SN Computer Science (2020) 1:251
https://doi.org/10.1007/s42979-020-00243-7

ORIGINAL RESEARCH

A Blockchain-based Decentralized Electronic Marketplace for Computing Resources

Matteo Nardini[1] · Sven Helmer[2] · Nabil El Ioini[1] · Claus Pahl[1]

Received: 1 April 2020 / Accepted: 24 June 2020 / Published online: 6 August 2020

Abstract

We propose a framework for building a decentralized electronic marketplace for computing resources. The idea is that anyone with spare capacities can offer them on this marketplace, opening up the cloud computing market to smaller players, thus creating a more competitive environment compared to today's market consisting of a few large providers. Trust is a crucial component in making an anonymized decentralized marketplace a reality. We develop protocols that enable participants to interact with each other in a fair way and show how these protocols can be implemented using smart contracts and blockchains. We discuss and evaluate our framework not only from a technical point of view, but also look at the wider context in terms of fair interactions and legal implications.

Keywords Decentralized electronic marketplaces · Computing resources · Blockchains

Abbreviations

ABI:	application binary interface
Berkeley Open Infrastructue for Network Computing: BOINC	
CGI:	computer-generated imagery
cgroups:	control groups
CSS:	cascading style sheet
DApp:	decentralized application
DRIVE:	Distributed Resource Infrastructure for a Virtual Economy
e-marketplace:	electronic marketplace
ETH:	Ether
EVM:	Ethereum Virtual Machine
GNT:	Golem Network Token
HTML:	HyperText Markup Language
IoT:	Internet of Things
IT:	information technology
JSON-RPC:	JavaScript Object Notation remote procedure call
ODR:	online dispute resolution
OS:	operating system
PCP:	probabilistically checkable proofs
seccomp:	secure computing
SGX:	Intel Software Guard Extensions
SHA-256:	secure hash algorithm 256 bit
SONM:	Supercomputer Organized by Network Mining
TTP:	trusted third party
VM:	virtual machine

✉ Sven Helmer
helmer@ifi.uzh.ch

Matteo Nardini
matteo.shalen@gmail.com

Nabil El Ioini
Nabil.ElIoini@unibz.it

Claus Pahl
Claus.Pahl@unibz.it

[1] Faculty of Computer Science, Free University of Bozen-Bolzano, Piazza Domenicani 3, 39100 Bolzano, Italy

[2] Department of Informatics, University of Zurich, Binzmühlestrasse 14, 8050 Zurich, Switzerland

Introduction

In the last decade we have witnessed the emergence of the *sharing economy*, in which persons grant access to assets they own to others [1] (this has also been called an *access economy* [2]). It affects areas as diverse as lodging (Airbnb), transport (car and bike sharing schemes, Uber), parking spaces (JustPark), and labor (timebanks), just to name a few. In information technology (IT), cloud computing has had a big impact on how people purchase computational power. Instead of setting up and maintaining their own infrastructure, many users and organizations turn to cloud providers.

In large parts of the sharing economy we see transactions taking place on a peer-to-peer level. While there are peer-to-peer-based approaches in IT, such as open source software or volunteer computing, the cloud computing market is dominated by large players such as Amazon, Google, and Microsoft. As Subramanian points out, electronic marketplaces (or, short, e-marketplaces) controlled by firms come with certain downsides [3]. Companies are primarily interested in maximizing their profits rather than matching buyers to the products or services they really need. If there are only a few players, this can lead to price-fixing or even monopolies [4]. Additionally, all payments have to go through trusted third parties, adding an overhead to the transactions.

In decentralized e-marketplaces the matching of buyers and sellers could be done in a more transparent way: a buyer has more options to choose from, increasing the likelihood of finding a good match. Also, the payments could go directly from buyer to seller without passing through third parties. So, why do we not see peer-to-peer e-marketplaces for computational power? One important reason is a lack of trust. The large providers have established a good reputation and there are also legal frameworks in place to protect customers. It is not easy for smaller entities to enter this market due to high barriers, such as setting up sufficient infrastructure and earning a reputation.

Looking at the numbers for volunteer computing, e.g. the Berkeley Open Infrastructure for Network Computing (BOINC) [5], has convinced us that there is a potential market to be found here: according to Wikipedia [6], as of 9 June 2018 there were 311,742 active participants with 834,343 active hosts processing on average 26.431 PetaFLOPS. This is roughly comparable to the computational power of the Tianhe-2 supercomputer, which was the world's fastest computer at the time of its introduction in June 2013 [7]. Given the right financial incentives, we believe that there is an even greater number of people who would allow access to their computational devices when they are currently idle. The first steps in this direction are already taken by projects such as Golem [8], iExec [9], and SONM [10]. With the advent of the Internet of Things (IoT), for an example see [11], we expect the number of devices whose computational power is underutilized to rise dramatically, making it more and more attractive to monetize these resources.

Our goal is to provide a platform for a decentralized market that matches participants with computational needs with those providing computational power. We assume that we operate in a peer-to-peer setting in which the participants neither know nor trust each other. On the one hand, we cannot be sure that the computations are executed in a proper and reliable way and, on the other hand, we have no guarantees that the code used for the computations will not have malicious side effects. In order to deal with the trust issues, we propose to use blockchain technology as a basis for handling interactions between the participants. In particular, we make the following contributions:

- We develop protocols for managing the interactions of different parties in an e-marketplace for computational power. These protocols also cover cases in which disputes can arise and include ways to resolve them.
- The protocols run on top of a blockchain and we rely on smart contracts to enforce the terms and conditions agreed upon by the participants.
- For a start, we focus on deterministic computations that do not involve network connectivity or inter-process communication and propose to use portable container images as a light-weight, stand-alone, executable package containing the software executing the computations.
- We implement a prototype as a proof of concept using the Ethereum blockchain and the Docker container platform. We evaluate our approach and discuss its advantages and shortcomings.

Related Work

Although there is some overlap between volunteer computing [12] and electronic marketplaces for computing resources, volunteer computing uses different principles, such as a master-worker parallel computing model, in which a master node breaks down tasks into smaller chunks, distributes them among worker nodes, and then collects the results [13–15]. This is far from the decentralized set-up we envision, as the master is in full control of the process. Also, the participants donate their computational power without expecting a financial reward. We are looking for an approach that attracts users by providing financial incentives.

A number of projects proposing decentralized electronic marketplaces for computing resources are currently underway to fill this gap. However, unlike us, none of them envision a fully decentralized platform in which participants remain anonymous. Probably closest to our approach comes iExec [9], which is an Ethereum-based platform with the goal of building a marketplace for generic cloud-computing resources. However, in order to make this work, iExec relies on a form of reputation-based system, allowing users to choose partners according to "their provable reputation" [9]. The work done by less trusted nodes is replicated and the final outcome is decided via a majority vote. The importance of a node in a vote is determined by the node's reputation and the value of its security deposit. A proof-of-contribution mechanism is used to implement this scheme. We take a closer look at particular issues of replication of computations and reputation-based systems in Sections 2.1.1 and 2.3, respectively. There we explain why these mechanisms are problematic, which basically comes down to too

much overhead for replicating computations and the challenges of tracking reputation in an anonymous setting. These are the reasons we do not use replication and reputation in our approach, which sets our work apart from iExec.

Golem [8] and SONM [10] are two other electronic marketplace platforms that differ from iExec by not providing general-purpose services: Golem is (currently) limited to CGI rendering, while SONM focuses on fog and edge computing. Golem proposes a platform where users are paid in Golem Network Tokens (GNT) in exchange for their computing resources. Applications get certified by validators in order to make them more trustworthy. Users build community-driven trust networks by blacklisting and whitelisting other participants and/or applications. So, essentially Golem runs a form of reputation-based framework with all the drawbacks of such an approach. There is also a verification process relying on verifying specific parts of the rendered images. Currently, the plan is to introduce Intel Software Guard Extensions (SGX) into the system to increase its trustworthiness. We look at the particular issues of trusted hardware in Section 2.2, showing that this approach requires a wide-spread deployment of trusted hardware among all participant in the marketplace, which we do not expect to happen in the near future. Finally, there is the Supercomputer Organized by Network Mining (SONM) system for renting out fog and edge computing resources on a decentralized computing platform. All suppliers (and other participants) have a profile and a rating, utilizing different status levels (from strongest to weakest): professional, identified, registered, and anonymous. Thus, this is similar to iExec, as it relies on a reputation-based mechanism and the recomputation of tasks to increase trust.

Chard and Bubendorfer developed the Distributed Resource Infrastructure for a Virtual Economy (DRIVE), to support an open cloud market [16]. Their work is complementary to ours, as it focuses on the allocation of resources and negotiating prices in an untrusted decentralized environment (e.g. via auctions), not on running the infrastructure for executing the actual jobs.

In the following, we look at different aspects of implementing and running a decentralized electronic marketplace.

Checking results

When outsourcing computations to possibly unreliable systems, there is a need to check if the returned results are actually correct. The source of this may not necessarily be a malicious operator, incorrect results can also be caused by hardware or software faults or misconfiguration issues. Here we look at two different approaches, replication and verifiable computing, to solve the problem of checking whether a computation was done correctly or not.

Replication

Replicating computing tasks (and data) has long been used in fault-tolerant systems and involves the redundant execution of the same task on multiple CPUs or devices in order to detect faulty computations and resolve conflicts [17]. The resolution of conflicts usually involves a voting scheme in which a majority decides on the correct result. Mission-critical systems such as aircraft, spacecraft, and nuclear facility controls often employ fault-tolerant approaches [18, 19]. It is also popular in the context of volunteer computing, in which the same task is given to multiple workers to detect faulty computations [14, 15, 20].

While replication works in such an environment as the computational power is basically free, in an entrepreneurial setting this would be too expensive. Dong et al. estimate that moving computations to the cloud results in cost savings of around 50% to 70% [21]. Executing many tasks redundantly would eat up a lot of these savings. Nevertheless, Dong et al. use a scheme that combines replication with a factor of two, i.e., running each task twice, with game-theoretic concepts to create incentives for the cloud providers not to cheat [21]. Apart from the overhead caused by the duplication, there are a few more weaknesses. They assume that the providers are separate entities, each trying to maximize their own profits. We assume a peer-to-peer network with anonymous actors, meaning that multiple providers could be controlled by the same entity and that it is not unlikely that all bidders for a job are in fact controlled by the same entity (if bidding takes place right after this entity has learned of the new job). Moreover, the framework assumes that the client is honest. Again, in a peer-to-peer network we cannot assume this. We may encounter clients trying to game the system.

Verifiable Computing

Verifiable computing takes another approach. In addition to the result of the computation, the system performing the computation provides some more information with which the client can verify that the computation was done correctly. The two parties are called *prover* and *verifier* in the context of verifiable computing. The verifier asks the prover to perform a computation, the prover executes it and then tries to show to the verifier: (1) that the executed computation was actually the one requested and (2) that the executed computation was executed correctly. Early work in this area showed that this is theoretically possible, one of the groundbreaking results was on probabilistically checkable proofs (PCP) [22–25]. However, it was also shown that the costs are too prohibitive to use this scheme in practice: it requires exponential time on the side of the prover. Some more recent breakthroughs have brought down the costs considerably. For instance, Muggles achieves polynomial complexity for

the prover, albeit for specific types of computations expressible as certain kinds of circuits [26]. Other approaches, such as fully homomorphic encryption and non-interactive protocols are still not feasible, even after reducing the costs [27]. Generally, while verifiable computing is getting closer to its goals, i.e., there are actual implementations now, it is still not usable in practice [28]. The biggest open issues at the moment are how to make verifiable computing work for general-purpose programming languages and how to reduce the overhead for the prover even further (the currently implemented systems only work for toy examples).

TrueBit is a more recent approach [29], based on a so-called consensus computer [30]. The results of computational tasks generated by *solvers* are checked by *verifiers* in a multi-round verification game. A verifier can earn a reward by challenging an incorrect computation. The verification game follows a versatile and elaborate protocol, e.g. in order to provide enough incentives, the framework introduces *forced errors* to make sure that verifiers are able to find a sufficient number of errors to make their effort worthwhile. However, to attract enough verifiers, i.e., at least one per tasks, requires a sufficient payoff: the authors of [29] estimate that a verification tax of 500% to 5000% of the cost of performing a given task is necessary. This is a considerable overhead, and as mentioned in Section 2.1.1 running each task twice would already eat up the savings made by moving to the cloud.

For an overview on verifiable computing, see [28].

Trusted Hardware

Relying on trusted hardware [31] shifts the issue of trust from the cloud providers to the manufacturers and vendors of that hardware. This approach creates a different chain of trust and raises the barrier of entry even further, as all the players in the e-marketplace need to acquire specialized hardware and be able to satisfactorily show that they use this hardware. Also, faults in the hardware can still lead to incorrect results without any malicious intent by any party.

Reputation-based Systems

A commonly used technique to increase trust in centralized e-marketplaces, such as Amazon or Ebay, is a reputation-based scheme. Users of the marketplace leave feedback or write reviews on their experiences interacting with other users. This feedback is publicly accessible and helps other participants in deciding whether to go ahead with a transaction or not: if the other party has received a lot of positive feedback, it makes them more trustworthy [32].

