Lightweight Robust Framework for Workload Scheduling in Clouds

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

***Abstract*—Reliability, security and stability of cloud services without sacrificing too much resources have become a desired feature in the area of workload management in clouds. The paper proposes and evaluates a lightweight framework for scheduling a workload which part could be unreliable. This unreliability could be caused by various types of failures or attacks. Our 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 specifically, 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. We show how to set up the objective function and the corresponding scanning procedure to make the system provably stable, provided it satisfies a specific stability condition. As a result, we show that our framework assures cloud stability even if naive scanning-all and scanning-none strategies are not stable. We extend the framework to decentralized scheduling and evaluate it under several popular routing procedures.**

I. Introduction

Cloud computing [1] enables ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources. Despite its growth major concerns such as its reliability, security and stability remain [2].

Infrastructure as Service (IaS) model of cloud computing provides users with computing infrastructure in the form of Virtual Machines (VM). Following [3], 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. We extend the model in [3] by considering scenarios where 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. Completing unreliable jobs do not count as proper work done.

We adopt a classic reliability and security tool of packet scanning to detect these malicious packets [4]. 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.

The system is *stable* if the queues do not tend to increase without bound. We aim to characterize the maximum arrival rates of genuine and unreliable requests under which there is an algorithm to maintain the stability of the system and to develop such algorithm if it exists.

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 [5]. Optimizing other costs of running the services has been considered [6], [7]. The algorithms we propose here are inspired by the MaxWeight algorithm analyzed in [8] 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 [9], [10]. Detecting and distinguishing unreliable or malicious from genuine requests and a number of approaches have been proposed [4].

B. Our Contributions

We propose a lightweight robust framework to manage workload in clouds under unreliable workload scenarios. Extending the model in [3], [8], we propose to detect unreliable part of the traffic by scanning only some specifically selected jobs without sacrificing too much resources.

- We propose a theoretical model to capture the essence of this conditional scanning and show that under a certain system capacity region and stochastic arrival pattern of genuine and unreliable jobs there exists an algorithm, called RobustMaxWork, that manages the workload while maintaining queue stability (i.e., the queue is bounded and does not grow to infinity size).
- We show how to efficiently compute the optimal scanning strategy (vector). We propose several distributed versions of RobustMaxWork and discuss various extensions of theoretical results.
- We evaluate the algorithms and the proposed model using extensive simulations.

II. Model

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. We consider a set of n networked servers. physical machines), each having its own resources that it can distribute among jobs, that is, for each resource it has a fixed capacity.

Jobs. A job is specified by its type and length. Since there are limited number of virtual machine types, we only consider limited number, denoted by J, of job types. There are I different lengths of jobs possible: $L_1, \ldots, L_I$ We consider online random arrival model, where new jobs arrive independently of each other and are identically distributed across all time slots, and the variance of arrival length is finite. Let $\lambda_{i,j}$ denote 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. Each server can process a set of jobs simultaneously provided sum of all jobs does not exceed its capacity Processing jobs is done in synchronous time steps, also called rounds. The whole system capacity is a linear sum of capacities of all the servers. 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 tools. 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. Unreliable jobs are also arrive independently of each other and are identically distributed across all time slots, and the variance is finite. We assume that we have a scanning tool that, given a job, can detect whether it is a genuine user request (a *good job*) or a unreliable request (an *unreliable job*). Each scanning takes 1 time slot per job and requires same resources as the original job (scanning is done on the same virtual machine).

Central scheduler. We consider a central scheduler with a queue of all injected, but not yet finished, jobs. 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. The centralized algorithm, called RobustMaxWork, will be introduced in Section III.

Distributed scheduler. Here all servers maintain separate queues for jobs of type j, therefore when a job arrives we decide which server to route the job. Each server runs locally a protocol RobustMaxWork with respect to its local queues.

Notation.

- n denotes the number of servers in the cloud;
- I denotes the number of different job lengths;
- J denotes the number of different job types;
- $A(t) = (A_1(t), \ldots, A_J(t))$ denotes the vector of sets of type-j jobs, for $j \leq J$, which arrive to the system in the beginning of time slot t;
- ℓ_j is a (random) length of arriving type-j jobs.

