TEST

This section allows the cross link from pdf to latex code work. Delete this section before finalizing the paper.

Contents

Protocol, Strategies and Analysis for Enabling a Distributed Computation Market for Stream Processing

Anonymous

ABSTRACT

Online processing, frequently referred to as stream processing, allows the Internet of Things (IoT) and Smart City services to extract useful information from real-time data. There are a number of state of the art stream processing engines, however, deploying them in practice is hard because these frameworks don't necessarily solve the problem of deployment and finding the necessary computation resources. Cloud computing is still expensive for city stakeholders, especially for real-time high-velocity and high-volume data analysis. We propose an architecture for aggregating outsourced computing services where customers purchase compute cycles from suppliers that have surplus compute cycles. The advantage of using these surplus resources is reduced costs. In particular, improving over the state of the art we consider the problems caused by the possibility of selfish and cheating participants. Further, we consider the challenge of acquiring resources for long enough to make streaming computations feasible. The time-sensitivity of streaming processes means the interactions between services must be accounted for and they are especially sensitive to failures. Our contributions include the description of the protocol and showing a methodology to combine the use of blockchain (for resilience) with recent improvements in the distributed log-based publish-subscribe ledgers like Apache Pulsar. Further, we present the strategic analysis and experiments, which show that we can configure the system so that rational stakeholders will use the system correctly and will benefit from participation.

CCS CONCEPTS

• **Information systems → Computing platforms**; • **Computing methodologies → Distributed algorithms**; • **Computer systems organization → Cloud computing**.

KEYWORDS

stream computation, decentralized market, pub sub systems, game theory, decentralized job scheduling, smart contract

ACM Reference Format:
Anonymous . 2020. Protocol, Strategies and Analysis for Enabling a Distributed Computation Market for Stream Processing. In *6th ACM/IEEE Conference on Internet of Things Design and Implementation, Nashville, TN, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Emerging Trends Online processing, frequently referred to as stream processing, allows Internet of Things (IoT) and Smart City services to extract useful information from real-time data. Example applications include real-time availability maps for dockless scooters [7], improved emergency response procedures [22], energy usage estimation and subsequent optimization for transit vehicles [6], real-time occupancy maps [19], and traffic density and pedestrian density estimation from intersections [23]. These applications require processing the data quickly to extract the time-value of the data. This is the purpose of modern stream processing engines such as Spark, Flink, Kafka, StreamQRE, and Heron [8, 13, 24, 34]. Many of these applications involve multiple stakeholders, require hardware deployment, and are highly customized. Such systems involve a large number of quality-of-service configurations which are very difficult to fine tune, thereby making these systems expensive and inflexible.

Applications can also be hosted on cloud computing platforms. However, this is expensive for cities[1], especially for real-time high-velocity and high-volume data analysis. Additionally, for some applications the latency introduced by communicating with the cloud is not acceptable. Recently, an alternative paradigm known as dispersed edge computing has been conceptualized [16, 26], where the compute of power of nearby resources is leveraged. However, this paradigm is still in nascent stages and does not necessarily address the problem of resource aggregation, for example by utilizing the idle computing reserves at the edge. This is because these approaches lack principled mechanisms that incentivize owners to share resources with other stakeholders. It should be noted that this idle *surplus computing capacity* is the compute infrastructure that remains unutilized after a resource has satisfied the requirements for its intended use case. It is estimated that there are hundreds of ExaFLOPs of surplus compute capacity available [2, 10]. The advantage of using these surplus resources is reduced costs[2] and the distributed nature of devices, which promotes resilience and diffuses network congestion. However, these devices are not currently accessible. This creates the need to design a framework that can effectively utilize spare computing capacity to outsource computation.

State of the art Aggregating surplus resources for outsourcing computational tasks using a market has been explored in the past [9, 14, 20, 29]. However such approaches are not suitable for an open market for outsourcing streaming computation for several reasons. First, some of the approaches only accept surplus compute from trusted entities like internet service providers [9, 20]. This assumption ignores the possibility of selfish participants. In an

[1]In a personal communication with a large cloud vendor a southern metropolis had to undertake an expense of $100K dollars per year for setting up real-time data processing applications for managing scooters.
[2]The space, hardware, cooling, and operations costs are sunk costs which are already paid for to support the devices' primary applications.

open market, however, agents might not be cooperative and may instead be competitive. The strategic considerations make it imperative to design a robust mechanism that the stakeholders can trust. Blockchain based distributed ledgers have been of interest in recent years for systems where there is no central trusted entity, instead trust is decentralized among the participants. However blockchain-only systems are inefficient, slow, and have limited throughput making them unsuitable for streaming applications which are time sensitive and long-running. Other solutions for outsourcing computation that do account for trust [14, 29] do not account for the fundamental requirements of stream processing.

Challenges Streaming processes are often long-running; surplus capacity, on the other hand, is transient and subject to the demands of the primary application. Thus, it is likely that Suppliers (agents that provide surplus compute) will not be able to host a service for its entire life-cycle. The time sensitivity of streaming processes means the interactions between services must be accounted for and they are especially sensitive to failures. A failure that requires restarting the service results in the loss of value from the data that should have been processed. Further, it is required that the outputs from distributed computing frameworks be verified without delay. Any verification that occurs must be essentially instantaneous or performed after the output has been released. Verification must also be inexpensive due to the volume of data processed.

Our Contributions This paper presents a solution that addresses the issues of resource availability, trust, reliability, and time sensitivity posed by outsourcing stream processing. To gain access to surplus compute capacity we construct a market that incentivizes Suppliers to participate by offering value for their otherwise wasted compute cycles. To handle the volatility inherent in surplus resources, Customers (stakeholders who have services to outsource) specify a minimum service time, and their offers can be broken up into sub-offers with a duration no shorter than the minimum service time. To address the potential for Suppliers to fail accidentally, Customers are able to request that their service be hosted by multiple Suppliers. *Allocators* are employed to construct allocations, matching Customer and Supplier offers according to feasibility constraints, including replicas and minimum service times. To address the issue of trust the market state is decentralized among the stakeholders, using a proof-of-work based distributed ledger. The ledger is used to record allocations and collect security deposits. We use the ledger to implement a Verifier to ensure that our mechanism is robust and trusted by all stakeholders. To improve fairness we allow participants to designate a trusted Mediator. If one is specified then the Verifier detects errors and notifies the Mediator that then determines which participants are at fault. To implement the market, overcome the limitations of the blockchain-based distributed ledger, and handle streaming services we use Pulsar, a distributed messaging and streaming platform.

This solution can provide the substrate for multi-stakeholder applications that rely on a dispersed computing paradigm. However, it requires careful consideration of the protocol and parameters of the system because individual participants can be assumed to be selfish and will optimize their strategy to maximize their utility. We show how our mechanism ensures that strategies that would undermine trust in the system are not viable. Thus, we emphasize the following aspects of our solution in the paper.

(1) We formally describe the design of the architecture and protocol of the system.
(2) We show that our solution enables trust. For this we provide the design and game theoretic analysis of an incentive compatible and individually rational market mechanism. We prove that if the participants are rational the system operators can set parameters to discourage cheating.
(3) Lastly, we describe real-world scenarios and examples obtained through our partners and show how the system will work in practice.

Our results show that with this design, rational participants will follow the protocol and benefit from participating in the system, while participants that deviate from the protocol incur fines.

Outline The outline of the paper is as follows... Ensure that related research is added into the paper.

2 SETTING UP THE PROBLEM

The problem we seek to address is the construction of a robust and trusted market for outsourcing streaming computation. To describe our system model we use the example depicted in Fig. 1a. For a list of symbols used in the paper, see Table 1. We start by defining the basic problem.

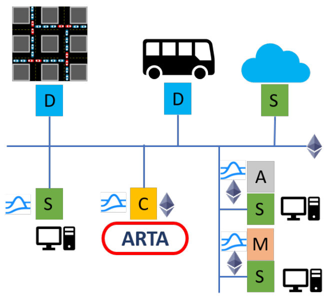
2.1 Basic Problem Formulation

Let μ denote a market for outsourcing of services that rely on stream computations. The market is formed by various agents. There is a set of Customers, denoted by C, who have computational services to outsource and can make offers to buy compute capacity. In Fig. 1a the Customer is an Area Regional Transportation Authority (ARTA). ARTA has access to a variety of sensor data including bus and traffic monitoring at intersections. These data sources are denoted by D in Fig. 1a. There is also a set of Suppliers, denoted S, who can make offers to sell surplus compute capacity and host the services. Examples of Suppliers include businesses, universities, end users, ISPs, and cloud providers.

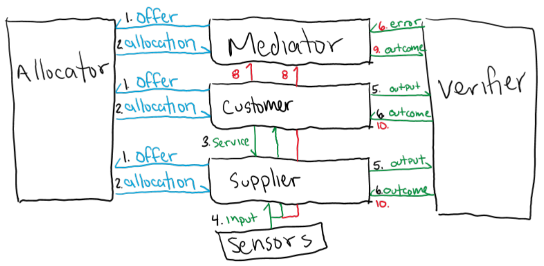
Services We denote a streaming service by ψ, which has a data input rate of λ and requires R_c MB of memory and I_c CPU instructions (in millions) to process each input. The service also has a finite lifecycle defined as $\Delta_c = c_{end} - c_{start}$ where c_{start} and c_{end} denote the start and end clock times of the service respectively. We assume that for a specific service, Customers can estimate I_c, R_c, λ, and Δ_s. An example of such a service is the occupancy detection application that a number of transit authorities deploy [21, 33].