Reputation-based systems are not foolproof, though. Typical issues are users inflating their reputational score by engaging in fake transactions and/or colluding with others by providing positive reviews to each other. Another type of fraudulent behavior is the exploitation of a good reputation that has been built up via many small legitimate transactions and then interacting maliciously on a large scale with unsuspecting customers. Post et al. developed Bazaar, a system that strengthens reputation in online marketplaces by keeping track of transactions in a *risk network* (modeled as a graph), which creates links between all users who have interacted with each other in the past [33]. The graph uses edge weights that summarize the total monetary volume of successful transactions between two users and Bazaar utilizes this information to calculate the max-flow between two users who want to make a deal. The trade can only go ahead if the value of the transaction is below the max-flow, limiting the potential damage. While this scheme has its merits, it is difficult to implement in an anonymous decentralized setting, as a single entity could control multiple accounts and use the techniques described above to boost their reputation.

Introducing a reputation-based scheme into an anonymous setting creates further problems, effectively making it unusable in our framework. Users whose reputation drops to a low level can re-enter the market under a new identity, getting rid of their previous track record. Malicious participants can stage Sybil attacks by simply creating and controlling multiple fake identities, i.e., there is no need to collude with other entities anymore to inflate ratings. Soska et al. attach a small cost to each transaction that has to be paid in order to generate feedback [34]. This makes the feedback more credible, since it provides a lower bound for the total sum that was spent creating the feedback. While this scheme makes the reputation more resilient against Sybil attacks, it does not prevent them completely, especially if side-payments are possible between colluding parties.

Casey et al. have applied game-theoretic concepts to establish identity in anonymous settings [35]. On the one hand, participating players want to preserve their privacy, but, on the other hand, there is a need to manage (pseudonymous) identities in many scenarios. Signaling games, which make it costly for a deceptive agent to fail a challenge questioning their identity, are at the core of this approach [36]. Although new insights have been gained recently and the problem has been formalized mathematically [35], there are still open questions that need to be answered before this technique can be used in practice.

Other Approaches

Spot-checking is a technique that deploys special jobs, so-called *spotter jobs*, whose sole purpose it is to check that a service provider is doing their work properly. The result of the computation is already known by the client in advance [14]. A weaker version of this is running a heartbeat protocol to check if the application is actually running [37].

However, these approaches assume that the additional jobs are not detectable and modifiable by cloud providers. In principle, they follow a security by obscurity approach, which makes them unreliable.

An approach merging a heartbeat protocol with verifiable computing is proposed by Khan and Hamlen [38]. They suggest to periodically checkpoint the computation state: these checkpoints can then be used to re-execute the entire computation in parallel, reducing the time needed for the check. However, this technique is based on basically re-executing the entire computation, which is not desirable in our case.

Premnath and Haas [39] describe an interesting application based on the idea of garbled circuits, which allows executing a computation in a way that preserves privacy and, as a side effect, is (partially) verifiable. While this would solve most of our problems, the time and storage costs are, similarly to the verifiable computing approaches, too high.

Klems et al. develop Desema, their DEcentralized SErvice MArketplace prototype, introducing trustless intermediation between the participants based on a blockchain [40]. Some of the ideas they present, such as smart contracts and deposits made by service providers, are similar to aspects we have integrated into our approach. However, it is not quite clear how much of this is actually designed and implemented, as the authors often use the subjunctive when describing their approach.

Problem Specification and Constraints

We now specify which criteria a fully decentralized e-marketplace for computational power needs to satisfy in order to function properly. The transactions have to be validated, traceable, and made persistent for users to be able to trust the marketplace. Additionally, there should be a fair exchange of money for computational services, i.e., one side should not be able to cheat the other. Finally, we have to make sure that the e-marketplace is usable for a wide range of users, even though they may rely on diverse, heterogeneous infrastructures.

What we are currently not covering are privacy aspects and matching buyers and sellers. In this approach we do not yet look at how to secure the code and data sent to a cloud provider to keep it private. So, at the moment this approach is not suitable for processing sensitive data. Also, apart from sketching how to publish and advertise tasks up for computation we do not explicitly discuss how buyers and sellers find each other. In the following we take a closer look at the criteria we do cover.

Decentralized E-Marketplaces

Our goal is to create a fully decentralized e-marketplace, as this has several advantages. First, there is no single point of failure: even if individual servers break down, the overall market is not affected. Second, this lowers the barriers to entry to the cloud computing market and opens it up to many smaller players, preventing monopolies or collusion among a few large players. Finally, we want to keep the involvement of trusted third parties to a minimum and rely on self-enforcing protocols wherever we can. Preventing or resolving disputes among the participants automatically will help in keeping the costs down.

Secure Transactions and Trust

For transparency, we need to be able to keep track of all the transactions in the e-marketplace. More concisely, this means checking them for validity, so that participants cannot create fake transactions. They also need to be made persistent in a way such that they cannot be changed or forged afterwards. Finally, the transactions need to be traceable, so it is clear who entered a contract with whom and who is responsible when things start going wrong.

A crucial aspect of a decentralized e-market is trust, as we expect participants who do not know each other to collaborate. Assume we have two parties, let us call them Alice and Bob, who want to exchange money in return for a good (or a service). In the physical world this is not an issue: Alice enters Bob's store and exchanges her money for whatever Bob is offering. As both are physically present, they can monitor what is going on. In an anonymous digital setting this becomes more complicated. If Alice first transfers the money, she runs the risk of not getting anything in return, and if Bob first provides the service he may end up not getting paid. This scenario is not new and *fair exchange protocols* have been proposed as solutions for this problem [41–43], which are about the efficient and fast exchange of electronic data between two parties that do not necessarily trust each other. Early work in this area started out by looking at the simultaneous exchange of secrets or gradually releasing a secret [44, 45]. Ideally, we would like to do this without relying on a trusted third party, but studies have shown that it is impossible to solve the general problem without one [46–48]. Consequently, there is a lot of work focusing on minimizing the influence and impact of this third party: these approaches are called *optimistic fair exchange protocols* [42, 49–51]. A common incentive to keep participants honest is to punish a misbehaving party by inflicting a monetary loss on them [51].

Heterogeneity of Systems

Another issue we have to deal with is the heterogeneity of systems in a peer-to-peer network. We cannot assume that all participants use the same configuration, let alone the same operating system or hardware. We have to be able to deal with service providers using a wide range of devices and machines with differences in computational power, memory capacity, and processing capabilities. Also, we need to limit the privileges of the code shipped to a provider in order not to compromise their system. The code should run isolated from the host system in a sand-boxed environment and should not be able to exhaust all the resources of the host system. At the same time, it should be easy to create, use, and share code and a user should be able to flexibly configure the execution environment of their code.

Preliminaries

For the purpose of self-containment we give a brief introduction to blockchains and container architectures before delving into the technical details of our solution. Readers who are already familiar with these technologies can skip this section.

Blockchains and Smart Contracts

The basic idea of a *blockchain*[1] is to create a digital ledger that records all transactions executed by the participants in an immutable and secure way. It does so with the help of a decentralized storage mechanism that maintains a continuously growing list of records, grouped into structures called blocks. Each block of the blockchain contains records of transactions, the hash of the previous block, and a timestamp. This chain of hash values ensures the immutability of the records, as changing a block either invalidates the chain or the entire chain from that point on must be recomputed, which is prohibitively expensive.

The system is maintained by a peer-to-peer network, each node of which collects transactions, joins them in a new block, and validates this block. The block validation is usually implemented with the help of a proof-of-work scheme, e.g. in the form of a cryptographic puzzle that is (moderately) hard to solve, but whose answer is easy to check. This also randomizes which node actually gets to validate a block (the node who solves the puzzle first). A node that successfully validates a block is rewarded with currency tokens usable in the blockchain. As long as a majority of the nodes adhere to the protocol rules, invalid extensions and tampering will be detected by the peer-to-peer network and rejected.

Operating in a peer-to-peer environment means that the (anonymous) participants do not trust each other. The trust in a blockchain is established by a combination of cryptographic protocols securing the ledger and incentives to keep the maintainers honest [53].

A *smart contract* is a function, represented by a piece of code, that resides on the blockchain and can be executed by the nodes of the peer-to-peer network. It extends the idea of putting data in a secure ledger to computation [54]. The distributed consensus protocol enforces the correct execution of the code: each node runs the function locally and checks that it gets the same results as the other nodes before validating it. For instance, a smart contract could check that certain conditions are met before going ahead with a transaction. For instance, if the transaction involved a monetary transfer, the smart contract would basically act as an escrow service.

The expressiveness of smart contracts depends on the employed language, Ethereum uses Solidity, a programming language influenced by C++, Python, and JavaScript [55]. Functions written in Solidity are compiled into byte code and executed on the Ethereum Virtual Machine (EVM).

The blockchain framework allows users to create decentralized applications (DApps) that are stored and executed on the blockchain and inherit all the properties provided by a blockchain environment: all the nodes agree on the current state of the various DApps and the history of each modification to the state is recorded on an append-only ledger. In principle, we implement our decentralized e-marketplace for computational power as a DApp running on a blockchain.

Containers

Deploying applications in heterogeneous environments by distributing its binaries is fraught with all kinds of problems. For instance, it is not clear whether the target system meets all the requirements of the application in terms of the operating system (OS), libraries, or other resources. *Containers* are one solution to these issues. A container holds packaged self-contained, ready-to-deploy parts of applications and, if necessary, middleware and business logic (in binaries and libraries) to run applications. With containers, applications share an OS (and possibly binaries and libraries). As a consequence, their deployments can be substantially smaller in size than hypervisor deployments traditionally used in cloud environments. This enables to store large numbers of containers on a physical host since containers use the host OS. More importantly, restarting a container does not require rebooting the OS, launching a hypervisor, on the other hand, requires initializing a whole OS.

[1] For more details on blockchains and a comprehensive introduction we recommend [52].

Containers are based on layers composed from individual images built on top of a base image that can be extended. Complete images form portable application containers, which can also be used as building blocks for application stacks. The approach is lightweight as single images can be changed and distributed easily. Additional system capabilities can be added or the access to system resources can be limited. A container ecosystem consists of an application container engine to run images and a repository or registry operated via push and pull operations to transfer images to and from host-based engines.

Docker, which is open-source and was released in 2013, is one of the most well-known and successful containerization frameworks [56]. It allows independent containers to run on a single Linux instance, relying on the host's kernel functionality in an isolated view of the host's operating system. Additionally, the containers are not aware of other containers running on the same kernel. Docker started on Linux platforms, but in the meantime has also been made available for Windows and MacOS.

Our Solution

After dealing with all the preliminaries, we now present the design, protocols, and implementation of our electronic marketplace. Before describing the protocols, we give an overview and then sketch the overall architecture and provide implementation details.

Overview

We start off by giving a general overview of our approach, introducing the different parties: we call them *publishers*, *farmers*, and *auditors*. Publishers are the entities who need someone to execute code for them in order to obtain the results of some computation. For that purpose, they publish a description of the task together with the code and a financial reward. Farmers[2] are the entities offering computational infrastructure and are willing to complete the publishers' tasks. As already mentioned, a completely self-enforcing protocol is out of reach, but we also do not want to run an arbitrated protocol in which the trusted third party (TTP) is involved in every step; so, we settled for an adjudicated protocol, in which the TTP only intervenes in case of a dispute [57]. We call the TTP an auditor in our protocol.

With the help of an auditor we implement an optimistic fair exchange protocol. While the concept of such a protocol is not new, implementing an optimistic fair exchange with cryptocurrencies, especially with smart contracts, is still not common. Utilizing protocols based on the Bitcoin blockchain is more complicated and convoluted than it needs to be in our case [58, 59], because this kind of blockchain does not directly support powerful smart contracts, which would have to be emulated in some way to get the same effect.

Liu et al. [60] and Klems et al. [40] propose approaches that consider more general smart contracts, such as the ones offered by Ethereum. While Liu et al. investigate a much simpler scenario compared to ours, i.e., the exchange of purely digital assets (which boils down to getting a receipt in the form of a digital signature in return for a payment), they identify a set of properties important for benchmarking the quality of fair exchange protocols. We come back to these properties when evaluating our own protocol in Section 7.2. Klems et al., on the other hand, look at a more complex scenario. Their vision is to offer a platform on which a customer can subscribe to an on-going service, which makes it necessary to constantly monitor the quality and integrity of the service. For that purpose, and some other tasks, such as dispute resolution, supporting actors providing functionality going beyond the capabilities of smart contracts are introduced. All the different supporting actors need to be integrated into the framework and need to be offered (financial) incentives, which adds to the cost.

Protocols

Standard scenario

The sequence diagram in figure 1 describes the standard scenario, in which both the publisher and the farmer behave correctly.

As can be seen from the diagram, the process is initiated by the publisher, who interacts with the smart contract to publish a new computation request. This request describes the computational task and its parameters and also transfers the reward for completing the computation to the account of the smart contract. After validating the publisher's input (e.g. here we could check a signed hash of the submitted code)[3], the smart contract emits a `ComputationPublished` event to announce the publication of a new publisher request. The event is broadcast on the blockchain, where farmers can pick it up.

