Decentralized Computation Market for Stream Processing Applications

Scott Eisele*, Michael Wilbur*, Taha Eghtesad†,Kevin Silvergold*, Fred Eisele *,
Ayan Mukhopadhyay*, Aron Laszka†, Abhishek Dubey*
*Vanderbilt University †University of Houston

***Abstract*—While cloud computing is the current standard for outsourcing computation, it can be prohibitively expensive for cities and infrastructure operators to deploy services, especially for streaming applications. At the same time, there are underutilized computing resources within cities and local edge-computing deployments. Using these slack resources may enable significantly lower pricing than comparable cloud computing; such resources would incur minimal marginal expenditure since their deployment and operation are mostly sunk costs. However, there are challenges associated with using these resources. First, they are not effectively aggregated or provisioned. Second, there is a lack of trust between customers and suppliers of computing resources, given that they are distinct stakeholders and behave according to their own interests. Third, delays in processing inputs may diminish the value of the streaming applications. To resolve these challenges, we introduce an architecture, combining a distributed trusted computing base, such as a blockchain, with an efficient messaging system like Apache Pulsar. Using this architecture, we design a decentralized computation market where customers and suppliers make offers to deploy and host streaming applications. The proposed architecture is independent of any particular blockchain implementation—as long as it supports smart contracts—and ensures that the market is robust to failures, while incurring the latency intrinsic in blockchain solutions only on deployment, rather than every output. We evaluate the market protocol using game-theoretic analysis to show that deviation from the protocol is discouraged. Finally, we evaluate the performance of a prototype implementation based on experiments with a streaming computer-vision application.**

I. Introduction

Edge computing is critical to balance the computing workloads necessitated by the growing integration of internet of things and smart city applications [1, 2, 3, 4]. However, the deployment and maintenance of edge computing infrastructure can be costly [5, 6]. Therefore, instead of relying only on new infrastructure, we consider the opportunity provided by the available slack computing resources in communities, owned by various stakeholders such as businesses, universities, and ISPs. By "slack" computing resources, we mean computing resources that remain after their owners' requirements are met. It is estimated that there are hundreds of exaFLOPs of surplus compute capacity available [7, 8]. The advantage of using slack resources is that the expenditure for space, hardware, and operation is already paid for in supporting the devices' primary applications. Prior efforts in volunteer computing [9] have promoted the idea of "volunteering" slack resources, it is not a reliable option since it does not provide economic benefits that could incentivize the resource providers to participate. We hypothesize that a market that enables resource providers to sell slack computing capability to customers who want to deploy an application is required.

There are several challenges that have to be resolved however. First, the resources are not effectively aggregated or provisioned. Aggregation requires participation, and provisioning is difficult because streaming applications are often long-running; slack capacity, on the other hand, is transient and subject to the demands of the primary application. Therefore, it is likely that agents who provide slack compute will not be able to host a service for its entire life cycle. Second, it is imperative to establish trust between the customers and suppliers since they may behave selfishly. For example, the resource providers could claim that they executed a job without actually doing so. Customers, on the other hand, could provide cleverly crafted jobs that induce failures in resource providers to avoid payment, even for parts of the input that were processed correctly. Therefore, some assurance about the veracity of the results is imperative. Third, the value of the outputs from streaming applications may diminish with time. This means that the mechanisms used to establish trust should not delay the output.

Existing approaches provide partial solutions to these problems [10, 11, 12, 13, 14]. For example, Mutable [13] and Aurora [14] aggregate surplus compute resources, but only from trusted entities like internet service providers. This constraint sidesteps the problem of validating results but leaves the bulk of the surplus resources untapped. Eisele et al. [11], Teutsch and Reitwießner [12] allow mistrusted entities and rely on blockchain-based distributed ledgers where there is no central trusted entity; instead, trust is distributed among the participants to provide trusted compute. Recognizing that blockchain-only systems are inefficient, slow, and have limited throughput, these protocols perform the computation and verification on standard compute nodes. However, Teutsch and Reitwießner utilize a costly mediation mechanism that involves storing computation data on the blockchain, while Eisele et al. is limited to batch processing applications.

To address these issues, we develop a decentralized market to incentivize participation. To handle the volatility inherent in surplus resources, *Customers* specify a minimum service time. To address the potential for *Suppliers* to fail accidentally, Customers can request that multiple Suppliers host their service. Similar to [10, 11, 12], we choose to rely on blockchain to provide a trust foundation. We chose this primarily because it does not require participants to trust a

single centralized entity. In addition, it is robust and has high availability. Recognizing the limitations, we restrict the use of the smart contract to recording allocation contracts, providing a minimal verification service, and transferring funds. All other functionality is implemented using another distributed ledger, in particular Apache Pulsar. These two mechanisms work in parallel and only synchronize during the final payout. As a result, the streaming applications may be deployed without incurring delays to waiting for blockchain transactions.

To retain trust despite the use of a second ledger, we deter some actions available to the participants by designing a protocol that is both incentive-compatible and individually rational. This protocol establishes a game and includes in the smart contract a mechanism that verifies outputs generated by the participants (who are treated as rational players) to ensure expected outcomes. Then, disputes, if they occur, are resolved by a *Mediator*. We show using game-theoretic analysis that this approach is sufficient to disincentivize deviation from the protocol. This analysis is an integral part of the system design since enforceable rewards and penalties are crucial in a decentralized and trustless setting.

The outline of the paper is as follows. We first explain the problem, then describe our approach, followed by the analysis of the protocol. Finally we present an experimental description. We show that the performance of our framework affects only the initial deployment and not the streaming performance., which does not depend on the market but rather on the resources available to the Supplier. The implementation is available at [15].

II. Problem Formulation

A. Assumptions

We consider that the actors are selfish but non-malicious entities, i.e., they may try to cheat; however, given options, they will make choices that optimize their utility. In particular, we assume that each s_i, c_i (a Supplier or Customer) has a utility function U, and a set of actions Γ to choose from. Since the actors are rational, agent i chooses action $\chi^* \in X$ such that $\gamma^* = \arg\max_{\gamma \in \Gamma} U$. The utility function takes the general form $U = \sum(benefits) - \sum(costs)$. We define these participants and some key attributes formally below.

Definition 1 (Suppliers of Computation Resources)**:** A Supplier s is a rational agent that, for some limited duration, has surplus computing resources available. Specifically, it has R_s MB of memory and I_s CPU cycles (in millions) per second available for a duration defined as $\Delta_s = s_{end} - s_{start}$, where s_{start} and s_{end} denote the start and end clock times of the availability, respectively. We assume that the Supplier knows its primary workloads[1] and can estimate I_s, R_s, and Δ_s. To ensure profitability, the Supplier must require payment in excess of its operating costs, specifically, the cost of the electricity consumed to host a service, denoted by $\pi_{s\epsilon}$.

