SDN-based and Multitenant-Aware Resource Provisioning Mechanism for Cloud-based Big Data Streaming

Cleverton Vicentini[1, 2], Altair Santin[1], Eduardo Viegas[1], Vilmar Abreu[1]
[1]Graduate Program in Computer Science / Pontifical Catholic University of Parana, Curitiba, Parana, Brazil
[2]Federal Institute of Parana, Curitiba, Parana, Brazil
{cleverton, santin, eduardo.viegas, vilmar.abreu}@ppgia.pucpr.br

***Abstract**: Cloud computing provides elastic on-demand resource allocation, enabling big data systems to process large amounts of streaming data in real time. However, a shared cloud infrastructure (multitenant at the hypervisor level) may reduce system performance or even resource availability, particularly when big data processing demands significantly increase through concurrent task allocations on the same physical hardware. Such situations are not easily detectable from the tenant's perspective, because the tenant may suffer from poor performance without knowing why, as the infrastructure is not under the tenant's control. Moreover, as task processing demand changes over time, the available infrastructure may be insufficient owing to increased processing load or multitenant interference. This paper presents a multitenant-aware resource provisioning mechanism that is independent of any hypervisor and can perform task scheduling and dynamic ongoing task rescheduling for big data streaming while considering the state of each virtual machine (VM). Moreover, the proposed mechanism ensures load balancing through several cloud-based clusters of VMs using a software-defined network (SDN). The prototype was implemented using Apache Storm (big data), Helion Eucalyptus (cloud computing), and Floodlight (SDN). The evaluation shows that when the resources are under multitenant interference, our proposal results in an improvement of 50.1% for CPU-bound tasks, 62.3% for disk-bound tasks, and 43.8% for network-bound tasks. In addition, the load balancer forwarded 72.04% of the load to a fully available cluster, meaning that our mechanism can realize a 22.04% improvement in effectiveness over traditional approaches.*

Keywords: *Dynamic rescheduling for big data, SDN-based load balancing, Cloud-based big data streaming, Dynamic resource provisioning mechanism*

1 INTRODUCTION

In general, big data techniques are used when it is infeasible to handle the amount of data through conventional processing and storage resources [1]. The processing of such data requires the development of a distributed infrastructure, traditionally composed of distributed filesystems, task schedulers, programming models, and batch or real-time (data streaming) processing. Batch frameworks for big data processing (e.g., Hadoop [2]) are characterized by the storage of all data in a distributed filesystem prior to processing. In contrast, streaming frameworks for big data processing (e.g., Storm [3, 4]) continuously process data arriving in the steam.

Big data streaming frameworks require adequate infrastructure for high-performance computing (HPC). Cloud computing is one alternative that supports HPC, as it offers massive storage capacity, scalability, resilience, and high availability [5].

Multitenant scenarios, which are common in cloud computing infrastructure, occur when several virtual machines (VMs) from different cloud tenants share the same physical hardware [6] controlled by a hypervisor. However, the benefits of physical hardware (resource) sharing contradict the time-multiplexing of the physical infrastructure. This conflict can cause undesirable performance variations in cloud client applications because of concurrent access requests for the same resources. Moreover, cloud providers can overbook resources by accepting more VMs than its infrastructure can support, causing further impacting VM performance [7].

When a customer contracts a cloud service, a service-level agreement (SLA) is established to ensure a specific quality of service. The customer believes that the service provider will honor the contracted SLA [8]. However, the provider may overbook its resources by allocating more VMs than the available physical infrastructure can support, thus affecting system performance [9]. The main problem is that the customer's view of the provided resources is virtual, given that the service provider exclusively performs the contracted SLA metrics monitoring. Thus, if a customer does not have an additional mechanism to monitor available resources, a customer can suffer from performance variability without being notified.

Performance variability in cloud computing becomes critical in big data streaming, as there are often strict real-time requirements. For instance, if performance declines, a data stream-based application may not work fast enough to adhere to real-time requirements. Ameliorating this performance degradation in a multitenant cloud infrastructure is a challenge because there is no control over the resources used by different tenants—they can only manage the virtualized resources, i.e., inside the VMs.

Several studies have focused on improving physical resource management and resource utilization [10, 11, 12] or evaluating and minimizing performance variability by choosing or creating homogeneous VMs [13, 14, 15]. Other research has sought to improve performance by scheduling tasks for distributed applications by considering the availability of virtual resources [16, 17, 18]. However, physical resources and their allocation are under the control of cloud providers. Thus, the performance variability caused by concurrent resource utilization remains an unaddressed issue.

Moreover, the processing load in big data streaming may change over time, thus demanding more processing resources. A common approach in such cases is increasing the number of VMs (nodes) available to the client of the cloud [19]. However, in general, this approach does not consider the state of physical resources in the cloud (host). Providing more available nodes to the client increases the infrastructure management complexity, and the application might still suffer from performance variability because of other cloud tenants. Nonetheless, the increase in the number of nodes requires the cloud resource provisioning framework to communicate with the big data processing framework [2, 4].

A popular approach that reduces the complexity of infrastructure management during resource provisioning is to provide resources horizontally. This method duplicates the cluster (the set of nodes of a specific tenant) instead of adding more nodes to it. Thus, the resource-provisioning framework does not need to communicate with the currently executing application [20]. Despite reducing the complexity of the resource-provisioning framework, this approach faces other challenges. The client application must be aware that different clusters (with different network addresses) can process the load and forward its request to the appropriate cluster. Nonetheless, identifying which cluster should process a request to ensure load balancing is an intensive task [21]. This task usually addresses network or system metrics to determine the target of a request or requires changes in the client application to obtain its advantages [20].

Nowadays, the system and network status are considered together to perform load balancing through software-defined networks (SDNs) [23, 24, 25]. SDNs leverage the packet forwarding mechanism of switches/routers to a centralized entity named the Controller, which defines the virtual network topology at the software level [26]. The main advantage of the SDN model is its capacity to define the network topology in real time. This eliminates the usage of legacy hardware with static policies for network route configuration [26]. Normally, when performing load balancing through an SDN, we simply compute which node is the current target of a request by considering the current network load [23, 24]. Although this approach can deliver network load to less-used nodes or through less-busy paths, it lacks the capacity to communicate with the cloud computing infrastructure. Thus, it cannot take advantage of the cloud node elasticity and does not consider multitenant features and is thereby unable to establish whether a node can execute a processing load with respect to the current state of physical resources. Moreover, when the whole cluster is exhausted, the processing load can be forwarded to a target node regardless of the cluster's capacity to process it.

This paper proposes a novel approach to provide cloud-based elasticity to big data streaming while considering the availability of physical resources in cloud-based environments. The main contributions of this paper are as follows:

- A dynamic scheduler and rescheduler for big data streaming processing frameworks that schedules and periodically reschedules processing tasks to less overloaded nodes according to their physical and virtual resource states.
- Elastic resource provisioning through network functions virtualization (NFV, an SDN facility to aggregate software applications). By using the cloud computing elasticity, the resource provisioning is conducted transparently, enabling the creation or termination of a cluster. However, in our proposal, it is done considering the general state of the infrastructure, which involves the state of each VM in a physical and virtual cluster. In other words, processing resources are transparently delivered according to the impact of multi-tenancy on the cluster and the required processing over time.
- Load balancing through NFV using the SDN controller data, which enables load balancing across several clusters according to the current state of each cluster while considering the physical and virtual current state of each cluster node (VM).

The remainder of this paper is organized as follows. Section 2 addresses the fundamentals of multitenant cloud infrastructure, SDNs, and Apache Storm. Section 3 presents related works. Section 4 introduces a case study that shows the impact of multitenant infrastructure usage on big data streaming applications. Section 5 describes the proposed transparent and multitenant-aware resource provisioning model. Section 6 discusses the prototype, and Section 7 presents the results of its evaluation. Finally, Section 8 draws together our conclusions.

2 BACKGROUND

2.1 Multitenancy in Cloud Computing

A cloud computing environment can be briefly characterized as a hardware infrastructure that provides on-demand computational services, such as processing, storage, and networking [5]. This is achieved through the virtualization concept [27], which abstracts the physical hardware layer and shares it among several hosts, known as VMs. The mechanism responsible for physical resource abstraction and controlling its access is known as the Hypervisor [28].

In general, cloud computing can be classified as private, public, hybrid, or community [5]. In private cloud computing, the VM administrator has access to the Hypervisor, thus allowing the administrator to handle the VMs disposition and control the access to physical resources. In public, hybrid, or community cloud computing, the VM administrator does not normally have access to the Hypervisor. Thus, the administrator cannot determine the status of VM physical resource, as the hardware is shared among several other tenants.

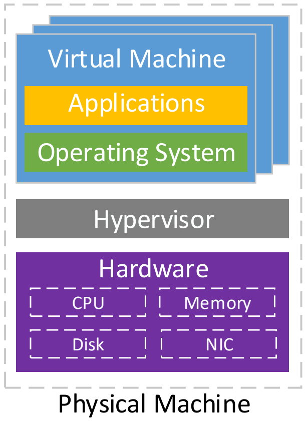


Fig. 1. Typical cloud computing shared environment scenario.

Figure 1 shows a typical hardware virtualization concept performed by a Hypervisor. The Hardware resources (CPU, disk, network, and other components) are abstracted and accessed through the Hypervisor. When a VM requests access to the hardware (physical resource), it must communicate with the Hypervisor. The sharing of physical resources among several other VMs is known as a multitenant scenario.

The way in which the VM accesses its host physical resource depends on the Hypervisor being used. For instance, the VMWare hypervisor [29], by default, shares the physical CPU among its VMs through the fair share algorithm, which aims to provide CPU access for VMs according to their historical usage. In contrast, the Xen [30] hypervisor provides equal CPU time regardless of historical usage [31]. The VM processing capacity is related to the way physical resources are currently used [32]. Resources such as network access and disk read and write are affected by the concurrent access occurring in multitenant environments.

A cloud computing manager must be used to enable a cloud computing environment. A well-known cloud manager is the HPE (Hewlett-Packard Enterprise) Helion Eucalyptus [33], which is an open-source solution for building private clouds that are compatible with Amazon Web Services (AWS).

2.2 Software-Defined Networks and Network Function Virtualization

The SDN model decouples the control and data planes [26]. The control plane operates network device management, whereas the data plane is the hardware layer responsible for forwarding network packets according to the policies defined in the control plane. This decoupling turns the network switches/routers into simple forwarding devices, while logic control is implemented in the controller that operates as a centralized network operating system. Thus, the SDN is composed of two entities: the Controller and the Forwarding Devices (SDN switch or router).

The Forwarding Devices have a set of flow rules that are applied in the network packets. A flow rule can be defined by combining different matching fields (e.g., IP and Port). As each network packet arrives, the Forwarding Device analyzes its characteristics and compares them to the current flow rules. In general, there is a default flow rule in which the unmatched network packets are forwarded to the Controller, which in turn establishes the network packet flow rule and updates the flow rule set in the forwarding device.

The Controller establishes the set of flow rules in each forwarding device, defining the communication policy between forwarding devices through a software-level function. The Controller demands a communication protocol (control/data plane) that installs the flow rules in the network devices. The emerging solution is the OpenFlow protocol, proposed by [34] and standardized by the Open Network Foundation [35], which allows Controllers to access and modify the flow rules set remotely in forwarding devices.

SDN characteristics such as agile network programmability, centralized controller, and open-standard usage (OpenFlow) have led to increased interest in the development of solutions that combine the cloud platform with the SDN architecture [36, 37, 38] to optimize services and provide new technological approaches.

The concept of NFV [39, 40] is generally used as a complement to SDN. The main motivation of NFV [39] is to migrate the network functions implemented by dedicated hardware that executes specific functions, such as routers, firewalls, gateways, load balancers, and other devices, to generic devices operating on common x86 architecture. This characteristic allows virtualized network functions to be installed at any network point, making the scenario dynamic and adaptable to consumer needs while also expanding the service provider possibilities. In this manner, it becomes possible to implement several NFVs, each having its own objective, and integrate them with the SDN that updates flow rules according to NFV needs.

Several SDN controllers [56, 57] that support OpenFlow protocols have been developed. Currently, the most popular is the open-source Floodlight [41]. Floodlight provides full integration with OpenFlow protocols and Representational State Transfer (REST) application programming interfaces (APIs), which enable the development of NFV applications, allowing forwarding device flows to be defined through a REST function.

2.3 Apache Storm

To evaluate the proposed architecture (Section 5), it is necessary to use a framework that typically operates in a multitenant cloud environment, performs distributed big data processing, and demands real-time (data streaming) decision making. This subsection describes Apache Storm, a well-known robust architecture for real-time stream data processing.

