applied sciences

MDPI

*Article*

# Trusted Electronic Contract for Enabling Peer-to-Peer HPC Resource Sharing

**Kajornsak Piyoungkorn, Siriboon Chaisawat and Chalee Vorakulpipat ***

Information Security Research Team, National Electronics and Computer Technology Center, Khlong Nueng 12120, Pathum Thani, Thailand; kajornsak.piyoungkorn@nectec.or.th (K.P.); siriboon.chaisawat@nectec.or.th (S.C.)

* Correspondence: chalee.vorakulpipat@nectec.or.th

**Abstract:** With the growing need for HPC resource usage in Thailand, this study aims to foster the creation of an HPC resource sharing ecosystem based on available in-house computing infrastructure. The model of computing resource sharing based on blockchain technology is presented for bridging communication between multiple clusters of HPC systems. The use of blockchain technology allows states among HPC systems to be synchronized and extends capabilities in enforcing governing rules. A smart contract was deployed on the blockchain network to enable users to request computing resources. Upon a request being made, a matching scheme performs the automatic selection of a suitable cluster based on current cluster utilization data and distance from users. Since users and clusters are anonymized from each other, a trusted payment scheme and permission access control are presented to assure both parties. As the system leverages off-chined and on-chained data exchange to carry out the operation, the secure gateway is proposed to mitigate technical difficulty from the client's perspective and ensure information is securely flowing to and from legitimate actors. The result of this work ensures HPC service providers can maximize the utilization of their resources and monetize idle computing time, while users can access demanded resources conveniently and pay at a reasonable price.

**Keywords:** high-performance computing; blockchain; decentralized network



**Citation:** Piyoungkorn, K.; Chaisawat, S.; Vorakulpipat, C. Trusted Electronic Contract for Enabling Peer-to-Peer HPC Resource Sharing. *Appl. Sci.* **2022**, *12*, 5153. https://doi.org/10.3390/app12105153

Academic Editors: Nasro Min-Allah and Ubaid Abbasi

Received: 6 April 2022
Accepted: 25 April 2022
Published: 20 May 2022



## 1. Introduction

High-performance computing (HPC) was introduced for research advancement in Thailand in 2007. Due to the niche in demand for intensive computation, applications of HPC are limited to some research areas, and most of the users are researchers, university professors, students, and a few experimental projects from the private sector. Currently, the main research fields that leverage HPC in Thailand include: synthesis and modeling in biopharmaceutical research, enhancing efficiency in energy supply, natural disasters, and environmental impact modeling and prediction (e.g., flooding, air quality).

The National e-Science Infrastructure Consortium was established to serve needs in the computational infrastructure of HPC systems, high-capacity storage systems, and network backbone at the national scale. Initially, the project aimed to support research in three main fields comprising (a) high energy particle physics, (b) computational science and engineering and (c) computer science and engineering. Displayed in Figure 1, statistics of research projects submitted to the e-Science service from the year 2010 to 2022 can be categorized into five research topics.

Due to huge investment and maintenance costs, as well as a limited number of trained personnel, most of the HPC services are supported by the government and intended to provide specifically to the collaborative research institutes and non-commercialized partner organizations. Nevertheless, with the emergence of big data analytics and artificial intelligence, many private companies or individuals have begun to pay more attention

to the value of data analytics. With the increase in the number of data-driven business models, HPC has started to gain demand for performing complex computations to extract data insight. To efficiently utilize in-house resources in response to the growing need, the providing scale of HPC in Thailand should be expanded to support new user groups with the means to facilitate ease of access in order to support nationwide innovation creation from all economic sectors. The model of computing resources sharing is a challenging topic that this study will explore. The goal is to create a unified network of computing resources regardless of the heterogeneity of the underlying infrastructure HPC services. Clients can easily make a request to the system for the type and amount of resources suitable for their job's requirements. The system will then help to find the best matches for HPC services, allowing clients to get the resource that best suits their tasks at a reasonable price with the service assurance guaranteed. On the other hand, the system enables HPC providers to benefit from their resource's idle time. With this freely competitive model, HPC providers are encouraged to consistently improve their service quality.

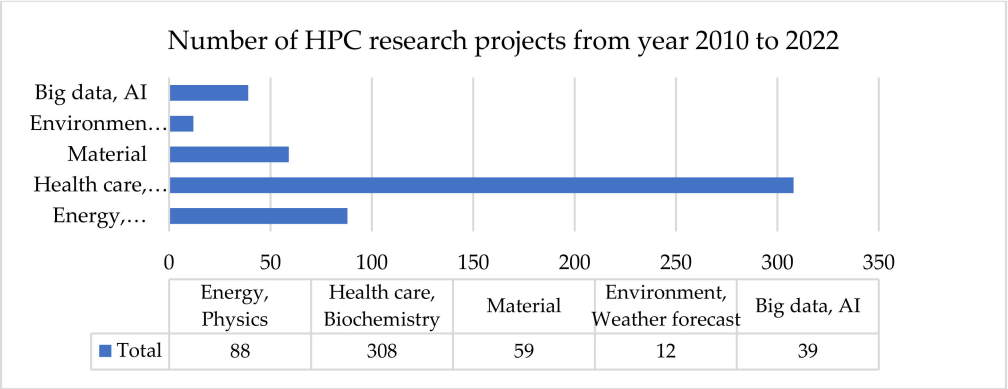



**Figure 1.** Statistics of Job Submission to e-Science Service Grouped by Research Topic.

The goal of this study is to promote the creation of a computing resource sharing ecosystem based on in-house available infrastructure. Our contributions to this work are as follows:

1. Present architecture design of trusted network based on blockchain technology, smart contract logic, and role-based permission for restricting entities' actions on network resources and function calls.
2. Present a secure gateway service for enabling communication between HPC clusters and blockchain. It facilitates authentication, initiating a secure connection, and asynchronously performing actions according to blockchain events.
3. Present a system monitoring service that periodically sends out updates on the cluster's data states to the blockchain (i.e., cluster health, job status). The service will help monitor HPC's states without interfering with the operational pipeline.

The sections in this paper are organized as follows. Section 2 reviews related works on applying blockchain technology to HPC and explores challenges and difficulties in adopting a decentralized sharing model in Thailand. Section 3 describes the architecture design and key modules in the proposed system. In Section 4, implementation of the design is presented followed by performance evaluation and security analysis in Section 5. The final section, Section 6, summarizes the study and findings.

## 2. Literature Review

Current research studies in HPC areas have been focusing on enhancing the efficiency of resource management. To increase HPC utilization and improve response time, various implementations of tasks scheduling algorithms are proposed [1]. To improve the performance of the scheduler, a study [2] proposes a method for predicting job waiting time, allowing the scheduler to make use of the information in decision making. In addition, a

resource reservation scheme [3] is proposed to enable advance schedule planning. Despite the aid from task management schemes, inefficient resource usage remains an issue as many HPC providers let users define the specification and amount of resources needed to carry out their tasks in which, most of the time, the reserve amount is imprecise compared to the actual usage at runtime. To mitigate the problem, an automated monitoring system [4] for detecting jobs with inefficient resource consumption is introduced by analyzing the relationship between the parent process and its spawning child processes [5]. Apart from job and queue management, computing performance and processing time can be improved through modifying operational mechanisms of underlying HPC components, such as kernel modification on a single compute node's Linux scheduler [6,7]. Recent studies have also focused on introducing machine learning and AI techniques to enable system administrators to better understand the behavior of HPC through analyzing the relationship between operational elements, e.g., determining an influential factor between application and resources usage at runtime [8]. Another research topic in HPC is related to improving security. Through the presence of application layer access control, anomaly user behavior can be detected through log data collection and analysis [9].

