Improving the utilization of an elastic resource: a client-side approach

Augusto Ciuffoletti
Department of Computer Science - University of Pisa - Italy
augusto@di.unipi.it

Abstract

Resource management is traditionally addressed by policies implemented inside the resource provider. Here we study the problem with an attitude that is specular but complementary, which consists in designing a distributed client-side access regulation algorithm that improves the utilization of an elastic resource.

The introduction of elastic resources — a feature of the cloud computing paradigm — complicates their management since, when the workload applied on the resource varies (for instance with the number of users) the resource automatically follows such variations with its capacity. But the presence of an extra computational cost related with capacity variations motivates a non linear, lazy response, that penalizes dynamic environments. Hence the interest for an algorithm that shapes the production of service requests on the client side.

To make our investigation more adherent to a practical environment, we introduce a real time requirement: each client must have access to the service at least every π time units. Examples of this requirement, that features a bounded degree of asynchrony, are found, for instance, in network streaming applications: stream chunks must feed the input buffer at the destination.

The algorithm we investigate is based on the random walk of a token. To evaluate the range of applicability of the algorithm, we define an analytic model of its stochastic behavior — described by a non-Markov process — and then we compare its performance with a benchmark algorithm, representative of an effective solution that is often used in practice.

Keywords: *elastic resource, cloud computing, token-based protocols, stochastic diffusion, non-markov process.*

1 Introduction

Resource virtualization has made practically feasible the implementation of an *elastic resource* that dynamically adapts its performance. This capability is one facet of the "cloud computing" paradigm (see NIST document [10]): elasticity is an optional feature of a resource that implements a service *in the cloud* ("rapid elasticity", in NIST terms). As of today, leading *Infrastructure as a Service* (IaaS)

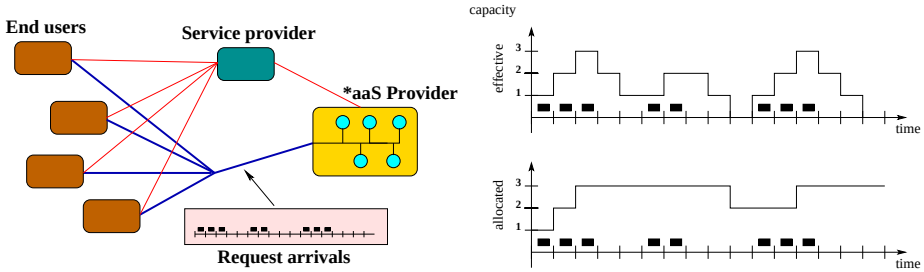


Figure 1. A typical *aaS architecture: application data path in blue

providers implement such feature: for instance, the corresponding Amazon AWS term is *auto-scaling* [1].

An instance of an elastic resource is found in a web server that makes available weather reports, snow levels and web-cam images of a ski resort: depending on more or less predictable variables (like weekday, weather, season etc.), the workload of the server is exposed to significant variations. With the adoption of an elastic resource for the web server, it is possible to use (and pay for) the amount of computing power strictly needed by web service operation, while preserving the Service Level. This is a remarkable advantage compared with the case of a statically allocated infrastructure that must resolve the trade-off between the waste of resources when the query rate is low, and service degradation when there is a high demand.

The above use case covers a service characterized by a non-critical quality, and the variability is associated with a computing resource: but our discussion covers a wider range. Service quality may become critical, if the service has a role in rescue or emergency related activities, and the variable resource may be related to the interconnection network (e.g, streaming of an IPTV channel) or storage (e.g., hosting of an inference engine applied to a dynamic database).

In figure 1 we see the actors in our milieu:

- the ***aaS provider** that implements the elastic shared resource: the relevant fact for us is that it is made available as a service, and we disregard whether it is encapsulated in an Infrastructure, or Platform, or Service;
- the **service provider**, the mediator that implements the service using the shared resource: in our example the ski resort portal plays this role;
- the **users**, that utilize the shared resource across the service provider; they are considered as a multiplicity of agents that may coordinate among them according with a peer to peer paradigm.

To model the dynamics of the workload supported by the elastic resource, we assume that the operation of the users results in a sequence of service requests randomly distributed in time, that are filled by jobs supported by the elastic resource: even without anticipating a formal statement of their distribution, an example conveys the feeling of the effects of a variable load on an elastic resource.

The time diagram in the upper left corner of figure 1 is obtained by randomly distributing 8 service requests over 16 time frames: a dark rectangle indicates the

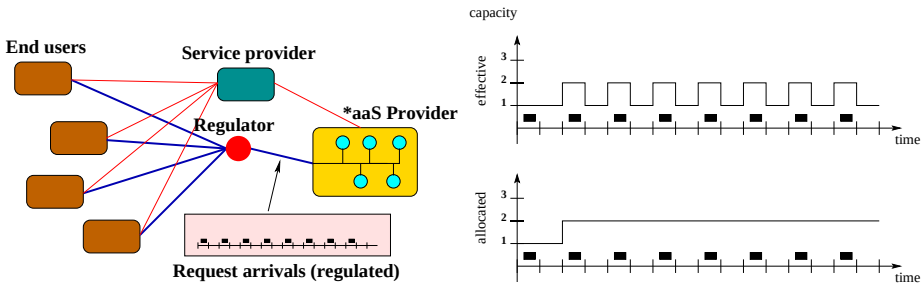


Figure 2. Introduction of a regulation on service requests

arrival time of a request. Each of them takes three time frames to complete, so that, in the example, the first instance starts on time frame one and terminates on time frame three, the second starts on time frame two etc. The total capacity used by the jobs during a time frame is the sum of the capacity consumed by all the jobs active during that time frame: the result is shown in the stairs diagram in the same graph.

