SCALABLE AND DISTRIBUTED RESOURCE MANAGEMENT PROTOCOLS FOR CLOUD AND BIG DATA CLUSTERS

Mansour Khelghatdoust

December 2018

Abstract

Recently, there has been an increasing attention to cloud computing as a powerful enabler for many IT services. To deliver such a service efficiently, cloud providers should be able to reliably manage their available resources. To this end, cloud data centers require software to perform as an operating system in order to manage the resources of a data center efficiently and autonomously. Such a cluster operating system requires to satisfy multiple operational requirements and management objectives in large-scale data centers. There have been several cluster computing frameworks to tackle cloud resource management problem. However, the growth of popularity in cloud services causes the appearance of a new spectrum of services with complicated workload fluctuations and resource management requirements. Also, the size of data centers is growing by addition of more commodity hardware and multi core and many core servers in order to accommodate the ever-increasing requests of users. Nowadays a large percentage of cloud resources are executing data-intensive applications which need continuously changing workload fluctuations and specific resource management. Finally, by increasing the size of data centers, cloud providers demand algorithms to manage resources in a cost-efficient manner more than ever. To this end, cluster computing frameworks are shifting towards distributed resource management algorithms for better scalability and faster decision making. In addition, such distributed autonomic systems benefit from the parallelization of control and are resilient to failures.
To address the problems mentioned above, throughout this thesis we investigate algorithms and techniques to satisfy the current challenges and autonomously manage virtual resources and jobs in large-scale cloud data centers. We introduce a distributed resource management framework which consolidates virtual machine to as few servers as possible to reduce the energy consumption of data center and accordingly decrease the cost of cloud providers. This framework can characterize the workload of virtual

machines and hence handle trade-off energy consumption and Service Level Agreement (SLA) of customers efficiently. The algorithm is highly scalable and requires low maintenance cost with dynamic workloads and it tries to minimize virtual machines migration costs. We also introduce a scalable and distributed probe-based scheduling algorithm for Big data analytics frameworks. This algorithm can efficiently address the problem job heterogeneity in workloads that has appeared after increasing the level of parallelism in jobs. The algorithm is massively scalable and can reduce significantly average job completion times in comparison with the-state of-the-art. Finally, we propose a probabilistic fault-tolerance technique as part of the scheduling algorithm.

Acknowledgements

I would like to express my sincere gratitude to my advisor Dr. Vincent Gramoli for his continuous support, patience, motivation, enthusiasm, and immense knowledge. Also, My sincere thanks goes to Prof. Joachim Gudmundsson and Prof. Albert Zomaya for their support and encouragement.

I thank my colleagues and friends in both the school of IT at The University of Sydney and Architecture & Analytics Platforms (AAP) team at Data61 for the happy time they have shared with me.

Last but not the least, I would like to thank my parents for supporting me spiritually throughout my life and my wife Shiva for her love and constant support.

List of Publications

Some of the material in this thesis have been published in peer-reviewed conferences or journals as follows.

- Mansour Khelghatdoust, Vincent Gramoli, ”A Scalable and Low Latency Probe-Based Scheduler for Data Analytics Frameworks”, IEEE Transactions on Parallel and Distributed Systems (TPDS) (Under review)
- Mansour Khelghatdoust, Vincent Gramoli ”Peacock: Probe-Based Scheduling of Jobs by Rotating Between Elastic Queues”, Proceedings of the 24th International Conference on Parallel and Distributed Computing (EuroPar’18) Aug 2018
- Mansour Khelghatdoust, Vincent Gramoli, Daniel Sun ”GLAP: Distributed Dynamic Workload Consolidation through Gossip-based Learning”, Proceedings of the 18th IEEE International Conference on Cluster Computing (Cluster’16) Sep 2016

Table of Contents

Abstract iii

Acknowledgements v

List of Publications vi

Table of Contents vii

List of Figures x

List of Tables xii

1 Introduction 1

1.1 Research Objectives 2

1.2 Main Contributions 2

1.3 Thesis Organization 3

2 Background and Related Work 5

2.1 Peer-to-peer overlay networks 6

2.1.1 Structured Overlay Networks 6

2.1.2 Unstructured Overlay Networks 7

2.2 Job Scheduling in Big Data Frameworks 8

2.3 Dynamic Virtual Machine Consolidation 21

3 Peacock: Probe Based Scheduling of Jobs By Rotating Between Elastic Queues 32

3.1 Introduction 33

3.2 Terminology 34

3.3 The Peacock Scheduler . . . 35
3.3.1 Probe Rotation . . . 37
3.3.1.1 Ring Overlay Network . . . 37
3.3.1.2 The Significance of Elastic Queues . . . 38
3.3.1.3 Shared State . . . 38
3.3.1.4 Rotation Intervals . . . 39
3.3.2 Probes Reordering . . . 40
3.3.2.1 Example . . . 41
3.4 Theoretical Analysis . . . 44
3.5 Fault Tolerance Mechanism . . . 47
3.5.1 Maintaining ring when a worker crashes . . . 47
3.5.2 Recovering probes that are being executed or stored in the queue of the crashed worker . . . 47
3.5.3 Handling scheduler failure . . . 48
3.6 Evaluation Methodology . . . 48
3.7 Experiments Results . . . 50
3.7.1 Comparing Peacock Against Sparrow . . . 50
3.7.2 Comparing Peacock Against Eagle . . . 54
3.7.3 Sensitivity to Probe Rotation . . . 55
3.7.4 Sensitivity to Probe Reordering . . . 56
3.7.5 Sensitivity to Imprecise Runtime Estimate . . . 57
3.7.6 The Impact of Cluster Sizes and Loads on the Number of Probe Rotations . . . 57
3.7.7 Distributed Experiments Results . . . 58
3.7.8 The Impact of Workers Failure on The Average Job Completion Time . . . 60
3.8 Conclusion . . . 61

4 GLAP: Distributed Dynamic Workload Consolidation through Gossip-based Learning 62
4.1 Introduction . . . 63
4.2 Background . . . 66
4.3 System Model . . . 67
4.4 Gossip-Based Solution . . . 68

4.4.1 Construction and usage of Q-Learning in Cloud 68
4.4.2 Gossip Learning Component 71
4.4.3 Analysis of The Convergence of Q-value Distribution 75
4.4.4 The Gossip Workload Consolidation Component 76
4.5 Performance Evaluation 79
4.5.1 Simulation Settings 79
4.5.2 Performance metrics 80
4.5.3 Experimental Results 82
4.5.3.1 Gossip learning component works correctly and Q-values converge rapidly 82
4.5.3.2 Minimizing the number of active PMs autonomously with much lower overloaded PMs 82
4.5.3.3 Minimizing the number of overloaded PMs 82
4.5.3.4 Minimizing the number of VM migrations 85
4.5.3.5 GLAP results in less continuous SLA violation . . . 86
4.5.3.6 Minimizing energy overhead migrations 88
4.6 Summary 89

5 Conclusions and Future Directions **90**

Bibliography **93**

List of Figures

2.1 Virtual Machine Consolidation 22

3.1 Different scenarios workers handle probes. 35
3.2 RPCs and timing associated with launching a Job. The top left figure is when worker is idle and the probe is executed immediately. The bottom left figure is When the worker is busy and the probe is queued for later execution. The top right figure is when the worker is busy, the probe cannot be queued and is rotated to the neighbor node. Finally, the bottom right figure is when the worker is busy and the probe is queued but at later time its place is given to new incoming probe(s) and hence is rotated to the neighbor worker. 36
3.3 How a worker handles incoming probe(s). 42
3.4 Average Job Completion times for heavy and light load scenarios. . . 49
3.5 Fraction of jobs with shorter completion time. 51
3.6 Cumulative distribution function of Jobs completion times.10000 workers. 52
3.7 Peacock compared to without either probes rotation or probes reordering through Google trace over 10000 nodes. 53
3.8 Effect of inaccurate job runtime estimate. Peacock with inaccurate estimate compared to Sparrow for heavy and light loads of Google trace over 10000 nodes. 55
3.9 Google trace. Avg number of rotations per task 58
3.10 Distributed experiments for 3200 samples of Google trace for heavy and light workloads. 59
3.11 Average job completion time for Google trace. Peacock versus 5 and 10 failures at second 20 60

4.1 Previous distributed solutions could not cope with the varying load of VMs 64
4.2 System Architecture 67
4.3 Learning Example. In each VM, the left number indicates the current demand and the right represents the average demand until now. . . . 73
4.4 Consolidation Example. **(a)** PM1 exchanges its state with PM2 and it performs as sender because it has lower average resource utilization (40%). **(b)** VM1 migrates from PM1 to PM2 because it has greater Q-value 90 than VM2 with 55. **(c)** VM2 cannot migrate to PM2 because (4xH, L) has a negative Q-value -70 in "*in*-map". The round is finished. 78
4.5 convergence Q-values after learning phase (WOG) and aggregation phase(WG) for VM-PM ratios 2,3, and 4. 81
4.6 The fraction of overloaded / active PMs with increasing workload ratio and various cluster sizes 83
4.7 The number of overloaded PMs with increasing workload ratio and various cluster sizes 84
4.8 The number of migrations with increasing workload ratio and various cluster sizes 85
4.9 The cumulative number of migrations with increasing workload ratio for 1000 nodes 87
4.10 Energy overhead of migrations with increasing workload ratio and various cluster sizes 88

List of Tables

3.1 List of notations 42
3.2 Workloads general properties 48

4.1 SLA Metric for various cluster sizes and workload ratios 86

Chapter 1

Introduction

Cloud computing is a model for delivering computing services (such as infrastructures, platforms, and software through a network, typically Internet). There are three fundamental models for delivering services in cloud computing, namely Software as a Service (SaaS), Platform as a Service (PaaS), and Infrastructure as a Service (IaaS). Cloud computing provides the rental of IT resources on-demand over a network and charge according to a pay-as-you-go model. In the cloud computing, resources such as computing, network, storage, etc. are delivered to the users in a virtualized form. In fact, virtualization enables sharing resources among different applications and hence several users, in order to optimize the server and accordingly cluster utilization. Data centers are easily found in every sector of the worldwide economy. They consist of tens of thousands of commodity servers and have become a major computing platform, serving millions of users globally 24-7, powering both large number of Internet services and data-intensive applications. Researchers and practitioners have been developing a diverse array of cluster computing frameworks to implement the cloud computing model on clusters. One of the most prominent frameworks which sits at the heart of data centers and performs as the operating system of a cluster is the resource management framework. Resource Management is the process of procuring and releasing resources and virtualization techniques are used for flexible and on-demand resource provisioning [38]. Resource assignment is performed either on the basis of Service Level Agreement (SLA) that is agreed between the service provider and the customer or for the benefit of service providers for the purposes such as energy efficiency, cost reduction, etc. It covers a broad spectrum of algorithms such as workload management including load balancing and consolidation, scheduling, SLA-awareness,

energy-awareness, fault tolerance, etc.

1.1 Research Objectives

Resource management algorithms are widely studied in cloud computing and big data clusters. However, with the growth of popularity for such services, the number of users and hence the workload are considerably being increased. It causes the appearance of more complex and heterogeneous workloads in data centers. To handle the increasing demand for cloud computing services, cloud providers are increasing the size of data centers by adding more commodity hardwares or more number of many-core and multi-core machines. Unfortunately, most of the proposed resource management algorithms are not scalable and can not perform efficiently in complex workloads. Yet, there is a high demand for scalable and complex resource management algorithms to fulfill the changing requirements of modern data centers and services. Although, resource management in cloud computing and Big data is a broad topic, throughout this thesis we try to address some specific resource management problems for modern data centers. To achieve highly scalable algorithms, we provide fully distributed algorithms and in particular we leverage peer-to-peer techniques to reach to scalability. We handle recent workload management requirements for cloud computing and big data environments. The biggest challenge is how to capture these workloads while we still preserve scalability of the algorithms.

1.2 Main Contributions

The thesis consists of two main contributions as follows:

- We propose a workload consolidation algorithm called GLAP for cloud data centers which is scalable and is able to characterize workload of virtual machines efficiently. Cloud providers would like to reduce the cost of maintaining data centers by reducing energy consumption of servers in data center. The common approach is to leverage the facility of virtual machine migration and move virtual machines to as few servers as possible to turn off or sleep the idle servers. However, there is a trade off between energy consumption and Service Level Agreement (SLA) of customers promised by cloud providers which should be

satisfied. Most of the existing algorithms satisfy scalability for workload characterization and are not able to handle this trade off while still remaining scalable. The main reason is that they either rely on heavy optimization algorithms in which a central server requires to monitor continuously the workload status of VMs or use low accuracy workload characterization to satisfy the scalability requirement. GLAP overcomes these problems and is able to handle the trade off between energy consumption and SLA of customers in a better way than the state-of-the-art and more importantly preserve scalability property.

- The second main contribution of the thesis is about Big data environments. Todays data analytics frameworks are increasing the level of parallelism by running ever shorter and higher-fanout jobs. Scheduling such highly parallel jobs that need to be completed very quickly is a major challenge. On the one hand, schedulers will need to schedule millions of tasks per second while guaranteeing millisecond-level latency and high availability. On the other hand, the existing centralized schedulers require running expensive and heavy scheduling algorithms to place tasks on worker nodes efficiently. It causes long scheduling times which makes scheduling a bottleneck for short-duration jobs. During the literature review, we found that there has been some attempts to provide fully distributed algorithms to resolve the scheduling time bottleneck but each of them suffers from problems such as lack of massive scalability or inefficiency in handling heterogeneous job durations. We propose a new fully distributed probe-based scheduler for data analytics frameworks which is able to efficiently handle the current heterogeneous workload of jobs. In addition, it is scalable and more computing resources can be added without any adverse impact on the scheduling time of the jobs.

1.3 Thesis Organization

The thesis consists of five chapters. Beside Chapter 1 presenting the introduction, the other 4 chapters can be described as follows. Chapter 2 introduces the necessary background knowledge, related works and technical terms used in this thesis. We explain peer-to-peer overlay networks and define two categories of structured and unstructured

peer-to-peer overlay networks. Besides describing the properties of each type, we introduce a number of well-known proposed algorithms. Then, we describe different types of schedulers in data centers and define metrics and properties that schedulers are designed based on them. After that, we introduce several proposed schedulers of different types and compare them on the basis of the given metrics and properties. Finally, we describe a comprehensive literature review about workload consolidation of virtual machines in cloud data center.

In Chapter 3, we propose Peacock, a scalable and distributed scheduler for data analytic frameworks. We motivate the contribution by explaining the shortcomings of the existing algorithms and then describe the algorithm followed by examples to clarify it. Next, we propose a novel fault-tolerance algorithm. Finally, we evaluate Peacock through extensive experiments and compare with the state-of-the-art algorithms.

In Chapter 4, we propose a scalable and distributed framework for workload consolidation of virtual machines in cloud data centers called GLAP. We formulate scalable resource management using reinforcement learning and combine it with a gossip-based overlay network. We prove and evaluate the correctness of the algorithm through theoretical analysis together with extensive experimental evaluations and comparison with the state-of-the-arts.

Finally, in Chapter 5 we present a summary of our contributions and future works.

Chapter 2

Background and Related Work

In this chapter, we provide an overview and basic knowledge of characteristics, models, and overall background related to Peer-to-Peer overlay networks. We concentrate more on techniques and algorithms that are used in contributions of the thesis. It is followed by a deeper review of related works regarding resource allocation mechanisms in cloud computing and Big data analytic frameworks. This section of the chapter mainly focus on literature review of thesis's contributions about dynamic workload consolidation of virtual machines in cloud data centers as well as a comprehensive review of different scheduling techniques for efficient resource management of data analytics frameworks. Throughout this thesis, the chapter can be used as the main reference for the reader, where further references will be provided for better understanding.

2.1 Peer-to-peer overlay networks

Peer-to-peer (P2P) overlay networks are distributed systems without any hierarchical organization or centralized control. Peers form a virtual self-organizing overlay network that is built on top of the Internet Protocol (IP) network. P2P overlay networks are utilized to provide several services such as robust wide-area routing architecture, efficient search of data items, redundant storage, hierarchical naming, trust and authentication, massive scalability, fault tolerance, etc. There are two classes of P2P overlay networks: *Structured* and *Unstructured*.

2.1.1 Structured Overlay Networks

In *Structured* P2P overlay networks, topology is tightly controlled and content is placed at specified locations that will make subsequent queries more efficient. Such structured P2P systems use Distributed Hash table (DHT) as a substrate in which data object (or value) location information is placed deterministically at the peers with identifiers corresponding to the data objects unique key. DHT-based systems assign uniform random IDs extracted from large space of identifiers to a set of peers. Data objects are assigned unique identifiers called keys, chosen from the same identifier space. Keys are mapped by the overlay network protocol to a unique peer in the overlay network. The P2P overlay networks provide scalable services for storing and retrieval of {key,value} pairs on the overlay network. Each peer maintains a small routing table consisting of its neighboring peers IDs and IP addresses. Look up queries or message routing are forwarded across overlay paths to peers in a progressive manner, with the IDs that are closer to the key in the identifier space. Different DHT-based systems have various organization schemes and algorithms for the data objects and its key space and routing strategies. DHT-based overlay networks have problems regarding latency of data object lookup. For routing, peers forward a message to the next intermediate peer that may be located far away with regards to physical topology of the underlying IP network. Also, such systems assume equal participation in storing data objects which leads to bottleneck at low-capacity peers. CAN [74], CHORD [56], TAPESTRY [75], PASTRY [76], KADEMLIA [77], and VICEROY [78] are some of the most well-known proposed DHT-based overlay networks. We used a structured overlay network for distributed scheduling algorithm explained in chapter 3. It is inspired by CHORD [56] which

is actually a simple form of CHORD without storing objects or routing algorithm to locate objects.

2.1.2 Unstructured Overlay Networks

In unstructured overlay networks, peers form a random graph and use flooding or random walks or expanding-ring Time-To-Live (TTL) to search or store contents in appropriate overlay nodes. More precisely, each visited peer evaluates query on its own content and might be inefficient since searching a low replicated content requires visiting a large number of peers. Gnutella [79], Freenet [80], and BitTorrent [81] are examples of unstructured overlay network applications.

A Gossip-based communication protocol is a specific type of unstructured overlay network which is widely deployed in order to provide services for information dissemination [82], aggregation [61], load balancing [83], network management [84], synchronization [85], etc. In such protocol, every peer exchanges some information with other peers in a periodic manner so that all peers eventually converge to the same common goal or value. More precisely, each peer requires to maintain a small subset of other participating peers called *view* and periodically selects one from the list and performs a gossiping round. The key point is that each peer should maintain a continuously changing subset of peers and more importantly the resulting overlay network should represent the properties of a uniform random graph at each time. In other word, at each round, peers should select a uniform random peer and perform a gossiping round. To this end, a fundamental service for gossip-based protocols is the underlying service that provides each peer with a list of peers is called peer sampling service. The crucial is that this service should be implemented in such a way that any given peer can exchange information with peers that are selected following a uniform random samples of all nodes in the system.

A naive solution is that each peer maintains a view of all peers in the network and at each round selects one peer uniformly at random. In fact, every peer knows all the other peers of the system. However, this solution is not scalable as maintaining such view has an overhead in a dynamic system in which peers join and leave the network regularly. Therefore, it is a wrong assumption if we apply this solution for the underlying peer sampling service. The idea behind an ideal solution is to use a gossip-based dissemination of membership information which enables the building of

unstructured overlay networks that capture the dynamic nature of such fully decentralized peer-to-peer systems through continuous gossiping of this information. Additionally, it provides good connectivity in the presence of failures or peer disconnections. Gossip-based peer sampling service consists of three dimensions as follows: (1) *Peer selection*, (2) *View propagation*, and (3) *View selection*. Many variations exist for each of three dimensions. In fact, peer sampling services are different as each of them has different algorithms or methods for three mentioned dimensions. Interested readers are referred to [63], [62], and [7] for more details. Chapter 4 proposes a scalable dynamic workload consolidation for cloud data centers. In this work we used a peer sampling service and a novel gossip protocol to migrate virtual machines between nodes in data center.

2.2 Job Scheduling in Big Data Frameworks

Job scheduling was already extensively studied in the domain of high performance computing for scheduling of batch long running CPU-intensive jobs. However, after the appearance of emerging data-parallel systems, it again has recently become a hot and on demand research topic. In this section, we try to review the corresponding literature broadly and evaluate the latest advances in this area. We found that all job scheduling algorithms can be evaluated from five different aspects which are Scheduling Architecture, Queue Management, Placement Strategies, Resource Allocation Policies, and Scheduling Amelioration Mechanisms. First, we explain each of these aspects and then discuss and compare various related works on the basis of the proposed metrics.

- **Scheduling Architecture**

  In one perspective, there could be two general scheduling architectures called monolithic and Two-level scheduling architectures. Monolithic schedulers use a single scheduling algorithm for all jobs and applications of cluster. Two-level schedulers act as coordinator or operating system for clusters and have a single active resource manager that offers compute resources to multiple parallel, independent scheduler frameworks. Until now there has been three different monolithic cluster schedulers proposed by different algorithms. First, centralized schedulers that use elaborate algorithms to find high-quality placements,

but have latencies of seconds or minutes. Second, distributed schedulers that use simple algorithms that allow for high throughput, low latency parallel task placement at scale. However, their uncoordinated decisions based on partial, stale state can result in poor placements. Third, hybrid schedulers that split the workload across a centralized and a distributed component. They use sophisticated algorithms for long-running tasks, but rely on distributed placement for short tasks.

- **Queue Management**

  Data centers have limited resources and it is typical if workload exceeds the maximum capacity of data center. Therefore, queues are required in order to provide projections on future resource availability. The use of queues enables schedulers and workers to dispatch tasks proactively based on future resource availability, instead of based on instantaneous availability. It is crucial that schedulers manage queues efficiently. Scheduling algorithms provide various queue management techniques by placing a queue on either scheduler or worker nodes. Also, some algorithms place queue on both worker and scheduler nodes. Basically, centralized schedulers use scheduler-side queues while distributed schedulers use worker-side queues. Essentially, queue management techniques are evaluated by appropriate queue sizing, prioritization of task execution via queue reordering, starvation freedom, and careful placement of tasks to queues.

- **Placement Strategies**

  Efficient task placement strategies by the cluster scheduler lead to higher machine utilization, shorter batch job runtime, improved load balancing, more predictable application performance, and increased fault tolerance. Centralized schedulers often make decisions through applying algorithmically complex optimization in multiple dimensions over global view of data center. However, achieving such high task placement quality conflicts with the need for a low latency placement. To this end, distributed schedulers often make quick task placement decisions with lower quality. Generally, various task placement algorithms can be divided into two types: Task-by-task placement and batching placement. In the former, the task waits in a queue of unscheduled tasks until it is dequeued and processed by the scheduler. Some schedulers have tasks wait in

a worker queues which allows for pipelined parallelism. Such placement has the advantage of being amenable to uncoordinated, parallel decisions in distributed schedulers. On the contrary, as disadvantages, the scheduler commits to a placement early and restricts its choices for further waiting tasks, and second, there is limited opportunity to amortize work. In the latter, processing several tasks in a batch allows the scheduler to jointly consider their placement, and thus to find the best trade-off for the whole batch.

- **Resource Allocation Policies and User Constraints**

  Schedulers might need to consider a set of policies and constraints imposed by environment and users. Such constraints can be, for example, over where individual tasks are launched or inter-user isolation policies to govern the relative performance of users when resources are contended. Such constraints can be defined as per-job or per-task and are required in data-parallel frameworks, for instance, for the sake of locality and performance, to run tasks on a machine that holds the tasks input data on disk or in memory. Another type of policy relates to the resource allocation in which schedulers allocate resources according to a specific policy when aggregate demand for resources exceeds capacity. For example, weighted fair sharing provides cluster-wide fair shares in which two users using the same set of machines will be assigned resource shares proportional to their weight.

- **Scheduling Amelioration Mechanisms** To cope with unexpected cluster dynamics, suboptimal estimations, other abnormal runtime behaviors which are facts of life in large-scale clusters, and also to drive high cluster utilization while maintaining low job latencies, cluster scheduling algorithms provide different amelioration techniques to improves initial scheduling decisions. This is vital for distributed scheduling when initial placement is made quickly on the basis of no or partial information of the cluster. It is also useful for centralized schedulers due to unexpected cluster dynamics that are not known at the scheduling time. We found several amelioration techniques in the literature including but not limited to work stealing, re-dispatching, load shedding, straggler detection, duplicate scheduling, queue reordering, and etc.

In the rest of this section we explore the most relevant works and explain their

pros and cons. Since the contribution of this thesis is about probe-based distributed schedulers which is explained in chapter 3, first we discuss and compare this type of schedulers and then we continue with other sort of studied schedulers.

Sparrow [27] is a fully distributed scheduler with design goals to perform at extreme scalability and low latency. The placement strategy of Sparrow is batch sampling followed by late-binding technique for better placement decisions. More precisely, it sends a number of probe messages two times more than the number of tasks and places probes in the least loaded workers. Sparrow performs well for lightly and medium loaded clusters, however, it faces with challenges in highly load clusters for heterogeneous workloads due to head-of-line blocking. This stems from taking scheduling decisions without benefiting from full load information. Moreover, Sparrow does not have amelioration mechanisms to compensate in case the initial assignment of tasks to nodes is suboptimal. Our algorithm Peacock as we illustrate in chapter 3 outperforms Sparrow in all workloads.

The authors in [89] augmented Sparrow by defining a new term called probe sharing. The main idea of the paper is that due to Sparrow's sample-based techniques, some tasks in subsequent jobs may be scheduled earlier than those in the earlier jobs, which results in scheduling disorder and inevitably causes unfairness. The proposed solution is jobs that arrive at the same Sparrow scheduler can share their probes to ensure that all tasks in the earlier arrived jobs can be scheduled earlier than subsequent jobs. This approach has a number of problems. First, the authors does not consider Head-of-Line blocking as executing earlier arrived jobs can not always guarantee better response time and fairness. Since jobs have heterogeneous durations and it is more efficient if some short duration tasks to be executed before long duration tasks to prevent happening Head-of-Line blocking problem even if the short job arrived later the long job. It also never necessarily violates fairness. The authors try to solve disordering problem for jobs each scheduler in Sparrow to prevent Head-of-Line Blocking. However, an efficient solution is the one can reorder all jobs of all schedulers in a distributed fashion in order to fix Head-of-Line blocking. This is what Peacock does efficiently.

The authors in Hawk [29] proposed an algorithm to resolve shortcomings of Sparrow in highly loaded clusters. Hawk divides jobs into two categories called short and

long duration jobs. There is a centralized scheduler in charge of placing long jobs and a number of distributed schedulers responsible for placing short jobs. Centralized scheduler has a global view of long jobs and make decisions based on that. The distributed scheduler is a probe based scheduler identical to Sparrow. Moreover, Hawk divides worker nodes into two parts. A big part is shared between both job types and a smaller part dedicated only to short jobs. The algorithm is augmented with a randomized task stealing approach to reduce Head-of-Line blocking. Once a worker becomes idle, it connects to another randomly chosen worker and gets a short job that gets stuck behind a long job in the queue of the worker node. Although Hawk outperforms Sparrow in highly load scenarios, it falls behind Sparrow in light and medium scenarios. Additionally, there is static separator between short and long jobs as well as partitioning of cluster. Such static separation might be misleading. Peacock outperforms Hawk in both light and heavy workloads.

