IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Received 19 August 2022, accepted 5 September 2022, date of publication 16 September 2022, date of current version 23 September 2022.

Digital Object Identifier 10.1109/ACCESS.2022.3207167

RESEARCH ARTICLE

A Volunteer Computing Architecture for Computational Workflows on Decentralized Web

ALESSIA ANTELMI, GIUSEPPE D'AMBROSIO, ANDREA PETTA, LUIGI SERRA, AND CARMINE SPAGNUOLO

Department of Computer Science, Università degli Studi di Salerno, 84084 Fisciano, Italy

Corresponding author: Luigi Serra (lserra@unisa.it)

ABSTRACT The amount of accessible computational devices over the Internet offers an enormous but latent computational power. Nonetheless, the complexity of orchestrating and managing such devices requires dedicated architectures and tools and hinders the exploitation of this vast processing capacity. Over the last years, the paradigm of (Browser-based) Volunteer Computing emerged as a unique approach to harnessing such computational capabilities, leveraging the idea of voluntarily offering resources. This article proposes VFuse, a groundbreaking architecture to exploit the Browser-based Volunteer Computing paradigm via a ready-to-access volunteer network. VFuse offers a modern multi-language programming environment for developing scientific workflows using WebAssembly technology without requiring the user any local installation or configuration. We equipped our architecture with a secure and transparent rewarding mechanism based on blockchain technology (Ethereum) and distributed P2P file system (IPFS). Further, the use of Non-Fungible Tokens provides a unique, secure, and transparent methodology for recognizing the users' participation in the network. We developed a prototype of the proposed architecture and four example applications implemented with our system. All code and examples are publicly available on GitHub.

INDEX TERMS Scientific computing, volunteer computing, browser-based volunteer computing, decentralized web, Web 3.0, P2P, WebAssembly, distributed computing, parallel computing.

I. INTRODUCTION

Over the past decade, personal computers (PCs) have become one of the most consolidated markets. In 2021, approximately 340 million PCs were shipped worldwide [1], considering a revert of the trajectory since 2011 [2]. Still, today smartphones represent the most dominant technology, with around 1.5 billion devices sold per year in the last five years [3], [4]. Further, Internet users are currently growing at an annual rate of 4.0 percent, equating to an average of more than half a million new users each day [5]. This colossal number of computational devices represents an enormous opportunity from a computing perspective. According to the 2021 TOP500 rank [6], the most powerful supercomputer FUGAKU provides around 8 million cores. Based on these statistics, the computational power obtainable by a tiny number of all possible internet-accessible devices is significantly more. Taking advantage of this huge (and mostly unused [7]) processing capacity represents a powerful opportunity for science and society.

The paradigm of Volunteer Computing [8] (VC) emerged as a prominent approach to harnessing the computational capabilities of such devices. VC is a type of distributed computing based on two pillars: computation and participation. The former refers to the ability of the network to orchestrate heterogeneous computational nodes to perform a given task. The latter is the cornerstone of the whole paradigm and refers to the mechanism by that people *voluntarily* donate their computing resources to the network to collaborate on a project. Although VC comes with peculiar technological challenges (e.g., managing nodes with heterogeneous hardware and software, high dynamicity of the environment, asynchronism),

The associate editor coordinating the review of this manuscript and approving it for publication was Fabrizio Messina.

this paradigm provides researchers with lower-cost computing power and reduced energy consumption. To alleviate some intrinsic limitations of VC systems and encourage joining volunteer networks, the paradigm of Browser-Based Volunteer Computing (BBVC) [9] gained popularity, also thanks to improvements in the processing capacity of web browsers and the release of powerful software libraries (e.g., WebGL, and TensorFlow.js) [10]. BBVC provides access to the volunteer network using web applications, which execute volunteer jobs in the background and transparently from the user's perspective. These systems inherit all benefits from web browsers, offering portability, flexibility, and ubiquity.

Contributions: Over the last four years, different BBVC platforms have been proposed. Still, there is room for improvements to make such platforms fully decentralized and scalable, offering users a trusted computing environment and providing complete control over the resources donated to the volunteer network. To make a step toward this direction, this paper introduces VFuse, a fully-distributed volunteer network based on a peer-to-peer (P2P) architecture. VFuse offers volunteers an easy and ready-to-use programming environment directly in their browsers through web-based interactive notebooks, which allow users to develop and monitor the requested computation and analyze its results. Further, our platform provides a secure and trustworthy execution environment and a reward mechanism thanks to the adoption of Blockchain technology to ensure results' reliability. VFuse represents an effort to make the web decentralized [11], allowing users to run intensive computing tasks in a free-to-use and trusted environment.

The major contributions of this paper can be summarized as follows:

- The design of a novel architecture for BBVC defined over a fully-distributed P2P network;
- The proposal of an innovative rewarding strategy based on blockchain technology to incentive users to join the volunteer network;
- A user-friendly interface based on web notebooks to easily access the volunteer network and benefit from its computing capabilities transparently;
- An empowered multi-language programming environment offered via interactive web notebooks;
- A detailed description of four applications, presenting how VFuse can be exploited in such contexts;
- A prototype of the VFuse system, available on a public GitHub repository [12], which includes workflow orchestration functionalities, storage capabilities using IPFS, and two execution backends offered with JavaScript and Python programming languages.

The remainder of this paper is organized as follows. Section II reviews the key ideas behind BBVC, describes the main existing frameworks in the context of BBVC and discusses the challenges these frameworks need to face. Section III illustrates the main features of VFuse, the rationale behind the choice of the technologies used, and how they impacted the platform's design. Section IV details the architecture of VFuse, delineating its functionalities and internal mechanisms. Section V describes how VFuse can be exploited through four use cases. Section VI discusses a first performance evaluation and the current limitation of the VFuse prototype. Finally, Section VII concludes this work by delineating our current work and possible future directions.

II. BACKGROUND AND RELATED WORK

VFuse is a volunteer distributed browser-based library to create and execute scientific workflows. To better clarify its position within the state-of-the-art, we first introduce the concept of VC, along with the most popular VC frameworks. Then, we focus on BBVC and its peculiar challenges. Finally, we present existing solutions, describe the peculiarities of our proposed system, and provide a detailed comparison among the available BBVC frameworks.

A. VOLUNTEER COMPUTING

VC is a computational paradigm based on the willingness of people to donate idle computing resources to run computational and storage-intensive tasks [13]. VC shares many similarities with online community-based projects, in which people's desire to voluntarily contribute resources - such as knowledge, time, and skills - underpins the sustainability of such initiatives [8].

The idea of exploiting idle resources from volunteer computers came from the GIMPS (Great Internet Mersenne Prime Search) project in 1995 [14]. The project is still running, and it allowed the discovery of the 51st Mersenne prime in 2018, the current largest known prime number. Other early projects include distributed.net [15], SETI@home [16], and Folding@home [17]. Today there are over 30 active projects.

Two of the most popular VC frameworks are BOINC [18] and XtremWeb [19]. Both frameworks exploit a centralized architecture for managing jobs and resources. Further, they require users to download and install a specialized client to execute project tasks. This approach has two critical drawbacks concerning the programmability and the trustiness of the applications. First, there is a constraint on the programming language used as each application must be implemented with the same framework language. Second, adopting a centralized architecture upper limits the number of volunteer nodes as their number cannot exceed the network and computational capabilities of the server node.

To address the issues related to the client-server architecture, other VC frameworks either rely on a P2P overlay network [20], [21] or a blockchain-based system [22], [23]. Specifically, the idea of using a blockchain in a VC framework is tailored to solve the issue of trust between devices and the lack of traceability, making it difficult for users to evaluate the contribution and credit of each volunteer [24]. Further, the characteristics of blockchain, such as decentralization and persistence, allow solving the problems of scalability

and single point of failure under the traditional client-server architecture [25], [26]. Golem [27] is an example of a P2P VC platform based on blockchain, specifically focused on computations such as computer graphics rendering and machine learning algorithms. BOID [28] is another example of a blockchain-based VC platform in which volunteers get paid in custom BOID cryptocurrency tokens for their contributing resources.

For a comprehensive review of VC frameworks, we refer the reader to the survey of Mengistu and Che [13].

B. BROWSER-BASED VOLUNTEER COMPUTING

The idea of BBVC comes from the need to overcome an intrinsic limitation of VC systems. Usually, these frameworks require the users to follow a given installation procedure, which can be challenging for many [29]. Further, many users fear installing unfamiliar software because of malware and spyware [30]; they may be simply lazy or not appealed by the project or even lack awareness of its existence [9]. In contrast, the underlying concept of BBVC is to increase participation in the volunteer network by providing more user-friendly access to the network via a web application, hence without requiring specific installation or configurations. In this approach, a browser automatically and transparently executes tasks during the user's visit to a particular website; as a consequence, users usually consider their impact negligible [31].

1) CHALLENGES OF BBVC FRAMEWORKS

Designing and implementing a VC system pose a set of non-trivial challenges. These derive from the intrinsic nature of the computing environment, and BBVC systems clearly inherit them. To ease this process, Fabisiak and Danilecki [9] defined a set of desirable features a BBVC system should have (the same desiderata hold for a VC system). A brief description of each desideratum follows.

- *Accessibility*. Easiness of accessing the platform and sharing resources.
- *Adaptability/Dynamicity*. The BBVC platform should be aware that the environment is ever-changing, as the number of nodes may vary.
- *Availability*. The BBVC platform should be accessible regardless of any problem.
- *Fault Tolerance*. The BBVC platform should be tolerant of faults and disconnections.
- *Heterogeneity*. The BBVC platform should consider that volunteer machines could have different hardware, operating systems, and performance.
- *Programmability*. Easiness of developing new tasks on the BBVC platform.
- *Scalability*. The BBVC platform must handle a growing amount of connections.
- *Security*. The code run by the platform should not harm the volunteer machine.
- *Usability*. The BBVC platform should be easy to deploy and use.

Along with the above desiderata, we also defined some additional features a BBVC system should have to encourage volunteers' engagement and improve the platform's reliability and functionalities.

- *Task deployment and scheduling*. Flexibility of the BBVC platform in supporting different deployment and scheduling policies.
- *Result reliability*. The BBVC platform must ensure the computed results' correctness and prevent any result manipulation or malicious execution.
- *Supported programming languages*. Flexibility of the BBVC platform in supporting different programming languages.
- *Supported computational paradigms*. Flexibility of the BBVC platform in supporting different computational paradigms.
- *User resource usage*. Possibility of configuring the number of local volunteer resources, such as CPU or memory, to allocate for computing the task.
- *Data management*. Ability of the BBVC platform to support data operations, such as data gathering, manipulation, and storage.

