CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
Concurrency Computat.: Pract. Exper. 2016; **00**:1–22
Published online in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/cpe

Mitigating Impact of Short-term Overload on Multi-Cloud Web Applications through Geographical Load Balancing

Chenhao Qu[1], Rodrigo N. Calheiros[2], and Rajkumar Buyya[1]

[1] *Cloud Computing and Distributed Systems (CLOUDS) Laboratory, School of Computing and Information Systems, The University of Melbourne, Australia, VIC, 3010* [2] *School of Computing, Engineering and Mathematics, Western Sydney University, Australia, NSW, 2751*

SUMMARY

Managed by an auto-scaler in the clouds, applications may still be overloaded by sudden flash crowds or resource failures as the auto-scaler takes time to make scaling decisions and provision resources. With more cloud providers building geographically dispersed data centers, applications are commonly deployed in multiple data centers to better serve customers worldwide. In this case, instead of sufficiently over-provisioning each data center to prepare for occasional overloads, it is more cost-efficient to over-provision each data center a small amount of capacity and to balance the extra load among them when resources in any data center are suddenly saturated. In this paper, we present a decentralized system that timely detects short-term overload situations and autonomously handles them using geographical load balancing and admission control in order to minimize the resulted performance degradation. Our approach also includes a new algorithm that optimally distributes the excessive load to remote data centers causing minimum increase of overall response times. We developed a prototype and evaluated it on Amazon Web Services. The results show that our approach is able to maintain acceptable QoS while greatly increase the number of requests served during overloading periods. Copyright © 2016 John Wiley & Sons, Ltd.

Received . . .

KEY WORDS: Web Application; Geographical Load Balancing; Resource Overload; Auto-Scaling; Multi-cloud; Request Forwarding

1. INTRODUCTION

Cloud computing continues to gain rapid adoption for hosting web applications. One of its appealing features is its elasticity, which allows application providers to dynamically expand or shrink the amount of resources provisioned to their applications using auto-scaling. However, detecting workload changes and provisioning enough resources in the cloud still demand considerable time. Mao and Humphrey [1] conducted an experimental study on the VM startup time of various types of instances in Amazon AWS, Microsoft Azure, and Rackspace. They found that the booting time ranges from about 50s to more than 900s depending on the sizes, cost models, and operating systems. This delay in resource provisioning will result in performance degradation and even unavailability of service during this period.

In web applications, it is common to observe rapid surges in requests once in a while. This situation is called flash crowd, and it can occur any moment with little or no warning. The cloud auto-scaler, in these cases, cannot timely provision enough resources to deal with these situations. In addition, sudden failures of either software or virtual machines can also lead to application overload.

[*] Correspondence to: Chenhao Qu, Cloud Computing and Distributed Systems (CLOUDS) Laboratory, School of Computing and Information Systems, The University of Melbourne, Australia. Email: chenhao.qu@unimelb.edu.au

Application providers now deploy web applications on low cost but unreliable cloud resources, such as spot instances [2, 3], which makes the applications more prone to resource failures.Therefore, solely relying on auto-scaling is not enough to ensure high performance all the time and certain level of over-provisioning is necessary for production environments in preparation for these events.

Cloud providers have established their data centers all over the world, which enables their customers to deploy their applications in multiple geographically dispersed regions to better serve the worldwide population. As multi-cloud deployment is becoming increasingly popular, we argue that in this type of deployment, when failures happen, or a flash crowd arrives at a data center, it is better to utilize the unused capacities already provisioned in other data centers[†] to process as many exceeding requests as possible through geographical load balancing, instead of processing all the requests locally and degrading the performance of all the clients served by the regional data center, or rejecting the exceeding requests directly. This approach is viable as failures are unlikely to happen simultaneously in multiple data centers and flash crowds also seldom take place on a global scale at the same time due to culture and time differences.

A common approach to implement geographical load balancing is through DNS resolution. However, it takes time to populate the DNS settings across layered DNS servers, which makes it impossible to react to overload situations timely. Furthermore, it is also difficult to accurately control the load directed to each data center using this technique. Another popular way is to utilize a centralized load balancer to distribute load among data centers. Though it allows fine-grained control over traffic, it introduces extra latencies to all the requests, which reduces the benefit of deploying the application in multiple clouds.

In this paper, we present a system that supplements and enhances state-of-the-art auto-scalers for applications deployed across multiple data centers. It follows the monitor-analyse-plan-execute (MAPE) loop architecture commonly adopted by other Cloud-based systems [4, 5, 6]. It aims to quickly detect and adapt to short-term overloads caused by resource failures and flash crowds in each data center through geographical load balancing and admission control before the auto-scaler finishes provisioning new resources. Different from previous geographical load balancing solutions, our system relies on decentralized agents deployed in each data center to implement fast and accurate geographical load balancing. During the overload situations, the agent deployed in the overloaded data center temporarily forwards certain amount of excessive requests to other data centers to keep the predefined SLA satisfied. Within our approach lies our proposed overload handling algorithm that optimally distributes excessive load among data centers have available capacities causing minimum overall increase of latencies. In this way, our approach only incurs little performance overhead to the forwarded requests and requests served by the receiving data centers during the short overloading periods and thus, preserves the benefit of a multi-cloud deployment. We implemented a prototype and evaluated it on Amazon Web Services, who offer IaaS infrastructure in multiple geographically dispersed regions. Results show that our approach can timely detect short-term overload events and effectively improve the application performance during resource contention periods.

The **main contributions** of the paper are:

- a system that supplements and enhances state-of-the-art auto-scalers in handling short-term overload situations for multi-cloud applications;
- a decentralized load balancing framework across multiple clouds that detects and handles short-term overload events;
- an optimal load distribution algorithm and admission control mechanism that quickly adapts to the overload situations; and
- a prototype implementation of the proposed system evaluated in Amazon's IaaS infrastructure across US, Europe, and Asia.

[†] Because load is balanced among all the servers provisioned in a data center, the unused capacities come from the spare processing capabilities in each server instead of from completely idle servers that are standing by.

DOI: 10.1002/cpe

The remainder of the paper is organized as follows. In Section 2, we illustrate the use case scenarios of our system. We then describe the multi-cloud deployment model and some assumptions in Section 3. We explain our proposed system and its implementation in detail in Section 4. Finally, we compare our approach with related works and conclude the paper in Section 7.

2. USE CASE SCENARIOS

Our system aims to mitigate performance degradation caused by short-term overloads that cannot be timely addressed by the cloud auto-scalers. It is not designed to replace state-of-the-art auto-scalers; instead, it is complementary to them and it should work cooperatively with them.

2.1. Resource Failures

Resource failures can happen at any time and the failed resources will become inaccessible immediately, which leaves the auto-scaler no time to provision new resources without causing performance degradation during the provision time, if the amount of resource loss is beyond the locally unused capacity. Such failures can frequently happen and in large scale if the application is deployed on unreliable resources, such as spot instances [2], which makes things worse. Spot instances are resources sold by the cloud providers through an auction-like mechanism. They are significantly cheaper than the corresponding on demand instances, but they will be terminated by the provider when the market price exceeds the user bid, thus, causing failures. Besides infrastructure level, failures can also happen in software level, including the operating system, the application server, and the application itself. Our system utilizes periodic health checks to detect failures, which is agnostic to the underlying failure types.

If any failure is detected, the local agent of our system and the auto-scaler will intervene at the same time. During the failures, the agent temporarily forwards some requests to other data centers and enforces admission control if necessary to protect the application from crashing and maintain acceptable performance; meanwhile, the auto-scaler restarts the faulty resources or provisions new resources. When the provision process completes, and there are enough local resources, the agent then stops geographical load balancing and admission control.

2.2. Flash Crowds

Flash crowds might arrive anytime at any data center. They are difficult to be managed by the auto-scaler alone due to their unpredictability and bursty nature. In the case of a flash crowd, widely used commercial auto-scalers, such as Amazon Auto-scaling Service[‡], launch new VMs only after the application has experienced high load for a consecutive period set by the user, instead of provisioning new resources right after the detection of application overload. This mechanism is useful to reduce resource wastage and prevent scaling oscillations [7], as when the provision completes, the flash crowd might have already passed. Our system can be an ideal partner of these commercial auto-scalers as it can help to reduce resource contentions not only during provision times but during waiting times as well.

3. DEPLOYMENT MODEL AND APPLICATION REQUIREMENTS

3.1. Deployment Model

We assume the target application of our system, including each of its composing services/components, is deployed in geographically dispersed data centers. Furthermore, the application instance deployed in a data center should be able to communicate with the instances

[‡] http://aws.amazon.com/autoscaling/

Copyright © 2016 John Wiley & Sons, Ltd.
Prepared using cpeauth.cls

DOI: 10.1002/cpe

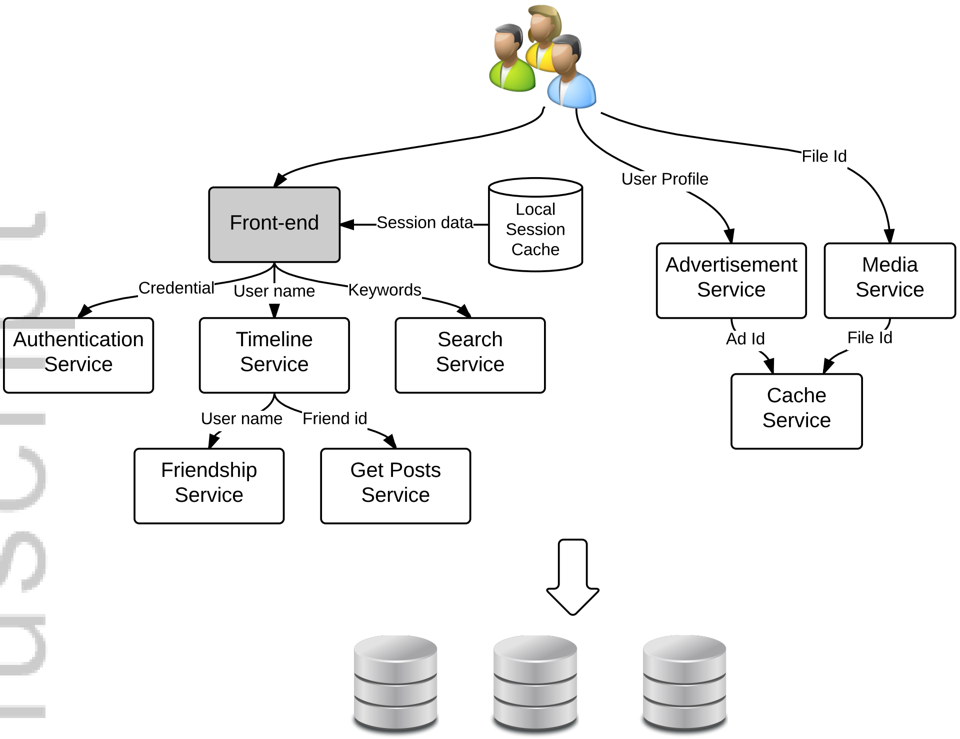


Figure 1. A service-oriented social network application

located in other data centers though network for request forwarding purpose, which will be explained afterward.

3.2. Application Requirements

Our approach requires requests can be processed by application replicas deployed in other data centers, which involves two important factors: session continuity and data locality.

Session continuity means that the end user should be able to seamlessly interact with the application without losing any internal state even if different data centers process his requests. Stateless applications, such as public knowledge services like Wikipedia and search engine, implicitly satisfy this requirement. For stateful applications, there are ways to make them geographically stateless, such as pushing the states into client side, and session replication across sites[§]. Besides, applications can be divided into multiple stateful and stateless services following the Service-oriented Architecture (SOA) [8]. Figure 1 shows an example of a service-oriented social network application. The application first loads the session data of the user and then relays the corresponding data to the underlying stateless services for further processing. The local data center should always serve the requests for the stateful services. Thus, stateful services cannot be managed by our approach. For these services, they should be sufficiently over-provisioned in preparation for failures and flash crowds. While the majority of the underlying stateless services, which consume the most resources, are eligible to be administrated by our approach.