A farmer receiving such an event has to decide whether to accept the request or not. In the case a farmer is willing to perform the computation, they reserve it by sending a request to the smart contract, which first checks whether the task is still available. If this is the case, the contract

[2] As the term *miner* is already used for worker nodes in the context of blockchains, we settled for a similar yet different term.

[3] Validation in this context means checking that the code was not modified or corrupted during transmission or by the publisher.

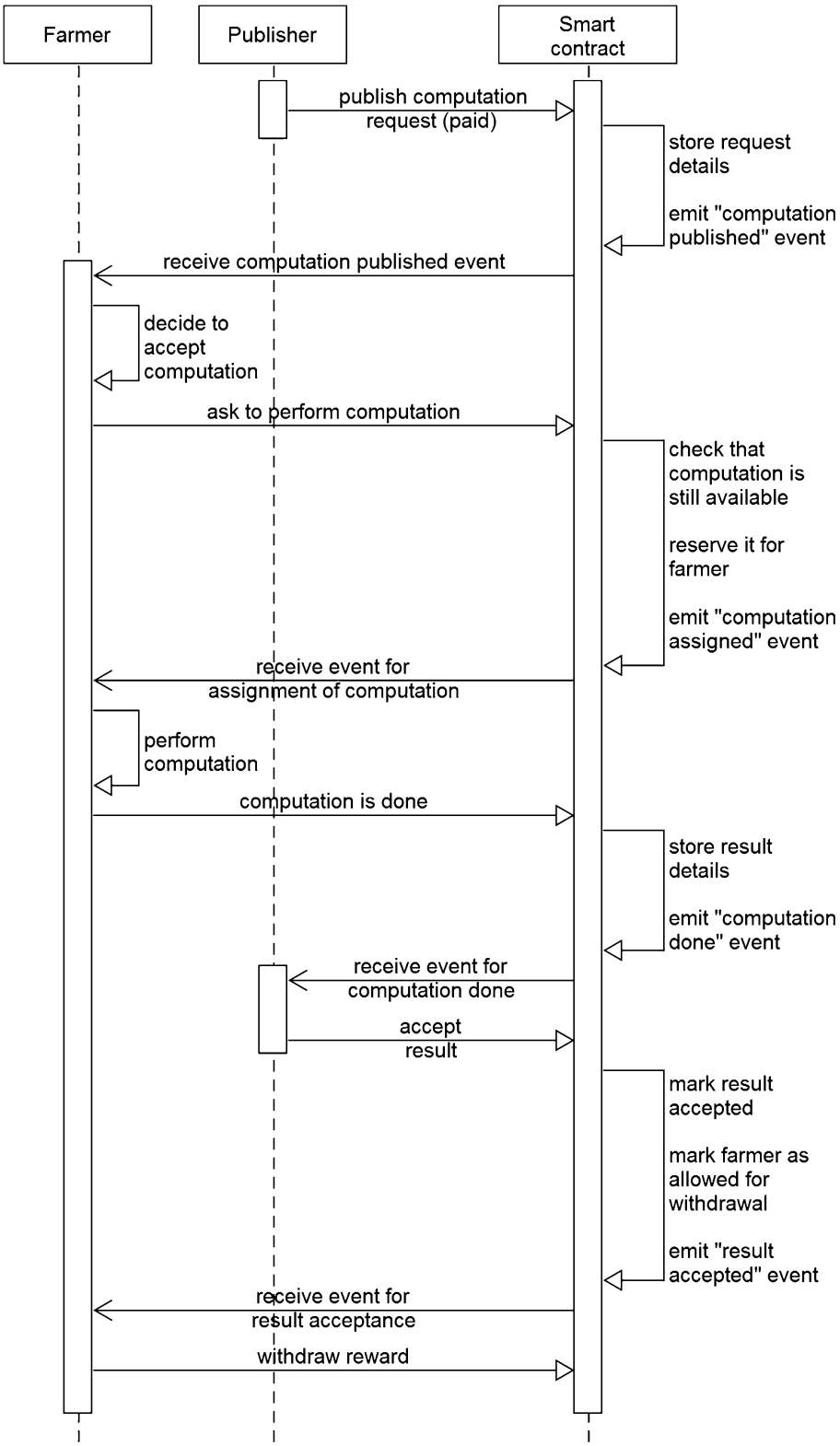

Fig. 1 Sequence diagram for the standard scenario. This figure shows the standard protocol if all parties follow the correct steps of the procedure

assigns it to the farmer and issues a `ComputationAssigned` event. Otherwise, the request by the farmer is simply rejected and they will not receive a `ComputationAssigned` event (see Figure 2). As soon as a farmer observes a `ComputationAssigned` event for their request, they can start with the computation, as this task has now been reserved for them. Currently, the tasks are not assigned to farmers via a bidding process, but on a first-come first-served

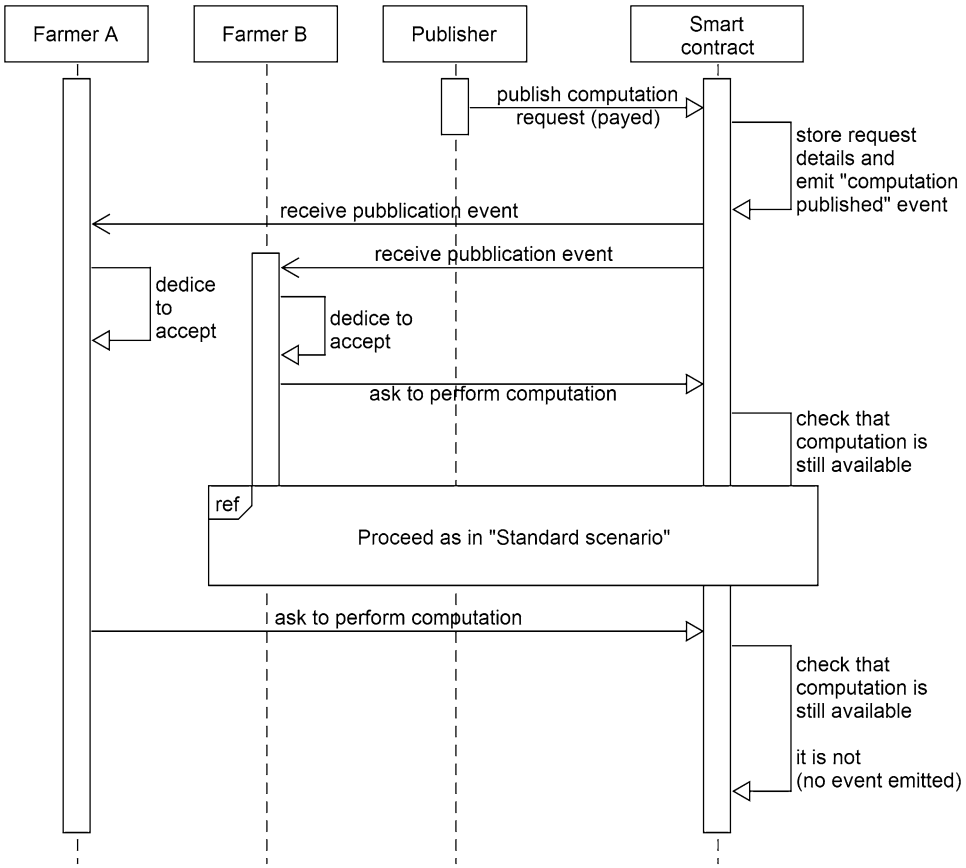

Fig. 2 Sequence diagram for a failed reservation. This figure describes the steps of the protocol that are followed in case a farmer is not able to reserve a task

basis. As soon as a farmer is accepted for a task, other contenders are blocked from it. Nevertheless, our protocol could be extended by a bidding component, which handles the assignment of tasks to farmers.

Once a farmer has finished the work, they inform the smart contract about this. In addition to some meta-data concerning the task, this message also contains a one-way cryptographic hash of the result, which is stored on the blockchain. The purpose of this hash is to commit a farmer to their computed result in case there is a dispute later on.[4] The smart contract also checks that the farmer who submits the result is actually the one who reserved it and, if the answer is positive, emits a `ComputationDone` event. This prompts the publisher to retrieve the result from the farmer (more details on the concrete implementation in Section 5.4). After checking the results, the publisher can now choose to accept or reject them. If the results are accepted, the smart contract marks the task as accepted and issues a `ResultAccepted` event, after which the farmer can withdraw their reward. We cover the case of a rejected result in the following section.

[4] While this hash is not a processing correctness proof, a publisher can check that the uploaded result is indeed the one that the farmer committed to on the blockchain.

Rejected Result

Verifying the correctness of a computation is difficult to do directly on the blockchain via a smart contract, which is the reason we opt for an optimistic fair exchange protocol. We assume, given the right incentives, that the parties will act honestly most of the time, resulting in the execution of the protocol shown in Figure 1. In this case there is no need for a TTP. However, if a publisher suspects a farmer to have not done the computation properly, they can appeal to an auditor. This invokes the part of the protocol depicted in Figure 3.

When a publisher rejects a result, the job of the auditor is to re-execute the published computation, using the same configuration parameters, and apply the already mentioned cryptographic hash function to the result. The hash value is sent to the smart contract, which compares it to the hash value submitted by the farmer. If the two values match, the farmer is acquitted and the smart contract releases the reward, allowing the farmer to withdraw the funds. If the two values do not match, because either the farmer was acting maliciously or their system failed in some way, they face some punitive measures. However, a farmer could upload a hash, but not make the result available, so that a publisher could not inspect the result when deciding to challenge the farmer. In case the result is unavailable, this is treated as a

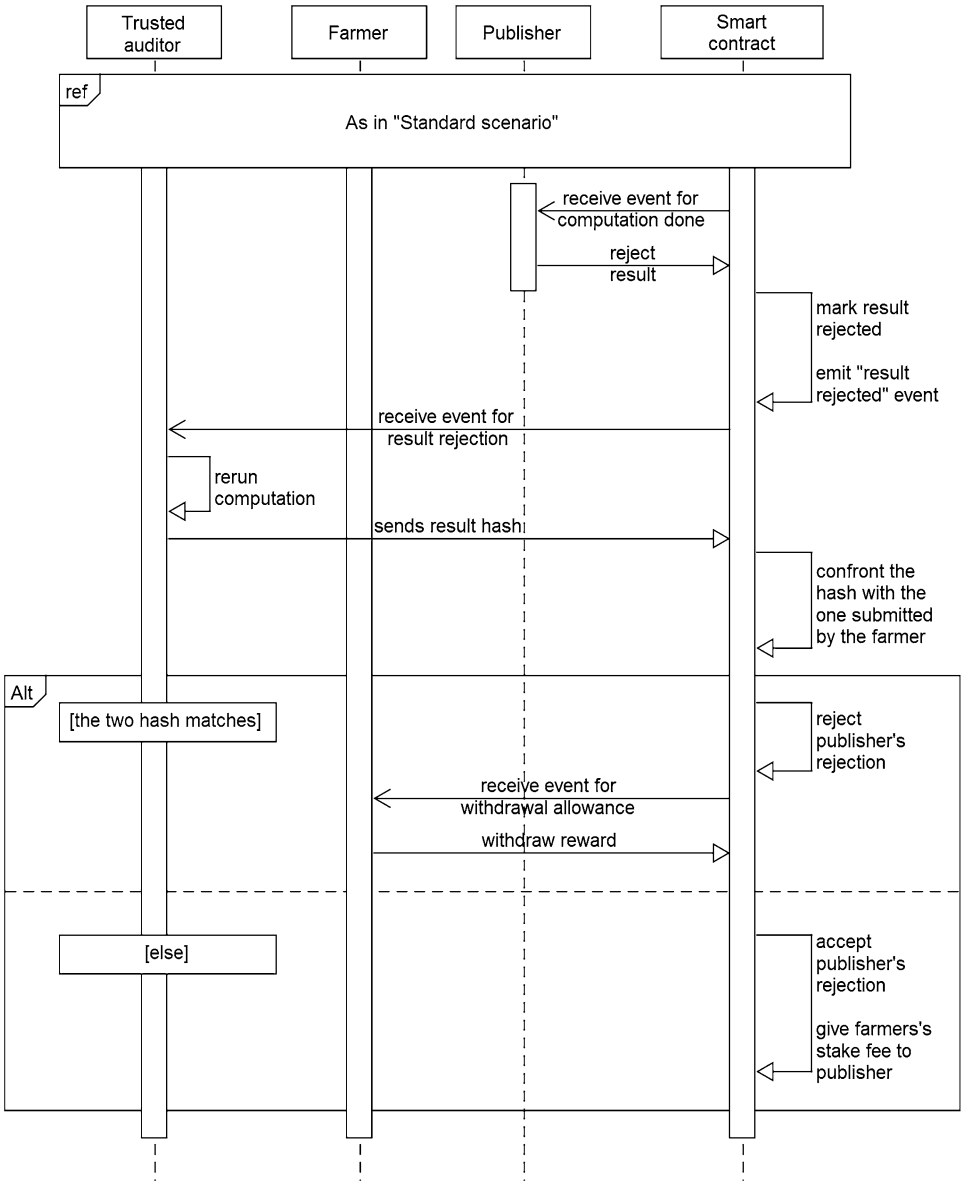

Fig. 3 Result is rejected by publisher. This figure describes the steps of the protocol that are followed when a publisher rejects the result produced by a farmer

failure on the farmer's side to fulfill the contract and is interpreted as a successful challenge by a publisher. The code that is re-executed by the auditor needs to be made available by the publisher. Unavailability of the code is interpreted as an unsuccessful challenge by a publisher, meaning that they paid for the audition and get nothing in return. As storing the complete result and code on-chain is prohibitively expensive, we only store the signed hashes there. The farmer and the publisher are responsible for making the results and code available, respectively. Not doing so will result in a financial loss during a challenge. Storing the result and code off-chain makes the system vulnerable to another type of attack, though. A farmer or a publisher could make the result or code available to the auditor but not to the other involved party, so that for the auditor everything looks fine. However, a farmer not able to access the code would not be able to produce a correct result, while a publisher would not gain access to the result and, when challenging the farmer, would not be compensated for it. Even though there is no financial gain for a farmer or a publisher to selectively deny access to the data or code, they could still maliciously hurt the other side. We will come back to this issue in Section 5.2.4.

