Lightweight Framework for Reliable Job Scheduling in Heterogeneous Clouds

Muhammed Abdulazeez*, Pawel Garncarek†, Prudence W.H. Wong*
*Department of Computer Science, University of Liverpool, UK, Email:[m.abdulazeez,pwong]@liverpool.ac.uk
†Department of Computer Science, University of Wroclaw, Poland, Email:pgarn@cs.uni.wroc.pl

Abstract—**It is crucial to ensure reliability, security and stability of cloud services without sacrificing too much resources in the area of workload management in clouds. The paper evaluates and compares lightweight decentralized algorithms, recently proposed in [1], for scheduling a workload part of which could be unreliable, in the context of heterogeneous cloud data centers. This unreliability could be caused by various types of failures or attacks. The framework for robust workload scheduling efficiently combines classic fault tolerant and security tools, such as packet/job scanning, with workload scheduling, and it does not use any heavy resource consuming tools, e.g., cryptography or non-linear optimization. More specically, the framework uses a novel objective function to allocate jobs to servers and constantly decides which job to scan based on a formula associated with the objective function. In previous work it was shown how to set up the objective function and the corresponding scanning procedure of the central job scheduler to make the system provably stable, provided a specific capacity condition is satisfied. As a result, it was shown that the framework assures cloud stability even though naive scanning-all and scanning-none strategies are not stable for both centralized and decentralized scheduling in homogeneous data centers. In this work we extend the work to heterogeneous data centers, for which we show that decentralized algorithms based on Join Shortest Queue and Join Shortest Work policies are stable for every workload within the system capacity, while the algorithms based on popular Power of Two Choices, Round Robin and Uniform Random policies are not stable for a substantial amount of workloads even within the system capacity.**

I. Introduction

Cloud computing [2] enables ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources, these resources can be rapidly provisioned and released with minimal management effort.

While there is a growth in the use of cloud services, many potential users are still reluctant to rely on cloud computing resources for their businesses. Major concerns are its reliability, security and stability [3]. There are different reliability and security issues depending on the delivery models of cloud services, including Software as Service (SaS), Platform as Service (PaS) and Infrastructure as Service (IaS). In this work we focus on the IaS model. This technology makes the users and provider reside at different locations and virtually access the resources over the Internet, therefore any security concerns threatening the Internet also threaten the cloud. In particular, we consider the scenarios when part of the workload is unreliable, e.g., fault-prone or generated by malicious sources, and build on [1], a lightweight framework that combines load management and detection of unreliable traffic. The authors investigate how to strike a balance between efficient workload scheduling and packet/job scanning to maintain stability (as a guarantee of bounded buffers at machines) without sacrificing too much resources to filter out the unreliable part of the workload. The work done in [1] considered data centers with homogeneous servers. In this paper, we extend the work in [1] to consider heterogeneous servers.

IaS provides users with computing infrastructure in the form of Virtual Machines (VM). Following [4], we assume that the users request resources such as memory, CPU and storage, for a certain amount of time in the form of VMs; this corresponds to a job to be done. Upon receiving the requests (typically in a form of packets), the system has to allocate the required resources by scheduling the VMs on the server. Part of the workload is genuine and the other unreliable. Genuine traffic comes from real users; completing these requests counts towards system's work done. Unreliable traffic is subject to failures or comes from attackers, who aim to disrupt the system by issuing requests that occupy resources; completing these does not count as proper work done. We adopt a classic reliability and security tool of packet scanning to detect these malicious packets [5]. While scanning is able to distinguish genuine from unreliable requests, it consumes and wastes resources that would normally be used for serving genuine workload. On the other hand, as we do not know whether the packets are faulty/fake until we scan them, we may also waste time and resources in scanning genuine packets. Therefore, the scheduling algorithm needs to strike a balance between the resources wasted by scanning and by performing unreliable requests without scanning them.

In this work we consider heterogeneous settings where each server has different amount of resources available. This is because when cloud providers want to increase the capacity of their data centers, it is natural for them to upgrade to servers with bigger capacity. We also consider the *distributed setting* in which each server has its own queues; upon arrival, a job request is forwarded to some server and stored in the server's local queue corresponding to the requested type of VMs. When the resources become available, the scheduling algorithm determines which set of jobs is to be served within the local queue in the server.