The second requirement is that persistent data should also be replicated across multiple data centers either partially or entirely as requests can be only forwarded to data centers that have the essential data available. Fortunately, data replication is commonly adopted by multi-cloud applications nowadays [9, 10], and thus, enhances applicability of our approach in many scenarios.

[§] http://www.alachisoft.com/ncache/multi-site-dotnet-session-sharing.html

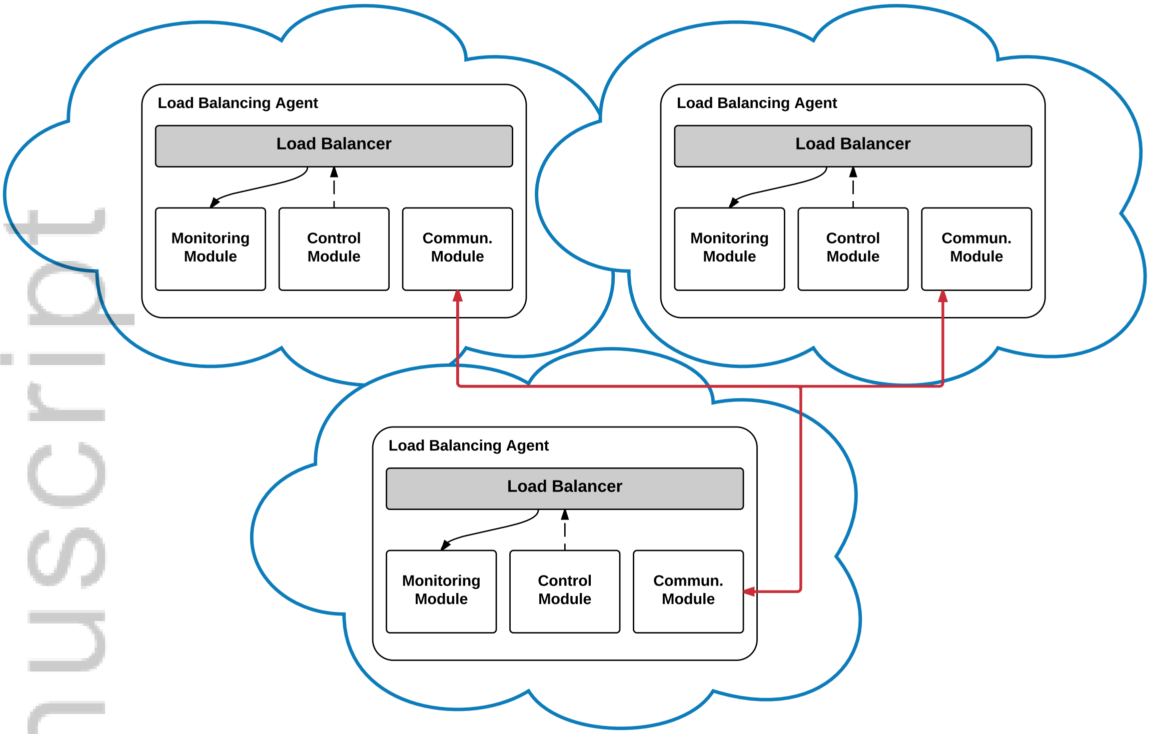


Figure 2. Proposed architecture with three data centers involved

4. THE PROPOSED APPROACH

4.1. Architecture

Our approach employs a decentralized architecture as shown in Figure 2. The load balancing agent is co-located with the application/service load balancer in the corresponding data center to realize fast detection of overload events and perform quick adaptations. They are fully connected with each other through network to work cooperatively. Each agent comprises the monitoring module which constantly monitors the incoming requests and the status of the available resources to detect application overload, the communication module which is in charge of broadcasting its status to other agents and receiving other agents' statuses, and the control module that quickly adapts the application/service to the detected overload events.

4.2. Overload Detection

Choosing the right performance indicator is critical for the detection algorithm. There are several potential indicators we can utilize, such as request rate (request arrival per second), session creation rate (newly created sessions per second), and average response time. In some cases, some indicators cannot truthfully reflect the actual load. For example, the downstream service in an SOA application is usually called by its upstream service using a persistent session, therefore, in this case, session creation rate is not suitable to serve as the overload indicator. In our prototype implementation, we adopt request rate as the indicator since it is more general purpose than other indicators. In addition to the incoming load, the agent also needs to monitor the availability of resources. This can be carried out by periodic health checks.

Another important task is to determine the monitoring frequency. A small monitoring interval enables our approach to timely detect the changes even in the spikiest workload. However, frequent monitoring not only consumes a lot of physical resources, but also causes more false positives. Such behavior can be observed in our experiments results shown in Figure 10 and Figure 13 with a small

DOI: 10.1002/cpe

amount of requests being rejected by the admission control approach in the one server down and 245 reqs/s flash crowd settings. Because by using our approach, false positives only result in some of the requests being unnecessarily forwarded to other data centers, instead of being rejected, we choose to favor sensitivity over accuracy and employ a high monitoring rate of every 2 seconds.

With the indicator chosen, we developed a detection mechanism. In the first stage, we profile the machine to determine averagely how many requests per second (c) it can safely handle under the predefined SLA, such as 90% of requests should be served within 1 second. Suppose the requests arrival is a Poisson process and the data center has n machines available for serving requests, the datacenter is considered overloaded when the incoming workload λ is larger than $n * c + \sqrt{n * c}$ or between $n * c$ and $n * c + \sqrt{n * c}$ for consecutively a few monitoring intervals. The rationality under this approach is that the probability of the result of a Poisson process deviates beyond its standard deviation $\sqrt{\lambda}$ is relatively small, and overload situations often cause much higher load. In this way, we can reduce the amount of false positives caused by highly fluctuant workloads. Note that our framework can employ other detection algorithms as well, such as the one proposed by Kamra et al. [11].

4.3. Overload Handling Algorithm

When overload is detected, the system needs to distribute as many excessive requests as possible to other data centers that have available capacities. Though the response times of the requests that are initially served by the receiving data centers will be negatively affected by the forwarded requests, the SLAs of the receiving data centers can still be ensured as the amount of requests forwarded by our approach should never exceed the remaining capacities available. Our software framework is modularized and can utilize various request distribution algorithms, such as random and greedy algorithms. To get ***optimal*** overall performance, we propose a new distribution algorithm that minimizes the increase of latencies caused by request forwarding. We define the observed latency increase as follows:

$$I(X) = \sum_{i=0}^{n} R_i^{wf}(x_i) - \sum_{i=0}^{n} R_i^{bf} + \sum_{i=0}^{n} F(x_i) \tag{1}$$

where x_i is the average amount of requests per second forwarded to the ith data center, $R_i^{wf}(x_i)$ is the latencies observed by all the users originally served by the ith data center with extra x_i requests per second forwarded to it, R_i^{bf} is the total latencies experienced by all the users originally served by the ith data center without extra requests forwarded to it, and $F(x_i)$ is the total latencies felt by the users whose requests are forwarded to the ith data center. Since R_i^{bf} is constant, the latency increase minimization problem is equivalent to:

$$\begin{aligned} \underset{x}{\text{minimize}} \quad & \sum_{i=0}^{n} R_i^{wf}(x_i) + \sum_{i=0}^{n} F(x_i) \\ \text{subject to} \quad & \sum x_i = N \\ & 0 \leq x_i < S_i \end{aligned} \tag{2}$$

where N is the average amount of excessive requests per second needs to be distributed, and S_i is the maximum average amount of extra load per second the ith data center can handle without violating the local SLA.

We model each remote data center as a M/M/1 - processing sharing queue. According to Little's Law, the average response time of the ith data center can be represented as:

$$r_i = \frac{1}{\mu_i - \lambda_i} \tag{3}$$

where μ_i is the service rates of the ith data center, and λ_i is the average incoming load to the $i$the data center. Based on Equation 3, $R_i^{wf}(x_i)$ and $F(x_i)$ can be respectively modeled as:

Algorithm 1: Overload Handling Algorithm

```
Input: $r$ : the average excessive requests per second
Input: $S_i$ : the maximum average requests per second the $i$th data center can handle without
       violating the local SLA
Input: $L_i$ : the latency to the $i$th data center from the forwarding data center
1 if $r > \sum_i^n S_i$ then
2     reject $r - \sum_i^n S_i$ requests per second;
3     $r \leftarrow \sum_i^n S_i$;
4 end
5 obtain the request distribution plan $X$ by solving the problem defined by Equation 7;
6 respectively forward $x_i$ requests per second to the $i$th data center;
7 Return;
```

$$R_i^{wf}(x_i) = \frac{\lambda_i}{\mu_i - \lambda_i - x_i} \tag{4}$$

$$F(x_i) = L_i x_i + \frac{x_i}{\mu_i - \lambda_i - x_i} \tag{5}$$

where L_i is the RTT latency between the ith data center and the data center sending the forwarding requests. The optimization function then can be formatted as:

$$\underset{x}{\text{minimize}} \quad \sum_{i=0}^{n} \left(\frac{\mu_i}{\mu_i - \lambda_i - x_i} + L_i x_i - 1 \right) \tag{6}$$

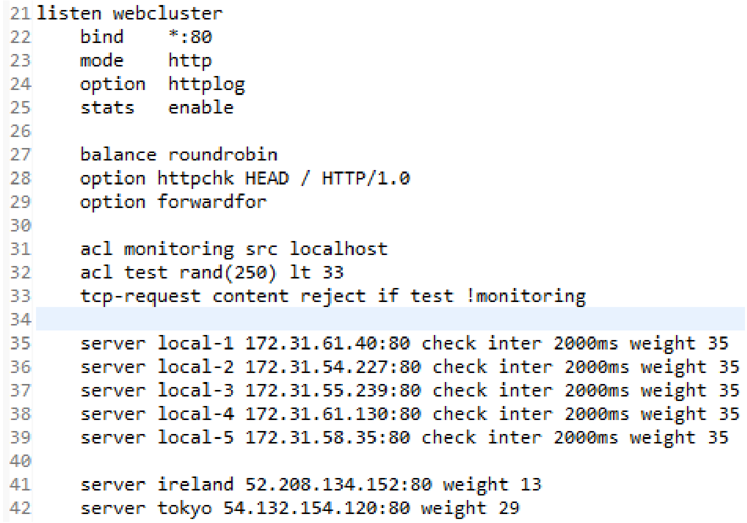
By removing the constant -1, the optimization problem is transformed to:

$$\begin{aligned} \underset{x}{\text{minimize}} \quad & \sum_{i=0}^{n} \left(\frac{\mu_i}{\mu_i - \lambda_i - x_i} + L_i x_i \right) \\ \text{subject to} \quad & \sum x_i = N \\ & 0 \leq x_i < S_i \end{aligned} \tag{7}$$

Since S_i is smaller than $\mu_i - x_i$, the optimization function is strictly convex in the feasible space. Besides, the constraints are also convex. Therefore, the optimization problem is strictly convex and can be efficiently and optimally solved using existing convex solvers. According to our experiment results and analysis presented in Section 5.8, this problem can be solved quickly enough within milliseconds scale to support instant online decisions. Besides, as long as $N < \sum_i^n S_i$, the feasible set is non-empty, and thus, the problem has a unique global optimal solution since it is strictly convex.

The overall flow of the overload handling algorithm is abstracted in Algorithm 1. It first checks whether the aggregated available capacities of remote data centers can cater all the excessive load (Line 1) and rejects the exceeding requests (Line 2) accordingly. After that, it solves the optimization problem defined in Equation 7 and distributes excessive requests among remote data centers (Line 5 and Line 6).

