SOFTWARE—PRACTICE AND EXPERIENCE
*Softw. Pract. Exper.* 2014; **00**:1–28
Published online in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/spe

# AVoCLOUDY: A Simulator of Volunteer Clouds

Stefano Sebastio[1,4*], Michele Amoretti[2] and Alberto Lluch Lafuente[3,4]

[1] *LIMS, London Institute of Mathematical Sciences, London, UK*
[2] *Dipartimento di Ingegneria dell'Informazione, University of Parma, Italy*
[3] *DTU Compute, Technical University of Denmark, Denmark*
[4] *IMT Institute for Advanced Studies, Lucca, Italy*

SUMMARY

The increasing demand of computational and storage resources is shifting users towards the adoption of cloud technologies. Cloud computing is based on the vision of *computing as utility*, where users no more need to buy machines, but simply access remote resources made available on-demand by cloud providers. The relationship between users and providers is defined by a Service Level Agreement, where the non-fulfillment of its terms is regulated by the associated penalty fees. Therefore, it is important that the providers adopt proper monitoring and managing strategies. Despite their reduced application, intelligent agents constitute a feasible technology to add autonomic features to cloud operations. Furthermore, the volunteer computing paradigm — one of the ICT trends of the last decade — can be pulled alongside traditional cloud approaches, with the purpose to "green" them. Indeed, the combination of data center and volunteer resources, managed by agents, allows one to obtain a more robust and scalable cloud computing platform. The increased challenges in designing such a complex system can benefit from a simulation-based approach, to test autonomic management solutions before their deployment in the production environment. However, currently available simulators of cloud platforms are not suitable to model and analyze such heterogeneous, large-scale and highly dynamic systems. We propose the AVoCLOUDY simulator to fill this gap. This paper presents the internal architecture of the simulator, provides implementation details, summarizes several notable applications, and provides experimental results that measure the simulator performance and its accuracy. The latter experiments are based on real-world world-wide distributed computations on top of the PlanetLab platform. 

Received . . .

KEY WORDS: Cloud Computing; Volunteer Computing; Autonomic Computing; Distributed Computing; Discrete Event Simulation

## 1. INTRODUCTION

Provisioning, using and maintaining computational resources as services is a hard challenge. On the one hand, there is an increasing demand of such services due to the increasing role of software in our society, while on the other hand the amount and variety of computational resources is growing due to the pervasiveness of computational devices in our lives. The complexity of such a problem can only be mastered by resorting to suitable technologies based on well-studied paradigms. Three prominent examples and ICT trends of the last decade, that can be fruitfully combined, are *cloud computing*, *autonomic computing* and *volunteer computing*.

*Correspondence to: LIMS, London Institute of Mathematical Sciences, 22 South Audley St. Mayfair, London W1K 2NY, UK. E-mail: stefano.sebastio@alumni.imtlucca.it

Contract/grant sponsor: Research partially supported by the EU through the HOME/2013/CIPS/AG/4000005013 project CI2C, FP7-ICT Integrated Project 257414 ASCENS, STReP project 600708 QUANTICOL, and the Italian PRIN 2010LHT4KM CINA. The contents of the paper do not necessarily reflect the position or the policy of funding parties.

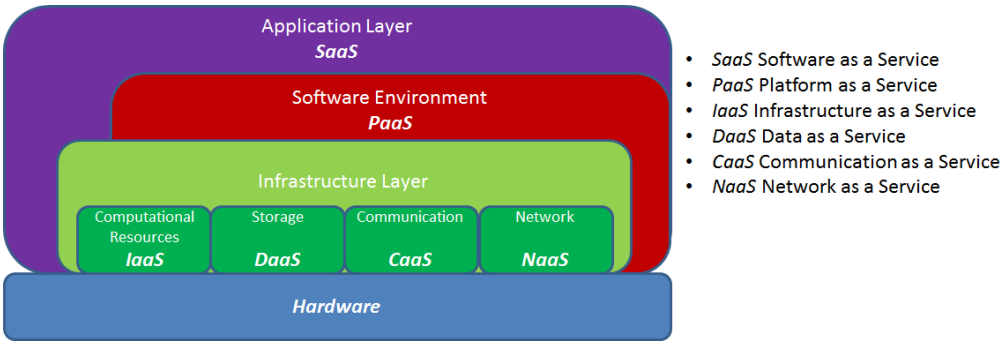


Figure 1. Cloud computing stack architecture.

**Cloud Computing.** Cloud computing is a recent ICT technology based on the concept of *computing as utility*. In this vision, users do not buy nor maintain their own computational resources, but rather use services based on their requirements, regardless of where the resources are hosted or how they are deployed. In the Cloud, resources are managed with a multi-tenancy approach, where each request is bound to a contract between the service provider and the consumer. Such a contract is specified in terms of Service Level Agreements (SLAs) and associated to penalty fees, in case of violation of the required Quality of Service (QoS). Thus, it is necessary that the Cloud is provided with performance monitoring functions. Another important feature of cloud computing is its ability to provide *elastic* high-performance services, adapting to user and application needs.

Cloud services can be of different nature, from high-level services such as applications to low-level services such as hardware resources. The reference model of cloud services is illustrated in Figure 1.

**Autonomic Computing and Clouds.** One of the greatest challenges of cloud computing is the efficient use of resources, while guaranteeing the fulfillment of SLAs. To this aim, particular attention is given to autonomic computing techniques, which support the development of clouds as self-managing systems. Cloud features should automatically adapt to their internal status and the dynamically changing environment, without human intervention. Self-management may involve system maintenance, awareness, evolution, configuration, healing, tuning and optimization.

**Grid Computing and Clouds.** Before the advent of the cloud, grid computing has been one of the most important technologies adopted by researchers to obtain high performance computing capabilities at low cost. The grid paradigm assumes the presence of machines deployed in different domains, running specialized middleware for the aggregation and provision of computational resources to participants of the same virtual organization (*e.g.*, researchers working in the same field). Grid users are required to have a high level of expertise. For instance, to suitably coordinate the workflow of a distributed application, a programmer has to be aware of low-level details such the hetereogeneity of computational resources like processors or network channels. Instead, with the cloud paradigm, all the shared resources are virtualized by the "cloud" and offered in an homogenous means that hides unnecessary low-level technicalities. A typical example would be resoure-as-a-service offering a unique super-computer to the programmer, that does not need to know if such a service is built upon a cluster or a grid or something else.

**Volunteer Computing and Clouds.** The above mentioned proliferation of computational resources in our society (e.g. in smartphones, in vehicles, in homes, etc.) can be turned into an opportunity. Indeed, the traditional cloud technology can be combined with mechanisms adopted from another paradigm that has gained popularity and research attention in the last decades: *volunteer computing*. The volunteer paradigm considers the willingness of users to donate their spare and unused computational resources.



DOI: 10.1002/spe

Volunteer cloud computing [1] (i.e., the combination of P2P and cloud computing technologies, also referred as *P2P Cloud*) should be considered as a companion force to enhance traditional clouds in specific domains. For example, it is the case of academic institutions and industrial research centers that share the spared resources of their labs (not used 24 hours a day or underused for browsing and word processing activities) with other sites, to run complex large-scale simulations.

It is worth noting that grid computing envisages volunteer participation as well. However, to participate in a grid, an institution has to provide machines that must be configured and fully devoted to the purposes of the specific virtual organization. A paradigmatic example is the Worldwide LHC Computing Grid [2], whose machines are distributed worldwide, provided by different research institutes. Conversely, volunteer cloud computing leverages on idle resources of participant machines.

The volunteer cloud supports both computing- and storage-oriented cloud-based applications. Thus, MapReduce and streaming applications can be successfully implemented in a volunteer cloud fashion [1]. Instead, service-oriented applications (such as multi-tier web applications) are not suitable for being executed in a volunteer cloud environment, because of the volatile presence of the nodes participating in the network.

**Opportunities for volunteer cloud computing.** The benefits of combining the cloud and volunteer paradigms are the reduction of economic costs and consumed power at data centers [3–6]. The reduction of economic costs comes from better exploitation of available and unused computational resources, for the use of those made available free of charge from the volunteer community. Moreover, harnessing the distributed computing power, maintenance costs of massive data centers are reduced in terms of power supplies and heat dissipation, thus embracing the green computing initiative. Last but not least, volunteer cloud computing is beneficial in case of natural catastrophes that damage data centers, or when governments place restrictions on the location of sensitive data. Indeed, volunteer cloud computing prevents the presence of a single entity that owns and controls the cloud, with less risk in the vendor lock-in problem.

To foster the participation of volunteers while respecting the SLAs, it is possible to contemplate the registration of each volunteer (e.g., through a system of signed certificates) and the assignment of credits for their participation. For example, the PlanetLab platform [7] fosters the participation to the P2P network, requiring to share a minimum number of machines (with a specified minimum system requirements) to gain access to all other shared machines. We have precisely used this platform in our experiments aimed at validating the confidence of our simulator, as we shall discuss later.

In recent years, the volunteer cloud paradigm is gaining attention, alongside to the increased diffusion of cloud technologies [8]. For example, in France at INRIA (Clouds@Home), in Italy at the universities of Bologna (Peer-to-Peer Cloud System) and Messina (Cloud@Home), and in general within the European Community (NaDa — NanoDatacenter [9], EDGI — European Desktop Grid initiative [10]), the CERN's volunteer cloud [11], in the U.S. within the National Science Foundation (NSF) which supports the BOINC project (to date with five research grants [12]), HTCondor (to date with four grants [13]), Seattle (to date with three grants [14]), SETI@home (with support from NSF and NASA [15]) and the Federal University of Campina Grande in Brasil with the OurGrid initiative [16]. Other works have modelled resource discovery algorithms in volunteer clouds relying on queueing theory [17, 18] and explored the opportunity for distributing Matlab simulations in a university volunteer infrastructure [19].

One of the major drawbacks of volunteer cloud computing is the challenge posed by the increasing complexity in the management of heterogeneous and distributed computing resources. Despite the mentioned considerable amount of research and implementation effort, the current landscape lacks of a simulator targeting the peculiarities of the volunteer cloud environment which would be beneficial to prototype and analyse new solutions to the management of resources in volunteer clouds. Our work aims at filling this gap.

**Contribution.** Our vision advocates for the realization of a volunteer cloud computing architecture relying on the autonomic computing approach, realized through intelligent agents. The increased complexity of such a paradigm demands more attention to the design phase. A simulation-based



DOI: 10.1002/spe

approach can alleviate the deployment risk, giving confidence on the expected performance. To the best of our knowledge, the most used and affordable cloud simulators [20–22] only consider the presence of few interconnected data centers and thus cannot scale to the size of a typical volunteer network, and model the associated challenges. The proposed AVOCLOUDY simulator allows one to consider richer scenarios with data centers, volunteers or both types of nodes together. It is worth to remark that its layered architecture and the use of agents simplifies the transition from the simulator to the production environment. This has facilitated, for instance, the porting of some case studies from the simulator to the PlanetLab platform used to validate the confidence of the simulator.

AVOCLOUDY is a simulator of volunteer clouds targeted for the evaluation of task distribution and network management protocols. We do not focus here on the details of such protocols (we refer to our previous research, summarized in Section 5). Instead, we provide a detailed description of the simulator, using a tutorial form, as much as possible.

**Synopsis.** The paper is structured as follows. Section 2 introduces the reference platforms for AVOCLOUDY, and in particular the SCIENCE CLOUD. Section 3 presents the AVOCLOUDY architecture, whose implementation details are provided in Section 4. Some works that have already benefit from AVOCLOUDY are briefly described in Section 5, while Section 6 presents a simple experiment and a performance evaluation. A confidence validation based on a worldwide distributed application running on top of the PlanetLab network is presented in Section 7. Finally Sections 8 and 9 conclude the work respectively with an overview of the other available cloud simulators, and some final remarks.

## 2. EXISTING PLATFORMS

The proposed solution has its roots on the architecture of the SCIENCE CLOUD [23], an *autonomic and cooperative volunteer cloud computing platform* developed within the European project ASCENS [24]. The SCIENCE CLOUD is a decentralized platform whose goal is the sharing of computational resources among scientific communities. Differently from the traditional definition of cloud, the volunteer cloud is characterized by: (i) openness and dynamism (i.e., participants can join and leave the network at any moment), (ii) lack of a centralized control (i.e., participants directly interact in a peer-to-peer fashion). Shared resources are mainly used to execute distributed applications or services.

In this work, we refer to the abstract entity that shares its resources as *node, volunteer* or *participant* interchangeably. The different types of *agent* are the autonomic components that act in the network at different layers of the architecture to fulfill certain goals. The physical resources shared by a volunteer constitute a *device*.

There are different types of devices participating to the volunteer network. They can be grouped into three main categories: (i) dedicated servers and data centers, (ii) desktops and laptops, (iii) mobile devices. The groups are characterized by a different amount of available resources (CPU, memory and energy) and online presence. Anyhow, also in the same category, nodes are *heterogeneous* (i.e., they may offer different virtual resources) and highly *dynamic* (i.e., they may enter or leave the system at any time). Furthermore load and resources may change during the nodes execution.