Resources The surplus RAM and R_s ram and CPU resources I_s of Suppliers are only available for limited duration defined as $\Delta_s = s_{end} - s_{start}$, subject to the demands of the primary application. We assume Suppliers can estimate I_s, R_s and Δ_s.

We note that spare CPU cycles and RAM are not the only resources that may be required to provide a service; network bandwidth, disk space, GPU cycles, etc. may also be required. However, these resources do not significantly change the problem of outsourcing online computation. As a result, we do not include them in our system model. In particular, we point out that the surplus compute



(a) A potential deployment of participants in the market.



(b) High level outline of solution approach

market we design is meant for urban communities that have already begun implementing smart city and IoT stream-computing applications. To support these applications, these cities must have a robust high-speed network, represented by the blue edges in Fig. 1a. Regardless of whether the actual computation involved with the concerned processes happen on the edge or in the cloud, such networks are crucial to facilitate data transfer. As a consequence, we make the reasonable assumption that the market has access to a network that is reliable. We primarily focus on computation aggregation and not communication constraints. We explain how the proposed Market reacts in case the the assumption of network reliability is violated in Section 3.5.

Offers We denote the sets of Customer and Supplier offers as O_c and O_s respectively. A Supplier offer, $os \in O_s$, is a tuple which includes: *account*, the account that posted the offer; I_s, the amount of surplus instructions (in millions) per second available; R_s, the amount of RAM available; s_{start}; s_{end}; and π_{xmin}, the minimum price the Supplier is willing to be paid per million instructions. A customer offer $oc \in O_c$ is a similar tuple which includes: *account*, I_c; R_c, the amount of RAM required; c_{start}; c_{end}; π_{xmax}, the maximum price the Customer is willing to pay per million instructions; λ, data input rate of the service; *name*, the name of the service; r, the number of Supplier replicas the Customer want to host the service.

A Supplier offer and a Customer offer can be matched to form an *allocation* Ω if:

$$I_c\lambda \leq I_s \text{ and } R_c \leq R_s \text{ and } \pi_{xmin} \leq \pi_{xmax} \tag{1}$$

$$[s_{start}, s_{end}] \cap [c_{start}, c_{end}] \tag{2}$$

In other words, a pair of offers is matchable if there exists a price that both Customer and Supplier would accept, the times overlap, and the Supplier has sufficient resources. We assume access to an algorithm that, given a set of Supplier and Customer offers, can find a match if one exists. This is a classic example of resource allocation problem for which there are many different algorithms [32].

Allocation An allocation is a service contract between the participants according to their offers. We define an allocation as a tuple which includes: *customer*, the allocated Customer; $\{suppliers\}$, a list of allocated Suppliers; a_{start}, the allocation start time; a_{end}, the allocation end time; *name*, the service name; π_x, the transaction price per million instructions π_x such that $\pi_{xmin} \leq \pi_x \leq \pi_{xmax}$; r, the number of Supplier replicas. For the transaction price π_x any value between π_{xmin} and π_{xmax} is feasible, it is determined by the Allocator based on its allocation mechanism (*e.g.*, double auction, fixed price, etc). Effectively, an allocation declares which offers were matched, the service price, the start and end times, the transaction price.

The duration of the allocation is defined as $\Delta = a_{end} - a_{start}$. Recall: I_c is the number of CPU instructions (in millions) to process each input, π_x is the transaction price per million instructions. For convenience we define π_s to be the price to process a service input: $\pi_s = I_c\pi_x$. Therefore, the total value of an allocation is $\pi_{total} = \pi_s\lambda\Delta$.

Processing A Supplier to correctly processing an allocation means that the Supplier is able to process the incoming data at least as fast as the incoming data rate λ, otherwise data would eventually be lost. Formally: $I_c\lambda \leq I_s$.

2.2 Additional Considerations

To enable trust and robustness in the market, we must consider the resource availability, trust, reliability, and time sensitivity problems posed by outsourcing stream computations, while keeping overhead costs low.

Resource Availability Consider that a Customer needs to run a service continuously from c_{start} to c_{end} in order to receive the expected benefit. It is likely that Suppliers are unable to host a service for its entire life-cycle due to the demands of the primary application. To address this the total Customer offer must be "chunked" into sub-offers that correspond with the offers by the Suppliers in the System.

Trust Given a particular market mechanism, each agent i (a Supplier or Consumer) has a utility function U_i, and a set of a set of actions A_i to choose from. We assume that the agents in the system are rational and choose to maximize their utility. For example, agent i chooses action $a^* \in A_i$ such that $a^* = \arg\max_{a \in A_i} U_i$. This means that Suppliers have an incentive to neglect processing service inputs, because of the electricity costs $\pi_{s\epsilon}$, if the benefits are greater than the consequences. We therefore face the challenge of designing a mechanism which needs to achieve two crucial goals: first, it should incentivize rational agents to be participate in the market, and second, it should incentivize truthful behavior. To aid this endeavor we specify a security deposit π_d the Customer and Suppliers need to provide in order to participate. π_d is computed as $\pi_d = \pi_s \times \rho$, where ρ is a penalty rate defined by the market. We show the how the desired market is constructed by considering game-theoretic models based on rational actors. We do not consider malicious agents which might have incentives that are external to

Table 1: Symbol Table

Smart Contract (SC)	
ρ	penalty rate set by the contract
π_v	cost of Supplier submitting outputs to the Verifier.
π_{cc}	cost of Customer committing outputs to the Verifier.
π_{mc}	cost of Mediator committing mediation results to the blockchain.
Mediator (M)	
π_m	payout to the Mediator for being *available* for the duration of the service
$\pi_{v\epsilon}$	the Mediator's electricity cost to verify ouputs.
Customer	
π_{xmax}	Amount the customer is willing to pay per million instructions
I	The number of instructions (in millions) required to process a service input
b	benefit Customer obtains from service output
π_{cg}	the customer's cost of generating an output.
e_c	$\frac{\pi_{cg}}{\pi_s}$; the customer's efficiency of processing vs the price paid to outsource.
λ	rate at which sensor data is generated, these are the inputs to be processed in an allocation.
r	the number of replicas requested by the Customer
s_c	the customer choosing to process an input
s_v	the customer choosing to verify an output
Supplier	
π_{xmin}	payment the Supplier requires per million instructions
$\pi_{s\epsilon}$	the cost to process a service input
π_v	the cost to send output hash to the Verifier
$P(s)$	The probability that the Supplier will process a particular input
e_s	$\frac{\pi_{cg}}{\pi_s}$; the customer's efficiency of processing vs the price paid to outsource.
Allocator	
π_a	payout to the Allocator for providing an accepted allocation
Δ	$a_{end} - a_{start}$; Duration of a service allocation
π_x	market price per million instructions between π_{xmin} and π_{xmax} determined by the Allocator.
π_s	$\pi_x \times I$; amount to be charged/paid to a Custompmer/Supplier for a processed input
n	the number of outputs that must be provided by the Customer for verification
λ_Δ	$\lambda \times \Delta$; the total number of inputs to be processed by each Supplier replica during an allocation.
π_{cd}	the customer security deposit for collateral prior to transaction. Set to $\rho\pi_s$
π_{sd}	the supplier security deposit for collateral prior to transaction. Set to $\rho\pi_s$

the market. We also require access to a verification mechanism that can detect cheating.

Reliability In any large distributed system failures are inevitable so platforms must be designed to account for them. The Customer would like to minimize the costs that it pays to have a service hosted. However, if correct results are not provided on time then it receives no benefit. As a result, the Customer may need to hire multiple Suppliers to ensure reliability. The number of suppliers needed for a specific job is exogenous to our market, and is typically a function of the specific job and the risk profile of the Customer. We assume that the time to failures can be approximated by an exponential distribution, a widely used model in reliability engineering [18]. Let the average rate of failure (per unit time) be denoted by q. The probability of failure can then be denoted by $p_f = 1 - e^{-qt}$. We assume that the probability of failure is common knowledge to all participating agents. Conditional on p_f, the Customer must hire r *replicas* to ensure $P(b)$ probability of success such that:

$$r = \frac{1 - P(b)}{p_f} \tag{3}$$

Verification and Challenge of Time As mentioned we require a verification mechanism, however, introducing verification may also introduce delays in the output stream. Thus, any verification that occurs must be essentially instantaneous or performed after the output has been released. Verification must also be inexpensive due to the volume of data processed. To this end we assume that the services are deterministic, meaning that if repeated processing of a specific output provides the same output every time. This assumption forms the basis for verifying Supplier outputs. Specifically verification compares outputs to detect errors, because this is fast. Naturally, the assumption also means that the market we design cannot be used for applications that involve random sampling, or for training machine learning models that can involve non-deterministic estimators. However, it can be used for inference using trained machine learning models.