2.3.1 Storm Overview

In the Storm platform [3], data processing is performed through the definition of topologies. A topology defines the necessary infrastructure format to perform the processing over data sources. Two main components perform topology processing: (i) *Spouts* and (ii) *Bolts*. The *Spouts* are responsible for reading the data source (internal or external) and generating data streaming for the platform. The *Bolts* are responsible for consuming the data generated by the *Spouts* and performing related processing. Note that the *Bolts* may also consume the data generated by other *Bolts*. Each processing unit generated by the *Spouts* and *Bolts* is called a tuple (message exchanged between *Spout–Bolt* or *Bolt–Bolt*). A topology is composed of several *executors* (threads) that perform the predetermined processing of a *Spout* or *Bolt.* During execution, a topology formed by the *Spouts* and *Bolts* is known as a logic abstraction of the Storm environment.

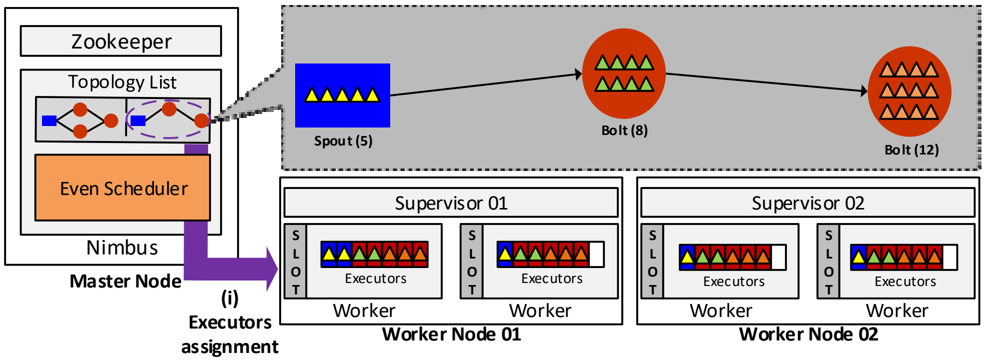


Fig. 2. Storm architecture overview with topology scheduling

The Storm architecture, called a Storm Cluster (Figure 2), is composed of a server named the *Master Node* and one or more *Worker Nodes*[1]. The *Master Node* executes the *Nimbus* process (daemon), which is responsible for submitting the topologies to be executed by the *Worker Nodes*. Each *Worker Node* has a daemon named as *Supervisor*, which is responsible for managing the tasks (based on the topologies) submitted to *Nimbus*. Besides the *Supervisor*, the *Worker Nodes* have *Slots,* which are the available communication ports. Normally, the number of available *Slots* for each *Worker Node* is related to the number of CPU cores on the node. To process the tasks in each *Slot*, it is necessary to associate a *Worker* (process) with one or more *executors* (threads). The communication between the *Master Node* and *Worker Nodes* is performed through the Zookeeper [42], an Apache tool that provides a service for the configuration, coordination, and discovery of services in distributed applications.

2.3.2 Scheduler

The Storm scheduling process defines how topologies are disposed of in the available infrastructure. Each newly- submitted topology must be scheduled to allow the *Worker Nodes* to execute their processing. Thus, the *Nimbus* uses *EvenScheduler* (ES) by default. The ES relies on a round-robin policy to uniformly distribute the *executors* among the *workers* allocated to the topology. To show such policy execution, the *Word Count* (WC) topology [43] is used, which is composed of the following:

- Sentence: *Spout* formed by five *executors*, operating as a topology data source, through periodic phrase generation.

[1] Note the distinction between *(i) worker* and *(ii) worker node*: a *worker* is a process, composed by the topology executors, whereas the *worker node* is a machine that belongs to the Storm cluster.

- Split: *Bolt* formed by eight *executors*. Responsible for reading the phrase generated by the Spout and splitting the phrase into words to send to the next component.
- Count: *Bolt* formed by twelve *executors*. Responsible for counting the words generated by the *Split.*

Figure 2 shows an overview of the *Storm Cluster* with topology scheduling. The WC topology is formed by 25 *executors,* which are distributed across the *Workers* through the ES policy. The *Nimbus* has a topology waiting to be scheduled according to the scheduler policy. In this case, the first topology to be submitted is the WC (Figure 2, event i). To illustrate such a scenario, the *Spout* is shown as yellow triangles and the *Bolts* are illustrated as green triangles. The *executors* are shown as orange triangles.

In Figure 2, it is possible to identify the ES behavior as it schedules the *executors* among the available *Workers*. The 25 *executors* are distributed such that all available topology *Slots* are occupied, regardless of their computational state. Briefly, the ES selects the first *executor* and allocates it to the first available *Slot*, then selects the second *executor* and allocates it to the next *Slot*. This process is repeated for all remaining topology *executors*. Although the ES relies on a strategy that uniformly distributes the *executors* to the *Workers*, this approach is not always efficient. In multitenant environments, applying the scheduling policy without considering the physical machine state may negatively affect performance during the topology execution.

3 RELATED WORKS

Performance variability in public and private clouds has been addressed by a number of studies. Schad et al. [13] reported a large variation in performance on Amazon EC2, mainly caused by different cloud loads at different times of the day and physical hardware variations. The authors were able to identify performance variability using microbenchmarks within the cloud client infrastructure.

On the other hand, to address performance variations in private clouds Rego et al. [14] and Galante et al. [15] proposed the creation of homogeneous VMs according to the physical hardware used. Although their work minimized performance variability, it is not applicable to public cloud infrastructures, as the client does not have control over physical resources. Shen et al. [10] presented a system to automate the elastic resource scaling for multitenant cloud computing infrastructures, performing resource provisioning according to a processing forecasting mechanism. Despite presenting significant improvements over traditional approaches, their work required access to the cloud physical infrastructure, while also did not consider the multitenancy impact. The evaluation of resources other than the CPU was proposed by He et al. [11] by using a probabilistic model to select physical hosts for VM allocation. Their results showed that not only the CPU must be considered but also other physical resources. However, they only considered the cloud provider perspective during resource provisioning. Finally, Tomas and Tordsson [12] showed that when a multitenancy scenario is considered, the performance must be continuously evaluated, as resource utilization changes overtime. In this context, our work leverages the approach from [13] to identify multitenancy issues through microbenchmarks while also considering several physical resources as in [11] during load distribution in an ongoing manner as in [12]. Finally, to the best of our knowledge, ours is the first approach that takes into account the cloud client perspective.

When the application level is considered, approaches in the Apache Storm scheduler try to choose the better distribution and redistribution of application tasks by considering virtualized resource availability. Aniello, Baldoni, and Querzoni [17] presented two scheduler approaches: a topology-based and network traffic-based approach. The topology-based scheduler works offline and considers only the components (spouts and bolts) and their connections in the topologies. The network traffic-based scheduler works online and considers network traffic among worker nodes in order to distribute and redistribute tasks. However, neither approach is transparent to applications, demanding the topology source code instrumentation and prior knowledge of the demanded topology processing. Xu et al. [16] presented T-Storm, an online Storm scheduler that distributes and redistributes tasks according to CPU load and inter-executor traffic among worker nodes. T-Storm is transparent to Storm applications and uses the virtual CPU status and number of tuples sent between executors (not the network traffic) to schedule tasks. Their approach, however, is not applicable in the multitenant context, where the virtual resource may not represent the actual physical resource state. A more comprehensive approach that considers several resources state is the R-Storm proposed by Peng et al. [53], which considers a homogeneous cluster and the processing, network, and memory resources. However, similar to [16] the resources are considered virtually, as only the internal stats of the operating system (OS) are used.

In other big data frameworks, the scheduling process also considers only the virtual resource availability. For instance, Grandl et al. [18] presented Tetris, a Hadoop multiresource scheduler that takes into account the CPU, memory, disk, and network status. Tasks are grouped by multiresource requirements, supporting complementary objectives and minimizing job completion time. Their approach requires source code instrumentation and prior knowledge of the demanded resources. Finally, Yan et al. [54] addressed scheduling tasks by considering heterogeneous multicore processors. Virtual resource pools were used to classify and distribute resources (based on core types) and tasks (based on requirements). However, physical resources and allocation are under the control of cloud providers, and the performance variability caused by concurrent resource utilization remains an open issue. In contrast to [53], our proposed scheduler takes into account the physical status of several resources. Moreover, we do not require the source code instrumentation as in [17, 18] by computing the physical node state over several resources.

Finally, according to Kreutz et al. [26] load balancing was one of the first applications developed in SDN. The Plug-n-Server [24] and Aster*x[25] presented by Handigol et al. were proposed by the SDN development team at Stanford University, with the aim of minimizing server response time by evaluating the communication link (congestion) and server virtual CPU load. Their approach enabled load balancing at the network-level without relying in application-level schemes. The approach proposed by Zhong, Fang, and Cui [55] implemented an SDN-based load balancer that distributes new requests according to the server response time, thus avoiding any reliance on traditional methods such as ping response times. Our proposed approach leverages the approach by [25, 26] to perform load balancing at the network level without relying in application-level approaches. Moreover, we consider the cluster state (computed through the current physical resource state) during the load balancing. This is a more comprehensive approach than [55], which only considers the response time. To the best of our knowledge, this study is the first to address multitenant clouds in SDN-based load balancing by employing the NFV approach.

4 Preliminary Study

This section evaluates the impact on the Apache Storm framework of operating in a multitenant environment and presents some factors that motivated this work.

4.1 Scenario Configuration

A series of tests were conducted to diagnose existing problems and motivate the proposed solution. To this end, a controlled cloud computing environment was built using HPE Helion Eucalyptus (version 4.3.0). The cloud computing environment consisted of four physical computers, each equipped with an 8-core Intel i7 CPU and 16 GB RAM, connected through a gigabit network interface (NIC). The host runs a minimal CentOS 7, the VMs on an Ubuntu server 16.04, through the Kernel-based VM hypervisor. The cloud computing was divided into two zones, each composed of two physical machines. The client's cloud infrastructure was composed of five VMs (one *Master Node* and four *Worker Nodes*), which were created and assigned to the physical machines according to the Eucalyptus cloud policy (round-robin VM disposal policy).

Apache Storm version 1.0.2 was used in the client's cloud infrastructure. A tool was developed to perform the information reading related to the generation and processing of data stream tuples. The Storm UI REST provided this information [44] and enabled interaction with the Storm cluster. This in turn enabled the management of its operations, such as starting or ending a topology, and provided information about the state of the currently executing topologies. The developed tool requests that the number of tuples be processed at 10 s intervals, which is the Storm update periodicity for this value.

To generate the workload, the WC topology was used, which is publicly available and commonly used in the literature. Additionally, the default topology was modified to allow words to be written and read in the hard disk attached to the physical machine. Three topologies were used, each focusing on the usage of a specific computational resource. To generate the CPU load, the WC topology [43] was used. To evaluate the network usage impact, the *Throughput Test* (TT) [45] topology was utilized to generate random strings of a fixed size of 10 KB. Finally, to evaluate the impact on hard disk throughput, we performed the same approach used by [16], changing the default WC topology to generate each tuple by reading a text file [46] from the hard disk while writing the same words back to the hard disk. This modified topology is named the *Word Count File* (WCF) topology.

With regard to the topologies and their parameters, tests were performed to evaluate their behavior in two distinct scenarios, which illustrated in Figure 3.

- *Baseline Cluster*: Each physical machine hosts a VM, with one machine operating as the *Master Node* and the others operating as *Worker Nodes* that comprise the main cluster. The *Baseline Cluster* VMs are distributed in all Eucalyptus zones. This scenario aims to evaluate the behavior of topologies without external interference to the VM by other cloud clients.
- *Multitenant Cluster*: This scenario aims to evaluate the behavior of topologies while experiencing multitenant interference. Similar to the *Baseline Cluster*, each physical machine hosts one VM to comprise the main cluster. However, the four physical machines add hosts to a secondary Storm cluster (Figure 3, Multitenant Cluster). Thus, this scenario holds two clusters: the main cluster, which is composed of five VMs (distributed in all zones), and a secondary cluster composed of five VMs (distributed solely in Eucalyptus Zone B). This scenario aims to evaluate the *main cluster* topology performance under multitenant interference in half of the physical machines.

Figure 3 shows the cluster distribution in the *Baseline Cluster* and *Multitenant Cluster*. The performance evaluation metrics were measured over the main cluster. The secondary cluster is responsible for the parallel real processing load generation, which occurred through the execution of topologies using the same parameters as the *Main Cluster*. In this manner, our scenarios mimic the real-world environment, in which a physical machine may run several VMs with different priorities and services.

