d.]. Signalling Theory. https://en.wikipedia.org/wiki/Signalling_theory. ([n. d.]). Accessed: 3 January 2018.