Docker as a service on the blockchain

Matteo Nardini

Supervisor: Prof. Sven Helmer
Prof. Nabil El Ioini

Faculty of Computer Science
Free University of Bolzano

This dissertation is submitted for the degree of
Bachelor in Computer Science

20 July 2018

Cryptocurrencies:
everything you don't understand about money combined
with everything you don't understand about computers.
- John Oliver, Last Week Tonight

Abstract

Cloud providers detain an oligarchical dominance on the market of computational power. Recent developments in blockchain technologies and decentralized markets can potentially solve this issue. We explored the possibility of creating a decentralized market for computational power by using the Ethereum blockchain and Docker containers. In order to do this, we designed a protocol to exchange computations requests and rewards for executing them. Additionally, we developed a smart contract implementing this protocol and a client application that allows users to interact with this contract.

I cloud provider detengono un dominio oligarchico del mercato del potere computazionale. I recenti sviluppi nelle tecnologie blockchain e nei mercati decentralizzati potrebbero concretizzarsi in una soluzione per questo problema. Abbiamo esplorato la possibilità di creare un mercato decentralizzato per il potere computazionale utilizzando la blockchain Ethereum e i containers Docker. Per poter fare questo, abbiamo inventato un protocollo per scambiare richieste di esecuzione di computazioni e ricompense per eseguirle. In aggiunta, abbiamo sviluppato uno smart contract che implementa questo protocollo ed un'applicazione che permette agli utenti di interagire con questo contratto.

Cloud-Anbieter besitzen eine oligarchische Herrschaft über den Markt der Rechenleistung. Jüngste Entwicklungen in Blockchain-Technologien und die Dezentralisierung der Märkte können dieses Problem möglicherweise lösen. Wir haben die Möglichkeit untersucht, einen dezentralen Markt für Rechenleistung mit der Ethereum Blockchain und Docker-Containern zu schaffen. Um dies zu erreichen, haben wir ein Protokoll erfunden, um Anfragen für die Ausführung von Berechnungen, sowie Belohnungen, auszutauschen. Darüber hinaus haben wir einen Smart Contract entwickelt, der dieses Protokoll implementiert, und eine Clientanwendung, die es Benutzern ermöglicht, mit diesem Vertrag zu interagieren.

Table of contents

1 Introduction

1.1 Background and motivation

Cloud providers allow anyone to perform complex computations at an affordable cost. Traditionally, in order to execute some intensive computation, one had to buy the hardware needed to run it in acceptable times, configure it and only then run the actual computation. This has several drawbacks, among which the initial cost of the hardware and the added complexity of maintaning it, which often were too high.

With the advent of cloud providers like *Amazon Web Services*, *Google Cloud Platform* and *Microsoft Azure*, this procedure became much more accessible: instead of having to buy the hardware and configure it, one could simply rent for a limited amount of time one (or more) already-configured virtual machine with the required characteristics from one of these providers and use it to perform the needed computation. This workflow is able to significantly reduce costs and has become a common usage pattern.

However, this shift from on-premise hardware to cloud providers requires the users to trust the providers: whenever a computation is offloaded to a virtual machine in the cloud, the issuer has to rely on the fact that the provider will not interfere with the normal execution of the computation and that it will keep the involved data private. Additionally, the high costs required to become a cloud provider make this an oligarchical market, in which only few providers are available: this makes it possible for them to collude and, for example, to artificially increase prices for their services.

1.2 Our goal

The recent developments in decentralized computing platforms, virtual currencies and markets has led us to question if and how they could be applied to the specific case of offloading computations. We would like to explore the possibility of selling and buying computational power in a decentralized manner, allowing also small players to offer their computational capabilities and therefore resolving the problem of the centralization of cloud providers. In this initial phase, we would like to focus on the simpler case of deterministic computations that do not involve any network connection or inter-process communication: in future works, other possibilities may be explored. Additionally, we would like to provide a system in which who performs the computation cannot cheat: in this way, we would also eliminate the problem of having to trust the fact that the performer of the computation will not interfere with it. In this project we are not going to focus on the privacy-related aspects of this problem: in particular, we will assume that the computation to be executed and its input and output data can be public.

1.3 Roadmap

The remaining of this thesis is structured as follows. Section 2 describes some of the current technologies that could be applied to our problem and why they are not feasible.

Section 3 lists in more details the actual requirements that our solution will have to meet and section 4 gives a quick introduction to container technology and to the Ethereum blockchain. Section 5 explains in details how our solution to the problem works and its architecture, describing also some significant implementation details. Section 6 analyses the solution we build, presenting some data on the overhead costs that have to be considered while working in a blockchain environment.

Finally, section 7 describes some of the upgrades that should be included in future versions of our application.

2 State of the art

2.1 Verifiable computing

Using current blockchain technology, building a fully decentralized market is possible without excessive engineering effort: the difficult part is ensuring that what the parties agree on on the blockchain is actually respected in the real world. In particular, in our case, the publisher of the computation does not trust who performs the computation: therefore, even if on chain they agree on the fact that a certain computation should be executed, the publisher cannot be sure that the results submitted are actually those generated by running the requested computation.

While researching a solution to this problem, we found that there exists an entire field of computer science known as *Verifiable Computing*, which investigates and addresses exactly this problem. In this field, the problem is usually described as involving two parties, a prover and a verifier. 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.

Of course, the verifier should be able to check the correctness of this proofs in a time that is substantially less than the one required to perform the computation by himself.

There has been a lot of research on the theoretical aspects of this problem, which led to some remarkable results. In particular, for every language $L \in NP$, a *Probabilistically Checkable Proof* (PCP) can be constructed such that a claim from a honest prover is always accepted and that a verifier is able to detect and reject a false claim from a dishonest prover with probability of at least $1 - 2^{-k}$ (for a suitable security parameter k). Additionally, these proofs can be made *non-interactive*, meaning that after an initial setup phase, the publisher and the verifier need to exchange a single message in order to show the validity of the PCP. The most important properties of these PCPs is that they are asymptotically less complex than the original computation: this means that a verifier can check a proof in asymptotically less time than that required to actually perform the computation. A more detailed summary of how PCPs works can be found in Walfish and Blumberg [9], while a more in-depth theoretical explanation of these results can be found in Goldwasser et al. [4].

These theoretical results have already found some very significant practical implementation in the form of SNARKs [1]: these proofs are already successfully employed in cryptocurrency environments like ZCash [8]. More related to our specific problem, we found in Pepper [6] an implementation that allows to perform efficient verifiable computing on arbitrary computations. Unfortunately, while Pepper is efficient when compared to other implementations that aim at performing generic verifiable computing, it is still far from the performance required by real-world applications: transforming a common computation in a verifiable one entails a computational overhead of at least three order of magnitude, making this technology infeasible.