The reaction of the elastic resource to load variations is a matter of trade-off: how promptly the size of the allocated resource responds to such variations directly determines the quantity of resources that are allocated to the service. A trade-off is justified since resource resizing has a non negligible cost (an up to date investigation on the topic is in [14]): so that a lazy provider minimizes resizing costs, but incurs resource waste or service degradation, while a reactive one consumes the exact amount of resource needed at each time, but dissipates resources in the process of decreasing/increasing the shared resource.

So that, to proceed with our example, we need to make explicit the *allocation policy* of the provider; we use a simple rule reminiscent of network congestion control algorithms that envisions a fast response to increments, and a slow response to decrements. Such a rule preserves service quality, but incurs some waste of resource capacity. It is stated as follows:

1. when requests exceed the capacity already allocated, additional capacity is immediately provided;
2. otherwise, the allocated capacity corresponds to the Exponentially Weighted Moving Average (with $k = 8$) computed on historical data.

In the lower right corner of figure 1 we show the effect of this *buffering strategy* on our sample workload. The integral of the stair graph, measured as (units of capacity x time frames), represents the cost of the execution of the 8 instances of activity: the provision of extra resource capacity is a frequent event (e.g., from time frame 4 to 7). The net result is that, to provide 24 units of workload (3 for each of the 8 service instances), 42 units of (capacity x time) are used, wasting 43% of the allocated resources.

There are basically two options opened to improve resource utilization:

- refine the resource side policy outlined above; see [6] for a work following this approach, that consists in applying an algorithm that extracts regularities from the flow of requests in order to anticipate them and to improve their manage-

ment;

- design a client side algorithm that shapes the time distribution of requests arrival, to make it more regular.

In this paper we follow the second approach: its soundness is explained below using our example.

We first introduce a simplistic rule that regulates requests arrivals so that they are evenly distributed in time: we disregard the fact that this result is not always reachable, and possibly unacceptable for buffering reasons. We only consider it as the best result reachable in the direction of making requests regularly spaced in time. Using this approach coupled with the resource side policy explained above, the **aaS provider* uses 32 units of capacity instead of 42 to support the same benchmark workload: the waste rate lowers at 25% of the allocated resources, and the saving, with respect to the unregulated case, is 24%.

The key point is that a client-side access regulation is beneficial since it avoids load peaks that induce the provision of redundant capacity, with a financial impact. The approach is original, and specular compared with the one based on server-side resource management. But the two approaches may coexist in the same practical solution, and mutually enhance the overall performance: a client side regulation smoothly integrates, and does not conflict, with server side policies, and this makes their combination appropriate for private/hybrid clouds.

The goal of this paper is to put on a formal ground the ideas behind the example outlined above.

To define the flow of client requests we target a specific kind of application: those that require periodic access to the (elastic) resource, with timing guarantees in the *Service Level Agreement* they stipulate with the service provider. The flow of requests is defined by consequence as a *stochastic process*, and the regulation algorithm works on these premises.

The regulator we propose is based on a token performing a random walk among *end-users*: the end-user holding the token performs the service request, and then it delivers the token to another end-user selected at random. To meet timing requirements, a service request can be triggered also by a local watchdog timer, independently from the receipt of the token.

To evaluate the solution, it is of little use to compare it only with an unregulated stream of events: the result can be anticipated by analysis. Instead, we introduce a benchmark regulation algorithm: a simple one, frequently used in practice, that has the effect of shaping the time distribution of service requests. The algorithm consists of randomizing the lapse between two successive requests from the same host.

The comparison between the random walk algorithm and the benchmark is carried out with analytical tools: the relevant figures to this end are the average and the variance of inter-arrival times, obtained from the stochastic processes that model the events produced by the two algorithms. The analysis is non-trivial, since the timed generation of events turns out to be a non-Markov process. The analytical results are also checked against simulation.

The advantage of an analytical approach is that sometimes results are extended with little effort by corollaries. In our case we obtain a negative result for a modified version of the basic algorithm, showing that the variation does not improve the performance.

An implementation that fits our approach is client-based and distributed. It envisions the client side execution of server code, and fits the *Web Service* framework. In this perspective, many tools that ensure security exist (see for instance [11]). We exclude that the service provider implements the regulator inside the elastic resource, or that it configures the elastic resource with an embedded request buffering policy, for the reason that the Web Service interface must be kept simple: this conforms to indications coming from emerging standards like the OGF *Open Cloud Computing Interface* [13] — although its core model in [12] is, in principle, sufficiently flexible to allow a similar configuration.

The rest of the paper is organized as follows. In sect. 2 we introduce the system model and the random process that represents non-regulated resource access events. With these formal guidelines we study the benchmark algorithm (in sect. 3), and our *random walk* algorithm (in sect. 4). Their comparison (in sect. 5) proves that the random walk algorithm exhibits a narrower distribution of the workload during a frame. An appendix is dedicated to the discussion of the random walk model: a detailed analysis and a formal proof are motivated by the presence of events triggered by a timeout, that are incompatible with a plain Markov process.

