Joint Monitorless Load-Balancing and Autoscaling for Zero-Wait-Time in Data Centers

Yoann Desmouceaux, Marcel Enguehard, Thomas Heide Clausen

Joint Monitorless Load-Balancing and Autoscaling for Zero-Wait-Time in Data Centers

Yoann Desmouceaux, Marcel Enguehard, Thomas H. Clausen

***Abstract*—Cloud architectures achieve scaling through two main functions: (i) load-balancers, which dispatch queries among replicated virtualized application instances, and (ii) autoscalers, which automatically adjust the number of replicated instances to accommodate variations in load patterns. These functions are often provided through centralized load monitoring, incurring operational complexity. This paper introduces a unified and centralized-monitoring-free architecture achieving both autoscaling and load-balancing, reducing operational overhead while increasing response time performance. Application instances are virtually ordered in a chain, and new queries are forwarded along this chain until an instance, based on its local load, accepts the query. Autoscaling is triggered by the last application instance, which inspects its average load and infers if its chain is under- or over-provisioned. An analytical model of the system is derived, and proves that the proposed technique can achieve asymptotic zero-wait time with high (and controlable) probability. This result is confirmed by extensive simulations, which highlight close-to-ideal performance in terms of both response time and resource costs.**

***Index Terms*—Load balancing, auto-scaling, segment routing, application-aware, performance analysis.**

I. Introduction

Virtualization and cloud architectures, wherein different tenants share computing resources to run their workloads, have made fast task allocation and deallocation a commodity primitive in data centers [1]. To optimize costs while preserving *Quality of Service* (QoS), applications are thus (i) replicated among multiple instances running, *e.g.,* in containers or in virtual machines (VMs) [2], [3], and (ii) the number of aforementioned instances is automatically scaled up or down to meet a given Service Level Agreement (SLA) [4]. Two functions enable this: (i) a *load-balancer*, which dispatches queries onto identical replicas of the application, and (ii) an *autoscaler*, which monitors these instances and automatically adjusts their number according to the incoming load.

A challenge for network load-balancers is to provide performance and resiliency while satisfying per-application SLAs. Some architectures, such as Equal Cost Multi-Path (ECMP) [5] or Maglev [6], distribute flows among application instances pseudo-randomly, forwarding packets without terminating Layer-4 connections, and thus providing a high throughput. The use of consistent hashing also provides resiliency for when an existing flow is handed over to another load-balancer [6]. This requires, nonetheless, that flows be assigned to instances regardless of their load state, even though it has been demonstrated [7] that considering application load can greatly improve overall performance. Other load-balancing architectures do take application state into account, by terminating Layer-4 connections [8], and/or by using centralized monitoring [9] – thus incurring both a performance overhead and a degradation in resiliency.

Similarly, autoscalers use centralized monitoring, with an external agent gathering load metrics from all servers so as to make scaling decisions [10], [11]. The delay incurred by an external agent collecting these metrics causes such decisions to be made based on out-of-date information. Furthermore, such agents typically collect external metrics (*e.g.,* CPU load of a VM as seen by the hypervisor), ignoring application-specific metrics possibly more suitable for making scaling decisions.

A. Statement of Purpose

While workloads lasting hours or minutes (*e.g.,* data processing tasks) can be efficiently scheduled with offline optimization algorithms [12], and while sub-millisecond workloads require over-provisioning as the time to commission a new instance is too large as compared to the application execution time, mid-sized workloads (lasting from $100\,\mathrm{ms}$ to $1\,\mathrm{s}$, *e.g.,* Web workloads) are amenable to reactive autoscaling, as container boot times are typically sub-second [13]. Thus, in this paper, the problem of mid-sized workloads scalability under QoS constraints is explored, for replicated applications deployed, *e.g.,* as containers. In particular, a *centralized-monitoring-free* architecture for achieving *asymptotic zero-wait-time* is introduced. More precisely, the architecture is centralized-monitoring-free as it relies on the application themselves monitoring their load, without piggy-backing information to a central controller. It yields asymptotic zero-wait-time in the sense that each incoming query finds, with probability converging to one as the number of application instances goes to infinity, an idle application instance. The architecture relies on two interdependent components: a *load-aware* load-balancing algorithm and a *decentralized* autoscaling policy.

First, a centralizerd-monitoring-free load-balacing algorithm is introduced: *Join-the-First-Idle-Queue* (JFIQ). JFIQ relies on ordering the available application instances in a chain along which incoming queries are directed. Each of the instances in the chain makes a local decision based on its load, accepting the query if it has available capacity, and forwarding the query to the next instance in the chain otherwise. The proposed architecture operates entirely within the network layer (Layer-3) using IPv6 Segment Routing (SRv6) [14], thus removing

Y. Desmouceaux is with Cisco Systems, 92130 Issy-les-Moulineaux, France. M. Enguehard is with Polyconseil, 75008 Paris, France. T. H. Clausen is with École Polytechnique, 91120 Palaiseau, France. Emails: ydesmouc@cisco.com, marcel@enguehard.org, thomas.clausen@polytechnique.edu

the need from terminating or proxying network connections.

Second, to achieve asymptotic zero-wait-time, JFIQ is complemented with a centralizerd-monitoring-free autoscaling policy which uses the fact that the busyness of the last instance in the chain is an indicator of the busyness of the whole system. This allows offloading autoscaling decisions to that last instance, by measuring its occupancy ratio over time. Upscaling/downscaling is triggered if that ratio crosses pre-determined maximum/minimum thresholds. An analytical model demonstrates the validity of using this autoscaling policy conjointly with JFIQ to achieve asymptotic zero-wait-time, and quantifies the behavior of the system in terms of response time.

Finally, this analytical model is complemented with extensive simulations, capturing the dynamics of the architecture, and showing that the proposed mechanism allows to precisely control the tail of the response time distribution. These simulations illustrate that the propose mechanisms reduce the resource cost (i.e., the number of necessary instances) for an identical target response time by an order of magnitude in the evaluated scenario, when compared to the simpler policies used in consistent-hashing-based load-balancers.

B. Related work

This section discusses the literature on network load-balancing (section I-B1) and autoscaling (section I-B2).

1) Load-balancing: The goal of a load-balancer is to assign incoming queries for a given service to one of several distributed instances of this service. As such, this requires: (i) selecting the instance so as to minimize response time, and (ii) making sure that the load-balancer does not become a bottleneck.

Several load-aware load-balancing algorithms exist [15], including Random (RND), where queries are assigned randomly to one of n application instances, and Round-Robin (RR), where the i-th query is assigned to the $(i \mod n)$-th instance. The optimal policy is the Least-Work-Left (LWL) policy, which assigns queries to the application instance with the least amount of pending work [16]. A simpler algorithm is Join-the-Shortest-Queue (JSQ), which assigns queries to the least loaded of the application instances. JSQ does not require knowledge of the remaining work time of currently-served queries, and provides near-optimal performance [17], even in high-load regimes [18]. JSQ needs to query the state of all application instances for each incoming query, which incurs a monitoring overhead of n messages per query. A more scalable algorithm, Join-the-Idle-Queue (JIQ), has been proposed in [19]: queries are assigned to an idle application instance if one exists, or to a random instance otherwise. This is implemented by maintaining a centralized *idle queue* of the identities of currently idle application instances, minimizing the monitoring overhead as compared to JSQ. Another algorithm is Join-the-Shortest-of-d-Queues (JSQ_d) [7], which assigns queries to the least loaded of d randomly sampled application instances, and which is therefore more decentralized but less efficient than JIQ (as stated in [20]). The algorithms algorithm listed above have been analyzed in the *heavy-traffic limit* (where the query rate approaches stability), allowing to quantify the achieved expected waiting time as a function of the number of application instances [20], [21].

The above has summarized a set of algorithms for assigning flows to applications, as well as their key performance characteristics. It is equally important to be able to actually distribute network flows across application instances, at the network layer. This consists of directing flows (*e.g.,* TCP packets) corresponding to queries for a given service (described by a *virtual IP address*, VIP) to the *physical IP address* (PIP) of a deployed instance. This load-balancing function can itself be replicated, in which case it is deployed behind a layer of ECMP routers, which can arbitrarily redistribute packets between load-balancer instances, for new flows as well as for already-established flows. It is thus necessary to maintain Per-Connection-Consistency (PCC), *i.e.,* to ensure that already-established flows are always directed to the same application instance, regardless of the load-balancer they are handled by. Maglev [6] and Ananta [22] use a combination of consistent hashing and per-flow tables to ensure PCC. This has been complemented by enabling hardware-support [23], [24], [25], or by using in-packet state to maintain PCC [25], [26], [27]. While providing per-connection consistency, these architectures do not consider the application instance load, using a naïve RND policy at the cost of decreased application-performance [15], [28]. A first step towards considering the load of application instances is 6LB [29], where consistent hashing is used with a variant of the JSQ_2 algorithm that assigns queries to the first available from among two candidate instances. Some architectures [9] rely on Software-Defined Networking (SDN) to monitor the network and the servers, and thus make load-aware decisions – but at the cost of a monitoring overhead.

2) Autoscaling: Methods to provide autoscaling have been classified as reactive and proactive [4]. Reactive methods regularly gather measurements, and take actions when thresholds are crossed. For instance, in [10] up/downscaling is triggered when bounds on some observed metrics are reached; a similar approach can be found in [11], but with dynamic threshold adjustment. These incur an overhead from gathering statistics, and a time gap between detection of violations and appropriate reaction. Similar threshold-based approaches include [30], [31], [32].

Conversely, proactive approaches consist of anticipating state and acting correspondingly. For example, in [33], moving averages are used to anticipate the future value of metrics of interests. Similarly, [34] uses Machine Learning (ML) to classify workloads by their resource allocation preferences, and in [35], neural networks are used to predict CPU load trends of application instances and provision resources accordingly. A Tree-Augmented Naive Bayesian network is used in [36] to detect SLA violations, and scale resources up when this happens. In [37], [38], control theory is used to track CPU usage and to allocate resources accordingly, and in [39], control theory is used to adapt the amount of CPU resources allocated to each query so that they complete within a deadline. While solving the issue of timeliness, proactive approaches suffer the need to collect statistics and perform centralized computations.

Using queuing theory has also been proposed [40], [41]. In [42], an autoscaling scheme for JIQ is proposed, by creating a feedback loop that decommissions application instances that remain idle for a long period of time, and commissions a new application instance for each new query. In [43], a similar token-based mechanism is introduced, with a new application instance being commissioned only when a task only finds busy instances.

C. Paper Outline

The remainder of this paper is organized as follows. Section II gives an overview of the architecture introduced in this paper. An analytical model for the response time of the system with a fixed number of instances is introduced in section III, and the asymptotic behavior of the system if characterized. Numerical results are given in section IV, along with computational simulations providing further insight. Finally, section V concludes this paper.

II. Joint Load-Balancing and Autoscaling

In this paper, an application is replicated on a set of n application instances $\{s_1, \ldots, s_n\}$ with identical processing capacities. The goal is to minimize response time, *i.e.,* queries should be served with *zero waiting time*, by way of (i) ensuring that enough application instances are available, and (ii) mapping the query to an idle application instance. To address the challenges introduced in sections I-B1 and I-B2, this goal is attained through *joint* load-balancing and autoscaling strategies which provide not only close-to-ideal algorithmic performance, but which can also be efficiently implemented, *i.e.,* both the load-balancing and autoscaling functions must incur minimal state and network overhead. The proposed architecture relies on three intertwined building blocks: (i) a load-balancing algorithm that achieves asymptotic zero-wait-time if the number of application instances is correctly scaled; (ii) an enhanced IPv6 dataplane to perform query dispatching in a decentralized and stateless fashion; (iii) a centralized-monitoring-free autoscaling technique to adapt the number of application instances while incurring no monitoring cost.