2) BBVC FRAMEWORKS

BBVC frameworks rose to prominence over the last decade thanks to the incredible advancement in web technologies and the ever-increasing web usage. We can distinguish three generations of BBVC systems [9], which reflect the improvements in the web programming language, communication protocols, and thread support. In this paper, we specifically focus on reviewing and comparing the third-generation frameworks most similar to VFuse. For a comprehensive review of BBVC frameworks, we refer the reader to the survey of Fabisiak and Danilecki [9].

Madoop [32] leverages the power of WebAssembly [37] to implement a distributed MapReduce framework on browsers. The central server, which hosts the Hadoop software, handles the management of both jobs and results. Each job is written in C/C++ and compiled in a WebAssembly format to be run in the browser. The client web page, which runs a Madoop code snippet, requests a job to the main server, and, upon completion of its execution, it returns the results to the server. Then, the server sends back the results to the job initiator.

JSDoop [10] is a library for distributed collaborative high-performance computing in web browsers, based on the MapReduce paradigm as Madoop. Both JSDoop clients and servers are implemented in JavaScript. A queue server handles the task scheduling and the result management, while data are available in a centralized server. More queue servers could be used to guarantee load balancing.

BrowserCloud.js [33] proposes a decentralized architecture to find and utilize resources through a P2P overlay network. Participants join the network via a centralized rendezvous point; then, the message routing is handled via an adaptation of the Chord routing algorithm, designed for a P2P

TABLE 1. Main characteristics of the BBVC platforms. N/C stands for not clarified.

Platform	Adaptability/Dynamicity	Fault tolerance	Programmability	Programming language(s)	Scalability
VFuse (Ours)	P2P overlay network and asynchronous workflows.	Role replication and gossiping of the workflows.	Ready-to-use development environment via integrated web notebooks.	Multi-language (currently supported: JavaScript and Python).	Fully-decentralized gossiping strategy and distributed file system (IPFS).
Madoop [32]	Handled by the main server.	Depends upon the main server.	Installation and setting of the development environment.	C/C++ compiled into the WebAssembly format.	Depends upon the main server.
JSDoop [10]	Handled by one or more Queue Server(s).	Tasks are added back to the execution queue in case of problems.	Installation and setting of the development environment.	JavaScript or WASM code.	Depends upon Queue servers and database nodes.
BrowserCloud.js [33]	DHT-based P2P overlay network.	Based on the DHT routing protocol and the reliability of each node (no replication mechanism specified).	Installation and setting of the development environment.	JavaScript	DHT-based P2P overlay network.
Pando [34]	Faster devices receive more inputs, which are read when resources are available.	Depends upon a centralized Stream Lender.	Installation and setting of the development environment.	JavaScript	Depends upon a centralized Stream Lender.
Genet [35]	Pando network with a fat-tree overlay network.	Depends upon the intermediary (internal) nodes of the fat-tree overlay network.	Installation and setting of the development environment.	JavaScript	Fat-tree overlay network.
CollabChain [36]	P2P overlay network with a direct exchange of data between submitters and executors.	Based on the reliability of the Coordinator and Submitters.	Installation and setting of the development environment.	JavaScript	P2P network, but based on a centralized Coordinator to manage and schedule tasks.

Platform	Task deployment	Task scheduling	Result reliability	Computing paradigm	User resources	Data management
VFuse (Ours)	Gossip of the requested workflow on the P2P network.	Chosen by each node based on the task priority.	Blockchain-based.	Multiple paradigms (e.g., workflow-based, MapReduce, Fork/Join)	Users can configure the resources to donate to the network.	Decentralized (IPFS), centralized (URL), and inline (sent with the task).
Madoop [32]	Registration of jobs and relative inputs to the main server.	Handled by a centralized node (Master).	N/C	MapReduce	N/A	Centralized database.
JSDoop [10]	Sent to the main server that distributes them across the queue servers.	Handled by one or more Queue servers.	N/C	MapReduce	N/A	Centralized database.
BrowserCloud.js [33]	Sent on the P2P network.	Chosen by each node according to their workload.	N/C	MapReduce	N/A	Inline (sent with the task).
Pando [34]	Sent to a centralized Stream Lender, via (possible pipelined) Unix commands.	Handled by a centralized node (Stream Lender).	N/C	Declarative concurrent programming paradigm.	N/A	Handled by the centralized Stream Lender.
Genet [35]	Based on Pando.	Based on Pando.	N/C	Declarative concurrent programming paradigm.	N/A	Handled by relay (internal) nodes of the fat-tree overlay network.
CollabChain [36]	Centralized database containing submitted tasks, stored in the Coordinator.	Chosen by the executor node from the database of the Coordinator.	Blockchain-based.	Computation of single JavaScript functions.	N/A	Sent in batch from the submitter to the executor.

distributed hash table. The P2P interconnectivity is obtained with WebRTC, a technology enabling Real-Time Communications in the browser via a JavaScript API. BrowserCloud.js provides a simple mechanism to define JavaScript functions, including inline data and the number of required peers to complete the task.

Pando [34] is a tool born with the intent of leveraging the potential of VC for personal projects. Its programming

model corresponds to a streaming version of the functional *map* operation: Pando applies a given function on a series of input values to obtain a series of results. Pando relies on the pull-stream design pattern to manage the input stream for functions, which are written in JavaScript and can be combined in Unix pipelines. The task deployment is based on a Node.js master server (Stream Lender) responsible for scheduling functions and collecting their results.

Genet [35] is an evolution of Pando that tries to overcome its scalability problems - due to direct connections handled with WebRTC - by using a fat-tree overlay network (where processors are located on the leaves and internal nodes relay data for all their children). Genet differs from Pando in managing browser connections, switching a node's role from management to relay when its direct connections (children) reach a given threshold.

CollabChain [36] is a browser-based volunteer platform that relies on blockchain technology to provide a trusted environment and foster users to make their resources available to the network. CollabChain is based on a P2P overlay network and defines three types of nodes: submitters, executors, and coordinators. Submitters require a task, i.e., a JavaScript function and its inputs, while executors compute them. A single coordinator acts as a bootstrap node and maintains a database of all tasks uploaded by submitters. The blockchain guarantees payment for the volunteers that complete their work and honesty of results by matching the output evaluated by the volunteer and the pre-computed output described in the smart contract.

CollabChain currently represents the most similar work to VFuse. Nonetheless, several major points distinguish the two architectures. The first significant difference relies on how tasks are deployed and scheduled. If a VFuse node wants to submit a workflow (see Section III-A), then it has to broadcast it over the P2P network and wait for the results. Even if the node disconnects, the network still gossips the workflow. Tasks of each workflow are then scheduled by each volunteer based on the associated priority. On CollabChain, submitters have to submit their tasks to the coordinator, and they are required to stay online even after delegating the process function and the inputs to the executors to obtain computed outputs from the executor. Executors directly choose tasks from the coordinator, and no scheduling policy is explicitly described. The second significant distinction regards the computing paradigm. VFuse offers users a platform to define the requested computation as a workflow, within which either dependent or parallelizable tasks may be specified and run by different volunteers. In contrast, CollabChain defines the requested computation via a JavaScript function that can be run by a single executor. Other main dissimilarities relate to *(i)* the management of the user resources offered to the volunteer network (CollabChain does not offer a direct control), *(ii)* how data are handled (VFuse relies on IPFS, while submitters and coordinators need to exchange data on CollabChain directly), and *(iii)* the rewarding mechanism (VFuse exploits the rewarding mechanism to prioritize workflows, while CollabChain provides an actual currency).

Table 1 compares VFuse with the main BBVC frameworks, considering the desiderata described in Section II-B1. It is worth noting that all systems inherit accessibility, availability, heterogeneity, security, and usability requirements from browsers.

III. METHODOLOGY AND DESIGN CHOICES

The VFuse architecture is built upon two cornerstones: *(i)* ensuring a high level of scalability and *(ii)* storing inputs, outputs, and authorship of users' tasks over a public blockchain. Table 2 describes the main characteristic of VFuse, clarifying how the platform addresses the challenges described in Section II-B1. The following sections illustrate the main design choices behind VFuse.

A. PROGRAMMING PARADIGM

Distributed computing offers mechanisms and tools for orchestrating distributed computational workflows and resources for transparently solving problems. In this context, a critical issue is to use the proper programming model to define the distributed computation. Scientific workflow [38] is a commonly used paradigm to manage the coordinated execution of actions that can be repeatable and dependent on each other. This design enables the plugging of problem-solving components within the workflow to prove a scientific hypothesis. Such a paradigm brings several benefits, such as automation, scalability, resilience, and verifiability.

VFuse adopts the workflow pattern, allowing the design of the requested computation as a sequence of interdependent jobs (or tasks). In other words, the computation is divided into self-consistent jobs, whose execution may depend on the termination of other tasks. Hence, a VFuse application is defined by a pipeline of jobs that is modeled via a directed acyclic graph (DAG) (see Section IV-B5.a). VFuse provides operations to build workflows, add jobs and describe dependencies between them. The generic nature of this approach allows programmers also to exploit other distributed paradigms in VFuse, such as fork/join and MapReduce.

B. APPROACH BASED ON INTERACTIVE NOTEBOOKS

We designed VFuse to support the development and execution of distributed applications via an interactive web notebook, à-la Jupyter, CodePen, Gitpod, or JS Fiddle. This choice offers VFuse volunteers a ready-to-use, quick programming environment without requiring software configuration and installation. Further, web notebooks provide a dynamic programming environment supporting multi-languages, workflow monitoring, submission, and visualization of results. Specifically, VFuse provides a set of asynchronous functions - that support different programming languages - to retrieve and store data from the network, build workflows, and add jobs to them, specifying input data and dependencies.

TABLE 2. Answers of VFuse to the challenges of BBVC.