The system is *stable* if the queues do not tend to increase without bound. In [1], it was shown that for homogeneous servers in a data center there exists a stable algorithm given maximal arrivals rates of genuine and unreliable requests when proper scanning procedure is selected. In this work, we aim to develop similar algorithms for a data center with heterogeneous servers. In addition to guarantee quality of service, we also measure job latency, which is defined as the amount of time a job resides in the system since its arrival. We present the precise model in Section II. In Section III we describe the job scheduler RobustMaxWork which was proposed recently in [1]. We then discuss the decentralized implementation of RobustMaxWork in Section IV. The experimental setting and the evaluation results of the performance of these algorithms are presented in Section V. In Section VI, we give mathematical proofs that some popularly used simple algorithms are not stable. Finally we conclude in Section VII.

A. Related Work

Apart from maintaining stability, there are many other design issues related to workload management in cloud computing. Cloud utilization has been considered in [6]. Optimizing other costs of running the services has been considered [7], [8], [6]. The algorithms we propose here are inspired by the MaxWeight algorithm analyzed in [9] in the context of scheduling genuine workload only, and could be seen as its efficient generalizations to unreliable environments. The MaxWeight algorithm has been since investigated extensively [10], [11], [12]. Detecting and distinguishing unreliable or malicious from genuine requests and a number of approaches have been proposed [5], [13]. In this paper, we assume that such a tool to scan a packet and detect potentially unreliable or malicious packages is available. The authors in [14] studied jobs with unknown duration and analyzed several decentralized approaches and showed that some are throughput-optimal while others are not. Several other algorithms for virtual machine allocation in heterogeneous data centers were analyzed through simulation in [15].

B. Our Contributions

We extend the study of managing workload in clouds under unreliable workload scenarios from homogenous servers setting [1] to heterogeneous servers setting. Extending the model in [4], [9], we detect unreliable part of the traffic by scanning only some specifically selected jobs without sacrificing too much resources.

- We extend the theoretical model [1] of capturing the essence of this conditional scanning to the heterogeneous server setting.
- We propose several decentralized versions of the algorithm RobustMaxWork on the heterogeneous setting.
- We evaluate the algorithms by extensive simulations, with respect to the maximum and average latency over time. The experiments illustrate that under a certain system capacity region and stochastic arrival pattern of genuine and unreliable jobs, coupling RobustMaxWork with server dispatching (routing) policies Shortest Queue First and Shortest Work First are stable while other routing policies Power of Two Choices, Round Robin and Uniform Random are unstable.
- We further support our experimental results by proving mathematically that Power of Two Choices, Round Robin and Uniform Random are unstable for a substantial amount of workloads even within the system capacity.

II. Model

The cloud model we consider is the one from previous works on multi-resource cloud scheduling, cf., [4], [10], enhanced by arrival of unreliable jobs and scanning capability as in [1].

We consider a cloud system modeled by a network of physical machines that have limited available resources (for instance, CPU, memory, storage, ...) and is supposed to be able to process an ongoing stream of jobs.

Servers. The system consists of n networked servers (physical machines), each having its own resources that can be used for processing jobs. Each resource has a fixed capacity, and the vector of resource capacities is called a *server capacity*. The whole *system capacity* is a linear sum of the capacities of all the servers.

Virtual Machines (VMs). When a job is processed on a server, a Virtual Machine must be created on the server. The Virtual Machine reserves specific amount of each resource defined by its *type* until the job is fully processed. The system has predefined all available Virtual Machine types. Let us denote the number of Virtual Machine types by J.

Jobs. A job has resource requirements and time requirement (length). For each job a specific Virtual Machine must be created on the server such that this virtual machine reserves enough resources to process the job. Therefore for a given job we reserve the resources required by a Virtual Machine rather than exact amount of resources required by the job. So we can define a job by the type of Virtual Machine that will be created for it and its length. This way the number of job types is limited by J. The system accepts only I different lengths of jobs: $L_1, \ldots, L_I$, for better control of queues and system stability. There are two classes of jobs: *genuine* (generated correctly by users) and *unreliable* (generated incorrectly or by malicious users/bots). We model job arrivals by an online random process, where new jobs arrive independently of each other and are identically distributed across all time slots, and the variance of arrival length is finite. We denote by $\lambda_{i,j}$ the expected sum of lengths of genuine (i.e., user-generated) type-j jobs of length L_i that arrive per time slot, for any positive integers $j \leq J$ and $i \leq I$.