Next we propose further incentives to encourage the involved parties to interact in a trustworthy manner. On the side of the farmer, this incentive takes the form of a deposit called a *stake fee*. When publishing a computational task, a publisher decides on the amount of this fee and when a farmer submits the results of the computation, they have to pay the stake fee. This payment is temporarily kept on hold by the smart contract, similar to an escrow. If the publisher accepts the result of a computation, the farmer is allowed to withdraw both, the reward and the stake fee. If the publisher

rejects the results, the deposit will not be released until the auditor has made a decision. If the auditor comes to the conclusion that the farmer has worked correctly, the reward and the stake fee are unlocked. If the auditor concludes that result is incorrect, then the farmer forfeits their stake fee and is also not allowed to collect the reward, both of which are transferred to the publisher.

The publisher and the auditor also need to be incentivized. We introduce an *audit fee*, which has two objectives. On the one hand, it serves as payment for the auditor, as we cannot expect and auditing service to be free. The audit fee is paid independently of the decision made by the auditor, which makes them impartial to the outcome. On the other hand, it keeps publishers from challenging every computation done by farmers. If the audits were free, we would expect almost every publisher to go for one to recheck the results of a computation and get a confirmation for their correctness. This would add too much overhead and defeat the purpose of the auditing mechanism. The minimum amount of the audit fee needs to be fixed in a way to guarantee that the auditor has enough funds to re-execute the computation and still make a profit. However, this amount can be topped up by a publisher to indicate a high priority and offer an incentive for faster processing on the side of the auditor. Naturally, a publisher will want to choose a stake fee that is higher than the usual audit fee.

The combination of stake and audit fees create incentives in the form of financial rewards and penalties to stimulate honest behavior. There is one open question on the publisher's side, though: when should a publisher request an audit? Clearly, if a first glance at the result revealed inconsistencies and discrepancies, a publisher would be well advised to go for an audit. However, we expect that not all cases will be straightforward to evaluate. In the related work section we discussed spot-checking as a technique that is too unreliable on its own. Nevertheless, as an auxiliary method it could have its place in our protocol to support a publisher's decision-making process. Integrating small spotter jobs into their computational tasks would allow publishers to quickly check the validity of results. This system does not need to be perfect: it just has to be made difficult enough for a farmer to analyze a publisher's code when trying to find spotter jobs so that it is more cost-effective to execute the task properly. Zhao et al. call these spotter jobs quizzes [61] and also provide a mathematical analysis on how to choose the ratio between actual tasks and spotter jobs. Generating generic spotter jobs efficiently that cannot easily be distinguished from actual tasks is still an open problem. Nevertheless, there are techniques for efficiently generating application-specific spotter jobs, such as ringer schemes for the inversion of one-way functions [62] and for map-reduce workloads [63].

Our protocol can be enhanced further by tweaking the auditing system. A minuscule amount of each reward posted by a publisher could go to a special auditing account. Whenever there are sufficient funds in this account, an auditor would do a random check on a farmer to confirm the validity of their results. Actually, these checks do not have to be completely random, but can be biased towards farmers whose record shows a higher number of irregularities or new farmers who have joined the market recently and do not have a record yet. Farmers with a good track record would be checked much less frequently. As the blockchain stores all the previous transactions and their outcomes, it is not difficult to track these statistics. In essence, we would be integrating a reputation-based method into our protocol, but only for farmers who have been around for a substantial amount of time and have worked reliable during this time. Similar to spot-checking, a reputation-based system on its own has its flaws, but used as an auxiliary technique would strengthen our protocol.

Time-Outs

There are a few cases regarding the timeliness that we have not covered yet. It could happen that a farmer accepts a job, but then vanishes and never delivers any results. Consequently, we need a mechanism ensuring that a publisher does not wait indefinitely for the farmer's return, in the meantime blocking the reward posted by the publisher. We have two mechanisms in place to deal with this case. Another farmer who is willing to perform the task can challenge the current farmer to whom the task is allocated. If the current farmer does not react within a certain timeframe, the task gets assigned to the challenging farmer. The individual steps of the protocol covering this case are shown in Figure 4. There is another time-out that releases the task even if no other farmer challenged the current farmer.

A more elaborate ploy is a farmer challenging themselves with a new identity effectively preventing the completion of a task. A farmer can continue this indefinitely by creating new identities and challenging previous unresponsive identities, leading to a denial-of-service attack. In order to prevent this scenario, we put the following mechanism in place. Each participant entering the marketplace has to make an initial deposit. When leaving the marketplace, an entity gets the deposit back. In case a participant remains unresponsive (and is challenged successfully on this unresponsiveness), they lose a part of this initial deposit. The money that is lost goes into the special auditor account mentioned in Section 5.2.2 and can be used to do some additional random checks on results or to lower the fee that has to be paid by a publisher to do an audit (basically subsidizing the audit process).

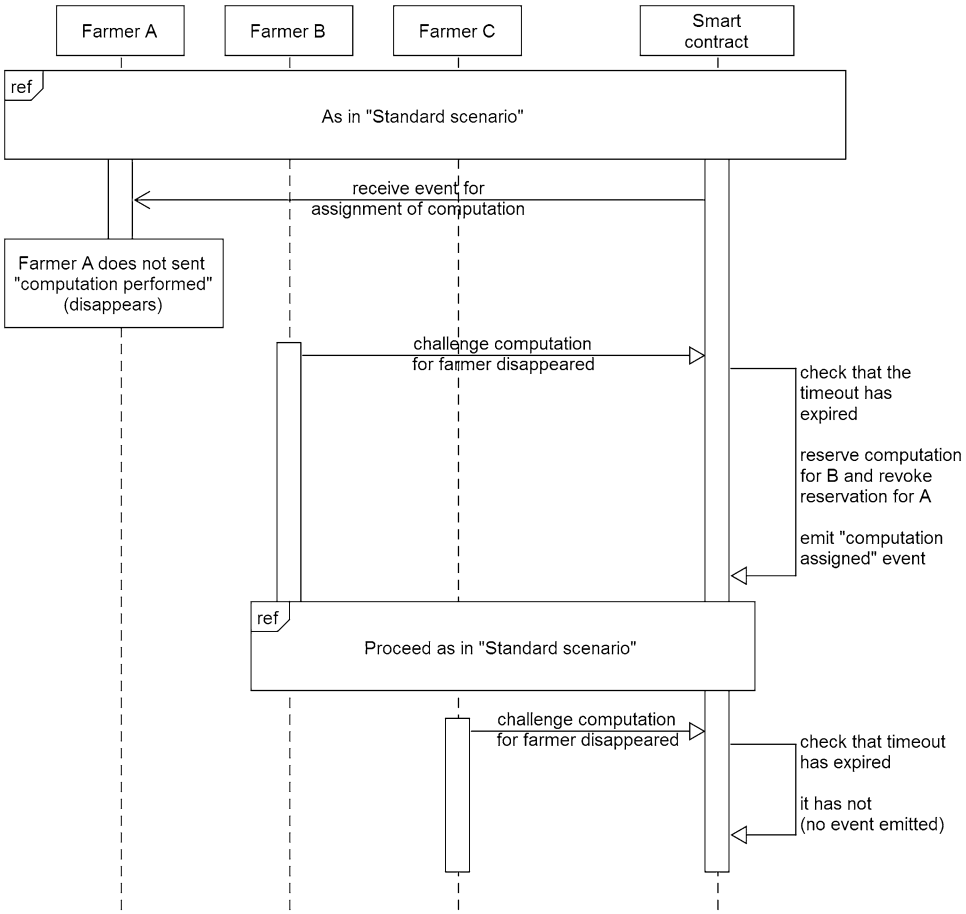

Fig. 4 Farmer does not deliver results. This figure describes the steps of the protocol that are followed when a farmer does not deliver the result to a publisher

An entity can only keep participating in the marketplace as long as the initial deposit remains above a certain threshold. If it falls below this threshold, it needs to be topped up again. This makes it expensive to run a denial-of-service attack by hopping from identity to identity. Clearly, if an attacker is willing to spend a significant amount of resources, we cannot prevent this kind of attack completely. However, this is a general difficulty in preventing denial-of-service attacks and we are not able to provide a solution for this fundamental problem here. We could make this situation even more expensive for an attacker by not reimbursing the deposit immediately when a participant leaves the marketplace. This delay will freeze assets for longer periods of time.

Additionally, we can also introduce a mechanism that allows a publisher to withdraw a task if no farmer is willing to accept it after a certain period of time. This would allow a publisher to get back the posted reward.

On the other side, we could have a non-responsive publisher. Once a farmer has completed a job, they wait for the acceptance by the publisher (or a potential auditing phase). However, if the publisher does not react in any way, the farmer would have to wait for their reward indefinitely. Here we also introduce a time-out, after which a farmer can challenge the publisher to obtain the reward and retrieve their stake fee. The sequence diagram for this procedure is depicted in Figure 5.

A publisher could also have a negative impact on the responsiveness of the marketplace by flooding it with lots of small jobs and then not responding to the completion of a task. However, we see less of a problem here, as a publisher will still have to pay the reward for the completion of the task. If this behavior by publishers is an issue, we can punish them further by deducting a certain amount from their initial deposit in case they do not respond.

Selectively Denying Access to Data

The issue of a farmer or publisher selectively denying access to off-chain data is part of a larger challenge faced by blockchains: how to track off-chain events reliably and map them correctly to the chain. In our case, storing the hash value of the code or the result on-chain allows us to verify that off-chain data is correct, but it does not guarantee that the data is actually transferred or made available to all parties. In the words of the think tank Freedom Lab, “the interface between the blockchain and the real world is of crucial importance

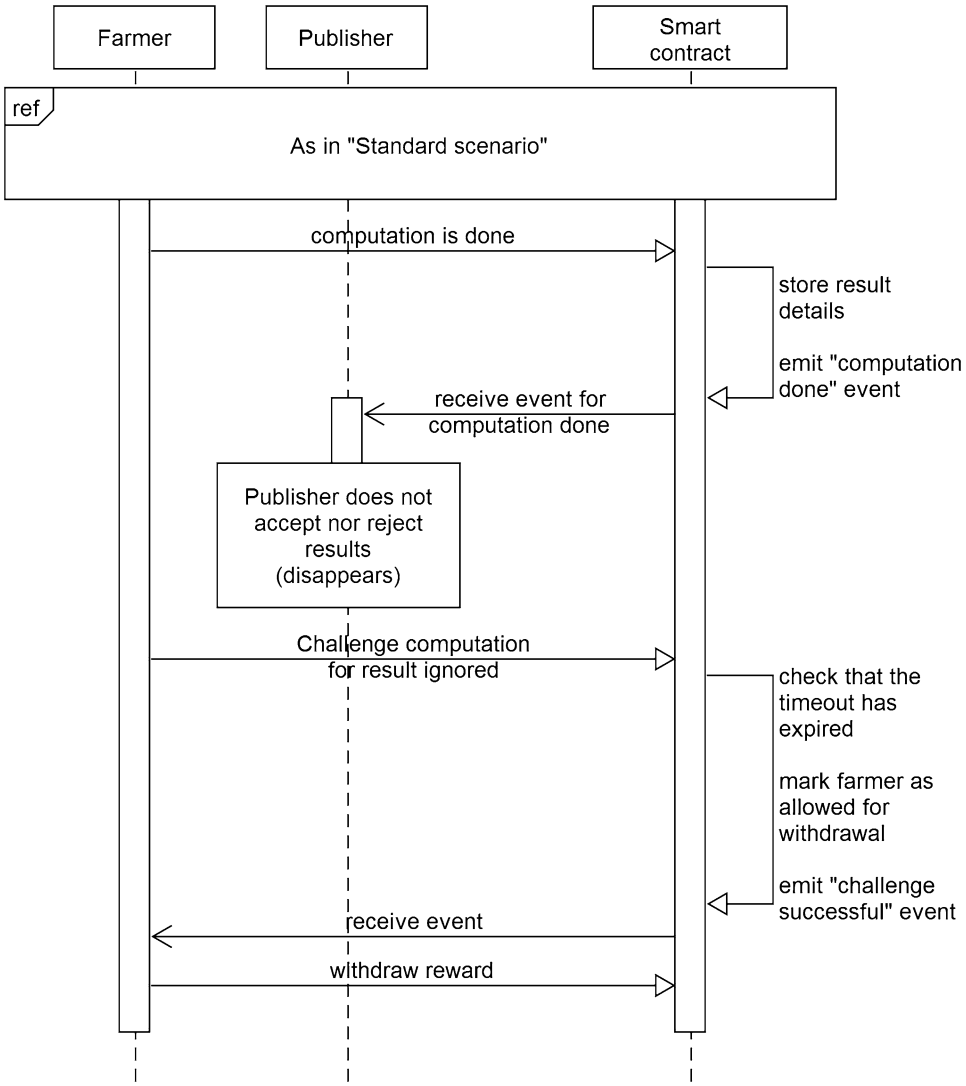

Fig. 5 Publisher neither accepts nor rejects results. This figure describes the steps of the protocol that are followed when a publisher disappears, i.e., they do not accept the results but also do not reject them

and it is no wonder that many initiatives seek to develop reliable and scalable solutions to this" [64].