Challenge	VFuse feature
Accessibility	The only requirement to participate as a volunteer in the VFuse system is to have a web browser.
Adaptability/ Dynamicity	VFuse is resilient to the arrival/removal of network nodes thanks to the implementation of asynchronous workflows over a P2P network. Tasks are gossiped within the network until their associated TTL expires, or the Initiator stops their execution.
Availability	Today, web browsers are ubiquitous on computing devices, such as personal and desktop computers, as well as mobile phones.
Fault Tolerance	If a node - either the Initiator or any other node - disconnects, the requested workflow will continue flowing within the network. Further, being a P2P network, there is no centralized entity that can fail.
Heterogeneity	Being a BBVC system, VFuse is inherently cross-platform. Any device can collaborate by simply connecting to the service via a browser.
Programmability	VFuse relies on the use of interactive web-based notebooks, which do not require installing and configuring any local development environment. Further, VFuse allows testing the workflow to submit (to the volunteer network) on the user's local machine.
Scalability	VFuse accommodates both communication and storage scalability. The former is guaranteed by using a fully-decentralized gossip strategy to propagate workflows within the network. The latter directly derives from the use of the distributed file system IPFS.
Security	Web browsers guarantee the safe execution of programs on the host device as they run web pages in a sandbox. Moreover, communication happens via the HTTPS protocol.
Usability	VFuse does not require the installation of additional software but a browser.
Task deployment and scheduling	VFuse exploits a self-organized mechanism (gossip flooding) in which all nodes exchange information on available workflows and the status of jobs' executions.
Result reliability	The reliability of the computed results is inherently guaranteed by using a blockchain to store them; once written on the blockchain, the outcome and its author cannot be modified. Further, VFuse produces a warning message if two or more clients generate different results for the same workflow's job.
Supported programming languages	VFuse uses the WebAssembly runtime module to support a multi-language programming environment. Currently, a VFuse application can be either implemented in JavaScript or Python.
Supported computational paradigms	VFuse stands as a workflow-based manager tool for a volunteer network, enabling programmers to exploit distributed computational paradigms, such as fork/join and MapReduce.
User resource usage	Each VFuse user can configure the number of concurrent functions to run on their device.
Data management	VFuse relies on the distributed file system IPFS to store data. Further, our platform offers methods to retrieve data from URLs and upload data inline.

C. TECHNOLOGIES BEHIND VFuse

The myriad of different technological challenges we incurred during the design of our system profoundly shaped the final architecture of VFuse. We exploited the following technologies to enable communications among several P2P-connected devices, guarantee the trustiness of job executions, manage distributed data, and support a multi-language programming environment. A description of the key technologies chosen and their impact on the VFuse follows.

- **WebAssembly** [37] (Wasm) is a low-level assembly-like language runnable in web browsers. Wasm is designed as a portable compilation target for programming languages, meaning that it allows languages like C/C++, Rust, or Python to run on the web with near-native performance. Wasm is also designed to run alongside JavaScript, offering programmers a way to take advantage of WebAssembly's performance and power and JavaScript's expressiveness and flexibility in the same application. Among the other strengths of Wasm, there is its safeness (memory-safe, sandboxed execution) and easiness of debugging (textual format). Further, Wasm maintains the versionless, feature-tested, and backward-compatible nature of the web.

Architecture Insight: The use of Wasm as a core technology of VFuse is critical to improve Web Workers' performance and provide support for programming languages other than JavaScript. Currently, VFuse supports the development of workflows written in JavaScript or Python. The rationale behind the choice of Python comes from the plethora of libraries the language offers to manipulate and analyze data, well-suited to implement scientific workflows. To implement Python Web Workers, we used Pyodide [39] as a Wasm-compiled Python interpreter. Specifically, Pyodide is a Python distribution for the browser and Node.js based on WebAssembly/Emscripten [40] that makes it possible to install and run Python packages in the browser with an embedded version of the pip python package manager. Hence, all general-purpose and scientific Python packages - such as NumPy, pandas, SciPy, Matplotlib, and scikit-learn - can be used. Further, Pyodide allows the programmer to easily mix JavaScript and Python in the same code script thanks to a robust foreign function interface. The use of WASM as underlying technology ensures that VFuse can be easily expanded to support other programming languages.

- **Libp2p** [41] is a network framework supporting the development of decentralized P2P applications based on WebSocket or WebRTC to enable communication among nodes. Built upon the Kademlia DHT [42], a network protocol that allows the development of P2P network applications, Libp2p leverages public-key cryptography [43] to manage peer identities and enable secure communication. Libp2p offers NAT traversal, circuit relay, stream multiplexing, and addressing functionalities.

Architecture Insight: The use of Libp2p enables VFuse to be aware of the status of the network by hindering the underlying network communication protocol and details on the routing tables. Libp2p also offers new VFuse volunteer devices the possibility to join the network through bootstrap nodes, whose number can be increased on-demand based on

the size of the network. Lastly, Libp2p enables direct communication between VFuse nodes, allowing them to inform other devices about a new workflow to be run. The flooding of the workflow within the network happens via a gossip strategy, which avoids a centralized orchestration, and terminates either when the Initiator ends its execution or its time-to-live (TTL) expires (see Section IV-B4).

- **IPFS** [44] is a distributed file system built upon Libp2p to store large data files and support blockchain operations. The main characteristic of IPFS is how content is identified. Rather than associating a location with a resource (like what happens with URLs), IPFS uses an immutable hash code - called Content Identifier (CID) - to identify resources in the network. To allow dynamic resource addressing, IPFS provides the Inter-Planetary Name System (IPNS) that leverages a unique hash pointer targeting different CIDs when the content changes. Further, IPFS ensures data distribution and replication to guarantee availability and fault tolerance.

Architecture Insight: The VFuse architecture is designed as a fully-distributed application running over a volunteer network of computational nodes. The use of IPFS as a component of VFuse ensures the unique identification of inputs and jobs' outputs, giving our platform the power of content-addressed storage. To safeguard the indefinite persistence of data and workflows on the VFuse network, these can be pinned to one or more IPFS nodes. Pinning gives the programmer control over disk space and data retention and guarantees that the pinned resources are not deleted during IPFS garbage collection. Further, the use of the IPFS Cluster, a distributed application that works as a sidecar to IPFS peers, enables VFuse to allocate, replicate, and track pinned resources among multiple peers; hence, guaranteeing data redundancy and availability without compromising the distributed nature of the IPFS network. Finally, IPFS empowers the exploitation of blockchain functionalities to implement a secure and trustworthy mechanism (see Section IV-B5.d).

- **Ethereum** [45] is a decentralized, open-source blockchain platform establishing a P2P network that securely executes and verifies smart contracts. Smart contracts are event-driven distributed programs stored on the blockchain that run when predetermined conditions are met. They allow participants to transact with each other without a trusted central authority. Transaction records on Ethereum are immutable, verifiable, and securely distributed across the network, giving participants full ownership and visibility into transaction data.

  Ethereum allows the creation of unique and indivisible tokens, called non-fungible tokens (NFTs). NFTs represent ownership of unique items, such as a piece of art, digital content, or media. Each NFT can only have one official owner at a time, and the Ethereum blockchain secures them (e.g., no one can modify the record of ownership or copying existing NFTs). In other words, NFTs embody an irrevocable digital certificate of ownership and authenticity for a given digital or physical asset.

Architecture Insight: VFuse exploits the Ethereum blockchain to implement a rewarding strategy in the volunteer network. Every time a node contributes to the computation of a job, it receives a reward in the form of a VFuse NFT, which guarantees the ownership of the produced digital asset (namely, the result of the computation). The number of NFTs collected by VFuse clients is then used to prioritize their submitted workflows. Section IV-B5.b and Section IV-B5.d detail this process. It is worth stressing that we did not adopt a VFuse currency given the volunteer nature of the network itself: users do not have to pay to use the network, but, at the same time, they are encouraged to offer their computing resources in exchange for a faster termination of their submitted workflows.

IV. VFuse ARCHITECTURE

VFuse is a decentralized network that acts as a workflow manager accessible via browser for volunteer-based distributed computation. VFuse's primary purpose is to enable users to access a robust and secure volunteer network without requiring the installation and configuration of any additional software. VFuse users can define asynchronous workflows made up of functions (jobs or tasks) with possible temporal dependencies on their execution. Thanks to the asynchronous nature of VFuse workflows, users are free to leave the network while their required workflow is running and gather its results at any moment in the future.

The VFuse architecture is designed on top of the following innovative objectives:

- providing a ready-to-use programming environment to access a distributed volunteer network using interactive web notebooks;
- designing a modular and expandable architecture that transparently exploits the underlying technologies;
- supporting a distributed volunteer network built on P2P communications, storage, and Blockchain technologies.

A. VFuse NODE TYPES AND THEIR INTERACTION

A VFuse node may be either a *computing*, *bootstrap*, or *pinning* node. Specifically, a computing node is a VFuse volunteer device that offers and may require network computing capabilities through a web browser. Being a P2P network, VFuse requires the presence of bootstrap nodes to allow new devices to join the network. These special nodes are responsible for the discoverability of the VFuse network as well as for the initialization of new connections. In particular, a VFuse bootstrap node runs the same software stack of computing nodes plus a WebRTC signal server (publicly accessible over the Internet and hosted on a NodeJS server) that allows direct connections with other peers, such as a browser-to-browser communication. It is worth noting that every computing node may also act as a bootstrap node to

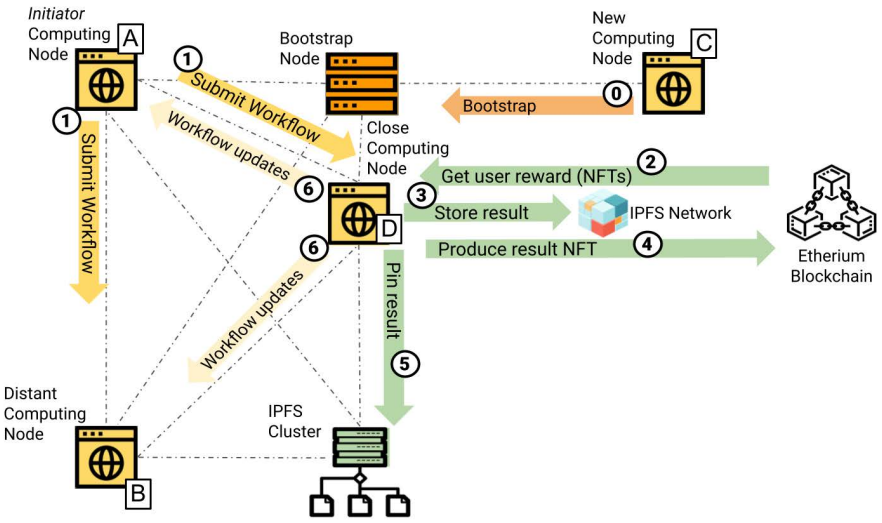


FIGURE 1. Interactions between nodes within the VFuse network in a typical execution flow. A new computing node *C* may enter the network by connecting with a VFuse bootstrap node (Step 0). A node *A* which submits a workflow (Step 1) becomes the Initiator of the requested computation. Each computing node of the network, such as *B* and *D*, receives the workflow, executes some jobs based on the associated priority (Step 2), stores the results locally (Step 3), gets a reward (Step 4), and pins the result to the IPFS cluster (Step 5). Eventually, each node broadcasts an update workflow message (Step 6).

avoid centralized bottlenecks. Lastly, pinning nodes belong to the IPFS cluster and protect data from the garbage collection of IPFS. Figure 1 shows an overview of the VFuse system, depicting the three possible roles a VFuse node may play and how they interact during a workflow's life cycle. First, each user who wants to join the volunteer network has to connect to a VFuse bootstrap node (Step 0, see Section IV-B1). The execution flow of a computing request then starts when a computing node submits a workflow, hence becoming its *Initiator*. The requested workflow is then gossiped within the volunteer network (Step 1, see Section IV-B5.a). Each computing node determines the task to compute based on the priority of the Initiator (Step 2, see Section IV-B5.b). Upon completion of each job, every computing node store the computed results on the IPFS Network (Step 3, see Section IV-B5.a), gets its reward (Step 4, see Section IV-B5.d), pins the results on the IPFS cluster (Step 5, see Section III-C), and gossips the workflow's updates (Step 6).