We also require the stream applications to be stateless. Stateful services cannot be split across streams do not scale well, are not robust to failure since the state is lost, and introduce time delays to recover. The assumption of statelessness does not mean that we cannot handle state at all. If the state is stored external to the service then it can be read to perform the processing. Alternatively, if the service operates on windowed time-series data then it can be transformed into a data stream which merges the inputs into a single sample, thereby allowing the service to be stateless. Specifically, "non-re-entrant functions" with implicit states stored in the service itself are not supported.

Mediation Verification by comparing result, though fast, may not be able to determine which inputs where actually correct. For this reason we introduce the notion of mediation. Mediation is performed by a trusted agent which duplicates the contended output and compares it against the previous results. Mediators are assumed to be few, or have limited resources, and for this reason cannot be used to host all services.

In Fig. 1b we show a high level workflow of how the agents in our system interact to allow Customers to outsource their streaming computation service. First, the Customer and Suppliers send offers to an Allocator in the system using a distributed messaging and streaming platform. The Allocator solves the resource allocation problem and returns an allocation to the involved agents. Then, the Customer sends the service to the Suppliers, which start the service and begin processing inputs from the sensors. Periodically the Customer also provides processed outputs to a Verifier (we explain why this is crucial in our market design later). The Verifier compares the Customer and Supplier outputs and if they disagree the Mediator is requested to resolve the dispute.

3 OUR SOLUTION

Our solution consists of a communication fabric, supporting smart contract, market protocol, and modeling the protocol as a game with corresponding analysis.

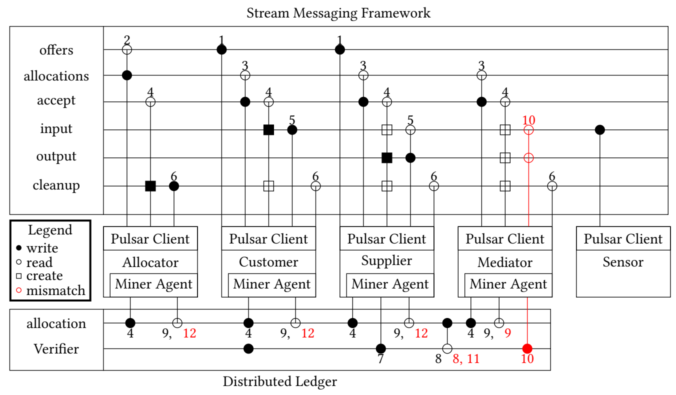


Figure 2: Detailed depiction of the solution approach introduced in Fig. 1b. The blue lines in Fig. 1b are replaced by the *Stream Messaging Framework*, while the green lines are replaced by the *Distributed Ledger*. Horizontal lines represent communication channels between participants through the messaging framework. Vertical lines represent functions that write to (filled circle) and/or read from (open circle) channels. For example, the Supplier reads that an allocation was accepted on the accept channel which causes it to create (denoted by a square) a reader on the input channel, a writer on the output channel and a reader on the cleanup channel. The functions occur in the numbered sequence. Red number only occur if the outputs checked by the Verifier do not match.

3.1 Enabling Market Capabilities

Allocators As discussed, offers are matched in our system to form allocations. We require some participant in the system to provide this service. To address this, we require some Suppliers in the system to implement an allocation algorithm to provide this service. We refer to such suppliers as *Allocators*. The set of Allocators is denoted A. Each Allocator in the system is free to implement an arbitrary (and correct) matching algorithm. We assume each Allocator is fair, *i.e.*, does not target offers to leave unmatched.

Blockchain Blockchain based distributed ledgers have been of interest in recent years for systems where there is no central trusted entity, instead trust is decentralized among the participants. However blockchain-only systems are inefficient, slow, and have limited throughput making them unsuitable for streaming applications which are time sensitive and long-running. For example, on Ethereum blocks are committed approximately once every 15 seconds[3] and adding a particular transaction can take anywhere from 2 to 24 blocks depending on the amount paid in transaction fees[4]. Stream Processing, on the other hand, needs to be comparatively faster. The challenge we face is to enable speed but preserve trust.

Messaging and Streaming platform With Blockchain To address this challenge we supplement the blockchain (box on the bottom of Section 3) with a distributed messaging and streaming platform (box on the top of Section 3). Participants communicate through both of these ledgers, and both ledgers record state.

To understand how the participants in the market communicate we reference Section 3. Communication happens using *channels*. A channel is a topic in a pub/sub system with authentication and authorization controlled by the stakeholders that communicate on that channel. For example, the offers channel is public so any stakeholder can read it, and any stakeholder can write their own offers[5]. In contrast the input channel is private and only the Customer and its sensors can write to it, while only the allocated Suppliers and Mediator can read it. Using a streaming platform we can efficiently communicate the bulk of information and deploy a smart contract on a blockchain to implement elements that are critical to preserving trust in the system such as tracking the state of allocations, providing verification, and transferring payments.

3.2 Protocol

The protocol is subdivided into stages of interactions. The sequence of events are denoted with the numbered labels in Section 3, which we use to assist in describing the market protocol. We refer to these events with circled numbers in the text (*e.g.*, ①). We refer to the state machine shown in Fig. 3 as we describe the protocol, which depicts how the smart contract tracks the state of each allocation. We also explain the participants and the cost structure we use. As a convention, text in `teletype` font represent smart contract functions and text that is *italicized* are state machine states.

Making Offers The protocol begins with the Customers and Suppliers constructing their offers as described in Section 2. These offers are then sent on the offers channel ①.

Creating Allocation The allocator then reads the offers channel ②. This causes it to execute a matching algorithm and construct an allocation, if one exists. It then sends the allocation on the allocations channel.

Accepting Allocation The Customers and Suppliers read the allocation channel ③ and send a message on the accept channel to specify if they accept the allocation or not. The participants read the accept channel ④, and if all the allocated participants accept, then the Allocator calls the blockchain smart contract function `createAllocation`. The state of the allocation is initiated to *Allocated*. Afterwards, for each Supplier for this allocation, the Allocator adds the Supplier and its offer hash (`AddSupplier`) to the network. When all Suppliers are added, the state of the allocation transitions to *Signing*. This function call incurs gas cost π_{ac}. Also in response to all of the allocated participants accepting, the other participants must construct the necessary channels for the service to operate.

[3]https://ethereum.org/en/developers/docs/blocks/
[4]https://www.ethgasstation.info/calculatorTxV.php
[5]they cannot modify the offers written by other stakeholders

In conjunction with this the participants read the blockchain for the *Signing* state change event from the smart contract. When it appears, the participants check the allocation to make sure that it matches the specific allocation that the Allocator sent on the allocation channel and, if it does, sign (*e.g.*, `customerSign`) the allocation on the blockchain by submitting their security deposits. This varies somewhat between participants. As part of signing the allocation the Customer commits to n test input/output pairs[6]. This causes the Customer to incur the cost to generate the hashed list π_{cg} and writing that hash to the smart contract π_{cc}. When all of the participants sign the contract, the state of the allocation is automatically changed to *Running*.

The Customer, Suppliers, and Mediator also incur a cost for signing the allocation and pay the Allocator for its service. These costs are lumped as π_a. Once all participants have signed, the Allocator receives π_a as payment for its service.

Service Execution After the channels are constructed the Supplier can begin reading ⑤ on the service input channel and sending outputs on the service output channel[7]. For each correctly processed input the Customer receives a benefit b and the Supplier incurs electricity cost $\pi_{s\epsilon}$.

Verifying Outputs At the end of the allocation, the Allocator sends a message on the cleanup channel ⑥ notifying the participants to end the allocation. The Customer informs the Supplier which n outputs are to be verified. The Supplier identifies the corresponding outputs and sends them to the Verifier on the blockchain ⑦, calling `PostOutput`. The Supplier incurs a cost of π_v performing this operation. The Verifier on the blockchain stores the output of the Supplier and compares it against the output provided by the Customer during allocation.

Mediation and Closing the Allocation If the outputs do not match then the Verifier emits a MediationRequested event (red ⑧) to the blockchain, where the Mediator reads it (red ⑨). The Mediator then reads ⑩ from the input and output channels and reprocesses the n inputs and compares its output against the Suppliers and Customer outputs to determine which participants are at fault. This incurs costs of $n(\pi_{s\epsilon}) + \pi_{cg}$. The Mediator then writes the result to the blockchain, calling `PostMediation`, which incurs a cost of π_{mc}. Then, the Verifier transfers payments and closes the allocation ⑪ by calling `ClearMarket`. The call causes the state of the allocation to transition to *Closed*, which is read (red ⑫) by the participants. If a Customer or Supplier output did not match the Mediator output they are fined π_{cd} or π_{sd} respectively. From these fines the Mediator is paid $n(\pi_s)$. If the outputs match, the Verifier transfers payments; the Supplier receives $\lambda_\Delta \pi_s$, the Customer pays $\lambda_\Delta \pi_s$, and the Mediator receives π_m for being available. The Verifier also closes the allocation (black ⑧), calling `ClearMarket` causing the state of the allocation to transition to *Closed*, which is read (black ⑨) by the participants.

Timeouts At each point in the protocol where a participant is waiting on the output from another (contract signing, service setup, service cleanup, service verification) we include timeouts in the implementation. If the output is not received within a specified time window the sender is considered failed, does not get paid, and is fined a portion of its deposit rather than the full deposit as it would have if it had submitted an incorrect result.