Though rare in probability, simultaneous overloads in multiple data centers can happen. In this case, if the remaining capacities in the rest data centers still can cope all the excessive requests, our approach will serve all the requests, as in the worst case, a request would only be forwarded multiple times due to staled statuses of remote data centers and finally be served by a data center with available capacity. Otherwise, all the data centers will be saturated and our approach will apply admission control to reject the excessive requests in the overloaded data centers.



```
listen webcluster
    bind    *:80
    mode    http
    option  httplog
    stats   enable

    balance roundrobin
    option httpchk HEAD / HTTP/1.0
    option forwardfor

    acl monitoring src localhost
    acl test rand(250) lt 33
    tcp-request content reject if test !monitoring

    server local-1 172.31.61.40:80 check inter 2000ms weight 35
    server local-2 172.31.54.227:80 check inter 2000ms weight 35
    server local-3 172.31.55.239:80 check inter 2000ms weight 35
    server local-4 172.31.61.130:80 check inter 2000ms weight 35
    server local-5 172.31.58.35:80 check inter 2000ms weight 35

    server ireland 52.208.134.152:80 weight 13
    server tokyo 54.132.154.120:80 weight 29
```

Figure 3. An example of the dynamically generated HAProxy configuration

We treat the network latencies as constants in the optimization problem. However, they often vary dynamically during runtime. Therefore, they need to be dynamically monitored and updated. Sometimes, some data centers can even become disconnected from the network. In this case, they are temporarily removed from the candidate set for request forwarding during the downtime.

4.4. Communication Protocol

Load balancing agents in each participating data center communicate among themselves to update their real-time statuses, including their service rate, current load, and available capacity for offloading. In our prototype, this is implemented through a broadcasting protocol. It makes each agent broadcast its status when its service rate has changed, its load has varied beyond a predefined percentage, or some time has elapsed since the last broadcast. Compared to a strategy that broadcasts only in a particular time interval, it not only confines the data error but also makes the system more robust when overload events happen simultaneously in multiple data centers, though such case is expected to be rare. Considering the situation that one agent detects the local application is in an overload condition, according to the protocol, it will immediately inform other agents that there is no available unused capacity offered by it, instead of waiting until the scheduled broadcasting time. This method prevents the agent in another data center happens to be overloaded as well to forward requests to the overloaded data center, leading it to more severe resource congestion or triggering cascading request forwarding which will incur unnecessary extra latencies.

It is inevitable that sometimes data are not updated timely and causes requests being forwarded to an already saturated data center. In this case, instead of directly rejecting excessive requests in the receiving data center, a cascading request forwarding will be triggered, if it believes there are extra capacities available in other data centers. Because the data center does not distinguish whether a request is originally submitted to it or is forwarded to it by another data center, the forwarded requests this time are formed by a mixture of requests submitted originally to it and requests that have been forwarded once. However, because it is unlikely that more than one data center fall into overloaded situations nearly at the same time, such scenarios will cause limited impact.

4.5. Prototype Implementation and Deployment

Since target of the proposed approach is to detect and handle application overload as soon as possible, it is preferable to develop it as a part of the load balancer so that the it can react instantaneously after the detection of the overload events. However, state-of-the-art load balancers,

DOI: 10.1002/cpe

such as HAProxy 1.6[¶], do not support to program such complex configuration. Therefore, we implemented the agent as a separate program. Fortunately, as some of the load balancers already have built-in monitoring and health check tools, we still can utilize them to ease the implementation and deployment of our approach.

The implementation follows the architecture shown in Figure 2. In the implementation, we use HAProxy 1.6 as the load balancer and rely on it to monitor the performance indicators and check the machines' health. The agent is written as a separate Java application. Its monitoring module periodically fetches the monitored information through HAProxy's stats console in CSV format. Then it extracts the required performance indicators and health statuses of the attached servers and passes them to the overload detector. The overload detector then uses the overload detection algorithm to judge whether the system is overloaded. In the case that application overload is detected, the control module configures the load balancer to adapt to the load based on the proposed overload handling algorithm. The Java agent program should be collocated with the HAProxy load balancer to minimize network latency.

Both request forwarding and admission control are implemented by dynamically changing the configuration of the HAProxy load balancer. In detail, the control module dynamically generates a new configuration file for the HAProxy when it is necessary to perform changes during runtime. The configuration change is performed through a script which automatically reloads the new configuration to the running HAProxy process.

The request forwarding mechanism is implemented by dynamically adding the IP addresses of the load balancers located in the remote data centers to the local load balancer's configuration file as normal servers. The number of forwarded requests is dynamically adjusted by assigning relative weights to the servers and remote data centers using the weighted round robin algorithm supported by HAProxy. The relative weights are obtained through solving the load distribution problem defined in Equation 7 using the primal-dual interior-point method built in the JOptimizer[‖] solver.

We take advantage of the Access Control List (ACL) mechanism in HAProxy to implement admission control. It is traditionally used to define the white list and black list of IP addresses to prevent abusing. Our agent utilizes it in a different manner. We define an ACL as a Bernoulli trial instead of a fixed list. In this way, the coming request will be served if the random test result is successful; otherwise, it will be rejected by the load balancer. Note that in a production environment, instead of directly rejecting requests; a better approach is to allow the load balancer to reply a customized error page or a default page, which is already supported by HAProxy.

Figure 3 shows an example of the dynamically generated HAProxy configuration by our system. In line 31, it defines an ACL called monitoring, which is used to prevent monitoring requests issued by the Java agent co-located in the same machine to be rejected by the admission control mechanism. Lines 32 and 33 specify the admission control configuration in which the load balancer will randomly drop 33 out of 250 incoming requests. The servers starts with "local" locate in the current data center. They are health checked every 2000ms as shown in the configuration, and the load balancer talks to these servers through their internal IPs. The last two lines specify the request forwarding settings to the remote data centers located in Ireland and Tokyo. They receive forwarded requests through the public IPs of their load balancers.

The communication module and its protocol are implemented by Java socket over persistent TCP connections. Each Java agent constantly maintains a connection to all the other known agents and continually listens to the updates sent by them.

[¶] http://www.haproxy.org/

[‖] http://www.joptimizer.com/

DOI: 10.1002/cpe

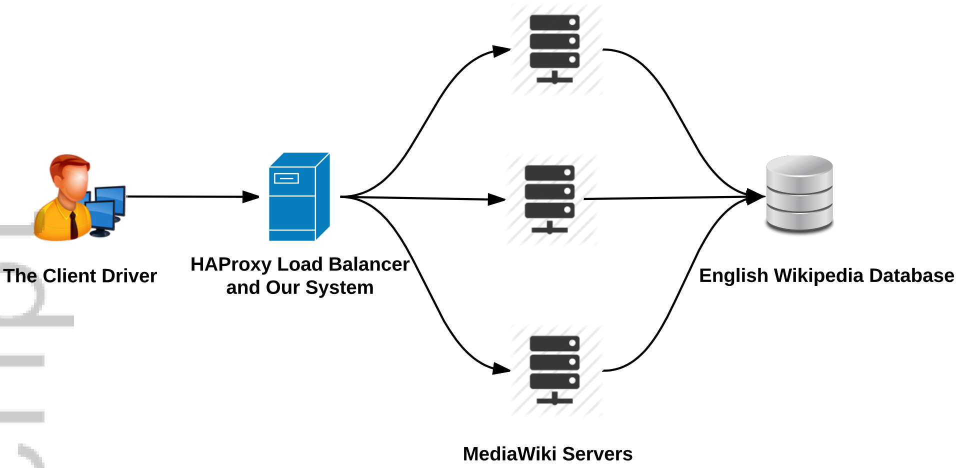

The Client Driver
HAProxy Load Balancer and Our System
English Wikipedia Database
MediaWiki Servers

Figure 4. The architecture of the benchmark application

Table I. Latencies between data centers in milliseconds

	Ireland	**Tokyo**
N. Virginia	76.3	167.2

5. PERFORMANCE EVALUATION

We evaluated our prototype implementation on Amazon Web Services IaaS infrastructure located in North Virginia, Ireland, and Tokyo. We first introduce the benchmark application and the experimental testbed. After that, we describe the workloads we used for experiments. Finally, we explain each experiment and present the results.

5.1. Benchmark Application

We used the Wikibench benchmark tool [12] as the testing application. The benchmark tool consists of three components:

- a stateless web application server installed with the MediaWiki[**] application — an open source version of Wikipedia.
- a MySQL database loaded with the English Wikipedia pages by January 2008.
- a client driver that mimics the behavior of users by sending requests to the MediaWiki server according to the given workload.

The reason we chose this benchmark is that it is stateless, which fits our prerequisite. Because our focus is on the application tier, to allow deploying a cluster of application servers, we put an HAProxy load balancer before the servers and changed the frontend configuration of the MediaWiki application servers to the IP address of the load balancer. As stated before, the load balancing agent is deployed along with the HAProxy load balancer on the same machine. Figure 4 demonstrates the architecture of the benchmark application.

5.2. Experimental Testbed

We set up the experimental testbed in three data centers owned by Amazon Web Services: US-east1 North Virginia, EU-West1 Ireland, and AP-Northeast1 Tokyo. Table I lists the RTT latencies

[**] https://www.mediawiki.org

DOI: 10.1002/cpe

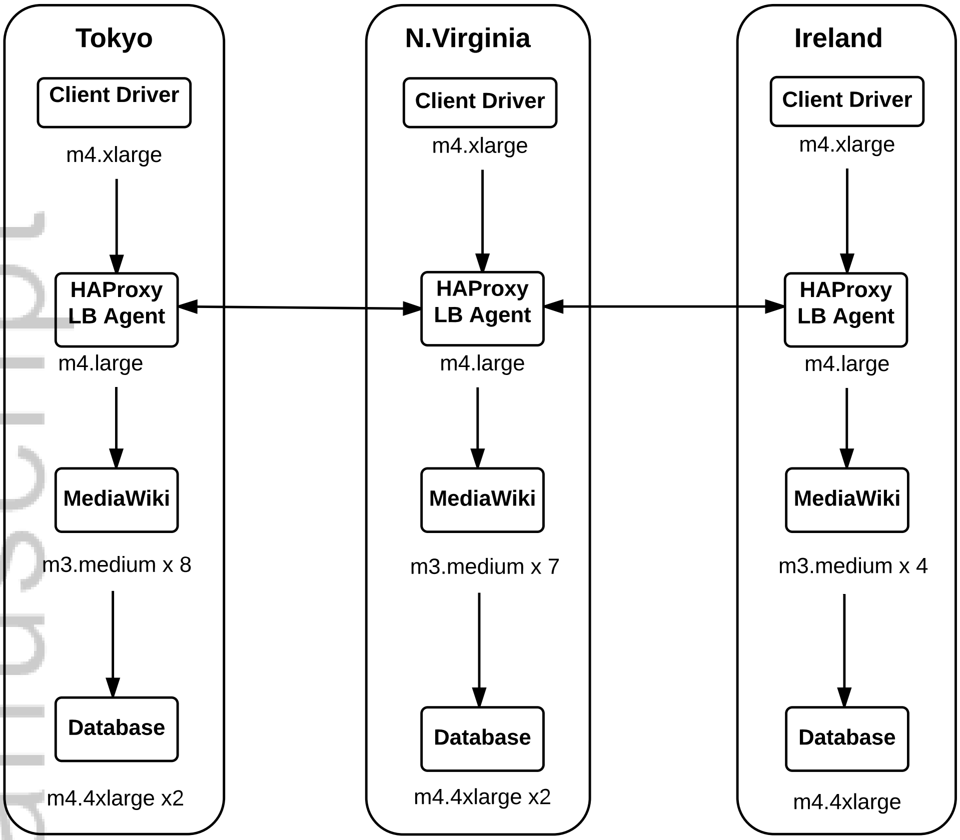


Figure 5. The experimental testbed

between North Virginia and the other two data centers tested using ping. In the experiments, we needed to emulate resource failures and flash crowds in one data center. We chose North Virginia data center as the data center that experienced failures and flash crowds. Besides serving their loads, the other two data centers accepted loads directed from North Virginia data center when overload occurred to it.

The detailed experimental testbed is illustrated in Figure 5. In each data center, we deployed one client driver using m4.xlarge instance, and one HAProxy server along with the load balancing agent running on an m4.large instance. We respectively launched eight and four m3.medium instances in Tokyo and Ireland acting as application servers. The Virginia data center, in the meantime, is equipped with seven application instances. Besides, to ensure that the data layer did not become the bottleneck, we launched different numbers of database instances to cope the load in each data center.