The many details of a token passing algorithm, like initialization, token recovery etc., are not dealt with in this paper, but there is a vast literature on the topic. For the sake of completeness, we refer to a token passing protocol specifically designed for the purpose, that is exhaustively described in [4] together with the results of a prototype implementation running in the Internet.

2 System model

The subject of our access regulation problem is an elastic *resource* shared among a set of $\mathbf{N}$ *end-users*. A *service instance* entails a resource consuming activity, assimilated to a *job run* in the case of a computing resource: all service instances are assumed to have the same cost in terms of consumed resource.

A *timing requirement* states that each end-user must be allowed to place an *service request* at least every π time units, the *period*: this is a deterministic bound, related neither with an "*eventually*", nor with a "*high probability*".

It is a restatement — in *cloud computing* terms — of the well-known *periodic scheduling* problem stated by Liu and Layland in [8]. In some sense our problem is also similar to that addressed in [7]: in that case the *aaS provider needs to meet a *Service Level Agreement* that explicitly indicates a limit to the response time from the cloud. In our case we assume a less restrictive, but equally firm, time limit.

The stochastic variable that we want to analyze is the workload on the shared resource, corresponding to the regulated stream of requests: we model it with the

discrete concurrency $\mathbf{c}$, a stochastic process that corresponds to the number of requests during a time frame of constant length. The specifications of the regulation activity mandates that the number of requests during a time *frame* of τ time units is concentrated around the value $\gamma = \frac{N\tau}{\pi}$: in probabilistic terms, the mean should be around γ, and the variance should be minimized.

The value γ corresponds to the rate between the number of users (N), and the number of *frames* during a *period* ($\frac{\pi}{\tau}$): the case of $\gamma = 1$ corresponds to the situation where there is exactly the time needed to run a request during each frame from all users. When $\gamma < 1$, π exceeds the time needed to enable each and every user during one period, while $\gamma > 1$ indicates that some frames necessarily contains concurrent requests: the stochastic process $\mathbf{c}(i)$ (the *concurrency*) represents the number of service requests issued during the i-th frame.

We observe that any application environment is described by the γ alone. As seen above, this parameter is a simple function of the three constants that describe a specific application environment: so that apparently heterogeneous environments, that are characterized by different values of N, π and τ, may correspond to the same problem instance when their combination correspond to the same value of γ. The analytic study of the algorithms given in the next section is therefore exhaustive, since it covers all values of γ.

Our discussion proceeds with the analysis of an algorithm (the *random scheduling*) that is frequently used and that fits our purpose: its performance is later used as a benchmark.

3 Random scheduling

This simple algorithm has the effect of randomizing the occurrence of timed events produced by independent sources: it is frequently used to remove spurious synchronizations of sequences of timed events that are possibly triggered by the same source. For instance, if many users periodically refresh a piece of data from a remote server, when many users start their operation at the same time they will periodically produce a load peak on the server. But there are cases where the application itself tends to synchronize such kind of events. The purpose of the random scheduling is to avoid unwanted event clustering that may degrade system throughput. It is adopted, for instance, to avoid congestion in the clock synchronization protocol IEEE 1588 (see page 105 in [2]).

The operation of the algorithm consists of anticipating the event scheduling of a random lapse. This is obtained by decrementing the timer. The final effect is that the timing requirements are met, but, after a few iterations, the exact timing of the event is not predictable with a precision better than the period π. We call k the parameter that determines the randomization, that we set to 5% of the period π. An increment of k accelerates the randomization process, but increases the frequency of events. The value of 5% is considered as a trade-off that privileges a low workload, which makes a demanding benchmark in our case.

The following is a programmer-friendly description of the algorithm:

$k = 0.05$
while $true$ do
    $jitter = uniform(0, k) * \pi$
    $sleep(\pi - jitter)$
    $background(run(\mathbf{S}))$
od

So the events tend to be uniformly distributed on time, and any lapse of $\pi * (1 - (jitter/2))$ time units contains on the average N service requests. This process corresponds to the *concurrency* defined in the previous section and, when N is large as in our case, it can be approximated with a Poisson process, whose intensity is $N/(\pi * (1 - (jitter/2)))$ requests per time unit: we indicate with $\mathbf{c}_{rs}$ this process. The fundamental figures for our analysis are the mean and variance of $\mathbf{c}$: in the Poisson approximation they share the same value, $N\tau/(\pi * (1 - (jitter/2)))$, that we approximate with $N\tau/\pi = \gamma$, without considering the injected random jitter. Since it is a lower bound of the exact mean and variance, it can be safely used for benchmarking.

4 Random walk

Our algorithm envisions the random walk of a token in a membership of N *end-users*. The membership can be recorded on the shared resource using plain CRUD operations, and other techniques have been designed for *peer to peer* systems (like [15]).

The basic parameter that defines the dynamics of the algorithm is the time interval between two successive token passing operations in the system: the duration of this lapse is τ, that here we call *hold time*. It does not correspond to the transfer delay of the token, though it is necessarily larger than this, and it is not related with the time taken to deliver the service. In our analysis, its value is significantly smaller than π, so that we are authorized to introduce the hypothesis that τ exactly divides π, i.e. there is an integer n such that $\tau * n = \pi$. This assumption allows us to use a discrete model for our discussion: we consider the time as split into a sequence of τ-wide intervals, called *frames*, with an index i is associated to each time frame.

There is no *global clock synchronization* assumption hidden behind this statement: time frames are separated by token switching events, not by absolute time values. Since such events are triggered by local clocks, which necessarily run at different speeds, the duration of each *frame* is affected by a jitter around τ.