A. Join-the-First-Idle-Queue Load-Balancing

An ideal load-balancing algorithm should achieve asymptotic zero-wait time for a properly-scaled set of application instances. In particular, this is the behaviour of the reference JIQ policy, which keeps track of available instances by means of a centralized *idle queue*, with instances communicating their availability to a centralized controller upon completion of a query. The drawbacks of JIQ are twofold: it requires centralized communication (which can create implementability and scalability issues), and it requires centralized load monitoring if used in conjunction with an autoscaler. To address these issues, this paper proposes a new load-balancing technique: *Join-the-First-Idle-Queue* (JFIQ), which does not rely on centralized load tracking.

JFIQ relies on ordering the n application instances in a chain (since the application instances are assumed to have identical capacity, the actual order of the instances in the chain does not matter, so long as it remains consistent throughout the lifetime of the system). Then, JFIQ enforces that each of the first $(n-1)$ instances never serves more than 1 query at a given time (see figure 1). Formally, each query is forwarded along the chain $(s_1, \ldots, s_n)$ of n application instances. Each instance $s_i \neq s_n$ in the list either accepts the query if it currently idle, and otherwise forwards it to the next instance s_{i+1}. To ensure that all queries are served, the last instance s_n must always accept queries. Thus, each of the first $(n-1)$ instances can hold only 0 or 1 query, ensuring *zero waiting time* for queries served by those. As shown later in section III-B, JFIQ allows to predictably control the probability of having a blocked task (*i.e.,* a task waiting for the last application instance to become idle) by varying the number n of instances.

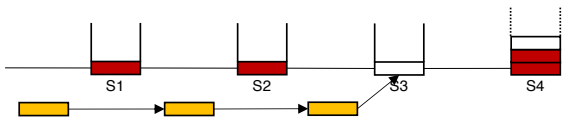


Figure 1. Join-the-First-Idle-Queue LB (Algorithm 1) with $n = 4$ instances

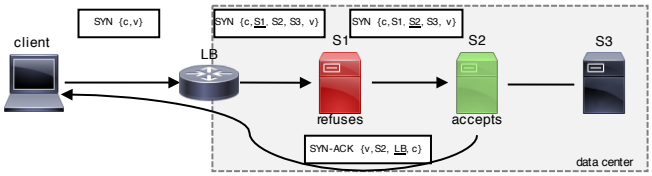


Figure 2. Example of *SR load-balancing* [29] with 3 instances, wherein the second one accepts the connection.

B. Network-level JFIQ using SRv6

To achieve JFIQ at the network layer while enabling application-awareness, this paper leverages the dataplane of 6LB [29] and SHELL [44], summarized in figure 2. This dataplane is based on SRv6, a source-routing architecture which allows specifying, within a specific IPv6 Extension Header [45], a list of *segments* to be traversed by a given packet, where each segment is an IPv6 address representing an instruction to be performed on the packet.

First, a control plane provisions the egress router with a fixed list of application instances to be used by the JFIQ algorithm. Then, when a connection establishment packet (*e.g.,* a TCP SYN) destined for the VIP is received by the egress router, it inserts an SRv6 header, with a list of PIPs corresponding to that list of instances. Instances then implement the JFIQ algorithm as described in algorithm 1, by either handling the packet locally or forwarding it to the next instance. To avoid perpetual triangular traffic, a "stickiness" mechanism is then used to let subsequent packets within this flow be directed to the instance having accepted the connection [44]. A specific field of the transport header is used as a covert channel to encode the index of the application instance that has accepted the connection – examples of such fields include QUIC session ID, low-order bits of TCP timestamps, or high-order bits of TCP sequence numbers. This field must be able

Algorithm 1 Local Connection Request Handling

$p \leftarrow$ connection establishment packet ▷ *e.g., TCP SYN*
$v \leftarrow p.\texttt{lastSegment}$ ▷ *VIP*
$b \leftarrow$ number of busy threads for v
if $b = 0$ **then** ▷ *application instance is available*
  $p.\texttt{segmentsLeft} \leftarrow 0$
  $p.\texttt{dst} \leftarrow v$
  forward p to local workload v
else ▷ *forward to next application instance*
  $p.\texttt{segmentsLeft} \leftarrow p.\texttt{segmentsLeft} - 1$
  $p.\texttt{dst} \leftarrow p.\texttt{nextSegment}$
  transfer p to $p.\texttt{dst}$
end if

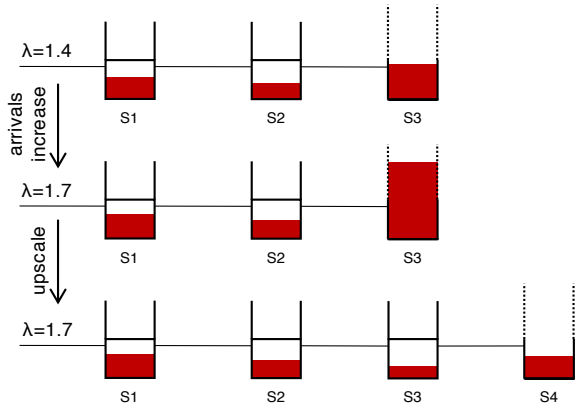


Figure 3. Autoscaling when $n = 3$ and $\mu = 1$. The level of red in each application instance shows the average number of concurrently-served queries as computed in section III. When the query rate increases from $\lambda = 1.4$ to $\lambda = 1.7$, the third instance observes that it has become highly occupied and thus requests upscaling.

to be set by the application instance and transparently echoed in packets sent by the client, thus allowing the ingress router to statelessly determine to which application instance non-SYN packets should be forwarded.

Therefore, the load-balancing function does not require per-flow state, consisting of (i) applying a fixed SR list on connection establishment packets, or (ii) applying a one-segment list on other packets, with a destination address that depends on the value encoded in the covert channel found in the packet. This makes the load-balancing function simpler, thus more amenable to low-latency, high-throughput hardware implementations. Plus, as the functionality performed by the ingress router does not require any synchronization, it can be distributed among several routers, yielding scalability and flexibility.

C. Autoscaling

A key feature of JFIQ (compared, *e.g.,* to JIQ) is that the last instance has a unique view on whether the system is overloaded or not. By construction, all instances but the last only accept queries when idle. This can be exploited to perform autoscaling: when the last instance detects that it serves too many or too few queries, it requests to the control plane that the chain be scaled up or down. The control plane then provisions or deprovisions an instance as needed, and updates the ingress router with the new list of instances to be used

Algorithm 2 Local Autoscaling at Last Application Instance

$p_e^{\uparrow}, p_e^{\downarrow} \leftarrow$ **parameter** ▷ *up/downscaling thresholds*
$r_{avg} \leftarrow$ **parameter** ▷ *average application execution time* $(1/\mu)$
$W \leftarrow 1000 \times r_{avg}$ ▷ *window size for EWMA*
$t_0 \leftarrow$ time() ▷ *timestamp of last event for EWMA*
$\widehat{p_e} \leftarrow 0$ ▷ *EWMA sample of* $p_e = \mathbf{P}[N_n = 0]$
$r \leftarrow 0$ ▷ *number of events*
for each connection establishment packet p from client **do**
  $r \leftarrow r + 1$
  $N_n \leftarrow$ number of busy threads for v
  $\alpha \leftarrow 1 - \exp(-(\text{time}() - t_0)/W)$
  $\widehat{p_e} \leftarrow (1-\alpha)\widehat{p_e} + \alpha \mathbf{1}_{\{N_n=0\}}$
  $t_0 \leftarrow$ time()
  if $r > 50$ **then** ▷ *make sure to have a significant sample*
    if $\widehat{p_e} > p_e^{\downarrow}$ **then**
      request downscaling; reset all variables
    else if $\widehat{p_e} < p_e^{\uparrow}$ **then**
      request upscaling; reset all variables
    end if
  end if
  $p.\texttt{segmentsLeft} \leftarrow 0$
  $p.\texttt{dst} \leftarrow v$
  forward p to local workload v
end for
for each connection termination packet p from application **do**
  $r \leftarrow r + 1$
  $\alpha \leftarrow 1 - \exp(-(\text{time}() - t_0)/W)$
  $\widehat{p_e} \leftarrow (1-\alpha)\widehat{p_e}$ ▷ N_n *was* > 0 *over the last period*
  $t_0 \leftarrow$ time()
  forward p
end for

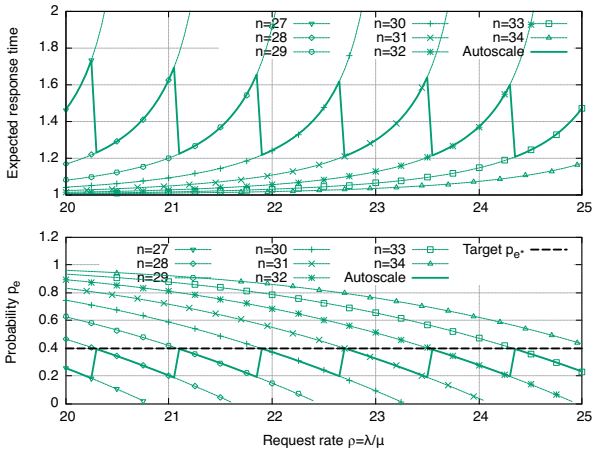


Figure 4. JFIQ autoscaling: example of *up*scaling for $p_e^* = 0.4$ and $\rho \in (20, 25)$: the number n of instances adapts to maintain p_e within p_e^* (thick line) and $p_e^{\uparrow}$. The top graph depicts the corresponding expected response time $\mathbf{E}[T]$, numerically computed with the method introduced in section III.

by the load-balancing function. This allows for centralized-monitoring-free autoscaling, as illustrated in figure 3.

As formalized in Algorithm 2, the last instance in the chain keeps statistics about its queue size over time. The fraction of time p_e during which the last instance is empty is sampled (with an Exponentially-Weighted Moving Average, EWMA) and the autoscaling mechanism tries to maintain it close to a fixed, tunable, target p_e^*. When p_e goes below a threshold $p_e^{\uparrow}$, the instance triggers upscaling of the chain. Conversely, when this goes above a threshold $p_e^{\downarrow}$, the instance triggers downscaling of the chain. To avoid oscillations, the proposed autoscaling method ensure that p_e^{n-1}, the fraction of time

Table I
NOTATION USED WHEN APPLYING THE RRR TECHNIQUE

Notation	Description
$n \geq 2$	Number of instances
$\bar{n} = n - 1$	All instances but the last
$\lambda > 0$	Query rate
$\mu > 0$	Service rate
$\rho = \lambda/\mu$	Normalized query rate
N_i	Number of queries handled by the i-th instance
$\Sigma N = \sum_{i=1}^{\bar{n}} N_i$	Number of queries handled by the first $\bar{n}$ instances
$\pi_k = \mathbf{P}[\Sigma N = k]$	Probability distribution of ΣN
$p_e = \mathbf{P}[N_n = 0]$	Probability that last instance is idle
p_e^*	Target idleness value p_e for autoscaling
State (i, j)	$\Sigma N = i$ and $N_n = j$, *i.e.,* i out of the first $\bar{n}$ instances are busy and the last instance handles j queries
$p^D_{(i,j)\to(k,j-1)}$	Probability that first state visited on row $j-1$ when starting from (i, j) is $(k, j-1)$
$p^D_{i\to k}$	Probability that first state visited on row 0 when starting from $(i, 1)$ is $(k, 0)$
$R^D_{(i,j)\to(\star,j-1)}$	Mean reward of $z^{N_n(t)}$ when going from (i, j) down to row $j-1$
R^D_i	Mean reward of $z^{N_n(t)}$ when going from $(i, 1)$ down to row 0
R^L_i	Mean reward of $z^{N_n(t)}$ when going from $(i, 0)$ left to $(0, 0)$
R^C	Mean reward of $z^{N_n(t)}$ over a cycle from $(0, 0)$ to $(0, 0)$

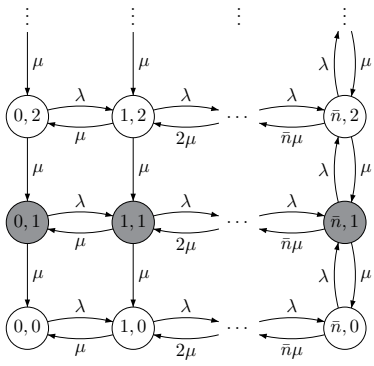


Figure 5. Markov chain for the JFIQ load-balancing scheme in the case of n instances (with $\bar{n} = n-1$ for readability). State (i, j) means i of the first $\bar{n}$ instances are serving one query, and the n-th instance is serving j queries. Space state is $\{0, \ldots, \bar{n}\} \times \mathbb{N}$. The border row $\{0, \ldots, \bar{n}\} \times \{1\}$ is grayed.

during which the $(n-1)$-st instance is empty just after a downscaling operation, is above $p_e^{\uparrow}$, so as not to re-trigger upscaling – and similarly when upscaling. How to tune p_e^*, and accordingly set $p_e^{\uparrow}$ and $p_e^{\downarrow}$, is discussed in section III. Figure 4 illustrates the behavior of the system when subject to an increasing load and when $p_e^* = 0.4$, by depicting the evolution of p_e (using the numerical computation that will be detailed in section III).