The same authors propose another algorithm called Eagle [28] which works better than Hawk. Eagle divides the datacenter nodes in partitions for the execution of short and long jobs to avoid Head-of-Line blocking, and introduces sticky batch probing to achieve better job-awareness and avoids stragglers. Eagle shares some information among workers called Succinct State Sharing, in which the distributed schedulers are informed of the locations where long jobs are executing. Eagle avoids stragglers and mitigate head-of-Line blocking by using a Shortest Remaining Processing Time reordering technique to prevent starvation. Eagle achieves it by replacing task stealing proposed in Hawk with a re-dispatching of short probes. In other words, once a short probe reaches a worker node running or queuing a long job, distributed scheduler either reschedule it to a new worker or directly sends to a worker belongs to short task partition. Eagle performs well under heavy loads. Eagle the same as Hawk suffers from static configuration values. Besides the same parameters as Hawk, Eagle needs to set a threshold value for reordering mechanism which might be misleading, too. Furthermore, probe re-dispatching is probabilistic and may not lead to a better result. However, as we show Peacock outperforms Eagle in different workload scenarios under heterogeneous jobs by proposing a deterministic algorithm for probe scheduling.

Omega [53] is a distributed, flexible and scalable scheduler for large compute clusters. Omega uses parallelism, shared state, and lock-free optimistic concurrency control in order to make scheduling decisions to increase efficiency and maximize utilization. Omega has a parallel scheduler architecture built around shared state, using lock-free optimistic concurrency control, to achieve both implementation extensibility and performance scalability. In Omega each scheduler has full access to the entire cluster. Schedulers compete in a free-for-all manner, and use optimistic concurrency control to mediate clashes when they update the cluster state. This eliminates two shortcomings of the two-level scheduler architecture which are limited parallelism due to pessimistic concurrency control, and restricted visibility of resources in a scheduler framework. Omega maintains a resilient master copy of the resource allocations in the cluster, which is called cell state. Each scheduler is given a frequently updated copy of cell state that it uses for making scheduling decisions. Once a scheduler makes a placement decision, it updates the shared copy of cell state in an atomic commit. The same as our algorithm Peacock, Omega schedulers operate completely in parallel and do not have to wait for jobs in other schedulers, and there is no inter-scheduler head of line blocking. Since it is distributed approach it does not guarantee global fairness, starvation avoidance, etc. Also, Omega is not able to provide amelioration mechanisms in order to efficiently handle dynamic behavior of cluster and unpredictable changes that are inevitable. One advantage of Omega is that it can support gang scheduling i.e., either all tasks of a job are scheduled together, or none are, and the scheduler must try to schedule the entire job again. Omega needs to keep a complete view of data center and schedulers need strictly to make consensus on the latest copy of view which makes it expensive and has a negative impact on the performance. On the contrary, shared state in Peacock is very lightweight and different schedulers does not need to make consensus about the latest updated version.

Tarcil [36] is another probe-based distributed scheduler that is designed to target both scheduling speed and quality which aims to make it appropriate for large, highly-loaded clusters running both short and long jobs. Tarcil like Sparrow, Hawk, and Eagle uses probe sampling. However, unlike them, sample size is dynamically changing based on load analyzes in order to provide guarantees on the quality of scheduling decisions with respect to resource heterogeneity and workload interference. It first looks up the jobs resource and interference sensitivity preferences which are CPU,

caches, memory, and I/O channels. Tarcil can obtain useful information about estimates of the relative performance on the different servers, as well as estimates of the interference the workload can tolerate and generate in shared resources. Tarcil is a shared-state scheduler and schedulers periodically collect statistics on the cluster state. These statistics are updated as jobs begin and end execution. Using that information, schedulers determine whether they can quickly find resources for a job, otherwise the scheduler queues it for a while. A queued application waits until it has a high probability of finding appropriate resources or until a queuing time threshold is reached. Tarcil is not performing well for light and medium workloads and even for high load cases is not as good as Peacock since it is still rely on batch sampling proposed by Sparrow. The authors ran experiments for up to 400 nodes and it is not clear if it is scalable and can work efficiently for large clusters. In addition, it targets more on placement efficiency and the authors do not propose any method for handling of Head-of-Line blocking. In addition, there is no amelioration mechanism to improve job scheduling decisions at run time after initial scheduling due to dynamic behavior of cluster.

Mercury [32] is a hybrid resource management framework that allows applications to trade-off between scheduling overhead and execution guarantees. The key point is that Mercury offload work from the centralized scheduler by augmenting the resource management framework to include an auxiliary set of schedulers that make fast/distributed decisions. Additionally, applications are divided into two execution models given by users called guaranteed or opportunistic. Applications may now choose to accept high scheduling costs to obtain strong execution guarantees from the centralized scheduler, or trade strict guarantees for sub-second distributed allocations which is suitable for applications with short tasks durations. Mercury uses queues in centralized scheduler node as well as worker nodes for distributed scheduling. Mercury follows different placement strategies for centralized and distributed allocations. The central scheduler allocates a guaranteed job on a node, if and only if that node has sufficient resources to meet the containers demands. By tracking when guaranteed jobs are allocated/released on a per-node basis, the scheduler can accurately determine cluster-wide resource availability. The placement policy of distributed schedulers is to minimize queuing delay by mitigating head-of-line blocking. Mercury applies two ways in order to maximize cluster efficiency as amelioration mechanisms. It performs

load shedding to dynamically (re)-balancing load across nodes so that it mitigates occasionally poor placement choices for opportunistic short length jobs. Moreover, it utilizes a simple queue reordering technique.

Apollo [35] is a highly scalable and coordinated scheduling framework. Apollo performs scheduling decisions in a distributed manner, utilizing global cluster information via a loosely coordinated mechanism. The distributed architecture of Apollo is different from probe-based distributed schedulers such as Sparrow [27], and Eagle [28]. In Apollo, besides the existence of a set of independent distributed schedulers, there exists a resource monitor node. Such node is responsible for communicating with worker nodes to get their state continuously in order to provide a global view of the cluster for scheduler nodes. Each task is scheduled on a server that minimizes the task completion time. The estimation model incorporates a variety of factors and allows a scheduler to perform a weighted decision, rather than solely considering data locality or server load. Similar to Mercury [32], Apollo creates two classes of tasks, regular tasks and opportunistic tasks in order to drive high cluster utilization while maintaining low job latencies. It guarantees low latency for regular tasks, while using the opportunistic tasks for high utilization. To cope with unexpected cluster dynamics, Apollo is augmented with a series of correction mechanisms that adjust and rectify sub optimal decisions at runtime. One of the amelioration techniques is duplicate scheduling. When a scheduler gets new information from resource monitor, Apollo reschedules task in optimal server if it finds that previous task scheduling is not efficient based on new information and duplicates are discarded when one task starts. Randomization and straggler detection are two more techniques that Apollo uses to enhance scheduling decisions.

The authors in [33] proposed a centralized fine grain cluster resource-sharing model with locality constraints called Quincy. The algorithm maps scheduling problem to a graph data structure, where edge weights represents the competing demands of data locality, fairness, and starvation-freedom. Also, a solver computes the optimal on line schedule according to a global cost model. Quincy proposed a multiple queues approach. The schedulers maintain three types of queues, one for each computer in the cluster, one for each rack and one cluster-wide queue. When a worker task is submitted to the scheduler it is added to the tail of multiple queues. When a task is matched to a

computer, it is removed from all the queues it had been placed in. Although Quincy is able to provide a high efficient scheduling decisions, it is not scalable and suffers from single point of failure problem.

Firmament [65] is a new centralized scheduler designed to augment Quincy [33] in order to overcome the weakness of it. The algorithm can scale to over ten thousand machines at sub second placement latency while it is able to continuously reschedules all tasks via a min-cost max-flow (MCMF) optimization technique. The authors claims that centralized sophisticated scheduling algorithms can be fast if they match the problem structure well, and also few changes to cluster state occur while the algorithm runs. Firmament reduces several minutes task placements of Quincy to latencies of hundreds of milliseconds on a large cluster while reaches the same placement quality as Quincy.

There are a set of Two-level centralized schedulers that are designed to perform as operating system for cluster consisting several independent applications. These schedulers aim to allocate resources at coarse granularity, either because tasks tend to be long-running or because the cluster supports many applications that each acquire some amount of resources and perform their own task-level scheduling. These schedulers sacrifice request granularity in order to enforce complex scheduling policies; as a result, they provide insufficient latency and/or throughput for scheduling sub-second tasks and are suitable for long-running tasks such as high performance computing jobs. Borg [44], Mesos [34], and Yarn [45] belong to this category.

Mesos shares resources in a fine-grained manner, allowing frameworks to achieve data locality by taking turns reading data stored on each machine. More precisely, Mesos follows a Two-level architecture in which it implements a centralized scheduler that takes as input framework requirements, resource availability, and organizational policies and decides how many resources to offer to each framework and then delegate control over scheduling to the frameworks through a new abstraction called a resource offer. Actually each framework decides independently which resources to accept and which applications or jobs to run on the allocated resources.

Another algorithm in the category of Two-level scheduling architecture is Apache Yarn [45]. It was designed to address two shortcomings of initial Apache Hadoop

scheduler which were tight coupling of a specific programming model with the resource management infrastructure, and centralized handling of jobs control flow which resulted in scalability problems for the scheduler. Yarn is a multi cluster scheduler that by separating resource management functions from the programming model, YARN delegates many scheduling-related functions to per-job components. This separation provides a great deal of flexibility in the choice of programming framework. YARN provides greater scalability, higher efficiency, and enables a large number of different frameworks to efficiently share a cluster. Specifically, there is a per-cluster Resource Manager that tracks resource usage and node liveness, enforces allocation invariants, and arbitrates contention among tenants. Therefore the central allocator can use an abstract description of tenants requirements, but does not need to be aware of the semantics of each allocation. There is an Application Master which coordinates the logical plan of a single job by requesting resources from the Resource Manager.

Borg [44] is another large scale cluster manager scheduler being used at Google data centers. Borg manages a site comprises of a number of clusters. Each cluster is divided into a number of cells in which each cell consists of thousands of machines. Each Borg cell consists of a logically centralized controller called the Borg master, and an agent process called the Borglet that runs on each machine in a cell. In nutshell, when a job is submitted, the Borg master stores it persistently in the Paxos store and adds the jobs tasks to the pending queue. This is scanned asynchronously by the scheduler, which assigns tasks to machines if there are sufficient available resources that meet the jobs constraints. The scan proceeds from high to low priority, modulated by a round-robin scheme within a priority to ensure fairness across users and avoid head-of-line blocking behind a large job. The Borglet is a local Borg agent that is present on every machine in a cell. It handles everything related to tasks assigned to the machine including starting and stopping tasks; restarts them if they fail; manages local resources by manipulating OS kernel settings, and etc. Borg uses a global queue at Borg master node for scheduling all coming jobs. Accordingly, task placement is done directly to worker nodes since there is no queue on workers.

Jockey [37] is a centralized scheduler aims to provide guarantees on job latency for data parallel jobs. Jockey uses a simulator to capture internal dependencies of jobs efficiently in order to computes statistics and predicates the remaining run time of jobs

with respect to resource allocations, etc. Using a previous execution of the job and a utility function, Jockey models the relationship between resource allocation and expected job utility. During job runtime, the control policy computes the progress of the job and estimates the resource allocation that maximizes job utility and minimizes cluster impact by considering the task dependency structure, individual task latencies, failure probabilities, etc. Jockey has three components which are a job simulator to estimate the job completion in off-line mode, a job progress indicator which is used at runtime to characterize the progress of the job, and a resource allocation control loop, which uses the job progress indicator and estimates of completion times from the simulator to allocate tokens such that the jobs expected utility is maximized and its impact on the cluster is minimized. However, Jockey has a number of limitations. First, it is centralized and is not scalable and cannot provide low latency scheduling for short length tasks. It does not provide any amelioration mechanism to handle dynamic nature of cluster efficiently. Jockey is only capable of guaranteeing job latency for jobs it has seen before. Jockey makes local decisions to ensure each job finishes in guaranteed latency and cannot reach a globally optimal allocations.

Bistro [43], is a centralized hierarchical scheduler developed for Facebook data center. It aims to run data-intensive batch jobs on live along with customer facing production systems without degrading the performance of end-users. Bistro is a tree-based scheduler that safely runs batch jobs in the background on live customer-facing production systems without harming the foreground workloads. Bistro treats data hosts and their resources as first class objects subject to hard constraints and models resources as a hierarchical forest of resource trees. Total resource capacity and per job consumption is specified at each level. As many tasks as possible are scheduled onto leaves of tree as long as it satisfies tasks resource requirements along the paths to root. More precisely, each task requires resources along a unique path to root. As mentioned, Bistro does not use any queue for scheduling and in fact it is a tree-based scheduler and not queue-based. Bistro the same as other centralized schedulers suffer from scalability and single point of failure problem.

TetriSched [30] is a centralized scheduler using a reservation system to continuously re-evaluate the immediate-term scheduling plan for all pending jobs. It divides jobs into two categories reservation and best-effort jobs. TetriSched leverages jobs

runtime estimate and deadline information supplied by a deployed reservation system and decide to wait for a busy preferred resource type or launch task on less preferred placement options. Reservation systems such as YARN reservation system [42] are designed to guarantee resource availability in the long term future. It serves as an admission control system to ensure that resource guarantees are not over committed. By contrast, a scheduling system is designed to make short-term job placement. Scheduling systems are responsible for optimizing job placement to more efficiently utilize resources. In fact, TetriSched is designed to work with reservation system provided in YARN and they complement each other. TetriSched performs global scheduling by batching multiple pending jobs and considering them for placement simultaneously, since constraints on diverse resources can arbitrarily conflict. Like other centralized schedulers, it suffers from scalability problems and also is not able to schedule short jobs quickly.

Paragon [66] is a centralized and energy-aware scheduler for Data Centers that tries to handle both workload heterogeneity and interference. By co-locating applications a given number of servers can host a larger set of workloads and hence by packing workloads in a small number of servers when the overall load is low, the rest of the servers can be turned off to save energy. Paragon leverages validated analytical methods, such as collaborative filtering to quickly and accurately classify incoming applications with respect to platform heterogeneity and workload interference. In addition, classification uses minimal information about the new application and instead it leverages information the system already has about applications that it has previously seen. The output of classification is used by a greedy scheduler to assign workloads to servers in a manner that maximizes application performance and optimizes resource usage. The major problem of Paragon is that scheduling time may take in minutes and it is not able to provide low latency scheduling time for short length jobs. It is centralized and hence it is not scalable. Paragon is strongly rely on prediction and it does not propose any runtime mechanism to improve placement decisions after initial scheduling.

Quasar [88] is a centralized scheduler for managing cluster of servers. It aims to increase resource utilization while providing consistently high application performance. Quasar employs three techniques to achieve the goals. First, users express performance constraints for each workload, letting Quasar determine the right amount of resources

to meet these constraints at any point. Therefore it does not rely on resource reservations, which lead to resource underutilization. The reason is that users do not necessarily understand workload dynamics and physical resource requirements of complex systems. Then, Quasar uses classification techniques to quickly and accurately determine the impact of the amount of resources, type of resources, and interference on performance for each workload and dataset. After that, it uses the classification results to jointly perform resource allocation and assignment, quickly exploring the large space of options for an efficient way to pack workloads on available resources. Quasar monitors workload performance and adjusts resource allocation and assignment when needed. Similarly to Paragon, it is not scalable and relies on prediction and it does not propose any amelioration mechanism to handle dynamic behavior of cluster.

GRAPHENE [86] is centralized cluster scheduler. The main contribution is to schedule efficiently jobs with a complex dependency structure and heterogeneous resource demands. Since scheduling a DAG of independent homogeneous or heterogeneous tasks is an NP-hard problem and also heuristics solutions perform poorly when scheduling heterogeneous DAG. GRAPHENE is trying to solve the problem by computing a DAG schedule, off line, by first scheduling long running tasks and those that are hard to pack and then scheduling the remaining tasks without violating dependencies. This off-line schedules are distilled to a simple precedence order and are enforced by an on-line component that scales to many jobs. Also, on-line component uses heuristics to compactly pack tasks and to handle efficiently trade of between fairness and faster job completion times. An obvious problem of the algorithm is that it is centralized and also does a lot of computation during scheduling time and hence is not scalable and not suitable for scheduling short duration tasks. Also, by considering the fact that firstly each hierarchy of a DAG consists of several recurring tasks with highly similar task duration and secondly each hierarchy can be seen as different job, paying such high cost for DAG may not be affordable.

The authors in [31] extensively evaluate how centralized and distributed modern cluster schedulers manage queues and discuss relevant pros and cons. They provide principled solutions to the observed problems by introducing queue management techniques, such as appropriate queue sizing, prioritization of task execution via queue

reordering, starvation freedom, and careful placement of tasks to queues. The authors propose two general schedulers. One is centralized scheduler which consists of both queues at scheduler and worker nodes. The distributed scheduler is designed with queues at worker nodes. The authors evaluated several techniques for task placement, queue sizing, and queue reordering for both types of schedulers. Although the proposed techniques are broad, there exists shortcomings in the proposed algorithms that causes Peacock outperforms this approach. The algorithm supports a Fixed-sized queue based on either task count or task duration. However, in Peacock we provide an elastic queue sizing which continuously and in a distributed manner update size of queue which results in more efficient scheduling decisions. In addition, using probe rotation proposed in peacock, our algorithm can continuously find a better place to accommodate tasks while in this algorithm the final placement decision is made at scheduling time. Accordingly, Peacock can lead to better job completion time than this algorithm.

2.3 Dynamic Virtual Machine Consolidation

Workload consolidation through virtual machine (VM) live migration (a feature provided by hypervisors) is one of the most important mechanisms for designing energy efficient cloud resource management systems. The aim is to migrate virtual machines into as few number of Physical Machines (PM) as possible to attain both increasing the utilization of Cloud servers while considerably reducing the energy consumption of the cloud data center by turning off empty PMs. In other words, any workload consolidation algorithm using live migration should capture trade off between energy efficiency and Service Level Agreement (SLA) of customers. More interestingly, workload consolidation is a demand of cloud service providers so that they can serve to as many customer as possible by using as few PM as possible to increase profit and make it more affordable and more importantly do not violate SLA of customers. However, the crux is that due to workload fluctuation of VMs during time, packing more VMs into a single server may lead to SLA problems, since VMs share the underlying physical resources of the PM. Workload consolidation is a NP-Hard problem and hence, a broad spectrum of heuristic and meta-heuristic algorithms have been proposed that aim to achieve near-optimality.

In Figure 2.1, there are 4 VMs running in 3 different PMs. Consolidation algorithm

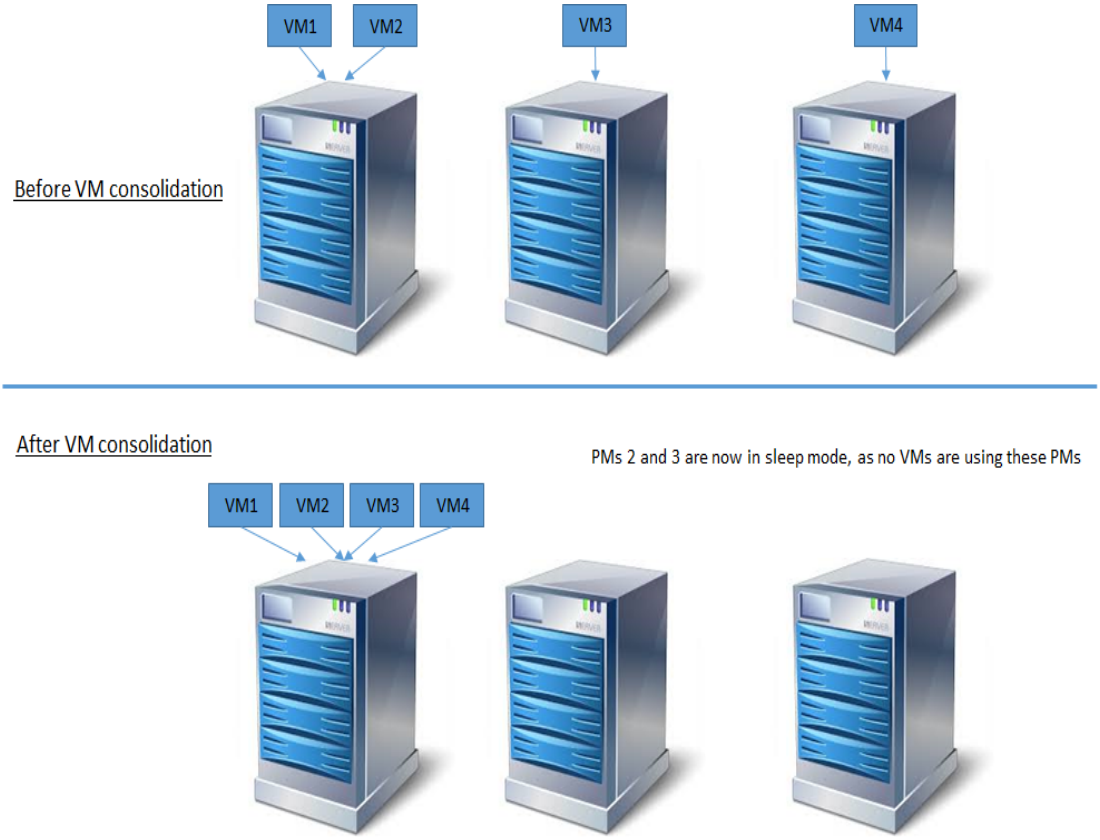
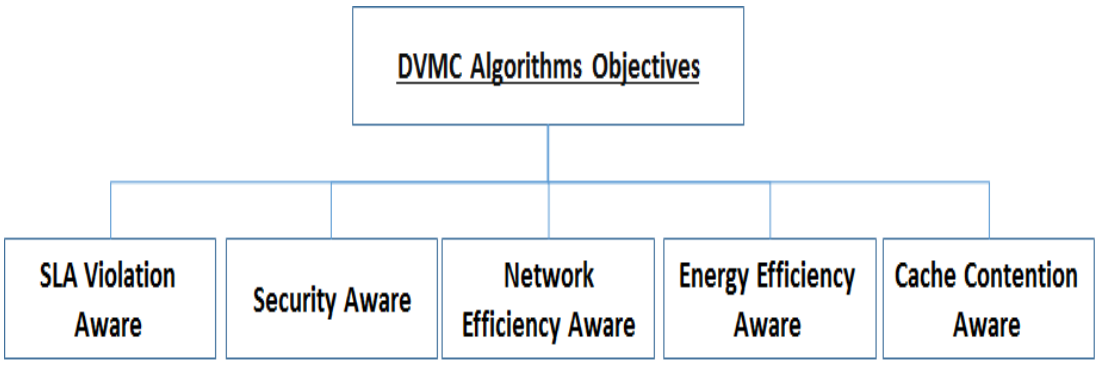


Figure 2.1: Virtual Machine Consolidation

migrates 2 VMs, now all are running on one PM and the 2 other PMs can move to sleep mode and hence, consume less energy. Recall that VMs hosted in a PM share the underlying physical resources of that PM. Therefore, with the increased number of VMs sharing underlying resources of a single PM, waiting time for a VM prior receiving its required resources becomes higher and it results in arising resource contention which would lead towards poor Quality of Service (QoS). Additionally, since resource demand of VMs varies from time to time, migration by considering only current resource demands of VMs would lead to arising SLA violation of customers. To sum up, an ideal virtual machine consolidation algorithm should satisfy resource utilization maximization, energy consumption minimization and Cloud profit maximization. Moreover, it should be scalable to hundred of thousands PMs as well as it needs to be robust such that the performance does not degrade with the fluctuation in resource demand of VMs.

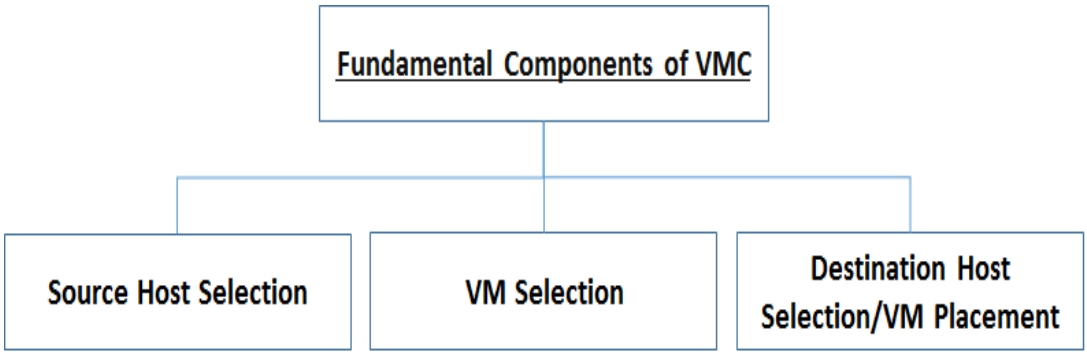
In the rest of this section, first we try to classify the important components of VM consolidation algorithms from different perspective. We will give a description for each one and then, we compare our GLAP algorithm with a set of the most relevant related works that we studied and show in which aspects it outperforms those proposed techniques.



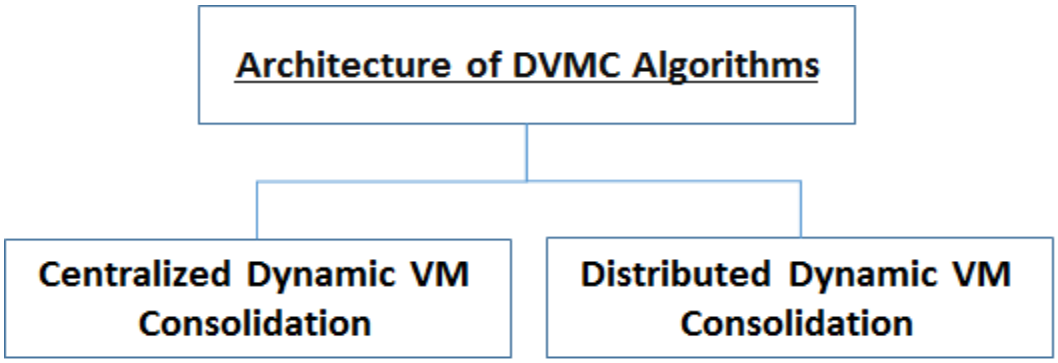
In general, different VM consolidation algorithms either distributed or centralized are designed to achieve one or more of 5 main objectives. One of the most important objectives is **SLA Violation Aware**. It happens when allocated resources (i.e., CPU, memory, IO, and network bandwidth) are less than VMs resource demand during life-time of VMs. The reason is that VMs running on one PM share underlying physical resources and hence inefficient consolidation of several VMs into a PM very likely leads to SLA violation. Another important metric is **Energy Efficiency Aware**. The maintenance of data centers require the consumption of too much electricity (e.g., for cooling) which makes it expensive for cloud providers. One efficient way is leverage VM live migration to consolidate as many VMs as possible into as few PMs as possible and change the state empty PMs to sleep mode. Through this way, cloud providers reduce energy consumption. As it is clear, there is a trade-off between SLA violation of customers and energy efficiency of data centers. **Cache Contention Aware** is another objective that has recently got the attention of researchers. Shared resources are not limited to CPU, Memory, IO, and Network bandwidth. In addition, VMs share various levels of CPU caches. One VM may experience extra cache misses, as collocated VMs on the same CPU, fetch their own data into the Lowest Level Cache (LLC), which leads to evict the VMs data from the LLC and causes cache misses to the VM. There

are a sort of algorithms that are **Network Efficiency Aware**. Basically, the objectives are minimization of network energy consumption, reduction of network congestion, etc. **Security Aware** is another objective which aims to secure cloud data center as a multi-tenancy environment. The reason is that VMs of different clients are hosted in same PM, and share the underlying physical resources.