**Devices.** Dedicated servers and data centers are made available by universities or research centers to the research community and, except for update or malfunctioning are always online. Moreover, generally they are made of multisocket - multiprocessor architectures, connected to high bandwidth network and to power source. Desktops and laptops are devices willing to share a quote of their resources when not needed locally. Their online presence is discontinuous and unknown a priori, they are usually connected to residential networks and have a limited amount of resources that prevent (or make less attractive) the execution of multiple Virtual Machines (VMs) at the same time. Mobile devices are characterized by limited power capacity that affects their online presence duration. Energy constraints and limited resources make mobile devices nothing more than generators



DOI: 10.1002/spe

of service requests (just like the *cloudlets* proposed by Microsoft [25]), leaving the network once such requests have been satisfied. Examples where mobile devices can benefit to request the cloud execution are computational intensive applications in which also the latency affects the user-perceived Quality of Experience (QoE), like in the augmented reality domain.

**Volunteer implementations.** Nowadays many concrete implementations of volunteer computing are available, such as: BOINC [26], HTCondor [27], OurGrid [28], Seattle [29] and SETI@home [30]. These architectures rely on a central server, the *node manager*, that is in charge of receiving presence messages from new nodes in the network. The node manager collects submitted task execution requests (often in batch mode) and finds the most promising nodes to execute them. The centralized control approach presents different kinds of weakness like the central point of failure and it can constitute a bottleneck when many nodes are connected (as it happens in a typical volunteer network). Moreover a (*sub*-)optimal decision about which nodes to involve in the execution of a given distributed application cannot be realized in practice since it requires a global knowledge of characteristics and load for each node. The only operations that can be done in a complete distributed manner are data storage and location in the network e.g., via a Distributed Hash Table (DHT) implemented in the P2P network.

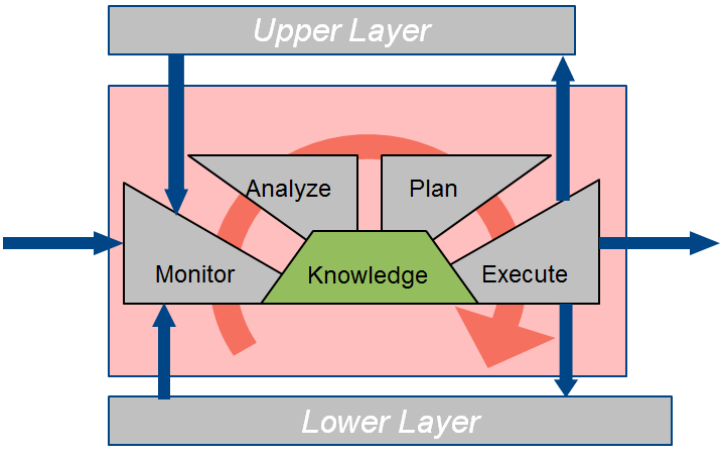


Figure 2. Agent layer according to the MAPE-K model.

**Autonomicity.** One of the main differences among the mentioned implementations and the SCIENCE CLOUD regards the *autonomic* aspects. The autonomic behavior of each node can be under the control of commercial or academic entities. Each entity can have its goal and policies according to which its machines must share and use the resources available in the network. Internally, all the layers are structured according to the IBM's reference architecture *MAPE-K* for autonomic systems [31], as illustrated in Figure 2. Briefly, the agent in each of the node's layer *monitors* itself and the ones closer (upper, lower and the corresponding agent layer of other nodes) for applications, resources and environment, *analyzes* its status, devises *plans* to improve its behavior (e.g., application execution, resources usage), and *enacts* those plans. A *knowledge* base supports this cycle of activities.

## 3. AVOCLOUDY ARCHITECTURE

This section illustrates the architecture of AVOCLOUDY. It starts with a description of the underlying tools, then continues with an overall picture of its volunteer network and the main interactions occurring among its autonomic entities. Finally a layer-by-layer bottom-up architectural description is provided.



DOI: 10.1002/spe

AVOCLOUDY[†] is built on top of DEUS [32, 33], a general-purpose, multi-platform, open-source, Java-based, discrete event simulation (DES) tool. Despite there are many free or commercial simulator tools, the flexibility of DEUS to support the analysis of every kind and size of complex system, like the one considered in our scenario, have led us to choose it. There are three main classes in DEUS: (i) *nodes*, the entities that interact in the system; (ii) *events*, that define internal actions and interactions among the nodes, or with the environment; (iii) *processes*, either stochastic or deterministic ones (many already implemented in the common processes library) that defines the timeliness of the events.

DEUS has been enriched with the distributed statistical analysis capabilities offered by MultiVeStA [34, 35]. The latter tool provides a language (MultiQuaTEx) to express system properties of interest in a compact fashion. Such properties are evaluated by performing independent distributed simulation runs, until the required accuracy is met.

The obtained toolchain is depicted in Figure 3. The cloud model and its scenario are specified by means of AVOCLOUDY, which in turn relies on DEUS to manage the basic DES functionalities (e.g., process generation, events queue, etc.). The system properties of interest are formally specified by means of MultiQuaTEx. Example of such properties are: mean number of tasks on queue, applications executed successfully, execution requests spread in the network, variance of executed tasks per node. MultiVeStA takes the MultiQuaTEx specification as input, together with the desired accuracy for each property expressed as the size of the Confidence Interval (CI) at a given percentage, evaluated according to the Student's t-test. Independent simulation runs are distributed on different cores or machines and are performed until the required accuracy is met. Finally MultiVeStA provides in output, for each property: the expected value, the CI (which can be larger than required, if the maximum number of simulation runs is reached) and the variance. If these properties concern the evaluation of the system at different points in time a tab-separated output file is generated together with a GnuPlot script and the corresponding plot. As the description of the integration of DEUS with MultiVeStA is out of the scope of this work, we refer to the recent work by Sebastio and Vandin [34] for a deeper discussion.

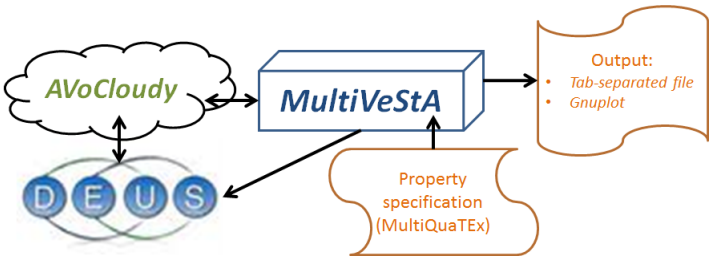


Figure 3. Toolchain: DEUS + MultiVeStA + AVOCLOUDY

The internal architecture of an AVOCLOUDY node is depicted in Figure 4, where the arrows represent the interactions among agents. It has many similarities with the architecture proposed in the SCIENCE CLOUD. The main architectural difference between SCIENCE CLOUD and AVOCLOUDY is that the latter has a modularized structure, where each layer is constituted by an agent that autonomously cooperates with other agents in the same and in other nodes. Each layer in the proposed stack corresponds to one of the packages according to which the simulator is organized.

It is worth to note that the considered architecture (Figure 4) significantly differs from the one of a typical cloud (Figure 1). This is due to the different purpose for which they are made: the volunteer computing is mainly oriented to resource sharing for on-the-fly application remote relocation and execution, while the typical cloud hosts persistent services. In particular, the topmost layer agent (the `ACaaS`) is in charge to provide autonomicity for the execution of applications in the volunteer cloud.

[†] https://www.dropbox.com/s/yjsn64ikm65j4gw/AVoCloudy_140405_speSubmission.zip?dl=0



DOI: 10.1002/spe

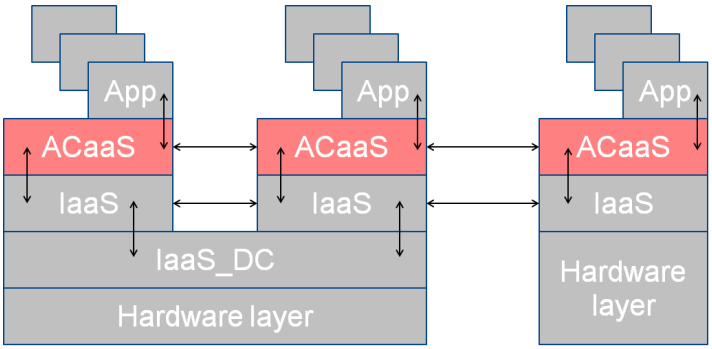


Figure 4. The AVOCLOUDY architecture.

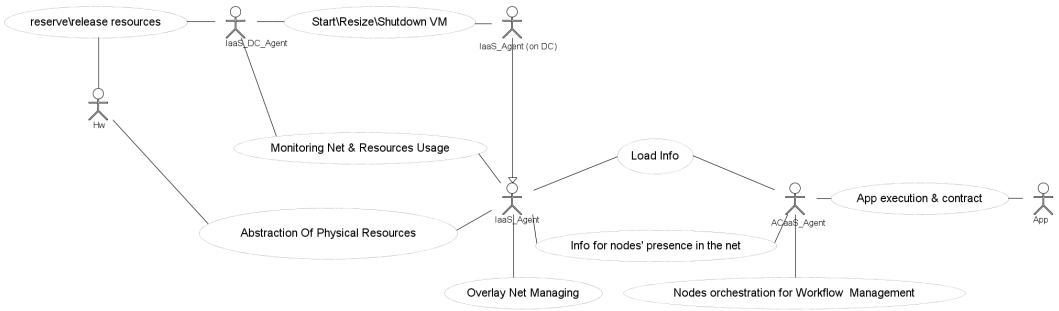


Figure 5. Use Case Diagram of the AVOCLOUDY simulator.

**Agent interactions.** The UML *Use Case diagram* in Figure 5 describes how agents interact in the system, providing functions and services. Physical resources (`Hw`) are managed by the `IaaS_DC_Agent` or, in its absence, directly by the `IaaS_Agent`. If the `IaaS_DC_Agent` is present, it is in charge to interact with its `IaaS_Agent`*s* to monitor and act on them for VMs resizing, shutdown and start. Each `IaaS_Agent`, cooperating with the other agents of the same layer, is in charge of managing the P2P overlay network. Load info and nodes knowledge present in the `IaaS_Agent` are exchanged with the `ACaaS_Agent` to contact the other agents of the same layer. Once the `App` is generated it contracts its execution requirements with the responsible `ACaaS_Agent`. The `ACaaS_Agent`*s* organize themselves in ensembles to orchestrate the application execution. AVOCLOUDY users define how agents interact to fulfill the goal of the designed scenario.

**Network.** Figure 6 illustrates the architecture of the volunteer cloud, representing with squares and circles respectively data centers and personal devices, and where, for sake of simplicity, only two nodes are represented with their internal layer architecture. The nodes are arranged in different *cloud sites*. Each site can be composed by only data centers (the left site in the figure), only personal devices (the right site) or both type together (the bottom center site). Regardless from the nodes participating to the site the connection is realized with an arbitrary P2P overlay network. With AVOCLOUDY, it is possible to define scenarios with one or more types of volunteer nodes, by properly editing the simulator configuration file.

Each `IaaS_Agent` resides on exactly one site. The general organization is a semi-structured P2P overlay network, where each *site* is organized in an unstructured overlay network and one `IaaS_Agent` for each site is the *supernode*, in charge of "mediating" the communication between sites. Every node entering/leaving the network, being it fixed or mobile, signals its action to the supernode, as usually happens in the protocols used by layered P2P networks. Despite this simple signaling protocol, it is still possible that a node abruptly leaves the network (e.g., due to a connection failure). To solve this issue, nodes should periodically ping the peers they know, in order to check



DOI: 10.1002/spe

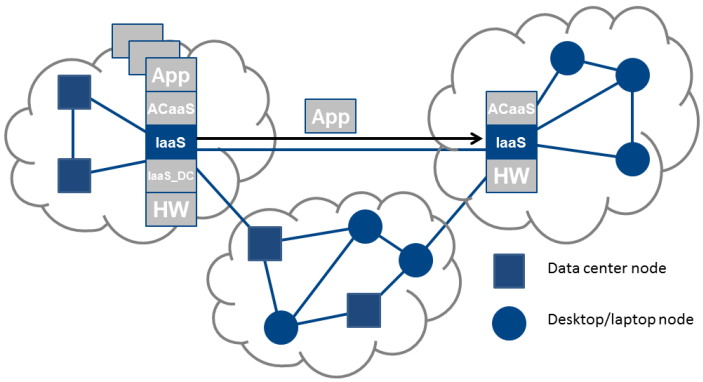


Figure 6. Network architecture.

for their aliveness and, if it is the case, to replace them with new contacts, provided by a bootstrap server or by other known nodes. Several strategies can be used to select supernodes of such sites: from classical leader election strategies [36–39] to more specific ones. For example, it is possible to choose the data center and most stable nodes, with the rationale that these nodes should experience better uptime and resources. Since the mobile devices are either handhelds, smartphones or tablets, with limited power and computing capabilities, they will act only as tasks producers and will never be considered as supernodes. AVOCLOUDY users can easily implement more sophisticated leader selection strategies [40–43], as described in Section 4.