Adhering to an Internet layer view, we consider that an undirected regular graph of degree $N - 1$ is used to connect the end-users. The results listed below can be extended to regular graphs with smaller degree (see [3]), like those found in *ad hoc* networks.

The *random walk* algorithm is described as follows:

```
while true do
        select(receiveToken, timeout(π))
        background(run(S))
        if holdToken() then
                                sleep(τ)
                                sendToken(random(Users))
        fi
od
```

Each client application starts the loop waiting for a token with a timeout of π time units. As soon as one of the two events occurs, the user issues the service request. Only in case the token is received, a delay of τ is triggered and, upon expiration, the token is forwarded to another peer selected at random. Otherwise the loop is immediately re-entered: in that latter case we label this request as *timed*. A consequence is that *timed* requests are always concurrent (according with our definition of concurrency) with the request issued by the client holding the token, and possibly with other timed requests.

Note that the measurement of the timeout π is affected by local clock inaccuracy: a *slow* clock will wait slightly longer before triggering a timed request. This fact has no impact on the algorithm, but will be reconsidered discussing its non-functional behavior.

We observe that the algorithm complies with timing requirements, since each end-user is served every at most π time units, and that it has a light footprint, since only one token exchange every τ time units is produced in the whole system. But the focus of our investigation is on the distribution of resource access events over time.

To this purpose, we need to compute the probability distribution of the *concurrency* process for the random walk algorithm, that we indicate with $\mathbf{c}_{rw}$: the task is complex since besides events related to token visits, which are approximated by an ordinary Poisson process, we need to take into account also *timed* events.

The formal proof is given in the appendix, where the transition graph is described exhaustively. Here we give the final result, summarized in the following theorem:

Theorem 1 *The discrete probability distribution of the concurrency of a random walk algorithm* $\mathbb{P}(\mathbf{c}_{rw} = i)$ *is the sum-one normalization of the eigenvector* $\mathbf{a}$ *of the infinite array* $\mathbf{T}$ *(i.e.,* $\mathbf{Ta} = \mathbf{1a}$*) where* $\mathbf{T} = T(k, k')(k, k' \in [1, \infty])$ *with*

$$\mathbf{T}(k,k') = \begin{cases} \binom{k}{k'-1} p^{k'-1}(1-p)^{k-(k'-1)} & \text{if } k' \in [1, k+1] \\ 0 & \text{otherwise} \end{cases}$$

and

$$p = e^{-\frac{\pi}{\mathbf{N}*\tau}} = e^{-\frac{1}{\gamma}}$$

Using the above theorem, it is possible to compute the approximated distribution of the concurrency process $\mathbf{c_{rw}}$ for a given γ: we need to introduce an approximation since the transition array $\mathbf{T}$ is infinite (although rapidly converging). Hence the

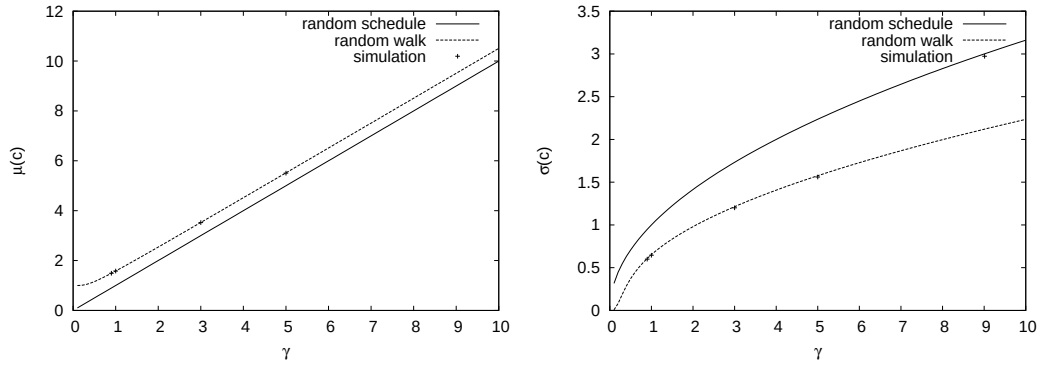


Figure 3. The comparison of $\mu(c)$ and $\sigma(c)$ for the random scheduling and random walk regulation algorithms

evaluation of the mean and of the standard deviation of the *concurrency* for the random walk algorithm, that are shown in figure 3. We have also validated the analytic model using a discrete event simulator of the protocol (approx. 100 lines in Perl): the results, that perfectly match, are shown in the same figure.

5 Discussion and comparison

In the previous section we have shown that a sequence of timed requests by a set of users can be reduced either to a Poisson process, when the regulator uses a *random scheduling* algorithm, or to a non-Markov process, using the *random walk* approach instead: now we want to compare the two algorithms to see how well the two flows of requests are managed by an **aaS provider*. In the previous section we have studied analytically the two processes, so that the relevant indicators for our discussion, the mean μ and the standard deviation σ, can be computed (see figure 3).

We observe that the interpretation of these results is not straightforward, since the comparison of the μ indicates that the *random schedule* solution exhibits a lower average concurrency, which is a plain advantage, while the σ is lower in the case of the *random walk* solution, which is an advantage when the workload has an elastic behavior, as explained in the introduction.

To discriminate the best solution we need to examine the amount of allocated capacity induced on the elastic resource by the two regulators, using the same server-side *buffering strategy*. To model the buffering strategy we do not describe its operational behavior, as we did in the example in the introduction, since the result would be specific for *that* policy. Instead, we prefer to describe analytically the final effect in a *black box* style.