[6]We describe how this is done and why it is necessary in Section 3.4.5
[7]Note: this can start before signing is complete

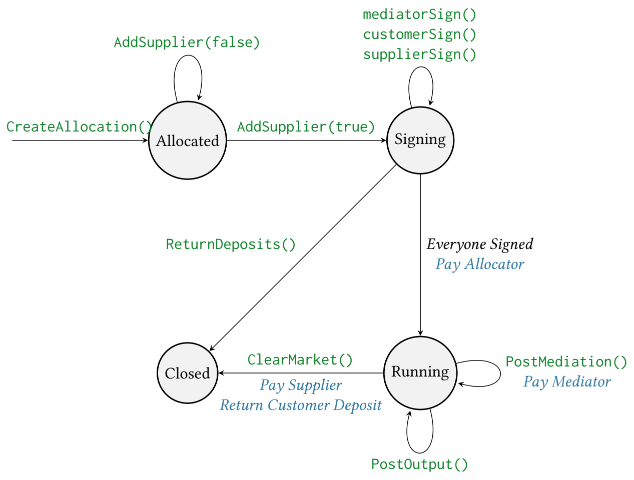


Figure 3: State of allocation on smart contract represented as a State Diagram.where are the functions that are used in this state diagram defined.

3.3 Allocation Duration

It may occur that a Customer needs to run a service for a duration that is longer than the availability of any of the Suppliers. In that case, the offer must be "chunked" into sub-offers with durations that correspond with the offers made by the Suppliers in the system. Some care must be taken when chunking since each offer that is allocated incurs monetary and time costs. Specifically, there is a time cost associated with constructing an allocation, denoted by δ_{alloc}, and with setting up the service after the allocation is signed, denoted by δ_{setup}. Allocation setup includes the Allocator writing the allocation to the allocation channel, allocated participants reading the allocation, writing to the accept channel, and optionally waiting for the allocation to be signed on the blockchain. Service setup includes transferring the service and starting the service. To be profitable and avoid thrashing, each service has a minimum service time δ_{min}. For an allocation to be valid $t_{end} > \delta_{min} + \delta_{alloc} + \delta_{setup}$ must be true. The time at which this becomes false is the expiry time is $t_e = t_{end} - \delta_{min} - \delta_{setup} - \delta_{alloc}$.

The Customer can read the active Supplier offers to identify a reasonable service duration, greater than the minimum service time, and construct offers using this value. As part of the allocation algorithm, the Allocator checks if the Customer offer has expired. If the offer has not expired, it checks to see how many replicas the Customer has requested. It then attempts to find that many Suppliers that are available for the duration. If the Allocator can not satisfy an offer it produces a void allocation which serves to notify the Customer that the specific offer is not allocated.

3.4 Mitigating Cheating, Collusion and Failures

To analyse the strategic interaction between the Customers and Suppliers, we consider our protocol as a game. We derive the utility functions for the agents by using the cost structure that the protocol imposes. Before we discuss the game we explain why the protocol enforces the following: (1) have the Customer check the Supplier outputs, (2) why the Customer must pay even if the Supplies do not work, (3) implement the Verifier on the blockchain, (4) include a Mediator, and (5) have the Customer commit to n outputs during the allocation,

3.4.1 Customer Verifying Supplier Outputs. The Customer checks the outputs generated by the Supplier in order to prevent collusion among the Suppliers. Consider a situation where the check is not enforced. In that case, the Suppliers can collude and agree on a common output such that their outputs match. Recall that the Verifier only matches the output, so a common (albeit incorrect) set of outputs would be accepted. Such behaviour is possible, since presumably, colluding can be significantly cheaper in practice than running the service. One way to prevent such behavior is to ensure the existence of an additional Supplier that does not participate in collusion. Such an idea is motivated by the role of "trusted agents" in multi-agent systems [30]. There are two relatively straightforward ways to achieve this. First, the Customer itself can occasionally act as a Supplier, and second, the Customer can occasionally hire a Supplier that it trusts to process an input. The presence of such a Supplier does not negate the benefits of the outsourcing market because the service workload can (and usually will) exceed the trusted resources available to the Customer. Since the Supplier in consideration is trusted by the Customer it does not collude with the other Suppliers. In such a scenario, the Verifier detects the collusion since the outputs agreed upon by the colluding Suppliers do not match with that of the trusted Supplier.

3.4.2 Customer pays regardless. An additional small but key design decision to eliminate undesired behavior is to have the Customer pay π_s at the end of the allocation regardless of the outcome. This decision avoids complications that appear in prior work [14]. If the Customer is refunded when a job fails, it naturally incentivizes Customers to construct jobs that can manipulate the system (since Customers can get work done for free by collecting the refund), or collude. We understand that this design choice may seem unfair — there is a possibility that the Customer pays for a service that is not delivered. However, in practice, we merely shift the overall cost incurred by the Customer. This shift is due to the fact that if Customers had the incentive to cheat, the system would have incurred costs to build infrastructure to regulate such behavior. Naturally, such cost would have been borne by all agents, including the Customer. Also, we explain shortly how this payment does not surface in the optimal strategy profiles for the agents.

3.4.3 Blockchain Implementation of the Verifier. To prevent the Verifier from being able to collude we implement it on the blockchain. The Verifier is only capable of detecting errors, not ascertaining which entities are at fault.

3.4.4 Inclusion of a Mediator. Including a Mediator to reprocess inputs allows us to only penalize participants who are at fault. Participants allowlist Mediators and include this information in their offers for the Allocator to construct valid allocations. If a participant chooses to no longer trust a particular Mediator then it no longer includes it in its offers.

Table 2: The utility of the Customer and Supplier given the 4 combinations of their pure strategies.

Supplier Customer	s_c	$\overline{s_c}$
s	$\boldsymbol{\pi_s - \pi_{s\epsilon} - \pi_v - \pi_a^*}$ $b - \pi_s - \pi_{s\epsilon} - \pi_v - \pi_a$	$\pi_s - \pi_{s\epsilon} - \pi_v - \pi_a$ $\boldsymbol{b - \pi_s - \pi_a^*}$
$\bar{s}$	$-\pi_v - \pi_d - \pi_a$ $\boldsymbol{b - \pi_s - \pi_{s\epsilon} - \pi_v - \pi_a^*}$	$\boldsymbol{\pi_s - \pi_v - \pi_a^*}$ $-\pi_a - \pi_s$

3.4.5 Verifying n outputs. We now explain the need for the Customer to commit to checking n inputs. Consider the situation where the Customer can check the Supplier outputs, but can choose not to.

We model the interaction between the Supplier(s) and the Customer as a game. The Supplier's actions are to either process an input (denoted by s) or not (denoted by $\bar{s}$). Similarly, the Customer can choose to process the input (s_c) or not ($\overline{s_c}$) (note that the Customer can hire a trusted supplier to do the processing or do it by itself; this choice is orthogonal to the strategic interaction we consider). The Customer pays π_a to accept the allocation and π_s for the Supplier to process the input. Recall that the Customer pays this regardless of whether the Supplier provides the output. The Customer receives benefit b when it receives the processed output, and pays $\pi_{s\epsilon}$ (the electricity cost of processing the input as defined in section 3.2) to process the input itself if it decided to check.

The Supplier also pays π_a to accept an allocation and π_v to have its output verified. To process an input the Verifier incurs $\pi_{s\epsilon}$ in electrical costs. If the outputs differ, the Verifier records this to the blockchain, which triggers mediation to determine which Customers and Suppliers to penalize. The Mediator posts the outcome back to the blockchain, finalizing the allocation and. triggering payments. If a Customer is at fault it is fined π_{cd}. If a Supplier is at fault it is fined π_{sd}. We show the game matrix for this scenario in Table 2.

THEOREM 1. *There is no pure strategy Nash equilibrium in the game between the Supplier and the Customer*

PROOF. Assume the Customer chooses to process all inputs (say, through a trusted Supplier), then: (1) the Supplier's utility for (s, s_c) is higher than for $(\bar{s}, s_c)$, and as a result, the Supplier's optimal strategy is to truthfully process all inputs. (2) However, the Customer then has a profitable deviation by changing its strategy to process no inputs. (3) If the Customer processes no inputs, then the Supplier also has a profitable deviation and processes no inputs, causing the Customer to deviate and process all inputs as we began. □

Notice, however, that if the the Customer makes a credible threat that it checks some of the inputs (n), then the game changes such that processing all inputs is the Supplier's dominant strategy. Naturally, the choice of n depends on a specific game and the payment structure. However, for the Supplier to be convinced the Customer's threat is credible, the Customer must have no profitable deviations from processing n inputs. We identify two ways to accomplish this. We describe one of the approaches here (the one that we actually implement as part of our experiments), and discuss the other approach in the online appendix.