**Infrastructure layers.** Data center machines have an `IaaS_DC_Agent` which is in charge of monitoring its `IaaS_Agents` and, according to its policies and available resources, decides when it is needed to start (instantiate a new one), resize (scaling-up or down) or shutdown the `IaaS_Agents` that are running on top of it. The knowledge of dynamic information (e.g., CPU load, memory consumption) and static information (e.g., processor speed and number of cores) on each `IaaS_Agent` is used by the `IaaS_DC_Agent`, where present, to support its decision process. According to the load monitoring information, it is possible to trigger a request to expand the resources available on a node, or to start another `IaaS_Agent` if a distributed `App` can take advantage from the presence of another node instead that of a single powerful one. When a node is underloaded for a certain amount of time, it can also perform the reverse action (i.e., reducing the available resources or shutdown a node), to spare energy.

Differently from the data centers, volunteer devices do not have the chance to run multiple VM instances if not by consuming a lot of resources. Taking into account that these machines should be available also for local use, the amount of shared resources is defined a priori and no resize policy is considered.

Regardless of the underlying layer (physical or IaaS_DC), the `IaaS_Agent` has in charge the communication with the other nodes to realize the P2P overlay network. Moreover, in addition to the typical duty of an IaaS cloud layer, the `IaaS_Agent` is in charge of assigning and sharing the resources available in its VM (i.e., CPU, memory, bandwidth, etc.). The P2P overlay ensures that each `IaaS_Agent` is able to find agents of the same layer and establish connections with them relying on the underlying network, such as the local area network or the Internet. Moreover this layer provides the support for the exchange of the data needed by upper layers (e.g., exchange of data and applications).

Each volunteer data center node can be modeled as a $G/G/m/K$ queue, where $G$ means that task arrivals and service times have generic distributions, $m$ is the number of `IaaS_Agents` in each, and $K$ is the maximum number of tasks in the system (1 being executed, $K-1$ waiting in a First Come First Served, FCFS, queue) [44]. Personal devices are modeled as $G/G/1/K$ queues, since they cannot resize their resources nor run multiple `IaaS_Agent` instances.



DOI: 10.1002/spe

*Definition 1* (`IaaS_Agent` resources)
The resources available to the `IaaS_Agent` are described by a tuple $\langle \kappa, \phi, \nu \rangle$, where $\kappa \in \mathbb{N}^+$ is the number of processors, $\phi \in \mathbb{N}^+$ is their clock frequency, and $\nu \in \mathbb{N}^+$ is the amount of memory.

**Autonomic layer.** The `ACaaS_Agent` is the topmost layer, charged to autonomously coordinate with the other nodes of the same layer, forming aggregations (called *ensembles*) in order to orchestrate the application executions and manage their workflows in a robust and optimized way, according to the node's policies and the application's requirements (e.g., the associated QoS constraints). For each application an isolated execution environment is assigned (like a sandbox or a VM), to ensure a basic level of security for the user data and the application. Such a layer should be defined by the AVOCLOUDY user to test the designed application distribution protocol. A few example protocols we implemented in AVOCLOUDY are described in Section 5.

**Application layer.** The volunteer cloud provides distributed application execution as its main functionality. Applications may range from batch tasks to interactive applications to persistent services, which may have different requirements in terms of resources (e.g., CPU and memory needed) and QoS (e.g., deadline). The application (`App`) can be generated by any node participating to the network. The node that generates the `App` makes use of its own responsible `ACaaS_Agent` to autonomously find the most promising nodes to distribute and execute the application, while respecting its resources and QoS requirements. When an `ACaaS_Agent` receives a task execution *request*, it autonomously responds, according to its policies, specifications and load, with a task execution *bid* containing the requirements (both *functional* and *non-functional*) that it is able to satisfy. Several mechanisms have been investigated in [45–47], but in general the AVOCLOUDY user is in charge to define his/her own application dispatch mechanism.

The `App` can be defined as a single task, a set of independent tasks or arranged in a workflow, with an associate QoS specified by a SLA. The `App` definition is taken into account by the `ACaaS_Agent` responsible. The `App` typology changes the goal that must be fulfilled, e.g., when an application is a set of tasks usually the goal is to minimize, or respect a constraint, on the *makespan* (defined as the time difference among the task that completes first, and the one that completes last in the given set). A simple example of SLA specification can impose a certain deadline for the execution of each task.

Every task belonging to an `App` is defined by and carries a description with its duration, required amount of memory and maximum degree of parallelism (with the size of the parallelizable fraction) that it is able to exploit. The declared task execution duration is defined on a "reference" architecture, but its actual duration can be reduced according to the CPU architecture and frequency available on the executing node. Moreover, the task's parallelism and the available CPU cores affect the task's duration. It is also worth to consider that, when an `ACaaS_Agent` evaluates the task completion time, in general, it is not able to accurately predict the tasks duration.

*Definition 2* (task)
A *task* is a tuple $\langle \delta, \rho, f, \mu, \sigma \rangle$, where $\delta \in \mathbb{R}^+$ is its duration (expressed in cycles), $\rho \in \mathbb{N}^+$ is its degree of parallelism, $f \in [0, 1]$ is the parallelizable section, $\mu \in \mathbb{N}^+$ is its memory requirement and $\sigma \in \mathbb{R}^+$ is its size on secondary (auxiliary) storage.

From the above definition, it follows that $1 - f$ is the task section that needs to be executed sequentially and $f$ is the fraction that can exploit the maximum machine parallelism.

The execution time $e(T, M)$ required to complete a task $T = \langle \delta, \rho, f, \mu, \sigma \rangle$ on a machine $M = \langle \kappa, \phi, \nu \rangle$ is computed according to some recent extensions of the Amdhal's law [48–50].

*Definition 3* (task execution time)
A task $T$ with degree of parallelism $\rho$ and parallelizable section $f$, executed on a machine $M$ whose cores frequency $\phi$, has the following execution time:

$$e(T, M) = \frac{(1 - f) \cdot \delta}{\phi} + \frac{f \cdot \delta}{\phi \cdot \min(\rho, \kappa)} \tag{1}$$



DOI: 10.1002/spe

That is, for the parallelizable section $f$, we divide the duration on one single CPU ($\delta/\phi$) by the maximum degree of parallelism that can be exploited, which is bounded by both the amount of available CPUs ($\kappa$) and the parallelism degree of the task ($\rho$). Instead, the task's sequential section can only exploit the cores frequency.

A task $T = \langle \delta, \rho, f, \mu, \sigma \rangle$ can be executed on a machine $M = \langle \kappa, \phi, \nu \rangle$ only if the memory requirement constraint is satisfied:

$$\mu \leq \nu \tag{2}$$

We considered this constraint in our model, assuming that memory-swapping operations could seriously degrade performance. In Section 4, it is described how to customize or change this behavior by sub-classing.

A machine $M$ should accept a task $T$ in its queue only if it can respect the task's deadline.

The `ACaaS_Agent` should take care of the `App`'s SLA until its execution completes, even in presence of unforeseen conditions or events that can compromise its execution, such as node disconnection, failure or completion time estimation error. In general, nodes can also be malicious, i.e., they accept tasks even if they know that are not able to satisfy the `App`'s requirements. In these events, the autonomic adaptive logic should acts migrating or restarting the `App` on another `ACaaS_Agent` that is able to execute it satisfying its requirements.

The `ACaaS_Agent` generates execution requests, based on functional and non-functional requirements of the `App`.

*Definition 4* (task execution request)
A *task execution request* is a tuple $\langle \delta, \rho, f, \mu, \sigma, \tau_a, \tau_d \rangle$, where $\langle \delta, \rho, f, \mu, \sigma \rangle$ is a task, $\tau_a \in \mathbb{R}^+$ is the task arrival date, and $\tau_d \in \mathbb{R}^+$ is its termination deadline.

Task execution requests can be generated by any node participating to the network. As any other event in AVOCLOUDY, these requests can be generated with an arbitrary time distribution (we refer the reader to Section 4 for more details).

**Requests handling.** Application execution requests can be submitted by any node participating to the network. Figure 7 is a generalization of the centralized control server approach that dispatches the task load, considered in the concrete implementations mentioned in Section 2. To each `App` is assigned an `ACaaS_Agent` responsible to *process* it, but not necessarily execute it (Figure 7, step 1). The responsible agent considers its resources, knowledge (Figure 7, step 2) and, if necessary, contacts other nodes that can satisfy the application requirements (Figure 7, step 3). Among the nodes that can satisfy the requests, the choice falls on the most promising one, according to the specified optimization `Policy`. Once the `ACaaS_Agent`*s* able to satisfy the request have been found, the underlying `IaaS_Agent` sends the `App` to the *executor* nodes (Figure 7, step 4), waits until they complete the execution and receives their response. The workflow depicted in Figure 7 describes the main steps to manage execution requests. The definition of how each step is accomplished must be defined by the AVOCLOUDY user.

In our completely distributed architecture there is not a single dispatcher (see Figure 7). Every node that generates a task request (task producer) acts according the defined task distribution algorithm (e.g., ACO, reputation, and others).

It is worth noting that all task transfers incur in a transmission cost which depends on where destination nodes are located.

*Definition 5* (Link delay $\xi$)
Nodes $a$ and $b$, whose geographical position are respectively $(x_a, y_a)$ and $(x_b, y_b)$, are affected by a communication delay $\xi$ [51].

$$link\_length = \sqrt{(x_a - x_b)^2 + (y_a - y_b)^2} \tag{3}$$

$$\xi = \frac{link\_length}{propagation\ speed\ in\ the\ medium} \tag{4}$$

The transmission cost is evaluated according to the minimum data rate among the two end-points.



DOI: 10.1002/spe

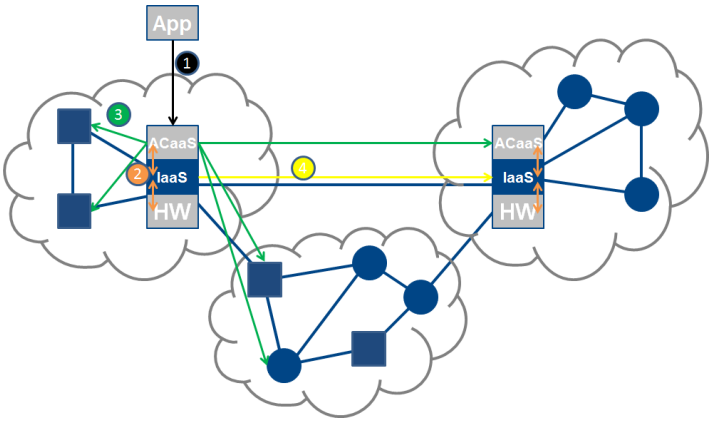


Figure 7. Execution request handling (without central dispatcher).

*Definition 6* (Transmission cost $\psi$)
The transmission of a task $T$ (with size $\sigma$) between nodes $a$ and $b$, that have respectively data rate $r_a$ and $r_b$, is affected by a data overhead $\psi$

$$\begin{aligned} r &= \min(r_a, r_b) \\ AND\ if\ lan_a \neq lan_b\ &: r = \min(r_{internet}, r) \end{aligned} \tag{5}$$

$$\psi = \frac{\sigma}{r} \tag{6}$$

The overall delay that affects task $T$, when it is transmitted among two nodes, is thus $\xi + \psi$.

Link length and connection type are required only during the simulation of task transmission and this is true even for the mobile device. It is assumed that during a task's transmission, the mobile device does not significantly change its position or connection type (e.g., switching from cellular to WiFi network - although DEUS has been used to simulate also this scenario [52]). All other node displacements do not affect the communication or the volunteer network, which is a logical network.

## 4. AVOCLOUDY IMPLEMENTATION

AVOCLOUDY is based on DEUS, enriched with MultiVeStA capabilities. For each layer of the architecture described in Section 3, there is a corresponding Java package.

In addition to the Java classes, as for all DEUS projects, there is an associated XML configuration file. It is parsed before the simulation and it is used to parametrize the simulation itself. For many of the classes described in this section a concrete implementation is provided. Each class can be used "*as is*", just parameterizing its behavior via the XML file.

**Package organization.** The root package containing the basic classes for the volunteer nodes, is `it.imtlucca.avocloudy`. The root class for the node instantiation is `AbstractCloudyNode`. It has been already instantiated in two different versions to discern among data center nodes (`DataCenterNode`) and volunteers (`VolunteerNode`). Furthermore, the same package contains the corresponding birth events.

The sub-packages are:

- `physical` (see Section 4.1)
- `infrastructure` (see Section 4.2)
- `autonomic` (see Section 4.3)
- `infrastructure.dc` (see Section 4.4)
- `application` (see Section 4.5)



DOI: 10.1002/spe

- `util` (see Section 4.6)
- `log` (see Section 4.6)

The following sub-sections describe the main features of each package.