III. JOIN-THE-FIRST-IDLE-QUEUE ANALYSIS

The *Join-the-first-idle-queue* load-balancing scheme introduced in section II-A is analytically studied in this section, for a fixed number n of instances. For readability, $\bar{n} = n-1$ will denote the number of instances that can reject a query. Table I summarizes the notation used in this paper. Proofs are provided in the appendix.

A. Markov Model

For the purpose of this analysis, arrivals are assumed to follow to a Poisson process of intensity $\lambda > 0$ (*i.e.,* the probability that the time between two arrivals is less than t is $1-e^{-\lambda t}$). Each application instance has an identical processing capacity $\mu > 0$, with exponentially-distributed service times (*i.e.,* the probability of a query completing in less than t is $1 - e^{-\mu t}$). To ease notation, $\rho = \lambda/\mu$ denotes the normalized query rate. As described in section II-A, a query is directed to the chain of application instances $(s_1, \ldots, s_n)$, where each instance s_i serves it if idle and otherwise forwards it to instance s_{i+1}, until reaching the last instance.

Therefore, the whole system can be modeled as a Markov chain. With x_i denoting the number of queries handled by the i-th instance s_i, the state of the system is $\mathbf{x} = (x_1, \ldots, x_n)$, and the state space is $\mathcal{S} = \{0, 1\}^{n-1} \times \mathbb{N}$. In each state, the system can transition *upwards* with rate λ to the state in which the first available instance serves *gained* one query, and *downwards* with rate μ to any of those states where one query has *completed*[1].

To simplify the analysis, a Markov chain with a reduced state space is used, which still captures the metrics of interest. The number of queries handled by the first $\bar{n}$ instances is denoted by i (with $0 \leq i \leq \bar{n}$), and the number of queries handled by the last instance by j. Formally, $i = \sum_{j=1}^{n-1} x_i$, and $j = x_n$. Then, the behavior of (i, j) can be modeled as a Markov chain with state space $\mathcal{S} = \{0, \ldots, \bar{n}\} \times \mathbb{N}$. Transitions that increase j can only happen when $i = \bar{n}$ (*i.e.,* queries can be accepted by the last instance only if all but the last are busy). Transitions that increase i can happen from any state with $i < \bar{n}$ (*i.e.,* as long as there is an idle instance among the first $\bar{n}$, queries will be directed to one such instance). In both cases, these transitions occur at rate λ, the overall query rate. Transitions that decrease i happen at rate $i\mu$ (because the processing capacity of the pool of the first $\bar{n}$ instances is μ times the number of busy instances), and those decreasing j happen at rate μ (because the processing capacity of the last instances is μ). Figure 5 shows a graphical representation of this Markov chain.

B. Applying the Recursive Renewal Reward (RRR) Method to the Model

To obtain the client response time, according to Little's law [46], the expected number of queries handled by the system must be derived. In the following, N_i will denote the number of queries handled by the i-th application instance. The total number of queries handled by the first $\bar{n}$ instances, $\Sigma N := \sum_{i=1}^{\bar{n}} N_i$, follows the law of an M/M/$\bar{n}$/$\bar{n}$ queue. The probability distribution π_k of ΣN is therefore:

[1]Formally, let $\mathbf{e_i}$ be the vectors of the canonical basis of $\mathbb{R}^n$: $\mathbf{e_i} = (\delta_{i1}, \ldots, \delta_{in})$. For each state $\mathbf{x}$, let the *upward direction* of $\mathbf{x}$ be $u(\mathbf{x}) = \mathbf{x} + \mathbf{e}_{\min\{i \in \{1,\ldots,n\} | x_i = 0\}}$, with the convention $\min \emptyset = n$. A set of *downward directions* is also defined as $d(\mathbf{x}) = \bigcup_{i:x_i>0} \{\mathbf{x} - \mathbf{e_i}\}$. With this notation, from each state $\mathbf{x}$ there is a transition to $u(\mathbf{x})$ with rate λ, and a transition to each state in $d(\mathbf{x})$ with rate μ.

$$\mathbf{P}[\Sigma N = k] = \frac{\rho^k}{k! \sum_{i=0}^{\bar{n}} \rho^i/i!} := \pi_k \tag{1}$$

and the expected number of queries handled by the first $\bar{n}$ instances is:

$$\mathbf{E}[\Sigma N] = \frac{\sum_{k=0}^{\bar{n}} k\rho^k/k!}{\sum_{k=0}^{\bar{n}} \rho^k/k!} \tag{2}$$

To obtain the probability distribution of N_n, the number of queries handled by the last instance, the RRR [47] method is applied. RRR consists of identifying a *home state* S_0 and a *metric of interest* $M(t)$ whose expected value shall be computed. Then, the expected value of M can be evaluated by computing its average earning rate over a cycle from S_0 to itself, divided by the average duration of the cycle:

$$\mathbf{E}[M] = \frac{\mathbf{E}[\int_{S_0 \to S_0} M(t)\mathrm{d}t]}{\mathbf{E}[\int_{S_0 \to S_0} 1\mathrm{d}t]} \tag{3}$$

RRR applies to bi-dimensional Markov chains with a one-dimensional repeating pattern (as in figure 5), in which case computing the average earning rate of the reward reduces to solving a finite number of equations, by restricting to the *border row* that generates the pattern.

The probability distribution of N_n is computed by deriving its probability generating function $f(z) = \mathbf{E}[z^{N_n}]$ ($z \in \mathbb{C}$). To that purpose, the *metric of interest* is chosen to be $M(t) = z^{N_n(t)}$, such that applying RRR will yield $\mathbf{E}[z^{N_n}]$. The *home state* is chosen as $(0,0)$. To compute the mean reward of $M(t)$ over a cycle from the home state to itself, the reward is decomposed into two parts: (i) the mean reward when going one level down in the Markov chain (*i.e.,* until a query served from in the last instance terminates), and (ii) the mean reward when going from a state $(i,0)$ to the home state (*i.e.,* until all queries have completed).

1) Reward when going one level down: For $j \geq 1$, let $R^D_{(i,j)\to(\star,j-1)}$ be the mean reward earned between entering state (i,j) and going "one level down" by reaching row $j-1$ (*i.e.,* reaching any state $(k,j-1)$). Due to the repeating structure of the chain, lemma 1 holds:

Lemma 1 *For $i \in \{0,\dots,\bar{n}\}$ and $j \geq 1$, $R^D_{(i,j+1)\to(\star,j)} = zR^D_{(i,j)\to(\star,j-1)}$. Therefore, only $R^D_{(i,1)\to(\star,0)}$ needs to be computed, which will be denoted by R^D_i.*

Let $p^D_{(i,j)\to(k,j-1)}$ be the probability that, starting from state (i,j), the first state visited when reaching row $j-1$ is $(k,j-1)$. Similarly to lemma 1, since the chain has a recursive structure along the vertical dimension, lemma 2 holds:

Lemma 2 *For $j \geq 1$ and $k \in \{0,\dots,\bar{n}\}$, $p^D_{(i,j)\to(k,j-1)}$ is independent from j, and will therefore be denoted by $p^D_{i\to k}$.*

With these two lemmas, it is possible to formulate a system of equations[2] for the $p^D_{i\to k}$, for $k \in \{0,\dots,\bar{n}\}$:

$$p^D_{0\to k} = \frac{\mu}{\lambda+\mu}\delta_{0k} + \frac{\lambda}{\lambda+\mu}p^D_{1\to k} \tag{4}$$

[2] δ_{nm} denotes the Kronecker symbol: $\delta_{nm} = 1$ if $n = m$ and 0 otherwise.

$$p^D_{i\to k} = \frac{\mu}{\lambda+(i+1)\mu}\delta_{ik} + \frac{i\mu}{\lambda+(i+1)\mu}p^D_{i-1\to k} + \frac{\lambda}{\lambda+(i+1)\mu}p^D_{i+1\to k}, \forall 1 \leq i \leq \bar{n}-1 \tag{5}$$

$$p^D_{\bar{n}\to k} = \frac{\mu}{\lambda+n\mu}\delta_{\bar{n},k} + \frac{\bar{n}\mu}{\lambda+n\mu}p^D_{\bar{n}-1\to k} + \frac{\lambda}{\lambda+n\mu}\sum_{\ell=0}^{\bar{n}} p^D_{\bar{n}\to\ell}\, p^D_{\ell\to k} \tag{6}$$

In these equations, the leftmost terms (of the right-hand side) denote the probability of reaching $(k,0)$ from $(i,1)$ directly, while the following terms express reaching $(k,0)$ from $(i,1)$ by transitioning to an adjacent state first. The rightmost in equation (6) comes from the possibility of reaching $(k,0)$ from $(i,1)$ by going to row 2 first, and uses lemma 2 to compute the transition from row 2 back to row 0.

Having computed the $p^D_{i\to k}$, it is possible to state a system of equations for R^D_i:

$$R^D_0 = \frac{z}{\lambda+\mu} + \frac{\lambda}{\lambda+\mu}R^D_1 \tag{7}$$

$$R^D_i = \frac{z}{\lambda+(i+1)\mu} + \frac{i\mu}{\lambda+(i+1)\mu}R^D_{i-1} + \frac{\lambda}{\lambda+(i+1)\mu}R^D_{i+1} \quad \forall 1 \leq i \leq \bar{n}-1 \tag{8}$$

$$R^D_{\bar{n}} = \frac{z}{\lambda+n\mu} + \frac{\bar{n}\mu}{\lambda+n\mu}R^D_{\bar{n}-1} + \frac{\lambda}{\lambda+n\mu}\left(zR^D_{\bar{n}} + \sum_{k=0}^{\bar{n}} p^D_{\bar{n}\to k}\, R^D_k\right) \tag{9}$$

The first terms of the right-hand side of these equations denote the mean reward earned when in state $(i,1)$, (*i.e.,* z times the expected time spent in $(i,1)$, since in these states $N_n = 1$ and $z^{N_n} = z$). The following terms come from the reward earned if going to an adjacent state instead of directly going to row 0. The bracketed term in equation (9) uses lemma 1 to express the reward when going from $(\bar{n},2)$ to row 1, and then lemma 2 for going from a non-deterministic state in row 1 to row 0. Solving the system of equations (7),(8),(9) yields the following for the values of R^D_i:

Lemma 3 *For $i \in \{0,\dots,\bar{n}\}$, $R^D_i = \frac{A_i z + B_i z^2}{C - Dz}$, where A_i, B_i, C, D are real constants, and $B_{\bar{n}} = 0$.*

2) Reward when going from row 0 to the home state: Having computed R^D_i, it is possible to compute R^L_i, the mean reward earned when going from a state $(i,0)$ (with $i \geq 1$) left to the home state $(0,0)$ – with $R^L_0 = 0$, since reaching $(0,0)$ means the end of the cycle. The system of equations for the R^L_i is:

$$R^L_0 = 0 \tag{10}$$

$$R^L_i = \frac{1}{\lambda+i\mu} + \frac{i\mu}{\lambda+i\mu}R^L_{i-1} + \frac{\lambda}{\lambda+i\mu}R^L_{i+1}, \forall 1 \leq i \leq \bar{n}-1 \tag{11}$$

$$R^L_{\bar{n}} = \frac{1}{\lambda+\bar{n}\mu} + \frac{\bar{n}\mu}{\lambda+\bar{n}\mu}R^L_{\bar{n}-1} + \frac{\lambda}{\lambda+\bar{n}\mu}\left(R^D_{\bar{n}} + \sum_{j=0}^{\bar{n}} p^D_{\bar{n}\to j}\, R^L_j\right) \tag{12}$$