One of these solutions is a trustworthy oracle [64, 65], which verifies real-world events and feeds this information to a smart contract. However, this is easier said than done. Current proposals for oracles either need a third party [66] or, in the case of decentralized methods, rely on reputation-based approaches [67]. Once trustworthy oracles become available, they could be used to solve our issue of selectively denying access to data by verifying that the data was transferred or made available correctly. However, as it is not clear when mature systems will be available, we suggest another solution.

The Ethereum blockchain is only part of a whole ecosystem. Swarm and Whisper are two other components of this system responsible for off-chain data storage and messaging [55]. Swarm is of particular interest to us, as it offers off-chain peer-to-peer storage that is built to resist denial-of-service attacks. At the time of writing, Swarm was not fully implemented yet. Nevertheless, it is in a proof-of-concept phase (release 3) [68] and can be seamlessly integrated into an Ethereum environment once it comes online. For our framework this would mean that the transfer of off-chain data between farmer and publisher could be done reliably via Swarm. Compared to trustworthy oracles, we believe that this is the more promising approach.

Further Thoughts

Even though we try to reduce the involvement of a TTP in our protocol, it might still become a bottleneck, especially if the number of users grows. The more we can automate the auditing process, the better our approach will scale. For instance, we could delegate the recomputation of a task due to a dispute by hosting one or more auditor services (with

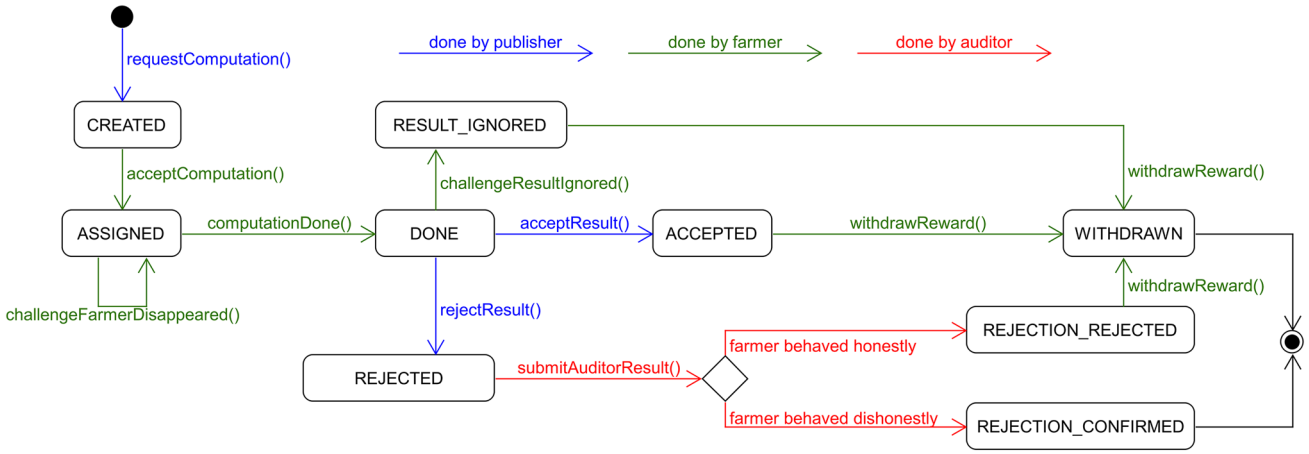

Fig. 6 State-chart diagram for transitions of a computation object including corresponding actors. This figure gives an overview of the states a computing object can transition through while being processed by our framework. It shows the states of the object as well as the different actors

their own Docker IDs[5]) on one of the large, trusted players in cloud computing. In the ideal case, the auditor would be implemented in the form of a smart contract overseeing the dispute resolution. However, in this case we may want to add a step to the auditing process if any of the involved parties want to challenge this purely algorithmic decision. We discuss this and other legal matters in further detail in Section 7.3.

Architecture

Our system is divided into two main components: a smart contract residing on the blockchain and a client application that interacts with it. The smart contract stores a list of `Computation` objects on the chain, each object containing all the details of that computation. The public interface of the contract is composed of a list of methods that implement the protocol described in the previous section. Each of these methods receives input from the client application, performs checks related to the status of the computation, and determines if the request is valid. If a request is valid, the requested modifications are applied to the stored computation object and the required events are emitted. Otherwise, an error is thrown. For each new published computation, the contract generates an ID that will be used to reference it. We used the Truffle framework[6] as a basis for the contracts to allow easier unit-testing and deployment.

The client interacts with the smart contract and is a web application divided into three components:

1 A Geth client connected to the blockchain (either the main one or one for testing purposes).
2 An Express server, interacting with the Geth client using JSON-RCP over WebSockets, that serves the frontend to the user. The server also listens for events on the blockchain and takes the required actions, interacting with the Docker daemon if needed. For example, if the server receives a `ComputationAssigned` event, which assigns a computation to the current farmer, it will download the Docker image associated with the computation, start the associated container, collect the result, and automatically send the hash of the result to the smart contract.
3 A frontend written with standard web technologies (HTML, CSS, Javascript) that presents the information to the user and relays user actions to the backend Express server.

Implementation Details

Smart contract

The smart contract is written in the Solidity programming language (the standard language for the Ethereum blockchain) and is split into two subcontracts that are combined to a single one before the deployment. The `Administrable` contract maintains information related to the owner of the contract and the auditors; the `Main` contract stores the metadata of all the computations and contains the methods needed for implementing the protocol described in 5.2.

We kept the `Administrable` contract simple in our prototype implementation. On contract deployment, the address of the deployer is stored as the owner of the contract. This account has full control over the contract, can set

[5] A Docker ID is a user name space for hosted Docker services and can be requested on the official Docker web page.

[6] https://trufflframework.com/

configuration parameters and is the only trusted auditor. A more sophisticated implementation may allow for multiple auditors and/or owners or require a voting system in order to change parameters.

The `Main` contract maintains a mapping that associates every computation ID to an object containing all the information related to that computation. The computation object itself follows the state machine described in Figure 6. This diagram also shows the names of the methods used by the smart contract to transition between different states. Every method can only be invoked by a specific entity (publisher, farmer, or auditor) and these constraints are enforced in the contract. A computation object is structured as shown in Listing 1. Each field of the object has a specific function, described below:

- `status`: maintains the current state of a computation in the state machine (see Figure 6)
- `publisher, dockerImageName, weiReward`: the address of the account that published the computation request, the full name of the Docker image describing the requested computation, and the amount of Ether (in Wei) that will be given to the farmer for performing the computation.
- `assignedTo, assignmentTimestamp`: the address of the farmer to whom the computation is assigned and the time when the computation was assigned. The timestamp is needed to check if a challenge from another farmer is valid or not.
- `stakeFee, auditFee`: used to store the incentives described above. All the amounts are in Wei.
- `resultHash, resultLink`: the hash of the result computed by the farmer and the link from which the publisher can download the full results.
- `resultSubmissionTimestamp`: used to check how long a farmer has to wait before they can challenge a non-responsive publisher to claim their reward.

Listing 1 Structure that describes the computation object

```
struct Computation{
  Status status;

  address publisher;
  string dockerImageName;
  uint weiReward;

  address assignedTo;
  uint assignmentTimestamp;

  uint stakeFee;
  bytes32 resultHash;
  string resultLink;
  uint resultSubmissionTimestamp;

  uint auditFee;
}
```

All the methods implemented in the smart contract roughly follow the same basic structure. In Listing 2 we show the method `acceptComputation` as an example. The first parameter of a method is usually the ID of the computation object. The first part of a method checks for the existence of an object with this ID and, when found, checks whether the object is in the correct state required for an operation (cf. Figure 6). Some methods require additional checks: for example, a farmer challenging the assignment of a job has to satisfy the time constraints. These checks are followed by the actual modification of a computation object. The modifications themselves depend on the specific operation. The changes usually affect the state of the computation object and are made persistent on the chain. After all modifications have been made, a corresponding event is emitted (if appropriate). The events are also stored on the chain, ready to be picked up by users listening for them.

Listing 2 Farmer asking for an assignment of a computation

```
function acceptComputation(uint id) public {
  Computation storage c = computations[id];
  require(c.publisher != address(0),
          "Computation does not exists");
  require(c.status == Status.CREATED,
          "Status not correct");

  c.assignedTo = msg.sender;
  c.assignationTimestamp = block.timestamp;
  c.status = Status.ASSIGNED;

  emit ComputationAssigned(msg.sender, id);
}
```

Client application

The Express server comprises modules that interact via a shared event bus. Most of the events on the bus are generated by a set of listeners registering specific events issued by the smart contract. Other modules listen for the events generated on the bus and, when triggered, initiate certain actions. In particular, we have the following modules:

1 The `WsEventQueue` module dispatches a subset of the events generated on the bus to the frontend, so that the information shown to the user can be updated.
2 The `WorkerManager` module (used by farmers) listens for `ComputationAssigned` events. If a computation is assigned to the current farmer, this module downloads the Docker container of the computation and starts it. Additionally, it monitors running containers and, when one of them finishes, reports this to the event bus via a `job-finished` event.
3 The `UploadManager` listens for `job-finished` events, collects the results, uploads them according to the specifications defined below, and submits the result hash to the main contract.
4 The `WithdrawalManager` listens for events that allow a farmer to collect their reward (either a `ResultAccepted` event from a publisher or the decision of an auditor). When one of these events is received, it creates the corresponding withdrawal request.

Docker

One challenge we faced was representing the computations and their environments in a standardized and portable way. The naive idea of simply distributing the binaries of the applications has serious compatibility issues: we have to be able to guarantee that the computation will actually be executable on the system of a farmer. Our solution to this problem is to publish the computation in the form of a container. Containers are light-weight virtual machines holding an application including the environment needed to run it. Using containers offers the following benefits.

First of all, containers are portable: every system for which the specific container engine is implemented can run that container. Additionally, they are very flexible. The content of the container can be configured freely according to the needs of the creator, i.e., a publisher in our case. On top of that, a single host can run multiple containers, allowing farmers to perform multiple computations simultaneously. Containers are also volatile, meaning that when the container is shut down, its current state is not persisted. Currently, we assume that input parameters are encoded in the docker containers. Although this implies that new container images need to be created for each set of inputs, these images can be built quite quickly and executing a container with a deterministic application twice will yield exactly the same results. This is important for the auditing process in our protocol. Another point is that a farmer can put a limit on the resource consumption of a container, so that it cannot completely exhaust the resources of a system. In principle, containers are also isolated from the host system running them, meaning farmers can execute computations without compromising their system. We will discuss this issue in more details in Section 7.1. Last but not least, containers are easy to create, use, and share.