The flexibility of the AVOCLOUDY architecture allows users to implement the desired behaviors by means of sub-classing, very simply.

Given that all the layers are constituted by autonomous entities/agents that cooperate (with other nodes in the same layer and with the ones in upper and lower layer on the same node), the main class for each layer is represented by an agent. This approach (i.e., layers' independence) allows for modularity, which eases the implementation of high-level protocols (e.g., infrastructure, autonomic behavior, application logic), while several low-level protocols are already provided by AVOCLOUDY.

### *4.1. The `physical` Layer Package*

The `Feature` class represents the physical characteristics of the underlying node resources, i.e., core number, clock frequency, amount of memory, machine identifier, type of device (data center, volunteer or mobile), network connection and geo-location.

Since AVOCLOUDY focuses on application level performance evaluation, considering reasonable task durations and arrival times, a deep modeling of the network layer is not strictly required, and may constitute a bottleneck for the simulation process. Anyway, despite DEUS is not provided with a communication package, it does enable the simulation of delays and packet losses, e.g., leveraging on statistical models obtained with specific network simulation tools (such as ns-3, as experimented by Amoretti *et al.* [32]).

In AVOCLOUDY, every node is assumed to be linked with each other through a local area or an Internet connection (customizable through the simulation configuration file). Each type of connection is characterized by a data rate, which in turn affects the transmission time of application messages from task producers to task consumers. We have modeled the most common types of connections (both wired and wireless), but AVOCLOUDY users can specify and add their own ones (e.g., Fibre Channel, Wi-Max), according to their specific needs.

The network connection is specified by `AbstractConnection`, which provides methods to obtain lan_id, min/max range (expressed in meters) and effective data rate (expressed in Mbit/s). Currently, there are a range of implementations already provided for both wired and wireless connections, i.e., LAN, Giga-LAN, 10-Giga-LAN, WiFi, WiFi-g, WiFi-n. For wired connections, the communication overhead is assumed to be 25% [53–56]; for wireless connections, it is assumed to be 50% [57, 58]. The Internet data rate has been obtained from the measurement performed by Lee *et al.* [59]. Other network connection specifications can be obtained implementing the `AbstractConnection` class.

The node's geographical position is expressed as $x$ and $y$ coordinates and it is used by the `evaluateOverheadTime()` method. If source and destination lan_id differ, or they are out of the other node's range, the model evaluates the time needed to transmit the application among the two nodes in the network also considering their geographical position. The `evaluateOverheadTime()` method calls `delayToOtherPoint()` and `taskTransmissionOvehead()`, and sums their returned values. The former function follows the communication cost computed by the simple yet realistic network model described by Saino *et al.* [51], which assigns link delays proportionally to the estimated link length, evaluated as the euclidean distance between the geographical position of the two nodes. We assume a geographical position expressed in meters, if needed, after an approximate conversion from degrees.

We are currently working to improve the communication model allowing to import, in the initial stage of the simulation, a description of the underlying network. Loading a network model (which can be generated by means of other tools [51]) can make the node communication model more realistic.



DOI: 10.1002/spe

### 4.2. The `infrastructure` Layer Package

The `IaaS_Agent` is the "logical node" and it is in charge to interact with the other nodes of the same layer, to manage the P2P overlay network and to execute the applications taken in charge by the above layer agent, using `Hw` or `IaaS_DC_Agent` resources.

In this layer, the `Behavior` class provides the methods to manage the overlay network: `connect()`, `join()`, `signalPresenceToOtherNode()`, `disconnect()`, `death()` and `getNodesInZone()`. Besides the basic signaling messages mentioned above, and used in any P2P overlay network, AVOCLOUDY users can specify how the nodes interact and how their knowledge of the network is kept updated, e.g., adopting heartbeat messages. For what concerns hardware resource management, it provides information about the time needed to execute a given application (`evaluateTaskDuration()`) and to estimate the network overhead that must be taken into account during the application transmission (`estimateNetOverhead()`). According to the estimated network overhead, execution time, memory and QoS requirements of the application, it is possible to `estimateTaskAcceptance()`.

The current implementation takes the time estimation obtained when the executing machine follows the *Rough Set* technique proposed by Selvi *et al.* [60]. Assuming that, since the boot, the node has a database to start the Rough Set techniques and thus it can estimate the exact runtime execution with a uniform error committed (e.g., of $\pm 13\%$). From the XML configuration file it is possible to set a different percentage error. Different estimation techniques can be obtained by sub-classing the `estimateAppExit()` function.

`Policy` refers to the `VmCriteria` and `NetworkCriteria` mentioned in the Iaas_DC layer e.g., to specify the node connection and the cloud site assignment.

`Knowledge` maintains the information about neighbors, supernodes and the presence of other cloud sites.

The IaaS `Features` are specified as VM resources (i.e., core number and clock frequency, main and local memory, network bandwidth) and overlay network (i.e., online status, cloud site identifier, node identifier and if it is acting as a supernode).

Moreover, all the events that allow the basic overlay network operations (i.e., join, reconnection, death, disconnection) are implemented.

As for the `Hw` layer, in the future we will add an option to load a pre-generated overlay network. In addition we are evaluating the implementation of an option allowing users to express the messages exchanged in the network. This feature will be optional, since, despite it does improve the model's realism, at the same time it inevitably brings to a great simulation performance decay.

### 4.3. The `autonomic` Layer Package

An `ACaaS_Agent` interacts with the other nodes of the same layer to manage the application workflow (`askFinishEstimation()`) and it is charged by the `App` to manage the application (`setNewApp()`).

When a local or a remote `ACaaS_Agent` accepts to execute the application, the `SendTaskToExec` event is added in the DEUS event queue, to submit the application in the corresponding node's queue. The time at which the event will be triggered is defined according to the time needed to transmit the application on the network. When the application execution is completed, the `ExecutionEndEvent` is triggered, the `notifyAppExecutionEnded()` is called and the performance statistics (e.g., hit, miss, sojourn time) are updated accordingly.

Also this package has a `Behavior` class, which is in charge to manage the application once a node is made responsible (`manageApplication()`). The main logic according to which an agent looks for other volunteers is defined by the `searchExecutingNodes()`. In the recent past, our research has focused on *self-** algorithms for distributed tasks execution in the volunteer cloud [45–47]. Thus, it has been sufficient to re-implement that function to specify different node behaviors. In some scenarios, it may be necessary to specify the order according to which other nodes must be asked for the execution of the application, through the `askExecToNodeList()` method, which takes as argument the list of all the nodes that must be contacted.



DOI: 10.1002/spe

The interaction with the other node agents that can execute the application takes place through the methods `reqFinishEstimation()`, `appExecReq()` and `sendTaskForExec()`.

The `Policy` class defines the `AppCriteria` according to which a node should look for collaborations with other nodes and should accept remote requests. E.g., the maximum number of attempts that must be performed before marking the task as missed (`maxNumOfAttempt()`); the miss rate tolerance, over which no more external requests should be considered (`missRateTolerance()`); and whether the cooperation of other nodes should be considered (`askToVolunteer()`).

The `Knowledge` class is to collect historical information about application executions and interactions with other nodes.
`AppTypeStatistics` stores, by application type, the information on hit, miss, received/unmet requests and queue statistics (waiting and sojourn times). `WorldHistory` is in charge to record a summary of the interactions with other nodes, during an application's execution: ack, nack and total number of requests.

The `Feature` class maintains the information on the applications on the node's queue to allow the node to compute the actual load and respond to the external requests accordingly.

The AVOCLOUDY user is encouraged to design and evaluate its own application distribution protocols. In our previous works, we have designed some protocols that have been briefly described in Section 5 and that can be freely downloaded, studied and modified.

### 4.4. The `infrastructure.dc` Layer Package

Data center nodes differ from volunteer nodes, as they are, in general, more powerful in terms of CPU, memory and storage, which makes them able to host much more VMs. Thus, functionalities to monitor and manage such VMs are provided in this layer.

`IaaS_DC_Agent` is the main class that refers to its components (i.e., `Knowledge`, `Behavior` and `Policy`), to the underlying `Hw` layer and to all the `IaaS_Agents` instantiated on the same node.

This layer's `Behavior` class manages all the above `IaaS_Agents` using the available hardware resources, via the methods: `manageStartup()`, `manageShutdown()` and `manageResize()`. In the same package are also provided the *events* that check if it is needed to call the above mentioned functions. These abstract class events can be sub-classed to customize the check functions e.g., to implement more sophisticated VMs management strategies [61].

The `Knowledge` class collects load history information using `LoadRecords`. If needed (e.g., to personalize the behavior of data center nodes), it records information about: execution time, RAM utilization, number of cores and clock frequency.

The `Policy` class is used to manage the available resources in response to the environment. It refers to the `NetworkCriteria` and `VmCriteria` classes, which specify the policies used to manage the network and the VM resources assigned to the `IaaS_Agent`. In particular, `VmCriteria` defines the quote of resources that must be assigned to each VM, if its resources can be changed during the operation and optionally a "stability factor". The latter can be a measure to define possible causes that bring a node down. `VmCriteria` can be instantiated with resource usage values, to trigger the startup, shutdown or resize of one of the managed `IaaS_Agent`'s resources. A flag that allows the VM to be re-sizable is provided. Moreover, it is possible to specify the percentage of hardware resources that should be assigned or increased during the startup or resize of one of the above `IaaS_Agent`'s resources.

### 4.5. The `application` Layer Package

The `application` layer defines the application characteristics, requirements and execution details.

An application (`App`) is generated when the `AppBirthEvent` is triggered, and as first action it assigns itself to the node that has generated the application (`setReferringAgent()`).

The `Behavior` class is used to ask the execution of the whole application (using `askWorkflowExecution()`) and, if needed, negotiates (`contract()`) with the `ACaaS_Agent` before assigning the execution.



DOI: 10.1002/spe

A `Policy` specifies the QoS and non functional requirements. The non-functional requirements can be used, for example, to restrict the geographical zone in which an `App` can be executed.

The `Knowledge` class provides information about the nodes that are currently executing the application and some statistics about the application's performance (e.g., a counter for the remote requests that have been performed or the nodes on which it is being executed).

The `Feature` class allows to describe the application's characteristics (generation, start and end times, fraction and degree of parallelism and type). Moreover, it maintains the status [62] for each of the tasks that compose the application.

Figure 8 shows the UML Sequence Diagram for a task's execution. The application is generated (`setReferringAgent()`) by the `ACaaS_Agent` X. Eventually, agent X checks if it can locally satisfy its execution (`estimateAcceptance()` on *step 3*) and then, according to its application distribution algorithm, cooperates with other `ACaaS_Agents` (e.g., Y and Z) to satisfy the request. It is worth to note that, within the application management (`manageApplication()`), the responsible agent (X) checks its own and its peers' (Y and Z) availability (`reqFinishEstimation()`), considering the resources that are available on the `IaaS_Agent` node (`estimateAcceptance()`). Once one or more agents that can satisfy the execution request have been found, it is possible to contract the non-functional requirements (`contract()`). Then, the `App` is sent to the chosen agent (`appExecReq()`) and finally its execution is performed (`execution()`).

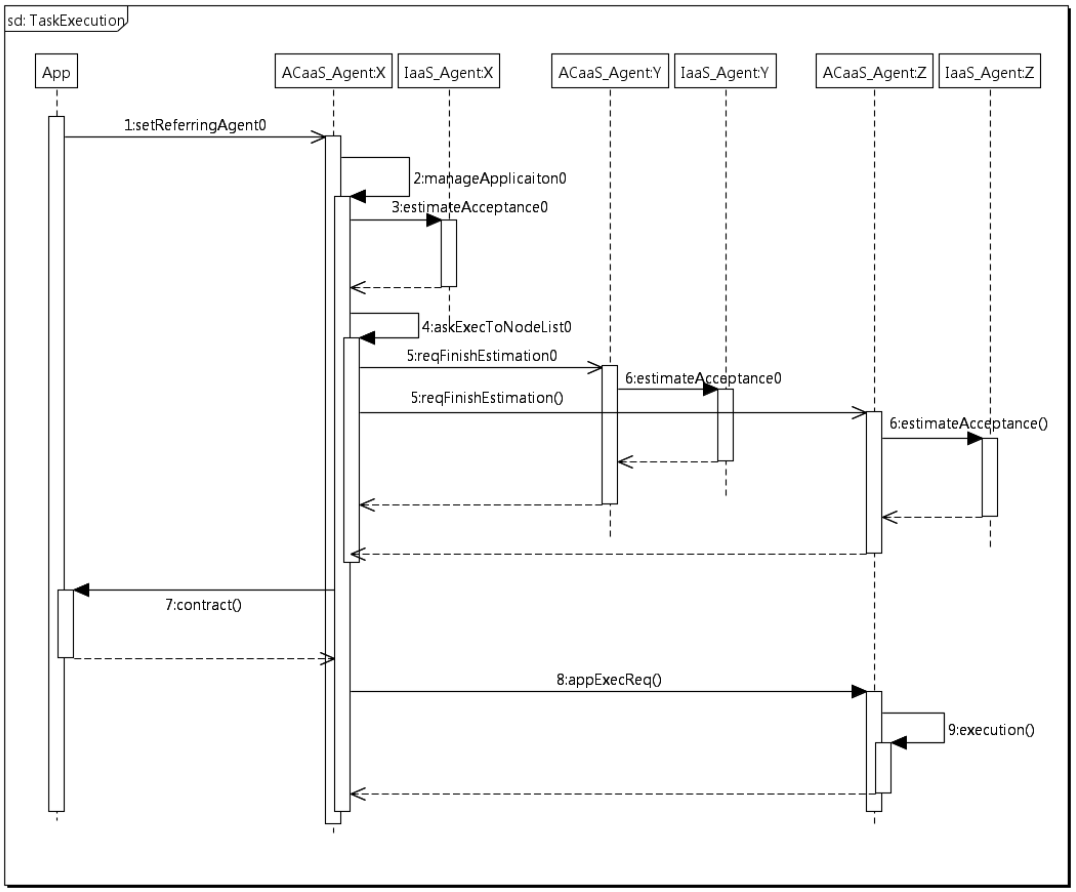


Figure 8. Sequence Diagram of the AVOCLOUDY simulator for Task Execution.