5.3. Workload

We used synthetic workloads generated according to real requests submitted to English-language edition of Wikipedia in September 2007 [13] to test our system. We first performed profiling tests to determine on average how many requests one m3.medium application server can handle without violating the SLA and its service rate. We defined the SLA as 90% requests should be replied within one second. We assume the workload arrival is a Poisson process and follows the exponential distribution, which is indicated by the literature [14]. This can be justified by the fact that each request is submitted independently and occurrence of each request does not affect the probability that a second request will occur. Base on this assumption, we respectively created three workloads with an average rate of 30, 35, and 40 requests/s.

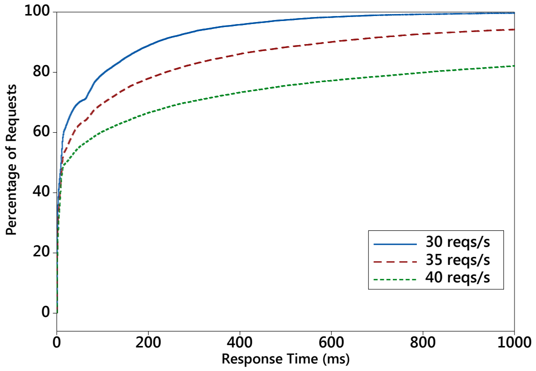


Figure 6. CDFs of the profiling tests with different average workload rates

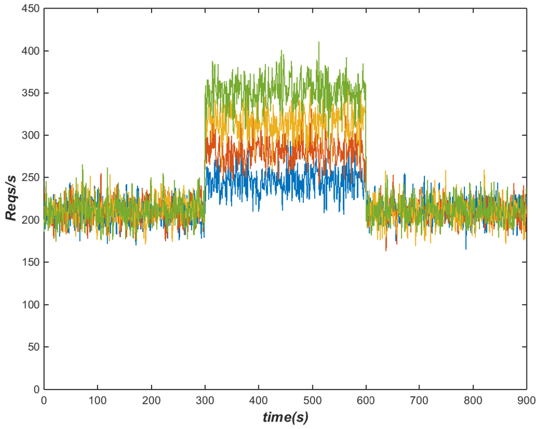
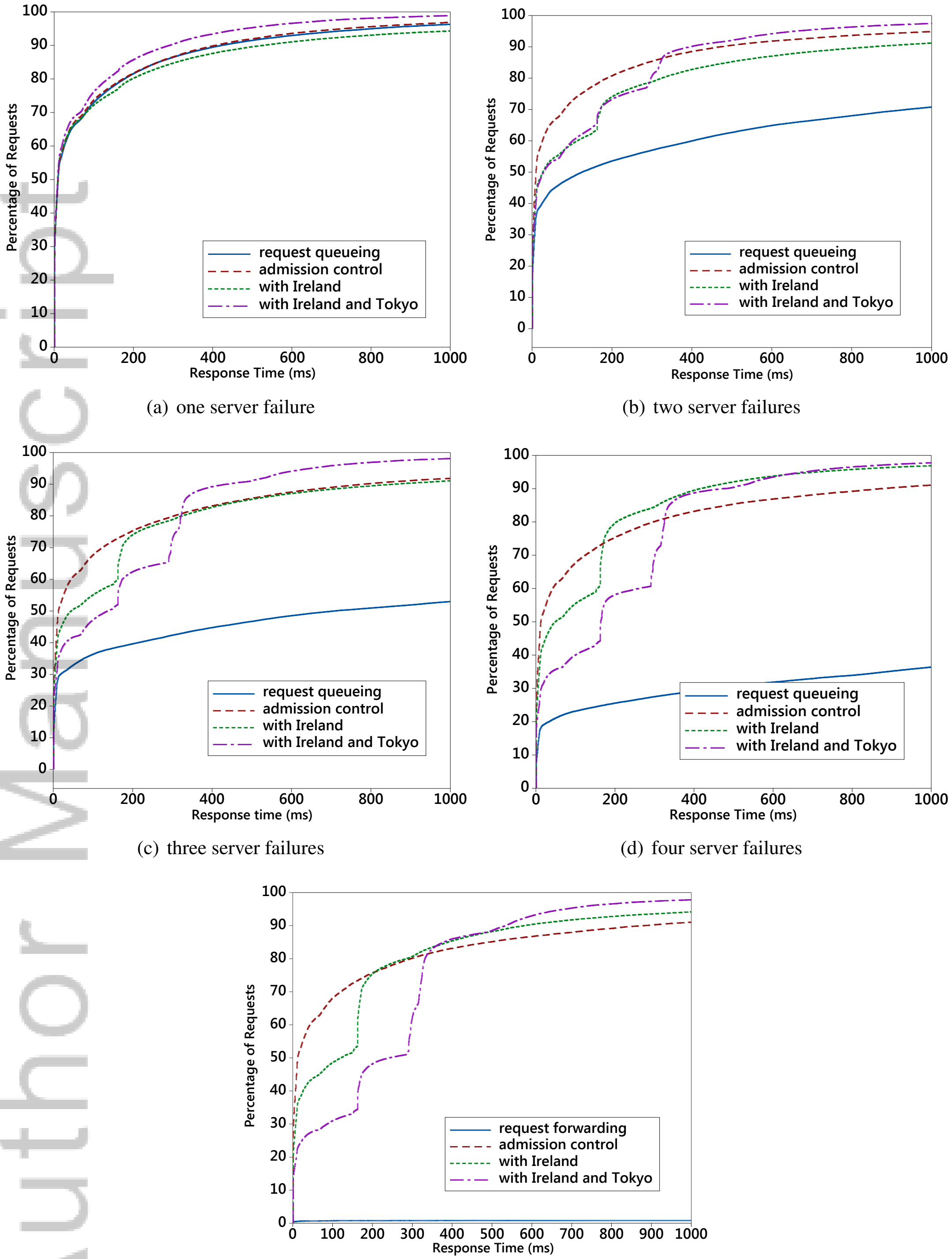


Figure 7. The workloads with flash crowds range from 117% to 183% of the normal load

Figure 6 depicts the cumulative distribution functions (CDF) of the response times obtained from the profiling tests. It shows that 35 requests/s is the largest amount of workload an m3.medium instance can handle without violating the SLA. Furthermore, we respectively calculated the service rates of the three tests according to Equation 3. Then we averaged them to obtain the estimated service rate for one instance, which is 41 requests/s.

In the following experiments, we assigned 35 requests/s unused capacities in Virginia and Ireland, and 70 requests/s unused capacities in Tokyo. According to the profiling results and the capacity setting, we generated synthetic workloads for the following experiments. We first generated the background workloads for the Ireland and Tokyo data centers respectively with average incoming rates of 105 requests/s and 210 requests/s. To test performance of the approach during resource failures in North Virginia data center, we generated a workload with average request rate of 210 requests/s. For flash crowds cases, we created four workloads with different levels of flash crowds as shown in Figure 7. Each workload experiences a total five minutes of flash crowd. The peaks of the flash crowd range from 117% to 183% of the normal load. All the workloads last for 15 minutes and suffer either server failures or flash crowds starting from the 300s time point for 300 seconds. We particularly chose the length of 300 seconds because it is the default value of the waiting time for server booting in Amazon Auto-scaling Service. In this way, we can emulate the situations that a commercial auto-scaler solely manages the application and demonstrate its resulted application performances during the overloading periods.



(a) one server failure

(b) two server failures

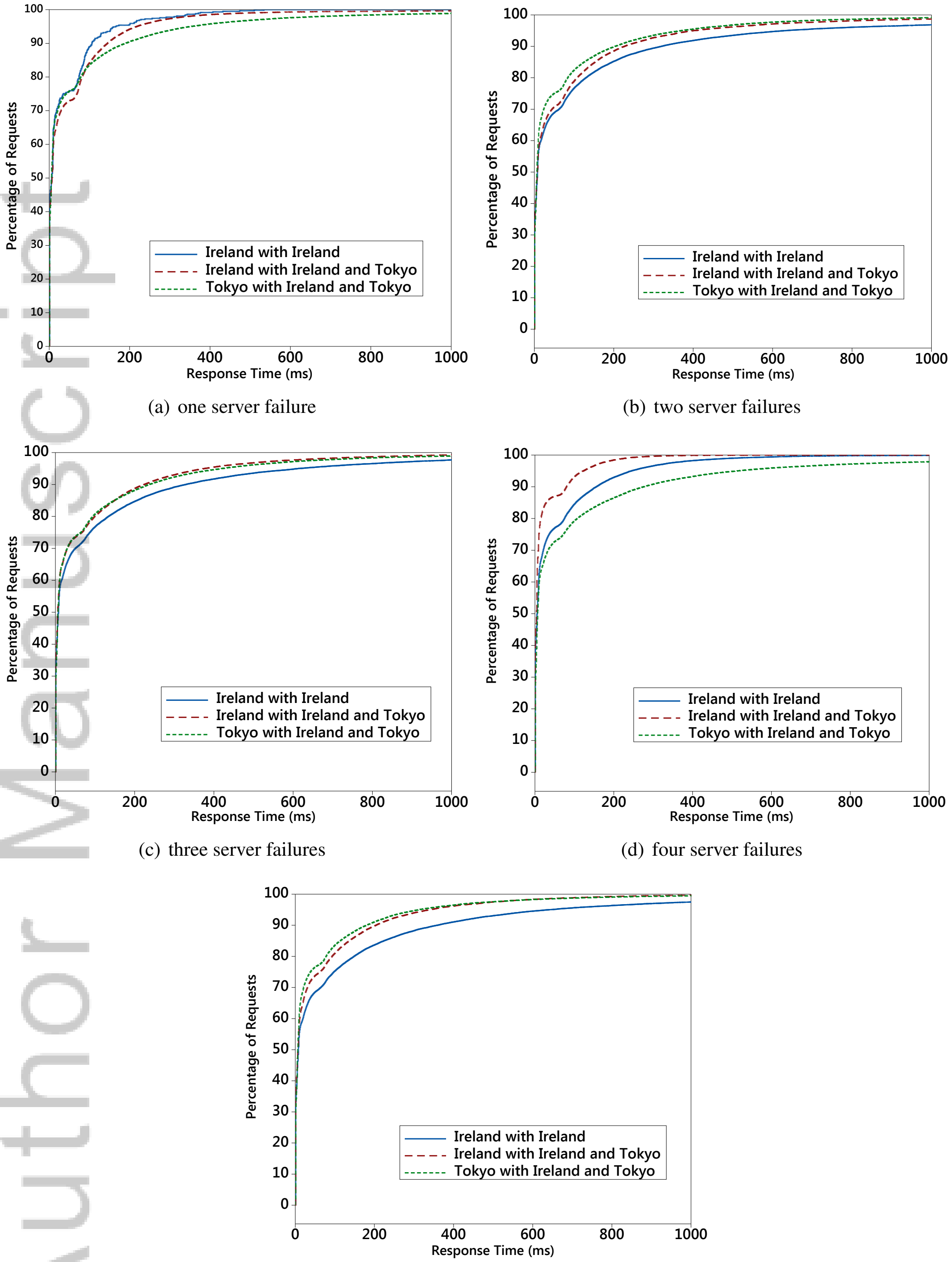
(c) three server failures

(d) four server failures

(e) five server failures

Figure 8. CDFs of the North Virginia data center during the failing periods with different approaches

Prepared using cpeauth.cls



(a) one server failure

(b) two server failures

(c) three server failures

(d) four server failures

(e) five server failures

Figure 9. CDFs of the data centers receiving forwarded requests during the failing periods

5.4. *Benchmarks*

To test the performance of our prototype, we compare our approach with the following two benchmarks:

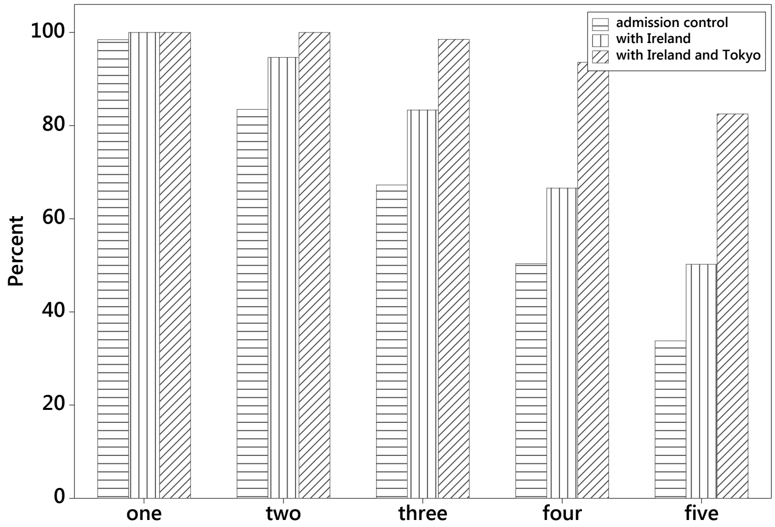


Figure 10. Percentage of admitted requests during the failing periods

- **Request queueing**: the first benchmark queues up all the requests in the local servers and imposes no admission control when the application is overloaded. It mimics the situation that the auto-scaler is booting new servers while the requests are queued up in the local servers.
- **Admission control**: the second benchmark directly imposes admission control when the application is overloaded. It emulates the case that the auto-scaler asks the load balancer to reject excessive requests while it is booting new servers.