Looking at the demand in HPC areas in Thailand, up to present, most of the services are supported by government agencies and mainly provide to non-commercial research innovations from education and research sectors. Nevertheless, with the recent launch of open government data in Thailand [10], over 6000 datasets (e.g., agricultural, logistic, and transportation data) were made publicly available, encouraging business transformation into a data-driven model. As a result, a greater number of individuals and companies began to pay more attention to using HPC as a crucial means to extract insightful value from data. This change can obviously be noticed from the usage statistics of the e-Science service, which shows that the demand for HPC in AI and Big Data areas has been growing significantly in recent years. Unfortunately, the current process for requesting HPC resources requires some paperwork and a pre-screening process. The goals are to ensure that the users are from the trusted institute and to allow the service provider to anticipate an incoming workload. Such processes are not attractive to companies or individuals, who do not have creditability and/or intend to use HPC for computing just a single or a few jobs, to fill out an intention form.

Currently, there are few studies on creating a foundational structure for sharing computational resources. In the widely known 'Grid System', jobs are scheduled to the selected HPC frontend and the results are returned after execution is done. The HPC systems under grid are required to have a homogenous environment (i.e., scheduler brand and versions, installed applications). The grid system, therefore, is suitable to run predictable tasks in a specific research area. In contrast, our objectives aim to support access from users from diverse fields; therefore, not all HPC systems possess all the needed requirements (e.g., software license, versions, dependencies). Therefore, the creation of a resource sharing system can enable tasks from different application areas to schedule for running on HPC systems with desirable environments. In addition, creating the system can offer a number of managerial benefits to HPC providers, such as the ability to audit and trace the sources of anomaly actions, and the capability of making use of statistical data on resource usage and trends for making a rational decision for future resource supply investment or purchasing high-demand software licenses to attract users. Currently, however, studies on creating such a system in the HPC domain are still limited in number.

Integrating blockchain with HPC aims to solve trust issues between public users and HPC resource providers by automating the process of digital agreement creation and regulating the actions with no central authorities required. Introduced by Satoshi Nakamoto, a practical peer-to-peer model of the electronic cash system emerged [11]. With the decentralized computation and management in blockchain technology, data security and transparency in business activities can be enhanced [12]. Transactions submitted to the network will be collaboratively validated against a *smart contract*, which is installed on all nodes. Only agreed actions will be propagated to the network, allowing all participants

to apply the changes to their local records. Regardless of states being replicated and locally maintained by individuals, consistency in network states can still be preserved through cryptographic mechanisms and efficient data structures (i.e., Merkle tree [13]) that allow the data validation process to be carried out with small time complexity. Instead of relying on a central authority to manage states, blockchain uses decentralized consensus algorithms to define how agreement in the network is reached. Examples of the algorithms are Proof of Work (PoW) [14], Proof of Stake (PoS) [15], and Practical Byzantine Fault Tolerance (PBFT) [16]. These benefits of blockchain expedite the development of various e-government services in which trusted data sharing among legitimate stakeholders is the primary concern. Examples of blockchain-based projects that have been implemented at a national scale are e-Transcript [17] (education), DLT scripless bond [18] and Letter of Guarantee on Blockchain [19] (finance), e-Referral system [20] (health care), National Energy Trading Platform [21] (energy), TradeLens [22] (logistic & supply chain).

Nevertheless, introducing blockchain to HPC is more challenging compared to integrating it with application software. The solution design must present an approach for enabling communication between the application layer and OS layer. In contrast to HPC, blockchain's processing is asynchronous and slow in speed. To make sure data exchange between these two technologies is correct and remains synchronized, there must exist an auxiliary module for enabling system interoperability by bridging communication differences and handling unexpected error. As a result, the use of blockchain in HPC systems is still limited in the number of applications. Previous studies have emphasized presenting fault-toleration schemes to promote resiliency in Message Passing Interface (MPI) by proposing a blockchain consensus protocol that is optimized for HPC compatibility [23,24]. To enable data consistency across shared storages in HPC systems, a study also focused on using blockchain technology for states synchronization [23]. With the ability of blockchain in preserving ownership [25], the technology was applied to improve data provenance in the HPC system by validating the source of data generated before being stored in a high-speed parallel file storage system [26]. The arrival of the data-driven era has expedited the generation of a large volume of sparse data. As a result, demand for intensive computing power is growing. On the contrary, people start to pay attention to the security and ownership control of their data. Trust, therefore, becomes a key requirement in designing large-scaled data computing [27].

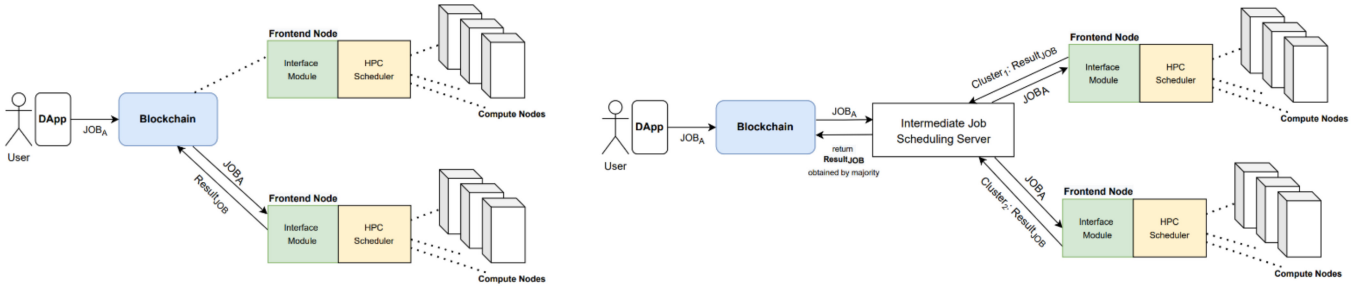
## 3. Proposed Design

This section will be divided into three parts. The first part focuses on system architecture design. The next part presents the details of smart contracts (including business logic and access control policy) and data exchange patterns among network participants. The final part discusses the design of an interface module that enables off-chained clients to participate in network activities.

### *3.1. Overview of System Architecture and Components in Trusted Computing Resource Sharing*

To create an automated trust system, every action on the systems must be ensured to comply with the network agreement. Two key players in this architecture are users, who demand computing resources, and service providers, who are willing to offer HPC service to the public. Two possible models can be implemented according to setting requirements. Users are connected to the blockchain via a decentralized application (dApp), where all activities, such as requesting resources and getting notifications for change in the summited jobs' status, are carried out automatically by having a client-side script make a request to a smart contract. For HPC, the frontend node requires the installation of an *Interface Module* to enable communication between blockchain and HPC. By having the module installed on HPC, any updates on blockchain or cluster state will automatically sync which benefits clusters in obtaining a symmetrical view of network states. (Details of this module will be elaborated in the subsequent section).

Figure 2a: In this model, all participants in the network are trusted. This can be achieved by having a thorough consideration process; for example, the HPC system must go through stages of historical creditability screening, verifying SLA documents, etc., prior to registering the computing cluster to the network.




(**a**) (**b**)

**Figure 2.** Computing Resources Sharing System Architecture. (**a**) Trusted System Model; (**b**) Untrustworthy System Model.

Figure 2b: In this model, we allow the registration process of HPC systems and users to be less strict. This approach encourages new service providers who may have no creditability profile to freely participate in the market. Since we assume no entity can be trusted, there is a possibility that HPC systems deliver corrupted computation results or correct computational results get denied by dishonest users. Two main mechanisms to resolve the issues are:

1. Majority voting-replicate job execution on multiple compute workers (let us say, $n$ workers). The results obtained by the majority are assumed to be correct. The drawback of the scheme is intensive resource consumption as the computational resources needed are multiplied by a factor of $n$.
2. Result validation—in this option, a user must implement a module to validate if the results he/she obtained are correct. However, the implementation of such a scheme is complex and requires efforts from users to some extent.

Several research efforts have contributed to the design of a majority voting scheme. In 2002, a dynamic voting scheme [28] is presented by allowing the number of participating workers for a particular job execution to vary according to the level of confidence. The scheduler is responsible for deciding the number of replicated factors (i.e., number of computing nodes) during execution time. This scheme reduces time in reaching consensus and ensures that only the necessary amount of resources to attain confidence are used. Developed upon the previous concept, iExec [29] has proposed a tweak of design that includes a monetary factor to the model. The goal is to offer an incentive (in the form of digital currency) to honest workers. The approach is to let the worker attach a disposable personal identifier along with its computing result. If the result is correct and the worker can present the proof of ownership, it will then receive an incentive according to the commitment fund.