B. ARCHITECTURE

This section delineates the main components of the VFuse architecture (see Figure 2), briefly describing their functionalities. Each VFuse node runs the VFuse suite, made up of five software components. The whole suite relies on the storage technologies offered through the IPFS distributed file system, along with the communication protocols implemented by Libp2p for P2P architectures. A description of each component of the VFuse protocol suite follows.

Access Component. This component gives users access to the VFuse network, offering a graphical web interface of the system. It allows VFuse users to manage their profile, create, define, submit, and stop workflows inside interactive web notebooks, which enable the (remote/local) verification of workflow executions.

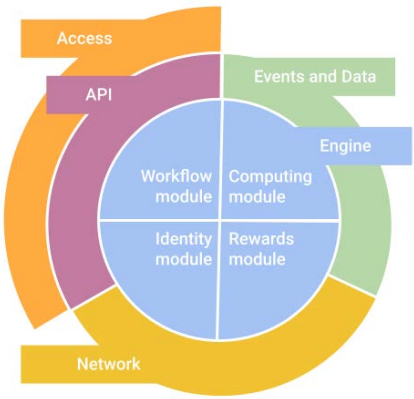
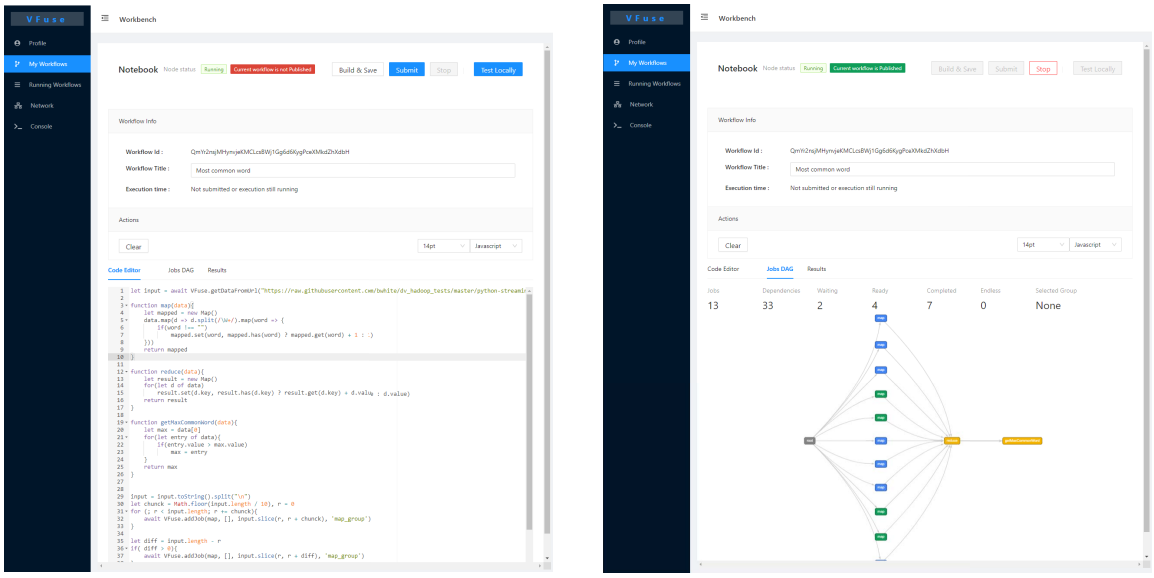


FIGURE 2. VFuse architecture.

API Component. This component defines the synchronous and asynchronous function interfaces to let users access the VFuse platform.

Events and Data Component. This component provides data and communication utilities across components. Communication happens via events, which transmit information about jobs' status and about creating, updating, and deleting data asynchronously.

Network Component. This component is responsible for providing the VFuse network communication protocol and data management (i.e., storage) functionalities by exploiting Libp2p and IPFS over HTTPS. It further



(a) Integrated programming IDE offered.

(b) DAG representation of a workflow.

FIGURE 3. Snapshots of the VFuse web client application (`My Workflows` page).

offers *(i)* a set of callbacks to get information on the nodes' status, *(ii)* bootstrap operations, and *(iii)* an interface to the IPFS Cluster.

Engine Component. This is the core component of the VFuse architecture, which provides all computing, rewarding, and user-related functionalities. It is made up of four modules: *(i)* the *Workflow Module* handling the workflows lifecycle; *(ii)* the *Computing Module* managing job executions from specific workflows; *(iii)* the *Identity Module* administrating user access and preferences; and *(iv)* the *Reward Module* defining and managing the rewarding mechanism via a blockchain for securely storing user rewards.

The following sections illustrate each component in detail.

1) ACCESS COMPONENT

The VFuse access component embodies the *middleman* between users and the VFuse system. It offers users access to the network through a web application (implemented via the ReactJS framework[1]) that exploits the API component to use the functionalities provided by VFuse. Specifically, a user who wants to join the volunteer network sends an HTTPS request to the VFuse central server, hosting the VFuse web application. Once the browser renders the application, the user can then join a specific VFuse volunteer network by specifying the required *(i)* bootstrap node, *(ii)* signal server, and *(iii)* pinning cluster. It is worth stressing that the VFuse web application runs all code client-side or via the P2P volunteer network. Hence, the central server's only task is to serve the VFuse application.

[1]https://reactjs.org/

The VFuse application provides access to the user profile configurations (`Profile` page), workflow management features (`My Workflows` and `Running Workflows` pages), network monitoring (`Network` page), and a logging console (`Console` page).

- In the `Profile` tab, each user can set up the information related to the particular VFuse network to join by specifying the IP address of its bootstrap node, signal server, and pinning cluster.
- On the `My Workflows` page, users can create, develop, locally test, and submit a new workflow to the volunteer network. Figure 3a shows a snapshot of the VFuse client application detailing the programming IDE offered to manipulate workflows. Users can also visualize an interactive graphical representation of the computation using a Job DAG (see Figure 3b), which is automatically updated according to the computation's status and provides the computed results for each job.
- In the `Running Workflows` tab, users can visualize the queue with the received and running workflows, representing all workflows the user received from the network and offered to their computational resources.
- In the `Network` tab, users can continuously monitor the status of the VFuse network by listing the peers they are connected with.
- Finally, on the `Console` page, users can check VFuse logging messages.

2) API COMPONENT

The API Component provides external access to the functionalities offered by the VFuse platform. Further, this

TABLE 3. VFuse API for developing workflows within the VFuse IDE panel.

Function/Object	Parameter(s)	Returns	Description
`VFuse`	-	-	The VFuse object to access the API methods/functions: `VFuse.`*`api_name(parameters)`*.
			Data API
`getData`	CID	String	Retrieve content from IPFS using the given unique identifier CID.
`getDataFromUrl`	URL, Start, End	String	Retrieve content from a given URL address. This method also permits the partial request of data if the server support the property `Accept-Ranges: bytes`.
`saveData`	Data	CID	Store data on IPFS and returns a new CID, which uniquely identifies the data on IPFS.
			Workflow API
`addJob`	Name, Input, Dependencies, Groups	JID	Adds a new job to the workflow DAG. • *Name*: string value that corresponds to the function name defining the job; • *Input*: the data in input to the job - may also be *null*; • *Dependencies*: an array of job dependencies - it contains job IDs or regular expressions identifying matching jobs or group names; • *Groups*: an array of string values corresponding to the groups' names to which the job belongs. It returns an integer value corresponding to the unique identifier of the job in a workflow (*JID*).
`addToGroup`	JID, Groups	Boolean	Assign a job to all groups defined by the given array of string.
`setRepeating`	JID	Boolean	Set a job as a repeating job, i.e., a job is (re)scheduled every time all its dependencies are satisfied.

component offers programmers an API to manage workflows and develop them using different programming languages. Specifically, the workflow API allows the programmer to *(i)* retrieve and store computation data asynchronously and *(ii)* manage new jobs and their dependencies. Table 3 briefly describes each function, along with the required parameters and return type.

3) EVENTS AND DATA COMPONENT

The VFuse Events and Data Component offers a software interface to orchestrate communication across components and manipulate local and remote data via the *Event Module* and the *Data Management Module*, respectively.

The Event Module controls the inter-component communication and network updates through events, which transmit information about jobs' status and data asynchronously. This module also handles the initialization of each node's workspace - comprising local data (such as the user profile, workflows, and settings), the *Gossip* and *Execution* queues (see Section IV-B4 and Section IV-B5.b), and local web workers for communicating and running jobs. Table 4 lists the available system messages to handle events.

The Data Management module abstracts the underlying IPFS services by offering a high-level interface to store, delete, and update data and workflows. Specifically, this module exploits the IPFS Mutable File System (MFS) to manage local user information, such as preferences, workflows, execution data, and publishing queues. Remote data, such as submitted workflows, shared data, and job results, are handled using the IPFS network API.

4) NETWORK COMPONENT

The VFuse network component is based on the event-driven programming paradigm and enables VFuse peers to exchange workflows and data through GossipSub [46], a publish/-subscribe protocol. GossipSub exploits the idea of *gossiping* [47]; namely, it floods the network with messages to ensure a reliable communication of data and workflows in a dynamic environment. Messages to gossip are stored in the nodes' local *Gossip* queue and are forwarded only if their TTL is not expired or the Initiator stopped the workflow. The message payload is compressed using the LZ77 [48], [49] algorithm to preserve bandwidth and memory.