Processing jobs and feasible configurations. Processing jobs is done in synchronous time steps, also called *rounds*. Each server can process a set of jobs simultaneously in any round, provided the cumulative amount of each resource used by these jobs does not exceed the corresponding server capacity. Given job types and server capacities, one can compute the set $\mathcal{S}$ of all *feasible configurations*, where feasible configuration denotes a vector $N = (N_1, \ldots, N_J)$ such that the system can process simultaneously N_j type-j jobs.

Unreliable jobs and reliability scanning. Let $\kappa_{i,j}$ denote the expected sum of lengths of unreliable jobs of type-j of length L_i that arrive per time slot. Similarly as genuine jobs, unreliable jobs arrive independently of each other and are identically distributed across all time slots, and the variance is finite. We assume that we have a reliability *scanning* tool, which can detect whether a given job is genuine (also called a *good job*) or unreliable. Each scanning takes 1 time slot per job and requires the same resources (i.e., same type of VM) as the original job.

Capacity region. A set of arrival rates (λ, κ) such that there exists an algorithm that is stable for (λ, κ) (such algorithm may be adjusted to be stable only under (λ, κ) arrival rate) is called the capacity region of the system. Note that given two systems with same total amount of resources, the system with multiple small servers may have smaller capacity region that a system with one large server. This is because the servers may be unable to utilize all of their resources, leaving too little unused resources for any additional job to fit in – such effect is more prevalent in the systems with many servers.

Job scheduler. The scheduler decides which servers process which jobs for the next time slot. After this time slot, all unfinished jobs return to the scheduler with saved progress and can be processed further at a later time and by a different server. This property of a system is called preemptiveness. However, as we will discuss in Section VII, all conclusions obtained in this work apply also to the non-preemptive setting, in which a job cannot be stopped or delayed while executing on a Virtual Machine. We consider a job scheduler introduced in [1], called RobustMaxWork; it will be described in details in Section III.

Distributed schedulers. In *distributed* (also called *decentralized*) approach all servers maintain separate queues for jobs of each type j. Upon job arrival, a decision is made as to which server to route the job; it is called a *routing* or *forwarding* protocol. Each server runs locally a job scheduler with respect to its local queues. The distributed implementations of RobustMaxWork with different routing protocols will be presented in Section IV.

Stability. We say that, given arrival rates λ and κ, the algorithm is stable if the expected queue size at any fixed time is bounded, i.e., $\limsup\limits_{t\to\infty} E[\sum_j Q_j(t)] < \infty$.

III. Job Scheduler RobustMaxWork

In this section we describe job scheduler RobustMaxWork, proposed recently in [1] (see Algorithm 1). It is parametrized by: scanning vector $\alpha = (\alpha_{i,j})_{i\le I, j\le J} \in [0,1]^{I\times J}$, vector of arrival rates of genuine jobs $\lambda = (\lambda_{i,j})_{i\le I, j\le J}$, and vector of arrival rates of unreliable jobs $\kappa = (\kappa_{i,j})_{i\le I, j\le J}$.

Upon arrival of type-j job of length L_i, RobustMaxWork decides to scan it with probability $\alpha_{i,j}$ (c.f., the first **for all** loop in Algorithm 1). The key idea of RobustMaxWork is to measure the expected time required to process all jobs of each type and prioritize the type which accumulated the most. The expected time (also called expected work) required to process all jobs of type j accumulated in queue at time t is denoted by $Z_j(t)$.