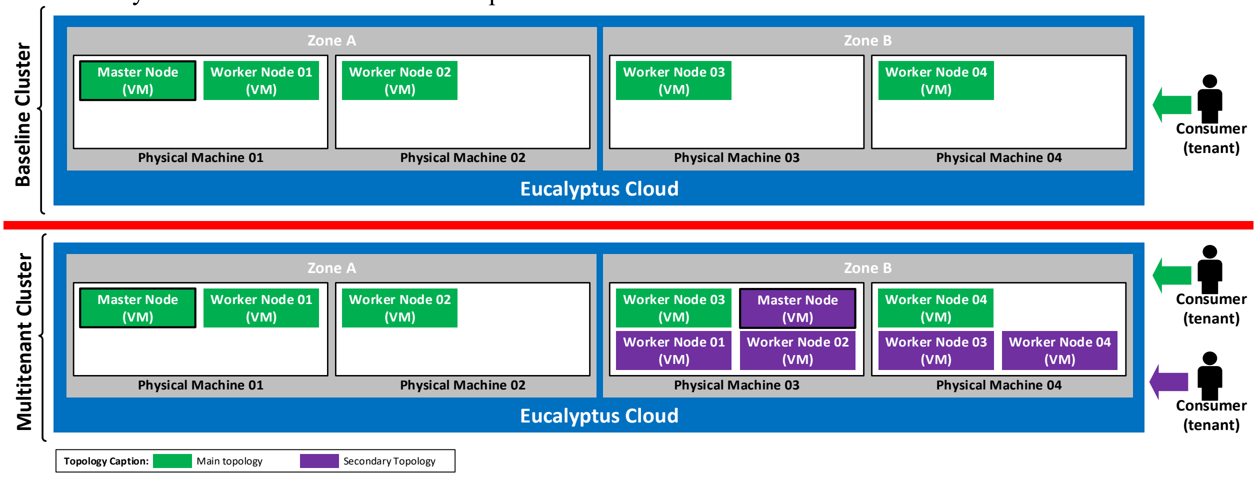


Fig. 3. Testing scenario

4.2 Performance Evaluation

This subsection presents the results of evaluating the multitenant impact of real-time cloud processing frameworks. The execution time for each test was 300 s. The topology parameters are presented in Table 1.

TABLE 1. Main and Secondary topologies configuration (Figure 3)

Topologies	*Number of Workers*	*Spout (Sentence)*	*Bolt (Split)*	*Bolt (Count)*
Baseline Cluster	4	4	8	12
Multitenant Cluster	4	4	8	12

The results are listed in Table 2, where the average (big data) tuples processed per second in both scenarios are given for each tested topology. The *Baseline Cluster* column represents the number of tuples generated per second in an environment without multitenant interference. The *Multitenant Cluster* column lists the percentage of tuples per second under multitenant interference when compared to the *Baseline Cluster*. Considering the former as the baseline proposal, it is possible to note that the *Multitenant Cluster* performance was degraded in all cases, mainly when the *Secondary Topology* executed the same topology (mirror) as the *Main Topology*.

Comparison of the results generated by the *Baseline Cluster* and Multitenant *Cluster* shows that the performance was degraded in all cases where the topology used the same type of resource. In the WC topology (CPU-bound), only 42.7% of tuples were processed when executed in a multitenant environment. For the WCF topology (IO-bound), only 33.5% of tuples were processed. Finally, for the TT topology (network-bound), only 53.1% of tuples were processed when compared to the *Baseline Cluster*.

TABLE 2. Scenario A and B comparison using ES

Topologies	**Baseline Cluster (tuples/s)**	**Multitenant Cluster (baseline related percentage)**		
		WC (CPU-bound)	**WCF (IO-Bound)**	**TT (Network-bound)**
WC (CPU-bound)	803,622	42.7%	35.8%	82.5%
WCF (IO-bound)	5,757	80.9%	33.5%	89.4%
TT (Network-bound)	1,333	80.4%	36.9%	53.1%

These results reveal the importance of scheduling methods that consider the state of physical nodes to prevent performance impacts in big data applications. Multitenant environments are a realistic proposition, especially in business-driven cloud computing (e.g., AWS, Salesforce, etc.), and drastically influence the performance of big data stream processing systems such as Apache Storm. The situation is complicated by cloud computing providers that sell Infrastructure as a Service (IaaS), but do not provide any kind of information about the status of the physical hardware running the VM in the IaaS.

Thus, during task scheduling, the scheduling algorithm must consider the physical state of the node to avoid performance degradation. Additionally, big data stream processing systems must also consider the periodic task rescheduling, as the performance of VMs may change dramatically over time because of concurrent hardware access by other cloud tenants (see Table 2, the worst case is 66.5% performance degradation in the WCF). Moreover, the resource provisioning and load balancing algorithms must also consider the multitenant property because the related increase in the number of big data processing clusters might not provide the expected increase in processing capacity, or the processing load might be forwarded to an already overloaded cluster because of multitenant interference.

5 Multitenant-Aware Mechanism

This paper proposes (i) a task scheduler for big data stream processing in cloud-based multitenant environments named Dynamic Scheduler (DySc); (ii) an SDN-based resource provisioning mechanism named Elastic Resource Provisioning (ERPr); and (iii) a load balancer for several multitenant environments named SDN-based Load Balancer (DyLB). DySc is responsible for scheduling and periodically rescheduling big data streaming tasks among nodes with available physical resources within a specific cluster.

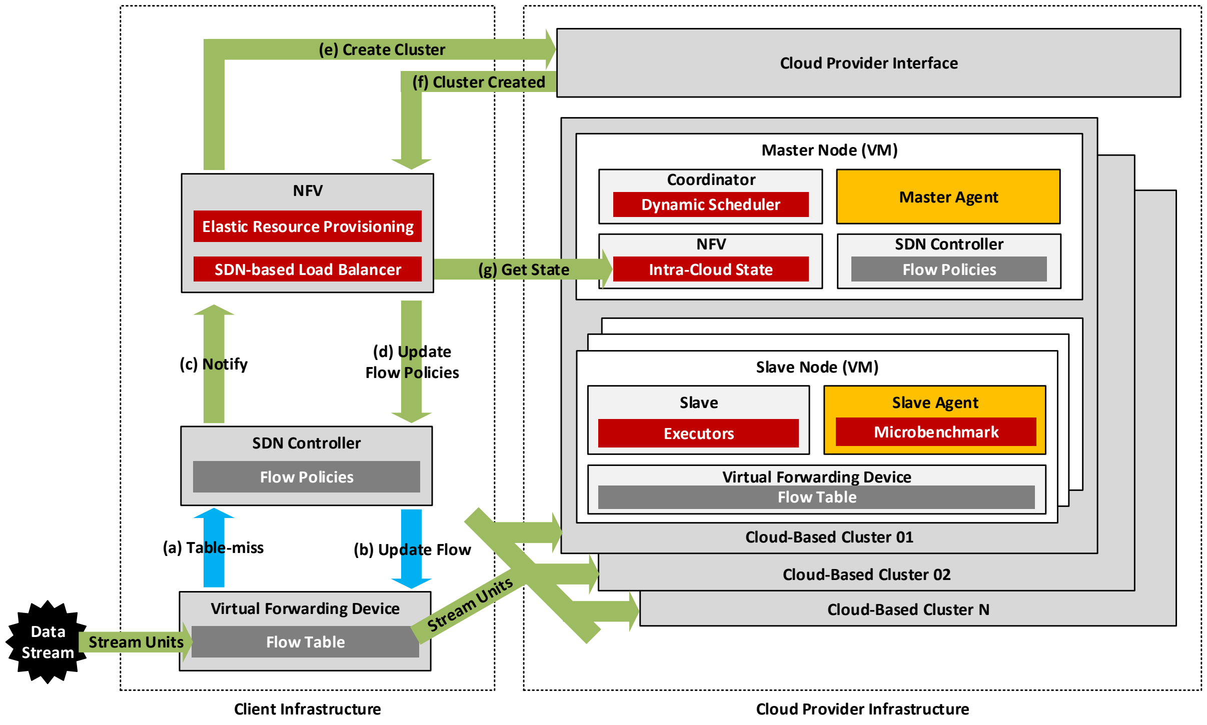


Fig. 4. Overview of the proposed architecture

ERPr is responsible for transparently detecting overloaded clusters, and then instantiating and terminating cloud-based clusters according to the processing load. Finally, DyLB is responsible for balancing the load between clusters, considering the processing and networking flow status. An overview of our proposal is shown in Figure 4, and further details are given in the following subsections.

5.1 Dynamic Scheduler

DySc relies on four main components (Figure 5) to schedule and reschedule tasks in a multitenant environment. The Dynamic Scheduler and Master Agent (MA) are responsible for monitoring the node states and scheduling and rescheduling tasks (β_x) accordingly in real time. The Slave Agent (SA) obtains the Slave Node state (using microbenchmarks) and reports it to the MA when requested. Finally, the Intra-Cloud State provides fine-grained control over the network flows. The following subsections detail each of the DySc elements.

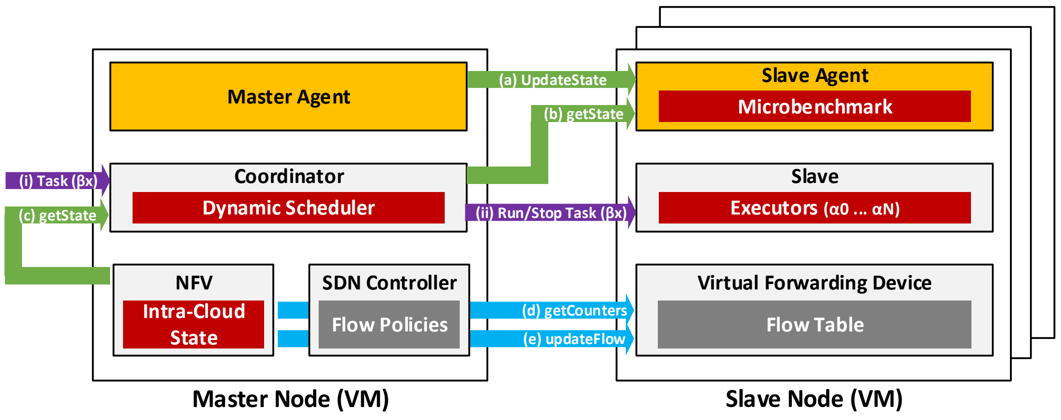


Fig. 5. Overview of dynamic scheduler architecture

5.1.1 Slave Agent

The proposed system considers a multitenant environment with shared physical resources. The SA reports the state of each node ($node_{state}$) by collecting information about the resources ($resource_{state}$). For each resource, the SA reports two sets of states: the physical state and virtual state. The physical state refers to the current condition of the shared resource, e.g., the hard disk write capacity. The virtual state refers to the current condition of an internal VM resource (virtual resource), e.g., the CPU consumption, provided by the guest OS.

A first proposal relies on microbenchmarking tools to collect information about the physical state of each resource. It is important to provide a hypervisor-independent solution and mitigate the conflicts of interest between the customer and service provider (SP), as the SP may report inconsistent states to obtain unfair advantages [47].

The process of determining the nodes' physical states is divided into two stages. Initially, the maximum throughput of each physical resource ($resource_{physical\_max}$) is obtained, establishing how a resource behaves without interferences from external entities to the VM. The $resource_{physical\_max}$ value can be obtained through values provided by the resource manufacturer, or through a benchmark tool without interference, in a controlled environment. Several metrics of VM utilization should be measured to ensure realistic values for each benchmark.

It is then possible to use microbenchmarks (specific benchmarks) to define, in real time, the current state of the physical resource of each node ($resource_{physical\_current}$). Through $resource_{physical\_max}$ and $resource_{physical\_current}$, the current state of each shared physical resource ($resource_{physical\_state}$) can be established using Eq. (1).

$$resource_{physical\_state} = \frac{resource_{physical\_current}}{resource_{physical\_max}} \quad (1)$$

Note that $resource_{physical\_state}$ can be affected by internal VM usage. For instance, the hard disk $resource_{physical\_current}$ value may have a lower read/write rate because of the VM usage itself. To consider such interference, our approach uses the internal (virtual) state of a resource ($resource_{virtual\_state}$) as a smoothing state attribute. The $resource_{virtual\_state}$ is obtained through the VM OS and defines the usage rate of a specific VM virtual resource. Finally, Eq. (2) defines the state of a resource ($resource_{state}$).

$$resource_{state} = resource_{physical\_state} + resource_{virtual\_state} \quad (2)$$

The product of all $resource_{state}$ values establishes the $node_{state}$, as indicated in Eq. (3), where n denotes the number of resources.

$$node_{state} = \prod_{i=1}^{n} resource^{i}_{state} \quad (3)$$

5.1.2 Master Agent

The MA is responsible for managing the SAs and computational demands. The MA requests that the Slave Node states are updated (Figure 5, event a). When the SA receives an update state request, it computes and stores its state value ($\text{node}_{\text{state}}$). The MA periodically requests the update node state for each cluster node (Figure 5, event b). During the scheduling and rescheduling process (Sections 5.1.3-A and 5.1.3-B), the MA requests the node states to be computed (Figure 5, event a), thus improving its response time.

5.1.3 Dynamic Scheduler Scheduling/Rescheduling Policies

DySc is responsible for scheduling and rescheduling tasks inside a cluster. The following subsections describe the scheduling and rescheduling task policies.