### *3.2. Smart Contract Design and Communication Flow*

Smart Contract is a business logic that is deployed on blockchain nodes to govern network activities. In this study, our smart contract composes of the main logic (i.e, ResourceSharingContract) and four types of objects (i.e., User, HPC System, Job, Cluster State), in which the functions, attributes, and the relationship between components are illustrated in Figure 3.

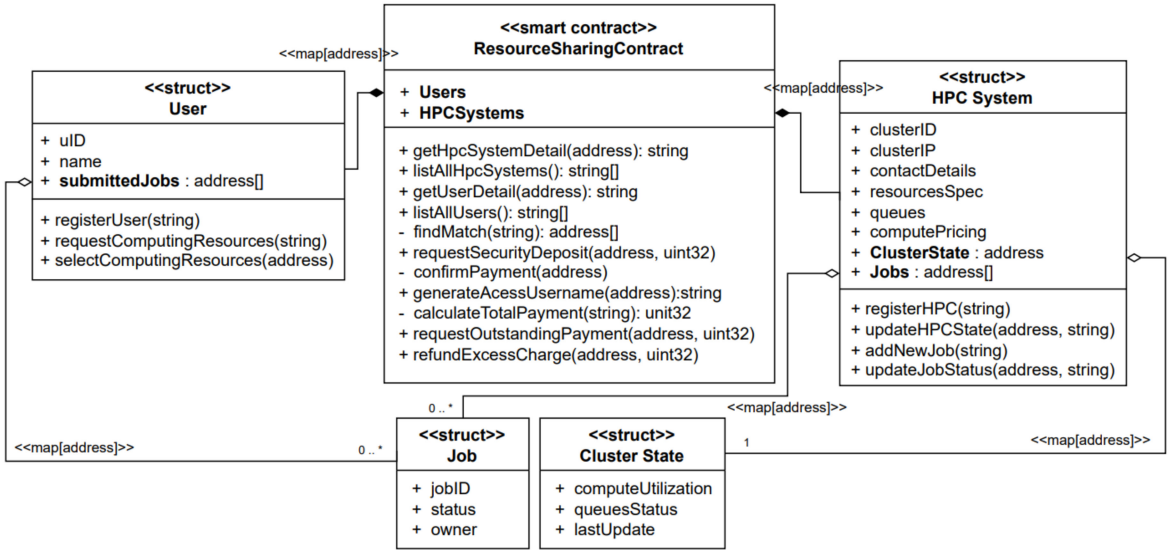



**Figure 3.** Class Diagram of Resource Sharing Smart Contract.

Displayed in Figure 4, communication flow can be segregated into main functional tasks as follow:

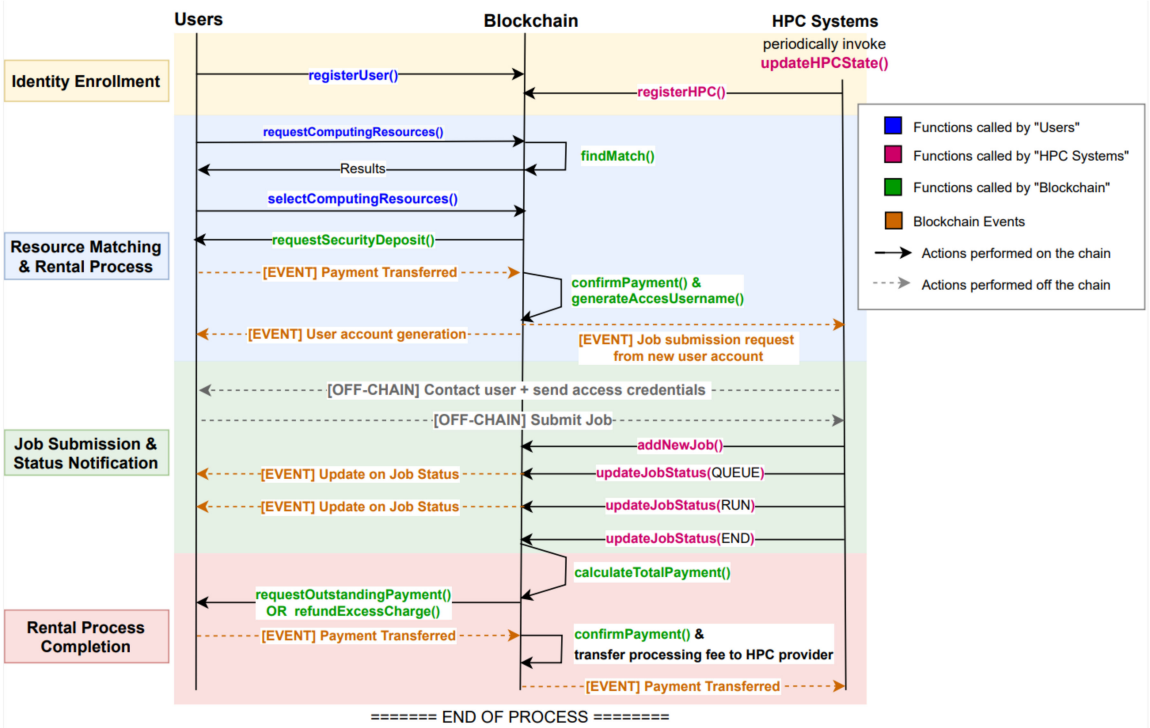



**Figure 4.** Communication Flow.

1. **Identity enrolment** For the first time entering the system, clients (users and HPC systems) are required to go through a registration process which could be carried out manually or automatically depending on the level of strictness of the onboarding process. Once the consideration process is passed, clients will be assigned to a specific role and cryptographic credentials corresponding to identity will be generated. The private key will be stored locally on client machines, while public credentials will be sent to the blockchain. These cryptographic materials will be used in signing transactions produced by clients, allowing the blockchain to validate if the transaction's

owner is permitted before executing the requested functions. Displayed in Table 1, role-based permission is defined on network resources and business logic execution.

2. **Resource Matching** The system needs to ensure users are matched to the suitable resources at satisfactory rental prices and providers can benefit from renting computing resources. On user requests for job submission, the blockchain will perform searching through the HPC system information (obtained during the HPC registration process (Figure 5a) to find a set of candidates based on the user's requirements (e.g., resource spec, programs, etc.). The result set will be returned along with the latest cluster state (Figure 5b) and rental price, which is slightly adjusted according to the distance between user location and the HPC system to encourage utilizing local resources. Users can select the options based solely on system information (without revealing provider names) to promote competitiveness and encourage new providers to compete in the ecosystem. Pseudocode for finding a candidate set of HPC systems is presented as follows:

**Table 1.** Permission on Network Resources and Smart Contract Functions Based on User's Role.

| Blockchain Resources / User Role | Network Resources | | | Smart Contract Functions |
|---|---|---|---|---|
| | **Transactions** | **System Activities** | **Event Hub** | |
| HPC provider | Able to submit READ & WRITE transactions | - | Able to subscribe to events | Functions specified in "HPC System" Object in Figure 3 |
| User | Able to submit READ & WRITE transactions | - | Able to subscribe to events | Functions specified in "User" Object in Figure 3 |
| Admin | Able to submit READ-only transactions | Able to register new clients | - | getHpcSystemDetail(), getUserDetail(), listAllHpcSystems(),listAllUsers() |
| System Node | - | Able to perform system action | Able to publish events | Functions specified in "ResourceSharingContract" in Figure 3 |

Let $DR_a$ represent computing resources and program requirement that are demanded by a user $a$ (JSON data displayed in Figure 5c) and $PR_A$ represent information regarding provided computing resources owned by cluster $A$ (JSON data displayed in Figure 5a), the pseudo code of the *findResourceMatching()* algorithm (Algorithm 1) is defined as follows

**Algorithm 1. Pseudocode of *findResourceMatching()*.**
**Return Lists of Resources That Match to User's Job Requirement**