2.2 Others

Verifiable computing would be the actual solution to our problem, but unfortunately it is not usable yet. Thus, we started looking for alternative strategies. One of them [5] suggests 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 techinque is based on basically re-executing the entire computation, which is not desirable in our case.

Another strategy [3] suggests to embed in the computation a piece of software that is able to perform a sort of heartbeat protocol to guarantee that the application is actually running. This idea relies on security by obscurity and we do not see it fit for our purpose.

Premnath and Haas [7] describe an interesting application of the idea of garbled circuits and allows to execute a computation in a way that preserves privacy and, as a side effect, is (partially) verifiable. While this would solve most of our problem, the performance and storage costs are again too high.

3 Requirements

Given that verifiable computing is not a viable technology yet, our goal has slightly shifted. Instead of trying to develop a decentralized market in which who performs the computation cannot cheat, we will focus on creating a decentralized market in which a dishonest participation is strongly disincentivized. In particular, our idea is to create an application that uses the Ethereum blockchain technology to provide an effective foundation for the decentralized market. This application should have the features described in the following paragraphs.

The application should allow users to *publish computation requests*. Every user should be able to publish to the market a computation request in the form of the name of a Docker image. This image contains the entire description of the computation that has to be performed and should be publicly available.

The application should allow users to *perform published computations*. Users should be able to see the list of computations that have been published but not yet executed and should be able to perform these computations, if so they decide. Users that perform computations will be called *farmers*[1] from now onwards.

The application should allow farmers to *submit* the results of the computation back to the publisher, once they perfomed the computation. While doing this, farmers should commit in some way to the result they publish, so that later one could check that the result received is actually the one produced by the farmer.

The application should allow publishers to either *accept* or *reject the results* submitted by the farmer for a computation they have published. When a result is marked as accepted, the farmer should be payed for his service. When a result is marked as rejected, it should be checked by a trusted party to determine if the farmer actually cheated or not.

All of these actions should be exposed to the users through a graphical interface which, behind the scenes, interacts with the blockchain to ensure that all the data is processed correctly.

Additionally, the application should implement some kind of protocol that strongly discourages dishonest players. In particular, this protocol should encourage farmers to execute the computation without cheating and publisher to reject the result only if it really is not correct. This protocol should be less resource intensive to run than one based on complete verifiable computing.

[1] We initially wanted to use the term *miner*, but in blockchain environment it is already used to mean something different.

4 Preliminaries

4.1 Docker

In order to build a decentralized market for computational power, we needed a standard and portable way of representing a computation and its environmnent. The naive idea could be to simply distribute the application's binaries, but this poses a series of challenges: even simply agreeing on the command to use to start the application could be problematic.

The solution to this problem is to publish the computation using a container, that is, a light-weight virtual machine that can contain an application and all the information needed to run it. Using containers offers a series of benefits:

- Containers are isolated from the host system running them. This means that farmers can execute computations without compromising their system.
- Containers are volatile, in the sense that every modification from the starting state is not persisted when the container is shut down: therefore, executing a container for a deterministic application twice will yield the same exact results.
- A single host system can run multiple containers, allowing farmers to perform multiple computations simultaneously.
- Containers can be limited, so that no single container can completely exhaust a resource of the hosting system.
- Containers are portable, in the sense that once a container is assembled, every system for which that specific container engine is implemented can run that container.
- The 'inside' of the container can be completely configured by whoever creates it, allowing the publishers to configure the execution environment of the published application as they want.
- Containers are easy to create, use and share.

An industry-standard implmentation of the container technology is Docker. This implementation is open-source and freely available and is used by many software vendors to run their systems. Therefore, we chose Docker [2] as the way of packaging and publishing computations in our system.

4.2 Ethereum and the blockchain

Among the many blockchain platforms available, one of the few that offers a Turing-complete computing environment is the Ethereum blockchain. Additionally to the standard transactions that allow to exchange the cryptocurrency, Ethereum transactions can also be used to either create a smart contract or to invoke some method on an already-deployed one. A smart contract is an application whoose executable code and internal state are stored on the blockchain. Whenever a user creates a transaction that invokes a method on a smart contract, the piece of program associated with that method is executed and validated by every peer that accepts the block containing that transaction. This allows users to create decentralized applications (DApps) that are stored and executed on the blockchain and that inherit all the properties that are usual in a blockchain enviroment: all the nodes agree on the current state of the various DApps and the history of each modification to this state is recorded on an append-only ledger.

In our case, we plan to write one of these smart contract and to use it as the backbone of our application. By creating a DApp we can easily create a distributed marketplace, where publishers and farmers can agree on which computations should be executed and which are the associated rewards and fees.

5 Proposed solution

5.1 Protocol

Standard scenario

Our work on the project implementation begun by designing the protocol that the smart contract should use when communicating with the other parties. 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, which interacts with the smart contract to publish a new computation request. This request contains some parameters describing the computation and transfers to the contract's account the amount of Ether that corresponds to the reward for performing this computation. After doing some validation on the input provided by the publisher, the smart contract emits a `ComputationPublished` event to signal that a new computation has been published. This event is broadcasted on the blockchain and all the farmers can listen for it.

Every farmer receives this event and decides whether to perform this computation or not. When a farmer chooses to execute this computation, he or she sends a request to reserve the computation to the smart contract, so that other farmers know that that computation is already taken. At this point, the smart contracts checks that that computation is still available: if this is the case, it assigns that computation to the requesting farmer and publishes a `ComputationAssigned` event. The requesting farmer can use this event as a confirmation that the request sent was accepted and thus can start executing that computation. Other farmers can listen for this event and know that that computation is already being performed. If the computation was already assigned to someone else, the smart contract simply rejects the request: the farmer will not see the `ComputationAssigned` events and thus will understand that the reservation request was denyed, as shown by the diagram in figure 2.

When the computation is performed, the farmer can send to the smart contract a message contatining the result hash: the smart contract checks that the farmer is the one for which that computation is reserved and, if this is the case, it saves the hash and emits a `ComputationDone` event. The publisher of the computation can listen for this event and promptly retrieve the result of the computation from the publisher (mode details in subsection 5.3). After checking the results, the publisher can choose to either accept or reject the results: if the results are accepted, the smart contract marks the computation as accepted and emits a `ResultAccepted` event. The farmer that executed this computation can listen for this event: when it is received, the reward can be withdrawn.