Further, the communication component also provides *(i)* a configurable proxy to enable the system to use HTTPS and Web Socket Secure protocols and *(ii)* a built-in IPFS gateway, granting direct access to the IPFS resources via the HTTP protocol and avoiding the use of an external public gateway.

5) ENGINE COMPONENT

The Engine component is the core of the VFuse architecture. It comprises four interoperable modules, through which it defines (*Workflow module*) and orchestrates workflows and their computation (*Computing module*), stores users' profile information and preferences (*Identity module*), and regulates the reward mechanism (*Reward module*). A detailed description of each module follows.

a: WORKFLOW MODULE

A VFuse computation is defined using a computing workflow. Specifically, each workflow is a sequence of jobs, i.e., functions with properties and data. The programmer can

TABLE 4. VFuse events and data component messages.

Event	Description
	Inter-component communication
NODE_ STATUS	Log the status update of a node's workspace. *Values*: • Initializing, Started, Stopped, Updated, Errored.
WORKFLOW_ UPDATE	Log the status update of a workflow. *Values*: • Create, Delete, Job Running, Job Ended, Local Execution, Error.
CONSOLE_ MESSAGE	Log system-related operations.
	Network communication updates
NEW_PEER	Inform the network that a new peer has entered the network.
EXECUTION _REQUEST	Inform the network about a new submitted workflow. Each peer receiving this message adds the workflow to its *Gossip* and *Execution* queues. *Payload*: • Workflow_ID, Timestamp, CID of workflow data (may be updated).
SELECTED _JOBS	Inform the network about which set of jobs (of a workflow) a node has selected to run. *Payload*: • Workflow_ID, List of selected jobs.
EXECUTION _RESPONSE	Inform the network that a node has run a set of jobs (of a workflow). *Payload*: • Workflow_ID, List of job updates.
DROP_REQUEST	Inform the network to stop gossiping a workflow (if it is expired or the Initiator wants to unpublish it). *Payload*: • Workflow_ID

define temporal dependencies between jobs and visualize them via the workflow DAG, representing jobs as nodes and dependencies as edges. Users can check the execution of the workflows they submitted by monitoring the associated DAGs since the status of each job (i.e., node) is continuously updated. VFuse workflows are *asynchronous*, meaning that any node of the network can run them, regardless of the presence of the workflow Initiator (i.e., the node submitting the workflow).

Each VFuse web notebook represents a workspace where the user can define the workflow's jobs, their dependencies, and properties. Before submitting each workflow, users must locally build it; this procedure implies compiling and storing it in the local storage. Once submitted, the workflow is pinned in the shared IPFS cluster. This operation returns the CID associated with the workflow, which is then assigned to its Initiator. From now on, other computing nodes can retrieve the submitted workflow from the shared IPFS cluster.

Job Status: During the computation, a job may assume different statuses, each one coded with a specific color: *(i)* `Ready` (green) when available to be run, *(ii)* `Waiting` (yellow) when waiting for the termination of other jobs, *(iii)* `Repeating` (grey) when the job is (re)scheduled until the entire workflow stops or expires, *(iv)* `Terminated` (blue) or `Errored` (red) when terminates with no or one or more errors, respectively.

Job Dependencies: The execution of a job `j` may depend on the termination of one or more other tasks. In this case, the status of the job `j` will switch from `Waiting` to `Ready` when all previous tasks have been completed by at least a node of the network.

Job Repeating: VFuse allows users to define repeating jobs (marked with the status `Repeating`) to (re)schedule the same job (hence, reiterate some workflow activities) until the entire workflow stops or expires. In practice, when a repeating job terminates, its status does never change to `Terminated`. The same dependency rules also apply in this case.

b: COMPUTING MODULE

The Computing module takes care of all aspects related to orchestrating workflows over the VFuse volunteer network and exchanging the computed results among nodes.

Workflow Orchestration: Upon submitting the workflow (see Section IV-B5.a), the Initiator transparently broadcasts a new `EXECUTION_REQUEST` message. As this message is also used to update a workflow already existing on the network, every peer receiving it compares the CID of the local copy of the workflow (if existing) with the information received and updates the *Gossip* and *Execution* queues. If the node receives the workflow for the first time, it adds it to both queues.

A workflow terminates in either one of the three following cases: *(i)* all jobs of the workflow have been computed at least once, *(ii)* the TTL associated with the workflow expires, or *(iii)* the Initiator stops the workflow. In this last case, the system broadcasts a `DROP_REQUEST` message, which forces each node receiving the message to remove the given workflow from its working queues. This message is then broadcasted until the TTL of the workflow runs out.

Workflow and Job Selection: A VFuse node selects the next workflow to compute by choosing a candidate in the *Execution* queue. The final choice depends on three factors: *(i)* whether the workflow has at least a job in the status `Ready`, *(ii)* whether the workflow has at least one job that has not been selected for execution by another node, and *(iii)* the associated priority (see next paragraph). In particular, this value is proportional to the amount of reward owned by the Initiator of the workflow.

After selecting the workflow, the node chooses a job uniformly at random among all `Ready` jobs not present in the *Selected Jobs List*. This list keeps track of all jobs that will be run in the network. Specifically, each node broadcasts information about the jobs it is about to process with the

message SELECTED_JOBS. Upon receiving this message, each node adds the received job ids in the Selected Jobs List, attaching to each of them the time the message arrived and a Selection_TTL. When the TTL expires, the job ID is removed from the list. The node repeats the whole process until the maximum number of concurrent jobs is reached.

Workflow Priority: Users offering their computational capabilities to a VFuse network collect VFuse NFTs, which are saved into a public blockchain (see Section IV-B5.d). The number of NFTs (i.e., reward) owned by the Initiator impacts the priority of the requested workflow. This design choice translates into granting workflows of users with higher rewards a higher probability of being scheduled before the workflow submitted by a user with a lower amount of reward. In other words, the more a user contributes to computing jobs on a VFuse network, the sooner its submitted workflow will be completed. To avoid the starvation of workflows requested by users with low rewards, we introduced a scaling factor based on the *waiting time* of the workflow, i.e., the time that a workflow has been waiting in the Execution queue before being selected.

Hence, the priority associated with each workflow keeps into account *(i)* how much the Initiator of the requested workflow has already contributed to the volunteer network (i.e., amount of rewards) and *(ii)* the waiting time of the workflow, according to the following rule:

$$\forall w \in \mathbb{E}_v : P_{(w,t)} = \frac{\mathbb{R}(u_w)}{\sum_{w \in \mathbb{E}_v} \mathbb{R}(u_w)} \times \frac{t - t_w}{\sum_{w \in \mathbb{E}_v} (t - t_w)},$$

where

- v is a volunteer node;
- w is a workflow;
- $\mathbb{E}_v$ is the Execution queue of a node v;
- u_w is the user that has submitted the workflow w;
- $\mathbb{R}(u_w)$ is the number of NFTs (reward) owned by the user u_w;
- t is the current scheduling time, such that $t > t_w$.
- t_w is the time when the client v has received the workflow w.

The so-designed priority function balances the contribution previously given to the network (in terms of computing capabilities) by the Initiator with the time they have to attend to see their requested workflow completed. If (the Initiators of) two different workflows have the same reward, the workflow with a higher waiting time will be computed first. Clearly, if a user with a low reward submits a workflow and there is no competition, then the requested workflow will be immediately computed.

Each VFuse node computes the priority of each workflow in the status Ready in its Execution queue before the selection phase.

Job Results: A job may terminate because *(i)* its computation naturally ends by producing the desired result(s), *(ii)* its computation errored, or *(iii)* its computation time exceeded the maximum running time offered by the volunteer node.

TABLE 5. VFuse local messages.

Message	Description
BUILD_DAG _REQUEST	Message sent to a worker to require the creation of a DAG associated with a workflow.
JOB_EXECUTION _REQUEST	Message sent to a worker to run a specific job. The worker *(i)* receives the job's implementation code, and its input data, *(ii)* executes the task, and *(iii)* returns the computed results or any error(s).
LOCAL_EXECUTION _REQUEST	Message sent to test the execution of a workflow locally. The module forwards the request to the local worker pool, collects the results, and updates the job DAG.

Each VFuse node that has run a job informs the entire network of the computed result(s) (or obtained error(s)), broadcasting an EXECUTION_RESPONSE message and pinning them on the shared IPFS cluster. When a VFuse node receives the result(s) related to a job it has already computed, it then compares its locally stored result(s) with the one(s) received. If they differ, the node launches a WARNING_RESULT to inform the Initiator that there has been a divergence in the job result(s). This protocol guarantees that each workflow will eventually end (with the expected results) since the same job will not be scheduled again for the same client. Consequently, it allows two or more peers to compute results for the same task if they scheduled the same job concurrently.

Execution Backend: Each VFuse node offers its computational capabilities by providing a pool of Web Workers running different computational backends developed using WebAssembly. In more detail, Web Workers are threads running in a private scope that allow the code to be executed in a sandbox. Hence, they ensure that a VFuse client cannot be damaged by any malicious code. The Computing module automatically runs the Web Worker associated with the specific implementation language and asynchronously communicates with them via JavaScript promises. Web Workers directly enable this module to build the workflow, locally execute a workflow and run jobs. These operations are piloted using the specific messages detailed in Table 5. It is worth noting that even though VFuse is designed to be executed in a web browser, it can also be run - without any changes - in other computing environments, such as desktop machines or servers running a NodeJS server.

c: IDENTITY MODULE

The Identity module is responsible for storing users' personal information within the browser using the Events and Data Component. Specifically, this module creates a new user environment inside the browser cache when the node is initialized. Table 6 lists the environment properties (which users can personalize), specifying the parameters for *(i)* entering a specific VFuse network, *(ii)* defining the computing capabilities offered to the network, and *(iii)* tuning the network performance for receiving and sending data.

TABLE 6. VFuse user profile properties, which can be configured in the setting panel of the web client application.

Name	Description
Network properties	
Signal Server	WebRTC signal server address.
Pinning Server	IPFS cluster address.
Bootstrap Nodes	List of bootstrap nodes' addresses.
Computing properties	
Max Jobs	Maximum number of concurrent job to run.
Max Job time	Maximum execution time for a job.
Network tuning properties	
Heartbeat	Frequency of heartbeat messages sent to neighbors.
Workflow Advertisement	Frequency of workflow-related messages.
Results Advertisement	Frequency of result-related messages.

d: REWARD MODULE

The Reward module handles all the utilities concerning the rewarding mechanism provided by VFuse, such as the definition of smart contracts and the integration with IPFS to implement the VFuse NFTs.