**Input: Object containing desired resources spec from user ($DR_{user}$)**
1: MatchedSpec ← query list of matched_cluster's IDs in which $PR_{<matched\_cluster>}$ exists some *queues* that has max_cores > DRuser.cores and gpu == DRuser.require_gpu
2: Candidates = **a subset of MatchedSpec** in which DRuser.program ∈ $PR_{<matched\_cluster>}$.programs
3: if Candidates == ø then
4: for all IDs ∈ MatchedSpec do
5: if $PR_{ID}$ .willingToInstallAdditionalPrograms then
6: Candidates.push(*ID*)
7: return Candidates

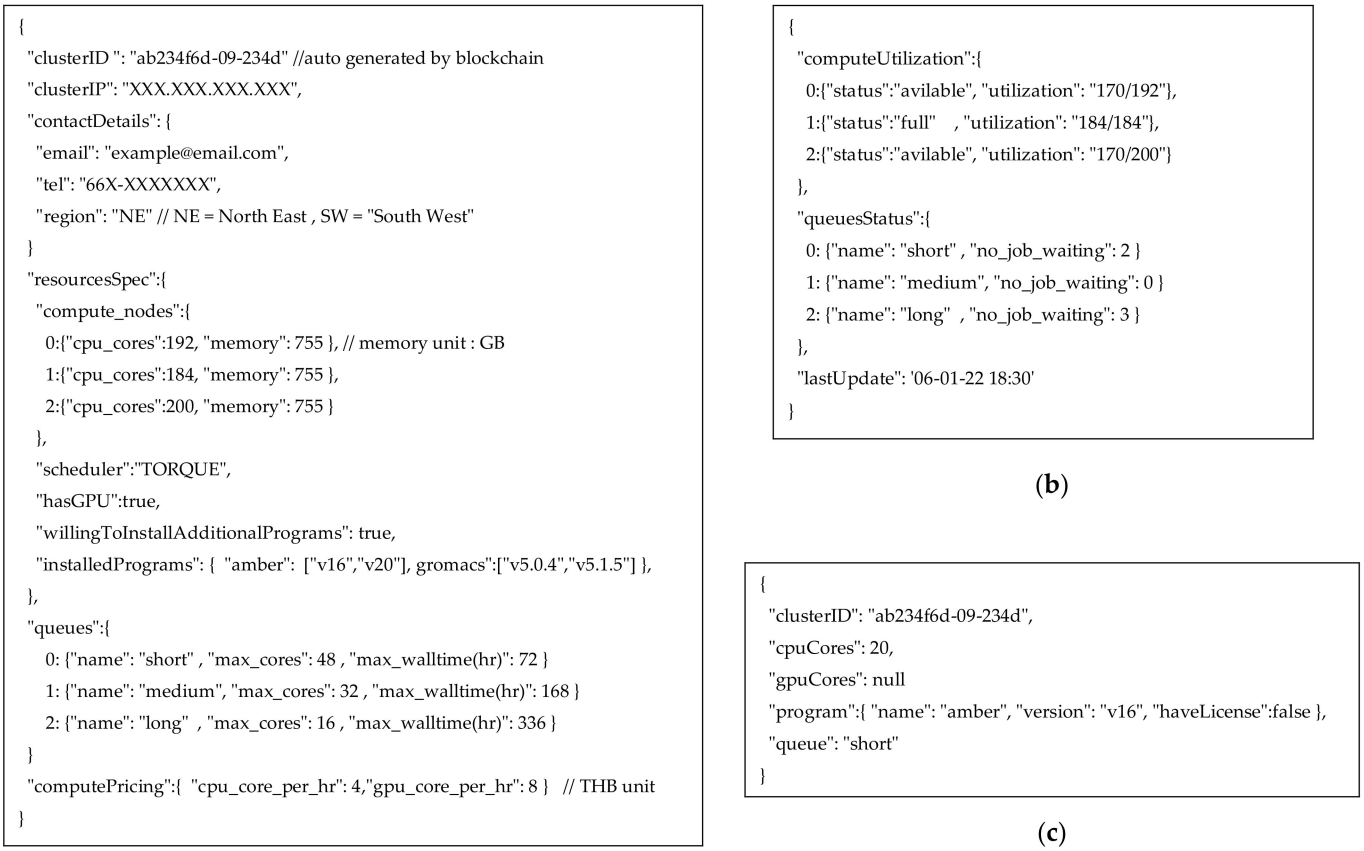



**Figure 5.** Data Object in JSON format. (**a**) HPC System Object (stored on blockchain state); (**b**) Cluster State Object (stored on blockchain state); (**c**) User's Job Submission Request Payload.

Let $HS_A$ denote the latest update on the utilization state of cluster $A$, pseudocode of *displayResourcesAndRentalPrice*() algorithm (Algorithm 2) is defined as follows

**Algorithm 2. Pseudocode of *displayResourcesAndRentalPrice*()**
**Return List of Clusters (with the Adjusted Rental Price and Utilization State) That Match to User's Job**

**Input: Object containing desired resources spec from user ($DR_{user}$), current user location**
1: results = []
2: Candidates = *findResourceMatching*($DR_{user}$)
3: for all *IDs* ∈ Candidates do
4: differentDistance = calculate approximated distance between 2 IPs (user and $PR_{ID}$.IP)
5: [adjustedCPUprice, adjustedGPUprice] = *weighPriceByDistance* (differentDistance , [$PR_{ID}$.rate.cpu_core_per_hr, $PR_{ID}$.rate.gpu_core_per_hr(*optional)]) // adjust price based on distance
6: $HS_{ID}$ ← query *HealthState* Object by cluster ID
7: results.push({ID: $PR_{ID}$.clusterID, queues:{ available: $PR_{ID}$.queues, current_state: $HS_{ID}$.queues], rate: {cpu: adjustedCPUprice, gpu: adjustedGPUprice }})
8: return results

3. **Rental Process** Once the user chooses an HPC system and a preferred queue type from the candidate list, he/she then composes and submits a transactional request to the blockchain. The user will then be requested to pay a security deposit which calculated from

security deposit = (requested cpu cores * *cpu_core_per_hr*) * 30% of queue's *max_walltime*

* same calculation apply for requesting GPU cores. Note that *cpu_core_per_hr*, *gpu_core_per_hr* (optional) and *max_walltime* are retrieved from *resourceSpec* attribute (JSON data displayed in Figure 5a) in *HPC System* object (Figure 3) from blockchain state. This amount of money will be held by the blockchain as a security assurance for both parties until the job has been completely processed. The system will invoke the *generateAcessUsername()* function which will generate a globally unique username for the user. This username along with the user's contact address will also be sent to the HPC administrator for generating login credentials to access the HPC cluster. Users can expect to receive an access passcode from the service provider via provided contact channels.

### 3.3. Job Submission & Status Notification

After the access passcode is sent, the user can now access the HPC system and submit his/her job. Upon a job being added to the queue or a change in a job's status (i.e., null → QUEUE, QUEUE → RUN, RUN → END), the events will automatically be detected and composed into a blockchain transaction. Once the transaction is committed, the user will be notified by the *events* generated by a smart contract. These functionalities are automatically performed by the interface module, in which the design and mechanism will be elaborated in the subsequent section.

#### Design of Interface Module

Since some operations have to deal with large data input size and/or required confidentiality preservation, performing such actions off the chain while synchronizing small-sized checkpoint data on-chain is one of the efficient approaches used in this research work. To ensure HPC activities are automatically synced with blockchain, an interface module (Figure 6) is implemented to facilitate the exchange of states between blockchain and HPC service. To enable portability and isolation from underlying infrastructure (environment on host machines), this module is operated within its own container separated from the rest of the system. Two major components of the module comprise the following:

1. *Blockchain Gateway Service*—is a service that facilitates data exchange between HPC and blockchain. Its key functionalities are composing, signing, and submitting transactions to the blockchain on behalf of the HPC cluster. It also subscribes to blockchain's event hub to listen for smart contract events. Upon a new event arrival, it will send a request to the system monitoring service to perform system actions corresponding to the event types. It uses a network connection profile* for discovering peers in the network and uses cryptographic materials* for presenting eligibility in performing on-chained actions. (* these files are available in a mounted directory).
2. *System Monitoring Service*—helps monitor changes in system states by spawning threads to monitor attentive processes. It also opens a port for listening to the incoming requests. There are two types of operations performed in this module. The first is an asynchronous operation that will execute once there is an incoming request from the blockchain gateway service. For example, blockchain gateway service has received an event notifying new job submission from blockchain, it will then call */addJobMonitoring* endpoint on the port listening by system monitoring service. The service then executes the command from bash scripts* to perform system action and return the result back correspondingly. The second type is scheduling operation. For example, for all jobs that are submitted to the system, the service will periodically perform checking if the states of jobs are changed (illustrated in Figure 7). If there are, the changes will be packed and sent to the Blockchain Gateway Service to update the job's states on the blockchain.