VM consolidation algorithms need to address three main components. The first com-

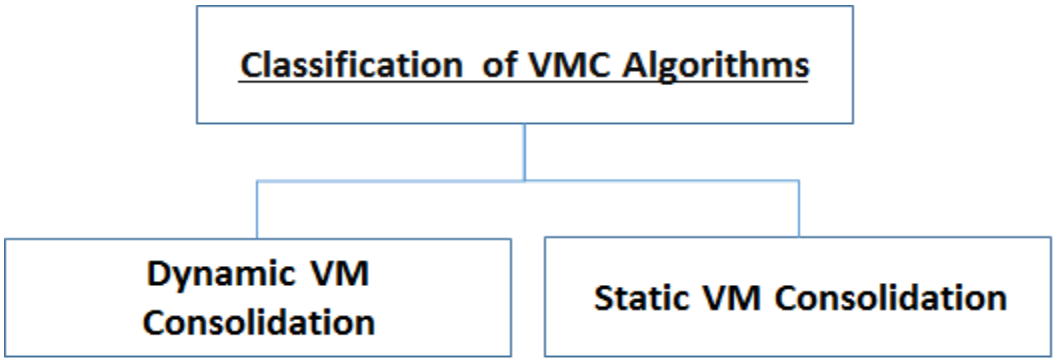


ponent is in charge of selecting one or more PMs whose VMs need to be migrated out. The second component takes one of the selected PMs and marks the most suitable VM(s) for migration. The last component selects the PM(s) as target to accept migrating VM(s) which was selected by VM selection component. In centralized ar-



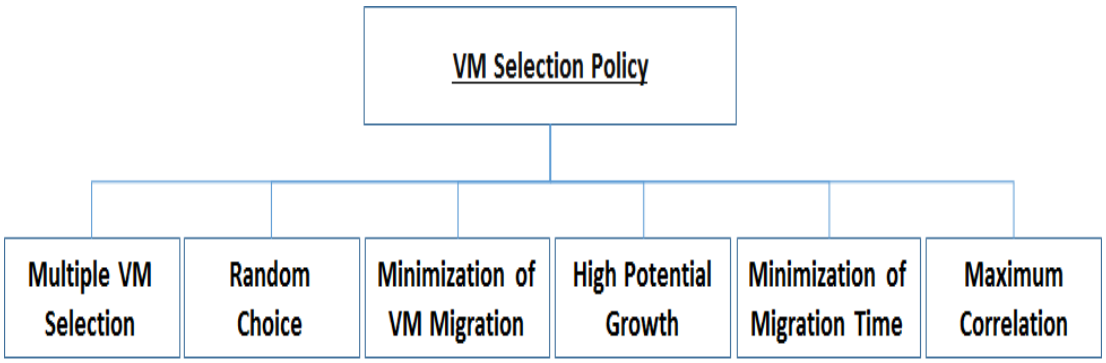
chitectures, a single server has the information about resource availability and current state of all PMs and VMs. Using the VM consolidation algorithm, the single server selects one or more target PMs as well as one or more VMs to migrate to those PMs. The main problem with centralized architecture is the single point of failure. Additionally, running centralized optimization VM consolidation algorithms are expensive and slow and are not scalable. Also, such algorithms cannot deal with VMs quick

behavioral changes efficiently. On the other hand, in distributed architecture, there is no centralized server responsible for running the algorithm. Instead, each PM run the algorithm and retains the information of a subset of other PMs/neighbors. The PMs collaboratively and asynchronously exchange information between themselves and take migration decisions. Such distributed algorithms are designed to overcome the shortcomings of the centralized VM consolidation algorithms. VM consolidation



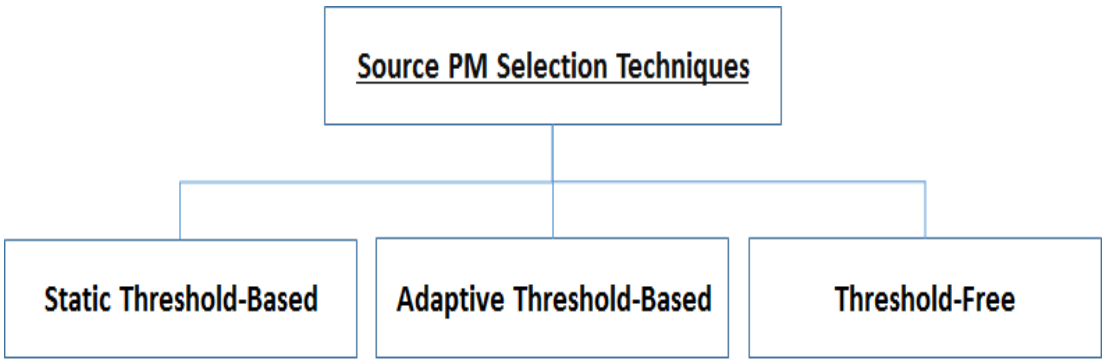
algorithms based on considering dynamic properties of data centers including VMs and PMs can be divided into Dynamic VM consolidation and Static VM consolidation algorithms. Static VM consolidation algorithms consider a set of empty PMs as well as a number of VMs and try to place all VMs into PMs in an efficient way. More precisely, static algorithms do initial VM placement in minimum number of active PMs in such way that energy-efficiency and resource utilization increases. Such algorithms are not able to improve placement VMs to PMs during time in an environment in which there are continuous arrival and departure of VMs. PMs might crash and resource demand of VMs fluctuate from time to time. On the contrary, dynamic consolidation algorithms take into account changing workload or resource requirement of any VM and its location continuously and reallocate existing VMs in lesser number of PMs such that the number of active PMs is minimized and hence data center consumes less energy.

One of the three main fundamental components is VM selection policy which specifies which VM(s) should be migrated from source PM to destination PM. VM selection policy can be divided into two categorization of single and multiple VMs. Typically, an application such as multi-layer enterprise applications consists of several VMs which work collaboratively to provide services to customers. There are a number of algorithms which take into account the relationship between VMs when making consolidation decisions. For single VM selection, we found five techniques that are

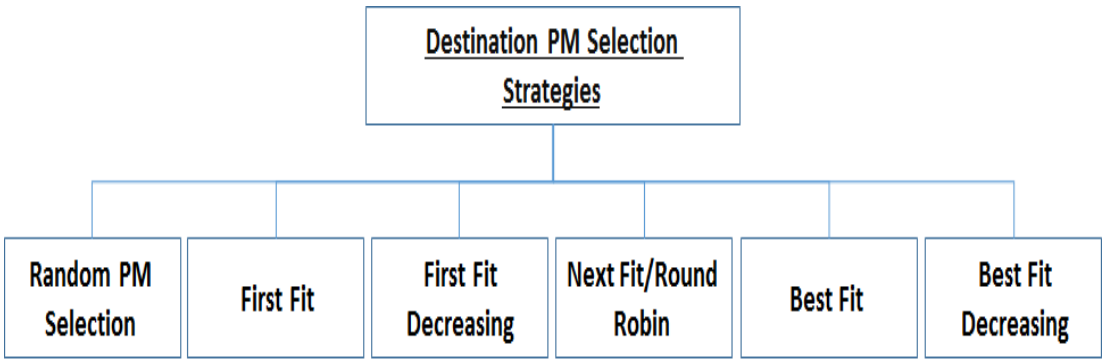


widely used in the proposed algorithms. In **Random Choice**, one VM from source PM is simply chosen for migration. In **Minimization of VM Migration**, the aim is to migrate the minimum number of VMs to make the current resource utilization of a PM lower than the upper utilization threshold to reduce the risk of occurring SLA violation. In **High Potential Growth** the VM with lowest ratio of actual resource usage to its initial claimed resource demand is selected. In **Minimization of Migration Time** each VM migration consumes energy and it is important to reduce the time of migration which is a proportion to the size of VM. Therefore, the VM which requires minimum time to complete the migration is selected. The estimated migration time is calculated based on the amount of RAM utilized by a VM divided by the network bandwidth available for the target PM. The last VM selection policy is **Maximum Correlation**. In this policy, the correlation between VMs is measured using techniques such as Multiple Correlation coefficient and VM with the highest correlation of the resource utilization with other VMs are selected.

One of the fundamental components in VM consolidation is Source PM Selection. There are totally two types of techniques called Threshold-Based and Threshold-Free. In threshold-based approach, an upper and lower threshold value is statistically defined to figure out when PMs are close to overloaded or underloaded states. Such PMs are selected to migrate their VMs in order to move towards the goals of balancing load or consolidating VMs. Threshold values are set either statically or in an adaptive way. In



the former, some static value is given while in the latter the value is determined dynamically based on parameters such as resource utilization, and etc. In Threshold-Free strategy, resource utilization ratio of the PMs is not compared against any threshold value to identify the PM as overloaded or underloaded. Instead, the source PMs are selected either randomly through applying some function. Another fundamental com-



ponent in VM consolidation is Destination PM Selection strategy. The first strategy is **Random PM Selection** which is applicable for both distributed and centralized VM consolidation architectures. As the name indicates one PM randomly is chosen as destination PM. All the rest strategies require the whole knowledge of PMs and thus are suitable for centralized approach. In **First Fit** policy, PMs are ordered in a sequence and for each VM, the first available PM from the ordered list of PMs is selected. **First Fit Decreasing** policy sorts VMs in decreasing order of resource demand and then the destination PM for the first VM with highest resource demand is chosen. In **Next Fit or Round Robin**, PMs are ordered in a sequence and for every single VM, the searching of destination PM always starts from the PM that is next in the list to the PM that placed the last VM. In **Best Fit**, the PM with the minimum residual resource

is selected as its destination PM. Last, in **Best Fit Decreasing**, VMs are first sorted in the decreasing order based on their resource demand. Then the destination PM for the VM with highest resource demand is searched.

The dynamic VM consolidation has been well studied in the literature. However, most algorithms are centralized solutions and few are distributed. Since the contribution of this thesis is proposing a distributed VM consolidation algorithm, we start by describing the proposed distributed algorithms in the literature in more details followed by an overall explanation of the centralized ones. Our algorithm GLAP, is an unstructured gossip based P2P VM consolidation algorithm. We found a number of distributed consolidation algorithms on the basis of unstructured gossip based P2P overlay network as well as structured P2P networks in the literature which we explain and compare them with GLAP.

In [9], [91], and [20], the authors propose a scalable gossip-based protocol. They designed algorithms for a number of management objectives such as load balancing, energy efficiency, fair resource allocation, and service differentiation. At each time only one of the management objectives can be deployed. It is modeled as an optimization problem and is formulated as bin packing problem. The algorithm jointly allocates compute and network resources. However, the main problem of this work is that they miss to consider that resources are shared between various VMs in a PM. This is a challenging problem in which consolidating VMs without considering this fact may lead to SLA violation. Additionally, designing an efficient solution for this problem becomes even harder in distributed fashion. This is what is addressed efficiently in GLAP. The paper [90] evaluates design and implementation of a gossip based resource management system that is built upon the Open Stack [94] for managing an Infrastructure-as-a-Service (IaaS) cloud platform. A set of controllers are the building blocks that allocate resources to applications cooperatively to achieve an activated management objective.
In [12], the authors present a decentralized approach towards scalable and energy-efficient management of virtual machine (VM) instances for enterprise clouds. They organize PMs into a hypercube structured Peer-to-Peer overlay network. Each PM operates autonomously by applying a set of distributed load balancing rules and algorithms. In hypercube, each PM is directly connected to at most n neighbors, while

the maximum number of compute nodes is N = 2^n. The data center scales up and down as PMs are added and removed, according to the hypercube node join and node-departure algorithms. Each PM is able to connect to its immediate neighbors in a periodic manner to migrate VMs. It is a static threshold algorithm in that underutilized nodes attempt to shift their workload to their hypercube neighbors and switch off while over utilized nodes attempt to migrate a subset of their VM instances so as to reduce their power consumption which in turn may lead to SLA violations. Although using hypercube is a new approach for managing VMs in a distributed manner, the authors only take into account the power consumption of each VM which is obtained from CPU usage. However,they do not take into account the resource demand and capacity of VMs and PMs. Moreover, an efficient algorithm should consider multiple resources such as CPU, memory, and network while this work does not do it. In addition, the algorithm does not propose any way to handle SLA violation when consolidate a set of VMs into one PM.
Mastroianni et al. [10], proposed ecoCloud approach which is an algorithm for power-efficient VM consolidation. In ecoCloud, the placement and migration of VM instances are driven by probabilistic processes. When a VM request arrives, it is broadcast to all the PMs and they respond to the coordinator if they can accept the request based on a Bernoulli trial. EcoCloud is a static threshold based algorithm in that it determines two static low and high threshold values. It considers both the CPU and RAM utilizations and when resources utilization of a PM fall below the low threshold value, that PM requires to migrate its VMs to move to sleep mode and when the aggregation of resources utilization exceed the high threshold value, the PM requires to migrate some of its VMs to reduce the probability of SLA violation. EcoCloud enables load balancing decisions to be taken based on local information. Although migration decisions are made based on local information, it still relies on a central manager for the coordination of PMs and hence is not scalable. Compared to our algorithm, GLAP, EcoCloud is not scalable and since it is a threshold-based algorithm, it is not able to handle trade off between energy efficiency and SLA violation efficiently.

Yazir et al. [21] propose a two steps distributed resource management algorithm in cloud. In the first step, they employed a distributed architecture in order to decompose

resource management into independent tasks, each of which is performed by independent PM in a data center. In the second step, the independent PM carries out configurations in parallel through Multiple Criteria Decision Analysis using the PROMETHEE method. By doing this, the algorithm distributes migration decisions to each PM, but PMs should have global knowledge of all PMs as well as require a performance model of the application, and do not consider SLA performance. Authors in [18] propose a fully decentralized algorithm based on unstructured P2P overlay networks. The system employs a dynamic topology which is built by periodically and randomly exchanging neighborhood information among the PMs. In addition, it considers cost of migration but unlike our work it does not take into account load of VMs and does not ensure SLA well.

Bloglazov et al. [11] used a power-aware best fit decreasing algorithm for initial VM placements and upper lower threshold based consolidation algorithm to migrate VMs if violation of resource utilization occurs. They compared several methods for capturing dynamic workload of VMs to determine an appropriate upper threshold such as Median Absolute Deviation (MAD), Inter-quartile Range (IQR), and Robust Local Regression. However, it only consider CPU as resource and the most important it is a centralized approach. Secron [17] is a centralized and static threshold based algorithm to prevent CPUs host from reaching 100% utilization that leads to performance degradation. However, setting static thresholds are not efficient in which different types of applications may run on a host. Farahnakian et al. [16] used reinforcement technique for dynamic VM consolidation. However, they do not consider utilization dynamism of each VM but instead they look at the arrival and departure rate of VMs to make migration decisions. In addition it is a centralized approach and thus is not scalable. The same authors in [19] propose a utilization prediction aware algorithm for dynamic consolidation of VMs using K-nearest neighbor regression based model. However, it is centralized and needs a global manager to have knowledge of resources utilization of all VMs during time and hence is not scalable. Dougherty et al. [95], proposed a model driven approach for optimizing the configuration, energy consumption, and operating cost of cloud infrastructures. The algorithm achieves energy efficiency by using a shared queue containing booted and configured VM instances that can be rapidly provisioned.

There are a number of network-aware workload consolidation algorithms [96, 97, 98,

99]. The authors in [96] proposed a centralized and threshold-free dynamic virtual machine consolidation approach in order to minimizes active PMs as well as active switches. In fact, the algorithm aims to reduce the energy consumption that network switches impose. However, one problem of this algorithm is that it only considers CPU as resource and is not a multi-resource DVMC algorithm. In [97], the authors proposed a centralized network efficient DVMC algorithm. Like the previous algorithm, it only considers CPU but unlike that it uses an adaptive threshold-based technique. [98] is another network-aware DVMC algorithm. It is a centralized algorithm that only takes into account CPU as resource and leverages an adaptive threshold-based algorithm. However, the authors did not use and performance evaluation strategy.The work proposed in [99] is another network-aware dynamic virtual machine consolidation algorithm which is different from the previous works in the sense that it aims just to reduce network traffic. It is a centralized and threshold-free algorithm. This algorithm only considers network-bandwidth as resource and neglects other resources. Accordingly, it cannot provide an efficient SLA of end users.

Chapter 3

Peacock: Probe Based Scheduling of Jobs By Rotating Between Elastic Queues

3.1 Introduction

To run machine learning algorithms on petabytes of data with low latency, data analytics frameworks such as Apache Spark [71] increase the level of parallelism by breaking jobs into a large number of short tasks operating on different partitions of data, hence achieving latency in seconds or even milliseconds. Centralized techniques such as Mesos [34] and Yarn [45] are able to schedule jobs optimally by having near-perfect visibility of workers. However, with the growth of cluster sizes and workloads, the scheduling time becomes too long to reach this optimality. To cope with this problem, probe-based distributed techniques [27, 28, 29] reduce the scheduling time by tolerating a suboptimal result. The key idea is to gather information from a small random sample of the cluster worker nodes. These solutions typically sample at least two workers and place a probe into the queue of the least loaded worker. They are often augmented with techniques such as re-sampling, work stealing, or queue reordering to ameliorate the initial placement of probes.

However, the existing probe-based algorithms are not able to perform efficiently under workload fluctuation and jobs heterogeneity. In particular, they cannot improve scheduling decisions *continuously* and *deterministically* to mitigate the *Head-of-Line blocking*, i.e., placing shorter tasks behind longer tasks in queues, efficiently. Moreover, the overall completion time of a job is equal to the finish time of its last task. Due to the distributed and stateless nature of probe-based schedulers, the existing solutions are not able to reduce the variance of tasks completion time of each job that are placed on various workers to reduce job completion time. Moreover, the contiguous load balancing between workers taking into account job heterogeneity leads to the minimization of occasionally appearing the idle workers which result in better jobs completion time. The existing probe-based schedulers are lack of it due to the distributed nature of such algorithms.

To cope with the aforementioned problems, in this paper we propose Peacock, a new fully distributed probe-based scheduling framework. The main difference between Peacock and the existing probe-based schedulers is that Peacock replaces the probe random sampling and the unbounded or fixed-length worker-end queues with deterministic probe rotations and elastic queues, respectively. The rational behind this design decision is to provide a wider visibility of workers to probes. This leads to better scheduling decisions while preserving fast scheduling of jobs. Intuitively, in probe

sampling approach, the scheduler submits two probes per task to two randomly sampled workers and the worker with the least loaded queue is selected to place the probe. However, in the probe rotation approach, the scheduler submits only one probe per task to one randomly sampled worker and then the probe rotates between workers until it is placed in a worker. This probe rotation approach finds an underloaded worker better than probe sampling because probes traverse a higher number of workers.

Workers are organized into a ring and send probes to their neighbors at fixed intervals. A probe rotation lets a loaded worker delegates the execution of a probe to its successor on the ring. *Elastic queues* regulate the motion of probes between workers and lets a worker dynamically adjust its queue size to balance load between workers. By decreasing the queue size, workers are forced to move some of their probes and by increasing the queue size they avoid unnecessary motion of probes. More interestingly, between the time a probe is submitted to the scheduler until it runs on an arbitrary worker, it moves between workers, stays in some workers and then continues rotating until eventually it executes. Furthermore, Peacock is augmented with a probe reordering to handle the *Head-of-Line blocking* effectively. The probes of one job are annotated with an identical threshold time equals to the cluster average load at the time of scheduling. This threshold determines a soft maximum waiting time for probes that are scattered independently between workers to reduce the variance of job completion time.

We evaluate Peacock through both simulation and distributed experiments. We use traces from Google [26], Cloudera [57], and Yahoo! [58]. We compare Peacock against Sparrow [27] and Eagle [28], two state-of-the-art probe-based schedulers. The results show that Peacock outperforms Eagle and Sparrow in various cluster sizes and under various loads. In addition, we evaluate the sensitivity of Peacock to probe rotations and probe reordering.

3.2 Terminology

We refer to the *scheduling time* of a task as the period from the time the relevant job of the task is submitted to the scheduler until the time a corresponding probe of the task is placed on the queue of a worker. The *rotation time* of a task is the period from the time the probe of a task is marked to be rotated until the time the probe is placed on

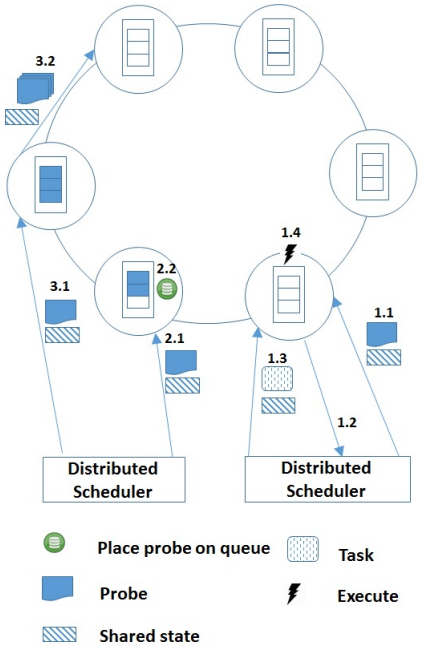


Figure 3.1: Different scenarios workers handle probes.

the queue of the neighbor. Since probes may be rotated several times, the rotation time is the sum of all rotations of the probe. The *queue time* is the total time the probe of one task waits in queue of one or more worker nodes. The *waiting time* of a task is the summation of *scheduling*, *rotation*, and *queue* times. We use the term *execution time* to describe the time the task spends executing on worker. The *task completion time* is the period a task is submitted to a scheduler until the time the task finishes. The *job completion time* is the period from the time the job is submitted to the scheduler until the time the **last** task of the job finishes. Accordingly, the *job completion time* is the aggregation of *waiting time* and the **latest** task completion time among all its tasks.

3.3 The Peacock Scheduler

Peacock comprises a large number of workers and a few schedulers. Each worker consists of two processes called the node monitor for running peacock and the executor for running the application. Workers shape a ring overlay network in that each worker connects to its successor and additionally stores descriptors to a few successors for fault tolerance purpose. Each scheduler connects to all workers. Schedulers manage the life-cycle of each job without exploiting expensive algorithms. Jobs are represented

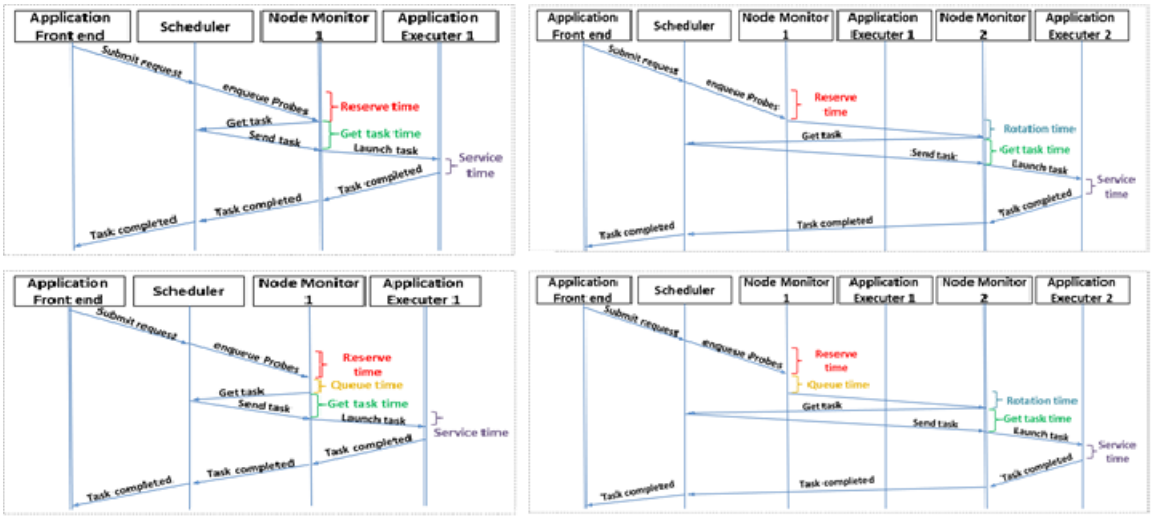


Figure 3.2: RPCs and timing associated with launching a Job. The top left figure is when worker is idle and the probe is executed immediately. The bottom left figure is When the worker is busy and the probe is queued for later execution. The top right figure is when the worker is busy, the probe cannot be queued and is rotated to the neighbor node. Finally, the bottom right figure is when the worker is busy and the probe is queued but at later time its place is given to new incoming probe(s) and hence is rotated to the neighbor worker.

as a directed acyclic graph (DAG), with tasks as vertices and data flow between tasks as edges. The DAG is divided into stages and Peacock considers each stage as a job and hence a DAG consists of a number of dependent jobs. Similar to [29, 28, 35, 31], Peacock needs to know the estimated task runtime of incoming jobs that is measured by the methods explained in [55, 35]. Jobs can be scheduled by any of the schedulers, however, all tasks of a job are scheduled by the same scheduler. When a scheduler receives a job, it submits probe messages to a number of random workers equals to the number of tasks. Each worker has a queue. As depicted in Figure 3.1, once a worker has received the probe, one of several options is possible:

(a) If the worker is idle (1.1), it requests the corresponding task of the probe from the scheduler (1.2) and the scheduler sends back the corresponding task data (source code) (1.3) and then the worker executes the task (1.4).

(b) If the worker is executing a task and its queue consists of a number of waiting probes like (2.1) and (3.1), the worker may enqueue the probe for the future execution or rotation (2.2).

(c) The worker may either rotate the incoming probe instantly or enqueue the probe and rotate other existing waiting probes (3.2).

Figure 3.2 represents sequence diagrams of different scenarios for remote procedure calls (RPCs) and timing associated with launching a Job.

3.3.1 Probe Rotation

In this section, we answer three important design questions:

(i) How should probes move between workers?

(ii) When should each worker rotate probes?

(iii) Which probes should each worker choose to rotate?

3.3.1.1 Ring Overlay Network

A challenging design decision is how probes move between workers. The easiest solution is that workers maintain a complete list of workers and send probe to a sampled

worker. However, it undermines the scalability and burdens some workers while some others might remain mostly idle. The efficient approach should be symmetric and balances load between workers, and maximize resource utilization. To this end, Peacock exploits a ring overlay network as depicted in Figure 3.1. We discuss whether exploiting a ring overlay network adversely impacts the scalability of Peacock. Peer-to-Peer overlay networks are extensively used to implement routing and lookup services [56]. In this respect, applying a ring overlay network with 1 in-out degree (i.e., 1 for in-degree and 1 for out-degree) in which lookup time grows linearly with the increment of ring size hampers scalability. However, there is no routing or lookup service in Peacock. It only rotates probes through a ring and typically probes are able to execute on any arbitrary worker node. Schedulers submit probes to sampled workers and probes are either rotated or stores at workers. Therefore, we can conclude that exploiting a ring overlay network does not undermine the scalability of the algorithm.

3.3.1.2 The Significance of Elastic Queues

Workers should decide when and which probes to rotate. Each worker utilizes one *elastic* queue, i.e., its size is adjusted dynamically. This elasticity is crucial for queues because it enables workers to rotate probes between themselves in order to distribute the probes uniformly. If queues are too short, the resources get under-utilized due to the existence of idle resources between allocations. If the queues are too long, then the load among workers gets imbalanced and job completion gets delayed. Determining a static queue size might lead to an excessive number of probe rotations when the cluster is heavily loaded and an inefficient reduction in the number of probe rotations when the cluster is lightly loaded. Peacock bounds queues using a pair (*<size, average load>*) which is called *shared state*. The size is calculated as the average number of current probes on cluster. The average load is calculated as the average estimation execution time of current probes on workers. This pair is adjusted dynamically to make queues resilient.

3.3.1.3 Shared State

A *Shared state* is a pair of information that are queue size and the average load of cluster (*<queue size, average load>*), and is changing from time to time since the cluster has a dynamic workload. Workers require to get the most recent *shared state*.