A) Scheduling Policy

The Scheduler algorithm considers a task to have a set of processing units (β) that must be distributed to the available Slave processing units (α).

Algorithm 1 details the DySc scheduling policy for a set of required processing units (β). DySc requests the SAs to compute the $node_{state}$ for all cluster (C) slave nodes. Slave nodes that do not have available processing units (α) are not taken into account during the scheduling. DySc computes the weight of each processing unit (weight_{β}) as the sum of all $node_{state}$ values divided by the number of tasks. The weight_{β} defines the weight of each unit β in relation to $node_{state}$. DySc then assigns the current β to a slave node with the highest $node_{state}$ ($best\_node$). Finally, the $best\_node_{state}$ is subtracted according to the prior computed weight_{β} and a parameterized *spreadability* value. The *spreadability* value defines the β distribution among the cluster slave nodes; a lower *spreadability* value is more likely to overload better slaves. In this manner, the node with the highest $node_{state}$ is not overloaded with all tasks β and the process is repeated until every unit β has been allocated to α. The proposed task scheduling algorithm assumes that every β demands node resources. The *spreadability* value prevents the best nodes from being overloaded.

Algorithm 1 Scheduling Policy

```
1: procedure DOSCHEDULE($C$, $task$, spreadability)
2:     $weight_\beta \leftarrow 0$                          ▷ Weight for each assigned $\beta$
3:     $cluster_{score} \leftarrow 0$                        ▷ Sum of all $node_{state}$
4:     $best\_node \leftarrow 0$                             ▷ Best node in cluster
5:     for all $node \in C$ do
6:         $node_{state} \leftarrow computeState(node)$
7:     end for
8:     for all $node \in C$ do
9:         if $node_\alpha = 0$ then
10:            $removeNode(C, node)$
11:        end if
12:    end for
13:    for all $node \in C$ do
14:        $cluster_{score} \leftarrow cluster_{score} + node_{state}$
15:    end for
16:    $weight_\beta \leftarrow cluster_{score}/task_\beta$
17:    for all $\beta \in task$ do
18:        $best\_node \leftarrow max\_node\_state(C)$
19:        assign_task($\beta$,best_node)
20:        $best\_node_{state} = best\_node_{state}$ - ($weight_\beta$ * spreadability)
21:    end for
22: end procedure
```

B) Rescheduling Policy

DySc is also responsible for rescheduling big data tasks through cluster monitoring and real-time redistribution of previously scheduled processing units (β). After the scheduling stage, rescheduling occurs periodically to optimize task performance by moving β from nodes that have poor $node_{state}$ values to those with better ones. This task reallocation aims to minimize the multitenant interference effect in the processing of streaming data. Interference can occur after the scheduling a task when another VM is instantiated alongside an existing slave node and starts to use the shared resources excessively.

The rescheduling algorithm (Algorithm 2) considers tasks running in the cluster, meaning that all β are already assigned to an α in a slave node. Thus, DySc requests the SAs to compute the $node_{state}$ for the slave nodes in all clusters (C). DySc then recomputes weight$_\beta$ for a running task. All $node_{state}$ values are then updated according to the number of units β previously assigned to the node. DySc selects two nodes: the node (*best_node*) with available α and the best state ($best\_node_{state}$), and the node (*worst_node*) with a β assigned and the worst state ($worst\_node_{state}$). If the $best\_node_{state}$ is better than the $worst\_node_{state}$, adjusted by a parameterized *meaningfulness* value, β is reallocated. The *meaningfulness* value prevents unnecessary rescheduling when the difference between the available resources in the nodes is not significant. Moreover, in the case of reallocation, the states of both nodes are updated. The rescheduling process is repeated until there are no available better nodes to reallocate β units.

Algorithm 2 Rescheduling Policy

```
1: procedure DORESCHEDULE($C$, $task$, spreadability, meaningfulness)
2:     $weight_\beta \leftarrow 0$                         ▷ Weight for each assigned $\beta$
3:     $best\_node \leftarrow 0$                           ▷ Best node in cluster
4:     $worst\_node \leftarrow 0$                          ▷ Worst node in cluster
5:     $cluster\_score \leftarrow 0$                       ▷ Sum of all $node_{state}$
6:     $rescheduling \leftarrow true$                      ▷ Reschedule verifier
7:     for all $node \in C$ do
8:         $node_{state} \leftarrow computeState(node)$
9:     end for
10:    for all $node \in C$ do
11:        $cluster_{score} \leftarrow cluster_{score} + node_{state}$
12:    end for
13:    $weight_\beta \leftarrow cluster_{score}/task_\beta$
14:    for all $node \in C$ do
15:        $node_{state} \leftarrow updateState(node, weigth_b)$
16:    end for
17:    while $rescheduling$ do
18:        best_node = max_node_state_with_avaible_$\alpha$ (C)
19:        worst_node = min_node_state_with_$\beta$(C)
20:        if (($best\_node_{state}$ * meaningfulness) > $worst\_node_{state}$)
21:            $\beta$ = getFirst$\beta$(worst_node)
22:            deassign_task($\beta$, worst_node)
23:            assign_task($\beta$, best_node)
24:            $best\_node_{state} = best\_node_{state}$ - ($weigth_b$ * spreadability)
25:            $worst\_node_{state} = worst\_node_{state}$ + ($weigth_b$ * spreadability)
26:        else
27:            rescheduling = false
28:    end while
29: end procedure
```

5.1.4 Intra-Cloud State NFV

The Intra-Cloud State NFV acts as the cloud client's infrastructure manager. Thus, the NFV is responsible for setting the intra-cloud (cloud client's infrastructure domain) network flows to the startup level (Figure 5, event e) through the SDN controller. The NFV also acts as the intra-cloud state provider, allowing the SDN-based Load Balancer (Section 5.2) and Elastic Resource Provisioning (Section 5.2.1) to determine the cloud client's infrastructure state. To this end, the NFV periodically requests the DySc to update the Slave Node states (Figure 5, event c) and, through the SDN controller, the flow metrics among the Slave Nodes (Figure 5, event d).

5.2 SDN-based Load Balancer

DyLB transparently provides elasticity to cloud-based processing infrastructure to improve task processing. The proposed system considers that a specific task processing cluster may, temporally or not, become unable to process a designated load (Data Stream that generates several Stream Units, Figure 4). This may be due to an increase in the processing load, implying processing load spikes [19], or a significant loss of cluster processing power, e.g., caused by multitenant interference.

DyLB relies on the Intra-Cloud State NFV to identify exhausted clusters (Figure 4, event g), and periodically requests the Intra-Cloud State NFV to determine the state of each cluster ($cluster_{state}$). Eq. (4) defines the state of a cluster as the average of each $node_{state}$ within that cluster, where N denotes the number of nodes within a cloud-based cluster.

$$cluster_{state} = \frac{\sum_N^i node_{state}^i}{N} \qquad (4)$$

From the $cluster_{state}$ value of each cluster, DyLB provides elasticity for resource allocation and load balancing. The following subsections describe the elastic resource provisioning and load balancing methods.

5.2.1 Elastic Resource Provisioning

Our proposed system performs horizontal scaling, in which the cluster is replicated, rather than vertical scaling by adding more Slave Nodes (Figure 5, Slave Nodes) to the cluster. In this way, it provides elastic resources without any knowledge of the application demand. Thus, ERPr relies on the $cluster_{state}$ values to identify the $infrastructure_{state}$. Eq. (5) defines the state of the infrastructure as the average of each $cluster_{state}$ within the infrastructure, where *N* denotes the number of clusters in the infrastructure.

$$infrastructure_{state} = \frac{\sum_N^i cluster_{state}^i}{N} \quad (5)$$

ERPr creates or terminates clusters according to the $infrastructure_{state}$. Therefore, it uses two parameterized values, $infrastructure_{lower\ threshold}$ and $infrastructure_{upper\ threshold}$. The former defines the minimum acceptable $infrastructure_{state}$. When the $infrastructure_{state}$ is less than the $infrastructure_{lower\ threshold}$, ERPr requests the cloud platform to create another cluster (Figure 4, event e). In contrast, when the $infrastructure_{state}$ is greater than the $infrastructure_{upper\ threshold}$, ERPr requests the cloud platform to terminate the cluster with the lowest $cluster_{state}$ (Figure 4, event e).

ERPr is periodically executed according to the sum of two parameters: *T* and $cluster_{creation\ time}$. *T* is a parameter that defines the frequency of cluster state updates, whereas $cluster_{creation\ time}$ is the time required by the cloud provider platform (Figure 4, event f) to create the cluster. Using both parameters, the elastic resource provisioning function waits for the cluster creation time (Figure 4, event f) before checking the $infrastructure_{state}$ again.

Note that our proposal assumes that the present load is not a spike—a type of load in which the time required to provide more resources ($cluster_{creation\ time}$) is greater than the time required to process the load [19].

5.2.2 SDN-based Load Balancing

ERPr increases the infrastructure process capacity by creating several clusters to perform the processing and flow tasks. However, the data stream unit generator (Figure 4, Data Stream) cannot define which cluster the Stream Units should be sent to. To perform such load balancing transparently, our proposal relies on the SDN model.

Our load balancing scheme establishes the cluster load capacity ($cluster_{load\ capacity}$) according to each $cluster_{state}$. Each $cluster_{load\ capacity}$ is periodically established through Eq. (6), where *N* denotes the number of clusters in the infrastructure.

$$cluster_{load\ capacity} = \frac{cluster_{state}}{\sum_N^i cluster_{state}^i} \quad (6)$$

The load balancing function performs its designated function at the granularity of the data stream units. During a period, each cluster receives a load rate per $cluster_{load\ capacity}$. For instance, a Storm cluster with $cluster_{load\ capacity} = 0.7$ should receive 70% of the data stream units (i.e., Storm tuples). The $cluster_{load\ capacity}$ values are updated according to a predefined time window.

An overview of the load balancing process is shown in Figure 4. The Data Stream periodically generates Stream Units and sends them to known-clusters. As no flow entry is specified for the Stream Unit according to its identifiers (e.g., source IP and Port), a table miss occurs. Thus, the SDN Switch transfers the first stream data unit packet to the SDN Controller (Figure 4, event a). The SDN Controller notifies the DyLB NFV (Figure 4, event c) to define the target cluster according to the $cluster_{load\ capacity}$ of each cluster. Then, the DyLB NFV updates the SDN Controller flow policies accordingly (Figure 4, event d). Next, the SDN Controller creates a flow entry for the chosen cluster and Stream Unit identifiers (Figure 4, event b). In this manner, further packets from the same stream unit are forwarded to the chosen cluster.

6 PROTOTYPE

This section describes a prototype of the proposed system, which is split into (i) ERPr and DyLB, and (ii) DySc prototypes. All tools, source codes, and binaries are publicly available in [58].

6.1 ERPr and DyLB Prototype

Figure 6 shows the ERPr and DyLB prototype. To make the prototype evaluation more realistic, a Data Stream generator (Figure 6, Data Stream) was developed that periodically generates tuple requests to a topology. In this manner, the load is generated outside the topology as it occurs in real-world applications. Floodlight [41] was used in the prototype as the SDN controller (version 1.2). A PACKET-IN module was implemented at the controller. This module receives PACKET-IN (when a packet has no matching flow policies) messages from the OpenVSwitch at each new tuple arrival (Figure 6, Data Stream Units), identified through the source IP and port addresses. The module requests the DyLB and waits for a response as to which cluster the tuple generation request should be forwarded to.

HPE Helion Eucalyptus (version 4.3.0) was used as the cloud provider. Each Eucalyptus cloud computing environment is composed of four physical computers equipped with an 8-core Intel i7 CPU and 16 GB RAM, connected through a gigabit NIC. The cloud client infrastructure (Figure 4, Cloud-Based Cluster) was created through a CloudFormation [33] template. The template creates five VMs (one *Master Node* and four *Worker Nodes*), and these dynamically execute the Apache Storm topology at startup through the cloud-init template field. The ERPr creates or terminates a cluster through the Eucalyptus CloudFormation template (Figure 6, event e).