[1]In our definition, we do not include other resources such as disk space, GPU cycles, and network bandwidth. While these are additional constraints on the resource allocation algorithm, they do not fundamentally change the problem under consideration.

Definition 2 (Customer and Application Service)**:** A Customer c is a rational agent that has an application service (organized as a docker image) to deploy using our platform. The application 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. Each deployment lasts for a specific duration, known as the *service lifetime*, and is 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_c. For every service output, the Customer receives a benefit b and is willing to pay up to π_{xmax}.

B. Requirements

Given a set of offers, the market must enable suppliers and customers to benefit from participating in the market. This implies that the market provides an allocation that assigns a job to a supplier. Further, we need to ensure that the participants trust the market. Unfortunately, the nature of the utility function (Section II-A) encourages the suppliers to neglect processing service inputs since electricity costs $\pi_{s\epsilon}$ can be saved by reducing processing. Therefore, we face the challenge of identifying the action space of the participants and designing a mechanism that makes undesired behavior expensive for the participating actors.

III. Our Approach

Before we introduce our market, we introduce additional actors that participate in our market protocol and formally describe *offers* through which the customers and suppliers interact with the market.

Definition 3 (Supplier Market Offers)**:** Both Customers and Suppliers participate in the market by making offers. A Supplier offer $os \in O_s$ is a tuple that includes: A_{si}, a unique account identifier associated with the Supplier that posted the offer; I_s, the number of surplus instructions (in millions) per second available; R_s, the amount of RAM available; s_{start}, when the resource is first available; s_{end}, when the resource availability ends; π_{xmin}, the minimum price the Supplier is willing to be paid per million instructions; and $[M]$, a list of trusted Mediators (we describe mediators below).

Definition 4 (Customer Market Offers)**:** A Customer offer $oc \in O_c$ is a similar tuple that includes: A_{ci}, a unique account identifier associated with the Customer that posted the offer; I_c, CPU instructions (in millions) to process each input; R_c, the amount of RAM required; c_{start}; c_{end} (defn 2); π_{xmax}, the maximum price the Customer is willing to pay per million instructions; $name$, the name of the application service to be deployed; λ, data input rate of the service; and $[M]$, a list of trusted Mediators.

Definition 5 (Allocators)**:** An Allocator is an agent that aggregates participant offers, solves the resource matching problem, and provides an allocation. In particular, an allocation is a tuple which consists of $customer$, the allocated Customer; $\{suppliers\}$, a set of allocated Suppliers; a_{start}, the allocation start time; a_{end}, the allocation end time; $name$, the service

name; and π_x, the service price per million instructions such that $\pi_{xmin} \leq \pi_x \leq \pi_{xmax}$. For the service price π_x, any value between π_{xmin} and π_{xmax} is feasible, and is determined by the Allocator according to its allocation algorithm (e.g., double auction, fixed price, etc.).

Effectively, an allocation declares which offers were matched, the start and end times, and the service price. This is only possible if

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

$$[s_{start}, s_{end}] \cap [c_{start}, c_{end}] > \Delta_{min} \quad (2)$$

i.e., the Supplier has sufficient resources to process inputs at the necessary rate, there exists a price that both Customer and Supplier would accept, and the times that the offers overlap exceeds the Customer's minimum viable service time Δ_{min}.

Definition 6 (Minimum Viable Service Time)**:** To offset the setup costs π_{setup} of making and matching offers, a Customer's service must run for a minimum amount of time. We call this the *minimum viable service time* and denote it by Δ_{min}. In addition to the start and end times of the service, the Customer also includes a minimum service time in the offer. While evaluating the offers, if the Allocator detects that $currenttime + \Delta_{min} \geq c_{end}$, then the offer becomes expired. Similarly if an allocation is created, it expires when $currenttime + \Delta_{min} \geq a_{end}$.

If an allocation is agreed to by all parties then it becomes an *allocation contract*. The duration of the allocation is defined as $\Delta = a_{end} - a_{start}$. The total value of an allocation is $\pi_{total} = \pi_s\lambda\Delta$, where π_s is the value of processing a single service input: $\pi_s = I_c\pi_x$. In our market, a number of Allocator agents can participate and incorporate various resource matching and constraint satisfaction algorithms to come up with their version of feasible allocation. The Allocator receives a payment π_A for providing this service. This cost contributes to the setup costs of consumers (π_{c_setup}) and suppliers (π_{s_setup}) .

Definition 7 (Mediators)**:** Mediators are actors that are trusted to perform mediation in the case of a dispute on a particular allocation. The process involves recomputing the contended output and determining fault. Customers and Suppliers specify in their offers which Mediators they are willing to trust for that offer, and an allocation is only feasible if there a trusted Mediator in common. If a participant chooses to no longer trust a particular Mediator then it no longer includes it in its offers.

A. Understanding Costs

In addition to monetary setup costs, there are costs associated with the time delay between when an offer is submitted and when that offer is deployed. This delay depends on Supplier availability, the time required to construct an allocation, denoted by δ_{alloc}, and the time to set up the service, denoted by δ_{setup}. Service setup includes transferring and starting the service. Assuming that the service inputs are being generated continuously, the cost of this delay is at least

$$\pi_{delay} = \lambda \times \pi_s \times (\delta_{setup} + \delta_{alloc}). \quad (3)$$

The Customer must account for this delay when specifying offer start time, specifically $c_{start} = c^*_{start} - (\delta_{alloc} + \delta_{setup})$ where c^*_{start} is the actual desired service start time. Finally, if a Supplier hosting a service fails, the cost to the Customer to recover from the failure is