Algorithm 1 RobustMaxWork(λ, κ, α)

```
$X \leftarrow \vec{0}$   // jobs that will not be scanned
$Y \leftarrow \vec{0}$   // jobs that will be scanned
$Q \leftarrow \vec{0}$   // all jobs
loop
    new time slot begins
    for all new type-$j$ job $\tau_{i,j}$ of length $L_i$ do
        $r \leftarrow$ random value from $[0;1]$
        if $r < \alpha_{i,j}$ then                 // $\tau_{i,j}$ to be scanned
            $Y_j \leftarrow Y_j + L_i$
            $Q_j \leftarrow Q_j + L_i$
        else                                   // $\tau_{i,j}$ not to be scanned
            $X_j \leftarrow X_j + L_i$
            $Q_j \leftarrow Q_j + L_i$
        end if
    end for
    for all $j$ do
        $Z_j \leftarrow X_j + Y_j(\lambda_j/(\lambda_j + \kappa_j) + E(1/\ell_j))$
    end for
    $N' \leftarrow \arg\max_{N\in\mathcal{S}} \sum_j N_j \cdot Z_j$
    for all $j$ do
        for $k \le N'_j$ do
            Process_job($j$)
        end for
    end for
end loop
```

It takes X_j time to process jobs that will not be scanned. Jobs contributing to Y_j will need to be scanned, which requires $Y_j \cdot E(1/\ell_j)$ expected time steps. $\lambda_j/(\lambda_j + \kappa_j)$ fraction of scanned jobs are genuine, in expectation, thus after scanning, they still must be processed, taking in total $Y_j \cdot \lambda_j/(\lambda_j + \kappa_j)$ time. $\kappa_j/(\lambda_j + \kappa_j)$ fraction of the scanned jobs are fake and after scanning they take no more processing time. Hence, $Z_j(t) = X_j(t) + Y_j(t) \cdot (\lambda_j/(\lambda_j + \kappa_j) + E(1/\ell_j))$. In each time slot t, the algorithm computes Z_j in the second **for all** loop in Algorithm 1, and finds configuration N from the set of feasible server configurations $\mathcal{S}$ that maximizes the objective sum $\sum_{j=0}^{J} Z_j(t)N_j$. This configuration is denoted by N'. Intuitively, the more jobs of a given type is accumulated, the more weight should be put to scheduling this job type to prevent further accumulation. Z_j can be understood as the weight given to jobs of type j.

In the last **for all** loop, the algorithm processes N'_j jobs of type j, for each $1 \le j \le J$; i.e., from each job processed it executes a unit of it and the total size of Q_j decreases by N'_j at the end of time slot t. It is done by calling procedure **Process_job**(j). If N'_j is larger than the number of different type-j jobs in the queues, RobustMaxWork processes as many type-j jobs as possible instead, each time processing a unit of each such job. If N'_j is smaller than the number of different type-j jobs in the queues, RobustMaxWork has to decide which type-j jobs to process. It repeats N'_j times:

- with probability $X_j/(X_j+Y_j)$ it processes a job that will

not be scanned (i.e., a job that contributes to X_j),
- with probability $Y_j/(X_j+Y_j)$ it scans a job pending for scanning (i.e., a job that contributes to Y_j).

If there are not enough jobs contributing to X_j, it processes all jobs contributing to X_j and as many jobs contributing to Y_j as possible, so that altogether it processes N_j type-j jobs (and vice versa for Y_j).

IV. Decentralization

In decentralized approach each server maintains its own queues for jobs of each type-j, therefore, when a job arrives a decision has to be made as to which server to route the job. In this section we specify and analyze six different decentralized, also called parallel or distributed, implementations of the main job scheduler RobustMaxWork from Section III with different routing procedures. RobustMaxWork scheduler is used at each server to make scheduling decision wrt the queues and resources available at this server.

Below we describe six specification of the decentralized RobustMaxWork based on different routing policies. Algorithms A and C were analyzed in [14], [1] and Algorithm D was analyzed in [4], [1], all in the context of MaxWeight scheduling. Algorithms B, E, and F were proposed in [1] in the context of reliable cloud scheduling.

Algorithm A: RobustMaxWork_JSQ. Join Shortest Queue (JSQ) paradigm is used to route a newly arrived job to the server with the queue with the smallest number of jobs of type-j, where j denotes the type of the arrived job.

Algorithm B: RobustMaxWork_JSW. Join Shortest Work (JSW) is used for routing a newly arrived job to the server with the minimum workload of type-j, where the workload is the sum of lengths of jobs in the local queue of type-j.

Algorithm C: RobustMaxWork_UR. Uniformly Random (UR) routing is used for forwarding newly arrived job to a server chosen uniformly at random.

Algorithm D: RobustMaxWork_RR. Round Robin (RR) routine is used for allocating newly arrived jobs to the servers, where RR is used separately for each type.