Result rejected

Unfortunately, verifiable computing is not practical yet. This means that the correctness of a computation has to be checked in another way: we decided to create an adjudicated protocol in which a trusted third party performs auditing duties, but only when the publisher rejects the results submitted by the farmer. This decision was made under the assumption that most of the users on the platform will behave honestly, especially given the incentives described below. The sequence diagram describing this part of the protocol is shown in figure 3.

When a publisher rejects a result, the trusted auditor comes into play. This auditor re-executes the published computation and then computes the hash of the result he or she obtains. This hash is then sent to the smart contract, that confronts it with the hash submitted by the farmer. If the two hashes matches, then the farmer behaved properly, which means that the publisher should have accepted the result: the smart contract acknowledges this and marks the farmer as allowed to withdraw the reward. If

Fig. 1 Sequence diagram describing the standard scenario of interaction with the smart contract

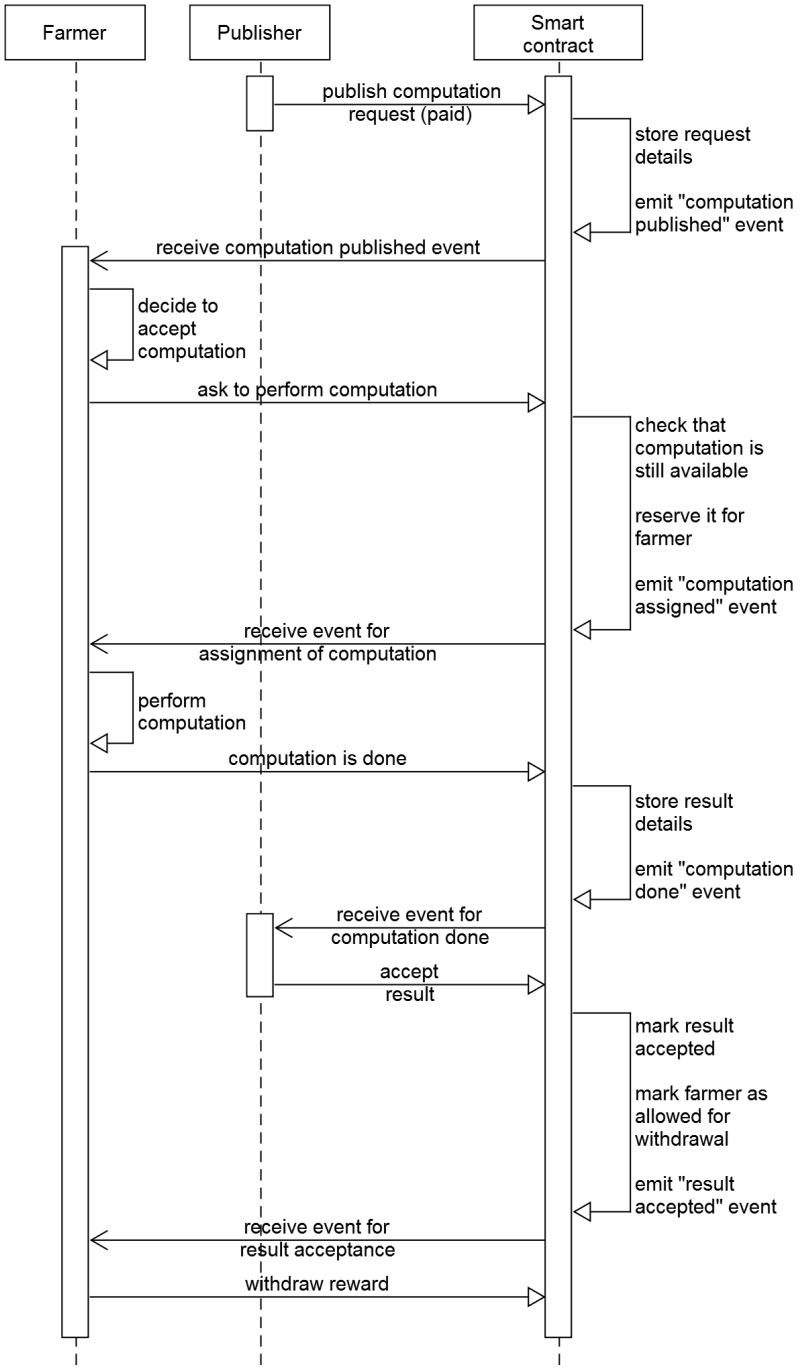


Fig. 2 Sequence diagram describing the alternative scenario in which a computation was already reserved for another farmer