In addition, regarding RobustMaxWork algorithm:

- $\alpha_{i,j}$ is a probability of scanning type-j job of length L_i; the algorithm may implement a specific scanning strategy, i.e., use a specific vector α;
- $Q(t)$ denotes the vector of queue lengths (i.e., sum of lengths of jobs in the queue) for each type of jobs in the beginning of time slot t;
- $Q_j(t)$ denotes the total length of users' and unreliable type-j jobs, for $j \leq J$, in the beginning of time slot t;
- $X_j(t)$ is the total length of queued type-j jobs that will not be scanned, taken in the beginning of time slot t (i.e., the algorithm scanned them already or decided not to scan them at all);
- $Y_j(t)$ is the total length of queued type-j jobs that will be scanned, taken in the beginning of time slot t (i.e., the algorithm has already decided to scan them, but has not scanned them yet);
- $Z_j(t) = Z_j(Q(t))$ is the expected time required to process type-j jobs stored in queue in the beginning of time slot t
- a_j is the expected time required to process type-j jobs that arrive in one time slot (see formula in Section **??**).

Whenever time slot t is clearly fixed or understood from the context, we may omit an argument t from the formulas.

Scanning strategies. We compare the following strategies:

- Scan-None — always executes a job without scanning, i.e., $\alpha_{i,j} = 0$ for all i, j;
- Scan-All — scans all jobs except those with processing time shorter or equal to the scanning time (recall that scanning takes 1 time slot), i.e., $\alpha_{i,j} = 0$ for $L_i \leq 1$ and $\alpha_{i,j} = 1$ otherwise;
- Scan-Opt — If there exists a vector of scanning frequencies $\alpha^{(0)}$ such that given job arrival rates λ and κ are inside the capacity region (i.e., $(1+\epsilon)a(\alpha^{(0)}, \lambda, \kappa) \in co(S)$), then there exists a vector of scanning frequencies $\alpha^{(1)} \in \{0,1\}^{I \times J}$ such that these job arrivals are inside the capacity region ($(1+\epsilon)a(\alpha^{(1)}, \lambda, \kappa) \in co(S)$).

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., $\limsup_{t \to \infty} E[\sum_j Q_j(t)] < \infty$.

III. Main Algorithm — Centralized Version

Algorithm RobustMaxWork (Algorithm 1) is parametrized by: scanning vector $\alpha = (\alpha_{i,j})_{i \leq I, j \leq J} \in [0,1]^{I \times J}$, vector of rates of genuine user's requests $\lambda = (\lambda_{i,j})_{i \leq I, j \leq J}$, and vector of rates of unreliable requests $\kappa = (\kappa_{i,j})_{i \leq I, j \leq J}$. 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 $Z_j(t)$.

It takes X_j time to process jobs that will not be scanned. Jobs contributing to Y_j will need to be scanned, requiring $Y_j \cdot E(1/\ell_j)$ expected time. In expectance $\lambda_j/(\lambda_j + \kappa_j)$ fraction of

Algorithm 1 RobustMaxWork(λ, κ, α)

```
X ← 0⃗   // jobs that will not be scanned
Y ← 0⃗   // jobs that will be scanned
Q ← 0⃗   // all jobs
loop
    new time slot begins
    for all new type-j job τ_{i,j} of length L_i do
        r ← random value from [0; 1]
        if r < α_{i,j} then                    // τ_{i,j} to be scanned
            Y_j ← Y_j + L_i
            Q_j ← Q_j + L_i
        else                                   // τ_{i,j} not to be scanned
            X_j ← X_j + L_i
            Q_j ← Q_j + L_i
        end if
    end for
    for all j do
        Z_j ← X_j + Y_j(λ_j/(λ_j + κ_j) + E(1/ℓ_j))
    end for
    N' ← arg max_{N∈S} Σ_j N_j · Z_j
    for all j do
        for k ≤ N'_j do
            Process_job(j)
        end for
    end for
end loop
```

scanned jobs are genuine, so 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 scanned jobs are fake and after scanning they take no more processing time. Therefore: $Z_j(t) = X_j(t) + Y_j(t) \cdot (\lambda_j/(\lambda_j+\kappa_j) + E(1/\ell_j))$.

The algorithm then computes Z_j (c.f., the second **for all** loop in Algorithm 1) and finds configuration N from the set of feasible server configurations $\mathcal{S}$ that maximizes the sum $\sum_{j=0}^{J} Z_j(t)N_j$, i.e., the objective in each time slot t is:

$$\max_{N\in S} \sum_{j=0}^{J} Z_j(t)N_j \ .$$

This configuration is denoted by N'. The intuition is that the more jobs of a given type accumulate, the more weight should be put to scheduling this job type to prevent further accumulation. Z_j here is the weight given to jobs of type j.