Algorithm E: RobustMaxWork_P2Q. Power of two Choices combined with selection of the Shortest Queue (P2Q) is used for routing a newly arrived job of a type-j: two servers are sampled uniformly at random, and the job is routed to the server with the shorter type-j queue.

Algorithm F: RobustMaxWork_P2W. Power of two Choices combined with selection of the Shortest Workload (P2W) is used for routing a newly arrived job of a type-j: two servers are sampled uniformly at random, and the job is routed to the server with the smaller workload of type-j (i.e., where the total length of type-j jobs in the local queue is shorter).

V. Simulations

A. Experiment Setting

The setup for simulations, described in this section, is based on the one in Maguluri et al. [4].

Table I
Representation of Instances in Amazon EC2

Instance type	Memory (GB)	vCPU	Storage (GB)
Standard	15	8	1,690
High-Memory	17.1	6.5	420
High-CPU	7	20	1,690

Servers and VMs. We consider two types of servers in the data center, the first with the following configuration: 30 GB memory, 30 EC2 computing units and 4096 GB (4TB) storage space. Arriving jobs are served in the cloud based on three types of Virtual Machines described in Table I. This gives three maximal configurations available at each server: $(2,0,0)$, $(1,0,1)$ and $(0,1,1)$. The second server has 68GB Memory, 80 EC2 computing units and 7168GB (7TB) of storage, based on the virtual machine configuration in Table I, this gives us maximal configuration $(4,0,0)$, $(2,2,0)$, $(0,0,4)$, $(3,1,0)$, $(3,0,1)$, $(1,3,0)$, $(0,3,2)$, $(1,0,3)$, $(0,2,3)$, $(2,1,1)$, $(2,0,2)$ and $(1,2,2)$.

Job arrivals. We use the generic arrival vector $\lambda^* = 0.99\cdot(1,1/3,2/3)$ for genuine users' workload, which is located close to the border of server one capacity area taking all three maximal configurations and also close to the server two capacity area taking three of its maximal configurations $(4,0,0)$, $(2,2,0)$, $(0,0,4)$ (observe that λ^* is a normalized linear combination of the six configurations mentioned, additionally re-scaled by factor 0.99). In each time step a job of type $j=1,2,3$ is selected with probability $\frac{\lambda_j^*}{130.5}$, and its length is chosen according to the length distribution described below with mean length 130.5 (calculation shown later).

Similarly as above, we define an ureliable workload using a generic arrival vector $\kappa^* = (0.7, 0.01, 0.01)$, and the procedure of generating an unreliable traffic is analogous as above for generating the genuine users' one. Note that each of the arrival rates λ^* and κ^* is within the capacity range of a server, whereas the combined workflow rate $\lambda^*+\kappa^*$ is not.

Job size distribution. When a new job is generated, with probability of 0.7 it is an integer uniformly distributed in the interval $[1,50]$, with probability of 0.15 it is an integer uniformly distributed in $[251,300]$, and with probability of 0.15 it is an integer uniformly distributed in $[451,500]$. Note that there are 150 possible job lengths, and the mean length is 130.5, as assumed in the definition of arrival rates.

Set up of simulations. In the *homogenous* setting we use 100 servers of type one. In the *heterogenous* setting we use 80 servers of type one and 10 of type two. We therefore get the maximal feasible arrival rates in heterogenous setting to be:
- $80\cdot\frac{1}{3}\cdot(3,1,2) = (80, 80/3, 160/3)$ for type one;
- $10\cdot\frac{1}{3}\cdot(6,2,4) = (20, 20/3, 40/3)$ for type two

which in total is the same arrival rate as for the homogenous setting $100\cdot\frac{1}{3}\cdot(3,1,2) = (100, 100/3, 200/3)$. Based on these the overall arrival rates are: $\lambda = 100\cdot\lambda^* = (99,33,66)$ for genuine workload, and $\kappa = 100\cdot\kappa^* = (70,1,1)$ for unreliable workload – they are inside the capacity regions for both homogenous and heterogenous settings (note that the capacity region of homogenous setting is slightly smaller than that of heterogeouns). The job size distribution is as specified above, same for each job type. We computed the following optimal scanning vector α^* for this setting, more precisely, the vector minimizing the expected arriving weight:

- $\alpha^*_{i,1} = 0$ for $L_i \leq 2$; $\alpha^*_{i,2} = 0$ for $L_i \leq 34$;
- $\alpha^*_{i,3} = 0$ for $L_i \leq 50$; $\alpha^*_{i,j} = 1$ otherwise.