$$\pi_{recover} = \pi_{c\_setup} + \pi_{delay}. \quad (4)$$

B. Middleware Components: Smart Contract and Pulsar

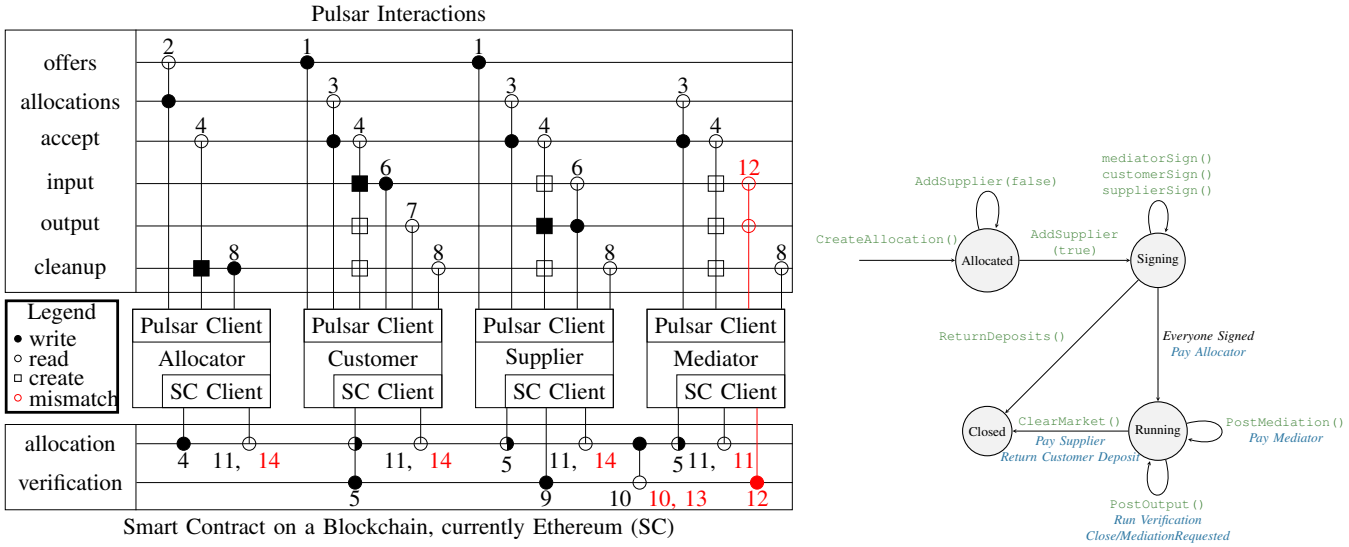
In our market, there are two middleware components that are critical to its operation. First, we use a smart contract and Blockchain[2] to enforce allocation contracts by recording signed allocations, providing a verification service, and providing registration to the market by accepting deposits and transferring payments upon successful completion of the contract. Second, we rely on a distributed ledger like Apache Pulsar [16] to record messages and transfer data between market participants. The fundamental properties we need from the distributed ledger are fast, efficient, durable storage of ordered messages that can be routed and are eventually delivered. Note that both blockchain and Pulsar record aspects of the market state, however, payments and contracts are only finalized through smart contracts in Blockchains that provide trust assurances.

IV. Market Protocol

We describe the market protocol (Fig. 1a) using numbers in the text (e.g., ①), which denote the events that take place. We also refer to the state machine shown in Fig. 1b to describe the protocol, which depicts how the smart contract tracks the state of each allocation. In our notation, text in `teletype` font represents smart contract functions and text that is *italicized* are state machine states.

1) **Making Offers:** The protocol begins with the Customers and Suppliers constructing their offers and sending them on the offers channel ①.
2) **Creating Allocation:** The Allocator reads the offers channel ② and executes a matching algorithm to construct an allocation, if one exists. It then sends the allocation on the allocations channel.
3) **Accepting Allocation:** The Customers and Suppliers read the allocations channel and send a message on the accept channel ③ to specify if they accept the allocation or not. The Allocator reads the accept channel and, if all the allocated participants accept, sends the allocation to the Smart Contract (SC) and requests that it calls the `createAllocation` function ④. The Smart Contract checks the feasibility and correctness of the allocation, and the state of the allocation is initiated to *Allocated*. As part of submitting the allocation to the SC, the Allocator sends additional requests (`AddSupplier`) to add the id of each Supplier and the hash of its offer to the allocation stored by the SC. These two function calls incur a cost of

[2]We can support different implementations, but we use Ethereum in the current prototype.



(a) (b)

Fig. 1: (a) Horizontal lines represent communication channels between participants. 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 a reader (denoted by a square) 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 numbers signify the sequence after the outputs checked by the smart contract deployed on the blockchain do not match. (b) State of an allocation on smart contract represented as a state diagram. Our smart contract functions are described in [15].

allocation π_{ca}. When all Suppliers are added, the state of the allocation transitions to *Signing*.

Once the allocation is in the *Signing* state on the SC, the participants check the allocation to make sure that it matches the specific allocation that the Allocator sent on the allocations channel; if it does, they sign the allocation on the SC by submitting their security deposits ⑤. Additionally, as part of signing the allocation, the Customer commits n *tests*, which are inputs that will be hidden in the data stream going to the service, and their corresponding outputs. It does this in two steps. First, by processing the n inputs and hashing the inputs and outputs incurring a cost $n\pi_{cg}$. Second, it sends those hashed tests to the SC incurring a commitment cost of π_{cc}. During the service lifetime those inputs are injected uniformly at random among the workload inputs[3]. See Section IV-A.

When all the participants sign the contract, the state of the allocation is changed to *Running*. The allocated Customer, Suppliers, and Mediators also incur a cost for signing the allocation and paying the Allocator for its service; we denote these costs, lumped together, as π_a. Once all participants have signed the contract, the Allocator receives π_A as payment for its service.

4) **Service Execution:** After the service-specific channels are constructed, the Customer can begin writing to the service input channel and the Supplier can begin reading ⑥, processing inputs and sending outputs on the service output channel (this process can actually start before signing is complete). For each correctly processed input the Customer reads ⑦, it receives a benefit b. The Supplier, on the other hand, incurs electricity cost $\pi_{s\epsilon}$ for processing each input.
5) **Verifying Outputs:** At the end of the allocation, the Allocator sends a message on the cleanup channel ⑧ 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 the them to the SC for verification ⑨, calling `postOutput`. The Supplier incurs a cost of π_v for performing this operation. The SC compares the Supplier output against the test data committed by the Customer when it signed the allocation and stores the result[4]. We discuss verification further in Section IV-A.
6) **Mediation and Closing the Allocation:** If the outputs match in the previous step, the SC calls *ClearMarket* (black ⑩) which transfers payments, with the Supplier receiving $\lambda_\Delta \pi_s$, the Customer paying $\lambda_\Delta \pi_s$, and the Mediator receiving π_m for being available. It also causes the allocation state on the SC to transition to *Closed*. The participants receive notification of this change via their respective SC clients (black ⑪). The allocation is then finished.

However, if the outputs do not match, the SC records this outcome and emits a MediationRequested event (red ⑩), which the Mediator receives from its SC client (red ⑪). The Mediator then reads ⑫ from the input and output channels and re-processes the n inputs, incurring a cost of

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

[4]The verification is only capable of detecting errors, not ascertaining which entities are at fault. To determine fault, we use a Mediator.