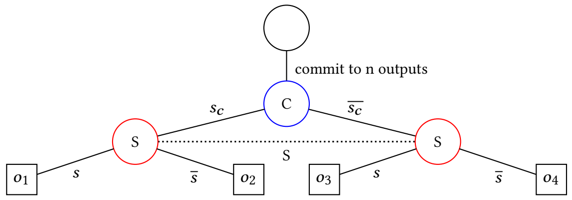


Figure 4: Extensive-form game produced by the protocol. Blue nodes indicate Customer moves, red nodes indicate Supplier moves. The game is sequential, but the decisions are hidden, so we treat it as a simultaneous move game. Each outcome has payouts for the agents, which are found in Table 3

Prior to the allocation, the Customer processes n inputs. Then, as part of accepting the allocation it hashes those n outputs and commits the hash of that hash to the blockchain. It then mixes those n inputs in with the regular workload inputs [8]. At the end of the allocation the Customer notifies the Supplier which outputs it must submit. If the Customer sent a simple list of outputs the Supplier could neglect to process them during the allocation and process them post hoc. Instead the Customer hashes its outputs and sends them to the Verifier, as we described in Section 3.2, where at the end of the allocation they are compared against the Supplier's output hash. If the Customer hash did not represent the correct number of correct outputs, then it is detected and the Customer is penalized. Thus, there is no profitable deviation from processing n inputs for the Customer.

The game that is derived from this protocol can be seen in Fig. 4 with payouts in Table 3. In this game, (s, s_c) is the dominant strategy for the Customer as long as the utility for $U(s, s_c) > U(s, \overline{s_c})$, which is true when $n\pi_{cg} < \pi_{cd}$. Setting variables as in Eq. (4) the Customer will always process n inputs as long as $n < \rho$.

$$\begin{aligned} n\pi_{cg} &< \pi_{cd} \\ \text{note: } \pi_{cg} = e_c\pi_s,\ e_c < 1, \text{ set } \pi_{cd} &= \rho\pi_s \\ ne_c\pi_s &< \rho\pi_s \\ ne_c &< n < \rho \end{aligned} \tag{4}$$

Similarly (s, s_c) is the dominant strategy for the Supplier as long as the utility for $U(s, s_c) > U(\overline{s}, s_c)$ which is true when $\lambda_\Delta(\pi_s - \pi_{s\epsilon}) > -\pi_{sd}$. This is true unless the Customer severely underestimates the resources required, in which case the Supplier's output states that the resources allocated were exceeded, or the Supplier underestimated its power consumption in its initial offer. Note that if the Customer sends too few, or bad outputs to the Supplier, the Supplier can call for Mediation.

This formulation presumes that the Supplier processed every input or none of them. However, it is possible that since the Customer is only checking n inputs the Supplier can risk skipping processing of some inputs to reduce its electricity costs. The utility for the Supplier then depends on if it processed all the inputs whose outputs would be checked. The utility of the Supplier in that case is:

$$\begin{aligned} U[S] =& P(s)^n(\lambda_\Delta\pi_s - \lambda_\Delta\pi_{s\epsilon}P(s)) + \\ & (1 - P(s)^n)(-\pi_{sd}) - \pi_v - \pi_a \end{aligned} \tag{5}$$

The Supplier can then solve for $P(s)$ to maximize its utility. Therefore, the Allocator must set n and π_{sd} such that P(s) is maximized.

$$\begin{aligned} \frac{\partial U[S]}{\partial P(s)} &= nP(s)^{n-1}\lambda_\Delta\pi_s - (n+1)\lambda_\Delta\pi_{s\epsilon}P(s)^n + nP(s)^{n-1}\pi_{sd} \\ &\text{<set equal to 0, solve for P(s)>} \\ P(s) &= \frac{n(\lambda_\Delta\pi_s + \pi_{sd})}{(n+1)\lambda_\Delta\pi_{s\epsilon}} \\ &<\pi_{sd} = \rho\pi_s,\ \pi_{s\epsilon} = e_s\pi_s, \text{ simplify}> \\ P(s) &= \frac{1}{e_s}\left(\frac{n}{n+1} + \frac{n\rho}{\lambda_\Delta(n+1)}\right) \\ &<\text{set } \rho = \frac{\lambda_\Delta}{n}> \\ P(s) &= \frac{1}{e_s} \end{aligned} \tag{6}$$

In Eq. (6) we see that by requiring the penalty rate ρ equal to $\frac{\lambda_\Delta}{n}$ the maximum utility of the Supplier occurs when $P(s) > 1$ (since $e < 1$) which is not possible. This means this requirement ensures that the Supplier will always process the inputs. ρ is a parameter that is set at the system level on the blockchain, while λ_Δ is specified in the Customer offer and n is computed by the Allocator.

3.4.6 *Keeping threats credible.* While describing the protocol, we make the following three statements: 1) the Customer "commits commits to n test input/output pairs", 2) "The Customer informs the Supplier which n outputs are to be verified", and 3) "The Supplier identifies the corresponding outputs and sends them to the Verifier on the blockchain.". If these outputs are sent in the clear our game breaks, or devolves back to a mixed strategy Nash equilibrium. For example, if the Customer sends its n test outputs raw to the blockchain, the Supplier can read them and copy those outputs. Then, it can simply provide those outputs at the end of allocation and not neglect processing of any inputs. For this reason we need to make sure that the outputs that are being shared in a way that does not break the game. To do this we have the Customer use a hash function to mask the output it sends to the blockchain. Then since the Verifier is comparing outputs the Supplier's output must also be hashed so the results can be compared. To prevent the Supplier from simply copying the Customer's hash it must provide its outputs so that the Verifier itself can hash them to match the Customer outputs.

To do this we first introduce some notation. Let $O_c = \{o_1, o_2, \ldots o_n\}$ represent the set of the Customers n committed outputs where o_i is a particular Customer output. Let $O_s = \left\{o_{s1}, o_{s2}, \ldots o_{s\lambda_\Delta}\right\}$ represent the set of λ_Δ outputs produced by the Supplier, where o_{si} is a particular Supplier output. Then to mask the outputs we use the following hash functions:

$$\begin{aligned} \alpha(K) &= \{hash(k_i) : \forall k \in K\} \\ \gamma(K) &= hash(K) \\ \Gamma(K) &= hash(hash(K)) \end{aligned} \tag{7}$$

[8] This assumes that the Supplier cannot distinguish test inputs from workload inputs

	o_1	o_2	o_3	o_4
	ss_c	$\bar{s}s_c$	$s\overline{s_c}$	$\overline{ss_c}$
Customer	$\lambda_\Delta(b - \pi_s) - n\pi_{cg} - \pi_{cc} - \pi_a$	$-\lambda_\Delta\pi_s - n\pi_{cg} - \pi_{cc} - \pi_a$	$\lambda_\Delta(b - \pi_s) - \pi_{cc} - \pi_{cd} - \pi_a$	$-\lambda_\Delta\pi_s - \pi_{cc} - \pi_{cd} - \pi_a$
Supplier	$\lambda_\Delta(\pi_s - \pi_{s\epsilon}) - \pi_v - \pi_a$	$-\pi_v - \pi_{sd} - \pi_a$	$\lambda_\Delta(\pi_s - \pi_{s\epsilon}) - \pi_v - \pi_a$	$-\pi_v - \pi_{sd} - \pi_a$
Allocator	π_a	π_a	π_a	π_a
Mediator	π_m	$\pi_m - n(\pi_{s\epsilon} - \pi_{v\epsilon} + \pi_s) - \pi_{mc}$	$\pi_m - n(\pi_{s\epsilon} - \pi_{v\epsilon} + \pi_s) - \pi_{mc}$	$\pi_m - n(\pi_{s\epsilon} - \pi_{v\epsilon} + \pi_s) - \pi_{mc}$

Table 3: Game outcomes and payments in Fig. 4. For example, o_2 is the outcome when the Supplier does not process all the validation inputs correctly and the customer does provide sufficient validation inputs.

For the hash function we need to use a hash function that is supported by the chosen blockchain implementation - for example, since we use Ethereum we can use keccak256, sha256 and ripemd160[9].

When the Customer sends its outputs to the Supplier, so the Supplier can identify which outputs it needs to send to the Verifier, the Customer sends $\alpha(O_c)$. Then to determine which inputs are the test inputs the Supplier hashes all of its outputs and find which hashes are common to the two sets. We call that subset of Supplier outputs O_v where $O_v = \{o_i : hash(o_i) \in \alpha(O_c) \cap \alpha(O_s)\}$. This is because if the Customer sent $\{o_i : o_i \in O_c\}$ then the Supplier could skip processing the inputs and simply use the outputs provided to produce O_v. Or if the Customer sent the the index i of each test output the Supplier could neglect processing the inputs until it received the indices to produce O_v.

When the Customer commits O_c to the blockchain it sends it as $\Gamma(O_c)$. The output set is double hashed because if the Customer sent $\gamma(O_c)$ then the Verifier on the blockchain would have to hash all of the Supplier's n outputs which is expensive. Instead the Customer sends $\Gamma(O_c)$, requiring the Supplier to send $\gamma(O_v)$ to the Verifier. Then the Verifier only has to hash a single element to compare against the Customer's output hash: $\Gamma(O_c) == hash(\gamma(O_v))$.

3.5 Other faults

If the assumption that the network is robust and high-speed is violated such that participants lose connection a timeout is triggered which either triggers Mediation, if the Mediator is still connected, or is interpreted as a Mediator failure.