To test the performance of the request forwarding algorithm, we compare with the following greedy algorithm:

- **Greedy**: it always forwards possibly maximum amount of excessive requests to the data center with largest available capacity one by one.

5.5. Performance under Resource Failures

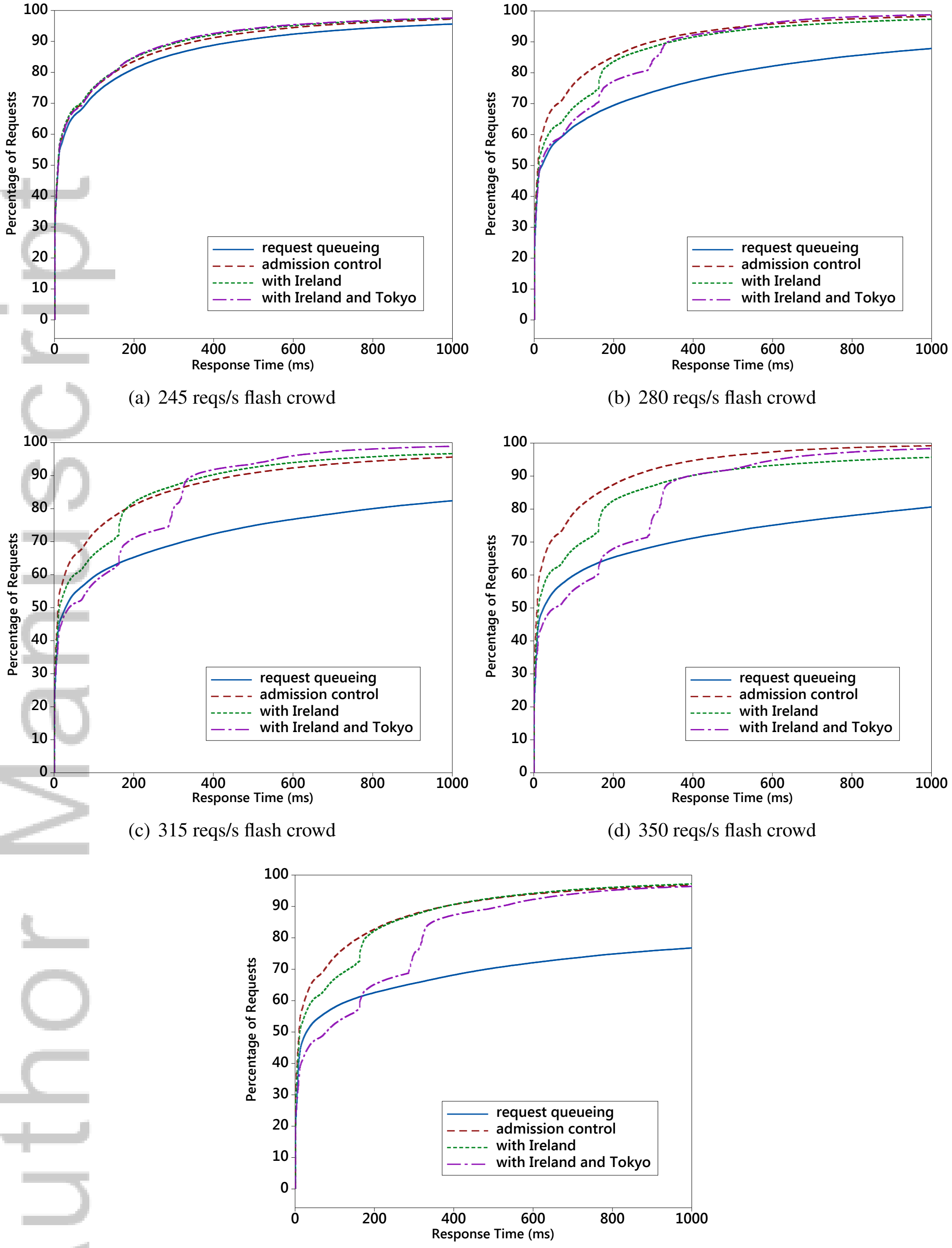
In the first set of experiments, we performed tests using our approach and the benchmarks in resource failure situations. In the experiments, we purposely removed some machines from the load balancer at 300s time point to create synthetic failures and then added them back to the load balancer after 5 minutes to mimic recovery from failures.

We ran our approach with two configurations. In the first configuration, we only utilized the unused capacity in the Ireland data center. In the second configuration, we considered unused capacity in both Ireland and Tokyo data centers. Figure 8 shows the CDFs of our approach and the two benchmark approaches respectively during the failing periods under different numbers of server failures[††].

Without proper overload handling mechanisms, the application in the North Virginia data center suffered severe performance degradation when requests were queued up and it became completely unresponsive in the case of five server failures. If we added simple admission control mechanism, the application performance was able to be maintained within acceptable level at the cost of rejecting plenty incoming requests. As shown in Figure 8, we can observe one and two shoulders in the CDFs of the two settings. This phenomenon was caused by the increased network latencies incurred by the request forwarding mechanism. In our experiments, though request forwarding increased the latencies of some requests, it did not result in the SLAs being violated. Comparing to the queuing

[††] In the reported results, rejected requests are not counted in the CDF graphs.

DOI: 10.1002/cpe



(a) 245 reqs/s flash crowd

(b) 280 reqs/s flash crowd

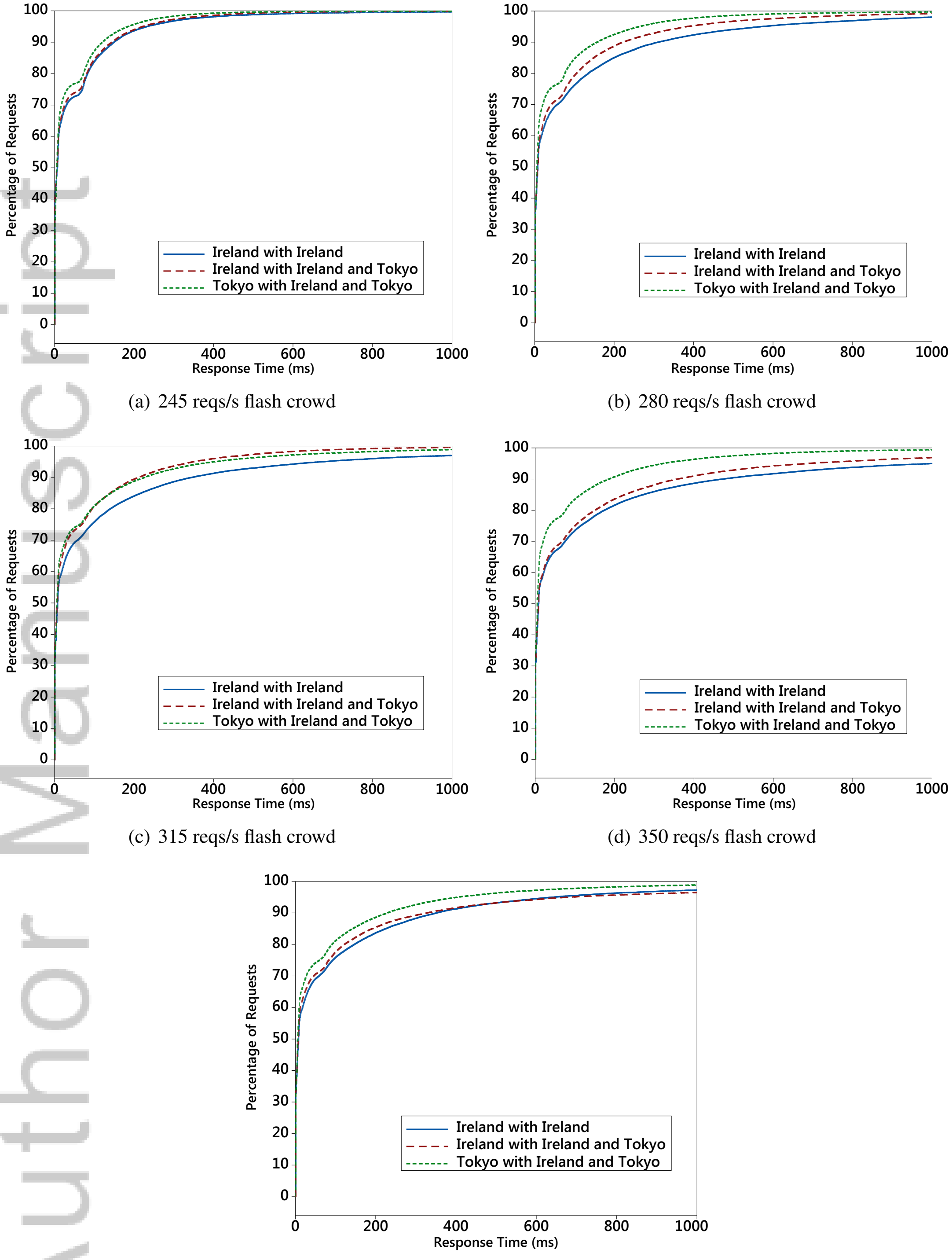
(c) 315 reqs/s flash crowd

(d) 350 reqs/s flash crowd

(e) 385 reqs/s flash crowd

Figure 11. CDFs of the North Virginia data center during the flash crowds with different approaches

delays when no overload handling mechanism is in place, the additional network latencies incurred by request forwarding are still acceptable as long as the latencies between data centers are moderate.



(a) 245 reqs/s flash crowd

(b) 280 reqs/s flash crowd

(c) 315 reqs/s flash crowd

(d) 350 reqs/s flash crowd

(e) 385 reqs/s flash crowd

Figure 12. CDFs of the data centers receiving forwarded requests during the flash crowds

Our overload handling algorithm is encouraged to forward requests to data centers that are close to the originating data center as it aims to minimize the overall latency increase.