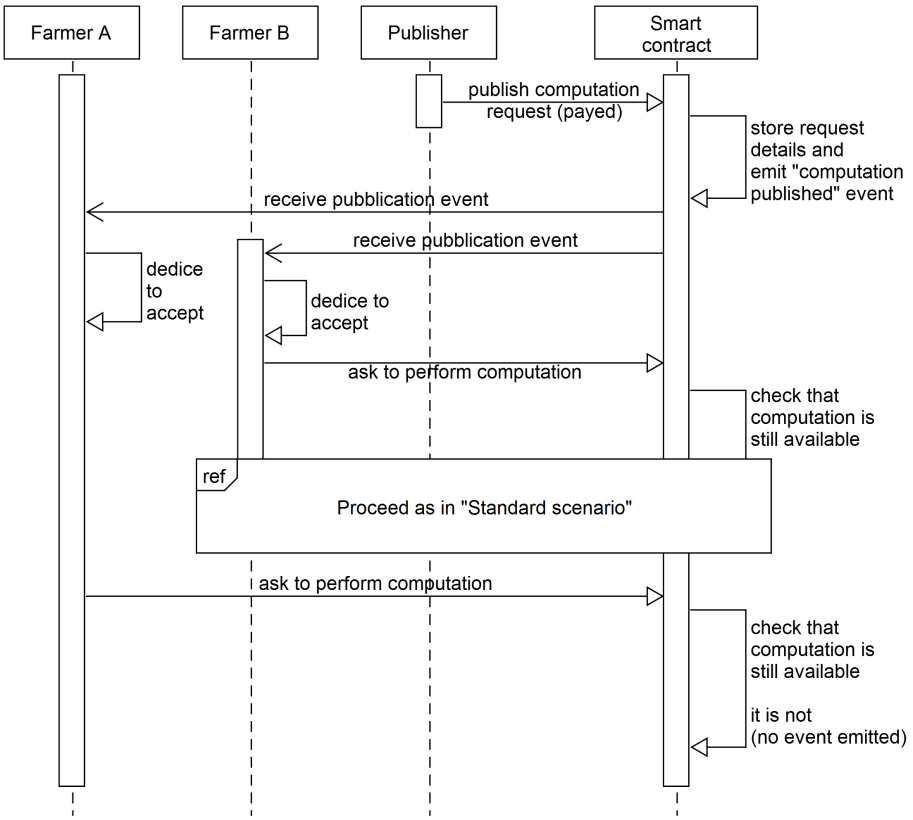
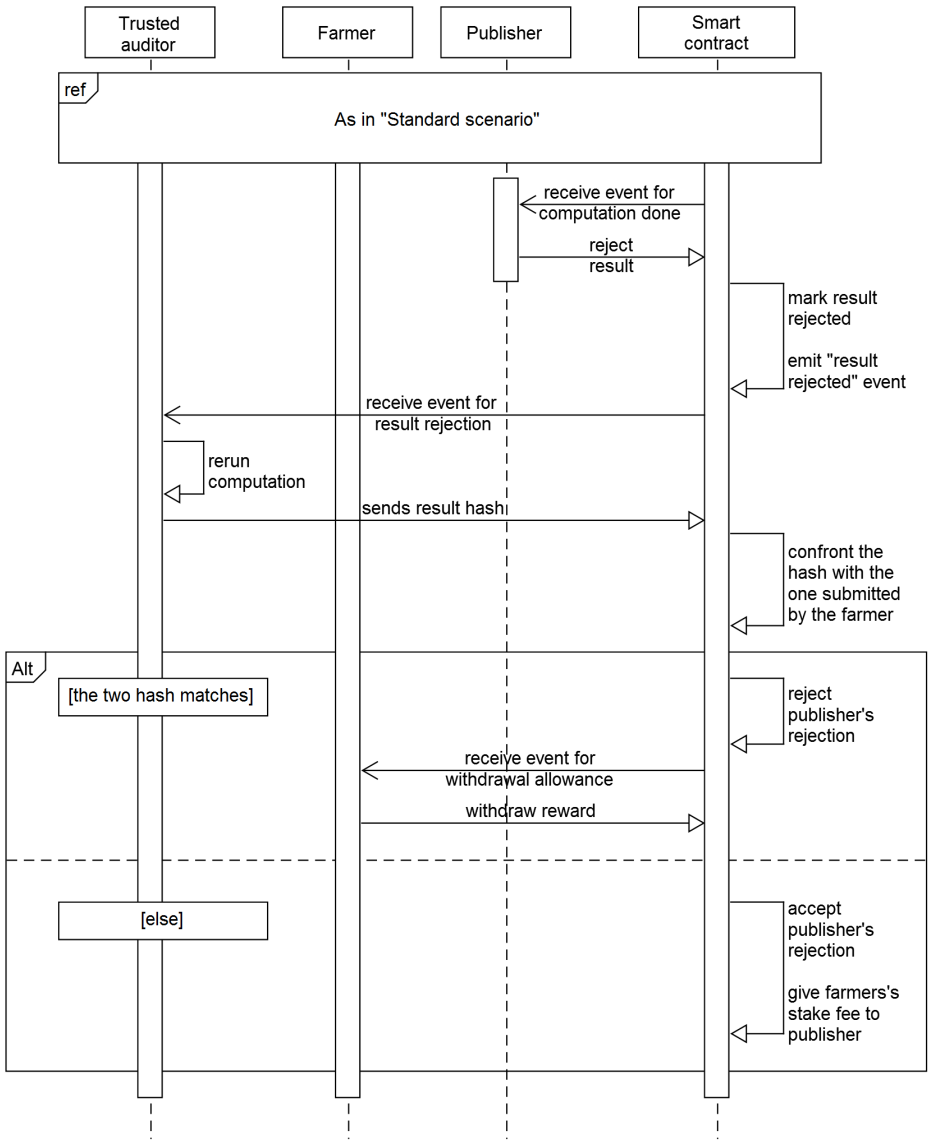


Fig. 3 Sequence diagram describing what happens when a publisher rejects the submitted results



the two hashes do not match, then either the farmer was dishonest or used faulty hardware or software and the publisher rejection is confirmed.

In order to encourage the parties to behave properly, we introduced in the protcol some incentives in the form of a *minimum stake fee*, a *stake fee* and an *audit fee*. Each computation is published with a minimum stake fee decided by the publisher. When the farmer submits the result of a computation, he or she has to pay as part of the transaction an amount that is at least as much as indicated in the minium stake fee: this payment is the farmer's stake fee and is kept temporarily on hold by the smart contract. If the publisher accepts the result, the farmer can withdraw both the reward for the computation and the stake amount payed when submitting the result. If the publisher rejects the result, the farmer's stake fee is kept on hold until the auditor validates the rejection. If the auditor proves that the farmer submitted a wrong result (either because of a dishonest behaviour or because of faulty hardware or software), the stake amount put on hold by the farmer is given to the publisher and the farmer loses every right on it.[2] If the auditor proves that the farmer submitted a correct result, the farmer is allowed to withdraw both the reward and the stake fee.

If this was the only incentive put in place, then every publisher would reject the submitted results, in order to have them checked by the auditor for free. In order to avoid this behaviour, we designed an additional incentive called *audit fee*. Alongside with the result rejection, the publisher has to pay an audit fee. This audit fee is given to the auditor as soon as the auditing duties regarding this computation are done, independently from the verdict of the auditor. The audit fee is not fixed and can be choosen by the publisher: auditors will privilege computations with higher audit fee, therefore a publisher desiring a quick check will have to pay an higher fee.

The combination of *stake fee* for the farmers and *audit fee* for the publisher should give the right incentives to the platform and reduce the number of situations in which entities choose to behave dishonestly.

Other alternative scenarios

While creating the protocol described above, we detected a couple of situations in which there could be deviations from the standard scenario.

The first potential problem happens when a farmer accepts a computation and then disappears. In this case, another farmer can challenge the assignment of the computation as shown in figure 4.

The second problem happens when the publisher disappears without neither accepting or rejecting the results. In this case, after a certain period of time has passed, the farmer that performed the computation should be able to challenge the publisher and be rewared in any case, as shown in the sequence diagram in figure 5.

5.2 Architecture

In order to implement the protocol described in subsection 5.1 we devised a system divided in two main component: a smart contract living on the blockchain and a client application that interacts with it.