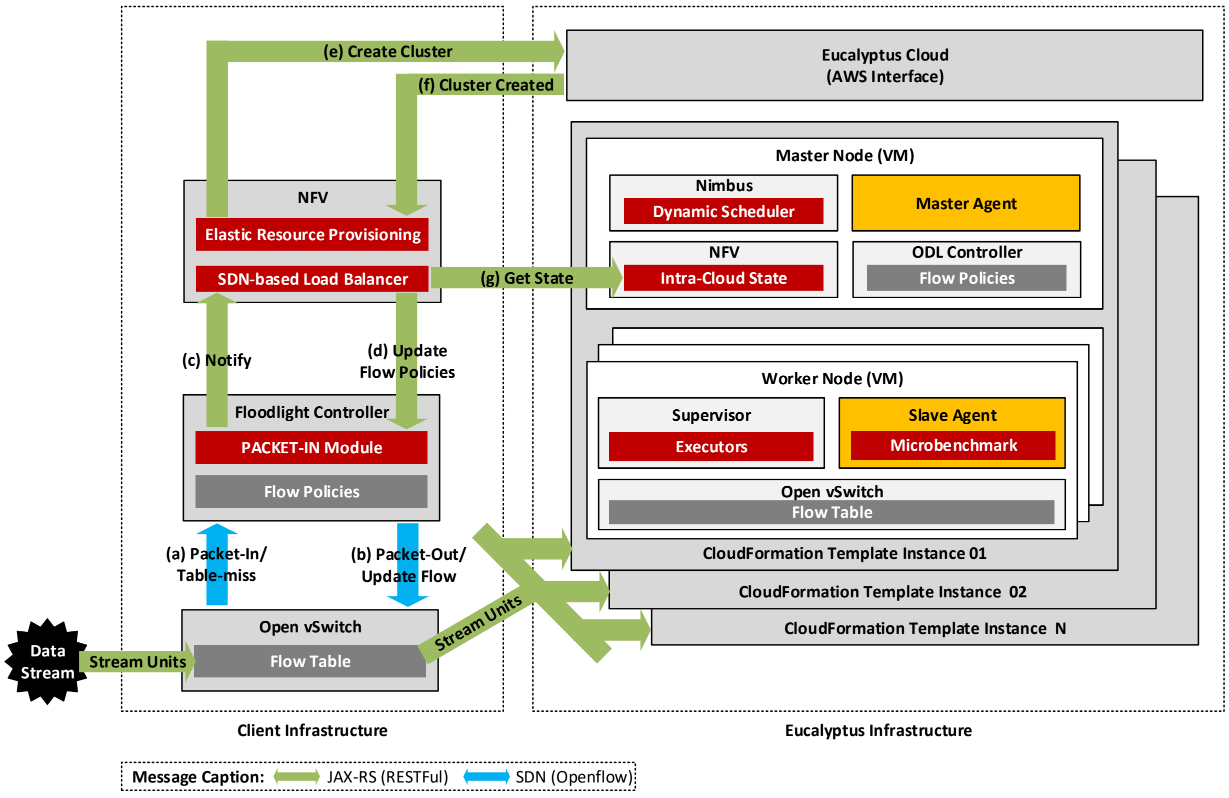


Fig. 6. ERPr and DyLB prototype implementation architecture

The DyLB periodically requests the Intra-Cloud State NFV for each CloudFormation instance at 60-s intervals. The ERPr also requests the DyLB to update the $cluster_{state}$ every 60 s. The DyLB receives the SDN Controller PACKET-IN notifications, which are called through a PACKET-IN module implemented in the Floodlight Controller. The DyLB load balancing scheme is defined at the tuple level according to the source IP and port addresses. The ERPr requests the Eucalyptus cloud for the creation or ending of clusters through the Eucalyptus client API. The cluster is defined through a Eucalyptus CloudFormation template.

6.2 Dynamic Scheduler Prototype

This section describes the DySc prototype executed within the CloudFormation template. The whole infrastructure is dynamically configured through the *cloud-init* field from the CloudFormation template, enabling the ERPr evaluation. Figure 7 shows the DySc prototype. A virtual OpenVSwitch is executed within each *Worker Node* VM, enabling fine-grained flow measurements (evaluated in Section 7.4) within the cloud. The flow setup is performed at CloudFormation template startup through the Floodlight Controller REST Interface. The flow counters are also obtained through the Floodlight Controller REST Interface. The counters are invoked every 60 s.

As mentioned in Section 2.2, Apache Storm allows the scheduling policy to be customized, enabling the creation of new scheduling policies according to the needs of the user/environment. The API that provides such customization is called the *IScheduler*, which provides a scheduling method to implement a customized scheduler.

The MA runs with the Storm *Master Node*. On the other side, the SAs run in each Storm *Worker Node*. The SAs were implemented as a RESTful web service through the JAX-RS API [48]. In this way, the SAs have two main methods: *updateState* (Figure 7, event a) responsible for starting the node state compute process, and *getState* (Figure 7, event b) which returns the last computed state. DySc starts or interrupts a task (a set of β processing units) through the functions *assign* and *freeSlot* [49].

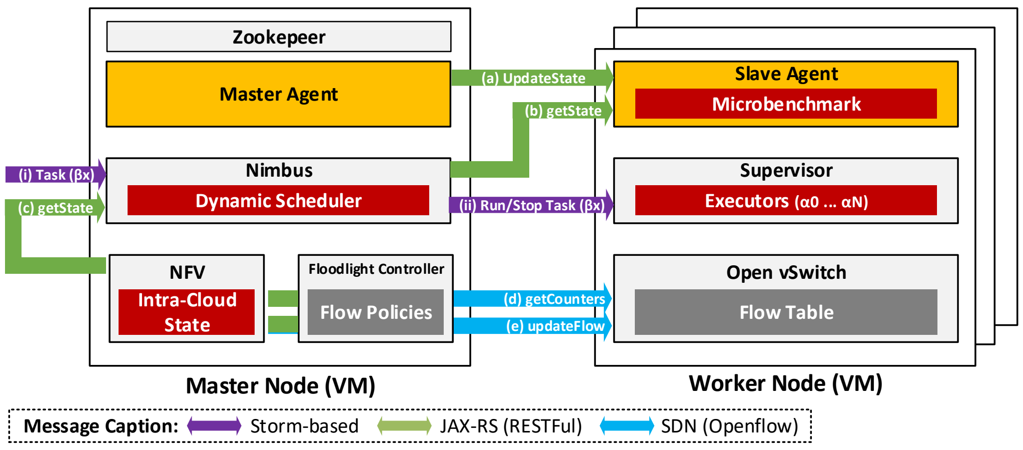


Fig. 7. DySc prototype implementation architecture

The following relations were applied between the notation from DySc and Storm (Figure 7):

- *Cluster*: set of Storm worker nodes
- *Node/Slave*: Storm worker nodes
- *Task*: Storm topology
- B: topology worker
- α: slot in a worker node

We used microbenchmarks to compute the node states in terms of the CPU, hard disk, and network. These resources were shared in a multitenant environment, and thereby suffered degradation of processing/throughput capacity when overloaded by another cloud client, as discussed in Section 4. Table 3 describes the tools and methods used to obtain the shared resource state (resource $_{\text{shared state}}$).

To ensure that the physical resource state measurements are made in real time, the MA relies on a cyclic algorithm to update the $node_{\ state}$ (Figure 7, event a). MA waits for a predetermined time before requesting the next node update, preventing two *Worker Nodes* allocated in the same infrastructure from performing simultaneous microbenchmarks, and thus degrading their state wrongly or having the results influenced by data caching.

The $resource_{virtual\_state}$ of the CPU, hard disk, and network is obtained from the Linux OS using the *proc* filesystem [52].

TABLE 3. Resource state computation methods

Resource	**Process to obtain $resource_{physical\_state}$**
CPU	Sysbench [50] performs a set of prime number computations. The average processing time with free resources was previously established ($resource_{shared\_max}$) and the real-time $resource_{physical\_state}$ was computed as $resource_{shared\ state} = 1 - \frac{resource_{physical\_current} - resource_{physical\_max}}{resource_{physical\_max}}$
Hard Disk	Sysbench [50] also performs reads and writes of a predetermined set of blocks. The average write time with free resources was established (resource$_{physical\_max}$) and the real-time $resource_{physical\_state}$ was computed as $resource_{physical\_state} = \frac{resource_{physical\_current}}{resource_{physical\_max}}$
Network	The Master Node hosts an iPerf [51] server. This measures the maximum available throughput for each Slave Node, which in turn has an iPerf client. The average connection throughput was previously established ($resource_{physical\_max}$) and the real-time $resource_{physical\_state}$ was computed as $resource_{physical\_state} = \frac{resource_{physical\_current}}{resource_{physical\_max}}$

These tools allow the microbenchmark execution time to be parameterized. This concern justifies itself, as this study does not aim to measure the total physical resource computation time through long-duration benchmarks. What is intended is to perform microbenchmarking to obtain real-time results and adequately use the available resources in a multitenant environment.

7 EVALUATION

This section describes the DySc, DyLB, and ERPr evaluation tests.

7.1 Dynamic Scheduler Evaluation

The performance of DySc was compared to that of the Storm default scheduler, EvenScheduler (ES), which employs a round-robin-based scheduling policy. The goal was to measure the benefits and overhead of DySc in a multitenant environment. The experiments were performed in the testbed scenario (Figure 3, Section 4).

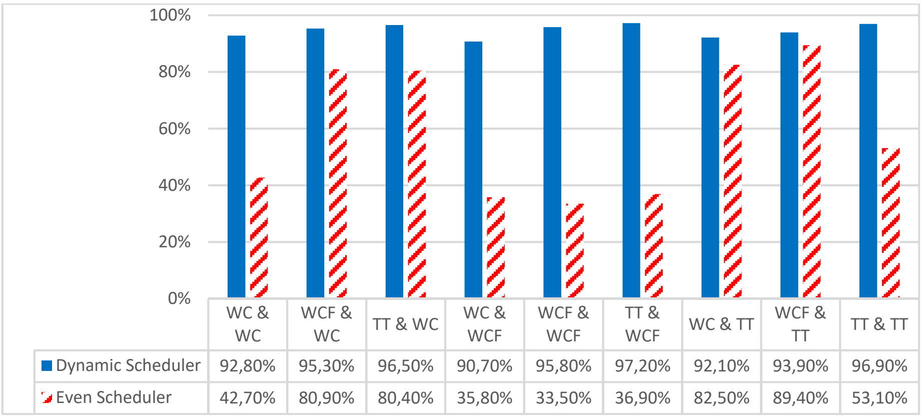
The same topologies used to evaluate ES in Section 4 were used. As presented in Table 2, all evaluated topologies (WC, WCF, and TT) were affected by the multitenant environment because ES does not consider the node state. It is expected that the use of DySc will improve the performance of these topologies.

DySc requires two parameters, as discussed in Section 5. For the *spreadability,* we used the value 1.0 to increase the distribution between nodes with the best states. For the *meaningfulness*, we used the value 0.5, which was established through rescheduling behavior analysis tests. DySc executes the rescheduling algorithm periodically (every 60 s) to allow the SAs to update their states. Each SA is given 15 s to update its state. After this period, the MA requests the next node to update its state, and so on. When all node states have been updated, the MA waits another 15 s before restarting the update process.

7.1.1 Scheduler Evaluation

This subsection presents the results obtained from running DySc and ES in a multitenant environment. The measurements compare the total number of tuples generated by DySc and ES in the *Baseline Cluster* (Figure 3), in which there is no interference from a third-party VM.

Figure 8 compares the performance of the topologies. The vertical axis represents the maximum relative percentage of tuples processed compared with the baseline cluster (Figure 3). The horizontal axis shows the topology pairs, wherein the first topology is executed in the main cluster and the second topology in the secondary cluster, generating parallel processing loads to mimic multitenant interference.



	WC & WC	WCF & WC	TT & WC	WC & WCF	WCF & WCF	TT & WCF	WC & TT	WCF & TT	TT & TT
■ Dynamic Scheduler	92,80%	95,30%	96,50%	90,70%	95,80%	97,20%	92,10%	93,90%	96,90%
▨ Even Scheduler	42,70%	80,90%	80,40%	35,80%	33,50%	36,90%	82,50%	89,40%	53,10%

Fig. 8. DySc and ES processed tuples ratio comparison

When the topologies are "mirrored" in clusters (same resources are overloaded), our proposal can provide an improvement of 50.10% when the WC is executed in parallel on both the main and secondary (mirrored) cluster (WC & WC in Figure 8). DySc was 62.30% better with parallel WCF executions (WCF & WCF in Figure 8) and 43.80% better when executing parallel TT (TT & TT in Figure 8). When the main topology resource is available, but the other resource is experiencing multitenant interference, DySc can also improve the performance to that of ES; for instance, in CPU-bound topology while executing IO-bound topology in a secondary cluster. However, ES has significantly poorer performance when the same topology is present in both clusters—as both processing/topologies require the same computational resources. One important thing to note is as long as the disk resource is being used by other cloud tenants, the topology performance significantly decreases, processing only an average of 35.40%, while the DySc can process an average of 94.57% compared to the baseline cluster.

Figure 9 shows the average processing time for each tuple of the WC topology when using DySc and ES. DySc processed a tuple in an average of 0.0014 ms, whereas ES required approximately 0.0022 ms. Thus, DySc improved the processing of each tuple by 57%.