We adopt a model that is based on the well known Chebyshev inequality (in the case of unimodal distributions, a.k.a. Vysochanskï-Petunin inequality): in a nutshell the inequality says that for a generic (unimodal) distribution, the probability of values larger than $\mu + 3\sigma$ is less than 5%.

Our reference *buffering strategy*, that we claim to be widely applicable since it aims at keeping a given quality of service, while avoiding frequent resizing, tends to allocate

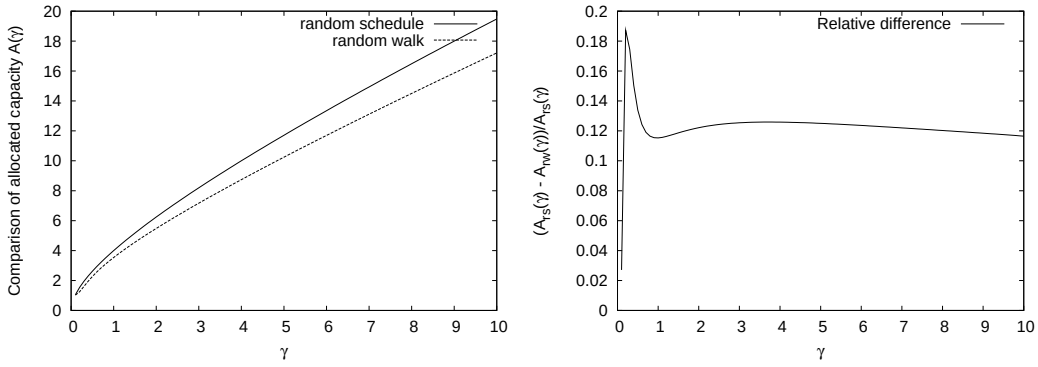


Figure 4. The comparison of allocated capacity over π/N for the two algorithms, and their relative difference ($(a-b)/a$)

a capacity of $\mu + 3\sigma$. Intuitively, the infrequent event of a peak that exceeds this capacity has some time to amortize the induced cost before the arrival of another such request, while a short low in service requests does not reflect in resource shrinking and incurs no cost. The constant 3 depends on the resizing operation, and increases with its cost. Here we stick with a value that is usually adopted in *rule of thumb* estimates.

So we use the value $\mu(\mathbf{c})+3\sigma(\mathbf{c})$ as an indicator of the amount of resources allocated on the elastic *aaS Provider to meet a service request flow modeled by process $\mathbf{c}$. Note that it is a linear combination of the two parameters that, taken alone, are unable to resolve a comparison between the two regulation algorithms, while their combination is a metric.

In figure 4, using this model for the buffering strategy, we compare the allocated capacity for the *random scheduling* and *random walk* regulators. The result is that the *random walk* algorithm exhibits a better performance, resulting in a lower allocated capacity, for values $\gamma > 0.2$. Below that threshold, that corresponds to the case where there are more than five frames in a period for each component request, the random walk algorithm performance rapidly degrades. Intuitively, this fact is justified observing that the *random scheduling* tends to put one event every $1/\gamma = 5$ frames, while the *random walk* will produce always one event for every frame.

We see that, when $\gamma > 0.2$, the difference of allocated capacity gradually increases with λ, so that the relative difference remains substantially around 12%. This result shows that the operation of the random walk algorithm is robust with respect to variations or inaccurate estimates of the λ within an order of magnitude.

5.1 Response to churns

A relevant case arises when the membership of *end-users* is subject to significant variations, also known as *churns*: in our reference environment, for instance, we need to take into account a long term variability of the number of end users, which entails a direct variation of γ. In that case the γ might fall in the "inefficient" zone below the 0.2 threshold when the number of users decreases significantly (i.e. more than

one order of magnitude).

The way to prevent the occurrence of such an event is to introduce a dynamic control of τ: in response to a relevant decrement of the number of *end users*, a corresponding increment of τ would keep the γ above the 0.2 threshold. From another perspective, this adjustment has the effect of slowing down the token. The tuning of the value of τ needs not to be extremely accurate, since the value of γ is not critical. One way to implement the tuning might be the maintenance of an approximate count (order of magnitude) of *end-users* in a public field hosted by the shared resource.

The case of a system that experiences a relevant increment of the number of end-users is not critical: the relative difference slopes down very slowly when $\gamma > 4$. However, when γ reaches the value 40 (not shown in figure) the relative difference of the allocated capacity degrades to 9%: also in this case, it is appropriate to tune the value of τ so that the value of γ falls in the range $[0.2, 10]$.

5.2 Multiple tokens: why not?

The option of introducing additional tokens is appealing, so its investigation is appropriate. The model for the general case of r tokens is only slightly different from the single token one, so we label it as a corollary:

Corollary 1 *The discrete probability distribution of the concurrency of a random walk regulator with r tokens $\mathbb{P}(c_{rw} = i)$ is the sum-one normalization of the eigenvector* $\mathbf{a}$ *of the infinite array* $\mathbf{T}$ *(i.e.,* $\mathbf{Ta} = \mathbf{1a}$*) where* $\mathbf{T} = T(k, k')(k, k' \in [1, \infty])$ *with*

$$T(k,k') = \begin{cases} \binom{k}{k'-r} p^{k'-r}(1-p)^{k-(k'-r)} & \text{if } k' \in [r, k+r] \\ 0 & \text{otherwise} \end{cases}$$

and

$$p = e^{-\frac{r*\pi}{\mathbf{N}*\tau}} = e^{-\frac{r}{\gamma}}$$

As shown in figure 5, the μ of concurrency is better for the single token case, while the σ is in favor of the multiple (two in figure) tokens. However, this time the overall balance is in favor of the single token.

