Stability Under Adversarial Injection of Dependent Tasks*
(Extended Abstract)

Vicent Cholvi[1], Juan Echagüe[1], Antonio Fernández Anta[2], and Christopher Thraves Caro[3]

[1] Universitat Jaume I, Spain
{vcholvi,echague}@uji.es
[2] IMDEA Networks Institute, Madrid, Spain
antonio.fernandez@imdea.org
[3] Departamento de Ingeniería Matemática, Facultad de Ciencias Físicas y Matemáticas, Universidad de Concepción, Chile
cthraves@ing-mat.udec.cl

Abstract. In this work, we consider a computational model of a distributed system formed by a set of servers in which jobs, that are continuously arriving, have to be executed. Every job is formed by a set of dependent tasks (i. e., each task may have to wait for others to be completed before it can be started), each of which has to be executed in one of the servers. The arrival and properties of jobs are assumed to be controlled by a bounded adversary, whose only restriction is that it cannot overload any server. This model is a non-trivial generalization of the Adversarial Queuing Theory model of Borodin et al. and, like that model, focuses on the stability of the system: whether the number of jobs pending to be completed is bounded at all times. We show multiple results of stability and instability for this adversarial model under different combinations of the scheduling policy used at the servers, the arrival rate, and the dependence between tasks in the jobs.

Keywords: Tasks scheduling · task queuing · dependent tasks · adversarial queuing models · stability.

1 Introduction

In this work, we consider a model of jobs formed by dependent tasks that have to be executed in a set of servers. The dependencies among the tasks of a job restrict the order and time of their execution. For instance, a task q may need some information from another task p, so that the latter must complete before q can be executed. This model embodies, for instance, the dynamics of Network Function Virtualization (NFV) systems [2, 5] or Osmotic Computing (OC)

* This submission is a short paper. This work was partially funded by the Spanish grant TIN2017-88749-R (DiscoEdge), the Region of Madrid EdgeData-CM program (P2018/TCS-4499), and the NSF of China grant 61520106005.

[4]. In a NFV system, network services (which are job types) are specified as service chains, obtained by the concatenation of network functions. These network functions are dependent computational tasks to be executed in the NFV Infrastructure (e.g., servers distributed over the network). In an OC system, an application is divided into microservices that are distributed and deployed on an edge/cloud server infrastructure. The user requests (jobs) involve processing (tasks) in several of these microservices, as defined by an orchestrator that takes into account the dependencies between the microservices. In that line, it also encompasses a number of features of Orchestration Languages (see, for instance, [3]), which propose a way to relate concurrent tasks to each other in a controlled fashion: the invocation of tasks to achieve a goal, the synchronization between tasks, managing priorities, etc.

In our model, we consider a dynamic system in which job requests (or jobs for short) are continuously arriving. Each job contains the whole specification of its dependent tasks: the collection of tasks to be executed, the server that must execute each task, the time the execution incurs, the dependencies among tasks, etc. In our model we assume the existence of an adversary that has full control of the job requests arrivals, and the specification of their tasks. The only restriction on the adversary is that no server can be overloaded in the long run (while some burstiness in the load is allowed). In this adversarial framework, the objective is to achieve stability in the system. This means that the system is able to cope with the adversarial arrivals, maintaining the number of pending job requests in the system bounded at all times. (This usually also implies that all the job requests are eventually completed.)

The study of the quality of service that can be provided under worst-case assumptions in a given system (NFV or OC, for instance) is important in order to be able to honor Service Level Agreements (SLA). The positive results we obtain in this paper show that it is possible to guarantee a certain level of service even under pessimistic assumptions. These results can also be used to separate resource allocation and scheduling as long as the resource allocation guarantees that servers are not overloaded, since we prove that it is possible to guarantee stability in this case.

2 Model

In this section, we define the *Adversarial Job Queueing* (AJQ) model. The AJQ model is designed to analyze systems of queueing jobs. The three main components of an AJQ system $(S, P, \mathcal{A})$ are:

- a set $S = \{s_1, s_2, \ldots, s_n\}$ of n servers,
- an *adversary* $\mathcal{A}$ who injects jobs in the system, and
- a scheduling *policy* P, which is the criteria used by servers to decide which task to serve next among the tasks waiting in their queues.

The system evolves over time continuously. In each moment, the adversary may inject jobs to the system while the servers process those jobs. In each moment as

well, some tasks may be waiting to be executed, others may be in process, and others may be completed. A job is considered *completed* when all its tasks are completed. When a job is completed, all its tasks disappear from the system.

Each job $\langle K, f^K \rangle$ consists of a finite set K of tasks and a function f^K that determines dependencies among the tasks. (For simplicity we will denote the job $\langle K, f^K \rangle$ by its task set K.) Let $K = \{k_1, k_2, k_3, \ldots, k_{l_K}\}$ be a job, where each k_i is a task of K. The integer l_K denotes the number of tasks of K. Each task k_i is defined by three parameters $\langle s_i^K, d_i^K, t_i^K \rangle$. The parameter $s_i^K \in S$ is the server in which k_i must be executed. The parameter $d_i^K \geq 0$ is the *activation delay* of k_i. The parameter $t_i^K > 0$ is the *processing time* of k_i, i. e., the time server s_i^K takes to execute task k_i.