After all participants write to the accept channel, then:

(1) If any participant that does not sign or request `ReturnDeposits` by the deadline is penalized.
(2) If all participants that respond by the deadline do not agree and request `ReturnDeposits`, the Allocator is penalized
(3) If some participants do not agree and request `ReturnDeposits`, they are penalized an amount less than the timeout penalty. The deposit of participants who have signed the allocation is refunded, and the allocation is *Closed.*
(4) If the Customer 1) signed, 2) specified a minimum Supplier threshold r_{min} in addition to the desired number of replicas, and 3) r_{min} replicas signed, then the allocation can proceed.

[9]https://solidity.readthedocs.io/en/v0.7.4/cheatsheet.html

4 IMPLEMENTATION

Since interacting with a public blockchain like Ethereum is slow (minimum of 15 seconds to record a transaction) and costly ($0.80 base fee with fastest transaction) we minimize the number of interactions with it. While using side-chains is an option, implementations of side-chains are still relatively new and immature [28]. Instead, we choose to use Apache Pulsar, which was developed for robust stream computing applications. These characteristics have been demonstrated in industrial applications [4]. Pulsar is a distributed messaging framework and ledger which supports multiple clusters and multi-tenancy [5]. A tenant can be likened to a user in an operating system. Users operate on processes and files, while tenants operate on topics and clusters of hardware resources. By representing a stakeholder as a tenant the stakeholder is able to manage which other stakeholders have access to its clusters while isolating cluster hardware. [10] Thus clusters and tenants make Pulsar a viable options for implementing and supporting a multi-stakeholder market. [11] [12]

To describe our implementation we refer to Table 4. Each stakeholder is represented by a tenant $(t_1, t_2, ..., t_n)$ and their compute resources are placed into clusters $(c_1, c_2, ..., c_n)$. To participate in the Market, the stakeholder must first add its clusters to the Pulsar instance. Then joining the Market is done by by giving the *marketplace* tenant read-access to its *public* namespace, which is on the stakeholders resource cluster [3]. The set of clusters that allow the *marketplace* tenant constitute the marketplace. Then, if the new stakeholder writes to the *market/public/offers* topic the other stakeholders are able to read them. Likewise the new stakeholder is able to read the offers and allocations of other stakeholders. Sharing of topics and by implication messages is handled with the Pulsar administrative mechanisms. All stakeholders are implicitly granted read access to the marketplace tenant.

4.1 Implementation Resilience Discussion

The market is robust since many stakeholder are present and maintain their own clusters. A failure for the blockchain based distributed ledger means that the network has been compromised where adversarial stakeholders control write access. On a proof-of-work blockchain this means they control more than 50% of the

[10]Pulsar tenants are applied to machine clusters. These clusters contain and isolate the resources owned by the market participants.

[11]This is not the only way to implement the multi-stakeholder functionality we desire, but it is sufficient for our purposes.

[12]There are some weaknesses in the current Pulsar implementation. The management of tenants is not itself distributed.

			Tenants $(t_1, t_2, \cdots, t_n)$			
			Customer	Supplier	Allocator	Mediator
Cluster $(c_1, c_2, \cdots, c_n)$	Customer	MB	rw	r	r	r
		SB	rw	r	-	r
	Supplier	MB	r	rw	r	r
		SB	-	rw	-	r
	Allocator	MB	r	r	rw	r
	Mediator	MB	r	r	r	rw
		SB	r	r	r	rw

MB	Marketplace Broker
SB	Service Broker

Table 4: Tenants represent stakeholders and can read and write to their own cluster of hardware. To participate in the market they allow read access to the *Marketplace broker* that they host on their cluster.

Function Name	Approx. Gas Cost	Approx USD ($) value
Constructor	4054778	33.43
Setup	227554	1.88
CreateAllocation	369506	3.05
AddSupplier	181139	1.49
MediatorSign	101853	0.84
CustomerSign	137281	1.13
SupplierSign	188145	1.55
PostOutput	130108	1.07
ClearMarket	157458	1.30

Table 5: Gas Costs and USD value of each smart contract function call. Assuming 20 Gwei per 1 unit of gas.

mining power, thus their security is based on having a large number of miners in the network. This failure would cause the market function to cease. The failure of other participants does not cause the market to fail, but may interrupt some allocations. If the Allocator loses connection, any allocations that have been recorded to the blockchain are still valid, and allocations that have not been recorded are void. If a Customer loses connection it may be penalized if it does not sign an allocation or if it does not send the Suppliers the outputs hashes the Supplier needs to determine which outputs to post to the blockchain. If a Supplier loses connection the smart contract will timeout and the Supplier will be considered failed. If Mediation is required, but the Mediator has lost its connection, then the fall back behaviour of the Verifier is, after waiting some timeout duration, fine all Suppliers and the Customer for not providing matching outputs. This does not impact the nature of the game, it only makes it possible for cheating to hurt others, it does not improve one's own utility.

Essentially, our market design ensures that consequences of failures are localized to the specific stakeholder that experiences the failure. In the proposed system, as long as there is more that one stakeholder that sends offers to an Allocator and stakeholders have a mutually trusted Mediator, the Market is considered to be operational. Pulsar can detect these failures and can automatically fail-over to continue operating if the stakeholder has configured its pulsar cluster appropriately.

5 EXPERIMENTS AND RESULTS

5.1 Blockchain Analysis

Gas measures the amount of computation required to execute a transaction or make a smart contract function call in an Ethereum network. The total gas cost of a transaction depends on *transaction costs* and *execution costs*. While the transaction costs counts for the amount of data written on the blockchain, execution costs count for CPU operations required. The exact code that is executed while making a function call to the smart contract can vary based on many factors. For example, the number of Suppliers affects the number of CPU operations required for making a `ClearMarket` function call since the smart contract must compute the hash of the output hash for each Supplier and compare it to the Customer's committed hash of output hash.

Further, the agent (*i.e.*, Supplier, etc.) making calls to the smart contract decides on the price of gas based on how busy the Ethereum

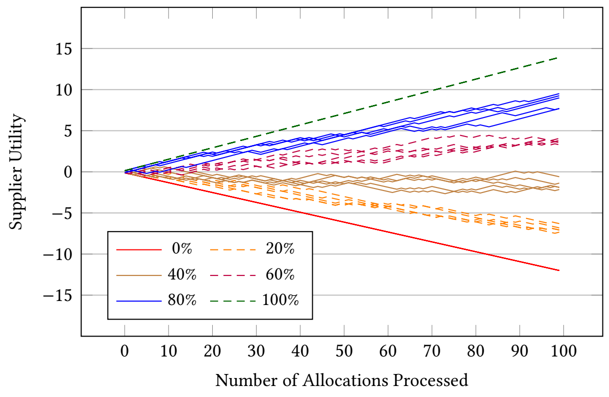


Figure 5: Cumulative supplier utility compared to number of allocations completed by that supplier. Suppliers are broken into six groups representing the probability of the supplier successfully processing a given job $P(s)$ (0% in the figure represents cheating 100% of the time). Each group consists of 5 suppliers. As shown, processing all allocations maximizes profit over time.

network is. Generally, more price on gas means that the transaction will be executed faster on the network. Table 5 shows the approximate gas cost required for making each function call to the smart contract based on sum of their *transaction costs* and *execution costs*, and their equivalent of USD price given a typical gas cost of 20 Gwei.

In [14] the gas cost for an nominal execution of the platform for the equivalent of the Customer was 592, 000, the cost to the Customer in this work is the cost of signing the allocation which is about 4 times less at 137, 281. re-emphasize why this is possible.

5.2 Strategy Experiments

The goal of this experiment was to validate the the conclusions determined from the analytical analysis of the game by empirically checking that correct behaviour resulted in maximum utility. To perform this we chose a set of parameters to compute values for the payments. These can be seen in Section 5.2. We set up the Supplier and the Customer so that we could configure their strategies. Specifically, the Suppliers were given a parameter representing

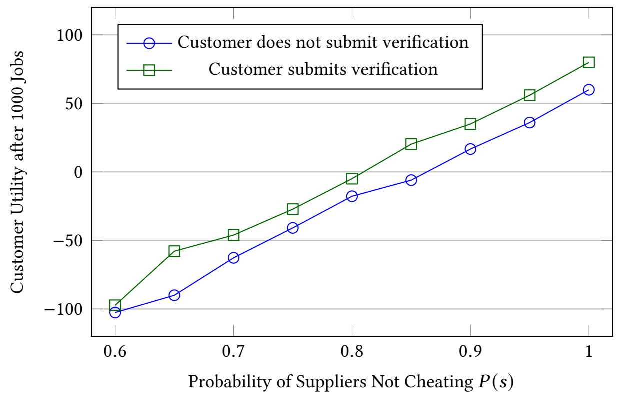


Figure 6: Customer utility after 1000 jobs compared to probability of suppliers in the market processing a given job ($P(s)$). The customer gets the maximum benefit when submitting the required number of verification values. As shown, there is a linear relationship between the global $P(s)$ of the suppliers and customer utility.