One relevant conclusion is that the presence of *exactly one* token is not a stringent requirement: the presence by accident of an additional token does not degrade significantly the performance of the random walk regulator.

5.3 Timing issues

If we exclude the multi-token option, a token passing algorithm is self-clocking by nature, since the event that triggers its advancement is performed by a single host: the tick period is the *hold time* τ, whose accuracy affects the return time of the token.

The *hold time* is a stochastic variable that is made of two independent stochastic components: the internal measurement of the interval, and the token latency during

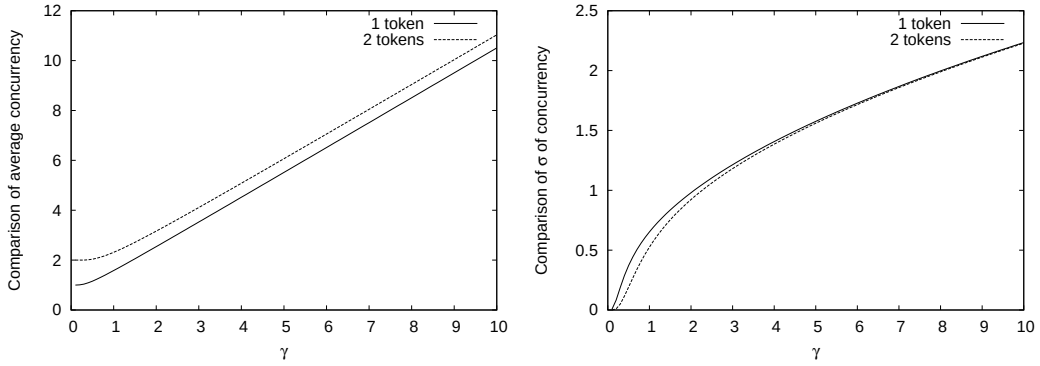


Figure 5. Comparison of average and standard deviation with one and two tokens

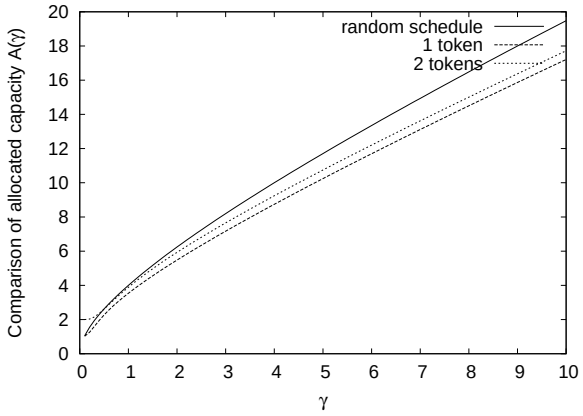


Figure 6. Comparison of allocated capacity with one and two tokens

the token passing operation. We know that clock accuracies ρ are usually better than one part over 10^5, while communication delay δ depends on the communication technology and may be heavy-tailed. We consider them separately.

Aside from possible interactions with the application, the presence of an accurate clock is a requirement introduced by the statement "*after π time units, only the units experiencing a timeout event may experience another such event, unless they are visited by the token in the meanwhile*", which is relevant for the consistency of the theorem (see its proof in the Appendix): if the units disagree in time measurement the timed run may be *displaced* from the expected frame. Let us analyze how this event occurs.

The lapse π is measured with accuracy ρ, and thus, for the error not to exceed the frame length τ, we need that $\pi\rho < \tau$, or, equivalently, $N < (\rho\gamma)^{-1}$. For instance, with a $\gamma = 2$ and a nominal ρ, we obtain $N < 50000$: if the number of end users is below that threshold only the displacement of timed runs on windows adjacent the expected ones may occur. The effects of these errors on the algorithm are marginal, and may be assimilated with a negligible jitter of γ.

In a large network with $N > (\rho\gamma)^{-1}$, we need to ensure that clocks are synchronized within $\tau/2$ time units. Since τ is necessarily larger than the communication delay, an NTP-like synchronization protocol is appropriate for the task. So we conclude that native clock accuracy is compatible with small networks, while larger networks need to deploy standard clock synchronization protocols.

Concerning the *heavy tail* of the communication delay δ, the token passing protocol (as well as any network dependent operation) is designed so that an error is generated whenever the δ exceeds a threshold δ_{max}: thus we reduce to a stochastic variable $\mathbf{TP}$ for the token passing operation delays with values in the interval $[0, \delta_{max}]$ and a probability of failure corresponding to $P(\delta > \delta_{max})$. Given the large number of *end-users*, the convolution of the token passing delays experienced during a closed loop has an average $n \cdot \mu(\mathbf{TP})$, with a standard deviation of $\sqrt{n} \cdot \sigma(\mathbf{TP}) < \sqrt{n} \cdot \delta_{max}$.