However, it is challenging to update the *shared state* of workers continuously in a decentralized manner. Peacock is designed in such a way that workers and schedulers are not strictly required to have an identical *shared state* all the time and hence workers may have different value of the *shared state* at times. Now, we describe how *shared state* is calculated and through what ways workers can get the latest value of the *shared state*. Each scheduler calculates the *shared state* continuously, based on the messages it receives. These messages are sent when a scheduler receives a job arrival event, receives a task finish event **or** receives an update messages from the other schedulers. For example, suppose the current aggregation load of cluster is $\langle 1500, 25000 \rangle$ (the number of probes, aggregation load) and a task finished event is received for a task with 20s estimated execution time. The scheduler updates the aggregation value to $\langle 1499, 24980 \rangle$ and sends asynchronously the message $\langle$- , 1 , 20$\rangle$ to the other schedulers. Upon receiving this message, the other schedulers update their aggregation value. Similarly, receiving a new job with 10 tasks and 15s estimated execution time changes the aggregation value to $\langle 1510, 25150 \rangle$, with update message $\langle$+ , 10 , 150$\rangle$ to the other schedulers. As an alternative solution, schedulers can manage the *shared state* through coordination services such as ZooKeeper. It eliminates direct communication between schedulers. Each scheduler calculates the value of the *shared state* through dividing aggregation value by the number of workers. Peacock does not impose extra messages to update the *shared state* of workers. The latest *shared state* is piggybacked by messages that workers and schedulers exchange for scheduling purposes. Figure 3.1 shows workers get *shared state* through three ways.

(i) When schedulers submit a probe message to workers

(ii) When schedulers send task data as a response of getting task by worker

(iii) When workers rotate probes to their neighbors.

3.3.1.4 Rotation Intervals

In the ring topology, workers rotate probes to their successor. Peacock rotates probes periodically in rounds. Once a probe has been selected for rotation, it is marked for rotation until the next round. In the next round, workers send all the marked probes in one message to their neighbors. Such design reduces the number of messages that workers exchange. Most jobs consist of a large number of probes and it is common

that in each round more than one probe of the same job are marked by the same worker to rotate. Peacock leverages this observation to remove the redundant information of such a subset of probes to reduce the size of messages. To reduce the number of messages, workers send rotation messages to their neighbors only if either there is/are probe(s) marked for rotation or when the *shared state* is updated from the last round. Interval between rounds is configurable from milliseconds to few seconds and it does not impact the job completion time since one probe is marked for rotation so that it does not wait in a long queue, otherwise it means that there is no reason for rotation of that probe.

3.3.2 Probes Reordering

It is crucial to reduce the variance of probe queuing time of one job since job completion time is affected by the last executed task of the job. It is also challenging since the probes of a job are distributed on different workers. However, the addition of the probes to queues in FIFO order (i.e., in the order in which they arrive) does not decrease the queuing time variance in the presence of heterogeneous jobs and workloads. Probe reordering is a solution to this problem [35, 28]. Reordering algorithms should be starvation-free, i.e., no probe should starve due to the existence of a infinite sequence of probes with higher priority. To this end, we propose a novel probe reordering algorithm. The algorithm performs collaboratively along with the probe rotation algorithm to mitigate the *Head-of-Line blocking*. Since probes rotate between workers, the algorithm cannot rely on the FIFO ordering of queues. Assume a scheduler submits probe p_1 to worker n_1 at time t_1 and probe p_2 to worker n_2 at time t_2. Then, n_1 rotates p_1 and reaches n_2 at time t_3. The problem is that p_1 is placed after p_2 in the queue of n_2 while it has been scheduled earlier. To overcome this problem, schedulers label job arrival times on probe messages so that workers place incoming probes into queues w.r.t the job arrival time. Then, schedulers attach task runtime estimation to probe messages. Once a worker has received a probe, it orders probes by giving priority to the probe with the shortest estimated runtime. While it reduces the *Head-of-Line blocking*, it may lead to starvation of long probes. To cope with this problem, schedulers attach a threshold value to all the probes of a job at arrival time. The value is the sum of the current time and the average execution time extracted from the current *shared state*. For example, if one job arrives at $t1$ and the *shared state* value is 10s, the threshold

will be $t1$ + 10 for all probes of that job. This threshold acts as a soft upper-bound to reduce tail latency and hence to reduce job completion time. It is starvation-free since probes do not allow other probes to bypass them after exceeding the threshold time and hence they eventually move to the head of queue and execute on worker.

We now present the algorithm. Workers receive a probe either because their predecessor rotates it along the ring or because the probe is submitted by a scheduler. Algorithm 1 depicts the procedure of enqueuing a probe and Table 3.1 explains the associated notations. Peacock maintains a sorted queue of waiting probes. Once a new probe has arrived, it is treated with the lowest priority among all waiting probes (Line 2) and tries to improve its place in the queue by passing other probes. It starts comparing its arrival time with the lowest existing probe (Line 4). If the new probe has been scheduled later than the existing probe, bypassing is not allowed unless it reduces head-of-line blocking without leading to starvation of the comparing probe. Bypassing the new probe can mitigate the *Head-of-Line blocking* if the execution time of the new probe is less than for the existing probe. Such bypassing should not lead to the starvation of the passed probe that is checked through threshold. If the threshold of the existing probe has not exceeded in advance or will not exceed due to bypassing, then the new probe can bypass the existing probe. Otherwise, it is either simply enqueued or rotated to the neighbor worker on the ring (Lines 4-10). If the new probe has been scheduled earlier, the existing probe has less execution time and the new probe does not exceed the threshold, then the new probe cannot bypass (Lines 11-16). Finally, the new probe waits in the queue if it does not violate starvation conditions, otherwise it is marked to be rotated in the next coming round (Lines 25-31). Once the process of enqueuing the probe has finished, peacock checks the shared state of the worker and may rotate one or more probes if needed (Lines 21-23).

3.3.2.1 Example

Figure 4.1 illustrates how a worker handles the queue when it receives probes. The corresponding notations are explained in Table 3.1. For the sake of simplicity, the impact of running tasks is ignored. The left-most queue shows the status of the worker at time 8, at which a new probe arrives $\{$(7,5,42),(6,30)$\}$. The triple represents the probe and the pair shows shared state. The incoming probe bypasses the last three waiting probes. Since it has more execution time and has been scheduled later than

Table 3.1: List of notations

Symbol	Description	Symbol	Description
ϕ	Queue Size	ω	Max threshold waiting probes
τ	Current time	μ	Max threshold waiting time for p
λ	Job arrival time	θ	runtime estimation of probe p
α	Total runtime of waiting probes	β	Arrival time probe p
γ	Waiting time estimation probe p	δ	Relict runtime of running task

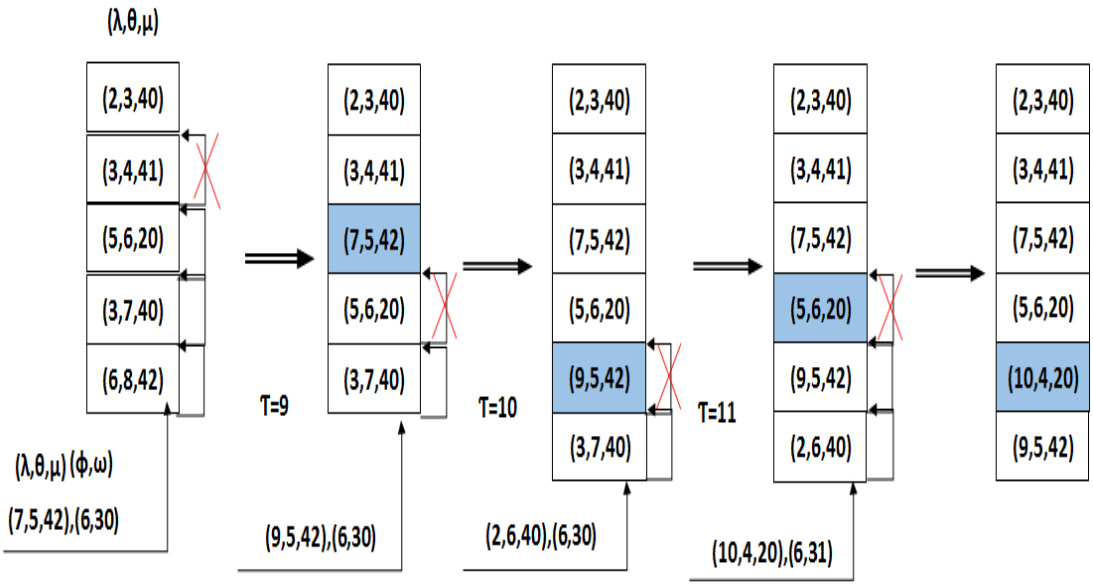


Figure 3.3: How a worker handles incoming probe(s).

Algorithm 1 Enqueue Probe submitted by scheduler or rotated by predecessor

```
1:  procedure ENQUEUEPROBE(p)
2:      γ_p ← δ + α
3:      for q in reversed waitingProbes do
4:          if λ_p ≥ λ_q then
5:              if θ_p ≤ θ_q AND λ_q + μ_q + θ_p ≤ τ then
6:                  γ_p = γ_p - θ_q
7:              else
8:                  PLACEORROTATE(p) ; decided = true ; break;
9:              end if
10:         else
11:             if θ_q ≤ θ_p AND τ + γ_p ≤ λ_p + μ_p then
12:                 PLACEORROTATE(p) ; decided = true ; break;
13:             else
14:                 γ_p = γ_p - θ_q
15:             end if
16:         end if
17:     end for
18:     if Not decided then
19:         waitingProbes.add(P , 0) ; α = α + θ_p
20:     end if
21:     while waitingProbes.size() ≥ φ OR α ≥ ω do
22:         q = waitingProbes.removeLast() ;α = α - θ_q ; rotatingProbes.add(q)
23:     end while
24: end procedure
25: procedure PLACEORROTATE(p)
26:     if τ + γ_p ≤ λ_p + μ_p OR λ_p + μ_p ≤ τ then
27:         waitingProbes.add(P) ; α = α + θ_p
28:     else
29:         rotatingProbes.add(p)
30:     end if
31: end procedure
```

the fourth probe (3,4,41), it cannot bypass more. The addition of new probe causes the overall waiting time to exceed threshold 30 and hence probe (6,8,42) gets deleted from the queue and marked for rotation. At time 9 the probe {(9,5,42),(6,30)} arrives and bypass the last waiting probe. Since threshold probe (5,6,20) is violated, it cannot bypass the probe even though it has less execution time. The addition of the probe does not violate shared state and hence no probe is marked for rotation. At time 10 the probe {(2,6,40),(6,30)} reaches and bypasses the last waiting probe. Since it has more execution time from probe (9,5,42) and stopping at this point does not violate its threshold, it does not bypass the probe even though it has been scheduled earlier. At time 11 probe {(10,4,20),(6,31)} arrives. The probe (10,4,20) has less execution time and has been scheduled later than the probe (2, 6, 40). Since bypassing does not cause the violation of the waiting probe threshold, it bypasses the last waiting probe. For the same reason, it bypasses the probe (9,5,42) as well. Although it is expected to bypass the probe (5,6,20), it stops since it violates the threshold (5,6,20). More precisely, the probe (5,6,20) is expected to start execution at time 5 + 20 = 25 but bypassing causes it to start at time 11 + 3 + 4 + 7 + 4 = 29 and hence it does not allow bypassing the probe (10,4,20). Moreover, probe (2,6,40) is marked for rotation since the addition of new probe violates queue size threshold.

3.4 Theoretical Analysis

We assume zero network and rotation delay, an infinitely large number of workers such that each worker runs one task at a time. Our experimental evaluation shows results in the absence of these assumptions. The first two theorems analyze probe rotation technique as global mechanism to mitigate *Head-of-Line blocking*. We examine the probability of placing all probes of a job on idle workers, or equivalently, providing zero wait time. Let r, y, and p denote the number of successful and unsuccessful probe rotations and the load of cluster respectively.

Theorem 1 *Probe rotation technique can place probes of a job with zero waiting time with probability* $Pr[Y_r > y] = \sum_{j=y+1}^{\infty} \binom{j+r-1}{j} p^r (1-p)^j$

Proof 1 *Given the above mentioned assumptions, we model probe rotation as a sequence of independent Bernoulli trials. We want to count the number* y *of unsuccessful rotations that lead to* r *placements in which* r *is the probe count (number of tasks) of*

a job.
Thus, the probability function of Y_r is:

$$Pr[Y_r = y] = \binom{y+r-1}{y} p^r (1-p)^y. \tag{3.1}$$

and then we have:

$$\begin{aligned} Pr[Y_r > y] &= \sum_{j=y+1}^{\infty} Pr(Y_r = j), \quad y = r, r+1, r+2, ... \\ &= \sum_{j=y+1}^{\infty} \binom{j+r-1}{j} p^r (1-p)^j. \end{aligned} \tag{3.2}$$

Fact 1 *(Maclaurin series)* $f(x) = f(0) + \frac{f'(0)}{1!}x + \frac{f''(0)}{2!}x^2 + \frac{f^{(3)}(0)}{3!}x^3 + \cdots$

Theorem 2 *Probe rotation technique unlike probe sampling is deterministic.* $Pr[Y_r > y] = 1$

Proof 2 *If there is at least r idle workers, we need to show that probe rotation technique eventually places all the probes with zero waiting time. In other words, in (2) all the probabilities sum to one.*
We derive the following:

$$\begin{aligned} \binom{y+r-1}{y} &= \frac{(y+r-1)(y+r-2)...(r+1)r}{y!}, \\ &= (-1)^y \frac{(-r)(-r-1)...(-r-(y-1))}{y!}, \\ &= (-1)^y \binom{-r}{y}. \end{aligned} \tag{3.3}$$

let $f(x) = (1-x)^{-r}$, by Maclaurin series expansion we obtain

$$
\begin{aligned}
(1-x)^{-r} &= 1 + rx + \frac{(r+1)r}{2!}x^2 + \frac{(r+2)(r+1)r}{3!}x^3 \\
&\quad + \cdots + \frac{(r+y-1)\cdots(r+1)r}{y!}x^y + \cdots, \\
&= 1 + (-1)^1(-r)x + (-1)^2\frac{(-r)(-r-1)}{2!}x^2 \\
&\quad + (-1)^3\frac{(-r)(-r-1)(-r-2)}{3!}x^3 + \cdots, \\
&\quad + (-1)^y\frac{(-r)(-r-1)\cdots(-r-y+1)}{y!}x^y + \cdots, \\
&= (-1)^0\binom{-r}{0}x^0 + (-1)^1\binom{-r}{1}x^1 + \\
&\quad (-1)^2\binom{-r}{2}x^2 + \cdots + (-1)^y\binom{-r}{y}x^y + \cdots, \\
&= \sum_{y=0}^{\infty}(-1)^y\binom{-r}{y}x^y.
\end{aligned}
\tag{3.4}
$$

Let $q = 1 - p$, by replacing the expression of Eq. 3 and then Eq. 4, we obtain

$$
\begin{aligned}
\sum_{y=0}^{\infty}\binom{y+r-1}{y}p^r\,q^y &= p^r\sum_{y=0}^{\infty}(-1)^y\binom{-r}{y}q^y \\
&= p^r\,(1-q)^{-r} \\
&= p^r p^{-r} \\
&= 1
\end{aligned}
\tag{3.5}
$$

We also demonstrate that probe reordering algorithm as local mechanism to mitigate *Head-of-Line blocking* is starvation-free.

Theorem 3 *Workload-aware probe reordering algorithm is starvation-free.*

Proof 3 *Each probe is annotated with a threshold value which equals the expected waiting time of that probe in the queue of any worker at the scheduling time. Probes might be executed before that time. Otherwise, by line 26 in Algorithm 1, we see that such probe not marked for rotation and remain in the current worker until execution. By lines 5 and 11, such probes do not allow bypassing any other probe to eventually comes to the head of the queue. Thus, even if we assume that the number of shorter*

probes scheduled after the probe p are infinite, eventually at some point probe p does not allow bypassing since it exceeds threshold.

3.5 Fault Tolerance Mechanism

We explain how Peacock handles the failure of workers and schedulers.

3.5.1 Maintaining ring when a worker crashes

Ring topology needs to be adapted upon worker failure. Inspired by the Chord algorithm [56] to handle the dynamism among joining and leaving nodes, Peacock's workers maintain a list of k successor workers to be able to tolerate k consecutive failures. Once a worker has crashed, the predecessor node of the failed worker directly connects to its successor node to stabilize the ring.

3.5.2 Recovering probes that are being executed or stored in the queue of the crashed worker

As the scheduler may loose track of the worker at which a probe has been rotated, this probe risks to never get executed after its worker failure. To solve this problem, Peacock exploits a maximum waiting time threshold μ assigned to each job. The scheduler waits for $\mu + \theta + t$ times for the job to be completed. The parameter t is the extra time the scheduler waits after the threshold has passed to ensure that a failure happened very likely. θ is the runtime estimation of the probe. If it is not complete within this time bound, the scheduler resubmits the probe that has not been notified yet. The value of μ for new job submission is set to the current time to give the highest priority. However, such uncertain decision making may occasionally result in the existence of more than one probe per task. Thanks to the architecture of Peacock, as all probes of a job are handled by one scheduler, a Noop message is sent to the worker as response to the request of getting task for any redundant probe. The shared state is treated like normal jobs. The exceptional case is for tasks that are being executed when worker crashes. As scheduler knows in which worker and when the task is started, it waits for a time (estimation time + start time + delay) to get the task finish event after which it responds Noop for possible similar probes. If it does not receive the finish event, it checks the

availability of the worker directly and may resubmit the missed probe. Hence, Peacock ensures exactly once semantic in the presence of workers failure. It provides a more precise fault-tolerance mechanism than Sparrow in which tasks need to tolerate at least once semantic of scheduler.

3.5.3 Handling scheduler failure

The frontends maintain a list of schedulers and randomly choose one of the schedulers to submit jobs. Schedulers operate independently so that the failure of one scheduler does not disrupt the operation of other schedulers. Since the number of scheduler nodes are limited, and each scheduler preserves information of its own ongoing scheduled jobs (such as tasks data, etc.), an appropriate approach is to manage schedulers failures by replication (e.g., factor 2).

Table 3.2: Workloads general properties

Workloads	Jobs Count	Tasks Count	Avg Task Duration
Google	504882	17800843	68
Yahoo	24262	992597	118
Cloudera	21030	5760714	162

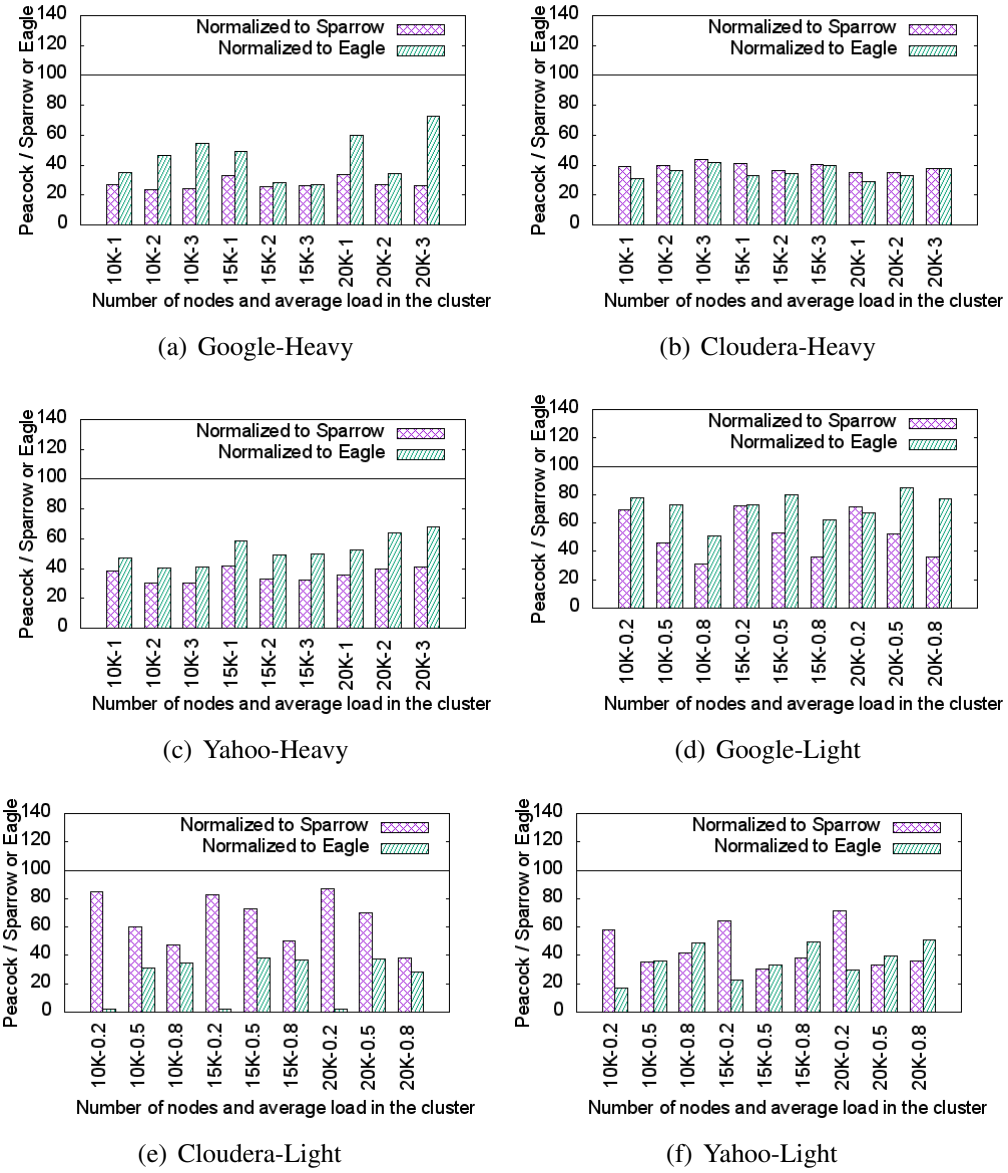
3.6 Evaluation Methodology

Comparison We compare Peacock against Sparrow [27] and Eagle [28], two state-of-the-art probe-based schedulers that use probe sampling. We evaluate the sensitivity of Peacock to probe rotation and probe reordering. We use both simulation for large scale clusters with 10k, 15k, and 20k workers and real implementation for 100 workers and 10 schedulers.

Environment We implemented an event-driven simulator and the three algorithms within it to fairly compare them for large scale cluster sizes. Also, we implemented Peacock as an independent component using Java and a plug-in for Spark [25] written in Scala. We used Sparrow and Eagle source codes for the distributed experiments.

Workloads We utilize workload traces of Google [26, 52], Cloudera [57] and Yahoo! [58]. Invalid jobs and tasks are removed from the Google trace and Table 3.2

gives the specification of the pruned traces. To generate various *average* cluster workloads, the job arrival time follows as Poisson process with a mean job inter-arrival time that is calculated based on expected average workload percentage, mean jobs execution time, and mean number of tasks per job. Since jobs are heterogeneous, workload becomes heterogeneous during the run with expected average percentage, too. We consider 20%, 50%, and 80% as light cluster workloads and 100%, 200%, and 300% as heavy cluster workloads. For distributed experiments, to keep it traceable, we sample



(a) Google-Heavy

(b) Cloudera-Heavy

(c) Yahoo-Heavy

(d) Google-Light

(e) Cloudera-Light

(f) Yahoo-Light

Figure 3.4: Average Job Completion times for heavy and light load scenarios.

3200 jobs of Google trace and convert task durations from seconds to milliseconds. We implemented a Spark job called *sleep task*. The current thread sleeps for a duration equals to the task duration to simulate the execution time that each task needs.

Parameters The estimated task runtime for a job is the average of its task durations. Each worker runs one task at a time which is analogous to having multi-slot workers, each served by a separate queue. The results are the average of runs. Error bars are ignored due to stable results of different runs. We set rotation interval to 1 s and network delay to 5 ms for simulation experiments. Eagle relies on several static parameters. For a fair comparison, we use the values used in the paper [28] even though any algorithm that relies on static values might be misleading under dynamic workloads.

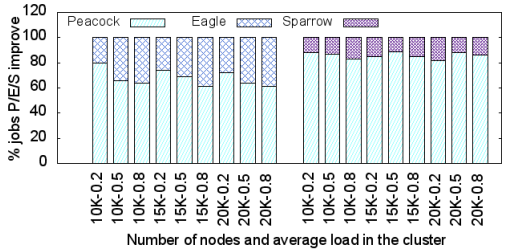
Performance Metrics We measure the average job completion times, cumulative distribution function of job completion times, and the fraction of jobs that each algorithm completes in shorter time to appraise how efficiently Peacock mitigates *Head-of-Line blocking*.

3.7 Experiments Results

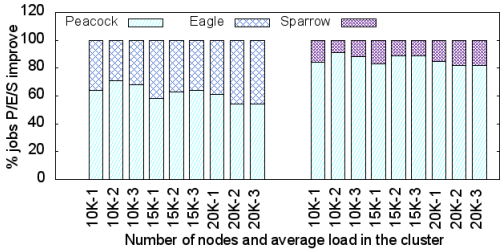
3.7.1 Comparing Peacock Against Sparrow

Figure 3.4 shows that Peacock achieves better average job completion times than Sparrow in all traces, loads and cluster sizes. Peacock outperforms in all loads in both traces but in heavy loads, such preference is more significant. The reason is that *Head-of-Line blocking* is reduced locally in each worker by reordering technique and collaboratively between workers by balancing the distribution of probes through both probes rotation and reordering. Under light loads, the improvement is mostly due to probes rotation and rarely due to the reordering. Furthermore, Sparrow only utilizes batch sampling and does not provide any technique to handle workload heterogeneity. Figure 3.6 shows that Peacock, unlike Sparrow, is job-aware in the sense that it reduces the variance of tasks completion times for each job. Beside probe rotation and reordering, the way that Peacock assigns threshold value for jobs is effective. Figure 3.5 shows that Peacock significantly outperforms Sparrow when comparing jobs individually. Under the 20% load, Sparrow shows better percentage than other loads because two samplings in Sparrow get empty slots faster than one sampling of Peacock even

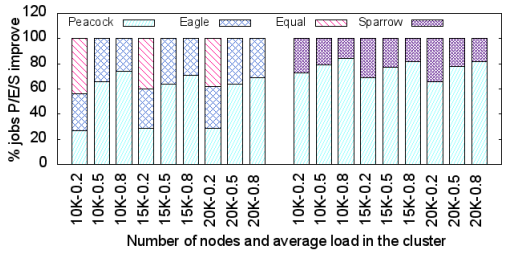
though probe rotation causes Peacock to outperform Sparrow under all loads. To provide further detail, Figure 3.4 shows that Peacock executes jobs in average between 13% to 77% faster than Sparrow in all settings. Figure 3.6(b) shows under the 50% load that Sparrow only completes 2.2% jobs in less than 100 seconds while Peacock completes 21.6% jobs within the same time. In Figure 3.6(a) and under load 300% Sparrow executes 0.3% jobs in less than 100 seconds while it is 31.8% for Peacock. Since the Yahoo! trace has longer task durations, we check for 1000 seconds where for 50% load (fig 3.6(d)), the percentages are in order 5% and 23.5% for Sparrow and Peacock. Figure 3.5 shows Peacock executes between 66% to 91% of jobs faster than Sparrow.