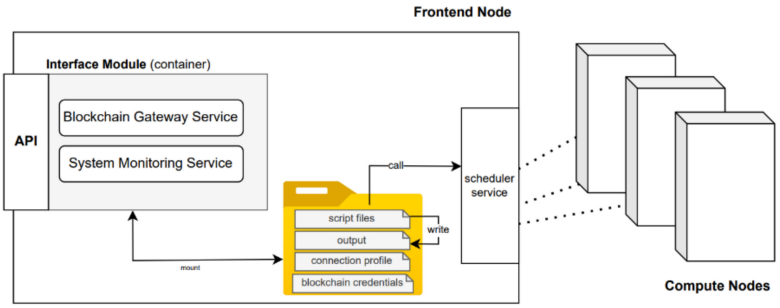



**Figure 6.** Interface Module.

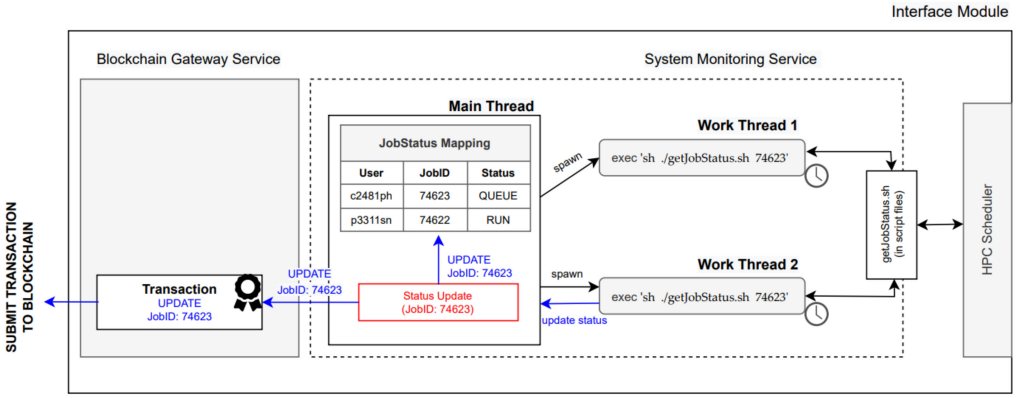



**Figure 7.** Job Monitoring Operation in Interface Module.

As the interface module is operated in an isolated container, to ensure operation on the host machine can be carried out, the container needs permission access to local files via the volume mount path. Three types of files are stored in this mounted directory.

1. *Action Scripts*—These are the scripts that are called by System Monitoring Service. It helps perform querying on the scheduler to attain the overall status of HPC components as well as job and queue state. Regardless of the differences in scheduler commands that vary according to scheduler brands, we define a standard structure format for output (i.e, stdout) produced by scripts to ensure the results are understandable by other dependent services.
2. *Blockchain's Crypto Materials*—Used by Blockchain Gateway Service, crypto materials (e.g., public key, private key, and certificates) are files that were used to represent HPC identity in blockchain space. Any transaction generated by the cluster must be digitally signed before being published out to the network. As mentioned earlier, these credentials are registered to a specific role according to RBAC policy to prohibit an entity from performing irrelevant or unauthorized on-chained actions.
3. *Network Connection Profile*—contains network topology, allows the cluster to discover and be discovered by other peers in the blockchain network.

## 4. Implementation

### *4.1. Blockchain Network Setup*

Hyperledger Fabric (HLF) [30] version 1.4 was used for setting up a permissioned blockchain network. To ensure the network can tolerate node crashes, a multi-node Ordering service [31] with Raft protocol [32] was used. The network was deployed on two settings: single-host setting and multi-host setting. For a single host setting, the system was operated on Ubuntu 16.04 LTS machine with 2 CPU cores 2.80 GHz, and 12 GB of RAM. Each network component was deployed as a Docker [33] container and communicates with others inside the Docker internal network. For the multi-host setting, we used four virtual machines (each has 2 CPU cores and 4 GB of RAM, Ubuntu 18.04) to create a Kubernetes cluster [34] of one master node and three worker nodes.

Role-based access control (RBAC) was configured according to the condition specified in Table 1 by overriding default Hyperledger Fabric's Access Control List. Therefore, only a specified set of roles were capable of accessing specific types of network resources (i.e., business logic chaincode*, system chaincode, and event service). In addition, to enforce another layer of control on the business logic level, built-in attributed-based access control (ABAC) was implemented on a smart contract by importing a class 'ClientIdentity' from the 'fabric-shim' library to perform credential checking on transactions prior to proceeding operation on the requested functions.

(* note that *chaincode* is the term used for representing smart contract in HLF)

**(Smart Contract –** function**)**

```
1: const ClientIdentity = require('fabric-shim').ClientIdentity;
2: function requestComputingResources(arguments) {
3:         let cid = new ClientIdentity(stub);
4:         if (cid.assertAttributeValue('node_name', 'HPC_user){    // assert if the requester is user of HPC service
5:                 // perform matching
6:          } else { throw new Error('Invoking Entity is Unauthorized');}
7: }
```

### *4.2. Implementation of Interface Module*

The module along with its dependencies was packed and built as a Docker image. Container runtime was installed on the HPC cluster, consisting of a single frontend node with Intel(R) Xeon(R) Silver 4110 CPU 2.10 GHz 16Cores and 64 GB of memory, five compute nodes each with Intel(R) Xeon(R) Platinum 8168 CPU 2.70 GHz 192 Cores and 768 GB of memory. To ensure the module was seamlessly operated regardless of differences in the host machine environment, the module was deployed as a Docker container in which the Docker volume was mounted to the path that kept blockchain's cryptographic credentials, network connection profile, and action scripts.

- Sub-module #1: Blockchain Gateway Service

The service was implemented in NodeJS using Hyperledger Fabric SDK for establishing a connection with the blockchain. Once the connection is created, the script will automatically make a request to the Certificate Authority node for registering the HPC system's identity. If registration is successful, cryptographic credentials will be generated and saved to a local folder named 'wallet'. Below displays the script for requesting identity generation.

```
1: const enrollment = await ca.enroll({ enrollmentID: <username>, enrollmentSecret: <secret>});          // request enrolment to CA
2: const identity = X509WalletMixin.createIdentity(<clusterMSP>, enrollment.certificate, enrollment.key.toBytes());
3: wallet.import(<username>, identity);                                                                  // import identity to wallet
```

- Sub-module #2: System Monitoring Service

This sub-module acts as a middleman that helps querying, formatting, and passing system data to the Blockchain Gateway Service once there is a request for updated status information. The goal of this module is to monitor users' jobs status. Once a new job has been submitted to the HPC service, the system monitoring service will spawn new threads to monitor the job in particular. To implement the module, the code was written with the *worker_thread* module in NodeJS. The code executed in the main thread (app.js) helped monitor and maintain the latest update of all users' jobs status. In addition, HTTP REST API endpoints were exposed to allow the blockchain gateway service to add a new monitored process. For the worker thread (worker.js), the script will periodically call *getJobStatus.sh* and pass the interested *jobID* as an input parameter. The script then performs querying on a scheduler and waits for the return status value of the job. If the thread finds the status is changed (by comparing with the variable *currentStatus*), it then sends a notification update to its parent (main thread).