This system is obtained similarly to the system (7),(8),(9). The bracketed term in equation (12) uses lemma 1 and expresses the possibility to go from $(\bar{n}, 0)$ to $(\bar{n}, 1)$, back to one of the possible states in row 0, and finally to $(0, 0)$. Solving the system of equations (11),(12) yields the following:

Lemma 4 *For $i \in \{1, \dots, \bar{n}\}$, $R_i^L = \frac{E_i + F_i z}{C - Dz}$, where E_i, F_i are real constants, and C, D the same as in lemma 3.*

3) Mean reward over a cycle: Finally, it is possible to express R_C, the mean reward over a cycle from the home state to itself, as follows:

$$R^C = \frac{1}{\lambda} + R_1^L \tag{13}$$

This allows deriving the total expected value of z^{N_n}, using equation (3):

$$\mathbf{E}[z^{N_n}] = \frac{\mathbf{E}[\int_{(0,0)\to(0,0)} z^{N_n(t)} \mathrm{d}t]}{\mathbf{E}[\int_{(0,0)\to(0,0)} 1 \mathrm{d}t]} = \frac{R^C(z)}{R^C(1)} \tag{14}$$

Applying lemma 4 to $i = 1$ then yields the following for $\mathbf{E}[z^{N_n}]$:

Theorem 1 *The probability generating function of N_n has the form $\mathbf{E}[z^{N_n}] = \frac{E + Fz}{C - Dz}$, where E, F are real constants and C, D are the same as in lemma 3. With this notation, the expected number of queries handled by the last instance is $\mathbf{E}[N_n] = \frac{\partial \mathbf{E}[z^{N_n}]}{\partial z}_{|z=1} = \frac{FC + ED}{(C-D)^2}$.*

This means that, restricted to $N_n \geq 1$, N_n follows a geometric distribution. Combining the expressions for $\mathbf{E}[\Sigma N]$ and $\mathbf{E}[N_n]$ yields the expected response time, as per Little's law [46]:

$$\mathbf{E}[T] = \frac{1}{\lambda}(\mathbf{E}[\Sigma N] + \mathbf{E}[N_n]) \tag{15}$$

4) Stability: The last (n-th) instance in the chain can be seen as a MMPP/M/1 queue [48], where the average query rate is $\lambda \pi_{\bar{n}}$ (*i.e.,* λ modulated by the probability $\pi_{\bar{n}}$ that all of the first $\bar{n}$ instances are busy), and the service rate is μ. As per [48, section 3.1.1], the system is stable if and only if:

$$\lambda \pi_{\bar{n}} < \mu \Leftrightarrow \frac{\rho^{\bar{n}+1}}{\bar{n}! \sum_{i=0}^{\bar{n}} \rho^i / i!} < 1 \tag{16}$$

C. Autoscaling

When using the autoscaling mechanism described in algorithm 2, the number of instances evolves so that the probability of the last instance being empty, p_e, remains close to a target p_e^*. The probability p_e is one minus the traffic intensity at the entry of the last instance:

$$p_e = \mathbf{P}[N_n = 0] = 1 - \rho \pi_{\bar{n}} \tag{17}$$

Using autoscaling with parameter p_e^*, the relationship between the query rate ρ and the number n of instances is therefore:

$$p_e^* = 1 - \rho \pi_{\bar{n}} \Leftrightarrow \frac{\rho^{\bar{n}+1}}{\bar{n}! \sum_{i=0}^{\bar{n}} \rho^i / i!} = 1 - p_e^* \tag{18}$$

which can be solved numerically to find n as a function of ρ. Furthermore, provided that p_e^* is chosen to be > 0, the system will be stable as per equation (16).

The following theorem provides insight on the behaviour of the system under the asymptotic limit $n \to \infty$.

Theorem 2 *Let $p_e^* > 0$, and consider a system (*not using autoscaling*) with $n \to \infty$ instances and a query rate ρ such that:*

$$\rho(n) = \bar{n} - \sqrt{\bar{n} \log \frac{\bar{n}}{2\pi(1 - p_e^*)^2}} \tag{19}$$

Then, as $n \to \infty$, we have:

$$\lim_{n\to\infty} 1 - \rho(n) \pi_{\bar{n}} = p_e^*$$

In other words, that system behaves as if using autoscaling *with parameter p_e^*, since it satisfies* (18)*.*

Theorem 2 states that, *using autoscaling*, and when $n \to \infty$, the relationship between n and ρ is described by equation (19). Equation (19) describes the influence of p_e^* on the performance of the system: decreasing p_e^* diminishes the number of instances required to sustain a fixed load, but increases the load on the last instance, approaching instability when $p_e^* \to 0$.

Corollary 1 *Using autoscaling,* asymptotic zero-wait time *is achieved: the probability w_n that a task is blocked (i.e., is assigned to a busy instance) vanishes with the number of instances:*

$$w_n = \mathbf{P}\left[\bigcap_{i=1}^{n} N_i > 0\right] = \mathcal{O}\left(\frac{1}{n}\right)$$

Proof: The blocking probability w_n is smaller than the probability that all of the first $\bar{n}$ instances are busy: $w_n \leq \pi_{\bar{n}}$, and $\pi_{\bar{n}} = \frac{1 - p_e^*}{\rho(n)} = \mathcal{O}(\frac{1}{n})$ as per theorem 2. ∎

IV. Numerical Results

The model detailed in section III yields insight about the performance of the proposed architecture. In this section, JFIQ autoscaling is compared to the centralized load-balancing policies introduced in section I-B1 In the remainder of this section, all response times are expressed in units of $1/\mu$ (*i.e.,* a response time of 1 corresponds to the mean service time of the considered application).

A. JFIQ Load-Balancing

To evaluate the performance of JFIQ when using a *fixed* number of instances, the expected number of queries handled by the system is computed (as described in section III-B) as a function of the query rate ρ, for different values of the number n of instances. The expected number of queries handled by the first $\bar{n}$ instances, $\mathbf{E}[\Sigma N]$ (obtained by equation (2)) exhibits an almost linear behaviour with respect to ρ and is therefore not plotted. As the last instance has a special role and accepts more traffic, it is interesting to analyse the expected number of queries assigned to the last instance, $\mathbf{E}[N_n]$. Figure 6 depicts that number, depending on the normalized query rate ρ/n (the

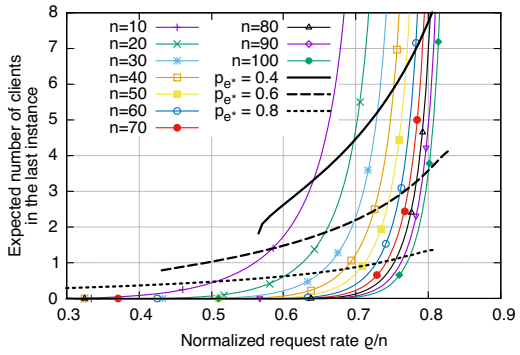


Figure 6. JFIQ LB: expected number $\mathbf{E}[N_n]$ of queries handled by the last instance (derived as per section III-B) as a function of ρ/n, for different values of $n \in \{10, \ldots, 100\}$.

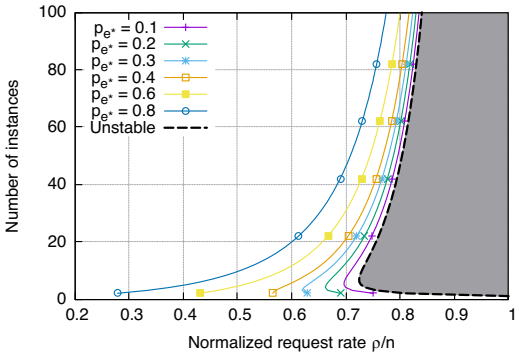


Figure 8. JFIQ autoscaling: number n of instances versus normalized query rate ρ/n, when following autoscaling with target p_e^*, for different values of p_e^*. When $p_e^* \to 0$, the system approaches the instability limit (grey area).

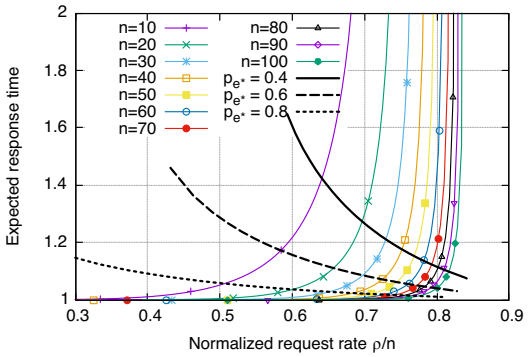


Figure 7. JFIQ LB: expected response time $\mathbf{E}[T]$ (derived as per equation (15)) as a function of ρ/n, for different values of $n \in \{10, \ldots, 100\}$.

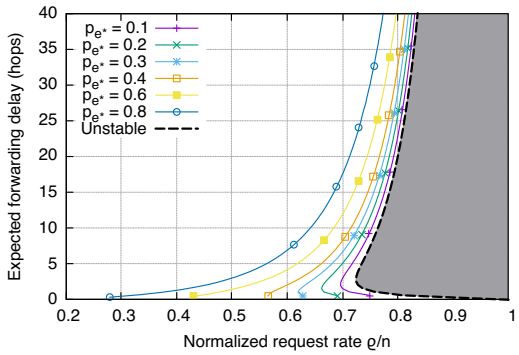


Figure 9. JFIQ autoscaling: forwarding delay (in number of hops) versus normalized query rate ρ/n, when following autoscaling with target p_e^*, for different values of p_e^*.

per-instance efficiency of the system). From figure 6, it appears that this efficiency increases as the number of instances increases (*i.e.,* each instance accomodates a higher query rate), as predicted by theorem 2. The figure also represents the value of $\mathbf{E}[N_n]$ when the number of instances evolves under autoscaling with $p_e^* \in \{0.4, 0.6, 0.8\}$ (thick lines). When an autoscaling policy with a high-enough target p_e^* is applied, the queue length of the last instance remains bounded under a reasonably-low value. For example, with $p_e^* = 0.6$ and $n \leq 100$, the last instance has always less than 3.6 clients enqueued in average.

Having computed the expected number of queries in the system, it is possible to derive the average response time using equation (15), depicted in figure 7: the average response time stays low as long as ρ is not too close to n: for instance, with $n = 100$, the average response time stays below 1.10 for $\rho \leq 81.7$. Under an autoscaling policy with a reasonable target p_e^*, the response time remains under relatively low values – for instance, the response time stays below 1.45 when $p_e^* = 0.6$.

B. Autoscaling scalability

Figure 8 depicts the number n of instances used when the system is using autoscaling as a function of the normalized query rate ρ/n, for different values of the threshold parameter p_e^*. As the the load increases, the number of instances in use increases, with lower values of p_e^* yielding a higher number of instances – as a lower value of p_e^* means that the last instance will be less utilized. For instance when $\rho/n = 0.7$, going from $p_e^* = 0.4$ to $p_e^* = 0.6$ leads to 10 more instances being used (a 48% increase). Thus, diminishing p_e^* has a benefit in terms of number of instances, but also incurs a higher response time, as shown in figure 7. At the extreme, when $p_e^* \to 0$, the system becomes unstable (grey valley in figure 8). Thus, an "optimal" operating point must consist of a trade-off between reducing the number of instances (by letting $p_e^* \to 0$) and ensuring a reasonable response time by staying far from the instability limit (by letting $p_e^* \to 1$).