Each execution includes $4,000,000$ time steps. We compute *average latency* and *maximum latency* at every time step and record the results every 200,000 steps. We ran the experiments 10 times and took the averages of the results for each recorded time step. We output the results of the above measurements of RobustMaxWork applied on simultaneous genuine and malicious flows, with scanning defined by vector α^* (we call this scanning ScanOPT). We study how different routing protocols influence stability, when applied to the RobustMax-Work with the optimally selected scanning vector α^*. We compare the six decentralized implementations of RobustMax-Work: namely, ScanOPT_JSQ, ScanOPT_JSW, ScanOPT_UR, ScanOPT_RR, ScanOPT_P2Q, and ScanOPT_P2W.

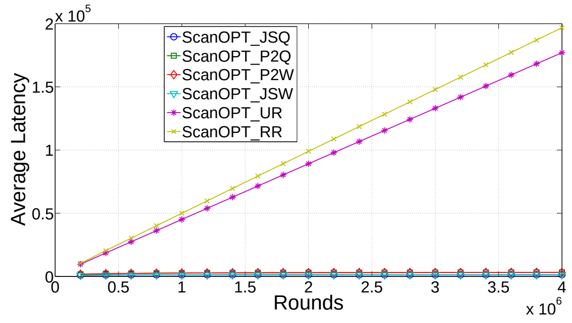

Figure 1. Comparison of average latency using ScanOPT_JSQ, ScanOPT_JSW, ScanOPT_P2Q, ScanOPT_P2W, ScanOPT_UR and ScanOPT_RR.

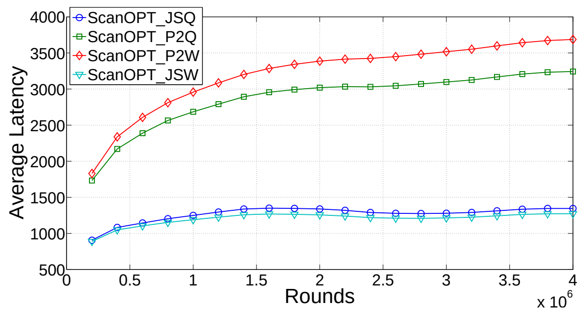

Figure 2. Comparison of average latency using ScanOPT_JSQ, ScanOPT_JSW, ScanOPT_P2Q and ScanOPT_P2W (zoom of figure 1).

B. Results

Figure 1 compares average latency of the six decentralized algorithms using ScanOPT strategy. Figure 2 zooms in on the four better algorithms. The best performing algorithms are the ones based on JSW and JSQ followed by the two algorithms based on the power of choices P2W and P2Q respectively. The worst performing algorithms are based on round robin and uniform random selection, which grow rapidly. JSW and JSQ is the best in terms of average latency performance, although the other two reasonable policies(P2W and P2Q) also show some stability trends. It was expected that the two algorithms based on power of two choices will not perform as well as the JSQ and JSW algorithms because they are random and majority of the selection will be from the small set of servers i.e. server one. Therefore, work will not be evenly distributed.

We also compared the heterogeneous servers with homogeneous servers in the data center using 100 of servers one and similar arrival vectors for both genuine and malicious traffic: the same setup as [1]. Figure 4 and 3 compared the difference in both average and maximum latencies between homogeneous and heterogeneous data centers. The general trend is that all algorithms perform better in homogeneous setting. The worst performing algorithms i.e. uniform random and round robin are getting worse with time for both maximum and minimum latency.

The interesting result is that for average latency (see Figure 4) the algorithm with the least difference is JSQ though it did not perform as well as the second performing algorithm JSW for both heterogeneous and homogeneous servers. This is similar with algorithms based on power of two choices with the one based on queue doing better than the one based on work.