**(app.js)**

```
1: var listOfActiveJobs = [ {"jobID": "57714", "status": "RUN", "owner": "kajornsak", “isMonitored”: true }, ….]   // job monitoring list
2: app.post('/addJobMonitoring', (req, res) => {                                                  // endpoint for requesting adding new job
3:      listOfActiveJobs.push({"jobID": req.body.job_id, "status": null, owner": req.body.username, “isMonitored”: false })
4:      res.send(`jobID "${jobID}" is being added`);    });
5: function createMonitoringWorker(jobID, status) {
6:   let worker = new Worker('./worker.js', { workerData: { jobID: jobID, currentStatus: status } });          // create job monitoring worker
7:   worker.once("message", result => {                                                                       // worker return an update in job status
8:          let newStatus = result[jobID]                                                                     // sample result { "jid01234": "QUEUE" }
9:          if(newStatus === "END"){              // if the job’s computing is done, send update to blockchain service and remove job from monitor list
10:                 axios.post('localhost:5000/updateJobStatus', result).then((response)=>{ console.log(response.data) })
11:                 listOfActiveJobs = listOfActiveJobs.filter(obj => obj["jobID"] != jobID);
12:           }else{                                  // if the job’s status is changed, send update to blockchain service and update the local list of job status
13:                 axios.post('localhost:5000/updateJobStatus', result).then((response)=>{ console.log(response.data) })
14:                 listOfActiveJobs = listOfActiveJobs.map(obj => obj["jobID"] === jobID? { ...obj, status: newStatus} : obj );
15:           }});
16: }
17: while (listOfActiveJobs.length !== 0) { listOfActiveJobs.map((job) => { if ( ! job. isMonitored) { createMonitoringWorker(job.jobID, job.status) }}) }
18: app.listen(3000, () => { console.log(`Listening at http://localhost:3000`); });
```

**(worker.js)**

```
1: function getJobStatusUpdate(){
2:    exec("sh ~/getJobStatus.sh " + workerData.jobID, function (err, stdout, stderr) {
3:       stdout.on('data', (data) => {
4:          let latestStatus = data
5:          if (latestStatus !== workerData.currentStatus) {
6:                parentPort.postMessage({"jobID": workerData.jobID, "new_status": latestStatus}) // all jobs of the user with the states
7:           }else{ setTimeout(getJobStatusUpdate, 3600000); }
8:       }); }); }
9: getJobStatusUpdate()
```

- Script Files

These bash scripts contain a sequence of Linux commands for directly interacting with a scheduler. The script needs to be customized according to the scheduler’s brand—in our setting, the TORQUE scheduler was used. The output is then converted into the pre-agreed format (Figure 5b) which is understandable by the System Monitoring Service. A code snippet is displayed below.

```
1: #!/bin/bash                                                    (getResourceUtilization.sh – for TORQUE scheduler)
2: PBS_PROGRAM="/opt/torque/bin"; USER_NAME=$(whoami); COMPUTE=Compute-name; # Select scheduler, Specify username and compute node name
3: for i in {0..x}                                                # Loop through all compute nodes (0-x)
4: do
# 1. Get raw output of CPU cores usage per job
5:      mapfile -t jobs < <($PBS_PROGRAM/pbsnodes $COMPUTE-$i-ib | grep "jobs =");   # Output: "core_start - core_end /<PID>.<compute_node_name>"
6:      echo $(jobs[0]) | sed -e 's/… Formatting raw data until obtain array of jobs' cores …/$COMPUTE-$i.raw > ./$COMPUTE-$i.cores;
# 2. Output: [ Job1's core_start–Job1's core_end , ….]
7:      mapfile -t state < <($PBS_PROGRAM/pbsnodes $COMPUTE-$i-ib | grep "state =" | head -n 1 | awk '{print $3}');   # 3. Get node status
8:      mapfile -t np < <($PBS_PROGRAM/pbsnodes $COMPUTE-$i-ib | grep "np =" | awk '{print $3}');                      # 4. Get node's max CPU cores
9:      IFS=', ' read -r -a array <<< "$(cat ./$COMPUTE-$i.cores)"                                                     # Read jobs' cpu cores to array
10:     total_cores = 0;
11:     for j in ${!array[@]};                                    # Loop through elements in array   (ex. ["2","58", "68-69", "73-84"] )
12:             do
13:                     if [[ "$((array[$j]))" =~ [-] ]]; then    # CASE : Multiple cores   (ex. "73-84")
14:                             item=$((array[$j]));              # "73-84"
15:                             used_cores=$((item*(-1)+1));      # Turn string into math operation : 73-84 = -11 → (-11)*(-1) → 11 + 1
16:                             total_cores=$(( total_cores + used_cores));
17:                     else                                      # CASE : Single cores   (ex. "2")
18:                       total_cores =$(( total_cores +1));
19:                     fi
20:             done
21:     echo -e "$COMPUTE-0$i: : { "utilization": $total_cores/$(cat . /$COMPUTE-$i.max_cores) , "status": $(cat . /$COMPUTE-$i.status)}
22: done
```

## 5. Performance Evaluation and Security Analysis

The experiment was conducted on the blockchain network and interface module to measure the performance and perform security evaluation.

### *5.1. Performance Evaluation*

a. *Blockchain Network*

In our blockchain implementation, ordering service with Raft protocol was configured instead of other service types that are available in Hyperledger Fabric. To ensure that Raft presents the best performance compared to the others, we used Docker to set up a blockchain network comprising two organizations with a single peer node each. The test was set up on a machine with Ubuntu 16.04 LTS with 2 CPU cores 2.80 GHz and 12,288 MB of memory. Hyperledger Caliper [35], an open-source blockchain benchmark tool, was installed to measure network performance. Displayed in Table 2, *Raft* exhibited lower latency in both query and invoke transactions, compared to another multi-node ordering service type (i.e., *Kafka* [36]). Latency in transaction querying in multi-node *Raft* was close to a *Solo*, a single-node ordering service that is suggested to be used in only a development setting.

**Table 2.** Average Latency in Transaction Processing (Total transactions submitted = 300, Rate = 100 TPS *).

| Transaction Types | Ordering Service Type | | |
|---|---|---|---|
| | Single-Node | Multi-Node | |
| | Solo | Kafka | Raft |
| Query | 0.01s | 0.06s | 0.02s |
| Invoke | 0.34s | 0.85s | 0.68s |

* TPS = Transactions per second.

As mentioned in the implementation section, the blockchain network with the same number of components and topology was deployed on two settings, which are on single-host machines and multi-host machines. For the single-host setting, Hyperledger Caliper was deployed as a container within the Docker network. While in a multi-host setting, Hyperledger Caliper was deployed as a Kubernetes Pod. Table 3 shows the latency in transaction processing of both setups. Multi-host setting exhibited a larger delay which possibly resulted from external uncontrollable factors, such as network connection delays between machines, which can be improved by adjusting network configuration.

**Table 3.** Performance of Raft Network on Different Settings.

| Network Deployment | Query Transaction Latency | | | Invoke Transaction Latency | | |
|---|---|---|---|---|---|---|
| | Min | Max | Avg. | Min | Max | Avg. |
| Single Host | 0.01s | 0.05s | 0.01s | 0.57s | 0.88s | 0.68s |
| Multiple Hosts | 0.08s | 0.36s | 0.18s | 0.61s | 1.96s | 1.24s |

b *HPC Interface Module*

The interface module is one of the core parts of this system. Without it, communication between HPC and blockchain is not possible. Two key components in this module are the blockchain gateway service and the system monitoring service. As the blockchain gateway service can only be implemented in a single structural way following the Hyperledger Fabric implementation guideline, the experiment focused on only the system monitoring service in which the implementation can be tweaked to finetune performance.