As a concrete implementation of a container framework we chose Docker [70]. It was first released in 2013 and in the meantime has reached a large user base, as it is open-source, freely available for different platforms, and used by many software vendors to run their system. According to Docker, over 3.5 million applications have been implemented using this technology and over 37 billion application containers have been downloaded [71]. We have successfully applied the Docker technology in an IoT context before [72, 73]

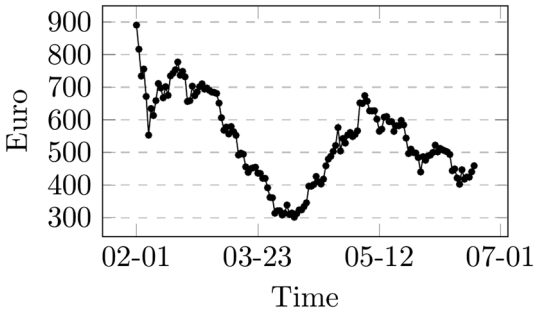

Fig. 7 Exchange rate: value of one Ether in Euro. This figure shows the exchange rate between Ether and Euro for the time between 1 February 2018 and 1 July 2018

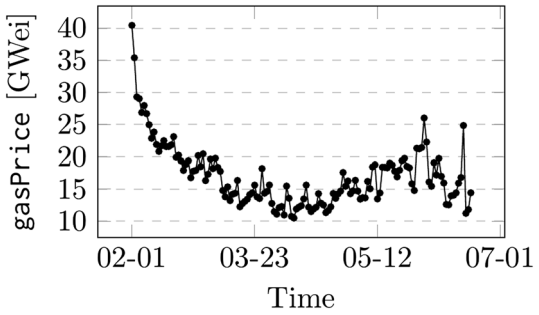

Fig. 8 The average `gasPrice` on Ethereum's main network. This figure shows the average gas price on Ethereum's main network for the time between 1 February 2018 and 1 July 2018

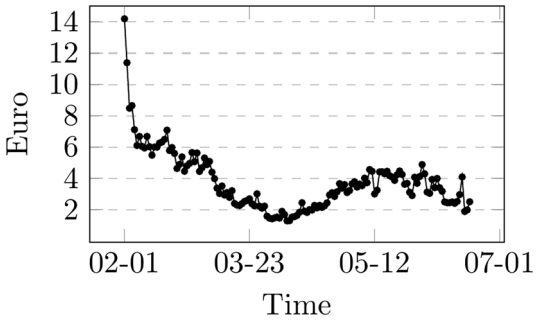

Fig. 9 Total cost for executing standard scenario. This figure shows the total cost for executing the standard part of the protocol for the time between 1 February 2018 and 1 July 2018

and we were also motivated by Naik, who proposes to use Docker as a platform for data processing in the cloud [74].

Further details

When a computation is published, the complete name of the Docker image containing the code for this computation needs to be provided. In principle, there are two different ways to refer to a specific version of an image: either via the image name and a tag or the image name and a digest (usually in the form of a `SHA-256` hash). We prefer the variant using a digest, since tags are mutable, whereas digests are not. This has an impact on the auditing process. A farmer should only execute Docker images providing a digest. In this way the farmer can check that the downloaded image actually belongs to this digest and when an auditor re-executes a computation, exactly the same image will be used. Otherwise, when using tags, a publisher could rebind the tag to another version of the image that produces a different result. If they then reject the result, the auditor will recompute the result using a different image, causing the farmer to be blamed.

Currently, the delivery of the result is done in the following way. The computation stores the results in the `/result` folder inside the container, which the farmer mounts on a host folder: in this way the results are retrievable and uploadable. For submitting the results, the farmer compresses the folder using gzip and computes the hash of this archive file; the hash value is sent to the smart contract. The auditor performs the same steps when resolving a dispute. The farmer also provides a link from which the compressed result file can be downloaded. In our prototype the link is accessible via a simple HTTP GET request and no additional authentication is required (we think of improving this in a future version).

Evaluation

We now discuss in more details advantages and disadvantages of our framework by looking at important aspects. In a first part, we look at an evaluation from a technical point of view, i.e., we investigate the costs for running our framework. In a second part, we consider more general aspects, such as security, fairness, and legal implications.

Cost evaluation

An important aspect of running computations on the Ethereum blockchain is the financial cost of doing so. When executing a transaction, every call of one of the methods of a smart contract that alters the state of the contract has a cost associated with it. We have to distinguish three different components here. The first component is a measure for the cost, while the other two components translate this cost into a real-world currency. First of all, there is *gas*, which measures the amount of computational work that is needed to complete a task. Every instruction executed on the Ethereum Virtual Machine comes with a certain gas cost attached to it. Second, when initiating a transaction, a user has to provide

Table 1 Costs for executing smart contract methods

Method	Consumed gas (in units)	Ether cost (in GWei)	Cost (in €)
requestComputation	138757	2360409.65	1.27
acceptComputation	69544	1183020.16	0.64
computationDone	113299	1927340.98	1.04
acceptResult	29141	495720.56	0.27
withdrawReward	43498	739948.97	0.40
rejectResult	49312	838851.52	0.45
submitAuditorResult	38551	655795.04	0.35
challengeFarmerDisappeared	34816	592258.57	0.32
challengeResultIgnored	29503	501878.58	0.27

Table 2 Costs for executing standard scenario

Party	Consumed gas (in units)	Ether cost (in GWei)	Cost (in €)
Publisher	167898	2856130.20	1.53
Farmer	226341	3850310.11	2.07
Total	394239	6706440.31	3.60

a `gasPrice`, which is not a cost in itself but states how much the user is willing to pay per unit of gas that is consumed. This price is measured in Ether or Wei: one Ether is 10^{18} Wei (or 10^9 GWei). If there is a lot of contention, those transactions with a higher `gasPrice` are prioritized by miners, which means that their results make it to the blockchain faster. Finally, there is the exchange rate between Ether and the utilized non-crypto currency of reference (Euro in our case), which determines the real-world cost of running a transaction. Due to the volatility of the exchange rate, the cost of using a service can vary considerably from day to day. However, a user has some influence over controlling this volatility. Instead of buying Ether at the point in time when they want to execute a computation, users can buy Ether at an earlier time, e.g. when the exchange rate is favorable.

We conducted experiments on our smart contract implementation, running both the standard and alternative scenarios.[7] These tests have been run on the Rinkeby test blockchain and can be reproduced by calling the `/api/estimate` REST endpoint of the client application's backend. This endpoint performs a list of transactions on the smart contract and returns the amount of gas consumed by each single transaction under different scenarios. Multiplying this amount of gas with a `gasPrice` and the Ether-to-Euro exchange rate yields the actual financial costs.

Table 3 Costs for scenario rejecting the result

Party	Consumed gas (in units)	Ether cost (in GWei)	Cost (in €)
Publisher	188069	3199261.17	1.72
Farmer	182843	3110361.14	1.67
Auditor	38551	655795.04	0.35
Total	409463	6965417.35	3.74

Table 4 Costs for deploying the smart contract

Contract	Consumed gas (in units)	Ether cost (in GWei)	Cost (in €)
Migrations	319470	5434537.14	2.92
Main	3158630	53731780.87	28.87
Total	3478100	59166318.01	31.79

In table 1 we report the costs for running the various methods of our smart contract. The following tables show the total amount of gas, Ether, and Euro spent (for some tables this is also split into the amount spent by different parties). While the units of consumed gas are fixed, the other two costs depend on the `gasPrice` and the Ether-to-Euro exchange rate. At the time of conducting the experiments, one Ether (ETH) was valued at € 537.257818083 and the average `gasPrice` came in at 17.011103191 GWei. (At the beginning of 2019, these costs were much lower: one Ether was valued at around € 150 and the average `gasPrice` was around 8 GWei.) We used the average price, as this will get the transaction processed fairly quickly.[8]

The costs for running the standard scenario without the intervention of an auditor or any other complications are depicted in Table 2. Even though the computations executed by the smart contract are straightforward and concise, the costs are not negligible, clocking in at roughly one-and-a-half Euros for a publisher and two Euros for a farmer.

Table 3 illustrates what happens to the costs when a result is rejected by a publisher and an auditor has to step in. In addition to the resources needed to pay the actions of the auditor and the stake fees, compared to the standard case, the split of the costs between farmer and publisher is different. This is due to calling different functions of the smart contract: the publisher has to call `rejectResult`, which costs more than `acceptResult`, and the farmer does not have to call `withdrawReward`.

[7] Our code is available on https://gitlab.com/shalen/bachelor-thesis

[8] Depending on the urgency of a task, it may be worth checking a site such as https://www.ethgasstation.info/ for current numbers.

The costs reported in the tables above have to be considered as indicative. First, the Ether-to-Euro exchange rate can vary from day to day. Figure 7 shows the conversion rate for the time period between 1 February and 21 June 2018. Depending on the amount of traffic on the blockchain, users also need to adjust their `gasPrice`, as they are competing with the transactions of other users. Figure 8 shows the average `gasPrice` on Ethereum's main network for the same time period. These two parameters (`gasPrice` and Ether-to-Euro exchange rate) determine the actual cost for running a transaction. Figure 9 illustrates what it would have cost us to execute the standard scenario for the given time period.[9] As we can see clearly, these costs vary considerably: from below € 2 up to more than € 14.

There is one more factor to consider, the costs for deploying and updating the smart contract. The deployment scheme we use is the one suggested by the Truffle framework. The first deployment on a chain requires deploying two contracts: the actual smart contract and an additional `Migrations` contract managed by Truffle. The Truffle framework simplifies the deployment and redeployment process by keeping track of contract addresses on the blockchain and automating the overall process. For instance, when redeploying contracts in a multi-contract application, it makes sure that only contracts that have actually changes will get redeployed. In Table 4 we can see the costs for deploying the two contracts on the Rinkeby test network.

From these results, we can see that the development costs for the smart contract are quite high: every update to the application requires deactivating the old contract and deploying a new one, incurring the associated costs. Proper testing and extensive review before every deployment are therefore necessary in order to avoid any unnecessary deployment costs.

One of the most important questions left to answer is whether the costs of using our platform are competitive with the costs of deploying the computational tasks on a public cloud. For a start, the pricing models used by cloud providers are completely different to our approach. Cloud providers rent out resources for a specific time period, while in our model a user pays per task. So, trying to come up with a comprehensive answer is far from trivial and researchers are just starting to analyze and to look into pricing models for the cloud [75, 76]. For instance, prices on the Amazon spot market can vary dramatically, sometimes reaching extremely high levels: Wu et al. report spot prices of $999 [75]. Even though there is a lot of uncertainty around these cost models, we think that our current approach is probably not competitive with deployment on a public cloud. However, we believe that the replacement of proof-of-work mechanisms with proof-of-stake ones could bring down the prices for operating blockchains considerably [77, 78]. In addition to this, there are other incentives besides financial ones: a publisher might want to distribute computations among many different entities or may be interested in not using one of the big players or entities that are located in certain countries.

[9] This is the total cost, i.e., costs of publisher and farmer.

Measuring a Workload

We suggest using the amount of gas consumed by the execution of a smart contract as a metric for measuring the performance of this contract, i.e., the consumed gas column in the tables of the previous section indicate the efficiency of our methods. We did not measure execution times, since it is not a very useful metric to track [69]. The time between the first instruction of a method and its last instruction depends on the specifics of the Ethereum VM implementation that is employed. It is also not clear if this is what we should actually be measuring. The smart contracts are executed on many different nodes (that may run different implementations of the Ethereum VM) and even if a method is successfully executed on a node, it does not mean that the transaction has actually gone through.

Discussion

We now discuss more abstract aspects of our framework, namely security aspects, the notion of fairness guaranteed by us and also address potential legal issues faced when deploying such a system in an international setting (or in certain countries).

Security Aspects

The documentation on the Docker web site provides some information about how the platform is made secure by using techniques such as kernel namespaces and control groups (cgroups) [79]. When starting a container, a set of namespaces is created for this container to isolate it from other containers (and the host system). This ranges from network stack and mount point management all the way to process isolation. With the help of cgroups the resource consumption of containers is managed to prevent denial-of-service-like attacks. The capabilities of containers started by Docker are already limited, as for most tasks special privileges are not needed. It is recommended in [79] to restrict this further by removing all unneeded capabilities from a container configuration.

While a lot has been done to make containers more secure, there are deeper underlying problems caused by the general architecture of container-based platforms. The main motivation for developing container-based platforms, such

as Docker, was not to create completely isolated environments, on the contrary, this was about sharing resources of the host system, e.g. the kernel [80]. Consequently, early versions of Docker had severe shortcomings when it came to security, e.g. mapping the privileged user in a container to the privileged user of the host system, which means if malicious software managed to break out of a container, it could subvert the host system.[10]

Container-based platforms are much more lightweight than virtual machines, such as Xen [81]. By sharing the kernel of the host system, containers offer fast instantiation times and low memory consumption. However, this comes at a price: containers are considered less secure than virtual machines [82]. In a virtual machine (VM), the guest software stack (including the guest kernel) runs on virtual hardware emulated by the VM. The hypervisor, which runs the VM, provides an (x86) application binary interface (ABI). A container, on the other hand, interacts with the host system via the kernel system call interface. There is a huge difference in the width of these interfaces: there are over 300 different system calls in Linux in contrast to the about twenty different hypercalls in the Xen hypervisor interface [83]. This makes it much easier to monitor and control the hypercalls in a VM compared to the systems calls in a container-based platform.

The system call interface can be hardened with the help of kernel mechanisms such as secure computing (seccomp) [84, 85]. Using seccomp, a set of fine-grained rules can be formulated to define which system calls a process is allowed to make. If an unexpected call is encountered, it is blocked (usually leading to the termination of the process). In practice, it is difficult to fine-tune an appropriate policy [83]: if it is too restrictive, this will result in a significant number of terminated processes. Additionally, policies tend to get large and complicated quite quickly. Consequently, most policies are too lenient, e.g. the default policy for Docker containers allows more than 250 system calls [86]. On top of this, we would have to formulate and fine-tune separate policies for different application domains. So far, this has been done for biomedical applications [87, 88], but in our case this might result in a large overhead, as we do not restrict ourselves to specific application domains.