The VFuse rewarding strategy is based on the concept of *artifacts*. Every time a node computes a job, it stores its result on IPFS, which will assign to it a unique and immutable CID. We consider this identifier (and, hence, the result written on the IPFS cluster) as an artifact produced by a user's contribution to the computation of a workflow. A VFuse peer is rewarded with a VFuse NFT per artifact. Specifically, an NFT is a smart contract [50] assigning the ownership of a particular thing (in our case, a file on IPFS) to a specific user (a VFuse node).

V. EXAMPLE WORKFLOWS

This section presents four use cases addressed with our system, describing the design and the workflow of each VFuse application. Each listing highlights the use of the VFuse API, leaving out the implementation details of the specific algorithms. Readers interested in the complete version of the code can refer to the VFuse GitHub repository.

A. MOST COMMON WORD IN A TEXT

Finding the word which occurs the most in a text is a variation of the famous *word count* problem. This class of problems is often exploited to demonstrate the benefits of distributed MapReduce paradigm, given their embarrassingly parallelizable nature.

1) VFuse WORKFLOW

The implemented VFuse workflow assigns to each job a different portion of the file that returns the number of occurrences for each word encountered (`map` function). Results are then collected and combined to compute the word with the highest number of occurrences (`reduce` function). Figure 4 depicts the resulting job DAG.

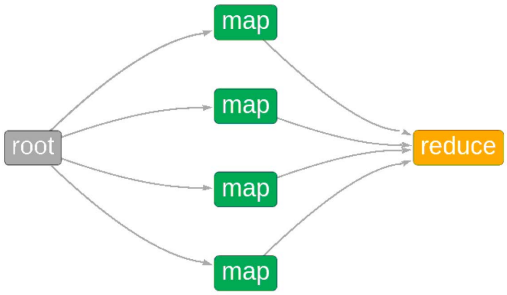


FIGURE 4. Job DAG of the *WordCount* example.

2) IMPLEMENTATION DETAILS

The implementation of the most common word in a text[2] is shown in Listing 1.

The workflow starts retrieving the size of the file using a fetch (Lines 25 – 27) to evenly split the text among the jobs (Lines 28 – 40). Each job runs a `map` function (Lines 1 – 18), implementing the core logic of the algorithm. This function receives two parameters in input: the URL of the file and two indices (bytes) limiting the chunk to compute. The `map` function uses the VFuse API `getDataFromURL()` to retrieve the associated chunk based on the input indices and computes how many times each word occurs, avoiding truncated words. Line 41 adds each job to the VFuse workflow and includes it in the group `map_group`. The `reduce` function collects the results calculated by these jobs (Line 43), finally computing the word with the maximum number of occurrences (Lines 20 – 23). It is worth highlighting how VFuse transparently handles the data dependencies between the *map* and *reduce* jobs. This mechanism is implemented via job groups; in this example, the function `reduce` waits for all jobs included in groups starting with the word corresponding to the regex "^*map*_".

B. ML ALGORITHMS COMPARISON

Binary classification is a common task studied in Machine Learning (ML) that aims to classify the instances of a dataset into two groups [51]. Binary classification problems are pretty common, and several models exist to address them. Each model has a different performance depending on the application domain; thus, it may produce optimal results on some data while performing poorly on another. Consequently, a crucial task in ML is to identify the best model for a specific problem. In this use case, we compare several binary classifiers available in the Python library scikit-learn.[3]

[2]https://github.com/luigser/js-vfuse/blob/master/packages/vfuse-core/src/examples/javascript/wordcountMapReduce_fromUrl.js

[3]https://scikit-learn.org/

```
async function map(data){
    let url =  data[0]
    let start = data[1]
    let end = data[2]
    [...]
    if (start == 0){
        let j = end+offset
        string = await VFuse.getDataFromUrl(url, start,
      j)
    } else {
        let i = start-offset
        let j = end+offset
        string = await VFuse.getDataFromUrl(url, i, j)
        local_start = local_start+offset
        local_end = j-i-offset
    }
    [...]
    return mapped
}

function reduce(data){
    [...]
    return max
}

let url = "https://172.16.149.100/wordcount.txt"
let response = await fetch(url)
let size = response.headers.get("content-length");
let num_jobs = 8
let chunk = Math.floor(size / num_jobs)
let r = size % num_jobs
let start = 0
let end = 0
for(let i=0; i<num_jobs; i++){
    if(i < r){
        start = i * (chunk + 1)
        end = start + chunk + 1
    }else{
        start = i * chunk + r
        end = start + chunk
    }
    await VFuse.addJob(map, [], [url, start, end], '
     map_group')
}
await VFuse.addJob(reduce, ['^map_'])
```

LISTING 1. Most common word in a text (JavaScript).

Specifically, this VFuse Python application trains and tests five different binary classifiers using the UCI PIMA Indian Diabetes dataset to predict whether a person has diabetes or not using the medical attributes provided. The algorithms used are Linear Discriminant Analysis (LDA), Decision tree classifier (specifically, CART), K-Neighbors Classifier (KNN), Naive Bayes (NB), and Support Vector Machine (SVM). All algorithms were run using their default parameter configurations.

1) VFuse WORKFLOW

The implemented VFuse application assigns a different binary classifier to as many jobs. Each job receives the training and testing data sets and returns the model's accuracy. Each algorithm is evaluated using 10-fold cross-validation using the same random seed to ensure consistency when splitting the training data.

2) IMPLEMENTATION DETAILS

The implementation of the comparison of ML models[4] is shown in Listing 2.

[4]https://github.com/luigser/js-vfuse/blob/master/packages/vfuse-core/src/examples/python/MLComparison.py

```
from sklearn import model_selection
[...]
import numpy

string = await VFuse.getDataFromUrl("https://raw.
     githubusercontent.com/jbrownlee/Datasets/master/
     pima-indians-diabetes.data.csv")
f = StringIO(string)
datanp = numpy.loadtxt(f, delimiter=",")
models = ['LDA', 'KNN', 'CART', 'NB', 'SVC']

def eval(input):
    [...]
    kfold = model_selection.KFold(n_splits=10,
     random_state=7, shuffle=True)
            cv_results = model_selection.
     cross_val_score(LogisticRegression(max_iter=1000),
      X, Y, cv=kfold, scoring='accuracy')
    [...]
    return model_results

def compare(data):
    [...]
    return max

result = []
for model in models:
    input = [model, datanp]
    model_res = await VFuse.addJob(eval, [], input)
    result.append(model_res)

await VFuse.addJob(compare, ['eval'], [])
```

LISTING 2. Comparison of different ML algorithms (Python).

First, the workflow loads the required libraries (Lines 1−3) with the standard Python syntax. Then, as in the previous use case, the workflow reads the data from an URL and returns a string (Line 5). In lines 22 − 25, the execution of each classification model is delegated to a different job (whose behavior is described by the function `eval()`, lines 10 − 15). The workflow then waits for the termination of all jobs and gets the computed data as input to compare the performance of all algorithms via the function `compare()` (Lines 27 and 10 − 15). The data transfer between the two computing functions - i.e., `eval()` and `compare()` - is automatically provided by VFuse.

C. PI ESTIMATION USING MONTE CARLO

Monte Carlo methods are a broad class of computational algorithms that rely on repeated random sampling to obtain numerical results. One of the Monte Carlo algorithm's primary applications is Pi's estimation. Specifically, this method considers a square space with a circle inscribed and generates several random points within this space. The value obtained by dividing the total number of points by the number of points within the circle represents an approximation of Pi. The random nature of the Monte Carlo algorithm implies that the more the points generated, the more accurate the result is. Adopting a distributed approach would allow the generation of hundreds of thousands of points, resulting in a more accurate value.

1) VFuse WORKFLOW

The workflow implemented by this VFuse application is based on a `Repeating` job, which never reaches the status

```
1  function getPoints(interval){
2      [...]
3      return {square_points : square_points,
        circle_points : circle_points}
4  }
5  function estimatePi(data){
6      [...]
7      return pi
8  }
9  let job_id = await VFuse.addJob(getPoints, [], 1000)
10 await VFuse.setRepeating(job_id)
11 job_id = await VFuse.addJob(estimatePi, ['getPoints'])
12 await VFuse.setRepeating(job_id)
```

LISTING 3. Estimating the value of Pi using Monte Carlo (JavaScript).

`Terminated` (see Section IV-B5.a). Every time a node publishes new random points, the application reevaluates the estimation of Pi.

2) IMPLEMENTATION DETAILS

The implementation of the Monte Carlo estimation of Pi[5] is shown in Listing 3. The application sets as a repeating job the function `getPoints()` (lines 9 – 10), that generates random points within a given interval (lines 1 – 4). In the same way, the application assigns the function `estimatePi` (lines 5 – 8) to a job (line 11) and declares the job as repeating (line 12).

D. SEQUENCE ALIGNMENT WITH SMITH-WATERMAN

One of the most common procedures in molecular biology is searching for similarities in protein and DNA sequences [52]. The established method to perform sequence alignment is the Smith-Waterman algorithm, based on the dynamic programming approach developed by Temple F. Smith and Michael S. Waterman. The algorithm computes optimal local alignments of two sequences identifying the two sub-sequences with maximal similarity scoring. The sequence comparison is made using the segments of all possible lengths instead of the entire sequence. The Smith–Waterman algorithm first determines the scoring matrix, then performs a trace-back measure to generate the segments with the highest similarity score using the previous scoring matrix. Although effective, this algorithm is costly in terms of computational cost since it requires a number of operations proportional to the product of the length of the sequences. Therefore, searching for similarities in large data sets requires a huge amount of time. The use of distributed computing in sequence alignment can drastically reduce the time required: the data set to match with a sequence can be split among different workers to find the most similar one.

1) VFuse WORKFLOW

The VFuse workflow assigns a portion of the sequence data set to each job. Upon receiving the data, each job computes the similarity between the string to match and the sequences included in its assigned data chunk and returns the sequence with the highest similarity score. The workflow waits for the results computed by all jobs and compares their outcomes to find the best matching sequence.