Since the system monitoring service leverages the concept of multiple threading to perform monitoring on different tasks simultaneously, the test was conducted by measuring the total time of a single execution loop which is calculated as follows:

$$\begin{aligned} \textbf{\textit{Total Execution Time}} &= \textbf{\textit{Time to spawn a worker thread}}\ (T_{spawn}) \\ &+ \textbf{\textit{Execution time in worker thread}}\ (T_{exec_{worker}}) \\ &+ \textbf{\textit{Time for update on local state}}\ (T_{update\_state}) \end{aligned}$$

Note that the task executed on the worker thread $T_{exec_{worker}}$ can either be retrieving cluster health data (executes bash script *getResourceUtilization.sh*) or monitoring for changes in the job's state (execute bash script *getJobStatus.sh)*. In the case of monitoring a job's state, $T_{exec_{worker}}$ can vary according to the job's queueing time (which depends on previous jobs' execution time) and processing time. Therefore, in this experiment, we assumed that the job's initial status was RUN, and at the moment that monitoring started, the job's status changed to END. Thus, $T_{exec_{worker}}$, in this case, was the only duration that the worker executed/getJobStatus.sh plus the time that the status update message was returned to the main process. Table 4 displays the time for executing each sub-process in system monitoring services. After conducting five rounds of tests, the results showed that the execution time on the average of $T_{spawn}$ was 8.6839734 ms, $T_{exec_{worker}}$ in monitoring change in the job's status was 38.0384 ms, $T_{exec_{worker}}$ in retrieving current cluster utilization data was 667.663 ms, and $T_{update\_state}$ was 0.2911276 ms. By summing up the average value of $T_{spawn}$ , $T_{exec_{worker}}$ ***and*** $T_{update\_state}$ , the result shows the total execution time takes 46.9 ms for monitoring jobs and 676.6 ms for retrieving cluster state. These values are very small compared to the duration that a typical job normally takes to change its states which could take from seconds (e.g., job crashed: RUN → END) to weeks (e.g., job wait in the long-running queue: QUEUE → RUN). In addition, the amount of resources (e.g., CPU, memory) needed in the execution is very small and insignificant compared to the capability of HPC's frontend. The script also uses only memory for keeping track of state data, no read/write to the storage. Therefore, the inclusion of the module in the HPC systems will not impact the overall performance and system workload or interrupt the core operational process of HPC.

### *5.2. Security Analysis*

1. *Access Control on Blockchain Network Resources*

Two permission layers were configured to enhance security on the blockchain. The first was an access control list (ACL) which defined permission on network resources (i.e., submit transactions, perform system activities, subscribe to event hub). Another permission layer was defined within the smart contract's functions to ensure that only legitimate entities are allowed to make a request. Using attribute-based access control

(ABAC), the identity of the transaction actor is checked if he/she is authorized to execute the requested operations. To test if the policy can enforce security control on network resources, we created a set of entities with different roles. By switching between roles and attempting to submit transactions of both invoke and query types, results showed that entities that do not acquire grant permission will receive a rejection response, which is displayed as follows.

**Error: failed evaluating policy on signed data during check policy**

**[signature set did not satisfy policy]**

**Table 4.** Execution time of System Monitoring Service.

| Test Round # | Execution Time (ms) | | | |
|---|---|---|---|---|
| | $T_{spawn}$ | $T_{exec_{worker}}$ (Monitoring Job Status) | $T_{exec_{worker}}$ (Retrieving Cluster Health) | $T_{update\_state}$ |
| 1 | 9.519627 | 36.556 | 616.693 | 0.175355 |
| 2 | 8.422683 | 39.495 | 718.583 | 0.378252 |
| 3 | 7.491831 | 38.483 | 681.012 | 0.28282 |
| 4 | 9.348326 | 38.234 | 649.233 | 0.325491 |
| 5 | 8.6374 | 37.424 | 672.796 | 0.29372 |
| Average | 8.6839734 | 38.0384 | 667.663 | 0.2911276 |

To validate ABAC permission, entities whose attribute (i.e., node_name) fails to satisfy the condition at the smart contract level (code snippet is displayed in Implementation Section), the custom error specified in the requested function will be displayed as follows.

**Error: Invoking Entity is Unauthorized.**

2 *Event Subscription—HTTP Systems, Users*

Apart from using transactions as a means to communicate data between network participants, the proposed model also emphasized using events for notifying changes in the shared network state. To ensure the event message (composed as an HTTP packet) is authentic, a mutual TLS connection was enabled on every client (i.e., users, HPC system) to ensure the packets are sent from legitimate sources via an encrypted channel. Thus, it is impossible for malicious users to send falsified events with the purpose of interrupting the system operation. Results from conducting packet analysis with Wireshark showed that the tool cannot extract any information from captured packets.

3 *Account Generation*

Once the user is matched to the HPC system, a smart contract will automatically generate a username, which is unique across the network, as a user account for login to the HPC system. This username is then returned to the user and sent to the matched HPC system to generate a temporary passcode for first-time access. This passcode will be sent to the user off the chain to ensure no confidential data are being recorded on the shared state. The user can then combine knowledge attained from the blockchain (username) and HPC system (passcode) to access the computing resource. This approach ensures that malicious users who capture the packets cannot acquire complete access to information that can potentially cause harm to the HPC system.

3 *HPC Interface Module*

The interface module is installed to the HPC frontend node as a container, which in our implementation, is the Docker container. Many studies suggest that Docker is risky for deploying in HPC as it requires administrator privilege in execution, and recommend using Singularity [37] as a better alternative. Nevertheless, the purpose of using a container

in our work is to ensure the *interface module*, along with its dependencies, can operate properly regardless of the host machine environment, not to execute jobs or any HPC-related commands from inside the container. Therefore, HPC providers can be certain that no actions can attain privilege from container runtime, and any container brand is fine to use in deploying the module.

## 6. Conclusions

This study focuses on the practical design and implementation of an HPC resource sharing system. Blockchain technology is leveraged to present trust and manage agreements between resource providers and lenders. To enhance security in the design, all network activities, including the ability to access system data, are governed by a two-layer access control policy which helps in validating the user's permission on network resources and smart contract function calls. To enable secure connection from the off-chain remote HPC system to the on-chain node, an interface module is proposed for establishing a secure connection and automating cluster data synchronization with blockchain states. Performance evaluation on the prototype shows that the design can efficiently respond to the request with little processing delay, and is also capable of tolerating node crashes. With the permission control presence, security analysis shows the design holds the principle of least privilege where users are granted to perform only relevant actions and acquire only information that is relevant to their roles and responsibilities. Prior to exchanging data, authentication is performed to ensure the connection is established to legitimate peers. With the transmission channel being encrypted, acquiring confidential information by eavesdropping on data in transit is infeasible.

**Author Contributions:** Conceptualization, K.P., S.C. and C.V.; methodology, S.C.; resources, K.P. and C.V.; writing—original draft preparation, S.C. and K.P.; writing—review and editing, K.P., S.C. and C.V.; supervision, C.V.; project administration, C.V. All authors have read and agreed to the published version of the manuscript.

**Funding:** This research received no external funding.

**Institutional Review Board Statement:** Not applicable.

**Informed Consent Statement:** Not applicable.

**Data Availability Statement:** Data are contained within the article.

**Acknowledgments:** The authors would like to thank the National e-Science Infrastructure Consortium (Thailand) project for supporting HPC resources, hardware usage data and user statistics which are essential for conducting this research study.

**Conflicts of Interest:** The authors declare no conflict of interest.

## References

1. Razzaq, S.; Wahid, A.; Khan, F.; Amin, N.; Shah, M.; Akhunzada, A.; Ihsan, A. Scheduling Algorithms for High-Performance Computing: An Application Perspective of Fog Computing. In *Recent Trends and Advances in Wireless and IoT-Enabled Networks*; Springer: Berlin/Heidelberg, Germany, 2019; pp. 107–117.
2. Park, J. Queue Waiting Time Prediction for Large-scale High-performance Computing System. In Proceedings of the 2019 International Conference on High Performance Computing Simulation (HPCS 2019), Dublin, Ireland, 15–19 July 2019; pp. 850–855.
3. Hovestadt, M.; Kao, O.; Keller, A.; Streit, A. Scheduling in HPC Resource Management Systems: Queuing vs. Planning. In Proceedings of the Job Scheduling Strategies for Parallel Processing (JSSPP), Seattle, WA, USA, 24 June 2003.
4. Piyounkorn, K.; Kasisopha, N.; Thaenkaew, P.; Vorakulpipat, C. Automating Job Monitoring System for an Ecosystem of High Performance Computing. In Proceedings of the 9th International Conference on Management of Digital EcoSystems (MEDES '17), Bangkok, Thailand, 7–10 November 2017; pp. 281–286.
5. Piyoungkorn, K.; Thaenkaew, P.; Vorakulpipat, C. A Resource-saving Job Monitoring System of High Performance Computing using Parent and Child Process. In Proceedings of the International Symposium on Grids & Clouds 2019, Taipei, Taiwan, 31 March–5 April 2019.