One solution to make containers more secure is to run them inside of a VM to gain the benefits of the much more secure hypervisor interface.[11] This still leaves us with potential breaches between containers (unless we run each container in its own VM environment), but this is a minor concern for us, as we assume that farmers do not have a lot of excess computational power, so most of them would be running a single-tenancy configuration. However, running a container inside of a VM adds a lot of overhead, which has a considerable impact on the profitability, maybe even making our scheme unsustainable. A farmer could separate the container platform physically from the rest of their system, e.g. by partitioning it into a dual-boot system. One of the partitions would then exclusively run a bare-metal container platform. While this is definitely a secure solution, it would render the other partition unusable while running jobs for publishers.

Although currently there does not seem to be a definitive answer addressing all the security concerns, there is promising work on lightweight VMs. Instead of running a whole operating system stack inside of a VM, an application is linked only against the parts that are needed, creating a lightweight unikernel (originally, this was done with OCaml-based applications and MirageOS [90]). Williams et al. have taken this a step further by running unikernels as processes on a host [83]. At first glance this seems to be a step backwards, as it exchanges the more secure hypervisor interface for the less secure system call interface. However, Williams et al. have shown in their prototype system Nabla that they only require nine different systems calls, which are much easier to manage via seccomp policies compared to hundreds of different system calls for Docker. The downside is that unikernels are not as easy to manage as Docker containers [80]. Once this technology becomes more mature and easy-to-use Nabla-containerized options become available, this could become an interesting component of our framework.

In summary, currently VMs are the more secure option, but they add too much overhead, while containerized approaches are more lightweight but less secure. Choosing the right platform for our framework is still an open question. Nevertheless, once a more secure containerized option becomes available, we can integrate into our marketplace.

Fairness

In [60], which is based on earlier work by Asokan [91], Liu et al. define requirements for the fairness of exchange protocols. This definition consists of four criteria: effectiveness, fairness, timeliness, and non-repudiation. Asokan states that, strictly speaking, non-repudiation is not an integral part of such a protocol, but that it helps in resolving conflicts. In the following we show that our protocol satisfies the criteria defined by [60] and [91].

Effectiveness means that two participants acting honestly will lead to a successful completion of the protocol. Under this definition, our protocol is effective: in this case we execute the standard scenario at the end of which the publisher receives the results of the computation and the farmer receives the publisher's payment.

[10] Since version 1.10, containers can be run as non-privileged users.

[11] This is how Google actually runs customers' containers[89].

Asoka distinguishes two different levels of fairness, strong fairness and weak fairness. A protocol using the notion of strong fairness results in one of two outcomes: either both participants have received what they wanted or neither of them has. If a protocol utilizes the notion of weak fairness, an honest party can prove to an arbiter that they fulfilled their side of the bargain. Our protocol follows the notion of weak fairness. When a fraudulent publisher claims that the result of a computation is not correct (to avoid paying a farmer), they have already received the results of the computation. However, a farmer can prove to a trusted third party that they have delivered the services and will get compensated for this (additionally, the publisher will be punished). In the case of a cheating farmer, they will not receive their payment, since the publisher has not received the agreed-upon services. This second scenario actually follows the notion of strong fairness, as neither side gets what they want.

Timeliness means that the protocol will eventually terminate at a certain point in time, either successfully or in a failed state. Our protocol satisfies this criterion as well, because each individual step of the protocol will time out after a certain period of time.

Finally, with non-repudiation a participant is able to prove the origin of the exchanged goods or services. In our protocol this is crucial for the container provided by a publisher and the results returned by a farmer. Both of these components are hashed and signed (by the publisher and farmer, respectively) and the signed hash is stored on the blockchain. While this is not processing correctness proof, it is there to make sure that a third party can verify which code was was executed and that the received result is the one actually submitted by the farmer. So, in our framework this happens indirectly, the non-repudiation is provided by the blockchain itself.

Legal Implications

Koulu analyzes the legal implications of employing blockchain technology and smart contracts for regulating online transactions [92]. In the following we summarize what this analysis means for our electronic marketplace.

Generally, as Koulu points out, cross-border transactions are on the rise, this is not just the case for our electronic marketplace, e.g. it has become common for consumers to order merchandise online from a vendor in a different country. In many cases these transactions are low-intensity, i.e., they have a low financial volume, which makes it too expensive to resolve a conflict in a court of law, especially if it is located abroad. An interesting alternative is an online dispute resolution (ODR) mechanism. According to Koulu [93], "ODR still lacks a uniform definition," the only common ground of different solutions seems to be the application of technology for a more efficient dispute resolution. Koulo points out in [92] that enforcement is a crucial issue: "Without a way to force compliance with a decision, the decision is mainly without effect. Although voluntary compliance is possible, an effective redress mechanism is needed to force compliance in case the final decision reached in the ODR process is not voluntarily followed." Since going through public institutions, such as courts, is too expensive for low-intensity transactions, implementing direct private enforcement via self-enforcing protocols is a promising approach. This could be integrated into the payment scheme of an electronic marketplace using methods such as escrows, changebacks, or insurance mechanisms [94].

While private enforcement can be realized on a technological level in the form of smart contracts (as we have done for our electronic marketplace), on the legal side this is controversially discussed, as "private enforcement bypasses the nation state's monopoly on violence" [92]. In some jurisdictions the introduction of binding pre-dispute arbitral clauses is not allowed, because it removes the concept of due process and the right to a fair trial from the dispute resolution process. It is not clear what happens if one of the parties is not satisfied with the outcome of a practically irreversible automatic enforcement. Currently, there are no mechanisms for handling follow-up disputes. Essentially, this shifts dispute resolution from state control to private third parties, potentially undermining a state's authority in the long term. A more subtle impact of utilizing smart contracts for ODR is that this blurs the distinction between substantive law (under which contracts fall) and procedural law (the due process mentioned before). While substantive law covers fundamental principles governing society, procedural law is about its practical application by courts and judges. Arnold argues that these two should be kept separate and that changes to a judicial system should not be made by judges who are involved in running the system, but by independent, objective scholars [95].

In summary, we think that the building blocks for putting private self-enforcing protocols already exist, but we agree with Koulu who states in [92] that "the implications of ...the overall automation of complex legal issues are not straightforward." Consequently, legal aspects may play an important role in which mechanisms will be used to implement the protocols in the end.

Conclusion and Future Work

We have developed a framework for implementing a decentralized marketplace for computational power that would allow a wide range of providers to offer cloud computing services. A major stumbling block was assuring the quality of the computational results. While a technique such as

verifiable computing would give us rigorous proofs that a task was executed correctly, the overhead is currently too high to make this a realistic option, so we opted for a different approach. Our solution revolves around an adjudicated protocol where in most cases the two parties interact without any intervention by a third party. Here the blockchain acts as an escrow service, making sure that the payment is actually there to begin with and that it gets released once the computational task has been completed and accepted. If there is a disagreement between the two parties, a third party steps in to resolve the conflict. We have put various incentives in place to encourage the participants to behave honestly.

In addition to developing the protocol, we have implemented a prototype as a proof-of-concept and have also conducted a number of experiments to test the performance. Although we have shown the viability of the approach by successfully running the prototype, a crucial aspect is the financial overhead imposed by the blockchain. Currently, the system has some non-negligible costs for the users, which would probably make the fees higher than those of large cloud providers. Ethereum, as many other blockchains, is not an ideal platform to run our framework on, as its network supports a range of applications, some of which are rather speculative, causing a considerable volatility of the exchange rate between Ether and traditional currencies. On top of this, Ethereum relies on a proof-of-work consensus mechanism[12], which incurs significant computational and in turn financial costs. There are plans to bring down these costs by switching to a proof-of-stake mechanism [77, 78]. Independent of this, it would be very interesting to develop cost models and compare the costs of running our platform versus the costs of deploying on a public cloud. Generally, current research is looking into overcoming scalability issues (and also reducing the costs) by introducing sharding into blockchain protocols [96]. Although latency is not our main issue at the moment, since we assume that the computational tasks will run for several hours or even days, improving the response times would certainly have a positive impact when it comes to users adopting our framework. In summary, we think that our framework is a promising approach that could become viable with a few more improvements in the underlying blockchain technology.

Acknowledgements We thank the anonymous reviewers of an earlier version of the paper for pointing out a flaw in our protocol.

Author Contributions Matteo Nardini implemented the framework in the context of his thesis. Sven Helmer and Nabil El Ioini were working together with him on the conceptual parts as supervisor and co-supervisor, respectively. Claus Pahl helped in writing large parts of the paper and in providing further guidance.

[12] This is also the case for many other blockchains.

Funding No external funding.

Availability of data and materials The source code of the implementation is available in the following repository: https://gitlab.com/shaleen/bachelor-thesis

Competing interests The authors declare that they have no competing interests.

References

1. Stephany A. The Business of Sharing: Making It in the New Sharing Economy. London, United Kingdom: Palgrave Macmillan; 2015.
2. Giana M, Eckhardt FB. The Sharing Economy Isn't About Sharing at All. Harvard Business Review, 2015-01-28, https://hbr.org/2015/01/the-sharing-economy-isnt-about-sharing-at-all. [Online; accessed July 2018] 2015
3. Subramanian H. Decentralized blockchain-based electronic marketplaces. Comm. of the ACM. 2018;61(1):78–84.
4. Weyl EG. A price theory of multi-sided platforms. The American Economic Review. 2009;100(4):1642–72.
5. Anderson DP. BOINC: a system for public-resource computing and storage. In: 5th IEEE/ACM Int. Workshop on Grid Computing, 2004; 4–10
6. Wikipedia: Berkeley Open Infrastructure for Network Computing. https://en.wikipedia.org/wiki/Berkeley_Open_Infrastructure_for_Network _Computing. [Online; accessed July 2018] 2018
7. Wikipedia: Tianhe-2. https://en.wikipedia.org/wiki/Tianhe-2. [Online; accessed August 2018] 2018
8. Golem Network. Online White paper, https://golem.network/doc/Golemwhitepaper.pdf. [Online; accessed May 2019] 2016
9. iExec. Online White paper, https://iex.ec/wp-content/uploads/pdf/iExec-WPv3.0-English.pdf. [Online; accessed May 2019] 2018
10. SONM. Online White paper, https://whitepaper.io/document/326/sonm-whitepaper. [Online; accessed May 2019] 2017
11. Taherizadeh S, Stankovski V, Grobelnik M. A capillary computing architecture for dynamic internet of things: Orchestration of microservices from edge devices to fog and cloud providers. Sensors. 2018;18:2938.
12. Nouman Durrani M, Shamsi JA. Review: Volunteer computing: Requirements, challenges, and solutions. J. Netw. Comput. Appl. 2014;39:369–80.
13. Heien EM, Fujimoto N, Hagihara K. Computing low latency batches with unreliable workers in volunteer computing environments. In: IEEE Int. Symposium on Parallel and Distributed Processing, 2008; 1–8
14. Watanabe K, Fukushi M. Generalized spot-checking for sabotage-tolerance in volunteer computing systems. In: 10th IEEE/ACM Int. Conf. on Cluster, Cloud and Grid Computing, 2010; 655–660