DOI: 10.1002/spe

### 4.6. The `util` and `log` Packages

These packages provide some basic mathematical functions, through the `IFunction` interface, and logging performance indicators (e.g., task hit rate, performed requests and refused rate, queue statistics). Basic performance statistics can be obtained also using MultiVeStA.

Every event in the simulator (e.g., task arrival, node departure) can be scheduled according to an arbitrary distribution, provided to the simulator by means of a configuration file. DEUS includes its own statistical distribution library, which we further extended while implementing AVOCLOUDY (currently available distributions are: exponential, lognormal, uniform, Weibull, periodic, Poisson, rectangular and Pareto).

Finally network logging capabilities for Pajek [63] and Gephi [64] (in GraphML file format) are provided, respectively in the `LogCloudyForPajek` and `LogCloudyForGraphml` classes.

It is worth to note that a single problem can be tackled from different layers of the architecture. E.g., it is possible to manage the volunteer node that goes down while it is executing an application in different ways: (i) merely restarting the application on another volunteer, if the execution is not completed within a reasonable time-frame, or using a proactive approach, by running application replicas in parallel on different volunteers (`autonomic layer`); (ii) or using heartbeat messages to monitor the online presence and the progress of the execution in a remote node (`infrastructure` layer). In some of our previous works ( [46, 47]) we have used AVOCLOUDY to model self-* approaches capable to deal with nodes that leave the volunteer network.

## 5. CASE STUDIES

Our simulator has been validated over several case studies [45–47] developed within the ASCENS project [24]. This section describes some exemplifying aspects of their implementation, in order to provide guidelines for using AVOCLOUDY.

### 5.1. Random Task Distribution

In designing a task distribution approach, the most simple algorithm that can be evaluated is the *Random* distribution. This subsection describes the random algorithm as implemented in [46]. For each `App`, the responsible `ACaaS_Agent` generates a random list with other `ACaaS_Agents` in the network, to whom it can submit an execution request.

To include this simple behavior in AVOCLOUDY we have only re-implemented the `searchExecutingNode()` function in the `AcaasNodeBehavior` class, to shuffle the list of known `ACaaS_Agents` before calling the built-in function `askExecutionToNodeList()`.

### 5.2. Cooperative Task Distribution

The impact of a cooperative behavior in the distributed execution of `Apps` for volunteer nodes has been evaluated in [45]. This scenario assumes that volunteer nodes want to contribute to the network with their resources but, at the same time, want to ensure that a certain amount of resources are available for local use without using pre-reservation techniques. The exploited approach considers a *partial volunteer* to ensure that the `ACaaS_Agent` avoids overloading situations. An `ACaaS_Agent` receiving an execution request accepts only if the miss rate that it perceives locally is under a specified threshold:

$$currentMissRate \leq MissRateTolerance \tag{7}$$

The implementation of this approach is quite straightforward using AVOCLOUDY. The `AcaasNodeFeature` computes the `actualMissRate()`, which is used by the `AcaasPolicy` upon the `ACaaS_Agent` receives an `appExecReq()`.



DOI: 10.1002/spe

### 5.3. Reputation-Based Task Distribution

In scenarios where node resources are unknown or when nodes do not want to disclose their status, a valuable approach in the choice of the `ACaaS_Agents` to target `appExecReq()` is represented by a reputation-based system [47]. Such a system provides a ranked list of `ACaaS_Agents`, according to the feedback received about the interactions among the `ACaaS_Agents`. E.g., feedback can be provided once a `ACaaS_Agent` accepts, rejects, completes or fails to execute a remote `App`.

The key points modeling such scenario in AVOCLOUDY are: registration with the reputation system and feedback request/report. The reputation system itself can be modeled as a node created in the early stage of the simulation. Node registration is performed by the `AbstractVolunteerNodeBirthEvent`. In particular, the abstract method `specifyAcaasNode()` has been used to contact the reputation system registering the presence of the new node in the network.

When an `ACaaS_Agent` is made responsible for an `App`, its first action is to contact the reputation system to obtain a ranked list of nodes, which is then used with `askExecutionToNodeList()`.

### 5.4. A Computational Field for Task Distribution

Given the high complexity and heterogeneity of volunteer cloud computing, a centralized solution to distribute the workload seems to be far from optimal. This is mainly due to impossibility to obtain a global knowledge of node resources and especially of node loads. In a recent work [46], we proposed a collaborative *Colored Computational Field* to allow an agent-based tasks distribution inspired by the Ant Colony Optimization and gradient-based approaches.

Aside from the algorithmic details of the proposed solution, that are out of the scope of the presented work, our approach has requested a more articulated implementation, with respect to what has been done for the other algorithms presented in this section. In particular, it has been necessary to implement new events for the periodic execution of agents that are in charge of building the above-mentioned computational field. These mobile agents are created when the `ACaaS_Agent` is generated, i.e., in `specifyAcaasNode()`. Each node stores a portion of the distributed computational field, thus the `AcaasNodeKnowledge` has been extended to this purpose, taking in memory the pheromone values. The policies according to which the `ACaaS_Agent` sends the mobile agent to explore the network are defined in the `AcaasNodePolicy` including information such as: time-to-live, timeout, component weights and considered resources. Finally, to be able to exploit the information stored in the `AcaasNodeKnowledge`, also the `searchExecutionNode()` has been modified.

Our ACO technique allows to distribute the workload in a way that every task execution request is completed as soon as possible, while minimizing the amount of resources that are reserved but not used, on a given node. The overall behavior brings to an increased probability to satisfy task requests, while respecting their SLAs (e.g., deadlines).

### 5.5. Case Studies Final Remarks

It is worth to note that all the above-mentioned case studies would not be easily implemented in the available cloud simulators (as described in Section 8). Instead, as shown in this section, with AVOCLOUDY the implementation effort has been limited and targeted only to the architecture layer involved in the evaluation of the proposed algorithms (the `ACaaS` layer in our case studies). E.g., the validation of a P2P overlay protocol would affect only the `IaaS` layer, while, for all other layers the basic implementations could be used without any further effort.

Finally the performed integration with MultiVeStA allows AVOCLOUDY users to decouple the definition of the system properties of interest from the model specification.

Example of performance parameters that have been evaluated during these works, and whose definition is already provided with AVOCLOUDY, are: successfully executed tasks, refused remote execution requests, total amount of addressed execution requests, mean task waiting and sojourn times, mean number of tasks executed on each node. For each of these parameters, the size of the Confidence Interval and the variance are also provided.



DOI: 10.1002/spe

Table I. Node attributes

| type of node | CPU frequency (GHz) | CPU (cores) | RAM (GBs) | Number of nodes |
|---|---|---|---|---|
| desktops and laptops | $1-2$ | $1-6$ | $0.1-4$ | $1,000-10,000$ |
| data centers | $1-3.73$ | $1-16$ | $3.75-60$ | 7 (in different sites) |

Table II. Task attributes

| size | duration (hours) | CPU (cores) | RAM (GBs) | Deadline offset (percentage) | Poisson mean arrival (ms) |
|---|---|---|---|---|---|
| small | $0-0.4$ | 1 | $0-0.5$ | 0.2 | 200 |
| large | $1-12$ | $1-4$ | $1-4$ | 0.4 | 600 |

## 6. EXPERIMENT AND EVALUATION

This section describes an experiment for evaluating the effectiveness and performance of AVOCLOUDY in simulating a volunteer cloud environment. The modeled scenario is quite realistic, considering the use of the Google workload and the characterization of data center and volunteer nodes.

A reliable AVOCLOUDY validation to assess its level of accuracy requires to develop accurate monitor techniques distributed over all the volunteers (that should be at least in the order of 1000 nodes for a realistic volunteer environment) and an evaluation of distribution and execution approaches over a wide range of task types. Instead, the validation using many `ACaaS_Agents` on the same machine would have been a less cumbersome activity, but at the same time would have significantly affected its accuracy not being able to test the network layer.

### *6.1. Simulated Scenario*

We have designed a scenario where some research institutions participate to the volunteer cloud with their data centers, while other universities and volunteers participate with their desktops (e.g., the labs machines unused during night) and laptops. The laptops are connected to the Internet through a WiFi-g, while the data centers are wired connected via a 10 Gbit Ethernet. To use realistic characteristics for the data centers, we have considered the specification of the Google Cloud Platform [65] (assuming the CPU frequency according to the currently available CPUs for the server). For desktops and laptops, we have assumed a reasonable amount of shared resources, considering common commercial devices. Node attributes are summarized in Table I, where values are uniformly distributed within the proposed intervals.

Concerning the workload characterization, we have considered the Google Cloud Backend [66] described in [67]. There, tasks are characterized by duration, CPU and memory requirements; basically, it is possible to discern two task categories: *small* and *large*. Examples of long-running tasks are compute intensive ones like scientific simulations. Short-running tasks are instead highly parallel operations, such as index lookups, searches and Map-Reduce. Usually, *small* tasks have a more stringent deadline, not to loose the advantage of the parallelism. It is worth to mention that the volunteer cloud paradigm has been adopted in the BOINC prototype (BOINC-MR) to support Map-Reduce jobs [1]. Table II summarizes the task attributes according to the description in [67].

The duration of the simulated scenario is 7 hours (with a granularity of milliseconds), the same of the Google Cloud Backend traces dataset [66].

### *6.2. Configuration File Snippets*

As an example, Listing 1 shows an excerpt of how the simulator has been configured, via its XML file, to describe the volunteer desktop characteristics reported in Table I. For each parameter, it is possible to specify a range of values with which each volunteer is characterized.

Listing 2 reports another excerpt of the same XML configuration file, describing the attributes of the small tasks reported in Table II, where the task duration is expressed in tens of milliseconds and the deadline is expressed as offset percentage respect to the task duration.



DOI: 10.1002/spe

Listing 1: Volunteer desktops

```
<aut:node id="desktop" handler="it.imtlucca.cloudyscience.VolunteerNode">
  <aut:params>
    <aut:param name="typeOfDevice" value="volunteer" />
    <aut:param name="minPhysCore" value="1" />
    <aut:param name="maxPhysCore" value="6" />
    <aut:param name="minPhysCoreFreq" value="1" />
    <aut:param name="maxPhysCoreFreq" value="2" />
    <aut:param name="minPhysRAM" value="128" />
    <aut:param name="maxPhysRAM" value="4096" />
    <aut:param name="connection" value="WiFiG" />
```

Listing 2: Small tasks characteristics

```
<aut:node id="applicationSmall" handler="it.imtlucca.cloudyscience.applicationLayer.Application">
  <aut:params>
    <aut:param name="minDeadlineOffsetToDuration" value="0.2" />
    <aut:param name="maxDeadlineOffsetToDuration" value="0.2" />
    <aut:param name="taskMinDegreeOfParallelism" value="1" />
    <aut:param name="taskMaxDegreeOfParallelism" value="1" />
    <aut:param name="paralFractionMin" value="1" />
    <aut:param name="paralFractionMax" value="1" />
    <aut:param name="taskMinDuration" value="0" />
    <aut:param name="taskMaxDuration" value="144000" />
    <aut:param name="taskMinRAM" value="0" />
    <aut:param name="taskMaxRAM" value="512" />
  </aut:params>
</aut:node>
```

For sake of brevity, neither the whole XML configuration file, nor the random task distribution protocol written in Java (and briefly described in Section 5) are shown here.

### 6.3. *Simulation Results*

As an example of possible performance parameters that can be obtained with AVOCLOUDY we have considered the overall number of tasks successfully executed or running, and the queue waiting time perceived by the *small* tasks varying the number of volunteer nodes that are participating to the network. The following plots show the mean and the confidence interval values for each number of volunteers that has been considered.