Finally, 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), c.f..

a) 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:

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

- 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 (vice versa for Y_j).

IV. DECENTRALIZED IMPLEMENTATIONS

Below we describe four specification of the decentralized RobustMaxWork based on different routing policies.

Algorithm A: RobustMaxWork_JSQ. Joint 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. Joint 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_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 D: 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

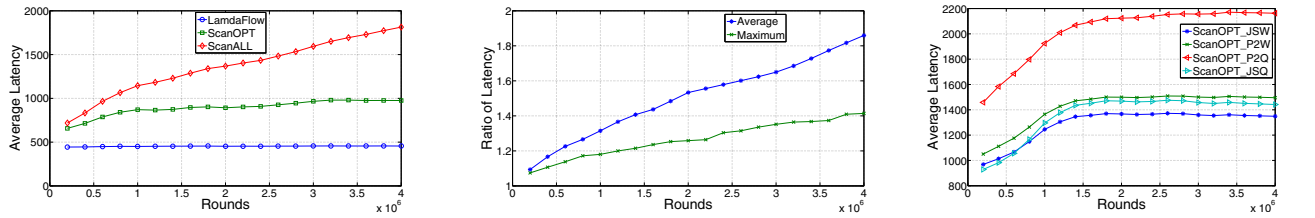
A. Experiment Setting

Experiments are based on Maguluri et al. [3].

Servers and VMs. We consider a server with 30 GB memory, 30 EC2 computing units and 4000 GB storage space. There are 100 identical servers in the cloud. 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 $(0,1,1)$.

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 the server capacity area 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.

Using the same procedure, we define an unreliable workload using a generic arrival vector $\kappa^* = (0.7, 0.01, 0.01)$. Each of



(a) Comparison of average latency of LambdaFlow, ScanOPT and ScanALL strategies.

(b) Ratio of ScanAll to ScanOpt latency

(c) Average latency using ScanOPT_JSQ, ScanOPT_JSW, ScanOPT_P2Q and ScanOPT_P2W

Figure 1. Comparison of different scheduling approaches.

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. Since there are 100 homogenous servers, 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. 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.

In the first part, we output the results of the above measurements for the following four centralized protocols:

LambdaFlow: RobustMaxWork applied for genuine flow λ only, and no scanning is applied (i.e., $\vec{\alpha} = \mathbf{0}$);

ScanOPT: RobustMaxWork applied for simultaneous genuine and malicious flows, with scanning defined by vector α^*;

ScanALL: RobustMaxWork applied for both genuine and malicious flows, scanning all jobs of size bigger than 1 (i.e., for every $j = 1, 2, 3$, $\alpha^{\vec{}}_{1,j} = 0$ and $\alpha^{\vec{}}_{i,j} = 1$ for every $i > 1$);

We expect the first two executions to be stable . Which would justify our research quest for searching of suitable scanning vector. We also display ratios between the second and the third executions — the stable and the potentially unstable one.

We also study how different routing protocols influence stability, when applied to the RobustMaxWork with the optimally selected scanning vector α^* i.e. ScanOPT_JSQ, ScanOPT_JSW, ScanOPT_P2Q, and ScanOPT_P2W.

B. Results

We provide simulational results regarding changes of latency over time for the different scanning strategies used: ScanALL, ScanOPT comparing them with the execution LambdaFlow. In Figure 1a, the average latency of the ScanALL strategy grows rapidly, while it stabilizes for the ScanOPT. In Figure 1b we analyzed the ratio of ScanALL to ScanOPT latencies, and both are increasing. It indicates that ScanALL is becoming worse and worse over time.

Figure 1c compares average latency of the four decentralized algorithms using ScanOPT strategy. The best performing algorithm is the one based on JSW, followed by JSQ the two algorithms based on the power of choices, P2W and P2Q respectively.

VI. Conclusions, Extensions and Open Problems

Further reading on decentralized scheduling, non-preemptiveness, algorithms without knowledge of arrival rates, probability of successful scanning and detailed results are presented in the full paper available at http://arxiv.org/abs/1705.02671.

Acknowledgment

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

References

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

[2] 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.

[3] 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.

[4] 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.

[5] 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.

[6] 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.

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

[8] 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.

[9] 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.

[10] 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.