C. Network Forwarding Delay

Since JFIQ relies upon forwarding the first paquet of a query along several application instances, it incurs additional network forwarding delay. If $\delta > 0$ (in arbitrary time units) is the one-way forwarding delay (half-RTT) between two application instances, then the expected forwarding delay D is $\mathbf{E}[D] = \mathbf{E}[F-1]\delta = \sum_{k=1}^{\bar{n}} \frac{\rho^k/k!}{\sum_{i=0}^{k} \rho^i/i!} \times \delta$ (see appendix V-E for the derivation). Figure 9 depicts this expected delay when the system is using autoscaling, as a function of the normalized query rate ρ/n, for different values of the parameter p_e^*. For instance, when $p_e^* = 0.8$ and for $n < 100$ instances, the expected incurred overhead is smaller than 40δ. In this situation, the overhead is insignificant as long as 40δ is

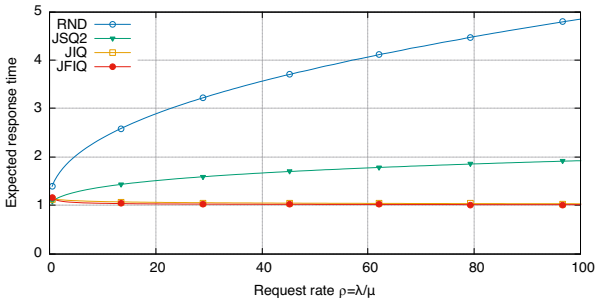


Figure 10. JFIQ autoscaling: expected response time as a function of the query rate ρ, when the number of instances n evolves under autoscaling for $p_e^* = 0.8$. As a comparison point, the response time for centralized policies (RND, JIQ, JSQ_2) for the same query rate and number of instances is represented.

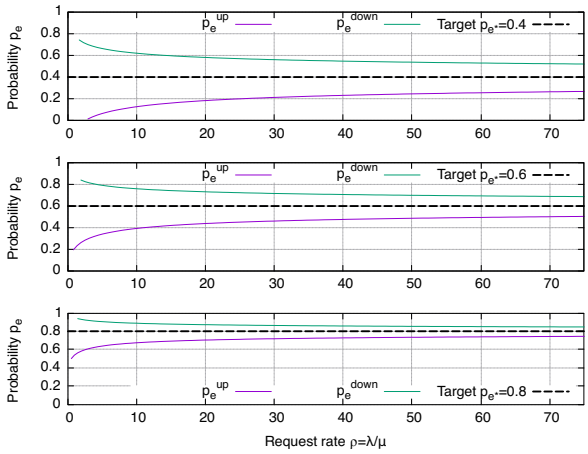


Figure 11. JFIQ autoscaling: down- and up-scaling thresholds $p_e^{\uparrow}$ and $p_e^{\downarrow}$ as a function of the query rate ρ, for different values of target p_e^*. The two lines depict the probability p_e in a system with the same query rate, but with $n-1$ or $n+1$ instances.

sufficiently smaller than the query service time μ^{-1}. For a mid-size application $\mu^{-1} = 100$ ms [49] and a typical data-center half-RTT $\delta = 0.25$ ms [50], there is one order of magnitude between the two values ($40\delta = 10$ ms $= 10\%\mu^{-1}$), thus the overhead can be considered to be insignificant.

D. Comparison to Centralized Policies

Three of the centralized schemes introduced in section I-B1 are compared to JFIQ in this section, namely: (i) "random" (RND); (ii) "shortest-of-two-queues" (JSQ_2); (iii) "join-the-idle-queue" (JIQ). RND and JSQ_2 are interesting because they model completely distributed load-balancing systems such as Maglev [6], Beamer [26] or 6LB [29]. JIQ is also interesting because it models a converse system, in which the load-balancer has a near-perfect view of the state of the application instances, but at the cost of centralized bookkeeping and maintenance of per-flow-state.

The behavior of these three policies is well-known. The RND policy is essentially a virtual M/M/1 queue with query rate λ/n and service time μ, for which the expected response time is [51, p. 98]:

$$\mathbf{E}[T_{\mathsf{RND}}] = \frac{1}{1-\rho/n}\frac{1}{\mu} \tag{20}$$

while the JSQ_2 policy yields the following response time [7]:

$$\mathbf{E}[T_{\mathsf{JSQ}_2}] = \left(\sum_{i=1}^{\infty}\left(\frac{\rho}{n}\right)^{2^i-2}\right)\frac{1}{\mu} \tag{21}$$

and while finally the JIQ policy has the following response time [19]:

$$\mathbf{E}[T_{\mathsf{JIQ}}] = \left(1+\frac{\rho/n}{(1-\rho/n)(1+n)}\right)\frac{1}{\mu} \tag{22}$$

Figure 10 depicts the expected response time $\mathbf{E}[T]$ as a function of the query rate ρ, for a system using JFIQ autoscaling with $p_e^* = 0.8$. As a matter of comparison, the response time for the centralized policies is depicted, when using the same number n of instances as that of the system using autoscaling for each ρ. When $\rho = 30$, the system with autoscaling uses 44 instances and yields a response time of 1.02, whereas RND, JSQ_2, and JIQ with $n = 44$ instances and $\rho = 30$ yield response times of 3.26, 1.60, and 1.05, respectively. Overall, response times are bounded by 1.16 for $\rho \leq 100$. Due to taking the load of all instances into account, JFIQ autoscaling performs better than policies RND and JSQ_2 (when $\rho > 1.2$), respectively, and yields results close to those of the reference policy JIQ. To conclude, using a conservative value for p_e^* (*e.g.,* $p_e^* = 0.8$) allows for JFIQ to provide a lower response time than the lightweight distributed policies (RND or JSQ_2) when using the same number of application instances. Further, the performance of JFIQ is close to that of the near-ideal centralized policy JIQ.

E. Scaling Thresholds

To avoid oscillations, the thresholds $p_e^{\downarrow}$ and $p_e^{\uparrow}$, above (and below) which autoscaling will be triggered, must be chosen appropriately. Notably, it must be ensured that, after an upscaling decision, a downscaling decision will not be taken by the newly-introduced instance (and vice versa). This is achieved by choosing $p_e^{\downarrow}$ and $p_e^{\uparrow}$ so that p_e will be equal to p_e^* after autoscaling. Formally, if $\rho^{n,\uparrow}$ is the target query rate which triggers upscaling with n instances, *i.e.,* if $\rho^{n,\uparrow}$ is the rate at which:

$$p_e^n(\rho^{n,\uparrow}) = p_e^{n,\uparrow}, \tag{23}$$

then $\rho^{n,\uparrow}$ is defined as the solution of:

$$p_e^{n+1}(\rho^{n,\uparrow}) = p_e^* \tag{24}$$

Solving equation (24) to find $\rho^{n,\uparrow}$ allows finding $p_e^{n,\uparrow}$ as per equation (23), *i.e.*:

$$p_e^{n,\uparrow} = p_e^n[(p_e^{n+1})^{-1}(p_e^*)]$$

For downscaling, the inverse operation is applied to find $p_e^{n,\downarrow}$:

$$p_e^{n,\downarrow} = p_e^n[(p_e^{n-1})^{-1}(p_e^*)]$$

Figure 11 depicts $p_e^{n,\updownarrow}$ as a function of ρ. With $p_e^* = 0.4$, the thresholds $p_e^{n,\updownarrow}$ are far away from the actual target p_e^* (with $|p_e^{n,\updownarrow} - p_e^*| \geq 0.13$ for $n \leq 100$) – meaning that the system

will deviate from its target operating point before up- or down-scaling is triggered. With $p_e^* = 0.8$, the thresholds are close to the target (with $|p_e^{n,\updownarrow} - p_e^*| \leq 0.13$ for $n \geq 17$), meaning that the system will deviate less from its expected behavior – but also implying that it is harder to tracker whether the threshold has been crossed.

F. Simulations

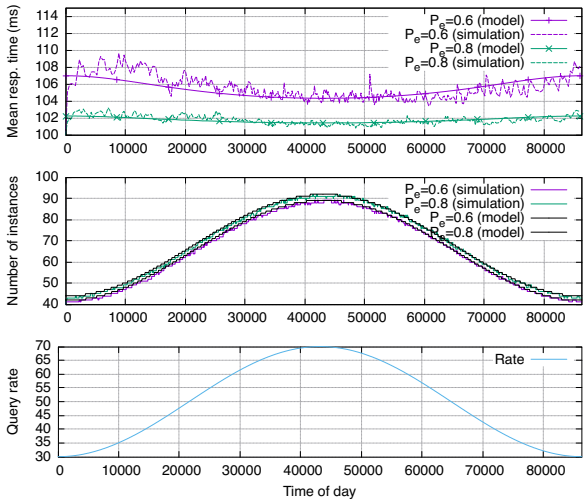
This section presents results of numerical simulations of a system implementing JFIQ, with the following objectives: (i) capturing the dynamics of algorithm 2 when the query rate $\rho(t)$ actually varies (since the model only considers a steady load rate ρ); (ii) analysing the whole distribution of response times (since the model only yields the expected response time); (iii) analysing the behavior of the system under various service time distributions and scheduling disciplines (thus alleviating the limitation of using a FIFO policy and exponential response times, introduced in the model for tractability); (iv) analysing the influence of the main parameter of the system, p_e^*, when actually running the autoscaling algorithm; (v) comparing the behavior of the system against other load-balancing algorithms when they actually undergo autoscaling; (vi) analysing the distribution of the network forwarding delay (since the model only yields the expectation); and (vii) understanding the behavior of the system under non-smooth query rates.

1) Simulation Setup: JFIQ has been implemented as part of a general-purpose queuing simulator [52]. Queries are drawn according to a Poisson process or a modulated Poisson process, and are assigned to a queue according to algorithm 1. Upon arrival or departure of queries at/from the last queue s_n, the tracking mechanism introduced in algorithm 2 is applied, and if autoscaling is deemed necessary, the queuing network is updated accordingly. For a more realistic evaluation, queues implement a Processor-Sharing (PS) policy with arbitrary response time distributions. Unless specified otherwise, service time distributions are taken with 100 ms as average response time (*i.e.,* $\mu = 10\ \mathrm{s}^{-1}$), as observed for example in large-scale Web deployments [49, figure 7(a)]. As a baseline, the centralized behaviors introduced in section IV-D (RND, JSQ_2, JIQ) have also been implemented.

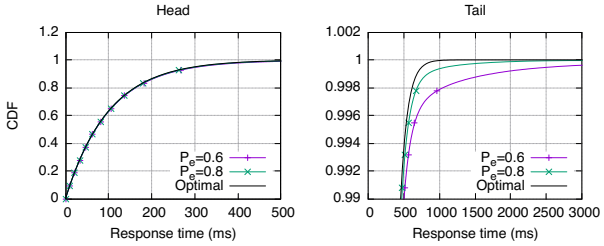
To understand the behavior of the system when exposed to a simple varying load, the simulations were conducted with a simple diurnal pattern $\rho(t)$, given by (with t in seconds):

$$\rho(t) = 50 - 20\cos\left(\frac{2\pi t}{86400}\right) \tag{25}$$

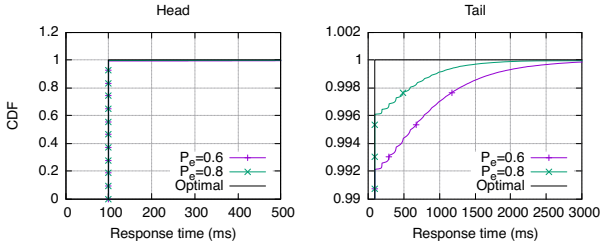
This choice of parameters provides confidence that the system behaves properly in reasonably large-scale scenarios[3]. Queries are injected with a Poisson process whose rate $\lambda = \rho \cdot \mu$ is modulated according to (25), for $t \in [0, 86400]$ s. In particular, query rates vary between 300 and 700 req/s, and the expected number of queries injected into the system is: $\int_0^{86400} \lambda(t)\mathrm{d}t = 43.2 \cdot 10^6$. This constitutes a sufficiently large sample to obtain statistically-significant results: the obtained response time distributions for a second run of the experiments were acceptably identical (the p-value from a Kolmogorov-Smirnov test [53] was $p = 0.048$).