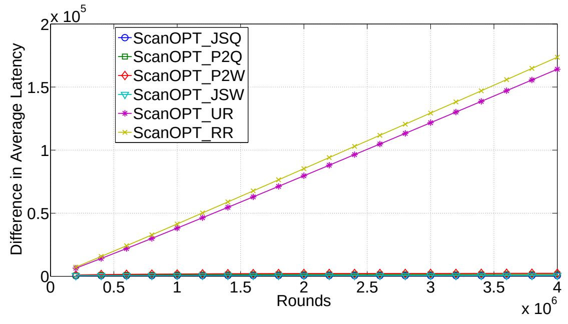

Figure 3. Comparison of difference in average latency between heterogeneous and homogeneous servers for all the algorithms.

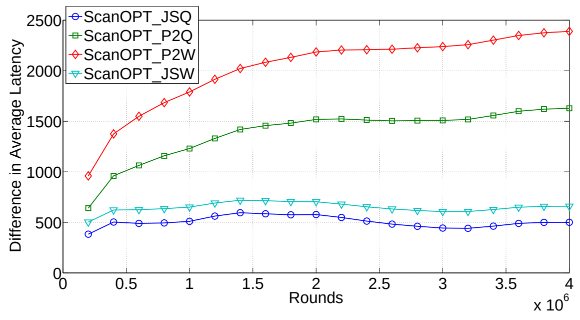

Figure 4. Comparison of difference in average latency between heterogeneous and homogeneous servers for ScanOPT_JSQ, ScanOPT_JSW, ScanOPT_P2Q and ScanOPT_P2W (zoom of figure 3).

VI. Instability of Power of two Choices, Round Robin and Uniform Random routing policies

In this section we present theoretical results that explain some of the results obtained from the experiments.

Theorem 1: No job scheduler is stable for all arrival vectors inside the capacity region when combined with Power of two Choices routing policy.

Proof: Consider our system of 80 small servers and 10 large servers as in Section V-A. With probability $(80/90)^2$, Power of two Choices randomly picks two small servers and chooses one of them as destination for a considered job. This means that there are only 80 small servers that will receive at least $(8/9)^2$ of all the jobs.

Consider arrival rate of genuine jobs of $c \cdot [80 \cdot (1, 1/3, 2/3) + 10 \cdot (0, 2, 3)]$ for some $c < 1$. This arrival rate lies inside the capacity region of the system. Let x, y, z be the average number of small servers that chose configuration $(2, 0, 0)$, $(1, 0, 1)$ and $(0, 1, 1)$, respectively, per time slot. Note that for the system to be stable, the following must hold:

$$x + y + z = 80 \quad \text{(there are 80 small servers)}$$

and type-j jobs must be processed at least as frequently as they arrive:

$$\begin{aligned} 2x + y &\geq (8/9)^2 c \cdot (80 \cdot 1 + 10 \cdot 0) && \text{(type-1 jobs)} \\ z &\geq (8/9)^2 c \cdot (80 \cdot 1/3 + 10 \cdot 2) && \text{(type-2 jobs)} \\ y + z &\geq (8/9)^2 c \cdot (80 \cdot 2/3 + 10 \cdot 3) && \text{(type-3 jobs)} \end{aligned}$$

If we sum up the above three inequalities we get $81/84 \geq c$. Hence, for $c = 0,97$ no job scheduler can process jobs in small servers as fast as Power of two Choices injects them, despite the arrival rates being inside the capacity region.

■

Note that in the scenario described in the proof of Theorem 1, Round Robin and Uniform Random routing policies would direct 8/9 jobs to the small servers instead of $(8/9)^2$, which makes the overload of the small servers even bigger than in case of Power of two Choices. Therefore, we get the following conclusion.

Corollary 1: No job scheduler is stable for all arrival vectors inside the capacity region when combined with Round Robin or Uniform Random routing policy.

VII. Conclusions, Extensions and Open Problems

The centralized algorithm can be extended to JSW, an interesting open problem is more detailed analysis of other decentralized approaches. Another interesting open problem is Non-preemptiveness: we assumed that in regular time intervals all machines can be reconfigured — all jobs could be rescheduled and redistributed among the machines for further process. In some systems, interrupting execution of some jobs may be very costly. It is known how to transform such preemptive algorithms into ones with no reconfigurations, within (roughly) the same stability region, c.f., [4].