DOI: 10.1002/cpe

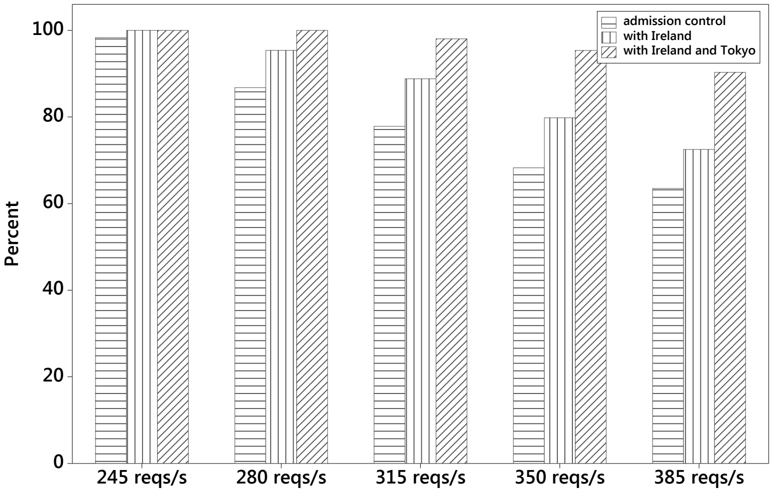


Figure 13. Percentage of admitted requests during the flash crowd periods

In addition to the failing data center, we also evaluated the performances of data centers that received the forwarded requests. Figure 9 presents the CDFs of the data centers receiving the forwarded requests when failures were happening in North Virginia data center. In all cases, the SLA was strictly honored because of the constraints on amount of forwarded requests the remote data centers can serve in the optimization problem.

Furthermore, comparing to just using admission control, our approach was able to increase the number of served requests. Figure 10 lists the proportions of the admitted requests during the failing periods in the corresponding experiments. It shows that the power of request forwarding depends on the amount of unused capacity available in other data centers. As shown in Figure 10, the configuration utilizing capacity of both Ireland and Tokyo data centers can serve more requests than the configuration that just used capacity in Ireland data center. When applying admission control only, a small portion of requests was rejected when one server was down even though the local capacity should be able to handle it, which was caused by false alarms generated by the overloading detector. While using our approach, these requests were still served by remote data centers.

5.6. *Performance under Flash Crowds*

We tested our system under flash crowd situations using the workloads depicted in Figure 7. We utilized the same settings of Section 5.5. The resulted CDFs for the baselines and our approach for North Virginia data center that was under flash crowds are delineated in Figure 11. The corresponding CDFs for the receiving data centers are presented in Figure 12. The percentages of admitted requests during the flash crowd periods are compared in Figure 13.

The experiments show similar results as the experiments in resource failure situations. Nevertheless, the impact of short-term overload on the application performance in the flash crowd experiments is not as severe because the extra load can be directed to more servers. For the same reason, more percentages of requests are served by the application.

5.7. *Performance of the Request Forwarding Algorithm*

We utilized the same testing platform to evaluate our request forwarding algorithm (specified as Min Latency Increase in the results), except we employed a workload with 70 requests/s for the North Virginia data center and considered all those requests were excessive requests need to be forwarded. In this way, we can eliminate the impact of the requests served by the North Virginia data center, which is not in our optimization target, to the results.

Figure 14 shows the results of our algorithm compared to the Greedy algorithm for the aggregated requests of the forwarded requests and the requests originally served by the remote data centers.

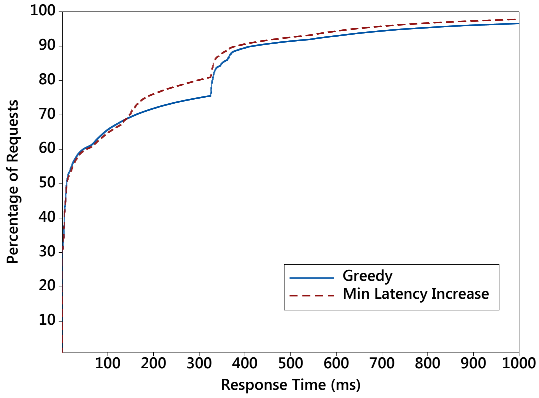


Figure 14. The performance of algorithms on the aggregated requests

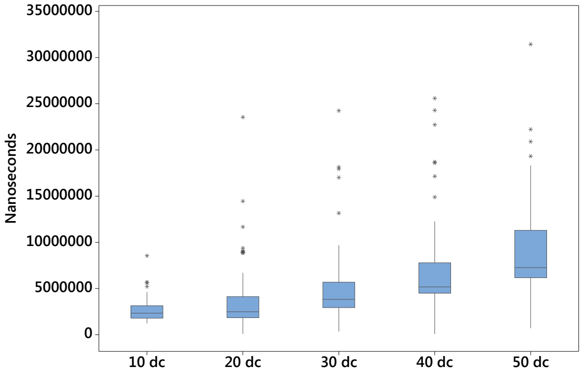


Figure 15. Measured running time for solving the workload distribution problem

Our algorithm is able to outperform the Greedy algorithm in our experimental setting, because the Greedy approach overlooks the longer network distance traveled by the forwarded requests.

5.8. *Algorithm Scalability*

Our system requires efficient solution for the workload distribution problem during runtime. In this experiment, we illustrate that this problem can be tackled very fast by a convex solver even when a large number of data centers are involved.

Since the input to the problem consists of the statuses of the data centers, the latencies to the data centers, and the amount of the excessive load, the algorithm complexity is dominated by the number of involved data centers. We randomly generated 100 problems with specific numbers of data centers ranging from 10 to 50 and measured their running time on a desktop equipped with 8 core CPUs and 16 GB of RAM. Results are presented in Figure 15. Even in the worst case with 50 data centers, the runtime was below 35ms, which is acceptable for making real-time decisions. In reality, the deployment usually involves less than a handful data centers, and the algorithm imposes negligible overhead in these cases.

6. RELATED WORK

6.1. *Overload Management*

There have been plenty of work that aims to tackle overloads caused by failures and flash crowds using cloud resources. However, all of them differ from our work in their target or approaches.

DOI: 10.1002/cpe

The approach that is commonly adopted by the industry and is intensively researched is auto-scaling [7]. It relies on dynamically provision new resources to meet the resource scarcity. Some of the auto-scaling works have been focusing on how to predict the future workloads and provision enough resources in advance [15, 16, 17, 18, 19, 20, 21, 22]. Other approaches provision resources reactively either after detecting the overload events [23, 24] or the utilization has reached certain threshold [2].

As stated before, resource failures and some flash crowds are often unpredictable, and it takes the auto-scaler considerable time [1] to provision new resources. Therefore, an auto-scaler alone cannot adequately deal with these situations. Our work can fill in this gap for multi-cloud applications by supplementing and enhancing state-of-the-art auto-scaling solutions.

In the previous work, we proposed an auto-scaler using unreliable spot instances for web applications [2]. It relies on sufficiently over-provision the application to counter the terminations of spot instances. Our new system can be an ideal partner for it. By working cooperatively with the proposed system, the spot-based auto-scaler can either reduce the amount of over-provisioned resources to reach the same level of protection or elevate the reliability of the application using the same amount of over-provisioned resources.

Cloud burst is a term often used in hybrid cloud settings referring to dynamically provisioning resources in cloud either to accelerate execution or to handle flash crowds when the local facility is saturated. Many systems have been proposed to realize this vision [25, 26, 27, 28]. In a sense, they are similar to our work as they also forward requests to remote data centers. Except that, they are more close to the auto-scaling approaches as their major focus is on how to provision and deprovision resources in clouds to meet the workload demand while our system aims to manage short-term workload distribution.

Another possible method to reduce the impact of overload is to enforce admission control. Chen et al. [29] proposed a flash crowd detection and mitigation system based on application-level measurements and admission control. In addition to protecting the application server from crashing, their target also covers protecting the network from being congested. However, in clouds, since the provider offers strong data center network and high incoming and outgoing bandwidth, this is not a concern. Different to their system, our approach uses both request forwarding and admission control as the last line of defense to protect the application from performance degradation and crashing.

Chandra and Shenoy [30] researched using dynamic resource allocation among different applications in a data center to cope with flash crowds. Different to them, our work addresses the overload management problem from an application provider's perspective instead of from an infrastructure provider's angle. Regarding applications that are composed of multiple components, Gandhi et al. [31] and Klein et al. [32] explored borrowing resources from components that have available capacity or can be terminated temporarily to support the core services under flash crowds.

6.2. *Geographical Load Balancing*

Geographical load balancing has been introduced to tackle different challenges. Commercial DNS Load Balancer, such as Amazon Route 53[‡‡] enables application providers to direct their customers to different data centers according to their location and other factors, such as energy consumption and carbon footprint [33]. However, such technique is not suitable to our needs as it takes time to populate the DNS settings across layered DNS servers and it is impossible to realize fine-grained control over the traffic flow.

Centralized geographical load balancing solutions gather all the user requests and then distribute them among data centers. They are largely developed for saving energy and carbon footprint [34, 35]. This architecture is also not applicable to reach our goal as it incurs extra network latency to every request and reduces the benefit of using a multi-cloud deployment.

Grozev and Buyya [36] proposed an approach that dispatches users to the underlying data centers at the entry point of the application framework according to the regulation requirements and the

[‡‡] https://aws.amazon.com/route53/

available resources in each data center. As the client only talks to the entry point at the start of the session, its impact on user experience is minimized compared to the centralized solutions. On the other side, this approach limits its capability to control the load on each data center accurately.

Our solution is different to the methods as mentioned earlier. We adopt a decentralized architecture composed of individual load balancing agents deployed in each participating data center to balance the extra load. The agent is only activated temporarily when overload occurs; hence, requests under normal situations can always be served by the closest data centers, and no extra network latency is introduced.

7. CONCLUSIONS AND FUTURE WORK

In this paper, we proposed a system that supplements and enhances state-of-the-art auto-scalers for applications deployed in multiple clouds. It is capable of quickly react to short-term overloads occurred to any participating data center by temporarily forwarding the excessive requests to data centers with available capacities according to our proposed optimal workload distribution algorithm, and enforcing admission control as the last line of defense. The system adopts a decentralized architecture that deploys a load balancing agent within each data center. The agent monitors the local application, detects overload events, and quickly adapts to them according to real-time resource availability in other data centers to minimize their impact on the application performance. We implemented a prototype of the system and evaluated it across AWS's US, Europe, and Asia data centers. The obtained results show that our approach can quickly detect overload situations caused by either resource failures or flash crowds and is effective in improving application performance during resource contention periods.

In the future, we plan to develop more accurate overload detectors and apply it to our approach. We will also explore how to directly return forwarded requests to users to reduce the latency cost of request forwarding. The last but not least, it is important to have a global capacity plan deciding how much resources should be over-provisioned in each data center at runtime when using our approach to minimize the overall resource cost.