$n(\pi_{s\epsilon}+\pi_{cg})$. The Mediator then sends the result to the SC, calling `postMediation`, which incurs a cost of π_{mc}. The SC calls *ClearMarket* (red (13)) which compares the Mediator output against the output of the Suppliers and Customer to determine which participants are at fault. If a Customer or Supplier's output did not match the Mediator output, they are fined π_{cd} or π_{sd} respectively. The Mediator is paid $n(\pi_s)$ from the fine. For any agent whose output does match the Mediator, payment is transferred as normal, i.e., the Supplier receives $\lambda\Delta\pi_s$ or the Customer pays $\lambda\Delta\pi_s$. The call causes the state of the allocation to transition to *Closed*. The participants receive notification of this via their respective SC clients (red (14)). The allocation is then finished.

A. Verification

As part of the verification process, we expect the Customers to commit to at least n inputs and their corresponding outputs which we refer to as *tests*, where n is a platform parameter as part of an accepted allocation. It is also important that the Supplier cannot read them; otherwise, the Supplier can copy the outputs (by reading from the blockchain) and provide those outputs at the end of allocation and neglect processing the actual inputs. To prevent such behavior, our protocol dictates that the Customer uses a hash function to mask the values it commits in the blockchain. The key idea behind the verification strategy is to check the hash of the Supplier output against the hash of the expected output, without letting the Supplier know which output is being checked.

We introduce some additional notation to describe the solution. Let $O_c = \{o_1, o_2, \dots o_n\}$ represent the set of the Customer's n tests where $o_i = (in_i, out_i)$ is the i^{th} test for a particular Customer. Let $O_s = \{o_{s1}, o_{s2}, \dots o_{sn}\}$ represent the set of all test solutions produced by the Supplier during an allocation. To mask the input and output values, we use hash functions that are supported by the chosen Blockchain implementation—for example, since we use Ethereum we can use keccak256, sha256, and ripemd160 [17]. We define the following functions for hashing. α hashes all elements of a set K and creates a set of hashes, γ applies a hash function to a given set and produces a single hash value, and Γ applies a double hash. The cardinality of outputs produced by γ and Γ is one, whereas α produces a set whose size is same as the size of input elements. That is, $\alpha(K) = \{hash(k_i) : \forall k_i \in K\}$, $\gamma(K) = hash(K)$, and $\Gamma(K) = hash(hash(K))$.

During the finalization of an allocation, the Customer sends $\alpha(O_c)$, a set of hashes to the Supplier. During execution, the Supplier records the hash of each input and output, so upon receiving the set provided by the Customer it is able to determine the set of test inputs that it must submit to the Smart Contract for verification. Note that this list does not directly specify the index of the input or output that must be sent. Rather, it identifies the $O_v \subseteq O_s$, such that $O_v = \{o_i : hash(o_i) \in \alpha(O_c) \cap \alpha(O_s)\}$. This approach ensures that the Supplier must have processed the test inputs to be able to correctly identify them. If the Customer directly provided the indices, the Supplier could neglect to process the inputs until it received them and then produce O_v. Finally, the Supplier sends $\gamma(O_v)$ to the smart contract, where the smart contract then checks if $\Gamma(O_c) = hash(\gamma(O_v))$. Recall that, during signing, the Customer committed O_c to the blockchain. In reality, it sent $\Gamma(O_c)$. The data is double hashed because if the Customer had sent $\gamma(O_c)$, then the verification process on the blockchain would have required to hash all the Supplier's n outputs, thereby incurring additional costs. Instead, the Customer sends $\Gamma(O_c)$, requiring the Supplier to send $\gamma(O_v)$ to the smart contract. This makes it so that the smart contract only has to

TABLE I: Key Symbols

Smart Contract (SC)	
ρ	penalty rate set by the SC
π_v	cost of Supplier submitting outputs to the SC
π_{cc}	cost of Customer committing outputs to the SC
π_{mc}	cost of Mediator committing mediation results to the SC
Mediator (M)	
π_m	payout to the Mediator for being *available* for the duration of the service
$\pi_{v\epsilon}$	Mediator's electricity cost to verify outputs
Customer	
c_i	Customer i
A_{ci}	Customer i's account ID
$c_{start,end}$	Customer offer start and end times
π_{xmax}	amount the Customer is willing to pay per million instructions
I_c	number of instructions (in millions) required to process a service input
b	benefit that the Customer obtains from service output
π_{cg}	Customer's cost of generating test output
e_c	$= \frac{\pi_{cg}}{\pi_s}$ Customer's efficiency of processing vs. the price paid to outsource
λ	rate at which data is set to deployed application instance
s_c	represents when the Customer chooses to provide n correct tests (true or false)
Supplier	
s_i	Supplier i
A_{si}	Supplier i's account ID
$s_{start,end}$	Supplier offer start and end times
π_{xmin}	payment that the Supplier requires per million instructions
$\pi_{s\epsilon}$	cost to process a service input
π_v	cost to send output hash to the SC
$P(s)$	probability that the Supplier will process a particular input
e_s	$= \frac{\pi_{s\epsilon}}{\pi_s}$: Supplier's efficiency of processing vs. the price paid to outsource
I_s	number of slack instructions (in millions) per second available
R_s	surplus RAM available
Allocator	
Ω	an allocation
π_a	cost to pay Allocator (π_A) and for signing the allocation
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 Customer/Supplier for a processed input
π_{ca}	cost of SC adding an allocation
n	number of outputs that must be provided by the Customer for verification
π_{cd}	Customer's security deposit for collateral prior to transaction (set to $\rho\pi_s$)
π_{sd}	Supplier's security deposit for collateral prior to transaction (set to $\rho\pi_s$)
Other	
$x_{start/end}$	Customer's service start/end time ($x := c$), start/end of Supplier's resource availability ($x := s$), start/end of allocated service time ($x := a$)
Δ_{min}	minimum viable service time

hash a single element to compare against the Customer's hash.

B. Handling Failures

Our protocol ensures that the consequences of failures are localized to the specific stakeholder that experiences the failure. If a Customer fails prior to signing, the Allocation is canceled. If the failure occurs after, the Supplier is unaffected. In the case of a Supplier failure the Customer submits a new service offer which incurs the cost $\pi_{recover}$ (see Eq. (4)). Since the Supplier is penalized in these cases, they choose their availability based on estimates of their reliability to reduce the cost of unintentional failures. Thus, from a market perspective, as long as more than one agent sends offers to an Allocator and the agents have a mutually accepted Mediator, the Market is operational.

C. Prototype

In the current prototype, we use the Ethereum blockchain. The system can also be implemented by using other ledgers that provide byzantine fault tolerance, such as Hashgraph [18]. Further, instead of Pulsar, we can use other distributed Ledger such as Kafka and even support the side-chains, however implementations of side-chains are still relatively new and immature [19]. We chose Pulsar in particular due to its support of multi-tenancy. This feature enables the ownership of the market to be distributed between the participants, who can each control access to their tenants.