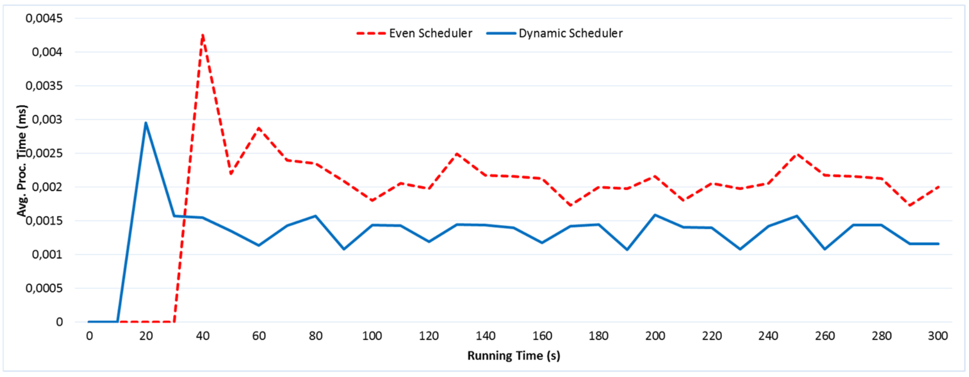


Fig. 9. WC tuples processing time by DySc and ES

Figure 10 shows the average processing time for each WCF tuple. DySc obtained an improvement of 172% compared with ES (average of 0.18 ms against 0.49 ms).

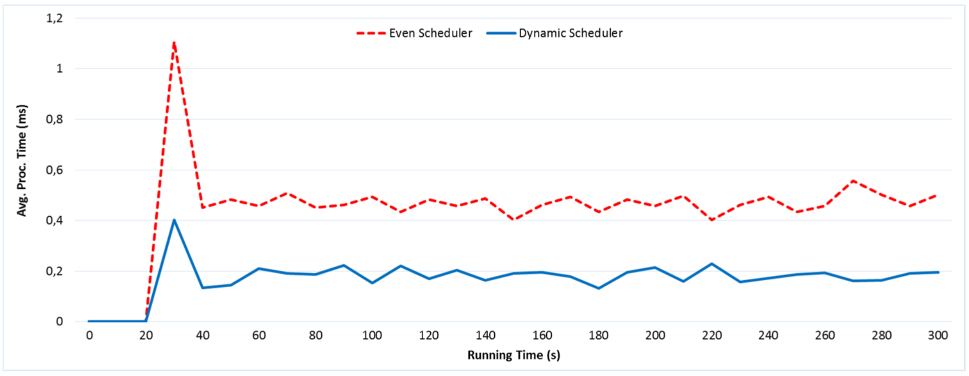


Fig. 10. WCF tuples processing time by DySc and ES

Figure 11 shows the average processing time for each TT tuple. DySc again produced better results than ES. The average processing time for each tuple was 0.78 ms for DySc and 1.52 ms for ES, representing an average gain of 94.87%.

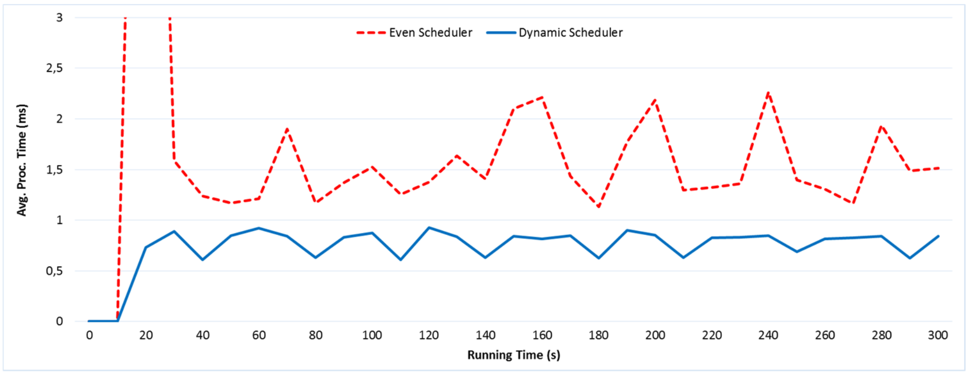


Fig. 11. TT tuples processing time by DySc and ES

Another important behavior is that the results generated by ES exhibited greater variation than those of DySc. This reflects the absence of scheduling policies evaluating the physical state of a node before sending a request. Big data streaming frameworks generate large amounts of data intermittently, and it is expected that the processing environment will not present a highly variable processing time.

7.1.2 Rescheduler Evaluation

Multitenant environments are characterized by highly variable usage of resources over time. A controlled environment was created, and the processing loads were varied to evaluate the DySc behavior during the rescheduling procedure. The objective was to evaluate the time required by DySc to identify the resource degradation, change the processing among cluster nodes, and redistribute tasks after the resources became fully available again.

The controlled environment consists of a rescheduling scenario that alternated between multitenant and single processing, i.e., alternating between the *Baseline Cluster* and *Multitenant Cluster* in Figure 3. The executions were divided across three distinct periods: (i) machine resources are fully available (*Baseline Cluster*); (ii) parallel processing has started in half of the physical machines, overloading its computational resources (*Multitenant Cluster*, Secondary Topology has started, Figure 3); and (iii) parallel processing has ended, returning to the original resource availability state (*Baseline Cluster*). Each period was executed for 20 min.

To measure the impacts and benefits of DySc rescheduling, we used the network-bound topology (TT) for both single and parallel processing (*Baseline Cluster* and *Multitenant Cluster*). Figure 12 shows the average tuple processing time of DySc and ES running the TT topology in the rescheduling scenario. The parallel processing started 1200 s later. DySc required 60 s to identify the processing change and perform the rescheduling. After 2400 s, parallel processing ended and DySc required 90 s to identify the change and redistribute the tasks for all cluster machines. This rescheduling time can be improved if the nodes updated their states more frequently (15 s during the experiments) or the rescheduling process was conducted less frequently (we considered 60 s). The total time required for rescheduling (identify and redistribute) in both changes of processing corresponded to only 4% of the total execution time.

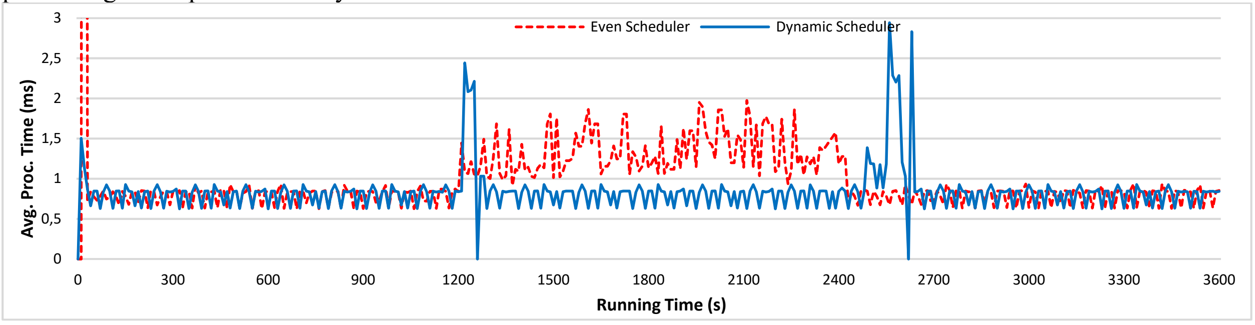


Fig. 12. DySc and ES comparison considering rescheduling in alternate multitenant interference

7.2 SDN-based Load Balancer Evaluation

To evaluate the DyLB, two HPE Eucalyptus Clouds were used, each with four physical machines. One cloud had all of its physical machine resources fully available (Figure 3, *Baseline Cluster*), whereas the other cloud had half of its physical machines running a cluster from another client (Figure 3, *Multitenant Cluster*).

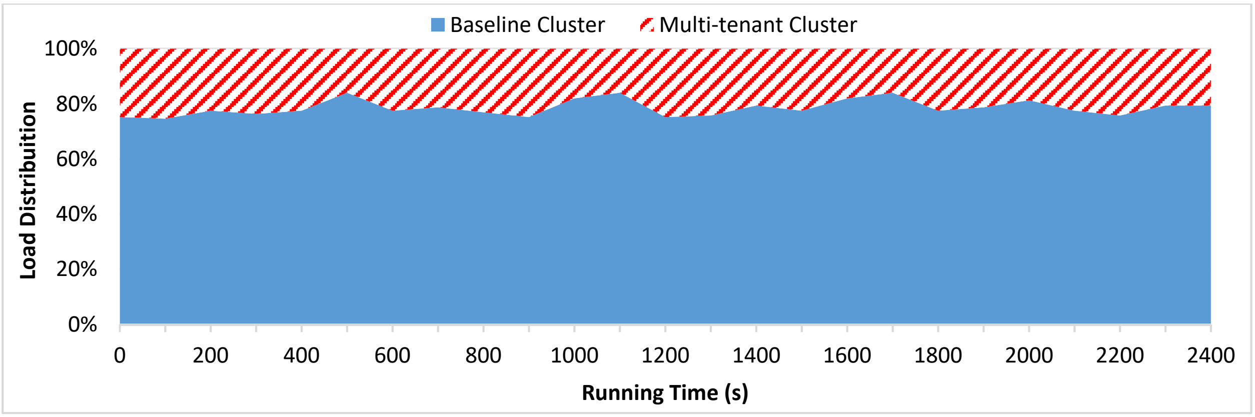


Fig. 13. DyLB load distribution among clusters

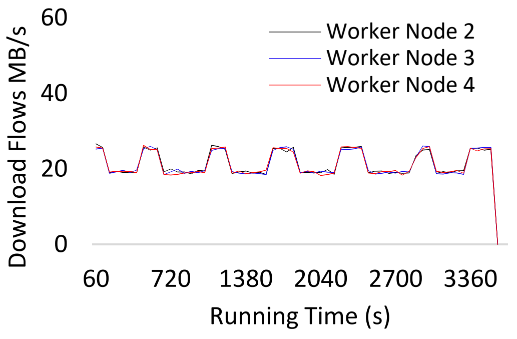
The CPU-dependent topology (WC) was executed on both clusters, and tuple processing requests were generated intermittently (Figure 6, Stream Units). For each tuple processing request, a Table-miss and Notify occurred (Figure 4), generating a request to the DyLB to establish which cluster the tuple processing request should be sent to. Eq. (6) was used to compute the load for each cluster. A 60-s interval was used to update the $cluster_{state}$ values. The load distribution among both clusters is shown in Figure 13.

As a general evaluation, it can be noted that the proposed load balancing approach distributed the load properly, because only 27.96% of the processing load (on average) was forwarded to the *Multitenant Cluster*. The *Baseline Cluster*, which did not experience multitenant interference, had more available resources and received 72.04% of the processing load (on average) during the evaluation. Thus, it is possible to conclude that the proposed DyLB is 22.04% (on average) more effective than the traditional round-robin approach (which has a distribution effectiveness of 50%).

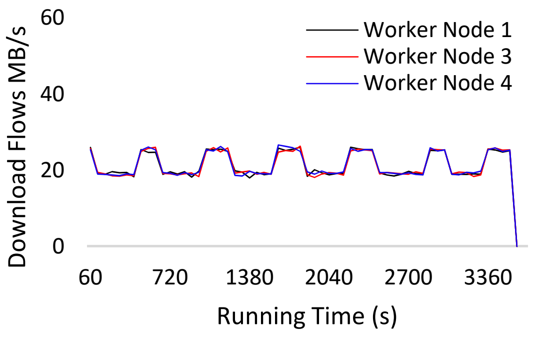
7.3 Fine-grained SDN Flow Counters Evaluation

Conducting microbenchmarking tests to identify multitenant interference in public clouds has two main drawbacks: (*i*) the performance degradation caused by microbenchmark processing; and (*ii*) the waste of resources in identifying such resource degradation, possibly implying the unnecessary use of resources. To further minimize such microbenchmarking impacts, Worker Node SDN flow counters were considered, as these represent a fine-grained point-to-point (Worker-to-Worker) bandwidth. The hypothesis is that, given the homogeneous distributed processing nature of big data processing frameworks, the bandwidth among the processing nodes must also be homogeneous, as the processing loads are equally distributed in most cases. Thus, multitenant interference can be identified through fine-grained flow counter analysis, eliminating the need to perform microbenchmarking during the scheduling process.

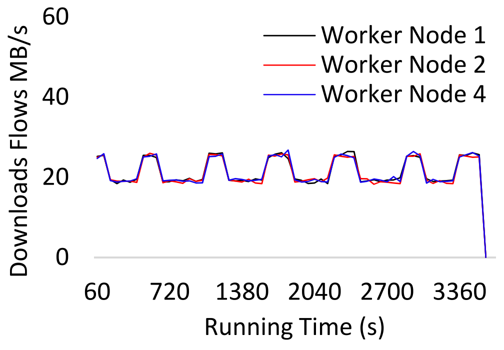
The Worker-to-Worker flow counters available in the Floodlight Controller were measured to verify the hypothesis. Two scenarios were evaluated: the *Baseline Cluster* and *Multitenant Cluster* (Figure 3). The evaluation tests were performed for 3600 s in both scenarios. Figures 14 and 15 show the download rates amongst the Worker Nodes while executing the TT topology in the *Baseline Cluster* and *Multitenant Cluster*.