The value $\sqrt{n} \cdot \delta_{max}$ is an upper bound of the jitter of the hold time induced by communication delays: as a consequence the stochastic variable γ exhibits a corresponding jitter around the expected value $\gamma = N\tau/\pi$, with an upper bound given by $\frac{\sqrt{n}\delta_{max}}{\pi}$. With some algebra, we obtain that the relative jitter is $\frac{\sqrt{n}\delta_{max}}{N\tau} \in O(\frac{1}{\sqrt{N}})$. This proves that the jitter is always small compared with γ, and that it becomes negligible for large networks.

Summarizing, network delays and failures reflect in small variations of the γ, while time measurement inaccuracies hit only on large networks, and can be totally overridden using well established clock synchronization techniques.

6 Conclusions

We claim that an elastic resource responds to variations in the workload in a non-linear way: this is because resource reconfiguration is a resource consuming activity, and therefore the provider applies a conservative attitude, allocating redundant ca-

pacity.

This fact opens a new way to approach resource management: on the side of the client, the resource consuming activities are regulated to reduce variability while complying with application requests, so to limit the amount of redundant resources that are allocated on server side.

In this way, the two sides of the resource management activity, client-side and server-side, cooperate towards the optimization of resource utilization: such cooperation may be implicit (e.g., without a common design team) since client activity makes more predictable an event flow the server tries to predict. This is similar to what happens in "best effort" traffic management – think for instance to gateway based congestion control algorithms for networking [5] – but exhibits interesting facets when applied in a cloud environment.

We propose a client-side distributed coordination algorithm, that has the effect of regulating client requests. The algorithm is probabilistic in nature, being based on a random walk, but implements also timeout driven activities to comply with a real-time SLA. This latter detail, which reflects a common practice, makes difficult to find an analytic model, useful to investigate the algorithm: here we succeed in giving a simple model that exhaustively describes the dynamics of the algorithm.

Acknowledgments

I want to acknowledge the contribution of Prof. Francesco Romani (Dept. of Computer Science, Univ. of Pisa) that suggested me the use of eigenvectors in the numerical solution of the probability distribution of the concurrency, which is a cornerstone of this paper.

Appendix - Proof of the theorem

The random walk of the token is modeled as a sequence of visits to one of the N end-users, each visit with an extent of τ time units, the *hold time*. The timeline is thus split in a series of *frames*, each lasting τ time units. Given the connectivity of the graph, we conclude that the probability of visiting a given node during a given frame is $1/N$ (see [9]): considered that we are interested to system with a large N, the arrival of the token on a given end-user can be approximated by a Poisson process. Under this approximation, the *return time* $\mathbf{\Delta t}$ has the same distribution of the interarrival time of the token:

$$\mathbf{P}(\mathbf{\Delta t} < \pi) = 1 - e^{-\pi/s}$$

where the scale parameter $s = (N * \tau)$ is the expected *return time*.

This model describes access events related to token visits; the next step is to take into account also timed events, that are triggered when the return time exceeds π. The existence of such events makes the process non-Markov, since the new state of the system depends on the last π/τ states of the system: in fact, timed events are associated with *return times* of exactly π time units.

In our system the probability p_{tr} that a given user timeout triggers a *timed run* is:

$$p_{tr} = 1 - \mathbb{P}(\mathbf{\Delta t} < \pi) = e^{-\frac{\pi}{N*\tau}} = e^{-\frac{1}{\gamma}} \quad (1)$$

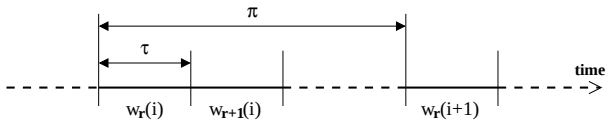


Figure 7. Window sequences

which therefore corresponds to the probability of a *timed run* during a certain time frame. The possibility of such event is restricted to the case that the same user was visited by the token exactly π/τ frames ago: if an event occurred on the same user prior to that time, a timeout would have been already triggered in a previous time frame.

To reduce to a Markov process we consider the sequences of windows $\mathbf{W_r}$ (with index $r \in [0, \frac{\pi}{\tau} - 1]$) , each other separated by the period π (see figure 7):

$$\begin{aligned} \mathbf{W_r} &= (w_r(0), w_r(1), ...w_r(n)) \\ & \quad where\ w_r(i) = [i * \pi + r * \tau, i * \pi + (r+1) * \tau] \end{aligned}$$

We have $\frac{\pi}{\tau}$ such sequences, and we consider each of them separately: as noted above, only a run in frame $w_r(i-1)$ may determine a *timed* run in frame $w_r(i)$; such event occurs with the probability p_{tr} in equation 1. So the process $\mathbf{c_r}(i)$ representing *concurrency* during the frames in $\mathbf{W_r}$ is a Markov process, since it has no memory of its past states, and the probability of each possible move to the next state in $\mathbf{W_r}$ is only determined by the current state.