Table 6: Test Parameters. For the device power consumption rate we used the specification of a Beagle Bone Black single board computer[11]. To convert gas to dollars we used [1]. Electricity cost per kWh was acquired from [15]. Tie back to table 3

Parameter	Test Value	Description
λ	20	frames per second
Δ	10	minutes
λ_Δ	12000	$\lambda \times \Delta$
n	12	.1% of λ_Δ
ρ	1000	Eq. (6)
$gasprice$	20 Gwei	price per unit of gas
π_{cc}	1.1257	$137,281$ gas at $gasprice$ to dollars
π_v	1.0669	$130,108$ gas at $gasprice$ to dollars
E	13.2	cents per kWh
TDP	4.6e-4	Thermal Design Power (kW) at max
TDP_{min}	2.1e-4	Thermal Design Power (kW) at min
$\pi_{a\epsilon}$	0.001	$E \times TDP \times \Delta$
π_a	3.02994	$\pi_{a\epsilon} + 369,506$ gas at $gasprice$ to dollars
π_{cg}	7e-8	$(E \times TDP \times n)/\lambda_\Delta$
π_{se}	7e-8	$(E \times TDP)/\lambda_\Delta$
π_s	2.0e-5	$b > \pi_s > \pi_{se}$
b	3.5e-5	Table 3
π_{cd}	0.02	$\pi_s \times \rho$
π_{sd}	0.02	$\pi_s \times \rho$
π_m	3.3e-5	$E \times TDP_{min} \times \Delta$

the probability that the Supplier would properly process all the service inputs (i.e. did not cheat), $P(s)$. The Suppliers were grouped by $P(s)$ into 5 groups with $P(s)$ set to 0, 20%, 40%, 60%, 80% and 100% respectively. Similarly, the Customers were given a parameter representing the probability that the Customer would supply the required outputs $P(c)$. For the 30 Customers we similarly broke them into five groups with $p(c)$ set to 0, 20%, 40%, 60%, 80% and 100% respectively.

We ran the experiment with 30 Suppliers and 30 Customers. Each of the Customers and Suppliers sent 100 offers on the offers channel, where they were read by the Allocator and matched. The Allocator wrote the resulting allocation to the allocation channel. Once the allocation was sent the participants executed their strategies by writing to the output channel what their strategy was. This was read by a a script that was emulating the Verifier and Mediator. Based on the strategies taken by the Supplier and Customers, the Verifier/Mediator would calculate the corresponding payments according to the game outlined in Section 3.4 and write to a channel representing the blockchain which the Customers and Suppliers would read to update their balance.

As shown in Section 5.2, the Supplier utility was maximized when the Suppliers processed all jobs and did not cheat. The utility was least when the Suppliers cheated every time and improved as the probability of not cheating increased. There was minimal variance between Suppliers with the same $P(s)$, showing that the behavior of the Customers had a negligible impact on the Supplier's utility. Therefore, it is always in the Supplier's best interest to process each job.

The Customer Utility is dependent on the behavior of the Suppliers, so to investigate the impacts of Supplier behavior on Customers, we ran nine simulations. Each of these simulations involved a set of Suppliers with a fixed $P(s)$. The first simulation therefore had all Suppliers with a $P(s) = 60\%$ and the last simulation had a $P(s) = 100\%$. For each of the simulations we recorded the final utility of an ARTA Customer that submitted n outputs for verification and an ARTA Customer that did not submit n outputs for verification.

As shown in **??**, it is always in the best interest of the Customer to submit the required n outputs for verification regardless of the system level $P(s)$. In the case of a perfect system with $P(s) = 100\%$, there was a 24% greater utility for submitting outputs for verification. Additionally, there is a clear linear relationship between the $P(s)$ of the Suppliers in the system and utility for the Customer. For the ARTA application as formulated **??**, the Customer had a break-even point when Suppliers in the system had a $P(s)$ of 82%. emphasize why this result show that we did something good and why that good happened. what did we do that made that possible

5.3 End to End Execution Experiments

- end to end message exchange between Customer and Supplier and other nodes. Once it gets the allocation - it subscribes to the input channel and start writing to the output channel. - number of messages - timing

5.4 Mediation Experiments

if we have the behaviors - goes to the Supplier - runs the process for some time. The behavior says what the Supplier already did - send those verifier compares them - it sends a message to mediator - mediator - spins and sends the output

- timing - number of messages

6 CONCLUSION AND FUTURE WORK

In this paper we developed an environment that can be used to implement decentralized market mechanisms. Using this environment we designed a market for outsourcing online computations. This market solution can provide the substrate for multi-stakeholder applications that rely on a dispersed computing paradigm, of which ARTA and multi-modal traffic monitoring are examples. This architecture is modular, this means that other the off-chain functionality of other market solutions, such as TrueBit, can be reproduced and integrated with their on-chain interpreter. This architecture allows participants to make trade-offs between speed and trust. By this we mean that as soon as an allocation has been accepted the participants may begin speculative working in order work quickly. However, it is possible that this work will wasted because that allocation is never added to the blockchain. Alternatively, participants may be cautious and avoid beginning the service until the allocation has been recorded in the blockchain.

We demonstrated that rational participants will follow the protocol and benefit from participating in the system, while participants that deviate from the protocol incur fines. We do not prevent agents from operating maliciously and returning erroneous results, however, we do make it so that the costs exceed the benefits within the system. We do not handle scenarios when there are benefits that are external to the system that make it worthwhile to misbehave. Considering these scenarios is part of our future work.

Since there be many Allocators in the system, each with their own allocation algorithm, effectively forming a multitude of Markets within the system. Exploring how these various markets interact is also an interesting avenue for future work.

Appendices

A PROOFS

B SMART CONTRACT

C BACKGROUND

Durable communication is a critical aspect in a system that requires service reliability when that service is excuted on unreliable nodes. Durability is a key feature of named data networking (NDN), where in theory, all data is immutable and can be accessed by simply providing its name. This means there are no lost messages, they can always be recovered. Point-to-point and publish/subscribe architectures are sub-classes of NDN and can be implemented using them however they have limitations in this context because of the dynamic nature of the system.

C.1 Auctions and Resource Allocation

need to set fair prices - use a combinatorial double auction, probably include a citation to more recent work (I would suggest using that but I don’t understand how it works and we have no information on what user preferences might actually be, so probably draw them from a distribution of some kind. http://www.nuffield.ox.ac.uk/users/klemperer/demandtypes.pdf)

Most other works in the area of fog and cloud computing focus on the efficient resource allocation. In a combinatorial market this must also be considered. Though the objective is different the problem is still np-hard - generally heuristics are used. (Find some sensible one, don’t spend too much time on it.)

C.2 Streaming Applications and Verification

C.3 Market

continuous or interval based double auction

C.4 Computation Outsourcing

C.5 Verification

There are several options for verifying the results of outsourced computation.

C.5.1 Repeated Executions. On approach is the once we utilize here which is repeating computation. This approach relies on the assumption that the computation is deterministic. It also assumes that failures are uncorrelated in the execution. An example of a correlated failure is when a server rack fails and all the disks on that rack also fail, if there were replicas on that rack they did not contribute to the reliability of the system because of the common failure mode. Another shortcoming is that the cost of repeated executions scales with the number of executions, and therefore must be limited in order to keep outsourced computing costs viable.

A way to reduce the overhead of repeating computations is to mix “dummy” computations (with already-known results) amongst the jobs that need to be done [12, 17, 25, 27]. These cited works refer to these tasks with known results as “ringers”, “spot-checks”, “quizzes”, and “chaff”. By combining simple, easy-to-verify computations with the real computations, it is possible for the verifier to only verify the easily verified result. The assumption is that if the easily verified result is correct, the real computation is also correct. This has the added benefit of enabling non-deterministic computation. The challenge in this scenario is ensuring that the supplier is unable to detect the easily verified computation, otherwise it would simply perform that and receive compensation. One domain that may offer guidance for how to accomplish this is steganography, where the messages are concealed within the non-secret data. These works also discuss the use of *reputation* to make the technique more effective as contractors survive multiple rounds of dummy tasks. These works do not evaluate the case where the requester is malicious; some consider the possibility of collusion, but not all. None of these works consider the use of financial incentives to influence the contractors.

Another way to reduce costs is duplicate only a subset of the computations. For stateless computations this is easy since any particular computation can be repeated without requiring state from the previous computations. The downside of both dummy tasks and checking subsets only provide a probabilistic assurance that a given result is correct, since any result that is unchecked we assume is correct.

C.5.2 Verifiable Computation. The costs of verifying a result should be less than the costs of executing a computation; otherwise we would just execute the computation. There are many computational problems wherein this property holds true (for example all problems in the complexity class of *NP*). For problems that do not have this property proof-based verifiable computation is the ideal method of verification, but it is not yet practical[13] for general-purpose computation. In a survey by Walfish and Blumberg [31], they note that the overhead for a prover is several orders of magnitude too high. Meanwhile, the overhead for the general-purpose verifiers requires the execution of thousands of instances of the computations to amortize setup costs.

C.5.3 *Trusted Execution Environments.*

Trusted Execution Environments like the Intel Management Engine and AMD Secure Processor are also possibilities, however this would introduce concerns about security vulnerabilities and whether they provide a backdoor. [14].