We also assumed that system has knowledge of arrival rates: there are many ways of removing the assumption of known arrival rates, without influencing stability and asymptotic performance. One is to start RobustMaxWork using the naive scanning-all-jobs strategy for a fixed but sufficiently long period, during which the scheduler learns the genuine/unreliable status of jobs (due to the scan-all strategy) and therefore it will be able to estimate user-generated and unreliable jobs arrival rates. It can then compute scanning frequencies that are optimal for the estimated arrival rates. The model can also be enhanced with the possibility of wrong result of scanning, which may fail with some probability p, i.e., an unreliable job may be scanned but still not discovered as a faulty one. The analysis of stability of RobustMaxWork job scheduler and its distributed implementations continue to hold, with a slight modification of the formula for job weights Z_j.

Acknowledgment

The authors would like to thank Dariusz R. Kowalski for helpful discussion.

This work was supported by the Polish National Science Centre grant DEC-2012/06/M/ST6/00459.

References

[1] M. Abdulazeez, P. Garncarek, and P. W. Wong, "Lightweight robust framework for workload scheduling in clouds," in *Proceedings of IEEE EDGE 2017*, to appear. [Online]. Available: https://www.dropbox.com/s/uvh2jdul7ri0xtv/main.pdf

[2] P. Mell and T. Grance, "The NIST definition of cloud computing," *National Institute of Standards and Technology*, 2011.

[3] S. Subashini and V. Kavitha, "A survey on security issues in service delivery models of cloud computing," *J. Network and Computer Applications*, vol. 34, no. 1, pp. 1–11, 2011.

[4] S. T. Maguluri, R. Srikant, and L. Ying, "Stochastic models of load balancing and scheduling in cloud computing clusters," in *INFOCOM*, 2012, pp. 702–710.

[5] C. Modi, D. R. Patel, B. Borisaniya, H. Patel, A. Patel, and M. Rajarajan, "A survey of intrusion detection techniques in cloud," *J. Network and Computer Applications*, vol. 36, no. 1, pp. 42–57, 2013.

[6] R. Xie, X. Jia, K. Yang, and B. Zhang, "Energy saving virtual machine allocation in cloud computing," in *33rd IEEE Int. Conf. on Distributed Computing Systems Workshops*, 2013, pp. 132–137.

[7] A. Stolyar and Y. Zhong, "A service system with packing constraints: Greedy randomized algorithm achieving sublinear in scale optimality gap," *arXiv preprint arXiv:1511.03241*, 2015.

[8] M. Wang, X. Meng, and L. Zhang, "Consolidating virtual machines with dynamic bandwidth demand in data centers," in *INFOCOM*, 2011, pp. 71–75.

[9] L. Tassiulas and A. Ephremides, "Stability properties of constrained queueing systems and scheduling policies for maximum throughput in multihop radio networks," *IEEE Trans. on Automatic Control*, vol. 37, no. 12, pp. 1936–1948, 1992.

[10] M. G. Markakis, E. Modiano, and J. N. Tsitsiklis, "Delay analysis of the max-weight policy under heavy-tailed traffic via fluid approximations," in *51st Allerton Conf. on Comm., Contr., and Comp.*, 2013, pp. 436–444.

[11] W. Sun, N. Zhang, H. Wang, W. Yin, and T. Qiu, "Paco: A period aco based scheduling algorithm in cloud computing," in *Int. Conf. on Cloud Computing and Big Data (CloudCom-Asia)*, 2013, pp. 482–486.

[12] S. T. Maguluri, R. Srikant, and L. Ying, "Heavy traffic optimal resource allocation algorithms for cloud computing clusters," *Performance Evaluation*, vol. 81, pp. 20–39, 2014.

[13] A. Bakshi and Y. B. Dujodwala, "Securing cloud from ddos attacks using intrusion detection system in virtual machine," in *Intl. IEEE Conf. on Comm. Software and Networks 2010*, 2010, pp. 260–264.

[14] S. T. Maguluri and R. Srikant, "Scheduling jobs with unknown duration in clouds," *IEEE Trans. on Net.*, vol. 22, no. 6, pp. 1938–1951, 2014.

[15] M. Stillwell, F. Vivien, and H. Casanova, "Virtual machine resource allocation for service hosting on heterogeneous distributed platforms," in *IPDPS*. IEEE, 2012, pp. 786–797.