Figure 9 (left) shows that, as many volunteers participate, the percentage of tasks executed within their deadline increase almost linearly. Modulo the assumptions and simplification considered in our simulation, even a simple random task distribution protocol using 10000 nodes can successfully execute about the 70% of the tasks workload.

The waiting time for the *small* tasks is shown in Figure 9 (right). It is possible to observe that, as the number of volunteers increases, even the small tasks, which are the most deadline-dependent, perceive a reduced waiting time in the execution queue.

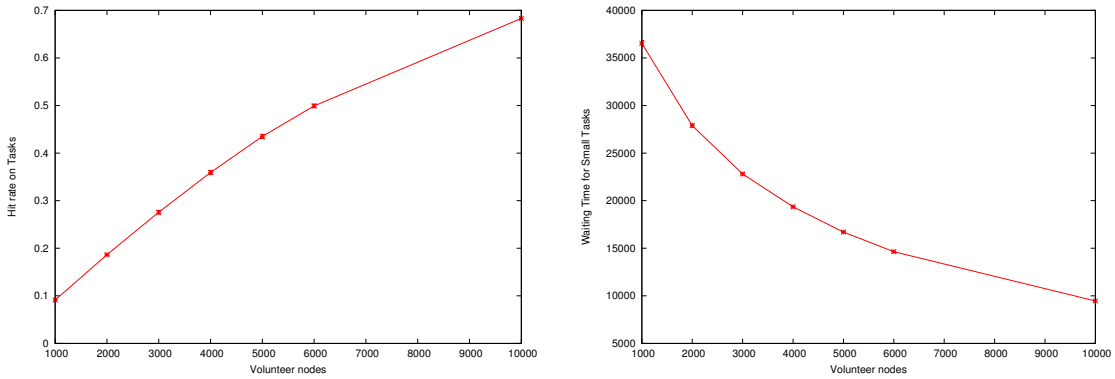


Figure 9. Overall Hit Rate (left), and Waiting Time for small tasks (right).



DOI: 10.1002/spe

AVOCLOUDY provided other performance parameters for the described scenario, which are not shown here, such as the number of execution requests sent and refused, waiting and sojourn time overall and discerned by task class.

### 6.4. *Performance Assessment of the Simulator*

All the experiments have been performed on a laptop equipped with a 2.0 GHz Quad Core CPU (with Hyper Threading) and 16 GB of RAM, running Windows 7 64-bit and Java 1.7.0_71-b14 64bit.

Recalling the toolchain depicted in Figure 3, it is worth to remark that MULTIVESTA allows to execute multiple simulation runs in parallel. Each run performed in parallel requires a MULTIVESTA server: a Java process that execute the simulation. To evaluate the performance scaling of AVOCLOUDY, we ran the simulated scenario different times, while varying the MULTIVESTA servers. Results are shown in Figure 10, where each server requires about 1.5 GB of RAM. Once the required confidence interval (evaluated with the Student's t-test) is specified by means of percentage and radius, MULTIVESTA automatically runs other simulations until the desired accuracy is reached. In our simulation we have requested a 95% confidence interval with a radius of 0.1.

As many servers are used in parallel to perform simulations, the overall time to obtain the results is considerably reduced. It is worth to note that the performance scaling is affected by the statistical nature of the analysis with a Monte Carlo simulation. Each simulation run is initialized with a different seed, thus affecting the sample variance of the considered simulation result (e.g., the hit rate), requiring a different number of simulation runs. Moreover it is worth to note that the actual number of events increase increasing the number of volunteers, e.g., due to signaling messages to manage the P2P overlay.

Finally, it should be taken into account that our CPU has four physical and eight logical cores. When the simulation time increases and more than four cores are used, the CPU is not able to use the Turbo Boost technology to speed up the cores frequency and thus we do not perceive a performance benefit by increasing the parallelism with our machine.

The reader interested in a deeper analysis of the performance scaling that can be achieved integrating MULTIVESTA with a discrete event simulator can refer to the recent work by Pianini *et al.* [68].

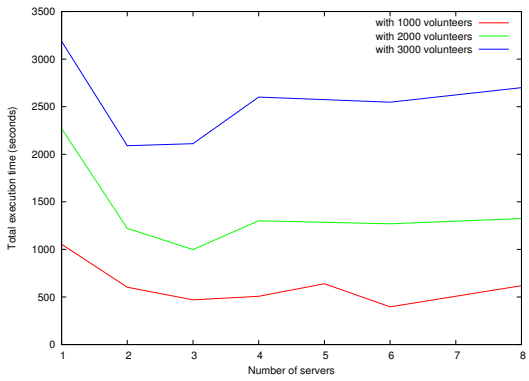


Figure 10. Performance scaling varying the number of running servers.

### 6.5. *Lesson Learned and Final Remarks*

Through the use of a simple experiment, this section has illustrated the use and the performance achievable modeling a volunteer cloud environment with AVOCLOUDY.

The section has begun showing the experiment that in turn has proved the usefulness of the volunteer cloud to run execution-oriented tasks, like the Google workload. The modeling of the volunteer cloud started with a straightforward parametrization of the AVOCLOUDY XML configuration file, to characterize tasks and nodes.



DOI: 10.1002/spe

Among the many cloud performance parameters that can be automatically obtained from AVOCLOUDY, we chose to show two plots that point out the scaling that can be obtained by increasing the number of participating nodes. AVOCLOUDY provided as output several Gnuplot data file, from which it was possible to draw the plots in Figure 9.

Despite the performance assessment has taken into account only few cores is still possible to appreciate the performance scaling that can be achieved executing simulation runs in parallel.

It is worth to remark that the AVOCLOUDY layered architecture and modularity (thanks to the MAPE-K model) helped us to perform step-by-step correctness evaluation of our code, separately validating each layer. This approach proved its effectiveness not only during the development of AVOCLOUDY itself, but also for every task distribution protocol we have defined and analyzed by means of AVOCLOUDY (summarized in Section 5). Some more work has to be done to improve the network layer, to allow for importing user-defined topologies.

Differently from a cloud simulator focused on the accurate modeling of the characteristics of a real physical machine, a volunteer cloud simulator is focused on the the modeling of a large number of heterogeneous machines that are not always available (entering and leaving the network). The complexity is than shifted to the overlay network, to the autonomic layer functionalities and to the choice of the significant aspects in the volunteer environment to allow the simulator to scale up to manage thousands of nodes.

## 7. A WORLDWIDE CONFIDENCE VALIDATION

This section presents our experiments aimed at validating the confidence of our simulator. Our simulator is able to deal with large-scale scenarios with several thousands of volunteers and beyond, thus accommodating to the size of a real-world volunteer cloud platform. However, performing a validation over such large-scale scenarios is difficult and challenging due to the need of keeping track of different performance measures for huge number of entities. In fact, as we report above the world-wide platform we have used did not allow us to scale the scenarios to thousands of volunteer nodes. In our experiments, we therefore focused on mid-scale scenarios with several hundreds of volunteer nodes deployed in the worldwide PlanetLab platform, of which the University of Parma is member.

### *7.1. PlanetLab, Experience and Testbed*

PlanetLab [7] is a network of worldwide distributed machines, which are shared by academic institutions and industrial research labs. PlanetLab is mainly used to test new technologies for distributed storage and processing.

The network currently consists of 1335 machines, each one running a minimal Fedora Core 8 Linux installation. PlanetLab users get *ssh* (Secure Shell) access to the remote machines, where they receive a reserved isolated environment, that they can configure and adapt to their needs.

To validate the simulator, we have developed simple *bots* that replicate the behavior of the AVOCLOUDY agents, where tasks are randomly dispatched. The deployment effort in PlanetLab is alleviated by *CoDeploy* [69], a network of high-performance proxy servers used to collectively provide a fast and robust web content delivery service. In our experiments, CoDeploy allowed us to deploy and start the bots on all the PlanetLab nodes.

The PlanetLab web interface allows one to add the nodes that take part in the experiments. Unfortunately, the use of PlanetLab is not smooth as it seems and running our testbed has required a significant effort. Among the 1335 nodes participating in the network, at the time of our experiments, only 665 nodes were reported as online and available, by the PlanetLab web interface. Among them some nodes were not actually online or reachable with ssh, and some were misconfigured (for firewall rules, absence of user disk space and other wrong permissions). The third issue encountered deploying our testbed was the absence of a Java run-time framework in all the machines that, for the great part of the nodes, was not even available for installation, since the Fedora repositories were offline for end-of-life (EOL) distributions (Fedora 8 was marked for EOL on 2009). Recovering as



DOI: 10.1002/spe

Table III. Task attributes for validation

| Poisson mean arrival (sec) | duration (min) | Deadline offset (percentage) | size (MB) | RAM (GB) |
|---|---|---|---|---|
| 50 | $0 - [0.125 \div 8]$ | 0.2 | $0 - 10$ | $0 - 0.5$ |

many nodes as possible for our testbed has therefore required a significant time consuming effort, in order to manually reconfigure the Linux repository on each machine and to install Java 5. With the best of our effort we have been able to build a volunteer cloud network of 233 bots.

After an initial idle period (used to stabilize the network and to propagate the *start validation* command), the bots (each one running on a different node) generate tasks with characteristics reported in Table III. The maximum task duration has been changed during the experiments, from 8 minutes and then halving six times (up to 7.5 seconds).

Each bot emulates the task payload generating a stub text file, which is transferred from the generating volunteer node to the one that accepts the task execution. To manage multiple execution requests at the same time, bots have been developed as multithread applications, where a new thread is generated for each incoming task execution request. Tasks are processed with a single core fed from a FIFO queue. The task execution has been emulated through a thread sleep for a length equivalent to the task duration.

A final issue encountered during the validation was the apparently significant discrepancy of waiting and sojourn times measured in PlanetLab with the ones obtained from the simulator, while the other performance metrics where consistent enough. This was due to a misalignment of system times among the nodes. The problem has been resolved revising the bot code, in order to take times relative to the local starting time of the experiment instead of absolute values.

The scenario was executed for 1 hour. Given the length of each real experiment, we have repeated the execution of each scenario only for three times, and averaged the results, without requiring a specific confidence interval.

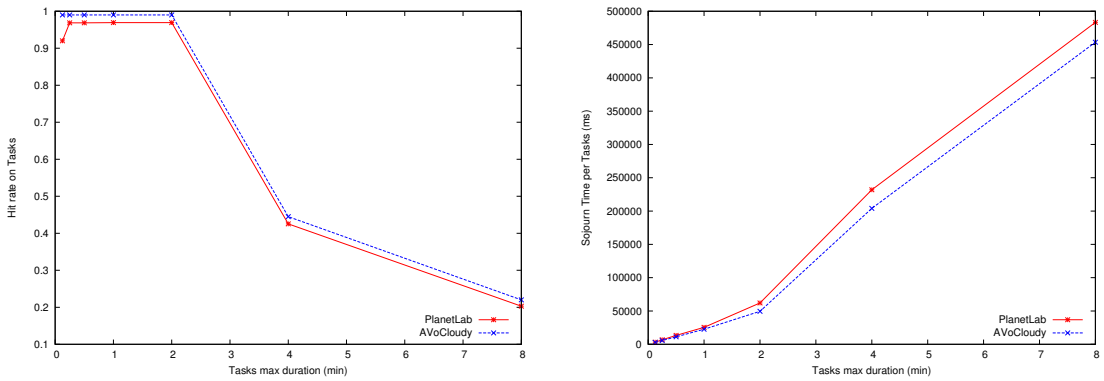


Figure 11. Hit Rate (left), and Sojourn Time (right) comparison between PlanetLab and AVOCLOUDY.

### 7.2. *Validation Results and Comments*

The results obtained from the volunteer network deployed in PlanetLab have been compared with the ones obtained with AVOCLOUDY. The following plots show the comparison of AVOCLOUDY and the real testbed, while varying the maximum task duration.

In Figure 11, is possible to observe that, when the scenario considers very short tasks, the accuracy of the results provided by the simulator decreases, in particular for what concerns the sojourn times (and similarly for the tasks waiting times). This behavior is expected, since, working with a coarser time granularity, the AVOCLOUDY choice to ignore the time required to transmit each task execution request becomes less significant. For reasonably long tasks, AVOCLOUDY allows us to achieve a good accuracy. Moreover, it should be considered that, having expressed the task deadline as percentage, shorter tasks implies less network delay tolerance.



DOI: 10.1002/spe

In particular, it is worth noting that, while the real testbed has required to re-execute the experiment spending an overall time of 21 hours, AVOCLOUDY has required less than one minute to simulate the same scenario, and obviously this advantage becomes more significant as the scenario of interest concerns a larger time window. A similar benefit is reflected in the number of volunteers, significantly limited in PlanetLab, while possibly unlimited in AVOCLOUDY. Obviously, the last two consideration are not strictly related to AVOCLOUDY, but more generally applicable to every simulator versus a real testbed approach.