(a) Evolution of the average response time and number of instances (exponential service times)



(b) CDF of response time (exponential service times)



(c) CDF of response time (constant service times)

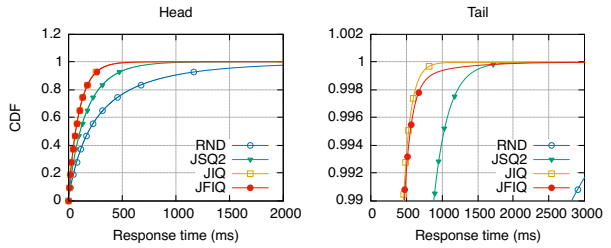
Figure 12. JFIQ autoscaling, when the number n of instances evolves under autoscaling for $p_e^* \in \{0.6, 0.8\}$, and $\rho(t)$ evolves with a diurnal pattern. Simulations with exponential and constant service times ($1/\mu = 100$ ms), under the processor-sharing discipline.

2) Influence of the target p_e^*: To explore the influence of the target p_e^*, experiments were run for $p_e^* \in \{0.6, 0.8\}$ (with exponential service times) – these values were chosen as they appear to yield stable enough results from figures 7 and 8. Results for these experiments are depicted in figure 12[4]. Figure 12a depicts the number of instances used by the autoscaling system over a 24-hour experiment goes, with the instantaneous response time observed by clients[5]. Analytical

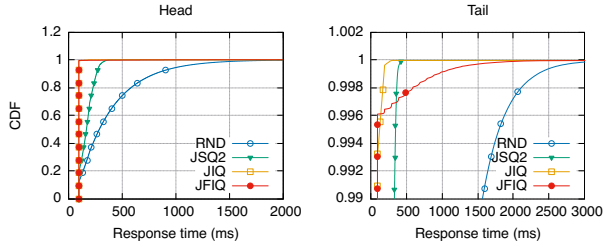
[3]This yields $n(t)$ between approximately 40 and 90. More instances would not be realistic, since an SRv6 header can accommodate $n = 89$ instance addresses in a MTU-sized TCP SYN packet of 1500 bytes.

[4]In all CDF figures, the left-hand side graph depicts the head of the distribution, and the right-hand side graph depicts the tail of the distribution

[5]Response times are smoothed with an EWMA filter with parameter $\alpha = 1 - \exp(-\delta t/\tau)$, with δt the time interval between two successive points, and $\tau = 10$ min.

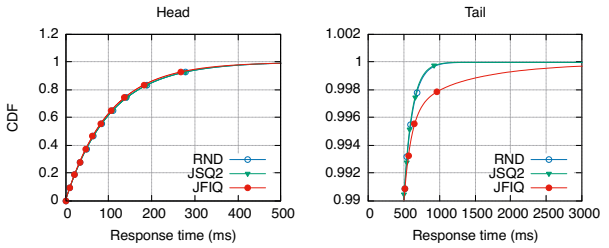


(a) CDF of response time (exponential service times)

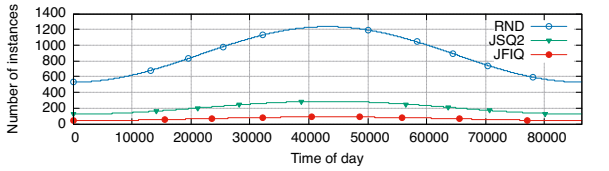


(b) CDF of response time (constant service times)

Figure 13. JFIQ autoscaling, when the number n of instances evolves under autoscaling for $p_e^* = 0.8$, and $\rho(t)$ evolves with a diurnal pattern. Simulations with exponential and constant service times (average service time 100 ms), under the processor-sharing discipline. Baseline: RND, JSQ_2, and JIQ with the same query rate $\rho(t)$ and number of instances $n(t)$.



(a) CDF of response time



(b) Evolution of the number of instances

Figure 14. JFIQ autoscaling, when the number n of instances evolves under autoscaling for $p_e^* = 0.6$, and $\rho(t)$ evolves with a diurnal pattern. Simulations with exponential service times (average service time 100 ms) under the processor-sharing discipline. Baseline: RND and JSQ_2 with the same query rate $\rho(t)$, and with a number of instances $n_{\mathsf{RND},\mathsf{JSQ}_2}(t)$ such that the overall average response time with RND, JSQ_2 is identical to that of JFIQ (106 ms).

values as obtained by using equation (15) from section III are superposed to the simulation results – showing a good fit between experimental and theoretical results. As predicted in section III, diminishing p_e^* reduces the number of necessary instances, but while increasing response time. The daily cost[6] with $p_e^* = 0.6$ is 1560 VM·h, as compared to 1620 VM·h with $p_e^* = 0.8$. This cost reduction comes at the cost of increasing the average response time from 102 ms to 106 ms.

Figure 12b depicts the distribution of client response times over the 24 hours. Overall, the distribution approximates the optimal[7], with almost identical 99$^{\text{th}}$ percentiles. The 99.9$^{\text{th}}$ percentile with $p_e^* = 0.8$ also approximates the optimal, whereas with $p_e^* = 0.6$ it deviates with a factor 2.5. Figure 12c depicts the response time distributions, when running the same experiment with constant service times. Again, the distribution approximates the optimal, with more than 99% of queries served in 100 ms. Those queries served in more than 100 ms amount to 0.8% with $p_e^* = 0.6$, and to 0.4% with $p_e^* = 0.8$. This aligns with corollary 1, which states that the probability for a task to wait is smaller than $(1 - p_e^*)/\rho$ (and using the average value $\rho = 50$ gives these values). The 99.9$^{\text{th}}$ percentile with $p_e^* = 0.8$ is half as much as with $p_e^* = 0.6$, but the former policy led to a resource cost of 1620 VM · h instead of 1560 VM · h with the latter.

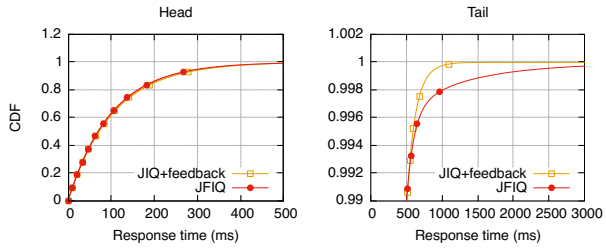
3) Response time gain with equal number of instances: To fairly compare JFIQ to the centralized policies defined in section IV-D, the *server schedule* (*i.e.,* the function $n(t)$ giving the number of instances as a function of the time of day) for $p_e^* = 0.8$ and exponential response times is recorded. The 24-hour experiment is then run with policies RND, JSQ_2, JIQ using the same query rate pattern $\rho(t)$ and server schedule $n(t)$. Figure 13 depicts the results, with figures 13a and 13b depicting the response time distributions for the four tested policies and the two tested service times. Tail response time latency is greatly improved: with constant service times, a reduction of 16× (RND) and 3.4× (JSQ_2) in the 99$^{\text{th}}$ percentile latency can be observed. Figure 13 shows that JFIQ autoscaling is able to closely match the reference policy JIQ, with noticeable differences only above the 99.6$^{\text{th}}$ percentile. This experiment shows that JFIQ outperforms standard consistent-hashing based load-balancing frameworks implementing RND [6] or JSQ_2 [29], and offers close-to optimal performance while necessitating no centralized monitoring.

4) Cost gain with equal average response time: This section attempts to quantify the potential resource cost gains that can be achieved by using JFIQ. Using equations (20) and (21), two schedules $n_{\mathsf{RND}}(t)$ and $n_{\mathsf{JSQ}_2}(t)$ are computed offline so that the expected response time is constant at 106 ms[8] for the modulated query rate $\rho(t)$ for RND and JSQ_2. This implements a "perfect" omniscient autoscaling algorithm for RND and JSQ_2 – as the most conservative possible comparison to JFIQ. The results are depicted in figure 14, where figure 14a shows that, as expected by the design of the experiment, the response time distributions achieved with JFIQ, RND and JSQ_2 are almost identical. Figure 14b depicts the resource cost gains of using JFIQ as compared to the two other policies: while with JFIQ the number of instances oscillates between 40 and 89, with JSQ_2 it was found to oscillate between 123 and 287, and with RND between 531 and 1237. Overall, the resource cost gain from using JFIQ autoscaling (1560 VM · h) is of 3.2× over JSQ_2 (4920 VM · h), and of 14× over RND (21200 VM · h).

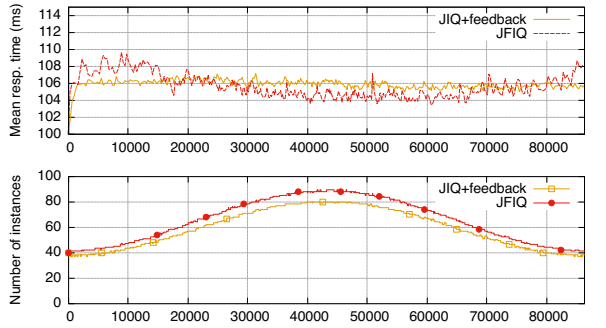
[6]defined as the total number of VM-hours used by the system: $\int_0^{86400} n(t)\mathrm{d}t$.

[7]defined as the response time distribution if there was an infinity of instances, *i.e.,* simply the service time distribution.

[8]106 ms is the average response time for the 24-hour simulation with $p_e^* = 0.6$ and exponential response times



(a) CDF of response time



(b) Evolution of the number of instances

Figure 15. JFIQ autoscaling, when the number n of instances evolves under autoscaling for $p_e^* = 0.6$, and $\rho(t)$ evolves with a diurnal pattern. Simulations with exponential service times (average service time 100 ms) under the processor-sharing discipline. Baseline: JIQ with the same query rate $\rho(t)$, and with a number of instances $n(t)$ varying according to a feedback mechanism, such that the overall average response time with JIQ is identical to that of JFIQ (106 ms).

5) Comparison to centralized monitoring: Section IV-F4 has compared JFIQ to two simple, distributed policies (RND, JSQ_2). To also provide a comparison with the near-optimal centralized policy JIQ, a mechanism to compute $n(t)$ such that $\mathbf{E}[T_{\text{JIQ}}]$ is equal to a target value is necessary. To that aim, a feedback mechanism is implemented, whereby the load-balancer keeps track of the response time of *all* queries: response times are passed through an EWMA filter, and if the up/down threshold is crosses, up/down scaling is requested. This emulates an efficient reactive centralized autoscaling system. As it assumes that all queries can be tracked by a *single* load-balancer, it overestimates performance with respect to realistic scalable implementations, hence acting as a conservative point of comparison. To match JFIQ autoscaling with $p_e^* = 0.6$, a response time of 106 ms is targeted (with up/down threshold of 108 ms and 104 ms) – with an EWMA window of 100 s to avoid oscillations.

Results are depicted in figure 15. Figure 15a shows that, for an identical average response time, JFIQ autoscaling yields a slightly lower response time than JIQ for the first 99 percentiles, and a slightly larger response time for the last percentile. Besides, figure 15b shows that JFIQ autoscaling had a cost of 1560 VM · h, approaching (within 11%) the cost achieved by the JIQ policy (1410 VM · h).