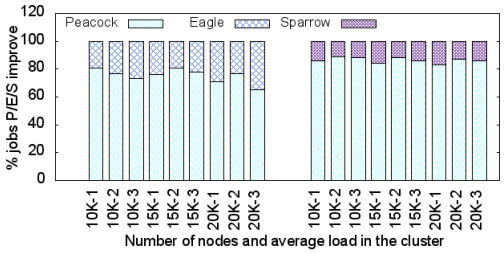
(a) Yahoo-Light



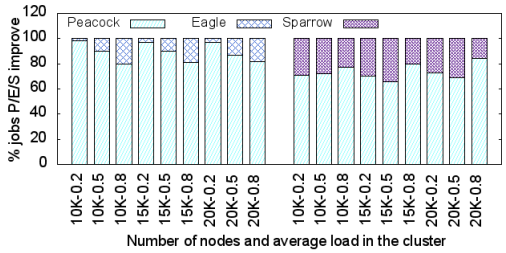
(b) Yahoo-Heavy



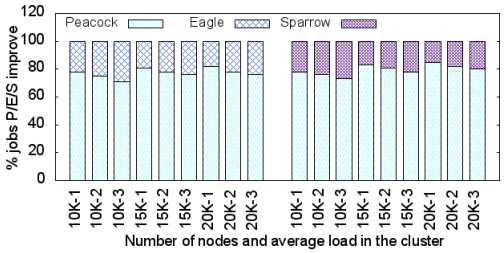
(c) Google-Light



(d) Google-Heavy

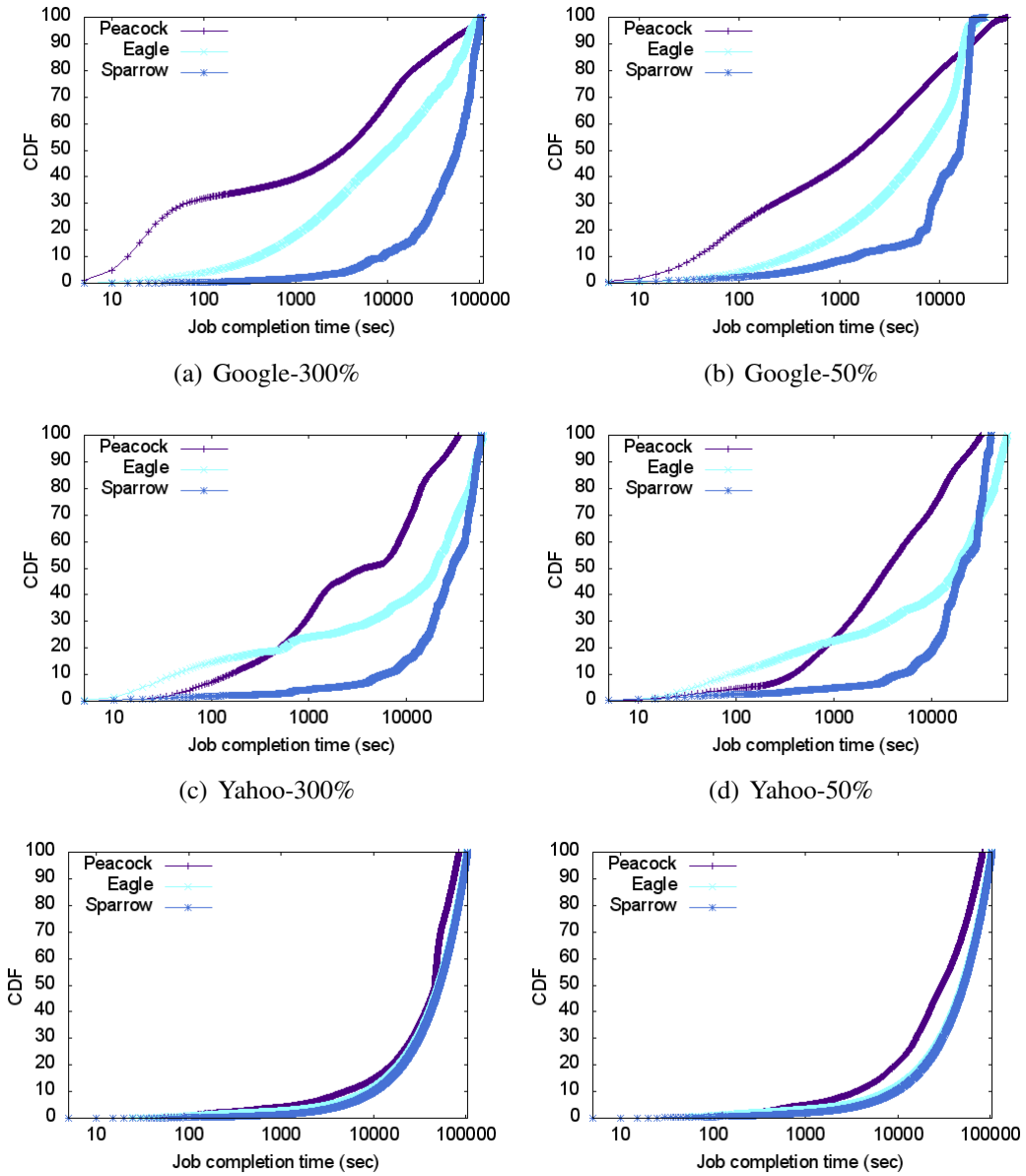


(e) Cloudera-Light



(f) Cloudera-Heavy

Figure 3.5: Fraction of jobs with shorter completion time.



(a) Google-300%

(b) Google-50%

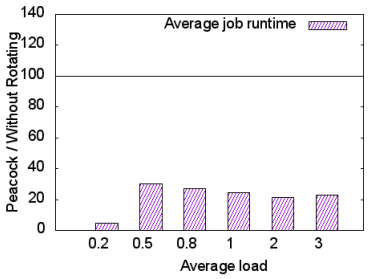
(c) Yahoo-300%

(d) Yahoo-50%

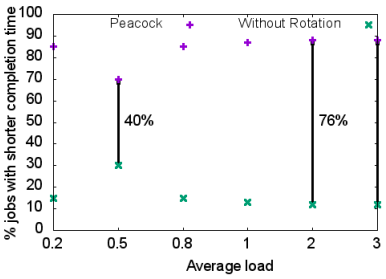
(e) Cloudera-300%

(f) Cloudera-50%

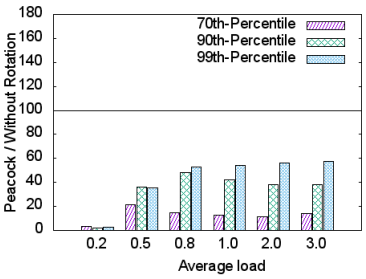
Figure 3.6: Cumulative distribution function of Jobs completion times.10000 workers.



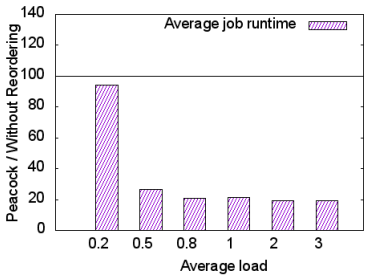
(a) Average job completion times (Peacock vs without rotating)



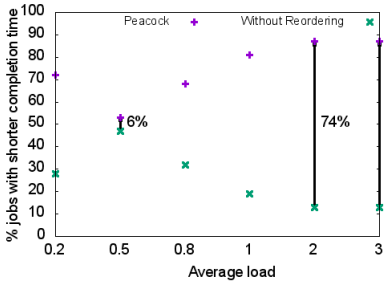
(b) Fraction jobs with better completion time(Peacock vs without rotating)



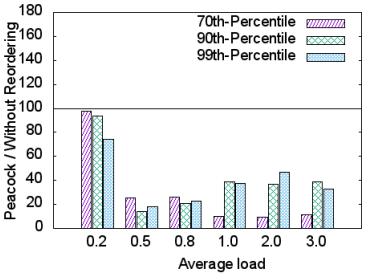
(c) Percentiles completion times (Peacock vs without rotating)



(d) Average job completion times (Peacock vs without reordering)



(e) Fraction jobs with better completion time(Peacock vs without reordering)



(f) Percentiles completion times (Peacock vs without reordering)

Figure 3.7: Peacock compared to without either probes rotation or probes reordering through Google trace over 10000 nodes.

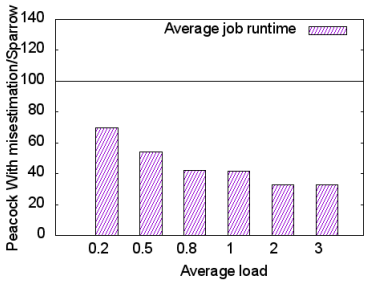
3.7.2 Comparing Peacock Against Eagle

Eagle is a hybrid probe-based sampling scheduler which augments Sparrow by borrowing batch sampling and late binding techniques. Eagle divides jobs statically into two categories of long and short jobs. A centralized node schedules long jobs and a set of independent schedulers using batch sampling schedule short jobs. The cluster is divided into two partitions, the smaller one is dedicated to short jobs and the bigger one is shared for all jobs. Eagle mitigates *Head-of-Line blocking* using re-sampling technique and a static threshold-based queue reordering.

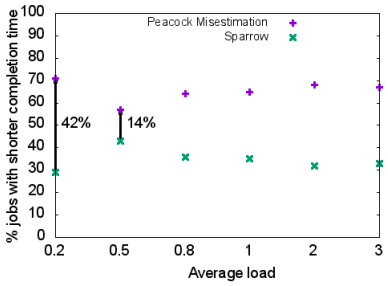
Figure 3.4 shows that Peacock outperforms Eagle in average jobs completion times in all traces and loads. Peacock completes execution of jobs in average 16% to 73% in Google, 32% to 83% in Yahoo!, and 65% to 78% in Cloudera faster than Eagle. Eagle unlike Peacock, relies on several static configuration values, which is misleading (e.g., for reordering threshold, segregation of long-short jobs, the number of retries for re-sampling of each probe, etc). However, three workloads have different properties and differ in terms of average task durations and average number of tasks in each job. Eagle is not practical and cannot execute efficiently various types of workloads as its static parameters are not workload aware. For example, Cloudera has a relatively larger number of tasks per job with longer average task runtime and in contrast, Google has a lower number of tasks per job with a shorter average task runtime. In light load we observe an amazing result for 20% load with 78% improvement for Peacock while it is much less improvement for Google. However, by playing with several static parameters in Eagle, we may get a bit better result for Cloudera but worse outcome for Google. Actually, workload-aware reordering together with probe rotation technique causes Peacock to outperform Eagle. More importantly, Peacock does not depend on any static parameter and intrinsically is workload-aware, which makes it practical. In heavy load scenarios, we also see that Peacock outperforms Eagle in all traces. In such scenarios, due to long-length queues, the probe reordering works efficiently and probes are able to rotate to continuously find a better worker. In contrast, Eagle may put a large number of short tasks in the queues of only a few workers in small dedicated partitions.

Figure 3.5(c)(d) shows that Peacock executes between 54% to 82% of jobs faster than Eagle in Google trace. Interestingly, in Google, we see that for 20% load, a

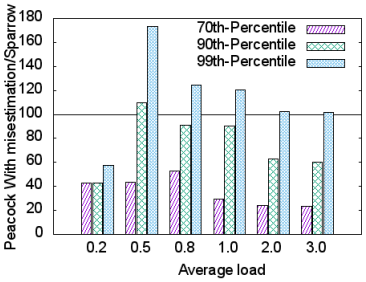
percentage of jobs have identical completion time in both Eagle and Peacock. Figure 3.5(a)(b) shows the result for Yahoo!, where Peacock executes 54% to 70% of jobs quicker than Eagle for all loads. Figure 3.6 shows, in Google, Peacock obviously executes a high percentage of jobs with lower latency than Eagle. In Yahoo!, while Eagle can run a small portion of jobs faster, Peacock shows an overall better performance because of its continuous probe rotation. We observe that Eagle can execute 16% jobs under 200 seconds while Peacock executes 11% of jobs at the same time. However, after that Peacock notably completes high percentage of jobs quicker than Eagle.



(a) Average Job completion time



(b) Fraction jobs with better completion time



(c) Percentiles Job completion time

Figure 3.8: Effect of inaccurate job runtime estimate. Peacock with inaccurate estimate compared to Sparrow for heavy and light loads of Google trace over 10000 nodes.

3.7.3 Sensitivity to Probe Rotation

We analyze the effectiveness of probe rotation on the performance of Peacock. Figure 3.7(a)(b)(c) indicates that the high performance of Peacock originates from the probe rotation technique on all loads. From Figure 3.7(a), we see the average job completion time negatively increases between 70% to 95% in all loads in comparison with

complete Peacock version because the probe rotation mitigates *Head-of-Line blocking*. Specifically, in light loads, probe rotation balances load between workers, which increases the cluster utilization and greatly reduces the formation of long-length queues. In heavy loads, due to the existence of long-length queues, Besides balancing the load between workers through probe reordering, Peacock utilizes probe rotation to mitigate *Head-of-Line blocking*. Figure 3.7(b) demonstrates that Peacock executes more fraction of jobs in less time than when it executes without probe rotation component. The highest difference is 78% for 200% load and the lowest is 47% for 50% load. Figure 3.7(c) shows that 70% and 90% percentiles in the high loads perform better than the same percentiles for the light loads. It indicates that in high loads probes reordering and probe rotation collaboratively mitigates *Head-of-Line blocking* while in light loads the performance of probes rotation is crucial as there is no long-length queues to apply probes reordering.

3.7.4 Sensitivity to Probe Reordering

Probe reordering is more influential when the cluster is under high load since workers have long-length queues when they are under high load. This is due to the novel starvation-free reordering algorithm that allows jobs to bypass longer jobs. The result in Figure 3.7(d) confirms this as the average job completion time for Peacock without its reordering component is close to the original Peacock for 20% load while by increasing load, we observe an increasing difference in average job completion time (The biggest difference is 81% for loads 200% and 300%). Figure 3.7(e) demonstrates that Peacock executes more fraction of jobs in less time than when it executes without probe reordering component. The largest difference is 74% for 200% load and the smallest is 6% for 50% load. From Figure 3.7(f) we can conclude that reordering causes most of jobs to be executed faster. It shows an improvement 90% in 70% percentile for loads 100%, 200%, and 300% while load 50% with 76% and 74% improvement has the best percentiles in 90% and 99%. As expected there is no significant difference for load 20% as there is no waiting probes in queues most of the time. It is obvious that the elimination of this component significantly increases the chance of happening *Head-of-Line blocking*.

3.7.5 Sensitivity to Imprecise Runtime Estimate

We evaluate the impact of mis-estimation of tasks on the performance of Peacock. To this end, we vary runtime estimation of each task by multiplying real runtime estimation to a uniformly random value in range [0 , 2] to create arbitrary both over-estimation and under-estimation times. This estimation times are used by scheduler, however, the actual task execution times remain unchanged. We only compare the results against Sparrow since Eagle also relies on task estimation times. Figure 3.8 shows that Peacock still outperforms Sparrow in spite of mis estimation of tasks length. For average job completion time (Figure 3.8(a)), load 20% with 30% improvement and load 300% with 68% enhancement are the lowest and the highest, respectively. Figure 3.8(b) reveals that in load 20% Peacock executes the largest number of jobs faster compared to Sparrow while the lowest is for load 50% with 14%. According to figure 3.8(c), Peacock with mis-estimation outperforms mostly Sparrow in 70% and 90% percentiles while in 99% percentile Sparrow shows an equal or better job completion time. Overall, the results confirm that Peacock is sufficiently robust to mis-estimation.

3.7.6 The Impact of Cluster Sizes and Loads on the Number of Probe Rotations

We investigate the average number of probe rotations per task for Google trace. We observe that by increasing the cluster size, the number of rotations decreases. For example, for 80% load, the number of rotations for 10K, 15K, and 20K nodes are 901, 656, and 513, respectively. Also, for higher loads, at 300%, the number of rotations are 1791, 1140, and 692 for 10K, 15K, and 20K, respectively. The larger the cluster size, the lower the number of redundant rotations. It indicates that probe rotation does not hurt the scalability and hence Peacock can be deployed on large scale clusters. Also, by increasing the load, there is a reduction in the number of rotations for all 3 cluster sizes. A heavier load leads to a higher number of rotations. For 10K the number of rotations are 17, 299, 689, 901, 1523, and 1791 for 20%, 50%, 80%, 100%, 200% and 300% loads, respectively.

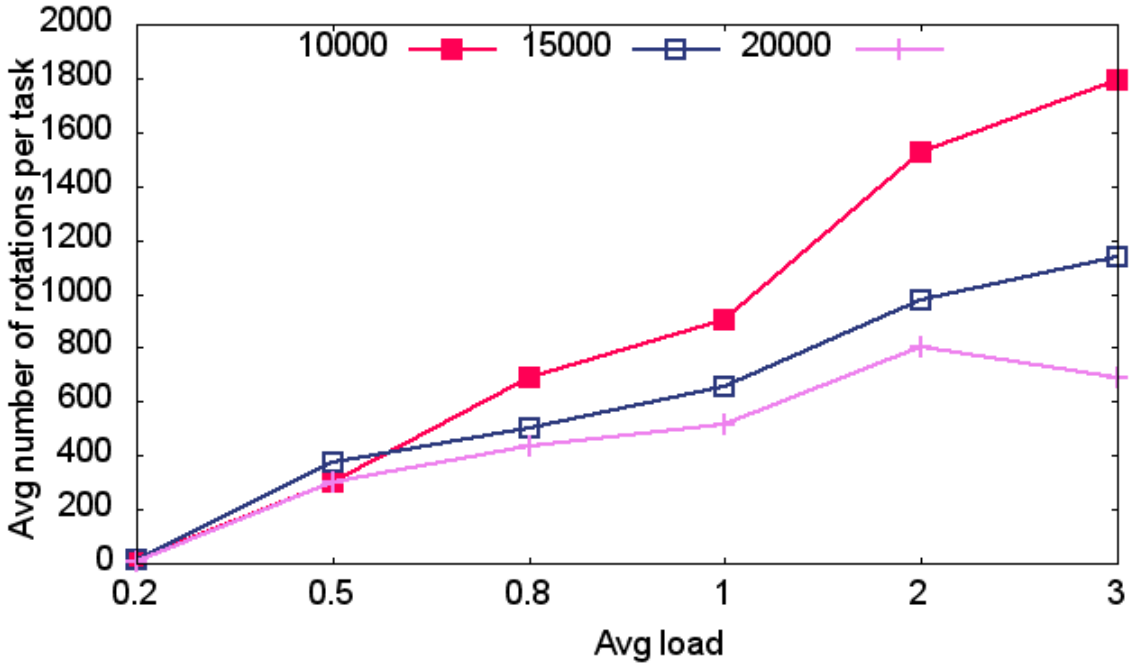
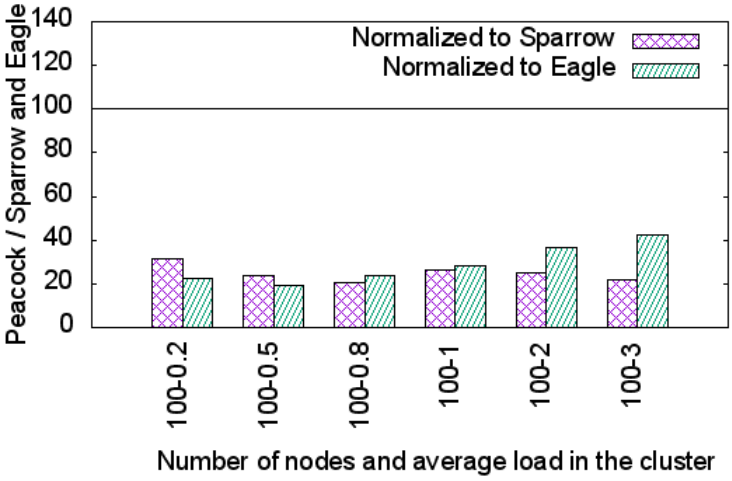


Figure 3.9: Google trace. Avg number of rotations per task

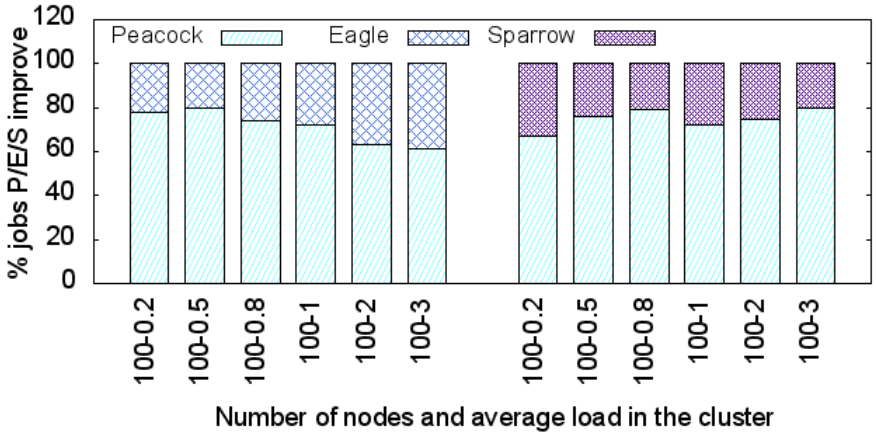
3.7.7 Distributed Experiments Results

We implemented Peacock as an independent component using Java and a plug-in for Spark [25] written in Scala. We ran experiments on 110 nodes consisting of 100 workers and 10 schedulers. We implemented a Spark job called *sleep task*. The current thread sleeps for a duration equals to task duration to simulate the execution time that each task needs. The method for varying the load is the same as the simulation experiments described in section 3.6. We run real implementations of Sparrow and Eagle with the same specifications to compare Peacock against them. In real cluster executions, algorithms are compared by adding various sources of cost that are not counted in simulation, including multi-threading efficiency, transport protocol overhead, local computing cost, network instability, and etc.

Figure 3.10(a) presents the average job completion time under both light and heavy loads. The result shows that Peacock significantly outperforms both the algorithms in all loads. Peacock outperforms Sparrow at most 80% improvements in 80% load and at least 69% improvement in 20% load scenarios. Moreover, compared to Eagle,



(a) Average Job completion time



(b) Fraction Jobs with shorter completion time

Figure 3.10: Distributed experiments for 3200 samples of Google trace for heavy and light workloads.

the maximum improvement reaches 81% when the load is 50% and the minimum improvement is 57% for the load 300%. Figure 3.10(b) shows the fraction of jobs that each algorithm runs in shorter time. Again we can see that Peacock runs much more percentage of jobs faster than both Sparrow and Eagle.

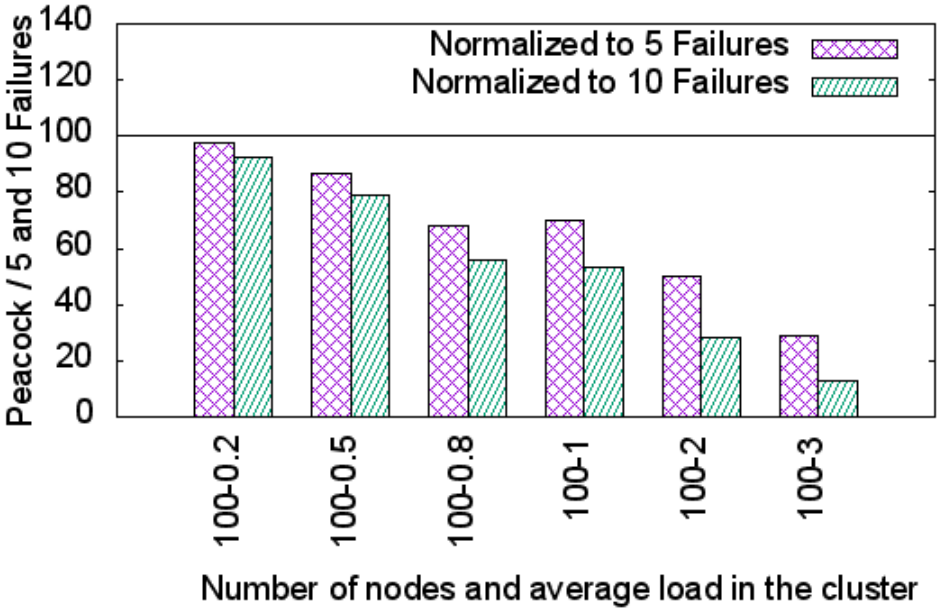


Figure 3.11: Average job completion time for Google trace. Peacock versus 5 and 10 failures at second 20

3.7.8 The Impact of Workers Failure on The Average Job Completion Time

In this experiment, we evaluate how much the failure of workers at the specific time affects the average job completion time. We ran these experiments on a cluster of 110 nodes including 100 workers and 10 schedulers for all heavy and light load scenarios. After 20 seconds, in two different configurations, we terminated manually 5 and 10 worker nodes out of 100 so that we can investigate how much it adversely impacts the average job completion time. More importantly, we could observe that the fault tolerance mechanism operates correctly and all involved probes at failed workers are rescheduled and executed completely. We verify at the end of the execution that all the jobs have been executed and we did not see any missed jobs. It proves that our fault

tolerance mechanism is able to detect missed probes automatically and in a probabilistic way and to reschedule them safely and correctly.
According to Figure 3.11, we can observe that by increasing the load the job completion time difference between failure and no failure execution rises and hence there are more number of missed probes recovered through the fault-tolerance mechanism. From figure 3.11 we can see that at load 20%, the job completion time increases 3% to 8% for 5 and 10 failures respectively. This difference increases for all loads and reaches to highest for load 300%, which is 71% to 87% for 5 and 10 workers failure respectively. Obviously, the reason for such difference is the formation of longer queues at workers due to the reduction of available resources in the cluster.

3.8 Conclusion

Today's data analytics frameworks divide jobs into many parallel tasks such that each task operates on a small partition of data in order to execute jobs with low latency. Such frameworks often rely on probe-based distributed schedulers to tackle the challenge of reducing the associated overhead. Unfortunately, the existing solutions do not perform efficiently under workload fluctuations and heterogeneous job durations. This is due to a problem called *Head-of-Line blocking*, i.e., short tasks are enqueued at workers behind longer tasks. To overcome this problem, we proposed Peacock, a new fully distributed probe-based scheduling method. Unlike the existing methods, Peacock introduces a novel probe rotation technique. Workers form a ring overlay network and rotate probes using elastic queues of workers. It is augmented by a novel starvation-free probe reordering algorithm executed by workers. We evaluated Peacock against two existing state-of-the-art distributed and hybrid probe based solutions through a trace driven simulation of up to 20,000 workers and a distributed experiment of 100 workers in Apache Spark under Google, Cloudera, and Yahoo! traces. Our large-scale performance results indicated that Peacock outperforms the state-of-the-art in all cluster sizes and loads. Our distributed experiments confirmed our simulation results.

Chapter 4

GLAP: Distributed Dynamic Workload Consolidation through Gossip-based Learning

4.1 Introduction

Dynamic virtual machine consolidation (DVMC) using live migration is one of the most promising solutions to reduce energy consumption in cloud data centers. It is the process of reducing the number of active physical machines (PMs) through virtual machine (VM) live migration to diminish energy consumption and improve resource utilization in data centers. Most of the solutions perform on the basis of initial resource requirements defined by VM types. However, VMs often utilize resources much less than their initial allocation. To maximize resource utilization efficiently, cloud providers tend to consolidate VMs based on resource demand. It may adversely impact the SLA because resource utilization of each VM on a PM varies over time [17] and thus PMs can become overloaded and SLA be violated, i.e., the aggregate resource demand from their VMs exceeds their capacity. It causes more VM migrations that in turn deteriorates SLA. To mitigate this deterioration, static and adaptive utilization threshold techniques were proposed [19, 14, 11]. In static techniques, each PM accepts migrating VMs until resource utilization reaches a threshold. However, this technique only considers the current state of VMs and neglects the varying demand of VMs. In adaptive algorithms, a historical trace of VMs resource utilization is monitored by a central PM to calculate a fixed threshold value to be used by all PMs. Yet, this approach is inefficient because the VMs workload patterns of PMs are different from one another and thus each PM requires a dedicated threshold value.