**Threats to Validity.** The limitation to some basic scenarios does not allow us to draw fully definitive conclusions. A more accurate validation could be done, which would require the design of rather sophisticated bots able to generate and execute real applications, even exploiting all the cores available on each machine. Furthermore, different types of application should be evaluated. Nonetheless, we believe that our validation experiments are promising and increase our confidence in the simulator.

## 8. RELATED WORK

In its early days the cloud has been mainly simulated through the already available grid computing simulators, e.g., GangSim [70], GridSim [71], OptorSim [72] and SimGrid [73]. The huge popularity gained by cloud computing in the subsequent years has encouraged the development of new simulator tailored for the cloud such as CloudSim [20], GreenCloud [21], iCanCloud [22] and GloudSim [74].

CloudSim [20] has been developed in the CLOUDS Lab at University of Melbourne. Its layered architecture enables the modeling of data center environments, providing supports to: network topology, VMs provisioning, CPU and memory allocation and application execution. The main focus of CloudSim is the evaluation of resource allocation policies, also with respect to costs, both from users' and providers' perspectives. CloudSim proves to be a solid simulator to accurately model the behavior of few interconnected data centers.

GreenCloud [21] is an extension of ns-2 [75] to simulate energy-aware data centers, focusing on cloud communications (i.e., it is a packet-level simulator). Its peculiarity is a detailed modeling of the energy consumption of the different components that constitute the data center equipment, e.g., computing server, network switches and communication links.

iCanCloud [22] has been developed on top of OMNeT++ [76], by the University of Madrid. It aims at predicting the cost-performance tradeoff when a set of applications is executed on a specific hardware architecture. It is sealed with many pre-built configurations for the storage and hypervisor functionalities, which can be directly used through its GUI (Graphical User Interface). Moreover, the simulation framework includes instance types provided by Amazon.

GloudSim [74], is a distributed cloud simulator based on virtual machines, originally aimed at reproducing the Google cloud environment departing from the same workload data we have used in our experiments [66]. Its main features are the ability to deal with dynamically changing computational resources and several types of events (e.g., killing or hibernating jobs). It is available under GNU GPL v3 license, and it has been applied to the optimisation of check-point intervals in Google tasks.

For all the above mentioned simulators, the detailed characterization of the data center behavior prevents them to simulate heterogeneous, large-scale, complex, cooperative environments such as volunteer clouds. Moreover, the AVOCLOUDY architecture facilitates the transition of the simulated solution to the production environment.

A direct point-to-point comparison [21] among the mentioned simulators and AVOCLOUDY can not be done, since, as already discussed, their purposes are different.

In the recent past, the SimBOINC [77] project has been developed to realize a simulator to test scheduling strategies for the BOINC architecture. But later, the French INRIA decided to hold on the project, due to rapid changing in the BOINC architecture, which prevented them to maintain the simulator updated. To the best of our knowledge, the only other effort to build a volunteer simulator



DOI: 10.1002/spe

can be found in a paper by Byrski *et al.* [78]. However, such a tool has no website, no research projects that use or have used it, no information other than the referenced paper.

## 9. CONCLUSION

Volunteer cloud computing platforms present a more complex environment with respect to traditional clouds, mainly due to their heterogeneity and dynamism. The increasing complexity in managing such systems demands for an accurate evaluation of new solutions, before their actual deployment in the production environment. A simulation-based approach can provide a statistical comfort on the expected performance. The available cloud simulators are not amenable to the design and study of such systems, but only to accurately characterize the behavior of few interconnected data centers, whose characteristics are well-know a priori.

In this work we propose a volunteer cloud computing architecture and a simulator that has its roots on agents, and the autonomic and volunteer paradigms. Our generalized cloud architecture allows us to consider highly heterogeneous environments where both data center and volunteer nodes cooperate. In the study of such kind of systems, it is worth to take into account that the layered and agent-based architecture of AVOCLOUDY allows for a smooth transition from the simulated model to a concrete implementation. The usefulness of the proposed AVOCLOUDY has been demonstrated with a set of case studies that have already benefited from its highly modular and flexible architecture [45–47]. The confidence of the simulator has been validated over a set of real-world experiments based on worldwide distributed computations on top of the PlanetLab platform.

We plan to extend the AVOCLOUDY in many directions, in particular to support the loading of a network structure at boot time. Moreover, to increase the realism, we are evaluating the introduction of an option to specify each network request as an event.

## ACKNOWLEDGEMENTS

We are grateful to the anonymous reviewers for their useful comments and suggestions that have significantly helped us to further improve our simulator and its presentation in the manuscript.

## REFERENCES

1. Costa F, Silva L, Dahlin M. Volunteer Cloud Computing: MapReduce over the Internet. *Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW), 2011 IEEE International Symposium on*, 2011; 1855–1862, doi:10.1109/IPDPS.2011.345.
2. Worldwide LHC Computing Grid. http://wlcg.web.cern.ch/.
3. Montresor A, Abeni L. Cloudy weather for P2P, with a chance of gossip. *Peer-to-Peer Computing (P2P), 2011 IEEE International Conference on*, 2011; 250–259, doi:10.1109/P2P.2011.6038743.
4. Kavalionak H, Montresor A. P2P and Cloud: A Marriage of Convenience for Replica Management. *Self-Organizing Systems*, *Lecture Notes in Computer Science*, vol. 7166, Kuipers F, Heegaard P (eds.). Springer Berlin Heidelberg, 2012; 60–71, doi:10.1007/978-3-642-28583-7_6.
5. Kavalionak H, Carlini E, Ricci L, Montresor A, Coppola M. Integrating peer-to-peer and cloud computing for massively multiuser online games. *Peer-to-Peer Networking and Applications* 2013; :1–19doi:10.1007/s12083-013-0232-4.
6. Babaoglu O, Marzolla M, Tamburini M. Design and Implementation of a P2P Cloud System. *Proceedings of the 27th Annual ACM Symposium on Applied Computing*, SAC '12, ACM: New York, NY, USA, 2012; 412–417, doi:10.1145/2245276.2245357.
7. PlanetLab: an open platform for developing, deploying, and accessing planetary-scale services. https://www.planet-lab.org/.
8. Babaoglu O, Marzolla M. The People's Cloud. *IEEE Spectrum* Oct 2014; :44–49.
9. European Integrated Project 223850 NaDa (Nano Datacenters). http://www.nanodatacenters.eu/.
10. European Project RI-261556 EDGI (European Desktop Grid Initiative). http://edgi-project.eu/.
11. Segal B, Buncic P, Quintas DG, Gonzales DL, Harutyunyan A, Rantala J, Weir D. Building a Volunteer Cloud. *Technical Report*, CERN September 2009. http://ben.web.cern.ch/ben/Ven_abs.pdf.
12. BOINC: A System for Public-Resource Computing and Storage. http://boinc.berkeley.edu/.
13. HTCondor. http://research.cs.wisc.edu/htcondor/.
14. Seattle: a platform for educational cloud computing. https://seattle.poly.edu/html/.

15. SETI@home: an experiment in public-resource computing. http://setiathome.ssl.berkeley.edu/.
16. OurGird. http://www.ourgrid.org/.
17. Ghafarian T, Deldari H, Javadi B, Yaghmaee MH, Buyya R. CycloidGrid: A proximity-aware P2P-based resource discovery architecture in volunteer computing systems. *Future Generation Computer Systems* 2013; **29**(6):1583–1595, doi:10.1016/j.future.2012.08.010. Including Special sections: High Performance Computing in the Cloud & Resource Discovery Mechanisms for P2P Systems.
18. Ghafarian T, Deldari H, Javadi B, Buyya R. A proximity-aware load balancing in peer-to-peer-based volunteer computing systems. *The Journal of Supercomputing* 2013; **65**(2):797–822, doi:10.1007/s11227-012-0866-7.
19. Caton S, Rana O. Towards autonomic management for Cloud services based upon volunteered resources. *Concurrency and Computation: Practice and Experience* 2012; **24**(9):992–1014, doi:10.1002/cpe.1715.
20. Calheiros RN, Ranjan R, Beloglazov A, De Rose CAF, Buyya R. CloudSim: a toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms. *Softw. Pract. Exper.* Jan 2011; **41**(1):23–50, doi:10.1002/spe.995.
21. Kliazovich D, Bouvry P, Khan SU. GreenCloud: a packet-level simulator of energy-aware cloud computing data centers. *The Journal of Supercomputing* 2012; **62**(3):1263–1283, doi:10.1007/s11227-010-0504-1.
22. Núñez A, Vázquez-Poletti JL, Caminero AC, Castañé GG, Carretero J, Llorente IM. iCanCloud: A Flexible and Scalable Cloud Infrastructure Simulator. *J. Grid Comput.* Mar 2012; **10**(1):185–209, doi:10.1007/s10723-012-9208-5.
23. Mayer P, Kroiss C, Velasco J. ASCENS Technical Report: TR20120500 - The Science Cloud Case Study Overview and Scenarios. *Technical Report*, ASCENS Jul 2012. URL http://www.pst.ifi.lmu.de/~mayer/papers/2012-05-00-TR-SCP-1.pdf.
24. European Integrated Project 257414 ASCENS (Autonomic Service Component ENSembles). http://www.ascens-ist.eu/.
25. Satyanarayanan M, Bahl P, Caceres R, Davies N. The Case for VM-Based Cloudlets in Mobile Computing. *Pervasive Computing, IEEE* oct-dec 2009; **8**(4):14–23, doi:10.1109/MPRV.2009.82.
26. Anderson DP. BOINC: A System for Public-Resource Computing and Storage. *Proceedings of the 5th IEEE/ACM International Workshop on Grid Computing*, GRID '04, IEEE Computer Society, 2004; 4–10, doi:10.1109/GRID.2004.14.
27. Thain D, Tannenbaum T, Livny M. Distributed computing in practice: the Condor experience. *Concurrency - Practice and Experience* 2005; **17**(2-4):323–356.
28. Brasileiro F, Araiijo E, Voorsluys W, Oliveira M, Figueiredo F. Bridging the High Performance Computing Gap: the OurGrid Experience. *Cluster Computing and the Grid, 2007. CCGRID 2007. Seventh IEEE International Symposium on*, 2007; 817–822, doi:10.1109/CCGRID.2007.28.
29. Cappos J, Beschastnikh I, Krishnamurthy A, Anderson T. Seattle: a platform for educational cloud computing. *SIGCSE*, et al et al SF (ed.), ACM, 2009; 111–115.
30. Anderson DP, Cobb J, Korpela E, Lebofsky M, Werthimer D. SETI@home: an experiment in public-resource computing. *Commun. ACM* Nov 2002; **45**(11):56–61, doi:10.1145/581571.581573.
31. Kephart JO, Chess DM. The Vision of Autonomic Computing. *Computer* 2003; **36**:41–50.
32. Amoretti M, Picone M, Zanichelli F, Ferrari G. Simulating Mobile and Distributed Systems with DEUS and ns-3. *HPCS*, 2013.
33. Distributed Systems Group, DEUS project homepage. http://code.google.com/p/deus/.
34. Sebastio S, Vandin A. MultiVeStA: Statistical Model Checking for Discrete Event Simulators. *VALUETOOLS*, 2013.
35. MultiVeStA: code.google.com/p/multivesta.
36. Baset S, Schulzrinne H. An Analysis of the Skype Peer-to-Peer Internet Telephony Protocol. *INFOCOM 2006. 25th IEEE International Conference on Computer Communications. Proceedings*, 2006; 1–11, doi:10.1109/INFOCOM.2006.312.
37. Montresor A. A robust protocol for building superpeer overlay topologies. *Peer-to-Peer Computing, 2004. Proceedings. Proceedings. Fourth International Conference on*, 2004; 202–209, doi:10.1109/PTP.2004.1334948.
38. Pyun YJ, Reeves DS. Constructing a Balanced, (log(N)/loglog(N))-Diameter Super-Peer Topology for Scalable P2P Systems. *Peer-to-Peer Computing, IEEE International Conference on* 2004; **0**:210–218, doi:10.1109/PTP.2004.1334949.
39. Amoretti M. A Modeling Framework for Unstructured Supernode Networks. *Communications Letters, IEEE* October 2012; **16**(10):1707–1710, doi:10.1109/LCOMM.2012.082012.121554.
40. Koo SG, Lee CSG, Kannan K. A genetic-algorithm-based neighbor-selection strategy for hybrid peer-to-peer networks. *Computer Communications and Networks, 2004. ICCCN 2004. Proceedings. 13th International Conference on*, 2004; 469–474, doi:10.1109/ICCCN.2004.1401710.
41. Huang K, Wang L, Zhang D, Liu Y. Optimizing the BitTorrent Performance Using an Adaptive Peer Selection Strategy. *Future Gener. Comput. Syst.* Jul 2008; **24**(7):621–630, doi:10.1016/j.future.2007.10.001.
42. D'Acunto L, Andrade N, Pouwelse J, Sips H. Peer Selection Strategies for Improved QoS in Heterogeneous BitTorrent-Like VoD Systems. *Multimedia (ISM), 2010 IEEE International Symposium on*, 2010; 89–96, doi:10.1109/ISM.2010.22.
43. Koo SG, Kannan K, Lee CG. On neighbor-selection strategy in hybrid peer-to-peer networks. *Future Generation Computer Systems* 2006; **22**(7):732–741, doi:10.1016/j.future.2006.02.015.
44. Bolch G, Greiner S, de Meer H, Trivedi KS. *Queueing Networks and Markov Chains*. 2 edn., Wiley-Interscience, 2006.
45. Amoretti M, Lluch Lafuente A, Sebastio S. A Cooperative Approach for Distributed Task Execution in Autonomic Clouds. *Parallel, Distributed and Network-Based Processing (PDP), 2013 21st Euromicro International Conference on*, 2013; 274–281, doi:10.1109/PDP.2013.47.
46. Sebastio S, Amoretti M, Lluch Lafuente A. A Computational Field Framework for Collaborative Task Execution in Volunteer Clouds. *Proceedings of the 9th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, SEAMS 2014, ACM: New York, NY, USA, 2014; 105–114, doi:10.1145/2593929.2593943.