The smart contract is written in the Solidity programming language, which is the standard language for the Ethereum blockchain. It stores on chain a list of `Computation` objects, where each object contains all the details of that computation. The public interface of the contract is composed by a list of methods that implement the protocol described in the previous subsection. Each of these methods

[2] Some may argue that the punishment for a farmer with faulty hardware should be lighter than the one for a cheaty farmer. We agree with this thought, but the fact is that the smart contract has no way of differentiating the two behaviours, thus we prefer to err on the side of caution. A honest farmer should interpret the result rejection as a signal that their system needs to be fixed.

Fig. 4 Sequence diagram describing the alternative scenario in which the farmer to whom the computation is assigned disappears

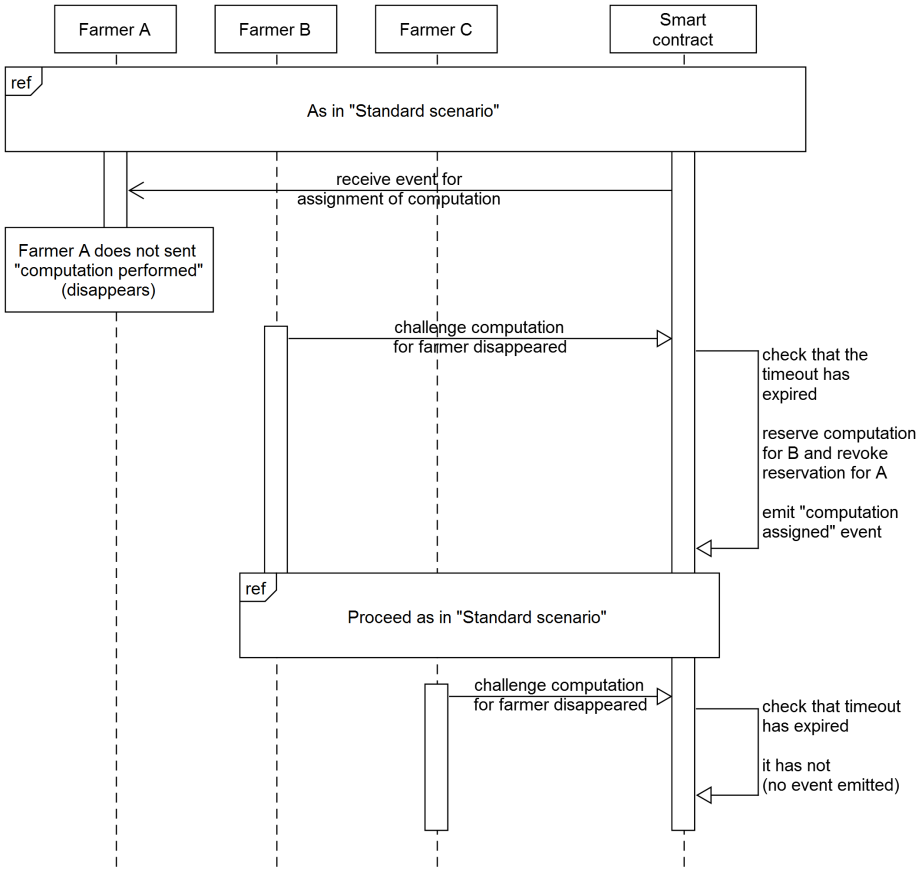
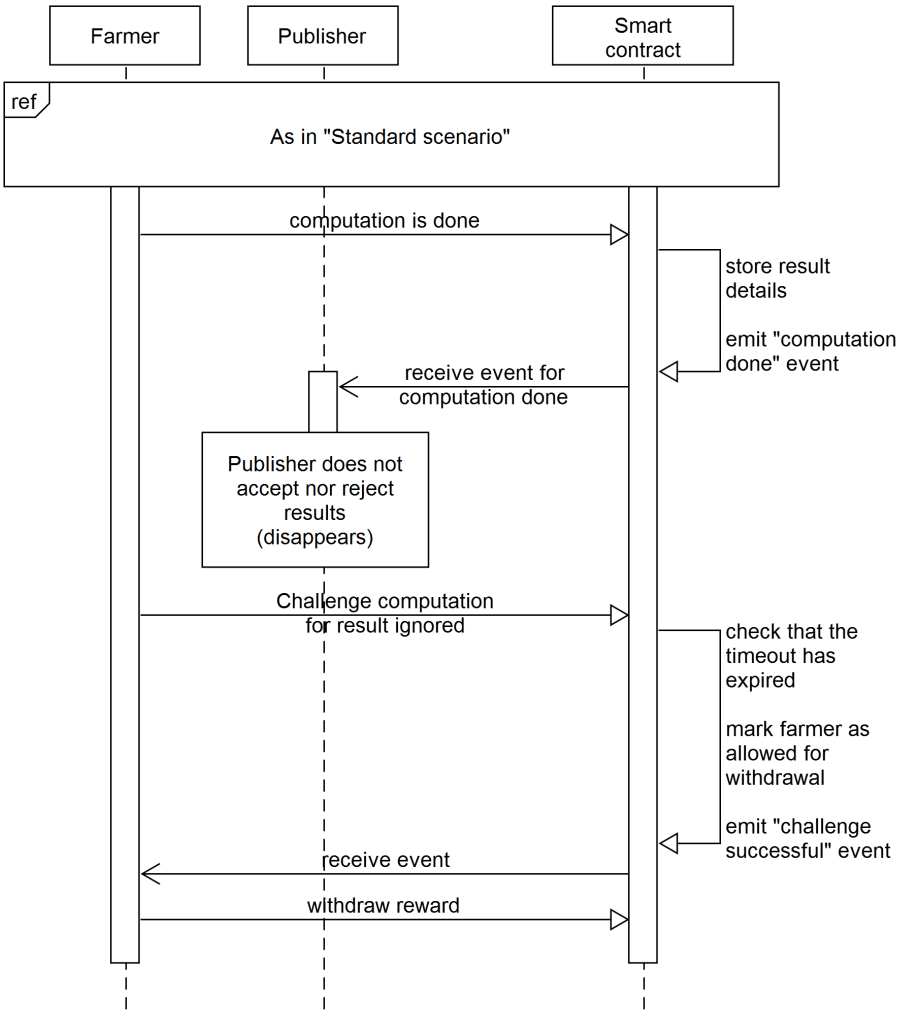


Fig. 5 Sequence diagram describing the alternative scenario in which the publisher of a computation disappears without neither accepting nor refusing the results



receives some input from the client application, performs some checks related to the status of the computation and determines if the request is valid. If not, an error is thrown. If yes, the requested modifications are applied to the stored computation object and the required events are emitted. For each new published computation, the contract generates an ID that will be used to reference that computation. On the development side, the contract uses the Truffle framework to enable easier unit-testing of the contract and easier deployment both to a test environment and to the real one.