[5]https://github.com/luigser/js-vfuse/blob/master/packages/vfuse-core/src/examples/javascript/PiEstimationEndless.js

```
1  import numpy
2  def similarity(char):
3      a = char[0]
4      b = char[1]
5      return match if a == b else mismatch
6
7  def fill_matrix(string):
8      [...]
9      return [max_v, max_v_i, max_v_j, matrix]
10
11 def trace_back(data):
12     [...]
13     return [align_a, align_b]
14
15 def sw(pair):
16     [...]
17     return [max_scr, max_str, max_aligns]
18
19 def compare(data):
20     [...]
21     return [max_scr, max_str, max_aligns]
22
23 data = await VFuse.getDataFromUrl ("https://raw.
     githubusercontent.com/giusdam/data/main/dna.txt")
24 str_cmp = 'AGTACTACAAGGGTCAACCATAACCACAGCACTAGTTAT
     CTCTACTTGACAAAAACTGGCCCCAATAGCC'
25 match = 1
26 gap = -1
27 mismatch = -1
28 jobs_n = 10
29 data = data.split()
30 data = numpy.array(data)
31 split = numpy.array_split(data, jobs_n)
32 for job in split:
33     await VFuse.addJob(sw, [], [str_cmp, list(job)])
34 await VFuse.addJob(compare, ['sw'])
```

LISTING 4. DNA Matching using Smith-Waterman (Python).

2) IMPLEMENTATION DETAILS

The implementation of the standard Smith-Waterman for sequence alignment[6] is shown in Listing 4.

First, the workflow loads the data set via the function `getDataFromURL()` (line 23). Then, it splits the data set and assigns each chunks to a different job (lines 28 – 33), which computes the Smith-Waterman algorithm on its input data (function `sw()`, lines 15 – 17). The functions `similarity()` (lines 2 – 5), `fill_matrix()` (lines 7 – 9), and `trace_back()` (lines 11 – 13) are three auxiliary functions used by the matching algorithm. Finally, the workflow waits for the termination of all jobs and gets the computed data as input to find the sequence with the maximum alignment score via the function `compare()` (Lines 19 – 21).

VI. DISCUSSION AND LIMITATIONS

We performed a preliminary evaluation of the VFuse prototype, experimenting with the *Most Common Word* example described in the previous section. Specifically, we benchmarked and analyzed the running time of computing the most frequent word among 64 equally-sized files for a total

[6]https://github.com/luigser/js-vfuse/blob/master/packages/vfuse-core/src/examples/python/SmithAndWaterman.py

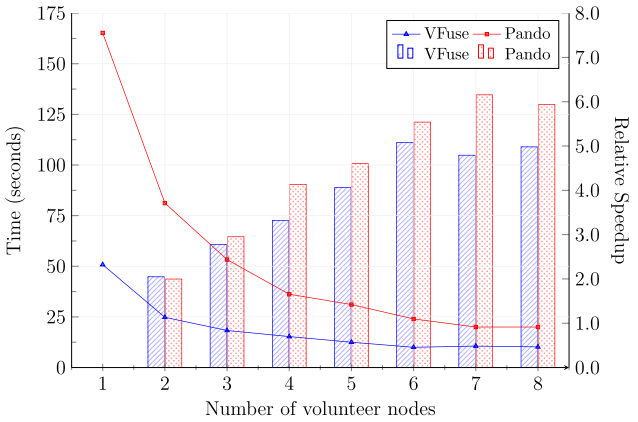


FIGURE 5. Running time and relative speedup of the P2P BBVC VFuse platform compared against the centralized BBVC system Pando.

of 4GB. We further exploited this use case to compare the performance of VFuse against the centralized BBVC system Pando [34]. We used the pre-configured Docker Image on the Pando official repository[7] and adapted the example according to its execution logic. We ran the experiment on a cluster of Windows 10 Pro desktop machines equipped with an Intel(R) Core(TM) i7-8700T CPU, 16GB RAM, 512GB SSD, and the Google Chrome browser. We employed the default settings for VFuse, configured to utilize four threads per node, while we chose an equivalent device configuration for Pando. It is worth stressing that, in this evaluation, we only considered Pando as it was the only framework providing a ready-to-run Docker container. Unfortunately, all other publicly available BBVC repositories required outdated dependencies, which made the systems too complex to run.

Figure 5 depicts the running time (left y-axis, lines) and the relative speedup (right y-axis, bars) at the variation of the number of volunteer nodes from 1 to 8. As shown in the plot, both systems exhibit comparable scalability. In particular, VFuse provides better performance when the number of nodes is low, while the gap between the performance of the two systems decreases when the number of nodes increases.

Another critical point worth mentioning is related to possible privacy and security issues that may arise in the context of BBVC systems. Specifically, at this stage, the proposed architecture does not consider possible leaks of data privacy since data are publicly available on the blockchain and IPFS. Nonetheless, this architectural choice does not harm the system's usability as computation using private data is not a frequent use case in the context of volunteer computing. Currently, VFuse relies on a standard approach in VC-based systems to ensure that the required computation and its outcome have not been corrupted in the process. As described in the Paragraph *Job results* of Section IV-B5.b, each job's result is assessed by every node involved in the computation of the same job. Still, blockchain technology enables the implementation of more complex security mechanisms.

[7]https://github.com/elavoie/pando-computing

VII. CONCLUSION AND FUTURE WORK

The paradigm of (BB)VC gained attention over the last years as a potential tool for allowing researchers and companies to access the enormous computing capabilities over the Internet for solving high-demand computational problems.

In this work, we proposed VFuse, a novel BBVC architecture that offers *(i)* a ready-to-access network through web browsers and *(ii)* a multi-language programming environment thanks to WebAssembly, *(iii)* stimulates users' participation by providing a secure and transparent rewarding mechanism based on Blockchain technology, and *(iv)* specifies an innovative definition of users' participation via NFTs that guarantee the user ownership of computing results. We demonstrated the advantages of VFuse and its added value by comparing our platform with the most common BBVC and discussing four example applications. A prototype of VFuse and the presented examples are freely available on GitHub.

Currently, we are working on developing the rewarding mechanism implemented with the Ethereum blockchain and IPFS. In future work, we plan to perform systematic experiments to assess the performance gain of our platform against other well-known systems. We also aim to introduce workflows with an associated deadline, whose importance could be reflected in the workflow priority and the rewarding strategy. Another interesting future directions is integrating VFuse into standard web pages to allow visitors of websites, such as online news and game sites, blogs, and social networks, to participate in a VFuse volunteer network by visiting some dedicated pages offered by the owner of the service. Future research should also consider designing a more sophisticated scheduling algorithm to allow a fairer job execution, including more advanced security mechanisms, extending the workflow definition by adding more programming constructs for enriching the programming model, and developing other programming backends supported by WebAssembly.

REFERENCES

[1] Gartner. *Gartner Says Worldwide PC Shipments Grew 1% in Third Quarter of 2021*. Accessed: Nov. 11, 2021. [Online]. Available: https://www.gartner.com/en/newsroom/press-releases/2021-10-11-gartner-says-worldwide-pc-shipments-grew-1-percent-in-third-quarter-of-2021

[2] Statista. *Personal Computer (PC) Unit Shipments Worldwide From 2009 to 2022, by Quarter*. Accessed: Dec. 4, 2022. [Online]. Available: https://www.statista.com/statistics/264467/global-pc-shipments-since-1st-quarter-2009/

[3] Gartner. *Gartner Says Global Smartphone Sales Grew 6% in 2021*. Accessed: Feb. 2, 2022. [Online]. Available: https://www.gartner.com/en/newsroom/press-releases/2022-03-01-4q21-smartphone-market-share

[4] Statista. *Number of Smartphones Sold to end Users Worldwide From 2007 to 2021*. Accessed: Mar. 14, 2022. [Online]. Available: https://www.statista.com/statistics/263437/global-smartphone-sales-to-end-users-since-2007/

[5] DatarePortal. *Digital Around the World*. Accessed: Apr. 22, 2022. [Online]. Available: https://datareportal.com/global-digital-overview

[6] *Still Waiting for Exascale: Japan'S Fugaku Outperforms all Competition Once Again*. Accessed: Apr. 22, 2022. [Online]. Available: https://www.top500.org/

[7] DatarePortal. *Digital 2019: Global Digital Overview*. Accessed: Apr. 29, 2022. [Online]. Available: https://datareportal.com/reports/digital-2019-global-digital-overview

[8] O. Nov, D. Anderson, and O. Arazy, "Volunteer computing: A model of the factors determining contribution to community-based scientific research," in *Proc. 19th Int. Conf. World Wide Web (WWW)*. New York, NY, USA: Association for Computing Machinery, 2010, pp. 741–750, doi: 10.1145/1772690.1772766.

[9] T. Fabisiak and A. Danilecki, "Browser-based harnessing of voluntary computational power," *Found. Comput. Decis. Sci.*, vol. 42, no. 1, pp. 3–42, 2017.

[10] J. A. Morell, A. Camero, and E. Alba, "Jsdoop and tensorflow.js: Volunteer distributed web browser-based neural network training," *IEEE Access*, vol. 7, pp. 158671–158684, 2019.

[11] MIT Digital Currency Initiative and the Center for Civic Media. *The Decentralized Web*. Accessed: May 26, 2022. [Online]. Available: https://dci.mit.edu/decentralizedweb

[12] A. Antelmi, G. D'Ambrosio, A. Petta, L. Serra, and C. Spagnuolo. *vFuse Public Repository*. Accessed: Apr. 29, 2022. [Online]. Available: https://github.com/luigser/js-vfuse.git

[13] T. M. Mengistu and D. Che, "Survey and taxonomy of volunteer computing," *ACM Comput. Surv.*, vol. 52, no. 3, pp. 1–35, Jul. 2019.

[14] *Great Internet Mersenne Prime Search—Gimps*. Accessed: Apr. 22, 2022. [Online]. Available: https://www.mersenne.org/

[15] *Distributed.net*. Accessed: Apr. 22, 2022. [Online]. Available: https://www.distributed.net/

[16] *Seti@home*. Accessed: Apr. 22, 2022. [Online]. Available: https://setiathome.berkeley.edu/

[17] *Folding@home*. Accessed: Apr. 22, 2022. [Online]. Available: https://foldingathome.org/

[18] D. P. Anderson, "BOINC: A system for public-resource computing and storage," in *Proc. 5th IEEE/ACM Int. Workshop Grid Comput.*, Nov. 2004, pp. 4–10.

[19] G. Fedak, C. Germain, V. Neri, and F. Cappello, "XtremWeb: A generic global computing system," in *Proc. 1st IEEE/ACM Int. Symp. Cluster Comput. Grid (CCGrid)*, 2001, pp. 582–587.

[20] M. Kuhara, N. Amano, K. Watanabe, Y. Nogami, and M. Fukushi, "A peer-to-peer communication function among web browsers for web-based volunteer computing," in *Proc. 14th Int. Symp. Commun. Inf. Technol. (ISCIT)*, 2014, pp. 383–387.