The various actors in the system, i.e., Customers, Suppliers, Allocators, and Mediators were implemented in Python and packaged as Docker Containers. Further, we use Docker to pass the application code between Customer and the Suppliers. The default allocators (given a set of current offers): 1) finds feasible mapping through brute force search, 2) utilize the Hopcroft-Karp [20] algorithm to output a maximum cardinality matching, 3) run a double auction to determine a fair price, 4) and construct the final allocation. Each time a new offer is received by the Allocator, it runs to see if any allocations can be constructed. In this way it provides a result in the minimum possible time. Other allocation algorithms can be easily integrated into the architecture.

V. Protocol Analysis

This protocol has been designed to disincentivize deviation from the protocol. We show this by deriving the participants' utility functions, the incentives associated with each action and model the protocol as a two-player simultaneous move game. We refer reader to Table I for reference.

A. Customer Utility

For each input sent, the Customer pays π_s, and for each correct output, the Customer receives a benefit b. Also, for each allocation the customer incurs a setup cost π_{c_setup} which is

$$\pi_{c\_setup} = n\pi_{cg} + \pi_{cc} + \pi_m + \pi_a \tag{5}$$

where π_{cg} is the cost of generating a test, n is the number of tests, π_{cc} is the cost of committing the hash of the tests to the SC, π_m is the payment to the Mediator for being available, and π_a is the payment to the Allocator for the allocation. The Customer utility then is

$$U_C = \lambda\Delta(b - \pi_s) - n\pi_{cg} - \pi_{cc} - \pi_m - \pi_a \tag{6}$$

Based on the utility function, the Customer can improve its utility if it takes actions that allow it to avoid paying π_s to the Supplier, or reduce the number n of test inputs that must be provided.

Therefore, we must ensure the customer cannot avoid paying π_s and will commit n test inputs. We can ensure that the Customer cannot avoid payment by having the Customer pay $\lambda\Delta\pi_s$ at the end of the allocation regardless of the outcome. This decision avoids complications that appear with refunding services that fail, as in prior work [11]. This design choice may seem unfair—there is a possibility that the Customer pays for a service that is not delivered. However, our justification for using such a mechanism is that in the game analysis we will show that this loss does not occur when all participants behave rationally.

To detect when a Customer does not submit n correct tests, the following measures are taken. During signing, in addition to the hash of the tests $\Gamma(O_c)$, the Customer includes a hash of the list of inputs $\Gamma(In_c)$. Similarly, when the Supplier calls `postOutput`, it includes a hash of the list of n test inputs $\gamma(In_v)$. If during the verification process $\Gamma(In_c) \neq hash(\gamma(In_v))$, then mediation is requested. To make sure the Customer commit was honest, the Mediator reprocesses the n test inputs and computes $\Gamma(O_c)$ and $\Gamma(In_c)$ and also recomputes $\alpha(O_c)$ and compares against the set of hashes $\alpha(O_c)$ the Customer sent to the Supplier for identifying the tests. This process ensures the proposed system detects if the Customer does not provide the correct commitment or data to the Supplier. Since there is no way for the Customer to avoid paying, the only deviation from the protocol the Customer can take is to provide an incorrect commitment of n tests which will be detected and penalized.

B. Supplier Utility

Recall λ is the incoming data rate and Δ is the duration of the allocation, therefore, $\lambda\Delta$ is the total number of inputs sent during an allocation. That is, if the supplier runs an application successfully, it will produce $\lambda\Delta \geq n$ outputs, where n is the number of tests used for verification (Section IV-A).

The supplier incurs a cost of $\pi_{s\epsilon}$ processing an input and z is the number of inputs processed. For each allocation, the Supplier incurs a setup cost of

$$\pi_{s\_setup} = \pi_v + \pi_m + \pi_a \tag{7}$$

where π_v is the cost of sending a `postOutput` request to the SC. The Supplier utility then is

$$U_S = \lambda\Delta(\pi_s) - z\pi_{s\epsilon} - \pi_v - \pi_m - \pi_a \tag{8}$$

Based on the utility function, the Supplier can improve its utility if it takes actions that allow it to reduce the number z of inputs processed, i.e. skip the processing of some inputs.

	Outcomes			
	o_1 (ss_c)	o_2 $(\overline{s}s_c)$	o_3 $(s\overline{s_c})$	o_4 $(\overline{s}\overline{s_c})$
Customer	$\lambda\Delta(b-\pi_s)-n\pi_{cg}-\pi_{cc}-\pi_a-\pi_m$	$-\lambda\Delta\pi_s-n\pi_{cg}-\pi_{cc}-\pi_a-\pi_m$	$\lambda\Delta(b-\pi_s)-\pi_{cc}-\pi_{cd}-\pi_a-\pi_m$	$-\lambda\Delta\pi_s-\pi_{cc}-\pi_{cd}-\pi_a-\pi_m$
Supplier	$\lambda\Delta(\pi_s-\pi_{s\epsilon})-\pi_v-\pi_a-\pi_m$	$-\pi_v-\pi_{sd}-\pi_a-\pi_m$	$\lambda_\Delta(\pi_s-\pi_{s\epsilon})-\pi_v-\pi_a-\pi_m$	$-\pi_v-\pi_{sd}-\pi_a-\pi_m$
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 II: Game outcomes and payments from Fig. 2. 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.

C. Incentives

The goal is to have Customers to provide correct test data and Suppliers to process all inputs. In order to disincentivize deviation from this behavior, the Customers and Suppliers are required to provide a security deposit π_d to participate. π_d is computed as

$$\pi_d = \pi_s \lambda\Delta\rho \tag{9}$$

where ρ is a penalty rate defined by the market. Therefore, if the Customer is detected for not providing the correct test data, or the Supplier is detected for skipping the processing of inputs, they are fined π_d.

D. Interaction Between Customer and Supplier

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 $\overline{s}$). Similarly, the Customer can choose to provide n correct tests (s_c) or not ($\overline{s_c}$). The resulting utility for each possible combination is shown in Table II, and the resulting game is shown in Fig. 2.