6) Influence of the forwarding delay: The JFIQ algorithm introduced in section II-B introduces forwarding latency on TCP SYN packets. Typical worst-case machine-to-machine RTTs observed in data-centers are around 400 μs (90-th percentile in [50, figure 4(a)]), *i.e.,* a machine-to-machine latency

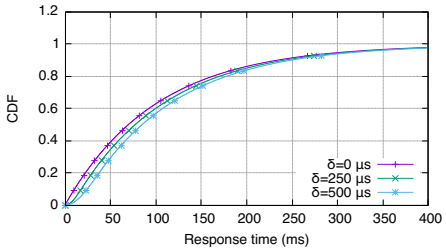
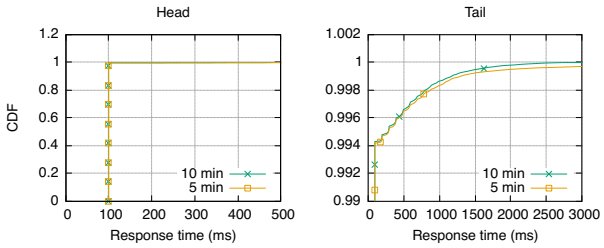
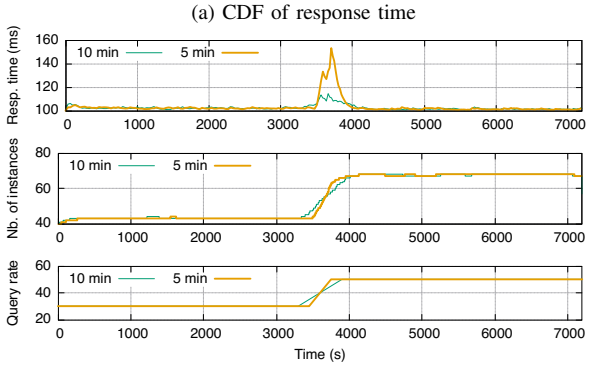


Figure 16. JFIQ autoscaling, when the number n of instances evolves under autoscaling for $p_e^* = 0.6$, and $\rho(t)$ evolves with a diurnal pattern. Simulations with exponential service times (average service time 100 ms), under the processor-sharing discipline. Different network forwarding delays $\delta \in \{0, 250, 500\}$ μs.



(a) CDF of response time



(b) Evolution of the number of instances

Figure 17. JFIQ autoscaling, when the number n of instances evolves under autoscaling for $p_e^* = 0.8$, and queries arrive according to a bursty patterns $\rho(t)$. Simulations with constant service times (service time 100 ms) under the processor-sharing discipline.

of 200 μs. The software latency incurred by processing an SRv6 header in algorithm 1 is typically 50 μs (for the kernel-bypass implementation described in [29]). Therefore, the total forwarding delay (software+network) incurred by algorithm 1 on TCP SYN packets is approximately 250 μs. Thus, to study the influence of the forwarding delay, the simulation described in figure 12b (with $p_e^* = 0.6$) was repeated, with an extraneous constant forwarding delay $\delta \in \{250, 500\}$ μs. Figure 16 depicts the results: the median response time is inflated by 7.50 ms with $\delta = 250$ μs, and by 15.2 ms with $\delta = 500$ μs. For compute-intensive workloads with response time of the order of 100 ms [49], this extra latency is negligible.

7) Reaction to Bursty Traffic: With the JFIQ algorithm, when exposed to a query rate increase, the last instance might have to accept an important number of queries before deciding that upscaling is necessary. To analyze this behavior, a simulation was conducted where a "flash crowd" query pattern $\rho(t)$ was applied [54], with a query rate first constant

(with $\lambda = 300\text{s}^{-1}$ *i.e.,* $\rho = 30$), then linearly increasing within a short ramp-up period (of 5 min or 10 min), then constant again for the remainder of the simulation (with $\lambda = 500\text{s}^{-1}$ *i.e.,* $\rho = 30$). Figure 17b depicts the evolution of the number of application instances. The number of application instances quickly increases in reaction to the flash crowd, and with a ramp-up period is 10 min long, the response time remains reasonably low. In the simulation with the shorter ramp-up period, the last instance is more overloaded, yielding a higher instantaneous mean response time (up to 154 ms). This is confirmed in figure 17a, which shows a longer tail for the response times in the latter simulation.

V. Conclusion

This paper has introduced a novel load-balancing policy, JFIQ, and a decentralized autoscaling policy, forming a unified load-balancing and autoscaling framework. Operating at the network-layer with SRv6, the proposed architecture is **distributed** (allowing for packets to be treated by any load-balancer instance), requires **no centralized monitoring** (as decisions are taken by the application instances themselves), and necessitates **no per-flow state** (as the identity of the accepting application is embedded in a covert channel).

In this paper, JFIQ has been specified, analyzed theoretically, and studied experimentally. JFIQ autoscaling is proven to provide ***asymptotic zero-wait-time*** when the number of application instances tends to infinity. JFIQ has been found to (i) have the same complexity, but better performance, than simple distributed approaches ($\mathsf{RND}, \mathsf{JSQ}_2$), and to (ii) have comparable performance, but lower complexity (by not requiring central monitoring) than JIQ. This paper has shown that, for the evaluated query patterns, using JFIQ instead of RND decreases the 99^{th}-percentile of response time by $16\times$ if using the same resources, or decreases resource costs by $14\times$ if targeting the same average response time.

References

[1] N. Dragoni *et al.*, "Microservices: yesterday, today, and tomorrow," in *Present and Ulterior Software Engineering*. Springer, 2017, pp. 195–216.

[2] D. Bernstein, "Containers and cloud: From lxc to docker to kubernetes," *IEEE Cloud Computing*, vol. 1, no. 3, pp. 81–84, 2014.

[3] R. P. Goldberg, "Survey of virtual machine research," *Computer*, vol. 7, no. 6, pp. 34–45, 1974.

[4] Y. Al-Dhuraibi *et al.*, "Elasticity in cloud computing: state of the art and research challenges," *IEEE Transactions on Services Computing*, vol. 11, no. 2, pp. 430–447, 2018.

[5] D. Thaler and C. Hopps, "Multipath issues in unicast and multicast next-hop selection," in *Requests For Comments*. IETF, 2000, no. 2991.

[6] D. E. Eisenbud *et al.*, "Maglev: A fast and reliable software network load balancer," in *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, 2016, pp. 523–535.

[7] M. Mitzenmacher, "The power of two choices in randomized load balancing," *IEEE Transactions on Parallel and Distributed Systems*, vol. 12, no. 10, pp. 1094–1104, 2001.

[8] "HAProxy: the reliable, high-performance TCP/HTTP load balancer." [Online]. Available: http://www.haproxy.org

[9] R. Wang *et al.*, "Openflow-based server load balancing gone wild." *HotICE*, vol. 11, pp. 12–12, 2011.

[10] M. Z. Hasan *et al.*, "Integrated and autonomic cloud resource scaling," in *Network Operations and Management Symposium (NOMS), 2012 IEEE*. IEEE, 2012, pp. 1327–1334.

[11] A. Beloglazov and R. Buyya, "Adaptive threshold-based approach for energy-efficient consolidation of virtual machines in cloud data centers," 2010.

[12] D. Kliazovich, P. Bouvry, and S. U. Khan, "Dens: data center energy-efficient network-aware scheduling," *Cluster computing*, vol. 16, no. 1, pp. 65–75, 2013.

[13] B. Xavier, T. Ferreto, and L. Jersak, "Time provisioning evaluation of kvm, docker and unikernels in a cloud platform," in *2016 16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*. IEEE, 2016, pp. 277–280.

[14] C. Filsfils *et al.*, "The segment routing architecture," in *2015 IEEE Global Communications Conference (GLOBECOM)*. IEEE, 2015, pp. 1–6.

[15] D. L. Eager, E. D. Lazowska, and J. Zahorjan, "Adaptive load sharing in homogeneous distributed systems," *IEEE transactions on software engineering*, no. 5, pp. 662–675, 1986.

[16] B. Schroeder and M. Harchol-Balter, "Evaluation of task assignment policies for supercomputing servers: The case for load unbalancing and fairness," *Cluster Computing*, vol. 7, no. 2, pp. 151–161, 2004.

[17] V. Gupta *et al.*, "Analysis of join-the-shortest-queue routing for web server farms," *Performance Evaluation*, vol. 64, no. 9-12, pp. 1062–1081, 2007.

[18] R. D. Foley, D. R. McDonald *et al.*, "Join the shortest queue: stability and exact asymptotics," *The Annals of Applied Probability*, vol. 11, no. 3, pp. 569–607, 2001.

[19] Y. Lu *et al.*, "Join-idle-queue: A novel load balancing algorithm for dynamically scalable web services," *Performance Evaluation*, vol. 68, no. 11, pp. 1056–1071, 2011.

[20] M. van der Boor *et al.*, "Scalable load balancing in networked systems: A survey of recent advances," *arXiv preprint arXiv:1806.05444*, pp. 1–69, 2018.

[21] X. Liu and L. Ying, "A simple steady-state analysis of load balancing algorithms in the sub-Halfin-Whitt regime," *arXiv:1804.02622*, pp. 1–14, 2018.

[22] P. Patel *et al.*, "Ananta: cloud scale load balancing," in *ACM SIGCOMM Computer Communication Review*, vol. 43, no. 4. ACM, 2013, pp. 207–218.

[23] R. Gandhi *et al.*, "Duet: Cloud scale load balancing with hardware and software," *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 4, pp. 27–38, 2015.

[24] ——, "Rubik: Unlocking the power of locality and end-point flexibility in cloud scale load balancing." in *USENIX Annual Technical Conference*, 2015, pp. 473–485.

[25] R. Miao *et al.*, "Silkroad: Making stateful layer-4 load balancing fast and cheap using switching asics," in *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*. ACM, 2017, pp. 15–28.

[26] V. Olteanu *et al.*, "Stateless datacenter load-balancing with beamer," in *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*. USENIX Association, 2018, pp. 125–139.

[27] J. T. Araújo *et al.*, "Balancing on the edge: Transport affinity without network state," in *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, 2018, pp. 111–124.

[28] J. McCauley *et al.*, "Thoughts on load distribution and the role of programmable switches," *SIGCOMM Comput. Commun. Rev.*, vol. 49, no. 1, pp. 18–23, Feb. 2019.

[29] Y. Desmouceaux *et al.*, "6LB: Scalable and application-aware load balancing with segment routing," *IEEE/ACM Transactions on Networking*, vol. 26, no. 2, pp. 819–834, April 2018.

[30] R. Han *et al.*, "Lightweight resource scaling for cloud applications," in *Cluster, Cloud and Grid Computing (CCGrid), 2012 12th IEEE/ACM International Symposium on*. IEEE, 2012, pp. 644–651.

[31] P. Marshall, K. Keahey, and T. Freeman, "Elastic site: Using clouds to elastically extend site resources," in *2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing*, May 2010, pp. 43–52.

[32] R. Han *et al.*, "Enabling cost-aware and adaptive elasticity of multi-tier cloud applications," *Future Generation Computer Systems*, vol. 32, pp. 82–98, 2014.

[33] P. D. Kaur and I. Chana, "A resource elasticity framework for qos-aware execution of cloud applications," *Future Generation Computer Systems*, vol. 37, pp. 14–25, 2014.

[34] N. Vasić *et al.*, "Dejavu: accelerating resource allocation in virtualized environments," in *ACM SIGARCH computer architecture news*, vol. 40, no. 1. ACM, 2012, pp. 423–436.

[35] S. Islam *et al.*, "Empirical prediction models for adaptive resource provisioning in the cloud," *Future Generation Computer Systems*, vol. 28, no. 1, pp. 155–162, 2012.

[36] Y. Tan *et al.*, "Prepare: Predictive performance anomaly prevention for virtualized cloud systems," in *2012 IEEE 32nd International Conference on Distributed Computing Systems*. IEEE, 2012, pp. 285–294.

[37] E. Kalyvianaki, T. Charalambous, and S. Hand, "Self-adaptive and self-configured cpu resource provisioning for virtualized servers using kalman filters," in *Proceedings of the 6th international conference on Autonomic computing*. ACM, 2009, pp. 117–126.

[38] L. Baresi *et al.*, "A discrete-time feedback controller for containerized cloud applications," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 2016, pp. 217–228.

[39] S.-M. Park and M. Humphrey, "Self-tuning virtual machines for predictable escience," in *2009 9th IEEE/ACM International Symposium on Cluster Computing and the Grid*. IEEE, 2009, pp. 356–363.

[40] Q. Zhang, L. Cherkasova, and E. Smirni, "A regression-based analytic model for dynamic resource provisioning of multi-tier applications," in *Fourth International Conference on Autonomic Computing (ICAC'07)*. IEEE, 2007, pp. 27–27.

[41] B. Urgaonkar *et al.*, "Agile dynamic provisioning of multi-tier internet applications," *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*, vol. 3, no. 1, 2008, article no. 1.