15. Watanabe K, Fukushi M, Kameyama M. Adaptive group-based job scheduling for high performance and reliable volunteer computing. Journal of Information Processing. 2011;19:39–51.
16. Chard K, Bubendorfer K. Co-operative resource allocation: Building an open cloud market using shared infrastructure. IEEE Trans. on Cloud Computing. 2019;7(1):183–95.
17. Avizienis A. Dependable Computing and Fault-Tolerant Systems Vol. 1: The Evolution of Fault-Tolerant Computing. Vienna: Springer; 1987.
18. Holt RM. MOS processor for the F14A CADC. Technical Report No. 71-7266, Garrett AiResearch Corp 1971
19. Briere D, Traverse P. AIRBUS A320/A330/A340 electrical flight controls: A family of fault-tolerant systems. In: 23rd Int. Symposium on Fault-Tolerant Computing (FTCS'93) 1993
20. Sarmenta LFG. Sabotage-tolerance mechanisms for volunteer computing systems. Future Gener. Comput. Syst. 2002; 18(4)
21. Dong C, Wang Y, Aldweesh A, McCorry P, van Moorsel A. Betrayal, distrust, and rationality: Smart counter-collusion contracts for verifiable cloud computing. In: ACM SIGSAC Conference on Computer and Communications Security (CCS'17), 2017; 211–227
22. Arora S, Lund C, Motwani R, Sudan M, Szegedy M. Proof verification and hardness of approximation problems. In: 33rd Annual Symposium on Foundations of Computer Science (FOCS'92), Pittsburgh, Pennsylvania, 1992; 14–23
23. Arora S, Safra S. Probabilistic checking of proofs; A new characterization of NP. In: 33rd Annual Symposium on Foundations of Computer Science (FOCS'92), Pittsburgh, Pennsylvania, 1992; 2–13
24. Arora S, Lund C, Motwani R, Sudan M, Szegedy M. Proof verification and the hardness of approximation problems. J. ACM 1998; 45(3)
25. Arora S, Safra S. Probabilistic checking of proofs: A new characterization of NP. J. ACM. 1998;45(1):70–122.
26. Goldwasser S, Kalai YT, Rothblum GN. Delegating computation: Interactive proofs for muggles. J. ACM. 2015;62(4):27–12764. https://doi.org/10.1145/2699436.
27. Gennaro R, Gentry C, Parno B. Non-interactive verifiable computing: Outsourcing computation to untrusted workers. In: Rabin, T. (ed.) Advances in Cryptology (CRYPTO'10) 2010
28. Walfish M, Blumberg AJ. Verifying computations without reexecuting them. Commun. ACM. 2015;58(2):74–84. https://doi.org/10.1145/2641562.
29. Teutsch J, Reitwießner C. A scalable verification solution for blockchains. CoRR arXiv:1908.04756 2019
30. Luu L, Teutsch J, Kulkarni R, Saxena P. Demystifying incentives in the consensus computer. In: Proc. of the 22nd ACM SIGSAC Conf. on Computer and Communications Security (CCS'15), 2015; 706–719
31. Sadeghi A-R, Schneider T, Winandy M. Token-based cloud computing. In: Trust and Trustworthy Computing, 2010; 17–429
32. Resnick P, Zeckhauser R. In: Baye, M.R. (ed.) Trust among strangers in internet transactions: Empirical analysis of eBay' s reputation system, vol. 11, pp. 127–157. Emerald Group Publishing Limited, Bingley, United Kingdom 2002
33. Post A, Shah V, Mislove A. Bazaar: Strengthening user reputations in online marketplaces. In: 8th USENIX Conf. on Networked Systems Design and Implementation (NSDI'11), 2011; 183–196
34. Soska K, Kwon A, Christin N, Devadas S. Beaver: A decentralized anonymous marketplace with secure reputation. Technical Report 2016/464, IACR Cryptology ePrint Archive 2016
35. Casey W, Kellner A, Memarmoshrefi P, Morales JA, Mishra B. Deception, identity, and security: The game theory of sybil attacks. Commun. ACM. 2018;62(1):85–93.
36. Casey W, Memarmoshrefi P, Kellner A, Morales JA, Mishra B. Identity deception and game deterrence via signaling games. In: Proc. of the 9th EAI Int. Conf. on Bio-inspired Information and Communications Technologies (BICT'15), New York City, New York, 2016; 73–82
37. Falcarin P, Scandariato R, Baldi M, Ofek Y. Integrity checking in remote computation. Technical report, Politecnico di Torino (January 2005)
38. Khan SM, Hamlen KW. Computation certification as a service in the cloud. In: 2013 13th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing, 2013; 434–441 https://doi.org/10.1109/CCGrid.2013.75
39. Premnath SN, Haas ZJ. A practical, secure, and verifiable cloud computing for mobile systems. CoRR. arxiv: 1410.1389 2014
40. Klems M, Eberhardt J, Tai S, Härtlein S, Buchholz S, Tidjani A. Trustless intermediation in blockchain-based decentralized service marketplaces. In: Service-Oriented Computing, 2017; 731–739
41. Jakobsson M. Ripping coins for a fair exchange. In: Advances in Cryptology (EUROCRYPT'95), 1995; 220–230
42. Asokan N, Schunter M, Waidner M. Optimistic protocols for fair exchange. In: 4th ACM Conf. on Computer and Communications Security(CSS'97), 1997; 7–17
43. Bao F, Deng RH, Mao W. Efficient and practical fair exchange protocols with off-line TTP. In: 1998 IEEE Symposium on Security and Privacy, 1998; 77–85
44. Even S, Goldreich O, Lempel A. A randomized protocol for signing contracts. In: Advances in Cryptology, 1983; 205–210
45. Okamoto T, Ohta K. How to simultaneously exchange secrets by general assumptions. In: 2nd ACM Conference on Computer and Communications Security(CSS'94). CCS '94, 1994;184–192
46. Even S, Yacobi Y. Relations among public key signature systems. Technical Report 175, Technion 1990
47. Pagnia H, Gärtner FC. On the impossibility of fair exchange without a trusted third party. Technical Report TUD-BS-1992-02, Darmstadt University of Technology 1999
48. Garbinato B, Rickebusch I. Impossibility results on fair exchange. In: 10th Int. Conf. on Innovative Internet Community Systems (I2CS'10), 2010; 507–518
49. Cachin C, Camenisch J. Optimistic fair secure computation. In: Advances in Cryptology (CRYPTO'00), 2000; 93–111
50. Micali S. Simple and fast optimistic protocols for fair electronic exchange. In: 22nd Annual Symposium on Principles of Distributed Computing (PODC'03), 2003; 12–19
51. Küpçü A, Lysyanskaya A. Usable optimistic fair exchange. Comput. Netw. 2012;56(1):50–63.
52. Narayanan A, Bonneau J, Felten E, Miller A, Goldfeder S. Bitcoin and Cryptocurrency Technologies: A Comprehensive Introduction. Princeton, New Jersey: Princeton University Press; 2016.
53. Sompolinsky Y, Zohar A. Bitcoin's underlying incentives. Commun. ACM. 2018;61(3):46–53.
54. Narayanan A, Clark J. Bitcoin's academic pedigree. Commun. ACM. 2017;60(12):36–45.
55. Dannen C. Introducing Ethereum and Solidity. Berlin, Germany: Springer; 2017.
56. Merkel D. Docker: Lightweight linux containers for consistent development and deployment. Linux J. 2014; 2014(239)
57. Schneier B. Applied Cryptography - Protocols, Algorithms, and Source Code in C. 2nd ed. Hoboken, New Jersey: Wiley; 1996.
58. Jayasinghe D, Markantonakis K, Mayes K. Optimistic fair-exchange with anonymity for bitcoin users. In: 11th Int. Conf. on e-Business Engineering, 2014; 44–51
59. Goldfeder S, Bonneau J, Gennaro R, Narayanan A. Escrow protocols for cryptocurrencies: How to buy physical goods using bitcoin. In: Financial Cryptography and Data Security, 2017; 321–339
60. Liu J, Li W, Karame GO, Asokan N. Toward fairness of cryptocurrency payments. IEEE Security Privacy. 2018;16(3):81–9.

61. Zhao S, Lo V, GauthierDickey C. Result verification and trust-based scheduling in peer-to-peer grids. In: Proc. of the 5th IEEE Int. Conf. on Peer-to-Peer Computing (P2P'05), 2005; 31–38
62. Golle P, Mironov I. Uncheatable distributed computations. In: Naccache D, editor. Topics in Cryptology (CT-RSA'01). San Francisco: CA; 2001. p. 425–40.
63. Bendahmane A, Bennasar H, Essaaidi M. An efficient approach to improve security for mapreduce computation in cloud system. In: Proc. of the Int. Conf. on Learning and Optimization Algorithms: Theory and Applications (LOPAL'18) 2018
64. FreedomLab: Where the blockchain meets the real world. No. 224, Theme 01, Week 48, http://freedomlab.org/where-the-blockchain-meets-the-real-world/. [Online; accessed March 2020] 2019
65. Zheng Z, Xie S, Dai H-N, Chen W, Chen X, Weng J, Imran M. An overview on smart contracts: Challenges, advances and platforms. Future Generation Computer Systems. 2020;105:475–91.
66. Zhang F, Cecchetti E, Croman K, Juels A, Shi E. Town crier: An authenticated data feed for smart contracts. In: Proc. of the 2016 ACM SIGSAC Conf. on Computer and Communications Security (CCS'16), 2016; 270–282
67. Ellis S, Juels A, Nazarov S. Chainlink: a decentralized oracle network. https://chain.link/. [Online; accessed March 2020] 2017
68. Trón V. Announcing Swarm Proof-of-Concept Release 3. Ethereum Blog, https://blog.ethereum.org/2018/06/21/announcing-swarm-proof-of-concept-release-3/. [Online; accessed March 2020] 2018
69. Yang R, Murray T, Rimba P, Parampalli U. Empirically analyzing Ethereum's gas mechanism. CoRR arxiv: abs/1905.00553 2019
70. Docker Inc: Docker - Build, Ship, and Run Any App, Anywhere. https://www.docker.com/ Accessed 2018-06-20
71. Vaughan-Nichols, S.J.: What Is Docker and Why Is It so Darn Popular? https://www.zdnet.com/article/what-is-docker-and-why-is-it-so-darn-popular/ Accessed 2018-10-30
72. von Leon D, Miori L, Sanin J, Ioini NE, Helmer S, Pahl C. A performance exploration of architectural options for a middleware for decentralised lightweight edge cloud architectures. In: 3rd Int. Conf. on Internet of Things, Big Data and Security (IoTBDS'18), Funchal, Madeira, 2018; 73–84
73. Pahl C, Helmer S, Miori L, Sanin J, Lee B. A container-based edge cloud paas architecture based on raspberry pi clusters. In: 4th Int. Conf. on Future Internet of Things and Cloud Workshops (FiCloud'16), Vienna, Austria, 2016; 117–124
74. Naik N. Docker container-based big data processing system in multiple clouds for everyone. In: IEEE Int. Systems Engineering Symposium (ISSE'17), 2017; 1–7
75. Wu C, Buyya R, Ramamohanarao K. Cloud pricing models: Taxonomy, survey, and interdisciplinary challenges. ACM Comput. Surv. 2019; 52(6)
76. Portella G, Rodrigues GN, Nakano E, Melo ACMA. Statistical analysis of amazon ec2 cloud pricing models. Concurrency and Computation: Practice and Experience. 2019;31(18)
77. Bano S, Sonnino A, Al-Bassam M, Azouvi S, McCorry P, Meiklejohn S, Danezis G. Consensus in the age of blockchains. CoRR arxiv: abs/1711.03936 2017
78. CoinDesk: Ethereum's Big Switch: The New Roadmap to Proof-of-Stake. https://www.coindesk.com/ethereums-big-switch-the-new-roadmap-to-proof-of-stake/. [Online; accessed October 2017] 2017
79. Docker: Docker security. Docker Docs, https://docs.docker.com/engine/security/security/. [Online; accessed March 2019] 2019
80. Frazelle J. Research for practice: Security for the modern age. Commun. ACM. 2019;62(1):43–5.
81. Barham P, Dragovic B, Fraser K, Hand S, Harris T, Ho A, Neugebauer R, Pratt I, Warfield A. Xen and the art of virtualization. In: Proc. of the 19th ACM Symposium on Operating Systems Principles SOSP'03, Bolton Landing, New York, 2003; 164–177
82. Manco F, Lupu C, Schmidt F, Mendes J, Kuenzer S, Sati S, Yasukata K, Raiciu C, Huici F. My VM is lighter (and safer) than your container. In: Proc. of the 26th Symposium on Operating Systems Principles (SOSP'17), Shanghai, China, 2017; 218–233
83. Williams D, Koller R, Lucina M, Prakash N. Unikernels as processes. In: Proc. of the ACM Symposium on Cloud Computing (SoCC'18), Carlsbad, California, pp. 2018; 199–211
84. Edge J. A seccomp overview. LWN, https://lwn.net/Articles/656307/. [Online; accessed March 2019] 2015
85. Kerrisk M. Using seccomp to limit the kernel attack surface. In: Linux Plumbers Conference (LPC'15), Seattle, Washington 2015
86. Docker: Seccomp security profiles for Docker. Docker Docs, https://docs.docker.com/engine/security/seccomp/. [Online; accessed March 2019] 2019
87. Witt M, Jansen C, Krefting D, Streit A. Fine-grained supervision and restriction of biomedical applications in linux containers. In: 17th IEEE/ACM Int. Symposium on Cluster, Cloud and Grid Computing (CCGRID'17), 2017; 813–822
88. Witt M, Jansen C, Krefting D, Streit A. Sandboxing of biomedical applications in linux containers based on system call evaluation. Concurrency and Computation: Practice and Experience. 2018;30(12):
89. Babcock C. Google: Docker Does Containers Right. InformationWeek, https://www.informationweek.com/cloud/infrastructure-as-a-service/google-docker-does-containers-right/d/d-id/1319146. [Online; accessed March 2019] 2015
90. Madhavapeddy A, Scott DJ. Unikernels: Rise of the virtual library operating system. Queue. 2013;11(11):30–44.
91. Asokan N. Fairness in electronic commerce. PhD thesis, University of Waterloo, Waterloo, Canada 1998
92. Koulu R. Blockchains and online dispute resolution: Smart contracts as an alternative to enforcement. SCRIPTed - A Journal of Law, Technology & Society. 2016;13(1):40–69.
93. Koulu R. Three quests for the justification in the ODR era: Sovereignty, contract and quality standards. Lex Electronica. 2014;19(1):43–71.
94. Kaufmann-Kohler G, Schultz T. Online Dispute Resolution: Challenges for Contemporary Justice. Alphen aan den Rijn, Netherlands: International Arbitration Law Library Series Set. Kluwer Law International; 2004.
95. Arnold TW. The role of substantive law and procedure in the legal process. Harvard Law Review. 1932;45(4):617–47.
96. Zamani M, Movahedi M, Raykova M. Rapidchain: Scaling blockchain via full sharding. In: Proc. of the ACM SIGSAC Conf. on Computer and Communications Security (CCS'18), 2018; 931–948