The Customer's dominant strategy is to honestly provide n test inputs as long as the utility of providing the inputs (s_c) is greater than the utility of not providing the inputs ($\overline{s_c}$). Formally if

$$\Big[U_C(s,s_c) > U_C(s,\bar{s_c})\Big] \wedge \Big[U_C(\bar{s},s_c) > U_C(\bar{s},\bar{s_c})\Big] \tag{10}$$

then $U_c(*,s_c)$ is the Customer's dominant strategy, where $*$ represents any strategy of the Supplier. To determine the conditions that make this true we reference the Customer outcomes in Table II and substitute them into Eq. (10) and simplify. Both inequalities in Eq. (10) result in the same simplified inequality $n\pi_{cg} < \pi_{cd}$. Recall that $\pi_{cg} = e_c\pi_s$, where $e_c < 1$ is the customer's processing efficiency and π_{cd} was set to $\pi_s\lambda\Delta\rho$. Substituting these values in and simplifying the inequality gives

$$\begin{aligned} n\pi_{cg} &< \pi_{cd} \\ ne_c\pi_s &< \pi_s\lambda\Delta\rho \\ ne_c &< n < \lambda\Delta\rho \end{aligned} \tag{11}$$

As shown in Eq. (11), the Customer will always process n inputs as long as $n < \lambda\Delta\rho$, where ρ is the penalty rate set by the market. Since we can ensure that the Customer's only reasonable strategy is to provide the test inputs, we do not need to consider (for the Supplier) the case when the Customer does not provide the inputs.

The Supplier can neglect processing inputs during the allocated service time. Therefore, processing all inputs (s,s_c) is the dominant strategy for the Supplier as long as $U_S(s,s_c) > U_S(\bar{s},s_c)$, which holds 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.

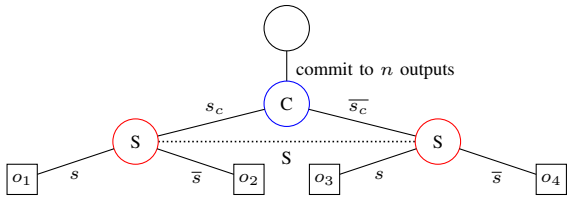


Fig. 2: Extensive-form game produced by our protocol. Blue nodes indicate Customer moves, red nodes indicate Supplier moves. The game is sequential, but the decisions are hidden, so we treat it as a simultaneous move game. Each outcome has payouts for the agents (Table II).

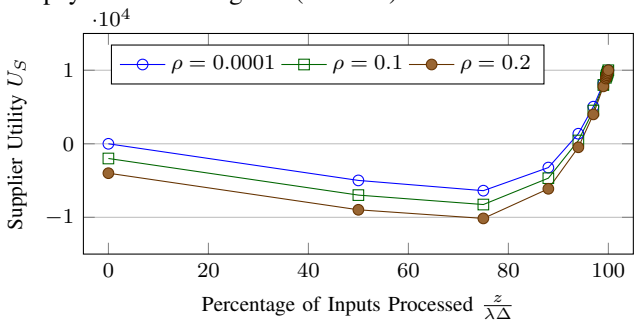


Fig. 3: Supplier utility U_S as a function of the percentage of inputs processed by the supplier ($\frac{z}{\lambda\Delta}$). Utility is highest when all inputs are processed, regardless of ρ.

This presumes that the Supplier processed every input or none of them. However, it is possible that the Supplier can risk skipping the processing of some inputs to reduce its costs. Given that the Customer only checks n inputs, as long as the Supplier processes those n inputs it will not be caught. To derive the Supplier's utility, recall that $\lambda\Delta$ is the total number of inputs, n is the number of test inputs and z is the number of inputs processed by the Supplier. In this case, there are $\binom{\lambda\Delta}{z}$ total ways to select which z inputs are processed by the supplier and $\binom{\lambda\Delta-n}{z-n}$ ways for the supplier to not be detected. Therefore, the probability of the supplier getting detected is represented by P_d, where P_d is

$$P_d = 1 - \frac{\binom{\lambda\Delta-n}{z-n}}{\binom{\lambda\Delta}{z}} \tag{12}$$

We can then define the utility of the supplier (U_S) as

$$U_S = \lambda\Delta\pi_s(1-P_d) - P_d\pi_{sd} - z\pi_{se} - \pi_a - \pi_m - \pi_v \tag{13}$$

TABLE III: Experiment Scenarios

Scenario	# of Customers	# of Suppliers	Supplier Type
S1	10	10	ideal
S2	20	10	ideal
S3	20	20	ideal
S4	10	10	dishonest
S5	20	10	dishonest
S6	20	20	dishonest

We plot this utility in Fig. 3, where we see that the Supplier obtains maximum utility when it processes all of the inputs.

This analysis shows that since the penalty multiplier ρ is set by the market, the market can ensure that the Customer will always provide n correct tests. Further, since ρ and n are market parameters, and λ and Δ are determined by the service the market can ensure a minimum value for P_d.

VI. Experiments

We have shown analytically that our protocol will deter deviation from the protocol for rational agents. We now present the experimental performance. To emulate a large market, we deployed the actors as Kubernetes Pods using the Google Kubernetes Engine (GKE). CPU and RAM utilization were extracted directly from GKE's monitoring framework using BigQuery. The blockchain was deployed on a private network locally in our lab on a machine with Intel Xeon CPU and 64 GB RAM.

The application deployed on this market is a real-world application from our partner transit agency that processes a stream of video frames to count the number of passengers in each vehicle [21, 22]. The data is used for guiding the planning of decisions that include deploying additional transit vehicles to an area later in the day. We considered six scenarios (Table III), where we varied the number Customers and the number and type of Suppliers. The types of Suppliers are: ideal Suppliers that do not cheat and process all inputs, and dishonest Suppliers that correctly process an input with a 50% probability, otherwise they provide a random output. Each of the Customers posted one offer to run the occupancy detection application on 600 frames.

A. Deployment Overhead

For each scenario, we measured the delay between submitting and offer and receiving the allocation from the Allocator. The result in Fig. 4 shows that the median allocation time across scenarios ranged from 9 to 12 seconds for Customers and 2 to 12 seconds for Suppliers. Additional delay is incurred if the participants wait until the participants sign the allocation stored by the smart contract. The block mean mining time on our private Ethereum network was 12 seconds. Thus the minimum time to call the `createAllocation`, `addSupplier` and the various `sign` functions is 45 seconds on average. To verify we measured the time between calling these functions and the transaction being added to block and measured 13.6 ± 6.5, 15.6 ± 18.5, and 14.4 ± 9.1 accordingly. The other smart contract functions (e.g., `postOutput`, `postMediation`) are not in the critical path of deploying and running the application and so do not delay the application deployment.

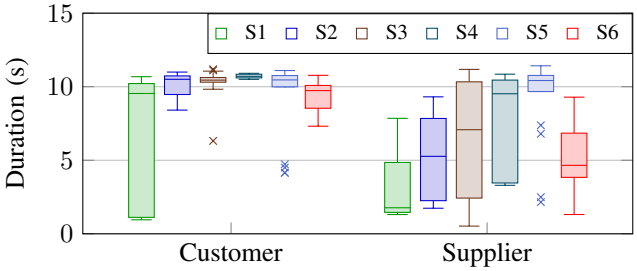


Fig. 4: Time between when a customer (or supplier) submits an offer and receives an allocation.