[42] D. Goldsztajn *et al.*, "Controlling the number of active instances in a cloud environment," *ACM SIGMETRICS Performance Evaluation Review*, vol. 45, no. 2, pp. 15–20, 2018.

[43] D. Mukherjee *et al.*, "Optimal service elasticity in large-scale distributed systems," *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, vol. 1, no. 1, 2017, article no. 25.

[44] B. Pit-Claudel *et al.*, "Stateless load-aware load balancing in P4," in *Proc. IEEE 26th International Conference on Network Protocols (ICNP), 1st P4 Workshop in Europe (P4WE)*. IEEE, 2018, pp. 418–423.

[45] C. Filsfils *et al.*, "IPv6 Segment Routing Header (SRH)," RFC 8754, Mar. 2020. [Online]. Available: https://rfc-editor.org/rfc/rfc8754.txt

[46] J. D. Little, "A proof for the queuing formula: L=λw," *Operations research*, vol. 9, no. 3, pp. 383–387, 1961.

[47] A. Gandhi *et al.*, "Exact analysis of the M/M/k/setup class of Markov chains via recursive renewal reward," in *ACM SIGMETRICS Performance Evaluation Review*, vol. 41, no. 1. ACM, 2013, pp. 153–166.

[48] W. Fischer and K. Meier-Hellstern, "The markov-modulated poisson process (MMPP) cookbook," *Performance evaluation*, vol. 18, no. 2, pp. 149–171, 1993.

[49] A. Roy *et al.*, "Inside the social network's (datacenter) network," in *ACM SIGCOMM Computer Communication Review*, vol. 45, no. 4. ACM, 2015, pp. 123–137.

[50] C. Guo *et al.*, "Pingmesh: A large-scale system for data center network latency measurement and analysis," in *ACM SIGCOMM Computer Communication Review*, vol. 45, no. 4. ACM, 2015, pp. 139–152.

[51] L. Kleinrock, *Queueing Systems*. Wiley Interscience, 1975, vol. I: Theory.

[52] M. Enguehard and Y. Desmouceaux, "marceleng/queueing-network-simulator: a simulator for queueing networks," https://github.com/marceleng/queueing-network-simulator, Jan 2019.

[53] I. T. Young, "Proof without prejudice: use of the Kolmogorov-Smirnov test for the analysis of histograms from flow systems and other sources." *Journal of Histochemistry & Cytochemistry*, vol. 25, no. 7, pp. 935–941, 1977.

[54] I. Ari *et al.*, "Managing flash crowds on the internet," in *11th IEEE/ACM International Symposium on Modeling, Analysis and Simulation of Computer Telecommunications Systems, 2003. MASCOTS 2003.* IEEE, 2003, pp. 246–249.

[55] M. Mitzenmacher and E. Upfal, *Probability and computing: Randomized algorithms and probabilistic analysis*. Cambridge university press, 2005.

APPENDIX

A. Proof of Lemma 1

Proof: Since the restriction of the chain to $\{0,\dots,\bar{n}\}\times\{j,j+1,\dots\}$ (including transitions to the $(j-1)^{\text{th}}$ row) is isomorphic to its restriction to $\{0,\dots,\bar{n}\}\times\{j+1,j+2,\dots\}$ (including transitions to the j^{th} row), and since there is exactly one more query in the n-th instance in the latter restriction, the mean reward of $z^{N_n(t)}$ earned when going from $(i,j+1)$ to row j is the same as the mean reward of $z^{N_n(t)+1}$ earned when going from (i,j) to row $j-1$. Hence, $R^D_{(i,j+1)\to(\star,j)} = \mathbf{E}[\int_{(i,j+1)\to(\star,j)} z^{N_n(t)}\mathrm{d}t] = \mathbf{E}[\int_{(i,j)\to(\star,j-1)} z^{N_n(t)+1}\mathrm{d}t] = zR^D_{(i,j)\to(\star,j-1)}$. ∎

B. Proof of Lemma 3

Proof: By induction, using (7) and (8), we can write $R^D_i = \alpha_i R^D_0 + \beta_i z$, where α_i, β_i are real constants. In particular, $R^D_{\bar{n}} = \alpha_{\bar{n}} R^D_0 + \beta_{\bar{n}} z$. But, (9) gives $R^D_{\bar{n}} = \frac{Gz + HR^D_0}{I - Jz}$. Combining both forms yields $R^D_0 = \frac{Gz-(I-Jz)\beta_{\bar{n}}z}{\alpha_{\bar{n}}(I-Jz)-H} := \frac{A_0 z + B_0 z^2}{C - Dz}$, and injecting in $R^D_i = \alpha_i R^D_0 + \beta_i z$ gives the desired form. For $i = \bar{n}$, this precisely gives $R^D_{\bar{n}} = \frac{\alpha_{\bar{n}}Gz - \beta_{\bar{n}}zH}{\alpha_{\bar{n}}(I-Jz)-H}$, hence $B_{\bar{n}} = 0$. ∎

C. Proof of Lemma 4

Proof: By induction, we can express the R^L_i as a function of R^L_1, using equation (11): $R^L_i = \gamma_i R^L_1 + \epsilon_i$, where γ_i, ϵ_i are real constants. On the one hand, this gives $R^L_{\bar{n}} = \gamma_{\bar{n}} R^L_1 + \epsilon_{\bar{n}}$. On the other hand, (12) gives $R^L_{\bar{n}} = K + LR^L_1 + MR^D_{\bar{n}}$. Combining these two forms gives $R^L_1 = N + OR^D_{\bar{n}} := \frac{E_1 + F_1 z}{C - Dz}$, hence the desired form for all i by injecting in $R^L_i = \gamma_i R^L_1 + \epsilon_i$. ∎

D. Proof of Theorem 2

Proof: Let $p^*_e > 0$, $\beta = \log(2\pi(1-p^*_e)^2)$, and $\rho(n) = \bar{n} - \sqrt{\bar{n}(\log\bar{n} - \beta)}$, the aim of this section is to prove that $\lim_{n\to\infty}\rho(n)\pi_{\bar{n}} = 1 - p^*_e$. Using equation (1), it is possible to write: $\rho(n)\pi_{\bar{n}} = \rho(n)^{\bar{n}+1}e^{-\rho(n)}/(\bar{n}!S_n)$, where $S_n = e^{-\rho(n)}\sum_{i=0}^{\bar{n}}\rho(n)^i/i!$. First, it will be shown that $\lim S_n = 1$. To that aim, one can notice that $S_n = \mathbf{P}[Y_n \le \bar{n}]$, where Y_n is a Poisson random variable of parameter $\rho(n)$. Applying the following Chernoff bound [55]:

$$\mathbf{P}[X \ge \alpha] \le e^{-t\alpha}\mathbf{E}[e^{tX}]$$

to Y_n with $\alpha = n$ and $t = \log(1 + \frac{1}{\sqrt{n}})$ yields:

$$\begin{aligned}\mathbf{P}[Y_n \ge n] &\le e^{-tn}\mathbf{E}[e^{tY_n}] = e^{-tn+\rho(n)(e^t-1)}\\ &= e^{-n\log(1+\frac{1}{\sqrt{n}})+(\sqrt{n}-\sqrt{\log n-\beta})}\\ &= e^{\frac{1}{2}-\sqrt{\log n-\beta}+O(\frac{1}{\sqrt{n}})} \to_{n\to\infty} 0\end{aligned}$$

leading to $\lim_{n\to\infty} S_n = 1$. Therefore:

$$\begin{aligned}\rho(n)\pi_{\bar{n}} &= \frac{\rho(n)^{\bar{n}+1}e^{-\rho(n)}}{\bar{n}!(1+o(1))} = \frac{\rho(n)^{\bar{n}+1}e^{-\rho(n)}}{\bar{n}^{\bar{n}}e^{-\bar{n}}\sqrt{2\pi\bar{n}}(1+o(1))}\\ &= \frac{(\bar{n}-\sqrt{\bar{n}(\log\bar{n}-\beta)})^{\bar{n}+1}e^{-\bar{n}+\sqrt{\bar{n}(\log\bar{n}-\beta)}}}{\bar{n}^{\bar{n}}e^{-\bar{n}}\sqrt{2\pi\bar{n}}(1+o(1))}\\ &= \frac{e^{(\bar{n}+1)\log(1-\sqrt{(\log\bar{n}-\beta)/\bar{n}})+\sqrt{\bar{n}(\log\bar{n}-\beta)}}\sqrt{\bar{n}}}{\sqrt{2\pi}(1+o(1))}\\ &= \frac{e^{\beta/2+\mathcal{O}(\bar{n}^{-1/2}\log^{3/2}\bar{n})}}{\sqrt{2\pi}(1+o(1))} \to_{n\to\infty} \sqrt{\frac{e^\beta}{2\pi}} = 1 - p^*_e\end{aligned}$$

∎

E. Derivation of the Network Forwarding Overhead

Proof: Let $F = \min\{k \in \{1, \dots, n\} : N_k = 0\}$ be the index of the first available instance. When a query enters the first instance, it will incur a forwarding overhead of $D = (F-1)\delta$ until being accepted by an application instance. We have, for all $k \in \{1, \dots, n\}$, $\mathbf{P}[F = k] = \mathbf{P}[F \geq k] - \mathbf{P}[F \geq k+1]$, and thus $\mathbf{E}[F] = \sum_{k=1}^{n} \mathbf{P}[F \geq k]$. But $\mathbf{P}[F \geq k] = \mathbf{P}[N_1 = 1 \cap N_2 = 1 \cap \dots \cap N_{k-1} = 1]$ (in other words, if all the $(k-1)$ first instances are busy, the query will be handled by instance k or greater). Since for all $k \in \{1, \dots, \bar{n}\}$, the total number of queries handled by the first k instances follows the law of an M/M/k/k queue, this can be rewritten as $\mathbf{P}[F \geq k] = \frac{\rho^{k-1}/(k-1)!}{\sum_{i=0}^{k-1} \rho^i/i!}$. This finally yields for the expected forwarding delay: $\mathbf{E}[D] = \mathbf{E}[F-1]\delta = \sum_{k=1}^{\bar{n}} \frac{\rho^k/k!}{\sum_{i=0}^{k} \rho^i/i!} \times \delta$. ■

Yoann Desmouceaux received a Diplôme d'Ingénieur from École Polytechnique, Palaiseau, France, in 2014, an M.Sc. degree in Advanced Computing from Imperial College, London, U.K., in 2015, and a Ph.D. degree in Computer Networking from Université Paris-Saclay (France) in 2019. He is currently working as a Software Engineer at Cisco Systems. His research interests include high-performance networking, IPv6-centric protocols, load-balancing, reliable multicast and data-center optimization algorithms.

Marcel Enguehard received the M.Sc. degree jointly from École Polytechnique, Palaiseau, France, and KTH - Royal Institute of Technology, Stockholm, Sweden in 2016 and the Ph.D. degree from Telecom ParisTech in 2019. He is currently working as a Software Project Manager at Polyconseil, Paris, France. He has authored 8 journal and conference papers, and 12 pending patents. His research focused on forwarding mechanism for Information-Centric Networking and load distribution for multi-tiered compute platforms.

Thomas H. Clausen is a graduate of Aalborg University, Denmark (M.Sc., PhD – civilingeniør, cand.polyt), and a Senior Member of the IEEE. Thomas has, since 2004 been on faculty at École Polytechnique, France's leading technical and scientific university, where he holds the Cisco-endowed "Internet of Everything" academic chaire.
At École Polytechnique, Thomas leads the computer networking research group. He has developed, and coordinates, the computer networking curriculum, and co-coordinates the Masters program in "Advanced Communication Networks" (ACN). He has published more than 80 peer-reviewed academic publications (which have attracted more than 12000 citations) and has authored and edited 24 IETF Standards, has consulted for the development of IEEE 802.11s, and has contributed the routing portions of the recently ratified ITU-T G.9903 standard for G3-PLC networks – upon which, e.g., the current SmartGrid & ConnectedEnergy initiatives are built. He serves on the scientific council of ThinkSmartGrids (formerly: SmartGridsFrance).