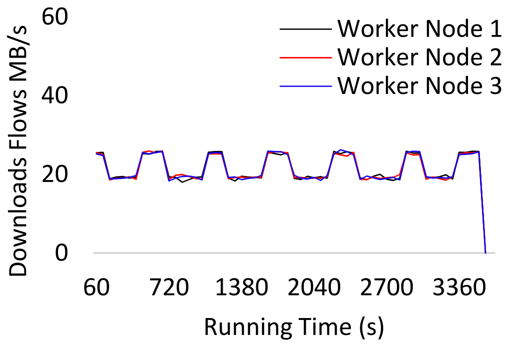
(a) Worker Node 1



(b) Worker Node 2



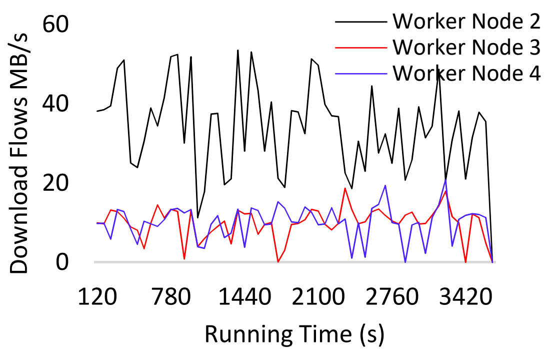
(c) Worker Node 3



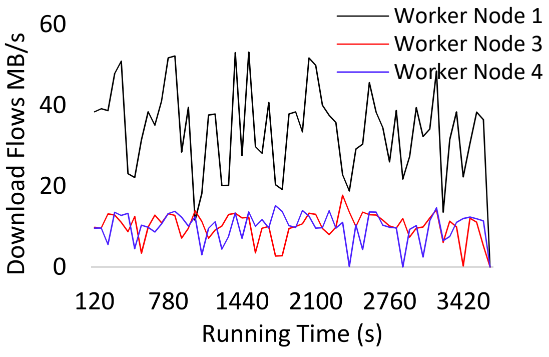
(d) Worker Node 4

Fig. 14. Download rates among Worker Nodes while executing in the Baseline Cluster.

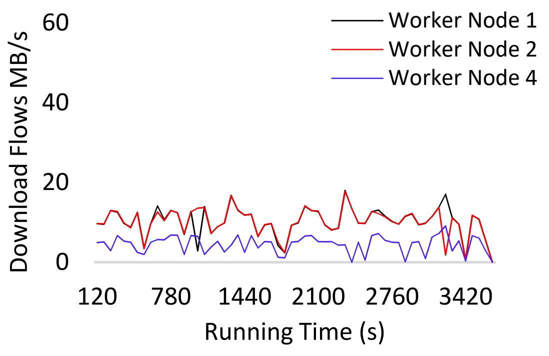
Note that the network flows between the Worker Nodes varied significantly when executing in the multitenant scenario. The download rates between multitenant Worker Nodes 01 and 02 were significantly higher compared to their counterparts in the *Baseline Cluster*. This indicates nodes that were overloaded in the multitenant scenario, as the resources of the other Worker Nodes (03 and 04) were degraded. The download rates of the multitenant Worker Nodes in the *Multitenant Cluster* (Worker Nodes 03 and 04) were significantly lower than those of Nodes 01 and 02. Finally, in the *Baseline Cluster*, the download rates between the Worker Nodes did not show such significant differences. Figure 16 shows the average download rates between the Worker Nodes in the *Multitenant* and *Baseline Clusters*.



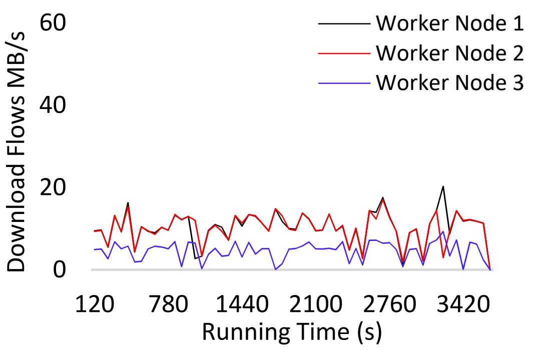
(a) Worker Node 1



(b) Worker Node 2



(c) Worker Node 3



(d) Worker Node 4

Fig. 15. Download rates among Worker Nodes while executing in the Multitenant Cluster

The blue lines show the download rates between Worker Nodes 01 and 02 in the *Multitenant Cluster*, which averaged 35.72 MB/s and 35.43 MB/s, respectively. In the *Baseline Cluster*, the average download rates between the Worker Nodes remained almost equal, showing at most a difference of 0.6 MB/s. Thus, it is possible to conclude that the fine-grained download flow analysis between the Worker Nodes allows us to establish whether a VM is suffering from multitenant interference. Such results eliminate the need to perform microbenchmarking to identify multitenant interference, reducing the waste of processing resources and performance degradation.

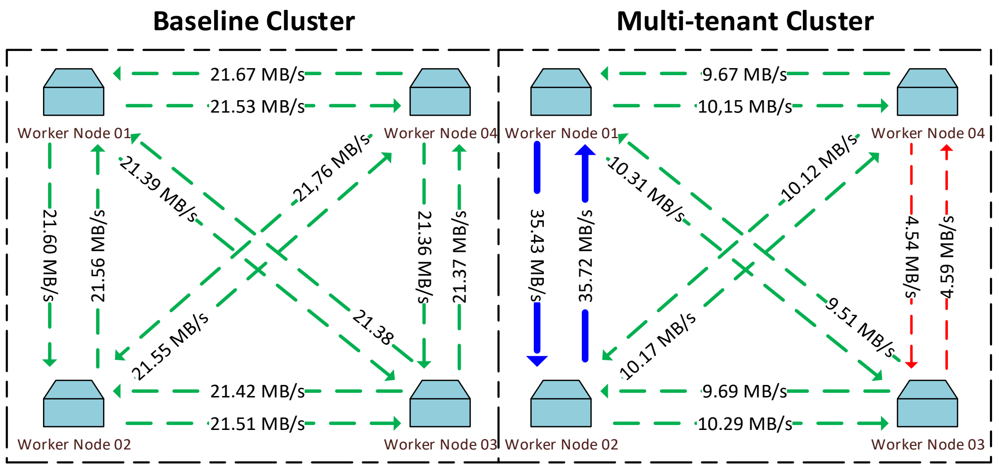


Fig. 16. Average download rates between the Worker Nodes (VMs) in the Multitenant Cluster and Baseline Cluster

7.4 Elastic Resource Provisioning Evaluation

Finally, DySc, DyLB, and ERPr were evaluated. For this purpose, two Eucalyptus clouds were deployed, in which each cloud had four physical machines (Figure 3). The testbed scenario ran for 9000 s. In the first 3000 s (0–3000 s) the first cloud did not experience multitenant interference. However, after 3000 s, the first cloud started to experience multitenant interferences (competing WC topologies ran on all physical machines), which also lasted for 3000 s (3000–6000 s). Finally, multitenant interference in the first Eucalyptus cloud terminated and the remaining test ran free of multitenant interference (6000–9000 s).

A single cluster was deployed in the first Eucalyptus cloud, which executed the WC topology, while the second Eucalyptus cloud was available for the ERPr for the allocation of further clusters through a CloudFormation template. The $infrastructure_{lower\ threshold}$ of 0.3 and $infrastructure_{upper\ threshold}$ of 0.9 were defined, since these provided the best cluster creation and ended the tradeoff through evaluation tests. Figure 17 shows the evaluation results for the load distribution performed by the DyLB, the average processing time through the DySc (for all allocated clusters), and the ERPr delay for the identification of the multitenant interference beginning and the end.

The average processing time, per tuple, during the whole testbed execution time did not significantly change. At 3000 s, when the First Cluster started to experience multitenant interference, the ERPr was able to identify such interference through the $infrastructure_{state}$ value and it requested the creation of another cluster in the second Eucalyptus cloud. A delay of 107 s was observed up to the creation of the second cluster owing to the time demanded by the second Eucalyptus cloud to instantiate the CloudFormation template. When the second cluster was created, the DyLB was able to properly balance the incoming load in a manner that the second cluster received an average 78% of the incoming load. During multitenant interference in the first cluster, the average processing time remained without multitenant interference. Finally, at 6000 s, the interference ended and the ERPr required 225 s to end the second cluster through the $infrastructure_{upper\ threshold}$ values.

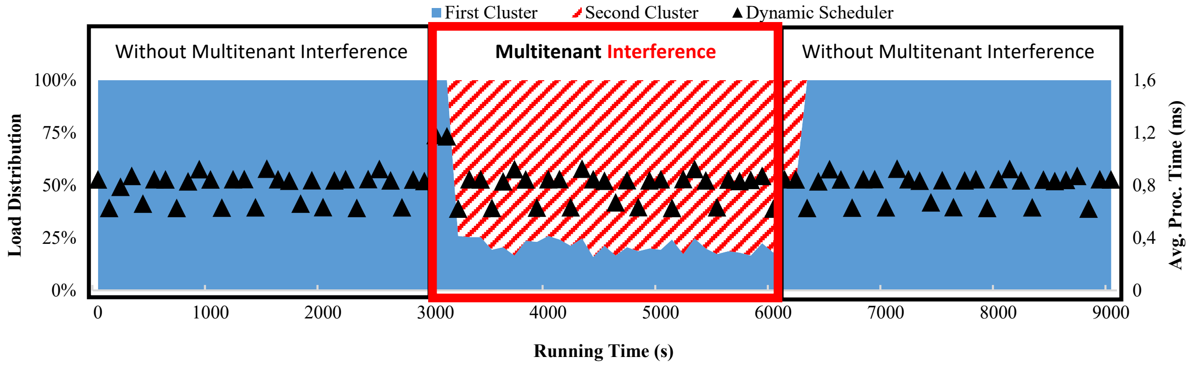


Fig. 17. Multitenant interference: highlighted (by red border) in the picture.

8 CONCLUSION AND FUTURE WORK

This study evaluated the impact of a multitenant cloud on the big data stream processing framework, Apache Storm. We revealed that scheduling computational demands without considering the physical state of the nodes (VM) that performed the processing, as done in previous methods, is an inefficient practice because the nodes resources may have been exhausted by other cloud customers (tenants).

The Dynamic Scheduler (DySc) was developed to deal with the cloud computing multitenant environment. This system continuously evaluated the state of available physical resources in the cloud computing environment through microbenchmarks. These measurements were linked to scheduling and rescheduling policies, making it possible to schedule computational demands in a manner that avoids nodes with exhausted resources. Experimental evaluations performed with DySc and comparison with ES show that it is possible to improve application performance using the same computational resources by 50.1%, 62.30 %, and 43.8 % for the CPU, hard disk, and network, respectively, in the *Multitenant Cluster* scenario where multitenancy caused its biggest impact.

Processing big data streams is extremely variable, even with an efficient task scheduling policy such as the proposed DySc, which considers both virtual and physical resource availability, as significant increases in processing demand might occur. Thus, a resource provisioning strategy is needed that considers the multitenant cloud properties. To this end, an Elastic Resource Provisioning and SDN-based Load Balancing approach was proposed. Using the SDN flow counters, a fine-grained flow analysis allowed us to determine whether a VM was suffering from multitenant interference. Such insight eliminates the need to perform microbenchmarking tests to identify multitenant interference, reducing the waste of resources and performance degradation. The proposed system was implemented through the NFV and used to balance the processing and flow loads through multitenant cloud-based clusters. It sent 72.04 % of the load to a fully resource-available cluster and yielded a 22.04% gain over the traditional round-robin algorithm.

As future work we consider that security aspects should be addressed, given that a dynamic environment as this one can be target of Distributed Denial of Service or vulnerabilities exploitation, for instance. In such a case, the proposal might wrongly consider a processing resulting from security weakness as system load.

ACKNOWLEDGMENT

This work was partially sponsored by the Brazilian National Council for Scientific and Technological Development (CNPq), grants 310671/2012-4 and 404963/2013-7. Cleverton Vicentini wishes to thank the Coordination for the Improvement of Higher Level Personnel (CAPES) for the scholarship, grant no. 140/2013.

REFERENCES

[1] NIST. Big Data Interoperability Framework: Volume 1, Definitions, NIST Special Publication 1500-1, 2015, pp. 1-32, doi:10.6028/NIST.SP.1500-1

[2] Apache Hadoop. [Online]. Available: http://hadoop.apache.org/docs/current [Accessed: September 2017]

[3] A. Toshniwal, J. Donham, N. Bhagat, S. Mittal, D. Ryaboy, S. Taneja, A. Shukla, K. Ramasamy, J. M. Patel, S. Kulkarni, J. Jackson, K. Gade, and M. Fu, "Storm@twitter," in Proc. of ACM SIGMOD Int. Conf. Manag. Data - SIGMOD'14, 2014, pp. 147–156, doi:10.1145/2588555.2595641.