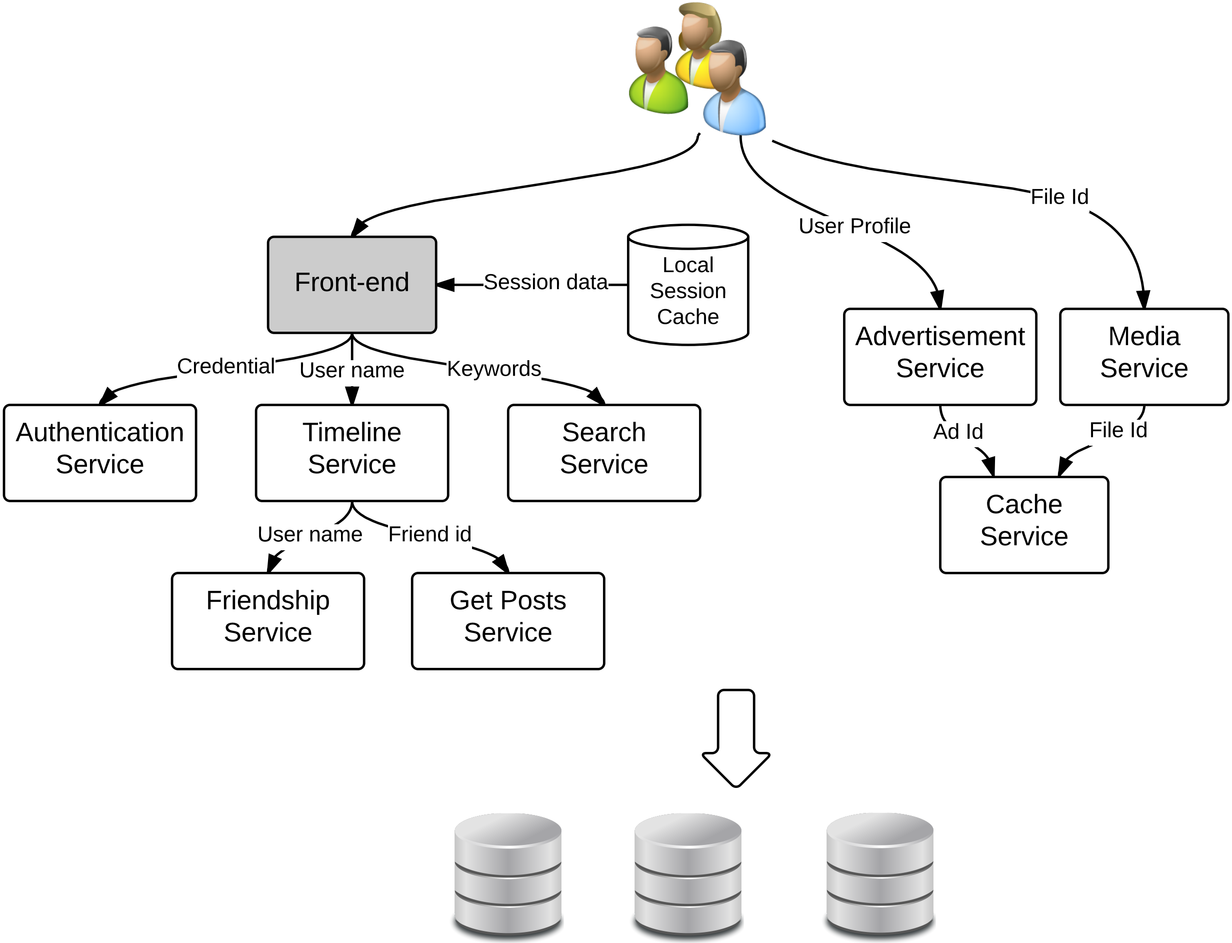
ACKNOWLEDGEMENTS

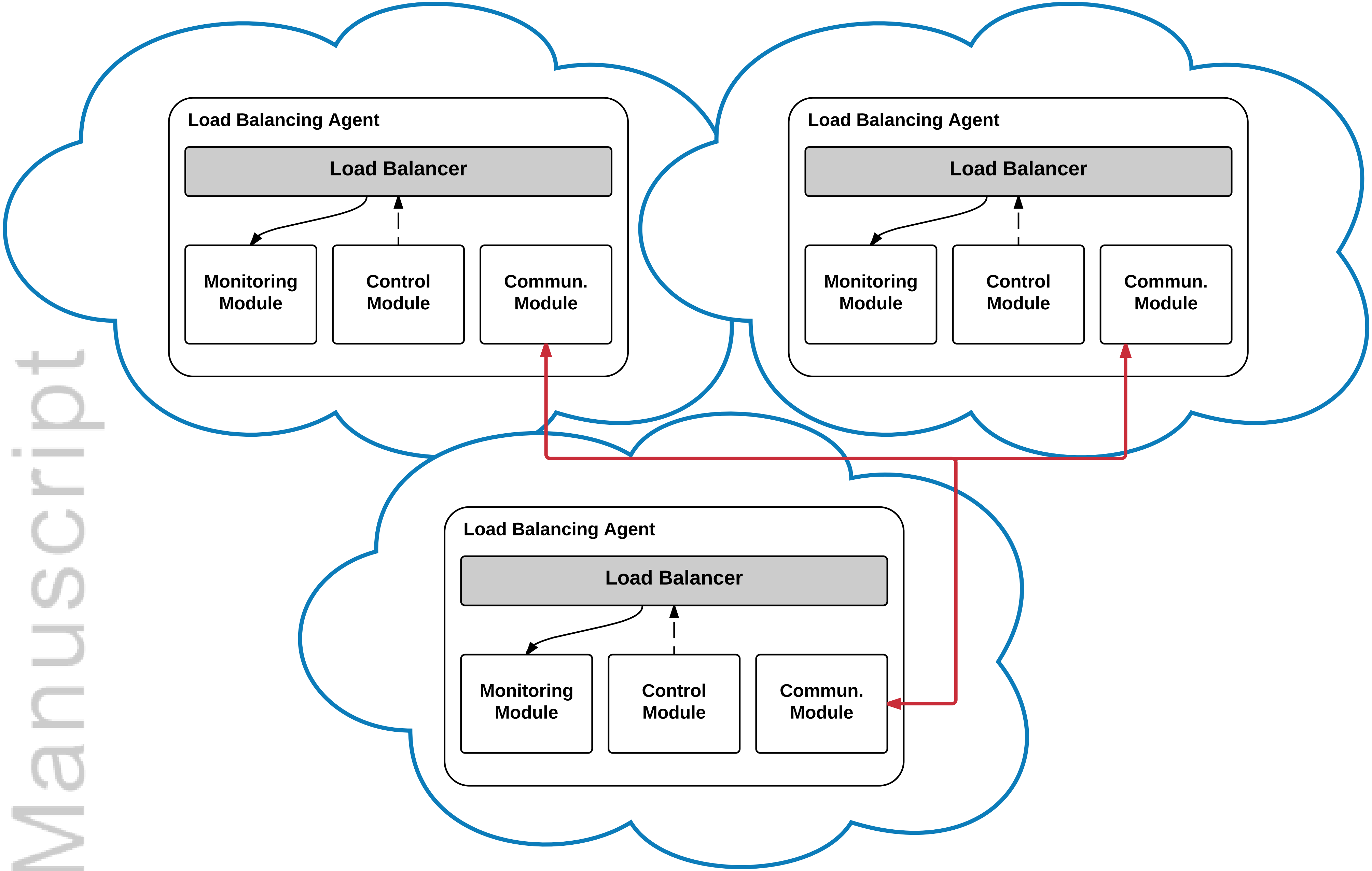
We thank Adel Nadjaran Toosi, Xunyun Liu, Yaser Mansouri, and Minxian Xu for their valuable comments and suggestions in improving the quality of the paper. This work is partially supported by an ARC Future Fellowship project.

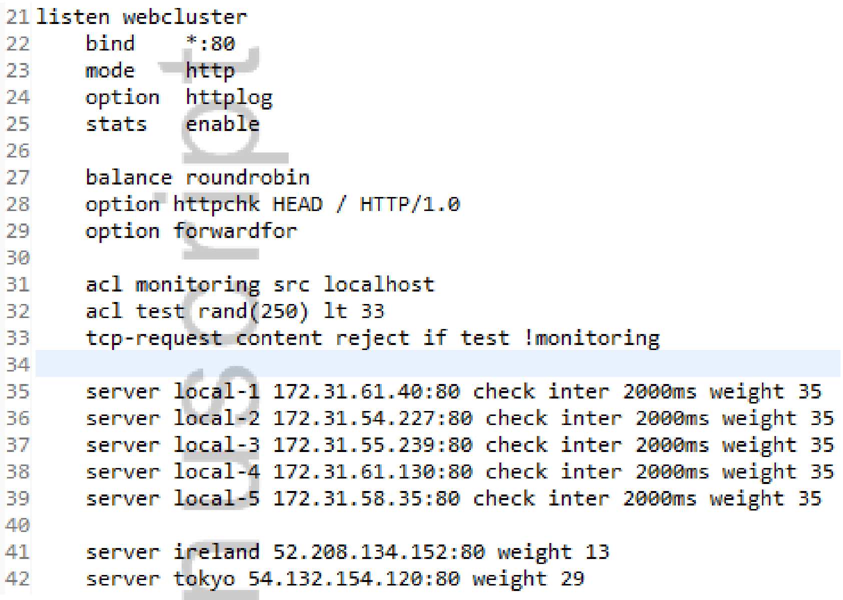
REFERENCES

1. Mao M, Humphrey M. A performance study on the vm startup time in the cloud. *Proceedings of 2012 IEEE 5th International Conference on Cloud Computing (CLOUD)*, 2012; 423–430.
2. A reliable and cost-efficient auto-scaling system for web applications using heterogeneous spot instances. *Journal of Network and Computer Applications* 2016; **65**:167 – 180.
3. Amazon. Amazon spot fleet api 2016. URL https://aws.amazon.com/blogs/aws/new-resource-oriented-bidding-for-ec2-spot-instances/.
4. Pandey S, Voorsluys W, Niu S, Khandoker A, Buyya R. An autonomic cloud environment for hosting {ECG} data analysis services. *Future Generation Computer Systems* 2012; **28**(1):147 – 154.
5. Montes J, Sánchez A, Pérez MS. Riding out the storm: How to deal with the complexity of grid and cloud management. *Journal of Grid Computing* 2012; **10**(3):349–366.
6. Zeng J, Plale B. Multi-tenant fair share in nosql data stores. *2014 IEEE International Conference on Cluster Computing (CLUSTER)*, 2014; 176–184.
7. Lorido-Botran T, Miguel-Alonso J, Lozano J. A review of auto-scaling techniques for elastic applications in cloud environments. *Journal of Grid Computing* 2014; **12**(4):559–592.
8. Wilder B. *Cloud architecture patterns: using microsoft azure.* " O'Reilly Media, Inc.", 2012.
9. DeCandia G, Hastorun D, Jampani M, Kakulapati G, Lakshman A, Pilchin A, Sivasubramanian S, Vosshall P, Vogels W. Dynamo: amazon's highly available key-value store. *ACM SIGOPS Operating Systems Review*, vol. 41, ACM, 2007; 205–220.
10. Nishtala R, Fugal H, Grimm S, Kwiatkowski M, Lee H, Li HC, McElroy R, Paleczny M, Peek D, Saab P, *et al.*. Scaling memcache at facebook. *Proceedings the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, 2013; 385–398.