Most of our focus was on developing a functional and correct smart contract, thus the client application has to be considered more as a proof of concept rather then an actual implementation. This application interacts with the smart contract and is a web application divided in three components:

1. A Geth client connected to the desired blockchain (either the main one or one of the test ones).
2. An Express server that interacts with the Geth client using JSON-RCP over WebSockets and that serves the frontend to the user. Additionally, the server listens for events on the blockchain and takes the needed actions, also by interacting with the Docker daemon. 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 that computation, start the associated container, collect the results and automatically send their hash 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.

5.3 Implementation details

Smart contract

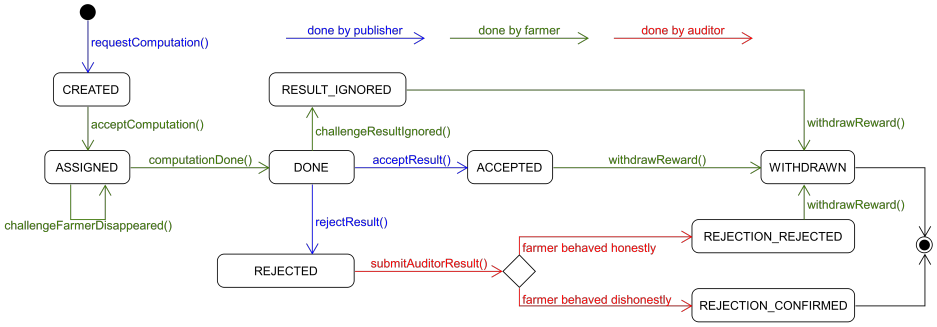
The smart contract is split over two Solidity contracts, that are then combined to a single one before the deployment. The `Administrable` contract maintains informations related to the owner of the contract and the auditors; the `Main` contracts stores all the computations' metadata alongside with their information and contains the methods that allow to implement the protocol described in 5.1.

In this proof of concept, the `Administrable` contract is very simple. 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 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 any parameter, but this was beyond the scope of our proof of concept.

The contract `Main` 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. In this diagram are also reported the name of the methods offered by the smart contract to perform that status transition. Every method can be invoked only by a specific entity (publisher, farmer, auditor) and these contraints 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 position of this computation in the state machine described in 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 after performing the computation.
- `assignedTo, assignationTimestamp`: the address of the farmer to whom the computation is assigned and when this assignation was made. The timestamp is needed to check if a challenge from another farmer is valid or not.

Fig. 6 State-chart diagram describing the state transition allowed by the computation object alongside with who can trigger each transition



- `minStakeFee, stakeFee, auditFee`: used to store the incentives described above. All the amounts are in Wei.
- `resultHash, resultLink`: the hash of the result obtained by the farmer and the link at which the publisher can download the full results.
- `resultSubmissionTimestamp`: used to check whether a challenge for result ignored can be successful or not.

The methods implemented in the smart contract are all quite similar to the `acceptComputation` method described in listing 2. Most of these method take as first parameter the `id` of the computation on which the action has to be performed and use that id to retrieve the computation object associated to it. After that, some checks are performed: usually, these checks regard the existance of a published computation with the given id and the fact that that computation is in the correct state for the required operation. Some methods may have check the additional input parameters received: for example, the `submitResult` method must check that the amount payed by the farmer is greater or equal to the `minStakeFee` requested by the publisher. After that, the retrieved computation object is modified in some way, depending on the specific operation: these changes are persisted on chain and usually change the computation's state. When all the modification required by the current operation are done, an appropriated event is usually emitted. Also these events are stored on chain and all the users can listen for them in order to be notified when a computation changes.

Client application

The Express server is composed by a few different modules that interact among each other by using a shared event bus. Most of the events on the bus are generated by a set of listeners that watch for the emission of specific events by the smart contract. The other modules listen for these events on the bus and initiate the needed actions. In particular:

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 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, this module monitors the running containers and, when one of them exits, notifies it to the event bus with a `job-finished` event.

Listing 1 Definition of the structure that describes the computation object

```
struct Computation{
    Status status;

    address publisher;
    string dockerImageName;
    uint weiReward;
    uint minStakeFee;

    address assignedTo;
    uint assignationTimestamp;

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

    uint auditFee;
}
```