Let $(S, P, \mathcal{A})$ be an AJQ system. Let $T_{max} := \max_{i,K}\{t_i^K\}$, $T_{min} := \min_{i,K}\{t_i^K\}$, $D_{min} := \min_{i,K}\{d_i^K\} \geq 0$, and $D_{max} := \max_{i,K}\{d_i^K\}$, be the maximum and minimum time required to complete a task, and minimum and maximum activation delay, respectively, among all tasks of any job injected in the system. We assume that all these quantities are bounded and do not depend on the time.

Feasibility. Let $\mathcal{P}(K)$ be the *power set* of K, i. e., the set of all subsets of K. Furthermore, let $\mathcal{P}^2(K)$ be the *second power set* of K, i. e., the set of all subsets of $\mathcal{P}(K)$. Given a job K, a *feasibility function* $f^K : K \to \mathcal{P}^2(K)$ determines which tasks of K are *feasible*, which means that they are ready to be executed, once the activation delay has passed. Let $f^K(k_i)$ be equal to $\{A_1, A_2, \ldots, A_{\ell_i}\}$. The sets A_x for $1 \leq x \leq \ell_i$ are called *feasibility sets* for k_i. Then, the task k_i is *feasible* at a time t if there exists a feasibility set A_x for k_i such that all tasks in A_x have been completed by time t. Otherwise, k_i is *blocked,* and still has to wait for some other tasks of K to complete before becoming feasible.

The activation delay d_i^k of a task k_i represents a setup cost, expressed in time, that k_i must incur once it becomes feasible and before it can start to be processed. If t is the time instant at which k_i becomes feasible, then k_i will incur its activation delay during time interval $[t, t + d_i^k]$. Hence, it cannot be executed during such interval, in which we say that task k_i is a *delayed* feasible task (or only *delayed task*). When k_i completes its activation delay at time $t + d_i^k$, it can be served, and since that moment will be referred to as an *active* feasible task, or simply *active task.* Equivalently, a feasible task is active if it has been feasible for at least d_i^k time. A job with at least one feasible (resp., active) task will be referred to as a *feasible* (resp., *active*) job.

The feasibility function provides the AJQ model with a high level of flexibility at the time of forcing the execution sequence of the tasks of a job. For instance, it allows the coexistence of AND dependencies and OR dependencies.

Doability. Let K be a job and k_i be a task of K. We say that k_i is an *initial* task of K if $\emptyset \in f^K(k_i)$. Observe that all initial tasks k_i are automatically feasible at the time the job K is injected, and they become active d_i^K time later.

We assign a *layer* $\lambda(K, i)$ to the tasks k_i of a job K as follows. All initial tasks have layer $\lambda(K, i) = 1$. For any $j > 1$, a task k_i is assigned layer $\lambda(K, i) = j$ if it is not feasible when all tasks of layers $1, ..., j - 2$ are completed, but it becomes feasible when additionally the tasks of layer $j - 1$ are completed. Let $\lambda_K \leq l_K$

denote the number of layers of job K. If a task k_i has layer $\lambda(K,i)=\ell$, then there is a feasibility set $A_x \in f^K(k_i)$ for k_i such that $A_x \subseteq \{k_j \in K : \lambda(K,j) < \ell\}$.

Observe that the above definition does not guarantee that all tasks of a job will be assigned a layer. In fact, it is not hard to create jobs that have tasks dependencies (e.g., cyclic dependencies) that prevent some tasks from being assigned a layer. This will prevent a job to complete. We want every job to be potentially completed. Therefore, we impose some restrictions over every feasibility function.

Definition 1. *Let K be a job and $f^K : K \to \mathcal{P}^2(K)$ be its feasibility function. We say that K is* doable *if every task k_i of K can be assigned a layer.*

It is worth mentioning that, deciding whether a job is doable or not as defined can be computed in polynomial time with respect to the size of the job (that takes into account the number of tasks and the size of the feasibility function). Indeed, layer 1 can be computed by checking which tasks have the empty set as a feasibility set. Then, a simple recursive algorithm computes all tasks in layer i using all the tasks in layers $1, 2, \ldots, i-1$.

The next proposition says that the doable condition is necessary for a job to be completed, and that it is also sufficient if it is the only job injected in a system and the scheduling policy is work conserving.

Proposition 1. *Let $(S,P,\mathcal{A})$ be a system where the adversary $\mathcal{A}$ injects only one job K and P is work conserving. Then, K can be completed if and only if K is doable.*

Topologies. Let K be a job, and tasks $k_i, k_j \in K$. We say that k_i *depends* on k_j if there exists a feasibility set $A_x \in f^K(k_i)$ for k_i such that $k_j \in A_x$. The *skeleton* of a job K is the directed graph $H_K = (V,E)$, where $V(H_K) := \{k_1, k_2, \ldots, k_{l_K}\}$ and $E(H_K) := \{(k_j, k_i) : k_i \text{ depends on } k_j\}$. It is worthwhile to mention that a skeleton does not define the feasibility function of a job.