6. Valles, D.; Williams, D.; Nava, P. Scheduling Modifications for Improvement of Performance on a Beowulf Cluster Single Node. In Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis, Washington, DC, USA, 10–16 November 2012.
7. Valles, D.; Williams, D.; Nava, P. Load Balancing Approach Based on Limitations and Bottlenecks of Multi-core Architectures on a Beowulf Cluster Compute-Node. In Proceedings of the 2012 International Conference on Parallel and Distributed Processing Techniques and Applications, Las Vegas, NV, USA, 16–19 July 2012.
8. Ferro, M.; Klôh, V.; Gritz, M.; Sá, V.; Schulze, B. Predicting Runtime in HPC Environments for an Efficient Use of Computational Resources. In Proceedings of the XXII Symposium on High Performance Computing Systems, Belo Horizonte, Brazil, 26–28 October 2021.
9. Liu, D.; Zhang, Y.; Zhang, H.; Lou, F. User behavior control method for HPC system. In Proceedings of the 2021 China Automation Congress (CAC), Beijing, China, 22–24 October 2021; pp. 2918–2922.
10. Digital Government Development Agency. Open Government Data of Thailand. Available online: https://data.go.th/en/ (accessed on 28 March 2022).
11. Nakamoto, S. Bitcoin: A Peer-to-Peer Electronic Cash System. Available online: https://bitcoin.org/bitcoin.pdf (accessed on 28 March 2022).
12. Zikratov, I.; Kuzmin, A.; Akimenko, V.; Niculichev, V.; Yalansky, L. Ensuring data integrity using Blockchain technology. In Proceedings of the 20th Conference of Open Innovations Association (FRUCT), Saint Petersburg, Russia, 3–7 April 2017; pp. 534–539.
13. Ralph, C.M. A digital signature based on a conventional encryption function. In Proceedings of the Advances in Cryptology-CRYPTO '87, Santa Barbara, CA, USA, 16–20 August 1987.
14. Arthur, G.; Ghassan, O.K.; Karl, W.; Vasileios, G.; Hubert, R.; Srdjan, C. On the Security and Performance of Proof of Work Blockchains. In Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS '16), Vienna, Austria, 24–28 October 2016; pp. 3–16.
15. Kiayias, A.; Russell, A.; David, B.; Oliynykov, R. Ouroboros: A Provably Secure Proof-of-Stake Blockchain Protocol. In Proceedings of the 37th Annual International Cryptology Conference, Santa Barbara, CA, USA, 20–24 August 2017.
16. Castro, M.; Liskov, B. Practical Byzantine fault tolerance. In Proceedings of the Third Symposium on Operating Systems Design and Implementation, New Orleans, LA, USA, 22–25 February 1999; pp. 173–186.
17. Sasin School of Management. Thailand's First Centralized Blockchain Transcript Verification Platform. Available online: https://www.sasin.edu/content/news/thailands-first-centralized-blockchain-platform-to-verify-transcripts (accessed on 20 March 2022).
18. Bank of Thailand. DLT Scripless Bond. Available online: https://www.bot.or.th/Thai/DebtSecurities/Documents/DLT%20Scripless%20Bond.pdf (accessed on 20 March 2022).
19. BCI (Thailand) Co., Ltd. Thailand Launches Blockchain Letters of Guarantee Network for 22 Banks. Available online: https://www.bci.network/post/thailand-launches-blockchain-letters-of-guarantee-network-for-22-banks (accessed on 20 March 2022).
20. Ministry of Public Health, Thailand. eHealth Strategy. Available online: https://ict.moph.go.th/upload_file/files/eHealth_Strategy_ENG_141117.pdf (accessed on 20 March 2022).
21. Electricity Generating Authority of Thailand. National Energy Trading Platform. Available online: https://www.egat.co.th/home/en/national-energy-trading-platform/ (accessed on 20 March 2022).
22. Bangkok Post Public Company Limited. Blockchain easing customs process. Available online: https://www.bangkokpost.com/tech/1738611/blockchain-easing-customs-process (accessed on 20 March 2022).
23. Abdullah, A.M.; Feng, Y.; Dongfang, Z. BAASH: Lightweight, efficient, and reliable blockchain-as-a-service for HPC systems. In Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC'21), St. Louis, MO, USA, 14–19 November 2021; pp. 1–18.
24. Al-Mamun, A.; Li, T.; Sadoghi, M.; Jiang, L.; Shen, H.; Zhao, D. HPChain: An MPI-Based Blockchain Framework for Data Fidelity in High-Performance Computing Systems. In Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC'19), Denver, CO, USA, 17–19 November 2019.
25. Toyoda, K.; Mathiopoulos, P.T.; Sasase, I.; Ohtsuki, T. A Novel Blockchain-Based Product Ownership Management System (POMS) for Anti-Counterfeits in the Post Supply Chain. *IEEE Access* **2017**, *5*, 17465–17477. [CrossRef]
26. Al-Mamun, A.; Li, T.; Sadoghi, M.; Zhao, D. In-memory Blockchain: Toward Efficient and Trustworthy Data Provenance for HPC Systems. In Proceedings of the 2018 IEEE International Conference on Big Data (Big Data), Seattle, WA, USA, 10–13 December 2018; pp. 3808–3813.
27. Khalil, A.; Maher, K.; Abdullah, B.; Fathy, E.; Kamal, M.J.; Khalid, A. Big Data Security and Privacy: A Taxonomy with Some HPC and Blockchain Perspectives. *IJCSNS* **2021**, *21*, 43–55.
28. Sarmenta, L.F.G. Sabotage-tolerance mechanisms for volunteer computing systems. In Proceedings of the First IEEE/ACM International Symposium on Cluster Computing and the Grid, Brisbane, QLD, Australia, 15–18 May 2001; pp. 337–346.
29. iExec. Whitepaper: Blockchain-Based Decentralized Cloud Computing. Available online: https://iex.ec/wp-content/uploads/pdf/iExec-WPv3.0-English.pdf (accessed on 20 March 2022).

30. Androulaki, E.; Barger, A.; Bortnikov, V.; Cachin, C.; Christidis, K.; De Caro, A.; Enyeart, D.; Ferris, C.; Laventman, G.; Manevich, Y.; et al. Hyperledger Fabric: A Distributed Operating System for Permissioned Blockchains. In Proceedings of the 13th EuroSys Conference, Porto, Portugal, 23–26 April 2018.
31. The Linux Foundation. The Ordering Service. Available online: https://hyperledger-fabric.readthedocs.io/en/release-2.2/orderer/ordering_service.html (accessed on 20 March 2022).
32. Diego, O.; John, O. In search of an understandable consensus algorithm. In Proceedings of the 2014 USENIX conference on USENIX Annual Technical Conference, Philadelphia, PA, USA, 19–20 June 2014; pp. 305–320. Available online: https://raft.github.io/raft.pdf (accessed on 20 March 2022).
33. Merkel, D. Docker: Lightweight Linux containers for consistent development and deployment. *Linux J.* **2014**, *2014*, 2.
34. The Linux Foundation. Kubernetes. Available online: https://kubernetes.io/ (accessed on 20 March 2022).
35. The Linux Foundation. Hyperledger Caliper. Available online: https://www.hyperledger.org/use/caliper (accessed on 20 March 2022).
36. Apache Software Foundation. Apache Kafka. Available online: https://kafka.apache.org/documentation (accessed on 20 March 2022).
37. Kurtzer, G.M.; Sochat, V.; Bauer, M.W. Singularity: Scientific containers for mobility of compute. *PLoS ONE* **2017**, *12*, e0177459. [CrossRef] [PubMed]