Listing 2 The method that allows a farmer to ask for the assignation 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);
}
```

3. The `UploadManager` listens for these `job-finished` events, collect 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 marks the current farmer as allowed to withdraw, like the acceptance of the results or one of the alternative scenarios. When one of these events is received, it creates the relative withdrawal request.

Other details

In order to properly work in the current system, there are a few other implementation details that should be known.

When a computation is published, the complete name of a Docker image has to be given. One can refer to a specific version of a Docker image either by giving the image name and a tag or the image name and the image digest (usually in the form of a `SHA-256` hash). Image tags are mutable: the tags associated to a specific Docker image can be changed over time without effort. Instead, image digests are immutable: whenever a image is generated, its digest is computed and cannot be changed. Additionally, clients downloading the image by using its digest can check that the image they get is actually the one corresponding to that digest.

Farmers should execute only computations that have a Docker image name that uses a digest: in this way, if the auditor needs to re-execute a computation, the farmer can be sure that the auditor will execute *exactly* the published computation and not something else. Farmers should not perform computation that use tags in their name because the publisher would be able to cheat. Take for example the situation in which a farmer accepts to perform the computation contained in the image with name `my-image:a-tag`, where `a-tag` is the tag associated with this image. After the result submissions, the publisher could rebind the tag `a-tag` to another version of `my-image` that produces a different result and only then reject the farmer's result: when the auditor will re-execute the computation, it will appear that the farmer results were wrong even though they were not. In order to avoid this situation, farmer should only perform computation that use digests in the image name.

The computation contained in the Docker image should store the results in the `/result` folder inside the container. The farmer will mount a host folder to that location: in this way the computation result will be retrievable and uploadable. In order to submit the result, the farmer should compress the folder mounted to the `/result` location inside the container by using gzip and then compute the hash of the archive obtained in this way. This is the hash that should be submitted to the smart contract. The auditor will perform the same steps when solving a dispute.

The farmer should also provide to the smart contract a link to where the result zip has been uploaded. This link should be accessible through a simple HTTP GET request and no additional authentications should be required. This is one of the aspects that could be improved in future works.

6 Evaluation and discussion

6.1 Cost evaluation

One of the important aspects that have to be evaluated when running computations on the Ethereum blockchain is cost. Every transaction, that is, every call to one of the smart contract's methods that alters the state of the contract, has a cost associated with it. This cost is determined by two parameters: the `gas` used by the execution of that method and the `gasPrice` associated with that transation. The first parameter depends on the requested computation: every assembly instruction executed by the Ethereum Virtual Machine has a certain gas cost associated with it. The second parameter describes the cost in Ether of one gas unit and depends on the blockchain situation when the transaction is performed: if a big number of transactions are pending, those with higher `gasPrice` have higher probability of being executed by a miner and be therefore added to the chain. There is a third parameter influencing the actual real-world cost of executing a transaction: the exchange rate between Ether and the non-crypto currency of reference (which, in our case, is Euro). The volatility of this exchange rate is often very high, therefore significantly altering the costs of using a service from day to day.

We executed some tests on our implementation of the smart contract, running both the standard scenario and the alternative ones. 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 and by the various scenarios: this amount of used gas can then be multipled by a choosen `gasPrice` to obtain the costs in Ether. Applying the Ethereum to Euro conversion factor to these prices allows to compute a cost in non-crypto currencies. All the tests have been run against the smart contract deployed at addess `0xd82429497c69208a358ece305efc7aba4b237fe2`.

In table 1 we report the cost in gas for running each of the method of the smart contract. We also computed the relative cost in Ether assuming a `gasPrice` of 17.011103191 GWei , which is the average amount currently used on the main Ethereum network. Finally, we also converted the Ether price in Euro assuming an exchange rate of 1ETH = 537.257818083EUR, which is the current exchange rate at the time of writing.

As can be seen, even tough the computation executed by the smart contract are very straight-forward and concise, the costs are still not negligible. This becomes more evident in tables 2 and 3, which show the costs of executing the full standard scenario and the cost of the auditor intervention. These tables show the total amount of gas, Ether and Euro spend by each party and are computed by summing the costs of the individual method call reported in table 1.

The costs reported in these tables have to be considered as indicative. As shown in figure 7, both the `gasPrice` and the Ether to Euro conversion rate vary significantly from day to day, causing the actual price of using this smart contract to fluctuate abountantly and making it impossible to compute accurate predictions.

The last cost that has to be discussed is the contract deployment cost. The deployment scheme we used is the one suggested by the Truffle framework. The first deployment on a chain requires deploying two contracts: the actual smart contract we are interested in and and additional `Migrations` contract managed by Truffle that keeps track of which migration scripts have been run on the current chain and which have not. Deploying these two contracts on the Rinkeby test network has some significant costs, as described in table 4. This data can be obtained by inspecting the data associated with the following transactions:

Table 1 Costs of executing the various smart contract methods. This table assumes a `gasPrice` of 17.011103191 GWei and an exchange rate between Ether and Euro of 1ETH = 537.257818083EUR.

Method	Gas consumed [Units]	Ether cost [GWei]	Euro cost [€]
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 of executing standard scenario. This table assumes a `gasPrice` of 17.011103191 GWei and an exchange rate between Ether and Euro of 1ETH = 537.257818083EUR.