The *topology of a job K* is the directed graph obtained by mapping the skeleton of K into the set of servers, where each task k_i is mapped into its corresponding server s_i^K. Given a system $(S,P,\mathcal{A})$, the *topology of the system* is the directed graph obtained by overlapping the topology of all jobs injected by $\mathcal{A}$ in the system.

Scheduling policy. We assume that each server has an infinite buffer to store its own queue of tasks. Every active task waits in the queue of its corresponding server. In each server, a scheduling policy P specifies which task of all active tasks in its queue to serve next. We assume that scheduling policies are greedy/work conserving (i. e., a server always decides to serve if there is at least one active task in its queue). Examples of policies are *First-In-First-Out* (*FIFO*) which gives priority to the task that first came in the queue, or *Last-In-First-Out* (*LIFO*) which gives priority to the task that came last in the queue.

Adversary. We assume that there is an adversary $\mathcal{A}$ who injects *doable* jobs into the system. In order to avoid trivial overloads, the adversary is bounded in the following way. Let $N_s(I)$ be the total load injected by the adversary during time interval I in server s (i. e., $N_s(I) = \sum t_i^K$ over all jobs K injected during

I and tasks k_i such that $s_i^K = s$). Then, for every server s and interval I the adversary is bounded as

$$N_s(I) \leq r|I| + b, \tag{1}$$

where $0 < r \leq 1$ is called the *injection rate*, and $b > 1$ is called the *burstiness* allowed to the adversary. Observe that (1) implies $\max_{i,K}\{t_i^K\} \leq b$, since jobs are injected instantaneously. An adversary that satisfies (1) is called a *bounded* (r, b)-adversary, or simply an (r, b)-adversary.

The system formed by an (r, b)-adversary $\mathcal{A}$ injecting doable jobs in the set of servers S using the scheduling policy P is called an AJQ system $(S, P, \mathcal{A})$. The number of active tasks in the queue of server s at time t is denoted $Q_s(t)$.

Definition 2. *An AJQ system $(S, P, \mathcal{A})$ is* stable *if there exists a value M such that $Q_s(t) \leq M$ for all t and for all $s \in S$, where M may depend on the system parameters (adversary, servers, and jobs characteristics) but not on the time.*

Definition 3. *A policy P is* universally stable*, if any system $(S, P, \mathcal{A})$ is stable against any (r, b)-adversary $\mathcal{A}$ with rate $r < 1$.*

3 Our Results

Finally, we present our results. Due to lack of space, we omit the proofs. The proofs and complementary figures are shown in [1].

Theorem 1. *Let LIS (Longest-In-System) be the scheduling policy that gives priority to the task (and hence the job) that has been in the system for the longest time. Then an AJQ system $(S, LIS, \mathcal{A})$, where $\mathcal{A}$ is an (r, b)-adversary with $r < T_{min}/(T_{max} + D_{max})$, is stable.*

Theorem 2. *FIFO and LIFO are unstable for every $r > 0$.*

These two theorems prove for AJQ systems that the scheduling policy used has an important impact on performance. In particular, they show that popular policies like *FIFO* and *LIFO* may not be the best choice. These were facts known for packet scheduling in networks, but it was not obvious they would also hold in AJQ systems. In the following theorem, we show that a feed-forward topology (i.e., a topology where there are no cycles) is a sufficient condition for stability in a system.

Theorem 3. *Let $(S, P, \mathcal{A})$ be an AJQ system with feed-forward topology. Then, for any (greedy) policy P and any (r, b)-adversary $\mathcal{A}$ with injection rate $r \leq 1$, the system $(S, P, \mathcal{A})$ is stable.*

This result has implications in the design of distributed systems. For instance, it implies that, if an order is defined among servers, and jobs order their tasks always respecting this order, then the system is stable using any scheduling policy, even at full load.

References

1. Cholvi, V., Echagüe, J., Fernández Anta, A., Caro, C.T.: System stability under adversarial injection of dependent tasks. arXiv:1910.01869v1 (2019)
2. Herrera, J.G., Botero, J.F.: Resource allocation in nfv: A comprehensive survey. IEEE Transactions on Network and Service Management **13**(3), 518–532 (2016)
3. Kitchin, D., Quark, A., Cook, W.R., Misra, J.: The Orc programming language. In: Proceedings of FMOODS/FORTE 2009. Lecture Notes in Computer Science, vol. 5522, pp. 1–25. Springer (2009)
4. Villari, M., Fazio, M., Dustdar, S., Rana, O., Ranjan, R.: Osmotic computing: A new paradigm for edge/cloud integration. IEEE Cloud Computing **3**(6), 76–83 (2016)
5. Yi, B., Wang, X., Li, K., Huang, M., et al.: A comprehensive survey of network function virtualization. Computer Networks **133**, 212–262 (2018)