C.6 Estimating Resource Requirements

Resource utilization depends on many parameters and so, on general purpose compute hardware, it is impossible to predict the exact amount of resources required. Despite this there are many works that strive to predict the resource requirements. In this work we assume that the customer is able to provide

One way to implement this is for the customer to benchmark the service many times and provide the mean and variance of the measured resource requirements since on given hardware

running the same computation on the same system will require similar amounts of resources

system performance can be compared using benchmarks

- 0 Dynamic resource provisioning for workflow scheduling under uncertainty in edge computing environment
- 0 Estimating Computational Requirements in Multi-Threaded Applications
- 0 Scientific benchmarking of parallel computing systems: twelve ways to tell the masses when reporting performance results
- 0 Boinc Whetstone benchmark
- 0 BHive: A Benchmark Suite and Measurement Framework for Validating x86-64 Basic Block Performance Models
- 0 Compiler Auto-Vectorization with Imitation Learning

C.7 Messaging Framework

For the messaging framework we compare several of the existing technologies to determine which are most suited to supporting a market for online computation.

Scott: what is the distinction between sync and communicate?

Scott: still use Ethereum for the market? If so, make it clear this is not the public network thus it can run faster and cost less, maybe there are no mining fees, but the miners get the leftovers from the double auction. Or consider alternatives (pulsar function that is replicated among stakeholders)

REFERENCES

[1] [n.d.]. ETH Gas Station. https://ethgasstation.info/calculatorTxV.php. Accessed: 2020-10-25.

[2] David P. Anderson. 2020. BOINC: A Platform for Volunteer Computing. *Journal of Grid Computing* 18, 1 (2020), 99–122. https://doi.org/10.1007/s10723-019-09497-9

[3] Apache. [n.d.]. Authentication and authorization in Pulsar. https://pulsar.apache.org/docs/en/security-authorization/

[4] Apache. [n.d.]. Companies using or contributing to Apache Pulsar. https://pulsar.apache.org/powered-by/

[5] Apache. [n.d.]. Multi Tenancy. http://pulsar.apache.org/docs/en/concepts-multi-tenancy/

[6] Afiya Ayman, Michael Wilbur, Amutheezan Sivagnanam, Philip Pugliese, Abhishek Dubey, and Aron Laszka. 2020. Data-Driven Prediction of Route-Level Energy Use for Mixed-Vehicle Transit Fleets.

[7] William Barbour, Michael Wilbur, Ricardo Sandoval, Caleb Van Geffen, Brandon Hall, Abhishek Dubey, and Dan Work. 2020. Data Driven Methods for Effective Micromobility Parking.

[8] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. 2015. Apache flink: Stream and batch processing in a single engine.

[9] Mitch Cherniack, Hari Balakrishnan, Magdalena Balazinska, Don Carney, Uğur Çetintemel, Ying Xing, and Stan Zdonik. 2003. Scalable distributed stream processing.

[10] Cisco. 2018. *Cisco Annual Internet Report - Cisco Annual Internet Report (2018–2023) White Paper*. Cisco. https://www.cisco.com/c/en/us/solutions/collateral/executive-perspectives/annual-internet-report/white-paper-c11-741490.html

[11] Gerald Coley. [n.d.]. BeagleBone Black System Reference Manual. https://github.com/beagleboard/beaglebone-black/wiki/System-Reference-Manual

[12] Wenliang Du and Michael T. Goodrich. 2005. Searching for High-Value Rare Events with Uncheatable Grid Computing. In *Applied Cryptography and Network Security*, John Ioannidis, Angelos Keromytis, and Moti Yung (Eds.). Vol. 3531. Springer Berlin Heidelberg, Berlin, Heidelberg, 122–137. https://doi.org/10.1007/11496137_9 Series Title: Lecture Notes in Computer Science.

[13] Ted Dunning and Ellen Friedman. 2016. *Streaming architecture: new designs using Apache Kafka and MapR streams.* " O'Reilly Media, Inc.", Sebastopol, CA 95472 USA.

[14] Scott Eisele, Taha Eghtesad, Nicholas Troutman, Aron Laszka, and Abhishek Dubey. 2020. Mechanisms for outsourcing computation via a decentralized market. In *Proceedings of the 14th ACM International Conference on Distributed and Event-based Systems* (Montreal Quebec Canada, 2020-07-13). ACM, New York, NY, USA, 61–72. https://doi.org/10.1145/3401025.3401737

[15] electricchoice. [n.d.]. Electricity Rates. https://www.electricchoice.com/electricity-prices-by-state/

[16] Marisol García-Valls, Abhishek Dubey, and Vicent Botti. 2018. Introducing the new paradigm of social dispersed computing: applications, technologies and challenges. *Journal of Systems Architecture* 91 (2018), 83–102.

[17] Philippe Golle and Ilya Mironov. 2001. Uncheatable Distributed Computations. In *Topics in Cryptology — CT-RSA 2001*, David Naccache (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 425–440.

[18] Ernest J Henley and Hiromitsu Kumamoto. 1981. *Reliability engineering and risk assessment.* Vol. 568. Prentice-Hall Englewood Cliffs, NJ.

[19] Soohwan Kim and Jonghyuk Kim. 2012. Building occupancy maps with a mixture of Gaussian processes. In *2012 IEEE International Conference on Robotics and Automation.* IEEE, Saint Paul, MN, 4756–4761.

[20] Mutable. 2020. Mutable: the public edge cloud. https://mutable.io/

[21] Agostino Nuzzolo, Umberto Crisalli, Luca Rosati, and Angel Ibeas. 2013. Stop: a short term transit occupancy prediction tool for aptis and real time transit management systems. In *16th International IEEE Conference on Intelligent Transportation Systems (ITSC 2013).* IEEE, 1894–1899.

[22] Geoffrey Pettet, Ayan Mukhopadhyay, Mykel Kochenderfer, Yevgeniy Vorobeychik, and Abhishek Dubey. 2020. On Algorithmic Decision Procedures in Emergency Response Systems in Smart and Connected Communities. In *Proceedings of the 19th Conference on Autonomous Agents and MultiAgent Systems, AAMAS 2020, Auckland, New Zealand.* ACM, Auckland, New Zealand, 1740–1752.

[23] Geoffrey Pettet, Saroj Sahoo, and Abhishek Dubey. 2019. Towards an Adaptive Multi-Modal Traffic Analytics Framework at the Edge. In *IEEE International Conference on Pervasive Computing and Communications Workshops, PerCom Workshops 2019, Kyoto, Japan, March 11-15, 2019.* 511–516. https://doi.org/10.1109/PERCOMW.2019.8730577

[24] M Mazhar Rathore, Hojae Son, Awais Ahmad, Anand Paul, and Gwanggil Jeon. 2018. Real-time big data stream processing using GPU with spark over hadoop ecosystem. *International Journal of Parallel Programming* 46, 3 (2018), 630–646.

[25] Luis FG Sarmenta. 2002. Sabotage-Tolerance Mechanisms for Volunteer Computing Systems. *Future Generation Computer Systems* 18 (2002), 561–572.

[26] Mary R Schurgot, Michael Wang, Adrian E Conway, Lloyd G Greenwald, and P David Lebling. 2019. A dispersed computing architecture for resource-centric computation and communication. *IEEE Communications Magazine* 57, 7 (2019), 13–19.

[27] Shanyu Zhao, V. Lo, and C. G. Dickey. 2005. Result verification and trust-based scheduling in peer-to-peer grids. In *Fifth IEEE International Conference on Peer-to-Peer Computing (P2P'05).* IEEE, Konstanz, Germany, 31–38. https://doi.org/10.1109/P2P.2005.32

[28] Amritraj Singh, Kelly Click, Reza M. Parizi, Qi Zhang, Ali Dehghantanha, and Kim-Kwang Raymond Choo. 2020. Sidechain technologies in blockchain networks: An examination and state-of-the-art review. *Journal of Network and Computer Applications* 149 (2020), 102471. https://doi.org/10.1016/j.jnca.2019.102471

[29] Jason Teutsch and Christian Reitwießner. 2017. A Scalable Verification Solution for Blockchains. https://people.cs.uchicago.edu/teutsch/papers/truebit.pdf

[13] https://arstechnica.com/gadgets/2020/07/ibm-completes-successful-field-trials-on-fully-homomorphic-encryption/

[14] https://en.wikipedia.org/wiki/Intel_Management_Engine#Security_vulnerabilities

[30] James Usevitch and Dimitra Panagou. 2020. Resilient Finite Time Consensus: A Discontinuous Systems Approach. *arXiv preprint arXiv:2002.00040* (2020).

[31] Michael Walfish and Andrew J Blumberg. 2015. Verifying Computations Without Reexecuting Them. *Commun. ACM* 58, 2 (2015), 74–84.

[32] Xinshang Wang. 2017. *Online Algorithms for Dynamic Resource Allocation Problems*. Ph.D. Dissertation.

[33] Michael Wilbur, Afiya Ayman, Anna Ouyang, Vincent Poon, Riyan Kabir, Abhiram Vadali, Philip Pugliese, Daniel Freudberg, Aron Laszka, and Abhishek Dubey. 2020. Impact of COVID-19 on Public Transit Accessibility and Ridership. arXiv:2008.02413 [physics.soc-ph]

[34] Yitian Zhang, Jiong Yu, Liang Lu, Ziyang Li, and Zhao Meng. 2020. L-Heron: An open-source load-aware online scheduler for Apache Heron. *Journal of Systems Architecture* 106 (2020), 101727. https://doi.org/10.1016/j.sysarc.2020.101727