Party	Gas consumed [Units]	Ether cost [GWei]	Euro cost [€]
Publisher	167898	2856130.20	1.53
Farmer	226341	3850310.11	2.07
Total	394239	6706440.31	3.60

Table 3 Costs of executing the result-rejected scenario. This table assumes a `gasPrice` of 17.011103191 GWei and an exchange rate between Ether and Euro of 1ETH = 537.257818083EUR.

Party	Gas consumed [Units]	Ether cost [GWei]	Euro cost [€]
Publisher	188069	3199261.17	1.72
Farmer	182843	3110361.14	1.67
Auditor	38551	655795.04	0.35
Total	409463	6965417.35	3.74

Fig. 7 The following plots display how the `gasPrice` on the Ethereum main network and the Ethereum to Euro conversion rate change from day to day, causing significant differences in the price of actually using this smart contract

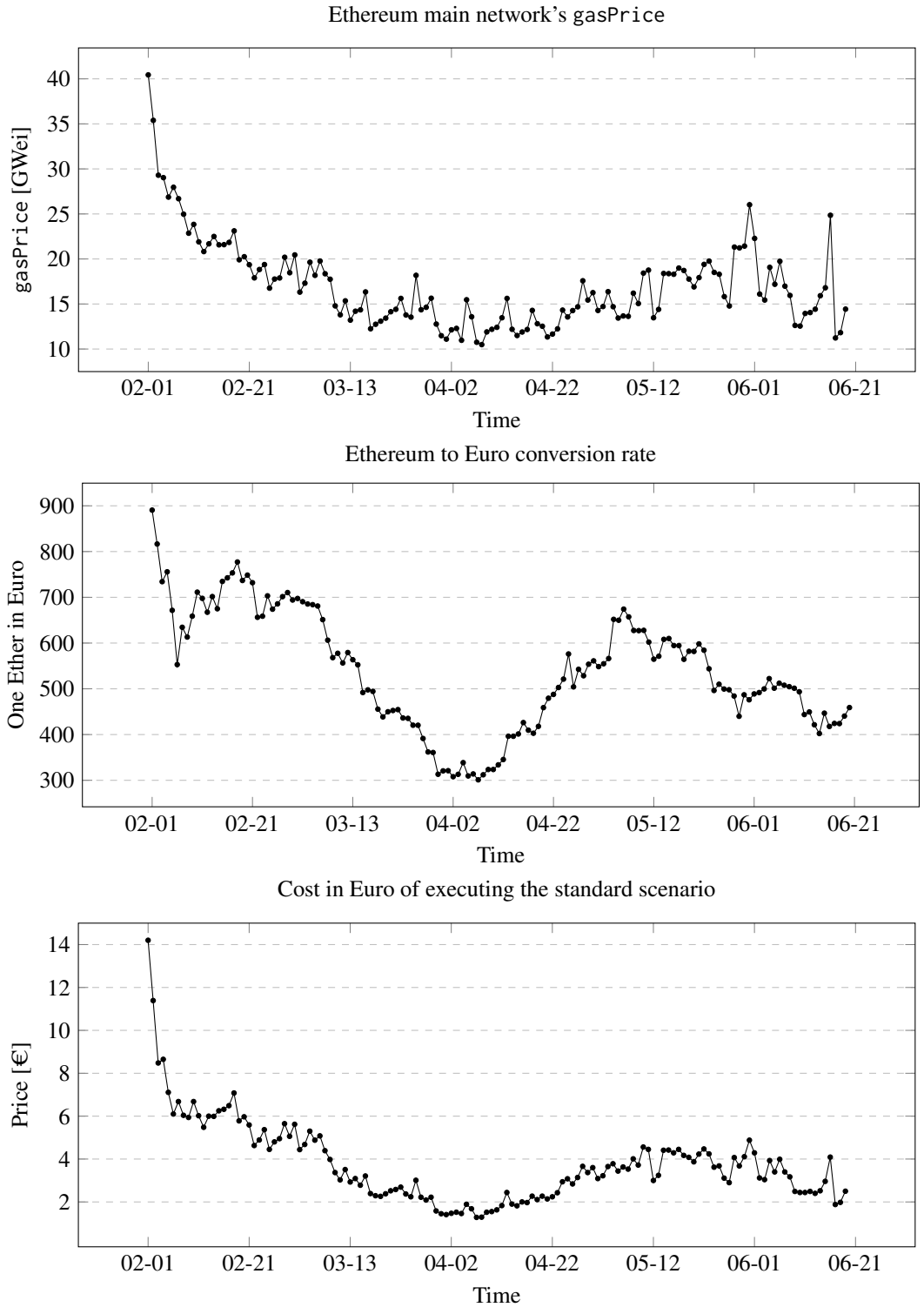


Table 4 Costs of deploying the smart contract. This table assumes a `gasPrice` of 17.011103191 GWei and an exchange rate between Ether and Euro of 1ETH = 537.257818083EUR

Contract	Gas consumed [Units]	Ether cost [GWei]	Euro cost [€]
Migrations	319470	5434537.14	2.92
Main	3158630	53731780.87	28.87
Total	3478100	59166318.01	31.79

- `0x9449167b4dd3e9e3d3a4c4920de87c94db0e4c03b6be8e8d42991a6aa6ff9121`: deployment of the `Migrations` smart contract
- `0x66ffefb48f188c02334eeaeb15bcd65a9237118d6fcfdc95ff6e791806426eee`: update of the last successful migration
- `0x64a94e73a84e716ab4b27947ff2b1c3c44ae86e22e48ca3e7314f8864c2093b1`: deployment of the `Main` smart contract
- `0xde1f4e7590972b5bc4caa1a2c84019d0959be22d756a145f671f1882357c4a19`: update of the last successful migration

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

6.2 Discussion

As already said, the true solution to the problem encountered while desining this application would be verifiable computing. Unfortunately, the technology is not viable yet and this forced us to take an alternative path. Overall, we think we succeded in our goal of providing a decentralized market for computational power in which the incentives put in place offer some degree of reliablity to the users of the system.

However, some important consideration that emerged from the development of our proof of concept have to be discussed. In particular, the system has some non-negligible costs for the users and these fees are much higher than those offered by common cloud providers: this could significantly slow down the adoption of our solution. An additional concern is the overall slowness of the system: in order to be mined, every transaction takes a minimum of fifteen seconds (the mining time of a block) and this could pose some severe limitions on the scale at which this system could be employed.

A final point that should be considered is the difficulty of updating the smart contract (that is, deactivating the old one and deploying a new one). The costs involved make this process very onerous and limit the number of updates that can be released. Another aspect of this update procedure, that makes it even more difficult, is that all the users of the service must be notified of the address of the new contract: this is notoriously difficult in a decentralized environment like the blockchain one.

7 Conclusion and future work

7.1 Smart contract

The smart contract implementation that we developed takes into consideration most of the use cases described in section 3. However, there are still some aspects that should be improved. In particular, in the current version, the handling of the ownership of the contract and of whom can perform auditing duties is very limited. The current system could be improved by having more than a single trusted member and by requiring some kind of voting procedure before changing the contracts' configuration parameters. Another enhancement would be to better distribute the auditing duties, instead of forcing the contract's owner to be the only auditor.

Another situation in which some improvements are required is the result's submission methodology, which, as of now, is again very limited: for example, the publisher could specify a public key with which the results should be encrypted before being uploaded. A final enhancement would be to provide an administrative method that allows to perform some garbage collection: as of now, all the metadata regarding past computations (e.g. Docker image name, publisher address, ...) are stored on chain. There should be a way for someone with administrative rights to trigger a cleanup that removes all computations that cannot change state anymore.

7.2 Client application

The client application is really bare-bone and has been developed only to have a practical way of interacting with the smart contract. Further versions of this application could be improved in several way and would probably require a complete redesing of the user interface. Additionally, this client application should become more automatic: in the current version, many actions are manual. In particular, accepting to perform a published computation is a manual step which could be automated: the farmer should simply specify some contraints (e.g. on the reward or the stake fee) and then the application should automatically try to accept the computation that satisfy these contraints. Another improvement could be to automatically download the results of a computation when they are accepted and check that the hash of the downloaded archive matches the one reported by the farmer.

An additional enhancement could be to only consider Docker image names that contain the image digest and automatically discard all published computations that rely on image tags instead. A final improvement that should be done is to embedd auditing functionalities in the application: as of now, the client application is not able to act as an auditor.

References

[1] Ben-Sasson, E., Chiesa, A., Tromer, E., and Virza, M. (2014). Succinct non-interactive zero knowledge for a von neumann architecture. In *Proceedings of the 23rd USENIX Conference on Security Symposium*, SEC'14, pages 781–796, Berkeley, CA, USA. USENIX Association.

[2] Docker Inc (2018). Docker - build, ship, and run any app, anywhere.

[3] Falcarin, P., Scandariato, R., Baldi, M., and Ofek, Y. (2005). Integrity checking in remote computation.

[4] Goldwasser, S., Kalai, Y. T., and Rothblum, G. N. (2015). Delegating computation: Interactive proofs for muggles. *J. ACM*, 62(4):27:1–27:64.

[5] Khan, S. M. and Hamlen, K. W. (2013). Computation certification as a service in the cloud. In *2013 13th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing*, pages 434–441.

[6] Pepper-project (2012). Pepper: toward practical verifiable computation.

[7] Premnath, S. N. and Haas, Z. J. (2014). A practical, secure, and verifiable cloud computing for mobile systems. *CoRR*, abs/1410.1389.

[8] Sasson, E. B., Chiesa, A., Garman, C., Green, M., Miers, I., Tromer, E., and Virza, M. (2014). Zerocash: Decentralized anonymous payments from bitcoin. In *2014 IEEE Symposium on Security and Privacy*, pages 459–474.

[9] Walfish, M. and Blumberg, A. J. (2015). Verifying computations without reexecuting them. *Commun. ACM*, 58(2):74–84.