[4] Apache Storm, 2016. [Online]. Available: http://storm-project.net/ [Accessed: September 2017]

[5] P. Mell, T. Grance, "The NIST definition of cloud computing (Draft)", National Institute of Standards and Technology, 2009, Security and Privacy Guidelines, pp. 97-101. doi:10.6028/NIST.SP.800-145.

[6] S. Subashini, V. Kavitha, "A survey on security issues in service delivery models of cloud computing," Journal of Network and Computer Applications, vol.34, no.1, 2011, pp. 1–11, doi:10.1016/j.jnca.2010.07.006.

[7] R. Weingärtner, G. B. Bräscher, C. B. Westphall, "Cloud resource management: A survey on forecasting and profiling models," Journal of Network and Computer Applications, vol.47, no.1, 2015, pp. 99–106, doi:10.1016/j.jnca.2014.09.018.

[8] P. Wieder, J. Seidel, O. Wäldrich, W. Ziegler & R. Yahyapour. "Using SLA for resource management and scheduling—a survey," in Springer US Grid Middleware and Services, 2008, pp. 335-347, doi:10.1007/978-0-387-78446-5_22.

[9] S. A. Baset, L. Wang, and C. Tang, "Towards an understanding of oversubscription in cloud," 2nd USENIX Conf. Hot Top. Manag. Internet, Cloud, Enterp. Networks Serv. USENIX Assoc. pp. 7–7, 2012.

[10] Z. Shen, S. Subbiah, X. Gu, and J. Wilkes, "CloudScale: elastic resource scaling for multi-tenant cloud systems," in Proc. of 2nd ACM Symp. Cloud Comput. - SOCC '11, 2011, pp. 1–14, doi:10.1145/2038916.2038921.

[11] S. He, L. Guo, M. Ghanem, and Y. Guo, "Improving resource utilisation in the cloud environment using multivariate probabilistic models," In, IEEE 5th International Conference on Cloud Computing (CLOUD), Honolulu, HI, 2012, pp. 574-581, doi:10.1109/CLOUD.2012.66.

[12] L. Tomas and J. Tordsson, "An autonomic approach to risk-aware data center overbooking," IEEE Trans. Cloud Comput., vol. 2, no. 3, 2014, pp. 292–305, doi:10.1109/TCC.2014.2326166.

[13] J. Schad, J. Dittrich, and J.-A. Quiané-Ruiz, "Runtime measurements in the cloud: observing, analyzing, and reducing variance," in Proc. of the VLDB Endowment, v. 3, n. 1-2, 2010, pp. 460–471, doi:10.14778/1920841.1920902.

[14] P. A. L. Rego, E. F. Coutinho, D. G. Gomes, and J. N. De Souza, "FairCPU: Architecture for allocation of virtual machines using processing features," in Proc. of 4th IEEE International Conference on Utility and Cloud Computing, 2011, pp. 371–376, doi:10.1109/UCC.2011.62.

[15] G. Galante, L. C. E. Bona, P. A. L. Rego, and J. N. Souza, "ERHA: Execution and Resources Homogenization Architecture," in Proc. of the Cloud Computing - Intl. Conf. on Cloud Computing, GRIDs, and Virtualization, pp. 253–259, 2012.

[16] J. Xu, Z. Chen, J. Tang, and S. Su, "T-Storm: Traffic-aware online scheduling in Storm," in IEEE 34th International Conference on Distributed Computing Systems, 2014, pp. 535–544, doi:10.1109/ICDCS.2014.61.

[17] L. Aniello, R. Baldoni, and L. Querzoni, "Adaptive online scheduling in storm," in Proc. of the 7th ACM international conference on Distributed event-based systems - DEBS '13, 2013, pp. 207, doi:10.1145/2488222.2488267.

[18] R. Grandl, G. Ananthanarayanan, S. Kandula, S. Rao, and A. Akella, "Multi-resource packing for cluster schedulers," ACM SIGCOMM Comput. Commun. Rev., vol. 44, no. 4, 2014, pp. 455–466, doi:10.1145/2619239.2626334.

[19] D. Segalin, A. O. Santin, J. E. Marynowski, and L. Segalin, "An Approach to Deal with Processing Surges in Cloud Computing," in Proc. of Int. Comput. Softw. Appl. Conf., vol. 2, 2015, pp. 897–905, doi: 10.1109/COMPSAC.2015.138.

[20] A. Verma, L. Cherkasova, and R. H. Campbell, "Resource provisioning framework for MapReduce jobs with performance goals," in Proc. of the ACM/IFIP/USENIX Int'l Conference on Middleware, 2011, pp. 165–186, doi:10.1007/978-3-642-25821-3_9.

[21] C. A. Bohn and G. B. Lamont, “Load balancing for heterogeneous clusters of PCs,” Futur. Gener. Comput. Syst., vol. 18, no. 3, 2002, pp. 389–400, doi:10.1016/S0167-739X(01)00058-9.

[22] Y. Fang, F. Wang, and J. Ge, “A Task Scheduling Algorithm Based on Load Balancing in Cloud,” Web Information Systems and Mining, Lecture Notes in Computer Science, Vol. 6318, 2010, pp. 271-277, doi:10.1007/978-3-642-16515-3_34.

[23] R. Wang, D. Butnariu, and J. Rexford, “OpenFlow-Based Server Load Balancing Gone Wild Into the Wild,” in in Proc. of 11th USENIX Conf. Hot Top. Manag. internet, cloud enterprise networks and serv, pp. 12, 2011.

[24] N. Handigol, S. Seetharaman, M. Flajslik, A. Gember, N. McKeown, G. Parulkar, A. Akella, N. Feamster, R. Clark, A. Krishnamurthy and V. Brajkovic, “Aster*x: Load-Balancing Web Traffic over Wide-Area Networks,” pp. 2, 2009.

[25] N. Handigol, S. Seetharaman, M. Flajslik, N. McKeown, and R. Johari. “Plug-n-Serve: Load-Balancing Web Traffic using OpenFlow”. In ACM SIGCOMM Demo, pp. 2, 2009.

[26] D. Kreutz, F. M. V Ramos, P. E. Verissimo, C. E. Rothenberg, S. Azodolmolky, and S. Uhlig, “Software-defined networking: A comprehensive survey,” in Proc. of IEEE, vol. 103, no. 1, 2015, pp. 14–76, doi:10.1109/JPROC.2014.2371999.

[27] B. Grobauer, T. Walloschek, and E. Stöcker, “Understanding cloud computing vulnerabilities,” IEEE Secur. Priv., vol. 9, no. 2, 2011, pp. 50–57, doi:10.1109/MSP.2010.115.

[28] M. Rosenblum and T. Garfinkel, “Virtual Machine Monitors: current technology and future trends,” IEEE Computer, vol. 38, no. 5, 2005, pp. 39–47, doi:10.1109/MC.2005.176.

[29] VMware virtualization. [Online] Available: www.vmware.com [Accessed: September 2017]

[30] The Xen Project. [Online] Available: www.xenproject.org [Accessed: September 2017]

[31] L. Cherkasova, D. Gupta, and A. Vahdat, “Comparison of the three CPU schedulers in Xen,” ACM SIGMETRICS Perform. Eval. Rev., vol. 35, no. 2, 2007, pp. 42–51, doi:10.1145/1330555.1330556.

[32] J. E. Smith and R. Nair, “The architecture of virtual machines,” IEEE Computer, vol. 38, no. 5, 2005, pp. 32–38, doi:10.1109/MC.2005.173.

[33] HPE Helion Eucalyptus. [Online] Available: http://www8.hp.com/us/en/cloud/helion-eucalyptus-overview.html [Accessed: September 2017]

[34] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, “OpenFlow: Enabling Innovation in Campus Networks,” ACM SIGCOMM Comput. Commun. Rev., vol. 38, no. 2, 2008, pp. 69, doi:10.1145/1355734.1355746.

[35]Open Networking Foundation (ONF), 2014. SDN architecture 1.0, [Online]. Available: https://www.opennetworking.org/ [Accessed: September 2017]

[36] G. Wang, T. S. E. Ng, and A. Shaikh, “Programming your network at run-time for big data applications,” in Proc. of first Work. Hot Top. Softw. Defin. Networks - HotSDN’12, 2012, p. 103, doi:10.1145/2342441.2342462.

[37] A. Das, C. Lumezanu, Y. Zhang, V. Singh, G. Jiang, and C. Yu, “Transparent and Flexible Network Management for Big Data Processing in the Cloud,” Present. as part 5th USENIX Work. Hot Top. Cloud Comput., p. 6, 2013.

[38] A. Ferguson, A. Guha, and C. Liang, “Participatory networking: An API for application control of SDNs,” Sigcomm, 2013, pp. 327–338, doi:10.1145/2534169.2486003.

[39] ETSI, Network Functions Virtualisation, 2012. [Online]. Available: http://portal.etsi.org/nfv/nfv_white_paper.pdf. [Accessed: September 2017]

[40] B. Han, V. Gopalakrishnan, L. Ji and S. Lee, “Network function virtualization: Challenges and opportunities for innovations,” in IEEE Communications Magazine, vol. 53, no. 2, 2015 pp. 90–97, doi:10.1109/MCOM.2015.7045396.

[41] The Floodlight OpenFlow Controller Platform. [Online] Available: https://www.floodlight.atlassian.net. [Accessed: September 2017]

[42] Apache Zookeeper, 2016. [Online]. Available: http://zookeeper.apache.org [Accessed: September 2017]

[43] Word Count Topology, 2013. [Online]. Available: https://github.com/apache/storm/blob/master/examples/storm-starter/src/jvm/storm/starter/WordCountTopology.java [Accessed: September 2017]

[44] Storm UI REST, 2014. [Online]. Available: https://github.com/Parth-Brahmbhatt/incubator-storm/blob/master/storm-ui-rest-api.md [Accessed: September 2017]

[45] Storm Throughput Test, 2012. [Online]. Available: https://github.com/stormprocessor/storm-benchmark/blob/master/src/jvm/storm/benchmark/ThroughputTest.java [Accessed: September 2017]

[46] Alice’s Adventures in Wonderland. [Online]. Available: http://www.gutenberg.org/files/11/11-pdf. [Accessed: September 2017]

[47] S. Bouchenak, G. Gheorghe, G. Chockler, H. Chockler, and A. Shraer, “Verifying Cloud Services: Present and Future,” ACM SIGOPS Oper. Syst. Rev., vol. 47, no. 2, 2013, pp. 6–19, doi:10.1145/2506164.2506167.

[48] Java API for RESTful Services. [Online]. Available: https://jax-rsspec.java.net/. [Accessed: September 2017]
[49] Storm Class Cluster. [Online] Available: https://storm.apache.org/apidocs/backtype/storm/scheduler/Cluster.html [Accessed: September 2017]
[50] Sysbench - Modular, cross-platform and multi-threaded benchmark tool for evaluating OS parameters. [Online] Available: https://launchpad.net/sysbench [Online] [Accessed: September 2017]
[51] iPerf - The network bandwidth measurement tool. [Online] Available: https://iperf.fr/ [Accessed: September 2017]
[52] Process information pseudo-file system. [Online] Available: http://linux.die.net/man/5/proc [Accessed: September 2017]
[53] B. Peng, M. Hosseini, Z. Hong, R. Farivar, and R. Campbell, "R-Storm: Resource-Aware Scheduling in Storm," in Proc. of the 16th Middleware, 2015, pp. 149–161, doi:10.1145/2814576.2814808.
[54] F. Yan, L. Cherkasova, Z. Zhang, and E. Smirni, "DySccale: a MapReduce Job Scheduler for Heterogeneous Multicore Processors," IEEE Trans. Cloud Comput., vol. PP, no. 99, 2015, pp. 1–14, doi:10.1109/TCC.2015.2415772.
[55] H. Zhong, Y. Fang, J. Cui, "LBBSRT: An efficient SDN load balancing scheme based on server response time," Future Generation Computer Systems, vol. 68, 2017, pp 183–190, doi:10.1016/j.future.2016.10.001.
[56] N. Gude, T. Koponen, J. Pettit, B. Pfaff, M. Casado, N. McKeown, and S. Shenker, "NOX: towards an operating system for networks," ACM SIGCOMM Comput. Commun. Rev., vol. 38, no. 3, 2008, pp. 105-110, doi:10.1145/1384609.1384625.
[57] Pox Controller. [Online] Available: https://openflow.stanford.edu/display/ONL/POX+Wiki [Accessed: September 2017]
[58] SDN-based and Multi-Tenant Aware Resource Provisioning Mechanism for Cloud-based Big Data Streaming: Tools. [Online] Available: https://secplab.ppgia.pucpr.br/?q=dynamicscheduler [Accessed: September 2017]