Fully distributed DVMC algorithms [9, 10, 12, 18, 21, 24] overcome the deficiencies of centralized [11, 16, 17] or hierarchical [22, 23] approaches. By having PMs making consolidation decisions based on a small part of the data center, they do not suffer from scalability and/or packing efficiency with the increasing number of PMs and VMs [18]. Unfortunately, none of the distributed DVMC solutions adapt to the varying load of VMs. To illustrate the problem, consider Figure 4.1 that depicts a data center with 4 PMs and 8 VMs and the relationships between PMs. In distributed solutions, PMs accept VMs up to a threshold, say 80%. At round t, PM4 communicates with PM1 and migrates VM7 and VM8 and switch off before the overlay network gets reconfigured. Similarly, PM3 migrates its VMs to PM2 and switches off. At round $t + 1$, the demand of VM2 and VM7 increases and PM1 becomes overloaded. However, PM1 cannot migrate any VM and remains in an overloaded state since its only

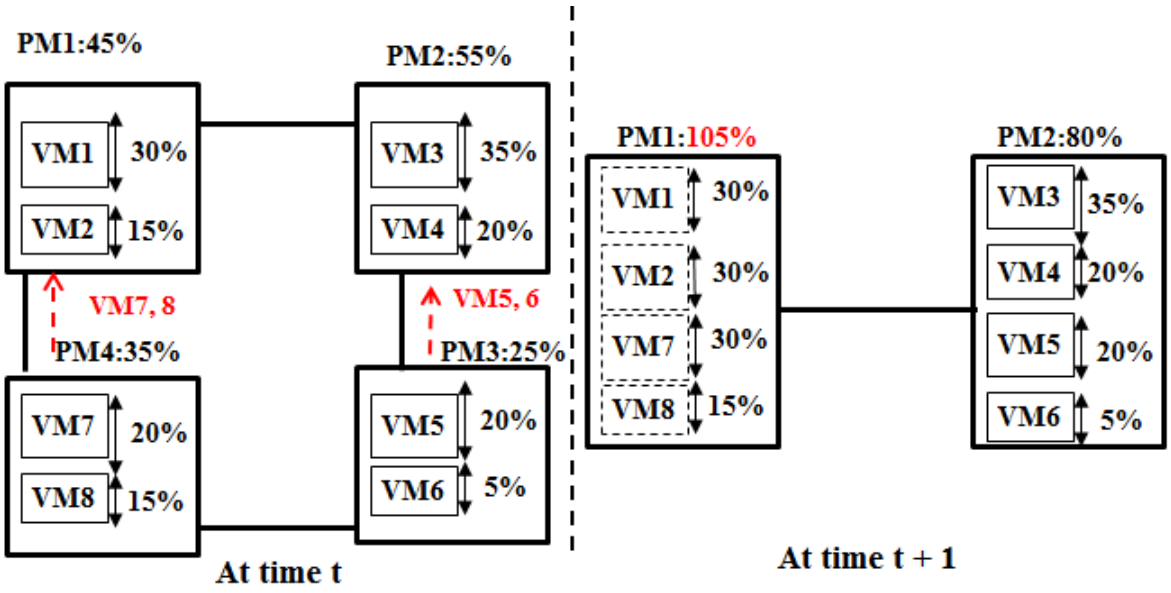


Figure 4.1: Previous distributed solutions could not cope with the varying load of VMs

neighbor (PM2) has reached the threshold. The load variation induces SLA violations. These distributed algorithms switch off a high number of PMs at the expense of overloading the most active PMs. In the previous example, one could migrate the VM by starting new PMs, however, this would tend to create several scattered under-utilized PMs which reduces the resource utilization efficiency.

Our contribution is GLAP (Gossip Learning Resource Allocation Protocol), the first fully distributed DVMC algorithm considering variable resource demand of VMs. The key idea is a gossip-based learning strategy to predict VM load variations. We devise an unstructured gossip-based protocol as well as a Q-Learning technique by which workload patterns of VMs are characterized. Using both techniques, each PM cooperates with its fellow PMs to improve its status to a new state incrementally moving towards the consolidation goals with minor SLA violations and without sacrificing scalability. To reduce the chance of SLA violation, we propose a threshold-free technique, using Q-Learning, that considers subsequent load state of VMs. Q-Learning consists of States (S), Actions (A), and Rewards (R). States are defined as calibrated PMs load state. An action is moving out/migrating any specific VM. Also, each VM

has a calibrated load state considering time-varying load of VMs. We design two reward systems one is to encourage PMs to migrate their VMs to switch off mode and the other assists PMs in accepting or rejecting migrating VMs to avoid moving to an overloaded state. In fact, on the basis of the PM state, GLAP predicts whether adding a specific VM would cause the PM to move to an overloaded state immediately or in the future. If so, it rejects the VM to ensure that the PM stays in a desirable state for a longer period of time. It is vital for all PMs to eventually own the identical set of Q-values as a global knowledge of the time-varying load of VMs to take appropriate consolidation decisions. Using a central manager to build Q-values becomes, however, a bottleneck. To preserve scalability, we also implement a novel two-phase (learning, aggregation) distributed learning protocol that computes the Q-values. In the learning phase, PMs locally calculate Q-values and in the aggregation phase, through a gossip-based protocol, PMs converge to their own unified values. Finally, through a gossip-based protocol, each PM periodically connects with one of its neighbors randomly and migrate its VMs according to Q-Learning system. This process is repeated continuously by all PMs to incrementally switch off unused PMs.

Finally, we conducted extensive experiments using the Google Cluster VMs traces [26]. To this end, we augmented an existing simulator of fully decentralized systems, PeerSim [2]. Under various workloads, we compared GLAP with two well-known distributed consolidation protocols, GRMP [9] and EcoCloud [10], and a centralized consolidation protocol, PABFD [11]. The results show that GLAP, as an autonomous distributed DVMC, addresses the aforementioned problem by not sacrificing SLA for reducing the number of active PMs. Our results show that GLAP reduces the number of overloaded PMs in EcoCloud, GRMP and PABFD by 43%, 78% and 73%, respectively.

4.2 Background

Reinforcement Learning is a type of Machine Learning which allows machines and software agents to maximize their performance by automatically determining the ideal behavior within a specific context. Simple reward feedback is required for the agent to learn its behavior. More precisely, Reinforcement Learning allows the machine or software agent to learn its behavior based on the feedback from the environment. This behavior can be learnt once and for all, or keep on adapting as time goes by.

Q-Learning is a reinforcement learning technique employed in many research areas [15]. It is used to find an optimal action-selection policy for any given (finite) Markov decision process (MDP). It attains an optimal policy by learning an action-value function that finally gives the expected utility of taking a given action in a given state. The strength of Q-learning is that, unlike MDP, it does not require a model or prior knowledge of the environment. More precisely, it consists of an agent, states S, a set of actions per state A, and a reward system R. The Q-Learning runs several error-and-trial iterations with a dynamic environment to obtain the optimal solution. After each iteration the Q-value is updated according to the following formula:

$$\begin{aligned} Q_{t+1}(s_t, a_t) &= (1-\alpha)Q_t(s_t, a_t) \\ &\quad +\alpha\big(R + \gamma * max_a\, Q_t(s_{t+1}, a_t)\big) \end{aligned}$$

where R is the reward observed after performing action a_t in s_t and γ is a discount factor that determines the importance of future rewards. A factor of zero causes the agent to only consider the current rewards, while a factor approaching one makes it strive for a long-term high reward. The learning rate α is the rate at which the new information overwrites the old one. It takes a value between zero and one such that a value one indicates a deterministic action in the sense that it only considers the latest value while a value between zero and one shows a stochastic behavior taking into account both the latest action and all the previous taken actions for the current state. We model cloud data center through Q-learning by defining states and actions as well as two novel reward systems to make decisions for efficient VM's live migrations which leads to an energy efficient consolidation of virtual machines.

4.3 System Model

We model a cloud data center as a set of N machines that are interconnected by a data center network. We deem that each PM has a CPU, Memory, and Network interface. The software layer of the system comprises of four components which are deployed on any participating PM (see Figure 4.2). VM monitor (VMM) is in charge of profiling total resources utilization of PM as well as monitoring variable resources demand of VM and resizing VMs according to their resource needs. Further, it serves GLAP components by sharing PM and VMs information as requested. Our protocol, GLAP, comprises of three components called Cyclon, Gossip Learning, and Gossip Consolidation. Gossip Learning component has two duties. Firstly, it executes an algorithm to build Q-values locally and secondly, through a gossip process, it ensures that PMs own identical Q-values. Finally, Gossip Consolidation component performs consolidation through gossip protocol. It decides when and which VMs should be migrated out and also decides when and which VMs suggested by other PMs should be accepted to be migrated in. As it can be seen, there is no centralized PM in the architecture of our data center.

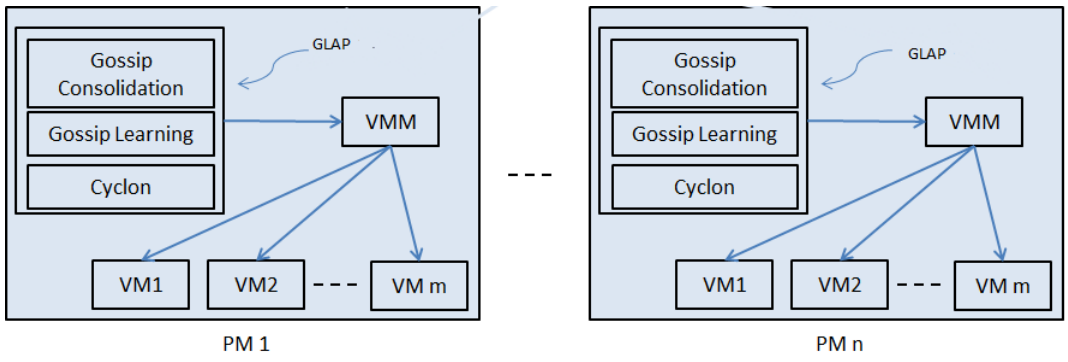


Figure 4.2: System Architecture

4.4 Gossip-Based Solution

We describe the algorithm in three sections. We explain how we model Q-Learning components including states, actions, and two reward systems. Then, we propose a novel two phase algorithm by which PMs train and unify Q-values in a decentralized manner. Finally, we propose distributed DVMC algorithm in which PMs using the Q-values consolidate VMs.

4.4.1 Construction and usage of Q-Learning in Cloud

A usual MDP model requires 4-tuple input (States (S), Actions (A), Transition Probabilities (P), and Rewards (R)). However, since Q-Learning is model free, knowing P is not required. We define two sets of Q-values, one is used for moving out VMs and the other for making decision to either accept or reject migrating VMs. We devote the former by ϕ_p^{out} and the latter by ϕ_p^{in} for node p.

States(S) and Actions(A):

We define states as PMs load state (PM-State) and actions as VMs load state (VM-State) which can be thought of as migration of a VM in a certain state. In cloud environment, there are different resources (CPU, Memory, IO, Network), each with variable and different workload demand. Thus, workload PMs and VMs are multi-attribute. We calibrate states and actions on the basis of average resource utilization degree of PMs and VMs respectively in order to limit to a finite number of states and actions. Consider a set of *n* resources $M = \{m_1, m_2, m_3, ..., m_n\}$ in the cloud system and L resource utilization level $L = \{l_1, l_2, l_3, ..., l_n\}$. The maximum total number of states of PMs and VMs are the same and is equal to the Cartesian product of the two sets $|L| \times |M|$. For example, if $M = \{$*CPU, Memory*$\}$ and $L = \{$*High, Medium, Low*$\}$, total number of possible states is $3^2 = 9$.

We consider 2 resources (CPU and Memory) and 9 resource utilization levels with thresholds used to distinguish between them. $x_p^{au}(t)$ indicates to the average utilization

x at time t.

$$S, A = \begin{cases} \textit{Low} & x_p^{au}(t) \leq 0.2 \\ \textit{Medium} & 0.2 < x_p^{au}(t) \leq 0.4 \\ \textit{High} & 0.4 < x_p^{au}(t) \leq 0.5 \\ \textit{xHigh} & 0.5 < x_p^{au}(t) \leq 0.6 \\ \textit{2xHigh} & 0.6 < x_p^{au}(t) \leq 0.7 \\ \textit{3xHigh} & 0.7 < x_p^{au}(t) \leq 0.8 \\ \textit{4xHigh} & 0.8 < x_p^{au}(t) \leq 0.9 \\ \textit{5xHigh} & 0.9 < x_p^{au}(t) < 1 \\ \textit{Overload} & x_p^{au}(t) = 1 \end{cases}$$

For example, if there is a VM with average CPU and memory demand 0.85 and 0.56 respectively, then it indicates an action (*4xHigh, xHigh*). If we assume that the PM includes another VM with specification 0.1 and 0.2 then the PM's state is measured as an aggregation of average resource utilization of VMs and equals to (*5xHigh, 3xHigh*).

Reward (R): Rewards are incentives that are given to a PM after performing an action a $\in$ A (VM live migration). In fact, rewards are given to PMs to persuade them to perform VMs migrations to consolidate VMs to as few PMs as possible. To this end, we assign rewards for doing actions. PMs are acting as either sender to migrate their own VMs to switch to sleep mode or recipient to decide whether to accept or reject the suggested VM. Therefore, we design two different incentive reward systems called reward *out* and reward *in*.

Reward out: In sender mode, if the state is overload, the reward system encourages the PM to move from heavily loaded state to a lightly loaded state to eliminate SLA violation such that it imposes minimum number of migrations. While a PM with any other state should migrate its VMs to switch to sleep mode. Usually the PM with lower resource utilization migrates its VMs to the other PM so that it can switch off with less number of migrations. However, often a PM can migrate a subset of VMs due to the filled capacity of destination PM. Thus, the sender remains active and may become a recipient when deals with another PM. Such fact leads to the several filling and emptying of PMs which results in redundant number of migrations. To mitigate this, the reward system encourages PMs to aggressively reduce their resource utilization so that they move to sleep mode earlier. Therefore, any transition to a state with a lower resource utilization is given higher reward. Let $R_{out} = \{r_L, r_M, r_H, ..., r_O\}\, \forall r \in R_{out}\ ,\ r > 0$

be a set of reward values for states when PM is sender then:

$$r_L > r_M > r_H > ... > r_O$$

Reward in: In recipient mode, the important factor is preventing SLA violations. It occurs when a PM moves to an overload state after acceptance of VM. On the other hand, PMs should be avaricious and accept as many VMs as possible to be able to maximize their resource utilization. However, this in turn increases the probability of moving to an overload state. To capture such contradiction, we divide level state of PMs and VMs into several smaller scales as shown before. We give a positive reward to PM for any action (live migration) that transit the state of PM to a state towards overload state (but not overload state itself). However, transition to an overload state, given a negative reward. The final Q-value of state and action is the resultant of several training sessions according to the formula 4.1. If the Q-value of (s_i , a_1) is less than zero, the suggested VM is rejected otherwise accepted. More precisely, if the Q-value is negative, accepting action a_1 (VM) when PM is in state s_i, very likely ends in an overload state immediately or in the near future and thus should be avoided. In fact, the smaller negative reward value, the less probability of producing SLA violations and inefficient resource utilization.

Let R_{in} = $\{r_L, r_M, r_H, ..., r_O\}$ be a set of reward values for states when PM is in destination mode then:

$$\{r_L, r_M, r_H, ..., r_{5xH} > 0 \, ; r_O \ll 0\}$$

Note for both *out* and *in*, the total reward of any transition from s to $\acute{s}$ is aggregation rewards of each resource.

Optimal Action Selection: The optimal action determination finds an action for each certain state to maximize the cumulative function of expected rewards. We define two action determination functions for *out* and *in* Q-values. Let π_{out} denotes the function returns the most suitable available VM when PM *p* is in sender mode (i.e., the greatest value among available actions for state s_p from ϕ_p^{out}, to move towards a lightly loaded state or to move towards emptying the PM). $V_p(t)$ is a set of available VMs within PM

p at time t.

$$\pi_{out}(s_p(t)) = \arg\max_{a} (\phi_p^{out}(s_p, a)), a \in V_p(t)$$

Assume q to be a destination PM with current state $s_q(t)$ and let π_{in} denotes the function to decide whether to accept or reject action a. This function rejects action a if it finds that very likely PM q moves to an overloaded state in the future after this transition. This is achieved by looking at ϕ_q^{in} and if the Q-value of certain state and action is less than zero.

$$\pi_{in}(a) = \begin{cases} 1 & ; \phi_q^{in}(s_q, a) \geq 0 \\ -1 & ; \phi_q^{in}(s_q, a) < 0 \end{cases}$$

4.4.2 Gossip Learning Component

To achieve the consolidation goals, it is vital for the algorithm to make efficient live migration decisions which among other factors strongly depends on characterizing workload variation of VMs. In other words, for any state-action pair, we should carefully evaluate and quantify how worth it is to do a certain action (VM live migration) when PM is in a specific load state. Technically, we need a component to calculate Q-values in our Q-Learning algorithm with respect to the reward systems that we have explained in the previous section. Although DVMC is a continuous service, the learning process does not need to execute endlessly. In fact, according to Figure 4.2, Cyclon and workload consolidation components run continuously while learning component runs as required by a predefined policy e.g., if the arrival and departure rates of VMs exceed a threshold compared to the last learning time or based on a fixed time interval.

A naive approach is to utilize a dedicated server. However, it is in contradiction with P2P overlay networks. We propose a decentralized gossip based protocol by which PMs cooperatively measure Q-values and exchange among themselves to obtain identical ones. The protocol consists of two phases, called learning and aggregation, executed consecutively. Once PMs are triggered to run the algorithm by an oracle, in the learning phase, each PM locally simulates consolidation process and trains Q-values until a predefined period, and in the aggregation phase, they exchange values using a merge function so that PMs converge to the unified ones.

The learning phase is executed within PMs. To eliminate any impact on collocating

VMs in highly loaded PMs, only PMs with resource utilization less than a threshold (e.g., 80%) execute it locally. Such PM needs to collect VMs profiles of only one neighbor and aggregate with local profiles. To cover highly loaded states, it may need to duplicate profiles. Then, each PM resembles both sender and destination types by assigning a subset of VMs profiles randomly to each one. Then, it simulates the consolidation process for both types of PMs by removing VM from the one and adding to the another and updating Q-values using Equation (4.1). Such process is repeated several times (Algorithm2).

However, consideration of only current resources demand of VMs to determine state and action is unsuitable for an environment with dynamic and unpredictable workloads. To capture VMs workload variation more efficiently, we remark both **current** and **average** resource demands of VMs that have been monitored up to now in order to specify states and actions of PMs and VMs. To calculate the average demand, each VM piggybacks a tuple $\{c, v\}$ in which c represents the number of times the resource demand is monitored and v indicates the average observed demands. In the next profiling time, the new average can be calculated simply by $((c * v) + d(t))/(c + 1)$ where $d(t)$ is the current resource demand.

The state of a PM before performing an action as well as the state of migrating VM are calculated according to the **average** VMs demand while the **current** VMs demand are used to calculate the state of a PM after performing an action. In Figure 4.3, the average resource demand VM1 is 41% which is mapped to *High*. The state of vPM1 before migrating VM1 is the aggregation of its VMs average demands (79% = *3xHigh*). While the state after migrating VM1 is calculated as aggregation of remaining current demand of its VMs (50% = *High*). Therefore, according to Formula 4.1, to calculate $Q_{t+1}(3xHigh, High)$, in vPM1, s_t = *3xHigh*, s_{t+1} = *High*, a_t = *High*, and $Q_t(3xHigh, High)$ and $Q_t(High, High)$ are extracted from out map and the reward value R for moving to *High* state is obtained from R_{out}. The same rule is applied for vPM2 as destination PM.

At the end of this phase, beside PMs without any Q-value (Due to lack of enough resources to execute the algorithm), others may possess different Q-values while it is essential for all PMs to own identical ones. To this end, we execute aggregation phase of the protocol (Algorithm3). This is a gossip protocol in which in each round, every PM exchanges its Q-values with one randomly selected neighbor and updates values via a merge function. More precisely, if a state-action pair exists in both PMs, the new

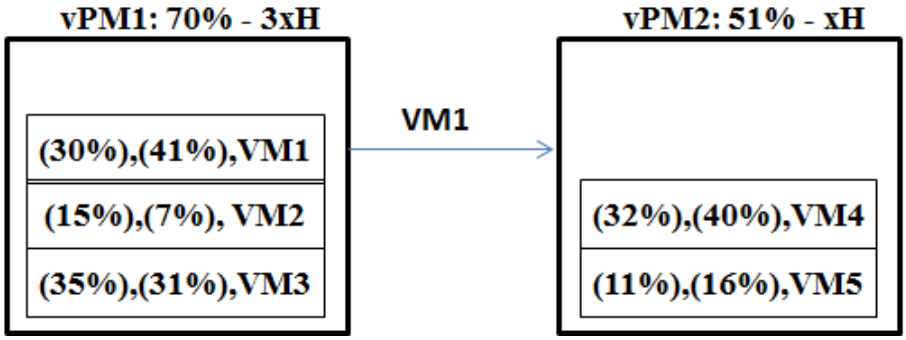


Figure 4.3: Learning Example. In each VM, the left number indicates the current demand and the right represents the average demand until now.

value is calculated as the average value but if the pair is in only one PM, the other just adds it to its Q-value list. This process is repeated several rounds until PMs converge to the unique Q-values.

Algorithm 2 Learning Phase

```
1:  procedure LOCALTRAIN(p)
2:      while triggered do
3:          if resource utilization p ≤ threshold then
4:              vms ← collect profiles of one neighbor
5:              vms ← aggregate with local VMs profiles
6:              vms ← duplicate vms if required
7:              for k times do
8:                  vmss ⊂ vms;                          ▷ sender vms
9:                  vmst ⊂ vms;                          ▷ target vms
10:                 vm ← select random vm of vmss to migrate;
11:                 updateOUT(vmss,vm);                  ▷ according to (4.1)
12:                 updateIN(vmst,vm);                   ▷ according to (4.1)
13:             end for
14:         end if
15:         sleep until end of Round
16:     end while
17: end procedure
```

Algorithm 3 Aggregation Phase

Require: Any node *p* has map of Q-values $\phi_p^{io} \leftarrow \phi_p^{in} \cup \phi_p^{out}$

```
1:  while triggered do                                          ▷ Active thread
2:      q = selectPeer();
3:      send(q , φ_p^io); φ_q^io = receive(q);
4:      UPDATE(φ_p^io , φ_q^io);
5:      sleep until end of round
6:  end while

7:  while triggered do                                          ▷ Passive thread
8:      φ_q^io = receive(q); send(q , φ_p^io);
9:      UPDATE(φ_p^io , φ_q^io);
10: end while

11: procedure UPDATE(φ_p^io , φ_q^io)
12:     for each {s,a} ∈ φ_p^io ∪ φ_q^io do
13:         if {s,a} exists in both φ_q^io and φ_p^io then
14:             calculate average two values, set new values
15:         else
16:             add {s,a} and value to the other map
17:         end if
18:     end for
19: end procedure
```

4.4.3 Analysis of The Convergence of Q-value Distribution

In this section, we show that the gossip-based process repeatedly aggregates Q-values that eventually converge to a normal distribution. To this end, we make the following assumption: the Q-values obtained during the aggregation phase by a node comes from a random node and are independent. Note that we do not require the distribution of selected nodes to be uniform or that Q-values are identical.

Theorem 4 *The random variable of the Q-value at round n converges to a normal distribution as n tends to ∞.*

Proof 4 *We consider $n+1$ rounds numbered from 0 to n. Let x_i be the Q-value at round i, $0 \leq i \leq \infty$. Since all Q-values are updated in parallel and independently, we only need to focus on independent x_i values as follows.*

Let X be the random variable of the Q-value at round n.

First, we have $X = x_0$. For $n = 1$, $X = \frac{x_0+x_1}{2}$. For $n = 2$, $X = \frac{\frac{x_0+x_1}{2}+x_2}{2} = \frac{x_0}{4}+\frac{x_1}{4}+\frac{x_2}{2}$. To generalize this, note that $\frac{1}{2^n}+\frac{1}{2^n}+\frac{1}{2^{n-1}}+\cdots+\frac{1}{2} = 1$, hence we obtain $X = \frac{x_0}{2^n} + \frac{x_1}{2^n} + \frac{x_2}{2^{n-1}} + \cdots + \frac{x_n}{2}$.

Let u_x be the expectation and σ_x^2 be the variance of x_i respectively, because of independent variables.

$$X - u_x = \frac{x_0 - u_x}{2^n} + \frac{x_1 - u_x}{2^n} + \frac{x_2 - u_x}{2^{n-1}} + \cdots + \frac{x_n - u_x}{2}.$$

Let y_i, $0 \leq i \leq \infty$, be random variables such that $y_i = \frac{x_i - u_x}{2^{n-i}}$. The expectation $u_{y,i}$ and the variance $\sigma_{y,i}^2$ of y_i are, respectively, $u_{y,i} = 0$ and $\sigma_{y,i}^2 = \frac{\sigma_x^2}{2^{2(n-i)}}$, given any fixed n and i.

Then according to Lindeberg or Lyapunov Central Limit Theorem, the summation of independent random variables, not necessarily identically distributed, converges in distribution and pre. Since y_i are the random variables in this kind, $X - u_x$ converges to the normal distribution. y constructing,

$$s_n^2 = \sum_{i=0}^{n} \sigma_{y,i}^2 = \sigma_x^2 \sum_{i=0}^{n} \frac{1}{2^{2(n-i)}} = \sigma_x^2 \Big(\sum_{i=0}^{n-1} \frac{1}{4^{(n-i)}} + \frac{1}{4^n} \Big) \tag{4.1}$$

and w.r.t. Lyapunov Central Limit Theorem[1]*, we know*

$$\frac{1}{s_n}\sum_{i=1}^{n}(y_i - u_{y,i}) \quad \xrightarrow{d} \quad \mathcal{N}(0,1), \tag{4.2}$$

as n *goes to infinity. That is,*

$$\frac{1}{s_n}(X - u_x) \quad \xrightarrow{d} \quad \mathcal{N}(0,1). \tag{4.3}$$

Note that in reality some of the Q-values may not be random because they are identical or null. However, the more random the Q-values in each round, the closer to expectation the final result could be. Also since we have known the distribution of final results, we can optimally decide how many rounds are needed at least to assure a satisfying convergence.

4.4.4 The Gossip Workload Consolidation Component

The consolidation component is built on top of the two other components (see Figure 4.2). Each PM runs two threads called active and passive threads and follows the push-pull interaction pattern, where a machine pushes its state to another machine and pulls that machines state. State is meta data information of the last monitored PM's resources utilization (Algorithm 4, Lines 1-10).

If the resource utilization of initiator PM (at least one of the resources) is overloaded, that PM requires to migrate some of its VM(s) to quit of the overload state (Lines 11-13). Otherwise, to consolidate VMs, the PM with totally less current utilized resources is selected as sender PM to migrate out its own VMs to switching to sleep mode (Lines 14-16). The sender PM, using its own current state, looks up the appropriate action from ϕ_p^{out} which is actually the action with the greatest Q-value for that state. Then, it picks a VM that corresponds to the chosen action. If there are several corresponding VMs for the action, the one with the least migration cost is selected. If there is no VM available with the specific action, the round is finished and no migration takes place. After selection of VM, sender PM finds the relevant Q-value from ϕ_p^{in} to check whether the target PM is able to accept this VM or not. If $\phi_p^{in}(s,a) \leq 0$,

[1]Lyapunov CLT says that the summation of independent random variables, not necessary to be identically distributed, converges to standard normal distribution. Here, we omit the long verification of Lyapunov condition in this paper.