In the transition graph (see figure 8), each node represents a system state and is labeled with a concurrency value: the system that experiences v timed events is in state $v+1$, since one further event is associated with the visit of the token. A timed event in window $w_r(i)$ can correspond only with one of the k ($k \geq v$) events in window $w_r(i-1)$ that was not followed, on the same node, by a visit of the token. Such event has a probability p_{tr} to occur, so we conclude that the probability of a transition from state k to state $v+1$ is:

$$T(k, v+1) = P(Binomial(k, p_{tr}) = v) = \binom{k}{v} p_{tr}^{v} (1 - p_{tr})^{k-v}$$

This means that the k-th row corresponds to the mass function of the binomial distribution of k trials with probability of success $p_{tr} = e^{-(1/\gamma)}$, or equivalently to the terms of the polynomial expansion of $(p_{tr} + (1 - p_{tr}))^k$. With the substitution $k' = v + 1$ we obtain

$$T(k, k') = \binom{k}{k'-1} p_{tr}^{k'-1} (1 - p_{tr})^{k-(k'-1)} \tag{2}$$

The probability distribution function of the concurrency during a window, is obtained as usual computing the eigenvector corresponding to eigenvalue 1 (normalized to have a sum of the elements of the eigenvector equal to 1). QED

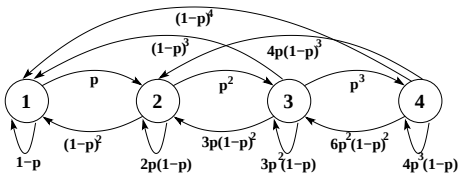
To extend to r tokens we consider that during each frame the tokens visits r *end users* simultaneously: this fact affects the probability of a timed run:

$$p_{tr}(r) = \mathbb{P}(\mathbf{\Delta t} \geq \pi) = e^{-\frac{r*\pi}{N*\tau}} = e^{-\frac{r}{\gamma}} \tag{3}$$

In addition, v timeouts now correspond to state $v + r$. The resulting transition array is (disregarding token collision):

$$T(k, v+r) = P(Binomial(k, p_{tr}(r)) = v) = \binom{k}{v} p_{tr}(r)^{v} (1 - p_{tr}(r))^{k-v}$$

with the replacement $k' = v + r$:



T	1	2	3	4	...
1	$1-p$	p	0	0	...
2	$(1-p)^2$	$2p(1-p)$	p^2	0	...
3	$(1-p)^3$	$3p(1-p)^2$	$3p^2(1-p)$	p^3	...
4	$(1-p)^4$	$4p(1-p)^3$	$6p^2(1-p)^2$	$4p^3(1-p)$	...
...					

Figure 8. Transition graph and array for the Markov process c_t (p stands for p_{tr})

$$T(k,k') = \binom{k}{k'-r} p_{tr}(r)^{k'-r}(1-p_{tr}(r))^{k-(k'-r)}$$

That can be processed as above to obtain, for a number of tokens $r << N$, the distribution of the concurrency. QED

References

[1] Amazon Web Services - auto scaling. http://aws.amazon.com/autoscaling/.

[2] Standard for a precision clock synchronization protocol for networked measurement and control systems. Technical Report IEEE Std 1588-2008, IEEE Instrumentation and Measurement Society, 3 Park Avenue, New York, NY 10016-5997, USA, July 2008.

[3] Ziv Bar-Yossef, Roy Friedman, and Gabriel Kliot. RaWMS - random walk based lightweight membership service for wireless *ad-hoc* networks. *ACM Transactions on Computer Systems*, 26(2):66, June 2008.

[4] Augusto Ciuffoletti. Secure token passing at application level. *Future Generation Computer Systems*, 27(7):1026–1031, July 2010.

[5] S. Floyd and V. Jacobson. Random early detection gateways for congestion avoidance. *IEEE/ACM Transactions on Networking*, 1(4):397–413, August 1993.

[6] Zhenhuan Gong, Xiaohui Gu, and Wilkes John. PRESS: PRedictive Elastic ReSource Scaling for cloud systems. In *International Conference on Network and Service Management*, 2010.

[7] W. Iqbal, M. Dailey, and D. Carrera. SLA-Driven Adaptive Resource Management for Web Applications on a Heterogeneous Compute Cloud. *Lecture Notes in Computer Science*, 5931:243–+, 2009.

[8] C. L. Liu and James W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM*, 20(1):46–61, January 1973.

[9] L. Lovasz. Random walks on graphs: a survey. In D. Miklos, V. T. Sos, and T. Szonyi, editors, *Combinatorics, Paul Erdos is Eigthy*, volume II. J. Bolyai Math. Society, 1993.

[10] Peter Mell and Timothy Grance. The NIST definition of cloud computing. Technical Report Special Publication 800-145, US Department of Commerce, October 2011.

[11] Anthony Nadalin, Chris Kaler, Ronald Monzillo, and Phillip Hallam-Baker. Web services security: Soap message security 1.1. http://docs.oasis-open.org/wss/v1.1/.

[12] Open Grid Forum. *Open Cloud Computing Interface - Core*, June 2011. Available from www.ogf.org.

[13] Open Grid Forum. *Open Cloud Computing Interface - Infrastructure*, June 2011. Available from www.ogf.org.

[14] Akshat Verma, Gautam Kumar, Ricardo Koller, and Aritra Sen. Cosmig: Modeling the impact of reconfiguration in a cloud. In *Modeling, Analysis, and Simulation of Computer Systems, International Symposium on*, volume 0, pages 3–11, Los Alamitos, CA, USA, 2011. IEEE Computer Society.

[15] Spyros Voulgaris, Daniela Gavidia, and Maarten Van Steen. CYCLON: Inexpensive membership management for unstructured p2p overlays. *Journal of Network and Systems Management*, 13:197217, 2005.

Biographical notes about the author

Augusto Ciuffoletti is a researcher at the University of Pisa, in Italy. His research interests are in distributed computing, with special attention paid to applications like Grid and Cloud computing. He had a long collaboration with the Italian Institute of Nuclear Physics (INFN) in the framework of International Projects for Grid monitoring and management, and he currently collaborates with the Open Cloud Computing Interface (OCCI) working group for the definition of a standard for cloud interfaces.