47. Celestini A, Lluch Lafuente A, Mayer P, Sebastio S, Tiezzi F. Reputation-Based Cooperation in the Clouds. *Trust Management VIII*, *IFIP Advances in Information and Communication Technology*, vol. 430, Zhou J, Gal-Oz N, Zhang J, Gudes E (eds.), Springer Berlin Heidelberg, 2014; 213–220, doi:10.1007/978-3-662-43813-8_15.
48. Hill MD, Marty MR. Amdahl's Law in the Multicore Era. *Computer* Jul 2008; **41**(7):33–38, doi:10.1109/MC.2008.209.
49. Sun XH, Chen Y. Reevaluating Amdahl's law in the multicore era. *J. Parallel Distrib. Comput.* Feb 2010; **70**(2):183–188, doi:10.1016/j.jpdc.2009.05.002.
50. Woo DH, Lee HH. Extending Amdahl's Law for Energy-Efficient Computing in the Many-Core Era. *Computer* Dec; **41**(12):24–31, doi:10.1109/MC.2008.494.
51. Saino L, Cocora C, Pavlou G. A Toolchain for Simplifying Network Simulation Setup. *Proceedings of the 6th International ICST Conference on Simulation Tools and Techniques*, SIMUTOOLS '13, ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering): ICST, Brussels, Belgium, Belgium, 2013.
52. Picone M, Amoretti M, Zanichelli F. Evaluating the robustness of the DGT approach for smartphone-based vehicular networks. *Local Computer Networks (LCN), 2011 IEEE 36th Conference on*, 2011; 820–826, doi:10.1109/LCN.2011.6115557.
53. Shoch JF, Hupp JA. Measured Performance of an Ethernet Local Network. *Commun. ACM* Dec 1980; **23**(12):711–721, doi:10.1145/359038.359044.
54. DR B, JC M, CA K. Measured capacity of an Ethernet: myths and reality. *Technical Report*, Digital, Western Research Laboraroty September 1988.
55. Lian FL, Moyne JR, Tilbury DM. Performance Evaluation of Control Networks: Ethernet, ControlNet, and DeviceNet. *IEEE Control System Magazine* Feb 2001; :66–83doi:10.1109/37.898793.
56. networks D. The Gigabit Experts. http://www.dssnetworks.com/v3/FAQs.asp#Performance.
57. Fritz! Transmission speed of wireless connections is low. http://en.avm.de/nc/service/fritzbox/fritzbox-7390/knowledge-base/publication/show/514_Transmission-speed-of-wireless-connections-is-low/.
58. Wikipedia. IEEE 802.11. http://en.wikipedia.org/wiki/IEEE_802.11.
59. Lee SJ, Sharma P, Banerjee S, Basu S, Fonseca R. Measuring bandwidth between PlanetLab nodes. *Proceedings of the 6th international conference on Passive and Active Network Measurement*, PAM'05, Springer-Verlag: Berlin, Heidelberg, 2005; 292–305, doi:10.1007/978-3-540-31966-5_23.
60. Thamarai Selvi S, Kumari MSS, Prabavathi K, Kannan G. Estimating job execution time and handling missing job requirements using rough set in grid scheduling. *Computer Design and Applications (ICCDA), 2010 International Conference on*, vol. 4, June; V4–295–V4–299, doi:10.1109/ICCDA.2010.5541135.
61. Casolari S, Colajanni M, Tosi S, Presti FL. Real-time models supporting resource management decisions in highly variable systems. *IPCCC*, 2010; 247–254.
62. Silberschatz A, Galvin PB, Gagne G. *Operating System Concepts*. 8th edn., Wiley Publishing, 2008.
63. Pajek: vlado.fmf.uni-lj.si/pub/networks/pajek/.
64. Gephi: gephi.org/.
65. Google Cloud Platform - Machine Type. https://cloud.google.com/compute/#us.
66. Wilkes J. More Google cluster data. Google research blog Nov 2011. Posted at http://googleresearch.blogspot.com/2011/11/more-google-cluster-data.html.
67. Mishra AK, Hellerstein JL, Cirne W, Das CR. Towards Characterizing Cloud Backend Workloads: Insights from Google Compute Clusters. *ACM SIGMETRICS Performance Evaluation Review* 2010; **37**(4):34–41.
68. Pianini D, Sebastio S, Vandin A. Distributed statistical analysis of complex systems modeled through a chemical metaphor. *High Performance Computing Simulation (HPCS), 2014 International Conference on*, 2014; 416–423, doi:10.1109/HPCSim.2014.6903715.
69. CoDeploy: A Scalable Deployment Service for PlanetLab. http://codeen.cs.princeton.edu/codeploy/.
70. Dumitrescu C, Foster IT. GangSim: a simulator for grid scheduling studies. *CCGRID*, IEEE Computer Society, 2005; 1151–1158. URL http://dblp.uni-trier.de/db/conf/ccgrid/ccgrid2005.html#DumitrescuF05.
71. Buyya R, Murshed M. GridSim: A Toolkit for the Modeling and Simulation of Distributed Resource Management and Scheduling for Grid Computing. *Concurrency and Computation: Practice and Experience (CCPE)* 2002; **14**(13):1175–1220.
72. Bell WH, Cameron DG, Capozza L, Millar AP, Stockinger K, Zini F. OptorSim- A Grid Simulator for Studying Dynamic Data Replication Strategies. *International Journal of High Performance Computing Applications* 2003; :2003.
73. Legrand A, Marchal L, Casanova H. Scheduling Distributed Applications: The SimGrid Simulation Framework. *Proceedings of the 3st International Symposium on Cluster Computing and the Grid*, CCGRID '03, IEEE Computer Society: Washington, DC, USA, 2003; 138–.
74. Di S, Cappello F. Gloudsim: Google trace based cloud simulator with virtual machines. *Software: Practice and Experience* 2014; :n/a–n/adoi:10.1002/spe.2303. URL http://dx.doi.org/10.1002/spe.2303.
75. Issariyakul T, Hossain E. *Introduction to Network Simulator NS2*. 1 edn., Springer Publishing Company, Incorporated, 2008.
76. Varga A, Hornig R. An overview of the OMNeT++ simulation environment. *Proceedings of Simutools '08*, Simutools '08, ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering): ICST, Brussels, Belgium, Belgium, 2008; 60:1–60:10.
77. SimBOINC: simboinc.gforge.inria.fr/.
78. Byrski A, Felus M, Gawlik J, Jasica R, Kobak P, Jankowski G, Nawarecki E, Wroczynski M, Majewski P, Krupa T, *et al.*. Volunteer Computing Simulation using RePast and MASON. *Compure Science* 2013; .

# APPENDIX

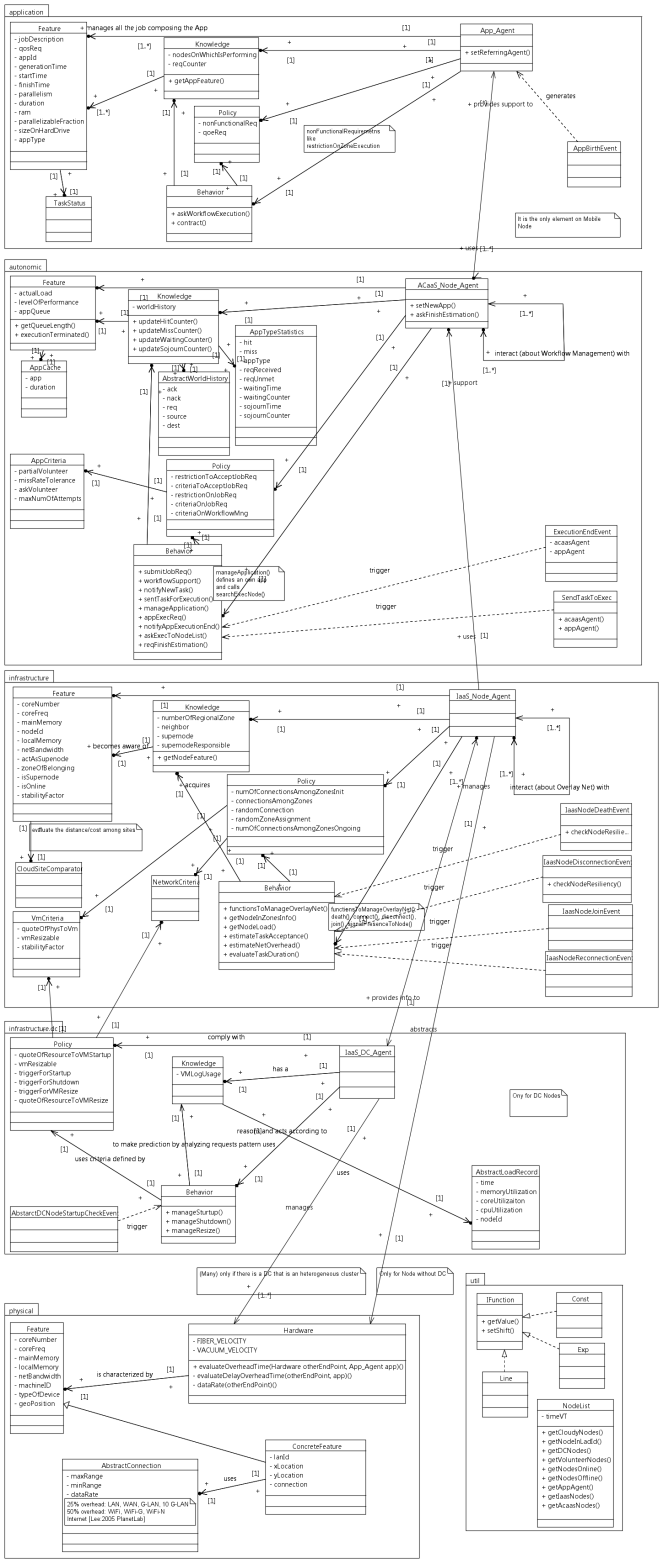


Figure 12. Class Diagram of the AVOCLOUDY simulator.



DOI: 10.1002/spe

The state diagram for the task status and its transitions is taken from [62] and depicted in Figure 13.

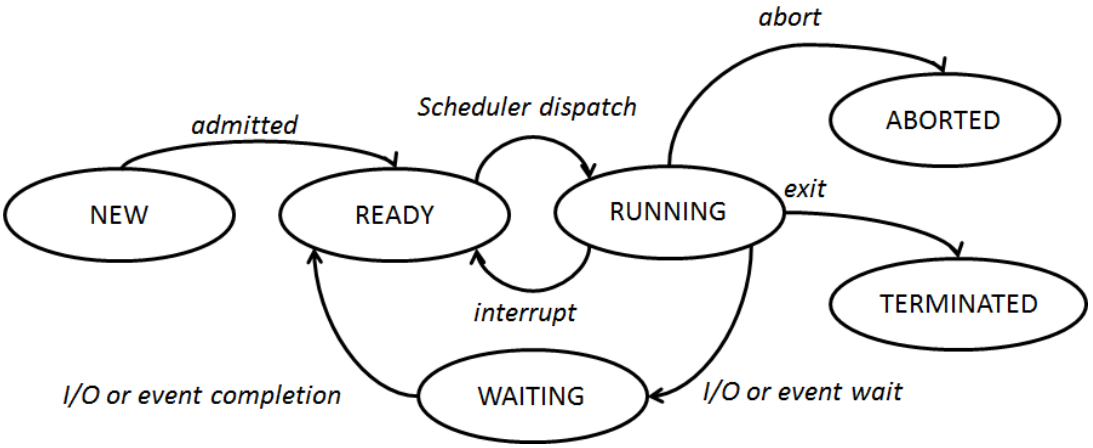


Figure 13. State diagram for the task status transitions.



DOI: 10.1002/spe