Algorithm 4 Consolidation Component

Require: Any node P has the following methods & variables:

- s_p, s_q, ϕ_p^{in} : states p and q, map in
- *findVM(s_p)*: find action/vm for state s_p

```
1:  while true do                                              ▷ Active thread
2:      q ← selectPeer();
3:      send(q , s_p(t)); s_q(t) ← receive(q);
4:      UPDATESTATE();
5:      sleep until end of round
6:  end while
7:  while true do                                              ▷ Passive thread
8:      s_q(t) ← receive(q); send(q , s_p(t));
9:      UPDATESTATE();
10: end while

11: procedure UPDATESTATE()
12:     if p is in overloaded state then
13:         call MIGRATE() as long as p is overloaded
14:     else if p = arg min_{n∈{p,q}} (s_n(t)) then            ▷ p is sender
15:         call MIGRATE() as long as switch off p
16:     end if
17: end procedure

18: procedure MIGRATE()
19:     a , vm := p.findVM(s_p)
20:     if φ_p^in(s_q, a) < 0 or vm = ⊥ or no capacity then
21:         return and sleep until end of round;
22:     end if
23:     migrate vm from p to q and update s_p , s_q
24: end procedure
```

it means that the target PM cannot accept the VM and the round is finished without migrating VM. The negative value indicates that such VM migration very likely leads to SLA violation of the target PM and should be avoided. The interesting point is that because PMs own identical Q-values and also sender PM is aware of the target PM state, the decision is made in the sender PM on behalf of the target PM to eliminate communication overhead. Further, sender PM checks to ensure that the target PM has enough capacity to place the current demands of VM. In the end, VM migrates and states of both PMs are updated (Lines 18-24). As we mentioned earlier, recalculation of Q-values is done by learning component. In fact, learning component feeds consolidation component. During this process, consolidation component can be configured to either continue using the previous Q-values or pause for a while and resume by using new Q-values. Clearly, it can be implemented through synchronization between two components running within PMs and thus there is no communication overhead or any need for centralized server.

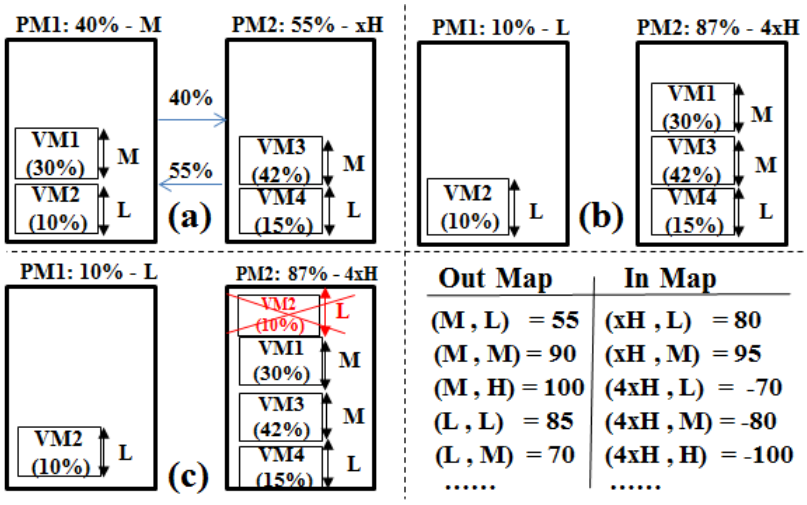


Figure 4.4: Consolidation Example. **(a)** PM1 exchanges its state with PM2 and it performs as sender because it has lower average resource utilization (40%). **(b)** VM1 migrates from PM1 to PM2 because it has greater Q-value 90 than VM2 with 55. **(c)** VM2 cannot migrate to PM2 because (4xH, L) has a negative Q-value -70 in "*in*-map". The round is finished.

4.5 Performance Evaluation

In this section, we compare the performance of GLAP against one centralized [11] and two distributed [9, 10] consolidation solutions using Google Cluster VMs traces [26].

4.5.1 Simulation Settings

To keep the environment under control so that we provide a stable configuration and execute repeatable experiments we carried out the experiments on a simulated cloud environment which supports running distributed P2P algorithms. We conducted experiments on PeerSim [2], a simulator for modeling large scale P2P networks. We simulated several configurations of cloud data center with 500, 1000, and 2000 nodes and VM-PM workload ratios of 2, 3, and 4 for each data center size. The PMs are modeled as HP ProLiant ML110 G5 servers (2660 MIPS CPU, 4GB memory, 10 GB/s network bandwidth) and the VMs are modeled from EC2 micro instance (500 MIPS CPU, 613 MB memory). At the beginning, VMs are allocated resources based on the demand of VM type, however, during execution they utilize less resources which gives the chance for dynamic consolidation.

The resource utilization trace from Google Cluster VMs are used to drive the VM resource utilization in the simulation [26]. The trace contains utilization of CPU and memory of VMs. We executed each experiment for 720 rounds such that each round mimics 2 minutes to simulate 24 hours and the evaluation metrics are sampled at the end of each round. It can show the efficiency of each protocol in long term how they deal with VMs variable workload. At the beginning, the VMs are randomly allocated to the PMs. For the sake of having fair comparison, such VM-PM mapping is used identically for all different algorithms in each experiment. We repeatedly carried out each experiment for 20 times and extracted the results. For GLAP, we executed 700 more rounds to calculate Q-values beforehand.

To evaluate GLAP, we compared it with two distributed and one centralized workload consolidation algorithms called GRMP [9], EcoCloud [10], and PABFD [11] respectively. GRMP is an aggressive gossip based protocol with a static upper threshold 0.8 while EcoCloud is a gradual probabilistic static upper and lower threshold based

protocol with the configuration of (T1 = 0.3 and T2 = 0.8). PABFD is a centralized dynamic threshold based heuristic consolidation algorithm in which a centralized server periodically monitors resources usage of PMs and using global information makes consolidation decisions. It calculates upper threshold by offline statistical analysis of historical data collected during the lifetime of VMs. The Median Absolute Deviation (MAD) is used as an estimator of upper threshold value. Continuously, we execute this algorithm for the same time as decentralized algorithms to be able to compare them fairly.

4.5.2 Performance metrics

One performance metric is SLA. It is often determined as throughput or response time ensured by applications. However, such characteristics vary for different applications and thus we define SLA violation (SLAV) as percentage of time during which the active hosts have experienced a CPU utilization of 100% (SLAVO) which indicates that some VMs may not be allocated required resource and performance degradation due to live VM migration (SLALM). [11]

$$SLAVO = \frac{1}{N}\sum_{i=1}^{N}\frac{T_{s_i}}{T_{a_i}} \quad , \quad SLALM = \frac{1}{M}\sum_{j=1}^{M}\frac{C_{d_j}}{C_{r_j}} \tag{4.4}$$

$$SLAV = SLAVO * SLALM \tag{4.5}$$

In the above formulas, N is the number of PMs, T_{s_i} is the accumulated time during which the PM i has encountered the CPU utilization of 100% , T_{a_i} is the total time during which the PM i is in active mode, M is the number of VMs, C_{d_j} is the performance degradation of the VM j caused by migration which is estimated as 10% of CPU utilization during all migrations of the VM j [11]. C_{r_j} is the total CPU capacity requested by the VM j during its lifetime.

Energy overhead that each live migration imposes is a metric for comparing the algorithms. We assume that the data center patronages VM live migration which is currently supported by some hypervisor technologies, such as Xen or VMware. The cost of migration is measured as energy overhead it imposes which is defined as the power consumption machine n, multiplied by the migration time τ. The migration

time strongly varies with VM's memory size and available transmission bandwidth at the source and destination servers. The power consumption of machine n is modeled as a linear function of its CPU consumption for migration and is represented as P_n^{lm}. Power consumption of machine n when it is in the idle mode is demonstrated as P_n^{idle}. The cost of migrating one VM from node i to node j is considered as energy overhead and is calculated with the following formula [3]:

$$E_{i\to j}^{lm} = ((\ P_i^{lm}\ -\ P_i^{idle}\ ) + (\ P_j^{lm}\ -\ P_j^{idle}\ ))\ *\ \tau_{i\to j}^{lm}\ , \tag{4.6}$$

Packing efficiency is another metric which shows how each algorithm captures energy-performance trade-off. However, only comparing the number of active PMs to the optimal one cannot show such trade-off. Beside, we calculate the number of overloaded PMs to active ones as well to see with which SLA cost, each algorithm consolidates PMs.

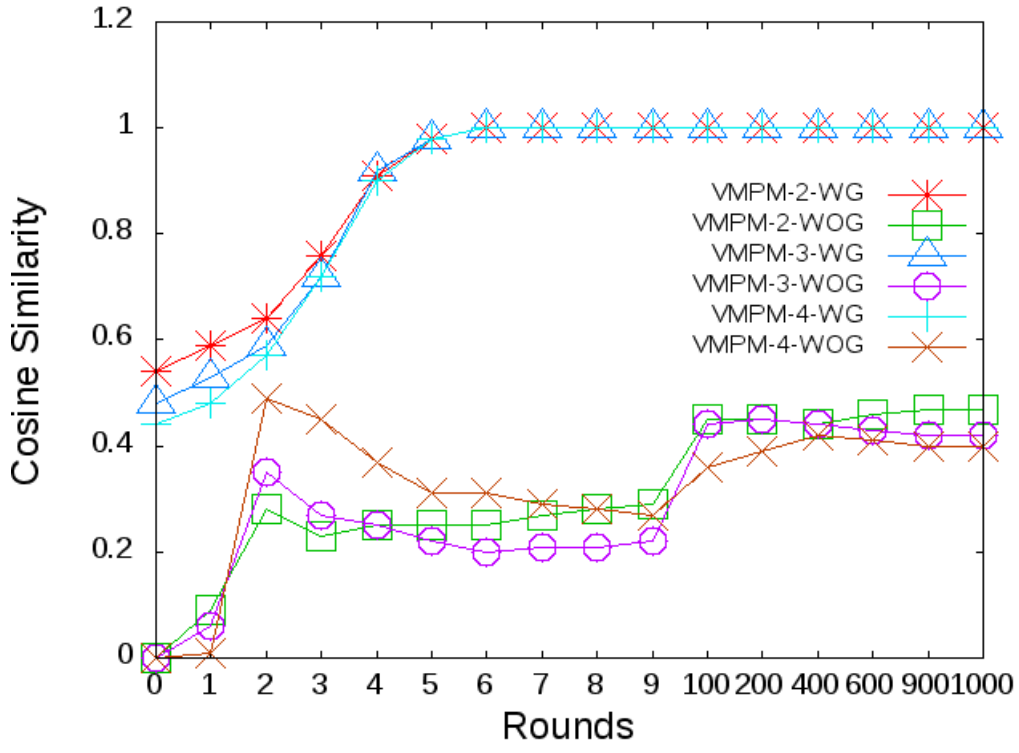


Figure 4.5: convergence Q-values after learning phase (WOG) and aggregation phase(WG) for VM-PM ratios 2,3, and 4.

4.5.3 Experimental Results

In this section, we present the results of the experiments.

4.5.3.1 Gossip learning component works correctly and Q-values converge rapidly

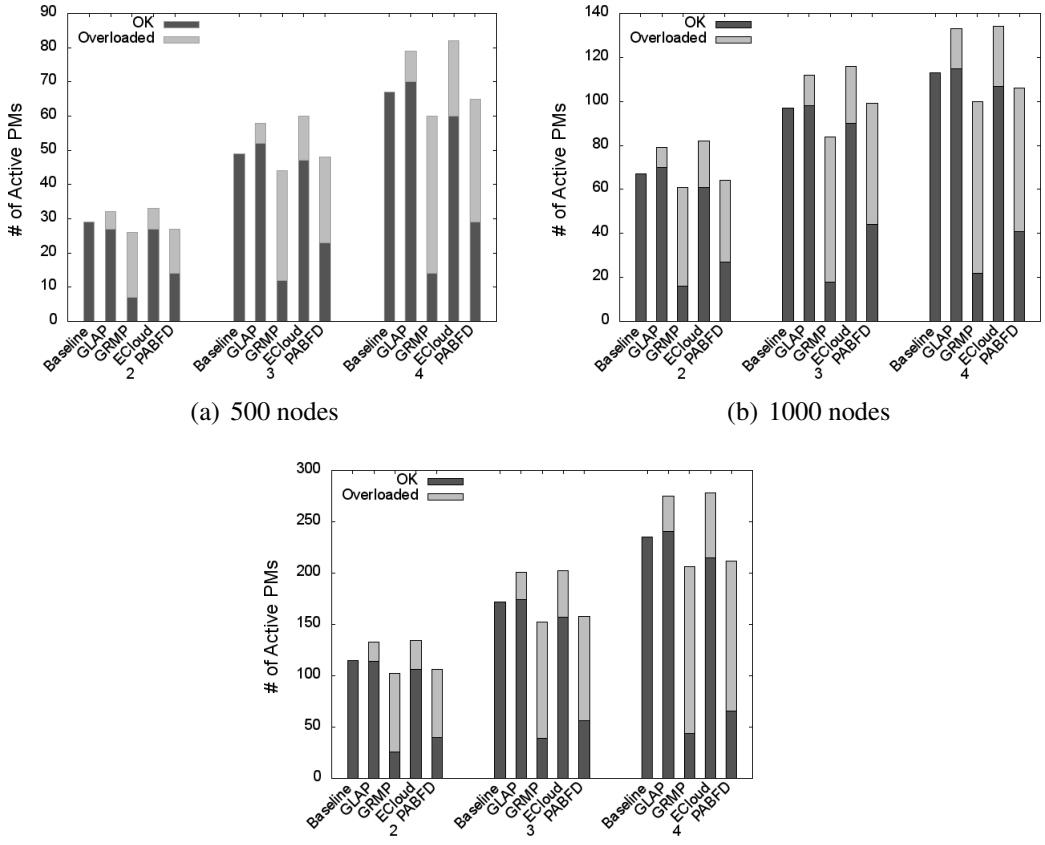
Q-values should be identical for all PMs. To evaluate the correctness of the 2 phase gossip learning protocol, we run experiments for 1000 nodes with workload ratios 2, 3, and 4. PMs with up to 50% free CPU are configured to run the algorithm locally to prevent any adversely impact on collocating VMs. We calculate Cosine similarity Q-values of PMs every cycle to observe how they converge towards identical values. In Figure 4.5, the protocol converges up to 45% after running the learning phase (WOG) for all VM-PM ratios. Then, in aggregation phase (WG), the PMs exchange their local values and as it can be seen in Figure 4.5 that Q-values converge rapidly for all PMs and VM-PM ratios. It shows the importance of the gossip learning protocol to make sure that every PM owns identical Q-values.

4.5.3.2 Minimizing the number of active PMs autonomously with much lower overloaded PMs

Figure 4.6 shows how aggressive each algorithm consolidate PMs. We calculated BFD (Best Fit Decreasing) using the VMs resource utilization of the last round to determine a baseline packing without producing any SLA violation. GRMP and PABFD switch off PMs even more than the baseline but at the high expense of SLA violations. While GLAP and EcoCloud keep a bit more number of PMs active than the baseline to impose much less SLA violation. It results that they autonomously consolidate PMs in a wisely manner. The results show that 58% of the PMs running PABFD, 22% of the PMs running EcoCloud and 75% of the PMs running GRMP are overloaded whereas only 12% of the PMs running GLAP get overloaded.

4.5.3.3 Minimizing the number of overloaded PMs

Figure 4.7 shows the number of overloaded PMs for all the algorithms. We extracted the value of overloaded PMs at the end of each round in all the executions and calculated the median, the 10th and 90th percentiles in the experiments. As it can be seen,

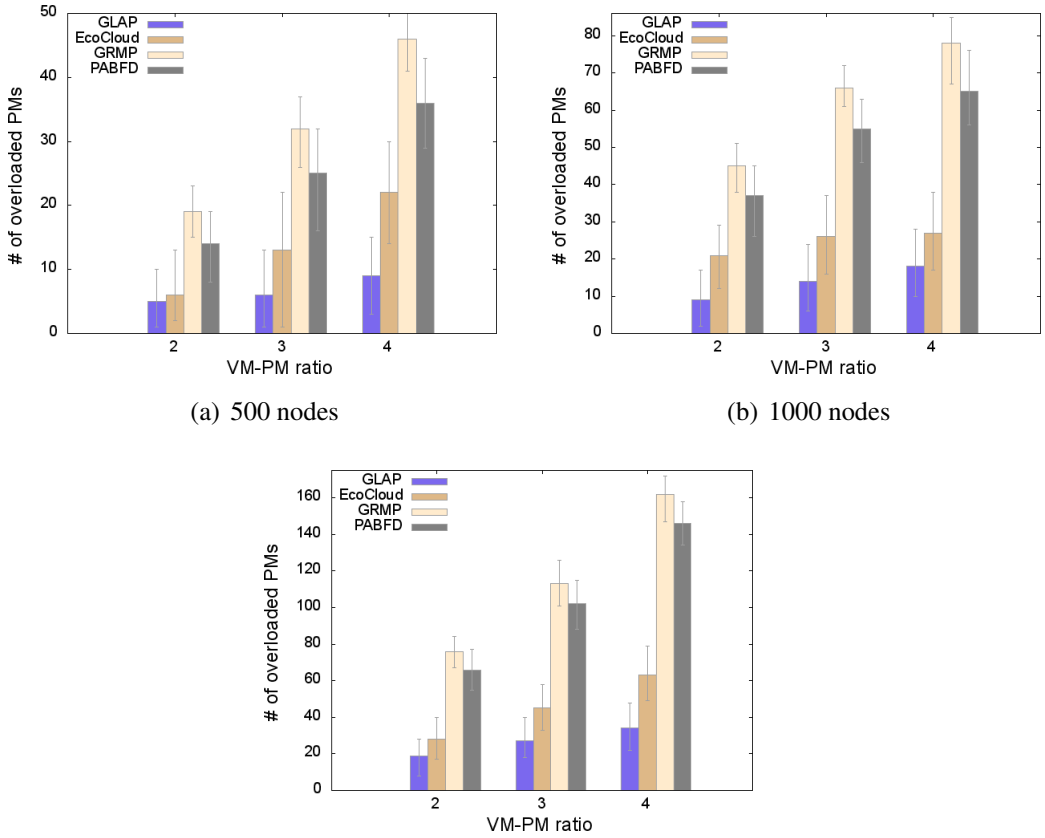


(a) 500 nodes

(b) 1000 nodes

(c) 2000 nodes

Figure 4.6: The fraction of overloaded / active PMs with increasing workload ratio and various cluster sizes



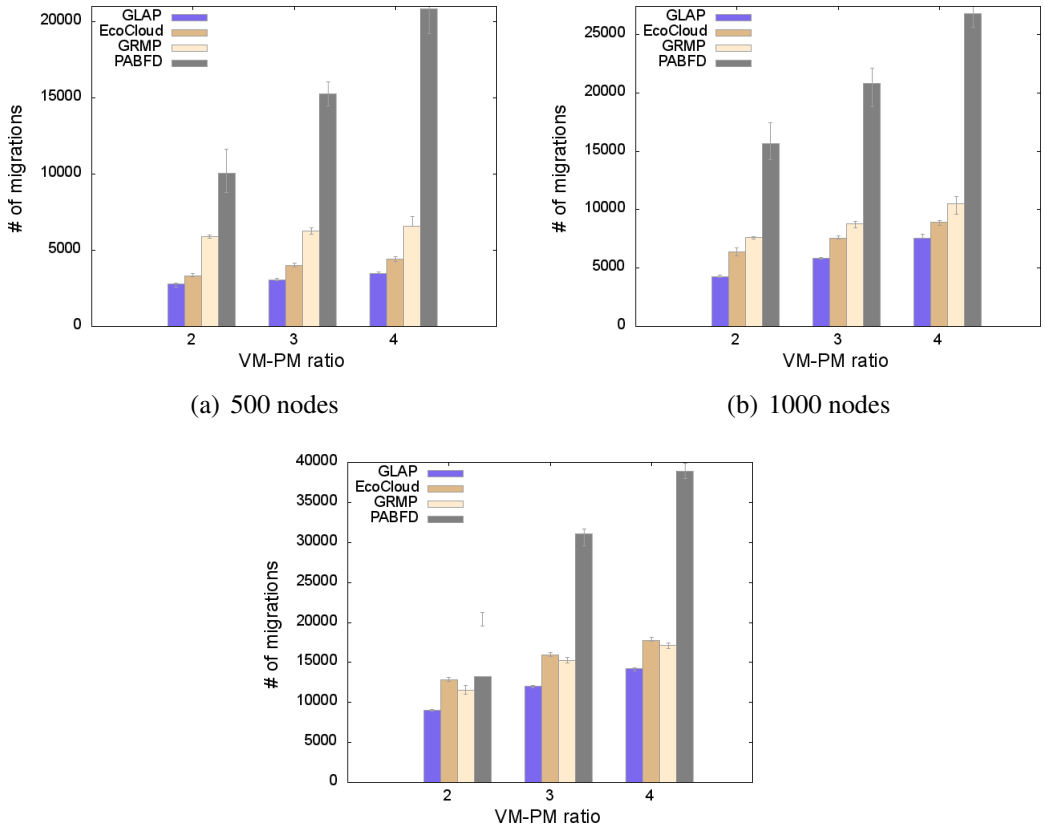
(a) 500 nodes

(b) 1000 nodes

(c) 2000 nodes

Figure 4.7: The number of overloaded PMs with increasing workload ratio and various cluster sizes

GLAP generates the smallest number of overloaded PMs. However, GRMP shows the worst result since it aggressively consolidates VMs into fewer PMs and switches off more PMs quicker. It improves power consumption at the high expense of performance degradation. Unlike GRMP, EcoCloud and GLAP, consolidate VMs in a slower slope and thus capture trade-off energy performance efficiently. GLAP outperforms EcoCloud, GRMP, and PABFD for 43%, 78%, and 73% less number of overloaded PMs, respectively. Figures 4.7 (a), (b), (c), show similar results for other cluster sizes and workload ratios, indicating the stability of the outcomes in various circumstances.



(a) 500 nodes

(b) 1000 nodes

(c) 2000 nodes

Figure 4.8: The number of migrations with increasing workload ratio and various cluster sizes

4.5.3.4 Minimizing the number of VM migrations

Figures 4.8 and 4.9 show the number and cumulative number of migrations during 1 day respectively. We measured the median, the 10th and 90th percentiles for this

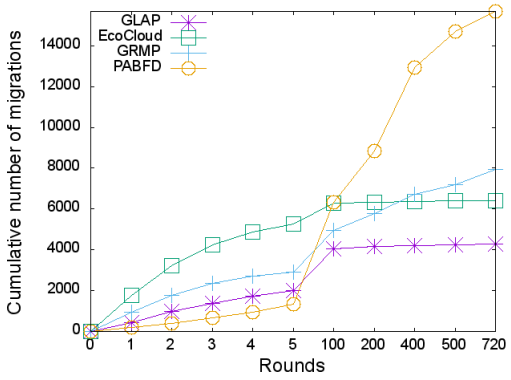
Table 4.1: SLA Metric for various cluster sizes and workload ratios

	GLAP	EcoCloud	GRMP	PABFD
500-2	0.00011	0.00016	0.27	0.07
500-3	0.00017	0.00045	0.48	0.19
500-4	0.00027	0.00078	0.72	0.36
1000-2	0.00017	0.00018	0.38	0.18
1000-3	0.00035	0.00078	0.61	0.36
1000-4	0.00059	0.00097	0.88	0.57
2000-2	0.00033	0.00076	0.41	0.29
2000-3	0.00066	0.0014	0.84	0.48
2000-4	0.001	0.002	1.24	0.48

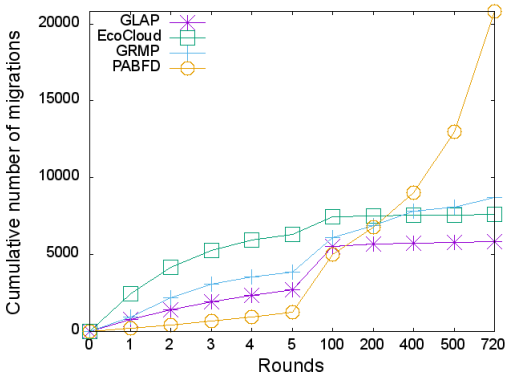
experiment. GLAP imposes the fewest number of migrations while PABFD considerably incurs the highest number of migrations. It stipulates this fact that such heuristic centralized algorithm is not efficient for continuous workload consolidation of PMs. GLAP outperforms EcoCloud, GRMP, and PABFD for 23%, 37%, and 70% less number of migrations respectively. These results confirm that our approach is advantageous for consolidation of VMs, because prediction of workload variation, it considerably reduces the probability of PMs being overloaded, which accordingly eliminates the need for excess migrations. It is noteworthy that with increasing the workload ratio, the total number of migrations increases. Figures 4.9 shows the cumulative number of migrations of 1000 nodes for three workload ratios of 2, 3, and 4. It can be observed that three distributed algorithms do most of the migrations in early rounds, however PABFD almost follows a linear relationship between time and number of migrations.

4.5.3.5 GLAP results in less continuous SLA violation

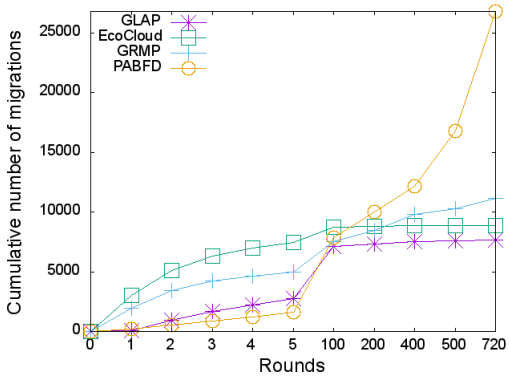
One important performance metric is service level agreement (SLA). We measured SLA metric for all the combinations of data center sizes and workload ratios. According to Table 4.1, GLAP causes less SLA violation (GLAP $<$ EcoCloud $<$ PABFD $<$ GRMP). With increment of workload, we can observe that SLA violation degree of the protocols increases, Yet, GLAP performs better than the other protocols. More precisely, for 500 node scenarios the SLA GLAP is 0.00011, 0.00017, and 0.00027 for loads 2, 3, and 4 respectively. The same pattern is for other cluster sizes, too.