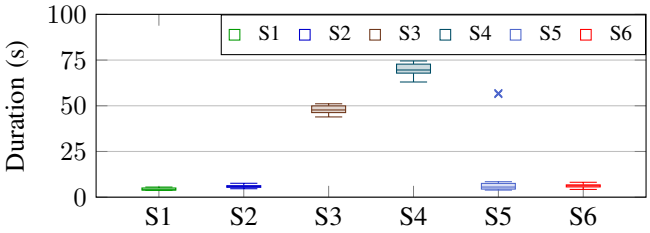


Fig. 5: Time spent setting up the application for all Suppliers in scenarios S1-S6.

We also measure the time to start the application once the allocation has been accepted. This result in Fig. 5 varies from 3 to 74 seconds. This time includes downloading and starting to run the application docker image, and depends heavily on the size of the image itself. In our case, the occupancy detection image was 3GB which impacted the setup time. It is important for Customers to take this into account when setting the start time of the service. Scenarios 1, 2, 5, and 6 have significantly shorter setup times, this is due to images persisting on disk between runs eliminating the time needed to download the image. In the scenarios with dishonest Suppliers we measure the time spent in mediation. This result in Fig. 6 shows that the median time for mediation ranged from 145 to 210 seconds. The duration of mediation increased as the number of sporadic Suppliers increased. One way to reduce Mediation time is load balance the Mediators available in the system.

During the scenarios we recorded the CPU and memory utilization of the pods. The results in Figs. 7 and 8 show the average CPU utilization and RAM requirements respectively for the Customers, Suppliers and Mediator in scenarios S1-S6. Both CPU and RAM usage was consistent for Customers and Suppliers between scenarios. Scenarios S4-S6 consisted of included dishonest Suppliers and thus required mediation which is reflected an increase in CPU and memory usage for the Mediator in these cases.

B. Monetary Costs

The monetary costs of the system are primarily a consequence of using Ethereum. The gas costs of each function can be found in table Table IV. The cost to a Customer of using the market for an ideal iteration of the protocol is 322,222 Gas which includes the cost of signing the allocation, plus

TABLE IV: Gas Costs and Delay of Each Smart Contract Function Call

Function	createAllocation	addSupplier	customerSign	supplierSign	mediatorSign	postOutput	postMediation
Gas	208,404	127,038	113,818	109,931	58,053	88,992	87,944
Response time[s]	13.6 ± 6.5	15.6 ± 18.5	14.4 ± 9.1	12.6 ± 7.1	16 ± 19.3	15.2 ± 11	28.5 ± 18

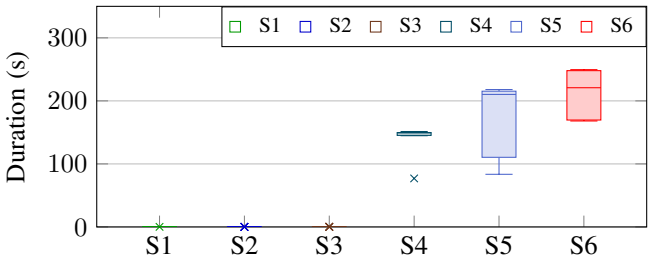


Fig. 6: Time spent in mediation for scenarios S4-S6 due to dishonest Suppliers.

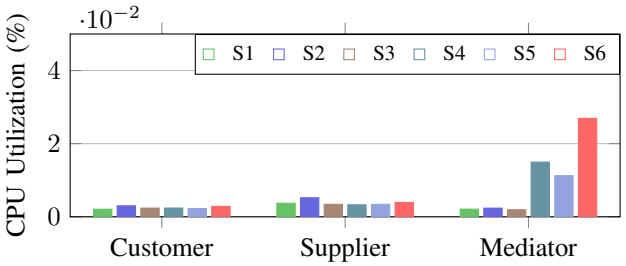


Fig. 7: Average CPU utilization per customer, supplier and mediator for scenarios S1-S6.

the amount it pays to the Allocator to create the allocation. The cost in dollars will depend on which Ethereum network is used. The cost on the Ethereum MainNet, using the conversion [23] (On 4/20/22 With cost of 30Gwei/gas), is $30.22. Alternatively using a Layer 2 scaling solutions such as the Polygon network [24] results in a cost of $0.0168 [25]. For our implementation we ran a private Ethereum Network and measured the power consumption. The load due to running a miner was approximately 15 watts on the lab machine with Intel Xeon CPU and 64 GB RAM, which has a base power usage of 145 watts. Thus the cost of operating the Ethereum miner, using $0.10/kWh for the cost of electricity, is $0.0015/h.

VII. Related Research

Market-driven approaches to outsourced computation have been studied in the context of residual cloud computing [26] and batch processing [11]. Cherniack et al. [27] outlined a federated market from which producers and consumers derive value from streaming data; however, they do not address trust in their design. TrueBit [12] is a platform designed to extend the computation capabilities of blockchain-based consensus computers (such as Ethereum) that provide strong guarantees that small computations are performed correctly. However, their platform is not suitable for stream computing because

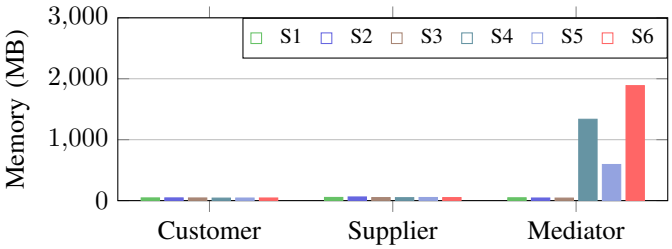


Fig. 8: Average memory used per customer, supplier and mediator for scenarios S1-S6.

it relies on the Solvers sending their results to the blockchain before they are released.

Dong et al. [28] determine that for verifying outsourced computation, the cryptographic approach is not practical and should instead use repeated executions. The computations, however, should not be duplicated more than twice. Their strategy is simple: outsource to two providers and compare results. However, their approach does not restrict collusion between suppliers, especially if interaction outside smart contracts is allowed. Our approach relies on mediation and using n inputs make collusion unlikely.

Coopedge [10] is a recently introduced edge computing platform implemented on HyperLedger. Their approach works for co-operative offloading of cloud computing tasks to edge servers participating in the network. Their primary trust mechanism relies on reputation, assuming that the edge servers have a long history and can gather enough reputation over time. Their incentive mechanism is task based, higher rewards are offered for completing a task sooner, and the time is determined through consensus on the blockchain. This mechanism is not well suited for stream applications with potentially high and sporadic data rates. They also do not address how to ensure that computations are performed correctly.