[21] W. Li, W. Guo, and E. Franzinelli, "Achieving dynamic workload balancing for P2P volunteer computing," in *Proc. 44th Int. Conf. Parallel Process. Workshops*, Sep. 2015, pp. 240–249.

[22] I. Al Ridhawi, M. Aloqaily, and Y. Jararweh, "An incentive-based mechanism for volunteer computing using blockchain," *ACM Trans. Internet Technol.*, vol. 21, no. 4, pp. 1–22, Jul. 2021.

[23] R. Lamba, V. Jain, and D. Saini, "Calculating the proof of work using volunteer computing," in *Proc. 3rd Int. Conf. Comput. Inform. Netw.* Singapore: Springer, 2021, pp. 427–437.

[24] D. Lázaro, J. M. Marquès, and X. Vilajosana, "Flexible resource discovery for decentralized P2P and volunteer computing systems," in *Proc. 19th IEEE Int. Workshops Enabling Technol., Infrastruct. Collaborative Enterprises*, Jun. 2010, pp. 235–240.

[25] B. Shan, "A design of volunteer computing system based on blockchain," in *Proc. IEEE 13th Int. Conf. Comput. Res. Develop. (ICCRD)*, Jan. 2021, pp. 125–129.

[26] Y. Kim and J. Park, "Hybrid decentralized pbft blockchain framework for openstack message queue," *Hum.-Centric Comput. Inf. Sci.*, vol. 10, no. 1, p. 31, 2020.

[27] *The Golem Project*. Accessed: Apr. 22, 2022. [Online]. Available: https://www.golem.network/

[28] *Boid, White Paper*. Accessed: Apr. 22, 2022. [Online]. Available: https://www.boid.com/statics/Boid-WhitePaper-v2.pdf

[29] I. Charalampidis, D. Berzano, J. Blomer, P. Buncic, G. Ganis, R. Meusel, and B. Segal, "CernVM webAPI—Controlling virtual machines from the web," *J. Phys., Conf. Ser.*, vol. 664, no. 2, Dec. 2015, Art. no. 022010, doi: 10.1088/1742-6596/664/2/022010.

[30] C. Cusack, C. Martens, and P. Mutreja, "Volunteer computing using casual games," in *Proc. Future Play Int. Conf. Future Game Design Technol.*, 2006, pp. 1–8.

[31] Y. Pan, J. White, Y. Sun, and J. Gray, "Gray computing: An analysis of computing with background Javascript tasks," in *Proc. IEEE/ACM 37th IEEE Int. Conf. Softw. Eng.*, vol. 1, May 2015, pp. 167–177.

[32] H. Matsuo, S. Matsumoto, Y. Higo, and S. Kusumoto, "Madoop: Improving browser-based volunteer computing based on modern web technologies," in *Proc. IEEE 26th Int. Conf. Softw. Anal., Evol. Reeng. (SANER)*, Feb. 2019, pp. 634–638.

[33] D. Dias and L. Veiga, "Browsercloud.Js: A distributed computing fabric powered by a P2P overlay network on top of the web platform," in *Proc. 33rd Annu. ACM Symp. Appl. Comput.*, Apr. 2018, pp. 2175–2184.

[34] E. Lavoie, L. Hendren, F. Desprez, and M. Correia, "Pando," in *Proc. 20th Int. Middleware Conf.*, 2019, pp. 96–109.

[35] E. Lavoie, L. Hendren, F. Desprez, and M. Correia, "GeNet: A quickly scalable fat-tree overlay for personal volunteer computing using webRTC," in *Proc. IEEE 13th Int. Conf. Self-Adapt. Self-Organizing Syst. (SASO)*, Jun. 2019, pp. 117–126.

[36] K. S. Sagar Bharadwaj, S. Dharanikota, A. Honawad, and K. Chandrasekaran, "Collabchain: Blockchain-backed trustless web-based volunteer computing platform," in *Proc. 18th Int. Conf. Comput. Inf. Syst. Ind. Manag. (CISIM)*, Belgrade, Serbia, Sep. 2019, pp. 509–522.

[37] A. Haas, A. Rossberg, D. L. Schuff, B. L. Titzer, M. Holman, D. Gohman, L. Wagner, A. Zakai, and J. Bastien, "Bringing the web up to speed with webassembly," in *Proc. 38th ACM SIGPLAN Conf. Program. Lang. Design Implement.* New York, NY, USA: Association for Computing Machinery, 2017, pp. 185–200.

[38] Y. Gil, E. Deelman, M. Ellisman, T. Fahringer, G. Fox, D. Gannon, C. Goble, M. Livny, L. Moreau, and J. Myers, "Examining the challenges of scientific workflows," *Computer*, vol. 40, no. 12, pp. 24–32, 2007.

[39] *Pyodide*. Accessed: Apr. 22, 2022. [Online]. Available: https://pyodide.org/

[40] *Emscripten*. Accessed: Apr. 22, 2022. [Online]. Available: https://emscripten.org/

[41] *Libp2P Project*. Accessed: Apr. 22, 2022. [Online]. Available: https://libp2p.io/

[42] P. Maymounkov and D. Mazières, "Kademlia: A peer-to-peer information system based on the XOR metric," in *Peer-to-Peer Systems* (Lecture Notes in Computer Science), vol. 2429. Berlin, Germany: Springer, 2002, pp. 53–65.

[43] S. L. Garfinkel, "Public key cryptography," *Computer*, vol. 29, no. 6, pp. 101–104, Jun. 1996.

[44] *IPFS*. Accessed: Apr. 22, 2022. [Online]. Available: https://ipfs.io/

[45] *Ethereum*. Accessed: Apr. 22, 2022. [Online]. Available: https://ethereum.org/

[46] D. Vyzovitis, Y. Napora, D. McCormick, D. Dias, and Y. Psaras, "GossipSub: Attack-resilient message propagation in the filecoin and ETH2.0 networks," 2020, *arXiv:2007.02754*.

[47] J. Leitao, J. Pereira, and L. Rodrigues, "HyParView: A membership protocol for reliable gossip-based broadcast," in *Proc. 37th Annu. IEEE/IFIP Int. Conf. Dependable Syst. Netw. (DSN)*, Jun. 2007, pp. 419–429.

[48] J. Ziv and A. Lempel, "Compression of individual sequences via variable-rate coding," *IEEE Trans. Inf. Theory*, vol. IT-24, no. 5, pp. 530–536, Sep. 1978.

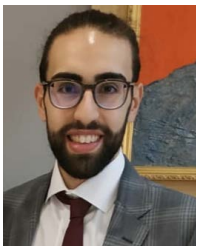
[49] J. Ziv and A. Lempel, "A universal algorithm for sequential data compression," *IEEE Trans. Inf. Theory*, vol. IT-23, no. 3, pp. 337–343, May 1977.

[50] I. Team. *Mint an NFT With IPFS*. Accessed: Apr. 22, 2022. [Online]. Available: https://docs.ipfs.io/how-to/mint-nfts-with-ipfs/

[51] Z. Ligang and K. K. Lai, "Benchmarking binary classification models on data sets with different degrees of imbalance," *Frontiers Comput. Sci. China*, vol. 3, pp. 205–216, Jun. 2009.

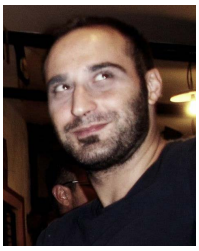
[52] S. A. Manavski and G. Valle, "Cuda compatible GPU cards as efficient hardware accelerators for smith-waterman sequence alignment," *BMC Bioinf.*, vol. 9, no. 2, p. S10, 2008.

ALESSIA ANTELMI graduated with full marks in computer science at the Università degli Studi di Salerno, Italy, in 2018, under the supervision of Prof. Vittorio Scarano. Before starting her Ph.D. studies in November 2018, she joined the Unit for Social Semantics, Data Science Institute, Galway, Ireland, lead by Prof. John Breslin, under the Erasmus+ Traineeship Grant. She currently holds a postdoctoral position with the ISISLab laboratory, Università degli Studi di Salerno, collaborating with Prof. Vittorio Scarano, Prof. Gennaro Cordasco, and Dr. Carmine Spagnuolo. Her research interests include the fields related to complex networks, especially hypergraphs, and agent-based models.



GIUSEPPE D'AMBROSIO graduated with full marks in computer science at the Università degli Studi di Salerno, Italy, in 2020, under the supervision of Prof. Vittorio Scarano. He is currently pursuing the Ph.D. degree with the ISISLab laboratory, Università degli Studi di Salerno, supervised by Prof. V. Scarano and Dr. Carmine Spagnuolo. His research interests include the reliability and scalability of scientific computing applications exploiting advanced computational paradigms, including cloud computing, parallel, and distributed computing.

ANDREA PETTA is currently pursuing the Ph.D. degree in computer science. His Ph.D. thesis concerns distributed systems and volunteer cloud computing. His goal is to build a public, reliable, cost-effective, secure, efficient, green, and privacy-aware volunteer cloud platform for scientific and general-purpose cloud computing. He is also interested in web technologies and computer-supported cooperative work.



LUIGI SERRA is currently pursuing the Ph.D. degree in computer science with the Dipartimento di Informatica, University of Salerno, Italy. He is mainly interested in topics, such as parallel computing, peer-to-peer architectures, and distributed volunteer systems. He has worked on web-based collaborative systems, and he has been with KAUST, Saudi Arabia, to join the Prof. Cavallo team working on topics, such as computational chemistry and bioinformatics.

CARMINE SPAGNUOLO received the master's degree *(cum laude)* in computer science from the Università degli Studi di Salerno, in 2013, and the Ph.D. degree in computer science from the Università degli Studi di Salerno, in 2017, under the supervision of Prof. Vittorio Scarano and Prof. Gennaro Cordasco. In 2012, he got a grant from the Office of Naval Research (ONR) for visiting the George Mason University (GMU). In May 2017 and from October to December 2017, he was a Visiting Student with the University of Chicago and the Argonne National Laboratory, under the supervision of Dott. Jonathan Ozik and exploiting a grant from ANL. In December 2019, he was a Visiting Researcher with George Mason University (GMU) under the supervision of Prof. Sean Luke. He is currently a Postdoctoral Researcher with the Università degli Studi di Salerno, where he is also a Senior Member of the ISISLab laboratory. He is the coauthor of more than 30 papers in international refereed journals and conferences. His research interests include parallel algorithms, distributed systems, graph theory, social networks, and agent-based simulations.