(a) VM-PM workload ratio 2



(b) VM-PM workload ratio 3

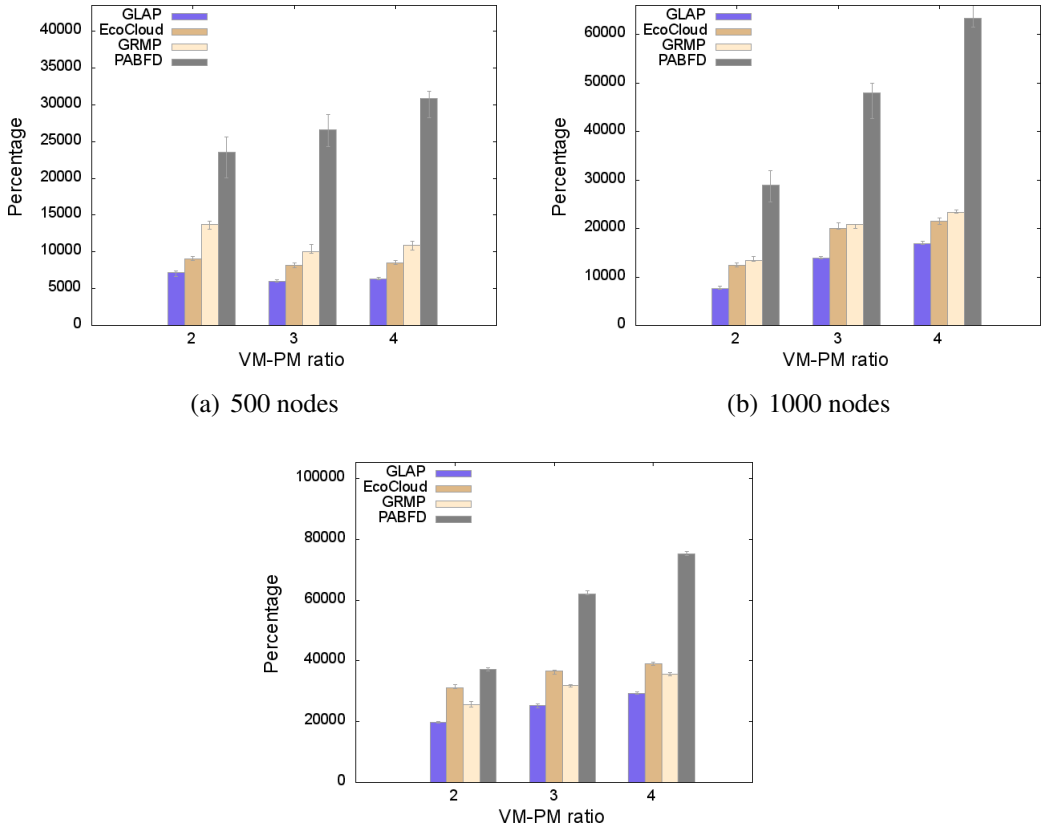


(c) VM-PM workload ratio 4

Figure 4.9: The cumulative number of migrations with increasing workload ratio for 1000 nodes

4.5.3.6 Minimizing energy overhead migrations

As Figure 4.10 reveals, the imposed energy overhead of each algorithm is because of the number of migrations. As it can be seen, among the evaluated solutions, PABFD consumes the highest energy while GLAP consumes the least. It is noteworthy that the higher number of migrations does not always lead to the highest energy consumption. For example, for 500 nodes, in GLAP, although workload ratio 4 has more number of migrations than 2, workload ratio 2 consumes more energy. The reason is that energy consumption of migrations is influenced by size of VMs and the time that each migration takes to complete.



(a) 500 nodes

(b) 1000 nodes

(c) 2000 nodes

Figure 4.10: Energy overhead of migrations with increasing workload ratio and various cluster sizes

4.6 Summary

In this chapter we proposed a gossip-based mechanism for consolidating VMs in a cluster of PMs. It handles VM load fluctuations (which reduces the number of unnecessary migrations) using a Q-Learning technique while remaining scalable and not relying on any fixed thresholds. The problem of highly sacrificing SLA for reducing the number of PMs is better addressed than with the state-of-the-art by incurring less SLA violations.

Chapter 5

Conclusions and Future Directions

With the growth of cloud data centers in terms of size and load, providing algorithms to efficiently manage resources at massive scale is in high demand. Peer-to-Peer algorithms have shown a suitable and efficient technique for scalability but they are inherently unreliable. However, deploying peer-to-peer algorithms behind cloud data centers benefits both scalability and reliability. In this thesis, we devised distributed and scalable algorithms for resource management in cloud and big data environments using peer-to-peer overlay networks.

Throughout the thesis work, we targeted two significant resource management problems. During the literature review which is explained in the chapter 2, we found that workload consolidation of virtual machines in cloud data centers is an important but well-researched topic. The main challenge in such algorithms is to handle efficiently trade-off between energy consumption of data centers and service level agreement of customers. During our study, we figured out that the existing distributed workload consolidation algorithms are not able to handle such trade-off efficiently and the root cause is that they are not able to capture dynamic resource demand of virtual machines in a fully distributed manner so that they can handle trade-off while retaining the scalability.

To fill the mentioned research gap, we proposed a fully distributed algorithm which is able to capture virtual machine workload fluctuations in a fully distributed fashion. We utilized the capability of virtual machine migration provided by hypervisors. First, we used a Q-Learning algorithm which is based on a reinforcement learning

technique. We modeled PMs and VMs of a cloud data center based on the components of Q-learning which are states and actions. In addition, we designed a reward system to decide when and which VMs to migrate between PMs. Also, we used gossip-based peer-to-peer overlay network to preserve scalability. The key challenge was how to train the learning model without requiring real virtual machine migrations and without using any centralized controller. To this end, each PM internally simulates training of Q-learning model based on the existing hosted VMs and then we designed a two-phase gossip based algorithm in which VMs exchange the trained model to eventually converge to the desire and stable model. Finally, through the second gossip based algorithm PMs utilizing the model decide when and which VMs to migrate so that they move VMs to as few PMs as possible and set the empty PMs to sleep mode and accordingly reduce the total energy consumption of the data center.

This work can be extended in different ways as future work. Currently, reward values for Q-learning model are set based on a simple role. As it is crucial to decide which VMs and when to migrate between PMs, we suggest using deep reinforcement learning or in particular deep Q-Learning technique. Through this technique, reward values can be obtained from neural networks with higher precision. It leads to more accurate prediction resulting in better handling trade-off, energy consumption and SLA of customers. During the literature review, we have seen that one of the main sources of energy consumption in data centers are network switches. Such observation indicates the significance of reducing the number of active network switches in the data center which together with reducing the number of PMs can save energy. Thus, as a future work, this algorithm can be extended to consider network switches as well and not just PMs. Last but not least, one can optimize to reduce the number of messages required to be exchanged for gossip protocols. It causes a more lightweight algorithm which decreases using the network bandwidth.

The second main contribution of the thesis is proposing a new probe-based and distributed scheduler for data analytics frameworks. To provide low latency job completion time, Modern data analytic frameworks increase the level of parallelism by breaking jobs into several short-length tasks. With the growth of cluster size and also a huge increase in the number of tasks of jobs due to the level of parallelism, the jobs scheduling time drastically increases which leads to higher job completion time. To

cope with this problem, the probe-based distributed scheduler is currently on demand. We studied the existing probe-based scheduler and figured out that the existing solutions suffer from a problem called Head-of-Line blocking. The existing probe-based techniques are almost based on two choices sampling or one of its variants such as batch sampling. Such techniques are augmented with amelioration mechanisms such as re-sampling, task stealing, etc. In addition, they use fixed-length or unbounded queues on worker nodes.To mitigate the Head-of-Line blocking problem, we proposed Peacock, a new probe-based scheduler. We introduced a new approach in which probes rotate between workers that form a ring overlay network. we also proposed a new probe reordering technique which is applied to queues of each worker. Moreover, unlike unbounded or fixed-sized queue of the existing algorithms, workers have elastic queues by which they can rotate probes between them. We showed that the algorithm is both scalable and provide low latency scheduling.

There are several future directions to this work. Peacock relies on the estimation of task execution time. Although we have shown that it is robust to such estimation time, this algorithm can be augmented to work without the need for the estimation of task execution time. In addition, peacock considers a homogeneous cluster and assume workers have identical capacity. Therefore, the algorithm can be extended to perform for heterogeneous clusters. Last but not least future work is handling strugglers. Some workers for different reasons may perform poorly even though they have the same specification as the other workers. Peacock algorithm can be extended to handle such malfunctioning workers.

Bibliography

[1] Khelghatdoust, Mansour, Vincent Gramoli, and Daniel Sun. ”GLAP: Distributed Dynamic Workload Consolidation through Gossip-Based Learning.” Cluster Computing (CLUSTER), 2016 IEEE International Conference on. IEEE, 2016.

[2] Montresor, Alberto, and Mark Jelasity. ”PeerSim: A scalable P2P simulator.” Peer-to-Peer Computing, 2009. P2P'09. IEEE Ninth International Conference on. IEEE, 2009.

[3] Strunk, Anja, and Waltenegus Dargie. ”Does live migration of virtual machines cost energy?.” Advanced Information Networking and Applications (AINA), 2013 IEEE 27th International Conference on. IEEE, 2013.

[4] P. Costa, V. Gramoli, M. Jelasity, G. Paolo Jesi, E. Le Merrer, A. Montresor, L. Querzoni. “Exploring the Interdisciplinary Connections of Gossip-based Systems”. ACM SIGOPS Operating Systems Review, 41(4):51-60 2007.

[5] Antonio Fernndez, Vincent Gramoli, Ernesto Jimnez, Anne-Marie Kermarrec, Michel Raynal. “Distributed Slicing in Dynamic Systems”. IEEE Transactions on Parallel and Distributed Systems, 2015.

[6] Vincent Gramoli, Ymir Vigfusson, Ken Birman, Anne-Marie Kermarrec and Robbert van Renesse. “Slicing Distributed Systems”. IEEE Transactions on Computers, 58(11):1444–1455, 2009.

[7] Voulgaris, Spyros, Daniela Gavidia, and Maarten Van Steen. “Cyclon: Inexpensive membership management for unstructured p2p overlays”. Journal of Network and Systems Management 13.2 (2005): 197-217.

[8] Chen, Liuhua, Haiying Shen, and Karan Sapra. "Distributed Autonomous Virtual Resource Management in Datacenters Using Finite-Markov Decision Process." Proceedings of the ACM Symposium on Cloud Computing. ACM, 2014.

[9] Wuhib, Fetahi, Rerngvit Yanggratoke, and Rolf Stadler. "Allocating compute and network resources under management objectives in large-scale clouds." Journal of Network and Systems Management 23.1 (2015): 111-136.

[10] Mastroianni, Carlo, Michela Meo, and Giuseppe Papuzzo. "Probabilistic consolidation of virtual machines in self-organizing cloud data centers." IEEE Transactions on Cloud Computing, 1.2 (2013): 215-228.

[11] Beloglazov, Anton, and Rajkumar Buyya. "Optimal online deterministic algorithms and adaptive heuristics for energy and performance efficient dynamic consolidation of virtual machines in cloud data centers." Concurrency and Computation: Practice and Experience 24.13 (2012): 1397-1420.

[12] Pantazoglou, Michael, Gavriil Tzortzakis, and Alex Delis. "Decentralized and Energy-Efficient Workload Management in Enterprise Clouds."

[13] GoogleTraceWebsite. Google cluster data. https://code.google.com/p/googleclusterdata/.

[14] Beloglazov, Anton, and Rajkumar Buyya. "Managing overloaded hosts for dynamic consolidation of virtual machines in cloud data centers under quality of service constraints." Parallel and Distributed Systems, IEEE Transactions on 24.7 (2013): 1366-1379.

[15] Kaelbling, Leslie Pack, Michael L. Littman, and Andrew W. Moore. "Reinforcement learning: A survey." Journal of artificial intelligence research (1996): 237-285.

[16] Farahnakian, Fahimeh, Pasi Liljeberg, and Juha Plosila. "Energy-efficient virtual machines consolidation in cloud data centers using reinforcement learning." Parallel, Distributed and Network-Based Processing (PDP), 2014 22nd Euromicro International Conference on. IEEE, 2014.

[17] Murtazaev, Aziz, and Sangyoon Oh. "Sercon: Server consolidation algorithm using live migration of virtual machines for green computing." IETE Technical Review 28.3 (2011): 212-231.

[18] Feller, Eugen, Christine Morin, and Armel Esnault. "A case for fully decentralized dynamic VM consolidation in clouds." Cloud Computing Technology and Science (CloudCom), 2012 IEEE 4th International Conference on. IEEE, 2012.

[19] Farahnakian, Fahimeh, et al. "Utilization Prediction Aware VM Consolidation Approach for Green Cloud Computing." Cloud Computing (CLOUD), 2015 IEEE 8th International Conference on. IEEE, 2015.

[20] Marzolla, Moreno, Ozalp Babaoglu, and Fabio Panzieri. "Server consolidation in clouds through gossiping." World of Wireless, Mobile and Multimedia Networks (WoWMoM), 2011 IEEE International Symposium on a. IEEE, 2011.

[21] Yazir, Yaiz Onat, et al. "Dynamic resource allocation in computing clouds using distributed multiple criteria decision analysis." Cloud Computing (CLOUD), 2010 IEEE 3rd International Conference on. Ieee, 2010.

[22] Feller, Eugen, Louis Rilling, and Christine Morin. "Snooze: A scalable and autonomic virtual machine management framework for private clouds." Proceedings of the 2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (ccgrid 2012). IEEE Computer Society, 2012.

[23] Graubner, Pablo, Matthias Schmidt, and Bernd Freisleben. "Energy-efficient management of virtual machines in eucalyptus." Cloud Computing (CLOUD), 2011 IEEE International Conference on. IEEE, 2011.

[24] Quesnel, Flavien, and Adrien Lbre. "Cooperative dynamic scheduling of virtual machines in distributed systems." Euro-Par 2011: Parallel Processing Workshops. Springer Berlin Heidelberg, 2012.

[25] Zaharia, Matei, et al. "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing." Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation. USENIX Association, 2012.

[26] GoogleTraceWebsite. Google cluster data. https://code.google.com/p/googleclusterdata/.

[27] Ousterhout, Kay, et al. "Sparrow: distributed, low latency scheduling." Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles. ACM, 2013.

[28] Delgado, Pamela, et al. "Job-aware Scheduling in Eagle: Divide and Stick to Your Probes." Proceedings of the Seventh ACM Symposium on Cloud Computing. No. EPFL-CONF-221125. 2016.

[29] Delgado, Pamela, et al. "Hawk: Hybrid Datacenter Scheduling." USENIX Annual Technical Conference. 2015.

[30] Tumanov, Alexey, et al. "TetriSched: global rescheduling with adaptive plan-ahead in dynamic heterogeneous clusters." Proceedings of the Eleventh European Conference on Computer Systems. ACM, 2016.

[31] Rasley, Jeff, et al. "Efficient queue management for cluster scheduling." Proceedings of the Eleventh European Conference on Computer Systems. ACM, 2016.

[32] Karanasos, Konstantinos, et al. "Mercury: Hybrid Centralized and Distributed Scheduling in Large Shared Clusters." USENIX Annual Technical Conference. 2015.

[33] Isard, Michael, et al. "Quincy: fair scheduling for distributed computing clusters." Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles. ACM, 2009.

[34] Hindman, Benjamin, et al. "Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center." NSDI. Vol. 11. No. 2011. 2011.

[35] Boutin, Eric, et al. "Apollo: Scalable and Coordinated Scheduling for Cloud-Scale Computing." OSDI. Vol. 14. 2014.

[36] DELIMITROU, Christina, Daniel SANCHEZ, and Christos KOZYRAKIS. "Tarcil: High Quality and Low Latency Scheduling in Large, Shared Clusters." (2014).

[37] Ferguson, Andrew D., et al. "Jockey: guaranteed job latency in data parallel clusters." Proceedings of the 7th ACM european conference on Computer Systems. ACM, 2012.

[38] Mustafa, Saad & Nazir, Babar & Hayat, Amir & Khan, Atta ur Rehman & Madani, Sajjad. (2015). Resource Management in Cloud Computing: Taxonomy, Prospects and Challenges. Computers & Electrical Engineering. 10.1016/j.compeleceng.2015.07.021.

[39] Zaharia, Matei, et al. ”Delay scheduling: a simple technique for achieving locality and fairness in cluster scheduling.” Proceedings of the 5th European conference on Computer systems. ACM, 2010.

[40] Harchol-Balter, Mor. Performance modeling and design of computer systems: queueing theory in action. Cambridge University Press, 2013.

[41] Hung, Chien-Chun, Leana Golubchik, and Minlan Yu. ”Scheduling jobs across geo-distributed datacenters.” Proceedings of the Sixth ACM Symposium on Cloud Computing. ACM, 2015.

[42] Curino, Carlo, et al. ”Reservation-based scheduling: If you’re late don’t blame us!.” Proceedings of the ACM Symposium on Cloud Computing. ACM, 2014.

[43] Goder, Andrey, Alexey Spiridonov, and Yin Wang. ”Bistro: Scheduling Data-Parallel Jobs Against Live Production Systems.” USENIX Annual Technical Conference. 2015.

[44] Verma, Abhishek, et al. ”Large-scale cluster management at Google with Borg.” Proceedings of the Tenth European Conference on Computer Systems. ACM, 2015.

[45] Vavilapalli, Vinod Kumar, et al. ”Apache hadoop yarn: Yet another resource negotiator.” Proceedings of the 4th annual Symposium on Cloud Computing. ACM, 2013.

[46] Melnik, Sergey, et al. ”Dremel: interactive analysis of web-scale datasets.” Communications of the ACM 54.6 (2011): 114-123.

[47] Dean, Jeffrey, and Luiz Andr Barroso. ”The tail at scale.” Communications of the ACM 56.2 (2013): 74-80.

[48] Sharma, Bikash, et al. ”Modeling and synthesizing task placement constraints in Google compute clusters.” Proceedings of the 2nd ACM Symposium on Cloud Computing. ACM, 2011.

[49] Ousterhout, Kay, et al. ”The Case for Tiny Tasks in Compute Clusters.” HotOS. Vol. 13. 2013.

[50] Venkataraman, Shivaram, et al. ”The Power of Choice in Data-Aware Cluster Scheduling.” OSDI. 2014.

[51] Tumanov, Alexey, et al. ”alsched: Algebraic scheduling of mixed workloads in heterogeneous clouds.” Proceedings of the third ACM Symposium on Cloud Computing. ACM, 2012.

[52] Reiss, Charles, et al. ”Heterogeneity and dynamicity of clouds at scale: Google trace analysis.” Proceedings of the Third ACM Symposium on Cloud Computing. ACM, 2012.

[53] Schwarzkopf, Malte, et al. ”Omega: flexible, scalable schedulers for large compute clusters.” Proceedings of the 8th ACM European Conference on Computer Systems. ACM, 2013.

[54] Kavulya, Soila, et al. ”An analysis of traces from a production mapreduce cluster.” Cluster, Cloud and Grid Computing (CCGrid), 2010 10th IEEE/ACM International Conference on. IEEE, 2010.

[55] Zhou, Jingren, et al. ”SCOPE: parallel databases meet MapReduce.” The VLDB JournalThe International Journal on Very Large Data Bases 21.5 (2012): 611-636.

[56] Stoica, Ion, et al. ”Chord: A scalable peer-to-peer lookup service for internet applications.” ACM SIGCOMM Computer Communication Review 31.4 (2001): 149-160.

[57] Chen, Y., Alspaugh, S., Katz, R. (2012). Interactive analytical processing in big data systems: A cross-industry study of mapreduce workloads. Proceedings of the VLDB Endowment, 5(12), 1802-1813.

[58] Chen, Yanpei, et al. ”The case for evaluating mapreduce performance using workload suites.” Modeling, Analysis Simulation of Computer and Telecommunication Systems (MASCOTS), 2011 IEEE 19th International Symposium on. IEEE, 2011.

[59] Khan, Md Anit, et al. ”Dynamic Virtual Machine Consolidation Algorithms for Energy-Efficient Cloud Resource Management: A Review.” Sustainable Cloud and Energy Services. Springer, Cham, 2018. 135-165. APA

[60] Lua, E. K., Crowcroft, J., Pias, M., Sharma, R., and Lim, S. (2005). A survey and comparison of peer-to-peer overlay network schemes. IEEE Communications Surveys and Tutorials, 7(2), 72-93. Chicago

[61] Jelasity, M., Montresor, A., & Babaoglu, O. (2005). Gossip-based aggregation in large dynamic networks. ACM Transactions on Computer Systems (TOCS), 23(3), 219-252. Chicago

[62] Jelasity, M., Voulgaris, S., Guerraoui, R., Kermarrec, A. M., & Van Steen, M. (2007). Gossip-based peer sampling. ACM Transactions on Computer Systems (TOCS), 25(3), 8.

[63] Jelasity, M., Guerraoui, R., Kermarrec, A. M., & Van Steen, M. (2004, October). The peer sampling service: Experimental evaluation of unstructured gossip-based implementations. In Proceedings of the 5th ACM/IFIP/USENIX international conference on Middleware (pp. 79-98). Springer-Verlag New York, Inc.

[64] Khelghatdoust, M., & Girdzijauskas, S. (2014). Short: Gossip-based sampling in social overlays. In Networked Systems (pp. 335-340). Springer, Cham.

[65] Gog, I. C., Schwarzkopf, M., Gleave, A., Watson, R. N., & Hand, S. (2016, November). Firmament: Fast, centralized cluster scheduling at scale. Usenix. Chicago

[66] Delimitrou, C., & Kozyrakis, C. (2013, March). Paragon: QoS-aware scheduling for heterogeneous datacenters. In ACM SIGPLAN Notices (Vol. 48, No. 4, pp. 77-88). ACM.

[67] Chen, Y., Alspaugh, S., & Katz, R. (2012). Interactive analytical processing in big data systems: A cross-industry study of mapreduce workloads. Proceedings of the VLDB Endowment, 5(12), 1802-1813. Chicago

[68] Bhattacharya, A. A., Culler, D., Friedman, E., Ghodsi, A., Shenker, S., & Stoica, I. (2013, October). Hierarchical scheduling for diverse datacenter workloads. In Proceedings of the 4th annual Symposium on Cloud Computing (p. 4). ACM.

[69] Armbrust, M., Fox, A., Griffith, R., Joseph, A. D., Katz, R. H., Konwinski, A., ... & Zaharia, M. (2009). Above the clouds: A berkeley view of cloud computing

(Vol. 17). Technical Report UCB/EECS-2009-28, EECS Department, University of California, Berkeley.

[70] Armbrust, M., Fox, A., Griffith, R., Joseph, A. D., Katz, R., Konwinski, A., ... & Zaharia, M. (2010). A view of cloud computing. Communications of the ACM, 53(4), 50-58.

[71] Zaharia, M., Chowdhury, M., Franklin, M. J., Shenker, S., & Stoica, I. (2010). Spark: Cluster computing with working sets. HotCloud, 10(10-10), 95.

[72] Zaharia, M., Chowdhury, M., Das, T., Dave, A., Ma, J., McCauley, M., ... & Stoica, I. (2012, April). Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation (pp. 2-2). USENIX Association.

[73] Buyya, R., Srirama, S. N., Casale, G., Calheiros, R., Simmhan, Y., Varghese, B., ... & Toosi, A. N. (2017). A Manifesto for Future Generation Cloud Computing: Research Directions for the Next Decade. arXiv preprint arXiv:1711.09123.

[74] Ratnasamy, S., Francis, P., Handley, M., Karp, R., & Shenker, S. (2001). A scalable content-addressable network (Vol. 31, No. 4, pp. 161-172). ACM.

[75] Zhao, B. Y., Huang, L., Stribling, J., Rhea, S. C., Joseph, A. D., & Kubiatowicz, J. D. (2004). Tapestry: A resilient global-scale overlay for service deployment. IEEE Journal on selected areas in communications, 22(1), 41-53.

[76] Rowstron, A., & Druschel, P. (2001, November). Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In IFIP/ACM International Conference on Distributed Systems Platforms and Open Distributed Processing (pp. 329-350). Springer, Berlin, Heidelberg.

[77] Maymounkov, P., & Mazieres, D. (2002, March). Kademlia: A peer-to-peer information system based on the xor metric. In International Workshop on Peer-to-Peer Systems (pp. 53-65). Springer, Berlin, Heidelberg.

[78] Malkhi, D., Naor, M., & Ratajczak, D. (2002, July). Viceroy: A scalable and dynamic emulation of the butterfly. In Proceedings of the twenty-first annual symposium on Principles of distributed computing (pp. 183-192). ACM.

[79] Ripeanu, M. (2001, August). Peer-to-peer architecture case study: Gnutella network. In Peer-to-Peer Computing, 2001. Proceedings. First International Conference on (pp. 99-100). IEEE.

[80] Clarke, I., Sandberg, O., Wiley, B., & Hong, T. W. (2001). Freenet: A distributed anonymous information storage and retrieval system. In Designing privacy enhancing technologies (pp. 46-66). Springer, Berlin, Heidelberg.

[81] Pouwelse, J., Garbacki, P., Epema, D., & Sips, H. (2005, February). The bittorrent p2p file-sharing system: Measurements and analysis. In IPTPS (Vol. 5, pp. 205-216).

[82] Eugster, P. T., Guerraoui, R., Kermarrec, A. M., & Massouli, L. (2004). Epidemic information dissemination in distributed systems. Computer, 37(5), 60-67.

[83] Jelasity, M., Montresor, A., & Babaoglu, O. (2003, July). A modular paradigm for building self-organizing peer-to-peer applications. In International Workshop on Engineering Self-Organising Applications (pp. 265-282). Springer, Berlin, Heidelberg.

[84] Voulgaris, S., & Van Steen, M. (2003). An epidemic protocol for managing routing tables in very large peer-to-peer networks. Self-Managing Distributed Systems, 299-308.

[85] Montresor, A., Jelasity, M., & Babaoglu, O. (2004, June). Robust aggregation protocols for large-scale overlay networks. In Dependable Systems and Networks, 2004 International Conference on (pp. 19-28). IEEE.

[86] Grandl, R., Kandula, S., Rao, S., Akella, A., & Kulkarni, J. (2016, November). G: Packing and Dependency-aware Scheduling for Data-Parallel Clusters. In Proceedings of OSDI16: 12th USENIX Symposium on Operating Systems Design and Implementation (p. 81).

[87] Grandl, R., Chowdhury, M., Akella, A., & Ananthanarayanan, G. (2016, November). Altruistic Scheduling in Multi-Resource Clusters. In OSDI (pp. 65-80).

[88] Delimitrou, C., & Kozyrakis, C. (2014, February). Quasar: resource-efficient and QoS-aware cluster management. In ACM SIGPLAN Notices (Vol. 49, No. 4, pp. 127-144). ACM.

[89] Li, W., Lin, C., Zhang, P., & Miao, M. (2017, July). Probe sharing: A simple technique to improve on sparrow. In Computers and Communications (ISCC), 2017 IEEE Symposium on (pp. 863-870). IEEE.

[90] Wuhib, F., Stadler, R., & Lindgren, H. (2012, October). Dynamic resource allocation with management objectivesImplementation for an OpenStack cloud. In Network and service management (cnsm), 2012 8th international conference and 2012 workshop on systems virtualiztion management (svm) (pp. 309-315). IEEE.

[91] Wuhib, F., Stadler, R., & Spreitzer, M. (2012). A gossip protocol for dynamic resource management in large cloud environments. IEEE transactions on network and service management, 9(2), 213-225.

[92] Chen, M., Mao, S., & Liu, Y. (2014). Big data: A survey. Mobile Networks and Applications, 19(2), 171-209.

[93] Tembey, P., Gavrilovska, A., & Schwan, K. (2014, November). Merlin: Application-and platform-aware resource allocation in consolidated server systems. In Proceedings of the ACM Symposium on Cloud Computing (pp. 1-14). ACM.

[94] OpenStack, http://www.openstack.org.

[95] Dougherty, B., White, J., & Schmidt, D. C. (2012). Model-driven auto-scaling of green cloud computing infrastructure. Future Generation Computer Systems, 28(2), 371-378.

[96] Ma, L., Liu, H., Leung, Y. W., & Chu, X. (2016, December). Joint VM-switch consolidation for energy efficiency in data centers. In Global Communications Conference (GLOBECOM), 2016 IEEE (pp. 1-7). IEEE.

[97] Deng, D., He, K., & Chen, Y. (2016, August). Dynamic virtual machine consolidation for improving energy efficiency in cloud data centers. In Cloud Computing and Intelligence Systems (CCIS), 2016 4th International Conference on (pp. 366-370). IEEE.

[98] Fioccola, G. B., Donadio, P., Canonico, R., & Ventre, G. (2016, December). Dynamic routing and virtual machine consolidation in green clouds. In Cloud Computing Technology and Science (CloudCom), 2016 IEEE International Conference on (pp. 590-595). IEEE.

[99] Jobava, A., Yazidi, A., Oommen, B. J., & Begnum, K. (2017). On achieving intelligent traffic-aware consolidation of virtual machines in a data center using Learning Automata. Journal of Computational Science.

[100] Khelghatdoust, M., Gramoli, V. (2018). Peacock: Probe-Based Scheduling of Jobs by Rotating Between Elastic Queues. Euro-Par 2018: 24th International Conference on Parallel and Distributed Computing (pp. 178-191).