VIII. Conclusion

Our goal in this paper was to develop a framework that would enable the creation of a decentralized market for outsourcing of streaming computation. We presented a protocol and showed that rational participants would follow the protocol and benefit from participating in the system, while participants that deviate from the protocol incur fines. While this does not prevent agents from operating maliciously and returning erroneous results it does ensure that costs exceed the benefits within the system in such cases. We do not handle scenarios when there are benefits exogenous to the system that make it worthwhile to misbehave; we would consider such scenarios in future work.

REFERENCES

[1] W. Yu, F. Liang, X. He, W. G. Hatcher, C. Lu, J. Lin, and X. Yang, "A survey on the edge computing for the internet of things," *IEEE access*, vol. 6, pp. 6900–6919, 2017.

[2] L. U. Khan, I. Yaqoob, N. H. Tran, S. M. A. Kazmi, T. N. Dang, and C. S. Hong, "Edge-computing-enabled smart cities: A comprehensive survey," *IEEE Internet of Things Journal*, vol. 7, no. 10, pp. 10 200–10 232, 2020.

[3] M. R. Schurgot, M. Wang, A. E. Conway, L. G. Greenwald, and P. D. Lebling, "A dispersed computing architecture for resource-centric computation and communication," *IEEE Communications Magazine*, vol. 57, no. 7, pp. 13–19, 2019.

[4] M. García-Valls, A. Dubey, and V. Botti, "Introducing the new paradigm of social dispersed computing: applications, technologies and challenges," *Journal of Systems Architecture*, vol. 91, pp. 83–102, 2018.

[5] R. van der Meulen, "What edge computing means for infrastructure and operations leaders," *Smarter with Gartner*, 2018.

[6] B. Varghese, N. Wang, S. Barbhuiya, P. Kilpatrick, and D. S. Nikolopoulos, "Challenges and opportunities in edge computing," in *2016 IEEE International Conference on Smart Cloud (SmartCloud)*, 2016, pp. 20–26.

[7] D. P. Anderson, "BOINC: A platform for volunteer computing," *Journal of Grid Computing*, vol. 18, no. 1, pp. 99–122, 2020.

[8] Cisco, "Cisco annual Internet report - cisco annual internet report (2018–2023) white paper," Cisco, 2018. [Online]. Available: https://www.cisco.com/c/en/us/solutions/collateral/executive-perspectives/annual-internet-report/white-paper-c11-741490.html

[9] D. P. Anderson, C. Christensen, and B. Allen, "Designing a runtime system for volunteer computing," in *Proceedings of the 2016 ACM/IEEE Conference on Supercomputing (SC)*. Piscataway, New Jersey: IEEE, 2006, pp. 33–33.

[10] L. Yuan, Q. He, S. Tan, B. Li, J. Yu, F. Chen, H. Jin, and Y. Yang, "Coopedge: A decentralized blockchain-based platform for cooperative edge computing," in *Proceedings of the Web Conference 2021*, ser. WWW '21. Association for Computing Machinery, 2021, p. 2245–2257.

[11] S. Eisele, T. Eghtesad, N. Troutman, A. Laszka, and A. Dubey, "Mechanisms for outsourcing computation via a decentralized market," in *Proceedings of the 14th ACM International Conference on Distributed and Event-based Systems*. New York, NY, USA: ACM, 2020, pp. 61–72.

[12] J. Teutsch and C. Reitwießner, "A scalable verification solution for blockchains," 2017. [Online]. Available: https://people.cs.uchicago.edu/teutsch/papers/truebit.pdf

[13] Mutable, "Mutable: the public edge cloud," 2020. [Online]. Available: https://mutable.io/

[14] M. Cherniack, H. Balakrishnan, M. Balazinska, D. Carney, U. Çetintemel, Y. Xing, and S. Zdonik, "Scalable distributed stream processing," 2003.

[15] "Code repository for this paper." [Online]. Available: https://github.com/TransactiveSCC/MODiCuM-Online

[16] "Apache Pulsar – An open-source distributed pub-sub messaging system." [Online]. Available: https://pulsar.apache.org/

[17] E. Solidity, "Solidity documentation," 2017.

[18] L. Baird and A. Luykx, "The hashgraph protocol: Efficient asynchronous bft for high-throughput distributed ledgers," in *2020 International Conference on Omni-layer Intelligent Systems (COINS)*, 2020, pp. 1–7.

[19] A. Singh, K. Click, R. M. Parizi, Q. Zhang, A. Dehghantanha, and K.-K. R. Choo, "Sidechain technologies in blockchain networks: An examination and state-of-the-art review," *Journal of Network and Computer Applications*, vol. 149, p. 102471, 2020.

[20] J. E. Hopcroft and R. M. Karp, "An $n^{5/2}$ algorithm for maximum matchings in bipartite graphs," *SIAM Journal on Computing*, vol. 2, no. 4, pp. 225–231, 1973.

[21] A. Nuzzolo, U. Crisalli, L. Rosati, and A. Ibeas, "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)*. Piscataway, New Jersey: IEEE, 2013, pp. 1894–1899.

[22] M. Wilbur, A. Ayman, A. Ouyang, V. Poon, R. Kabir, A. Vadali, P. Pugliese, D. Freudberg, A. Laszka, and A. Dubey, "Impact of covid-19 on public transit accessibility and ridership," 2020.

[23] "ETH Gas Station," https://legacy.ethgasstation.info/calculatorTxV.php.

[24] C. Sguanci, R. Spatafora, and A. M. Vergani, "Layer 2 blockchain scaling: a survey," 2021.

[25] "Gas fees calculator," https://www.cryptoneur.xyz/gas-fees-calculator.

[26] P. Bonacquisto, G. Di Modica, G. Petralia, and O. Tomarchio, "A procurement auction market to trade residual cloud computing capacity," *IEEE Transactions on Cloud Computing*, vol. 3, no. 3, pp. 345–357, 2014.

[27] M. Cherniack, H. Balakrishnan, M. Balazinska, D. Carney, U. Cetintemel, Y. Xing, and S. B. Zdonik, "Scalable distributed stream processing." in *CIDR*, vol. 3, 2003, pp. 257–268.

[28] C. Dong, Y. Wang, A. Aldweesh, P. McCorry, and A. van Moorsel, "Betrayal, Distrust, and Rationality: Smart Counter-Collusion Contracts for Verifiable Cloud Computing," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS)*. ACM, 2017, pp. 211–227.