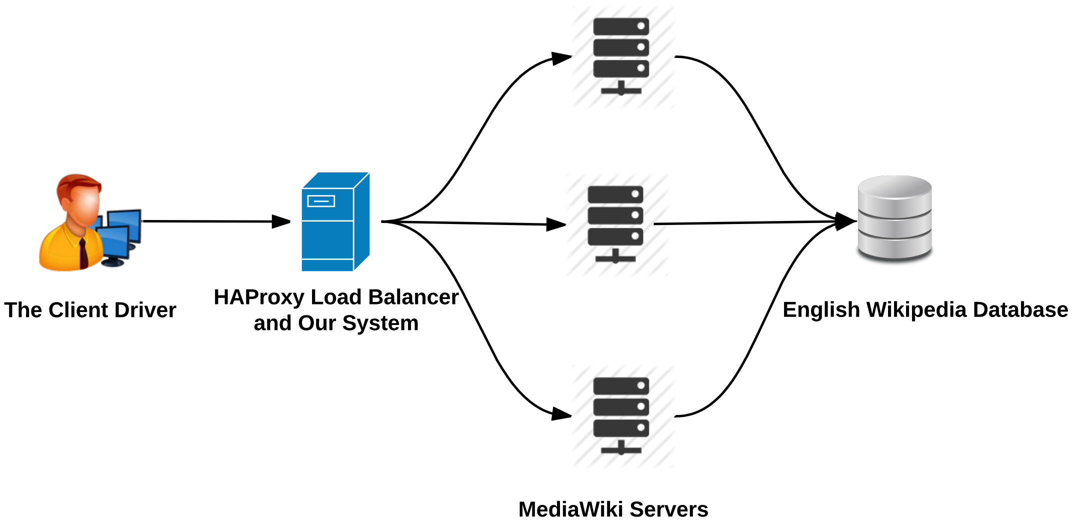
11. Kamra A, Misra V, Nahum EM. Yaksha: a self-tuning controller for managing the performance of 3-tiered web sites. *Proceedings of the Twelfth IEEE International Workshop on Quality of Service (IWQoS)*, 2004; 47–56.
12. van Baaren EJ. Wikibench: A distributed, wikipedia based web application benchmark. *Master's thesis, VU University Amsterdam* 2009; .
13. van Baaren EJ. Wikipedia access trace 2015. URL http://www.wikibench.eu/?page_id=60.
14. Chlebus E, Brazier J. Nonstationary poisson modeling of web browsing session arrivals. *Information Processing Letters* 2007; **102**(5):187 – 190.
15. Lassettre E, Coleman DW, Diao Y, Froehlich S, Hellerstein JL, Hsiung L, Mummert T, Raghavachari M, Parker G, Russell L, *et al.*. *Dynamic Surge Protection: An Approach to Handling Unexpected Workload Surges with Resource Actions that Have Lead Times*. Springer Berlin Heidelberg: Berlin, Heidelberg, 2003; 82–92.
16. Jiang J, Lu J, Zhang G, Long G. Optimal cloud resource auto-scaling for web applications. *Proceedings of 2013 13th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, IEEE, 2013; 58–65.
17. Roy N, Dubey A, Gokhale A. Efficient autoscaling in the cloud using predictive models for workload forecasting. *Proceedings of 2011 IEEE International Conference on Cloud Computing (CLOUD)*, IEEE, 2011; 500–507.
18. Jingqi Y, Chuanchang L, Yanlei S, Zexiang M, Junliang C. Workload predicting-based automatic scaling in service clouds. *Proceedings of 2013 IEEE Sixth International Conference on Cloud Computing (CLOUD)*, 2013; 810–815.
19. Islam S, Keung J, Lee K, Liu A. Empirical prediction models for adaptive resource provisioning in the cloud. *Future Generation Computer Systems* 2012; **28**(1):155–162.
20. Wei F, ZhiHui L, Jie W, ZhenYin C. Rpps: A novel resource prediction and provisioning scheme in cloud data center. *Proceedings of 2012 IEEE Ninth International Conference on Services Computing (SCC)*, 2012; 609–616.
21. Herbst NR, Huber N, Kounev S, Amrehn E. Self-adaptive workload classification and forecasting for proactive resource provisioning. *Concurrency and Computation: Practice and Experience* 2014; **26**(12):2053–2078.
22. Dutta S, Gera S, Akshat V, Viswanathan B. Smartscale: Automatic application scaling in enterprise clouds. *Proceedings of 2012 IEEE 5th International Conference on Cloud Computing (CLOUD)*, 2012; 221–228.
23. de Paula Junior U, Drummond LMA, de Oliveira D, Frota Y, Barbosa VC. Handling flash-crowd events to improve the performance of web applications. *Proceedings of the 30th Annual ACM Symposium on Applied Computing*, ACM: New York, NY, USA, 2015; 769–774.
24. Gandhi A, Dube P, Karve A, Kochut A, Zhang L. Adaptive, model-driven autoscaling for cloud applications. *Proceedings of the 11th International Conference on Autonomic Computing (ICAC 14)*, USENIX Association: Philadelphia, PA, 2014; 57–64.
25. Bjrkqvist M, Chen LY, Binder W. Cost-driven service provisioning in hybrid clouds. *Proceedings of 2012 5th IEEE International Conference on Service-Oriented Computing and Applications (SOCA)*, 2012; 1–8.
26. Javadi B, Abawajy J, Buyya R. Failure-aware resource provisioning for hybrid cloud infrastructure. *Journal of Parallel and Distributed Computing* 2012; **72**(10):1318–1331.
27. Yipei N, Bin L, Fangming L, Jiangchuan L, Bo L. When hybrid cloud meets flash crowd: Towards cost-effective service provisioning. *Proceedings of 2015 IEEE Conference on Computer Communications (INFOCOM)*, 2015; 1044–1052.
28. Zhang H, Jiang G, Yoshihira K, Chen H, Saxena A. Intelligent workload factoring for a hybrid cloud computing model. *Proceedings of 2009 World Conference on Services*, 2009; 701–708.
29. Chen X, Heidemann J. Flash crowd mitigation via adaptive admission control based on application-level observations. *ACM Transactions on Internet Technology* Aug 2005; **5**(3):532–569.
30. Chandra A, Shenoy P. Effectiveness of dynamic resource allocation for handling internet flash crowds. *TR03-37, Department of Computer Science, University of Massachusetts, USA* 2003; .
31. Gandhi A, Zhu T, Harchol-Balter M, Kozuch MA. *SOFTScale: Stealing Opportunistically for Transient Scaling*. Springer Berlin Heidelberg: Berlin, Heidelberg, 2012; 142–163.
32. Klein C, *et al.*. Brownout: Building more robust cloud applications. *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, ACM: New York, NY, USA, 2014; 700–711.
33. Liu Z, Lin M, Wierman A, Low S, Andrew LLH. Greening geographical load balancing. *IEEE/ACM Trans. Netw.* Apr 2015; **23**(2):657–671.
34. Zhang Y, Wang Y, Wang X. Greenware: Greening cloud-scale data centers to maximize the use of renewable energy. *Proceedings of ACM/IFIP/USENIX 12th International Middleware Conference*, Springer, 2011; 143–164.
35. Nadjaran Toosi A, Buyya R. A fuzzy logic-based controller for cost and energy efficient load balancing in geo-distributed data centers. *Proceedings of 8th IEEE/ACM International Conferencce on Utility and Cloud Computing (UCC)*, IEEE, 2015.
36. Grozev N, Buyya R. Multi-cloud provisioning and load distribution for three-tier applications. *ACM Transactions on Autonomous and Adaptive Systems* 2014; **9**(3):13:1–13:21.

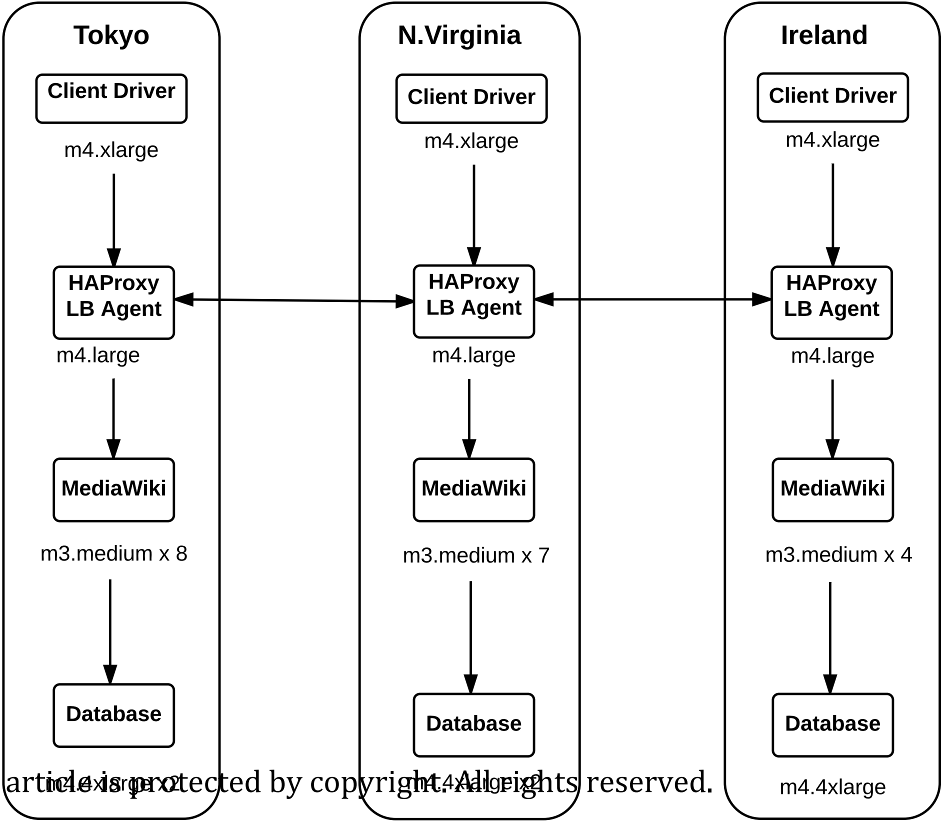

File Id
User Profile
Front-end
Session data
Local
Session
Cache
Advertisement
Service
Media
Service
Credential
User name
Keywords
Ad Id
File Id
Authentication
Service
Timeline
Service
Search
Service
Cache
Service
User name
Friend id
Friendship
Service
Get Posts
Service

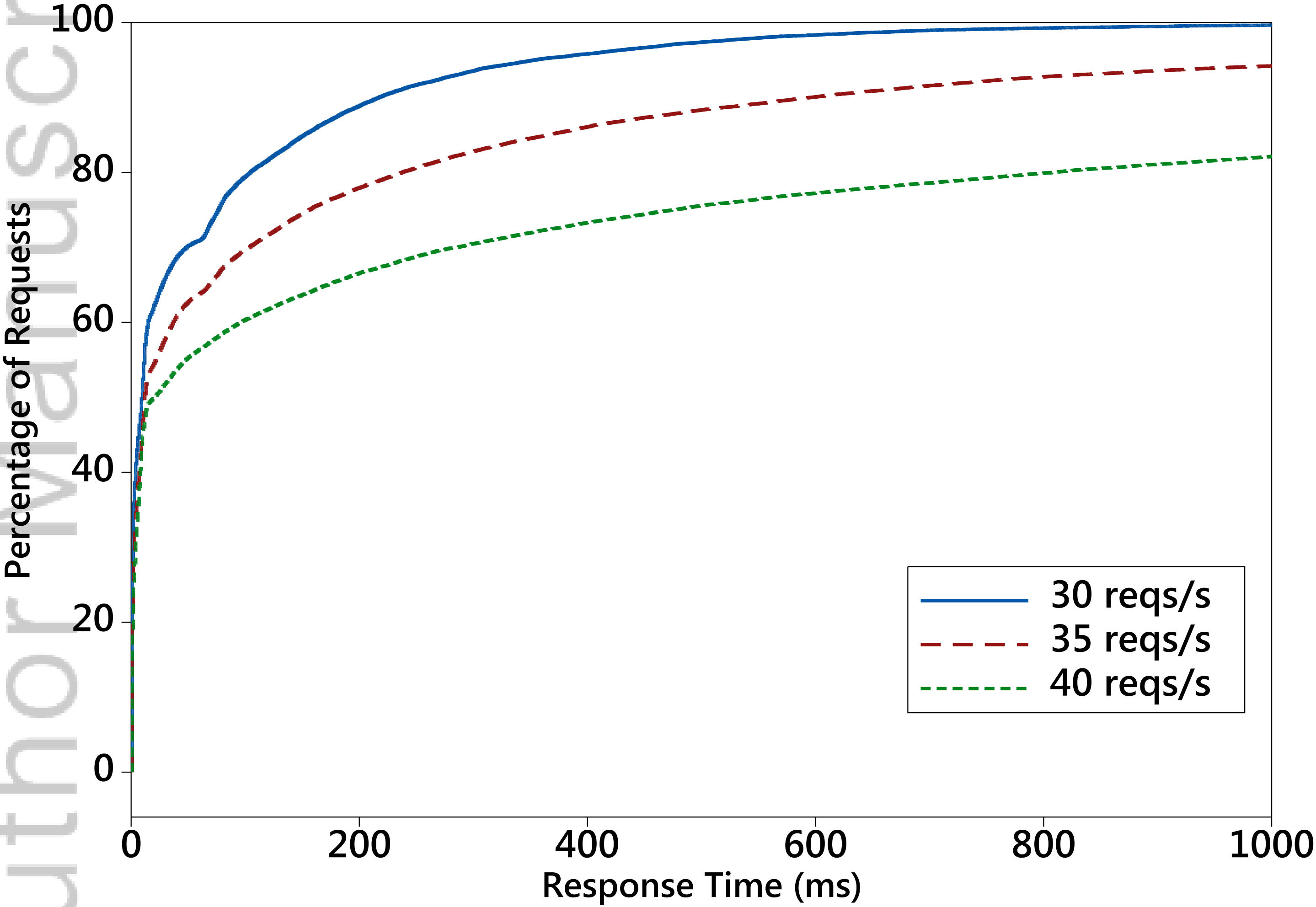

Load Balancing Agent
Load Balancer
Monitoring Module
Control Module
Commun. Module
Load Balancing Agent
Load Balancer
Monitoring Module
Control Module
Commun. Module
Load Balancing Agent
Load Balancer
Monitoring Module
Control Module
Commun. Module

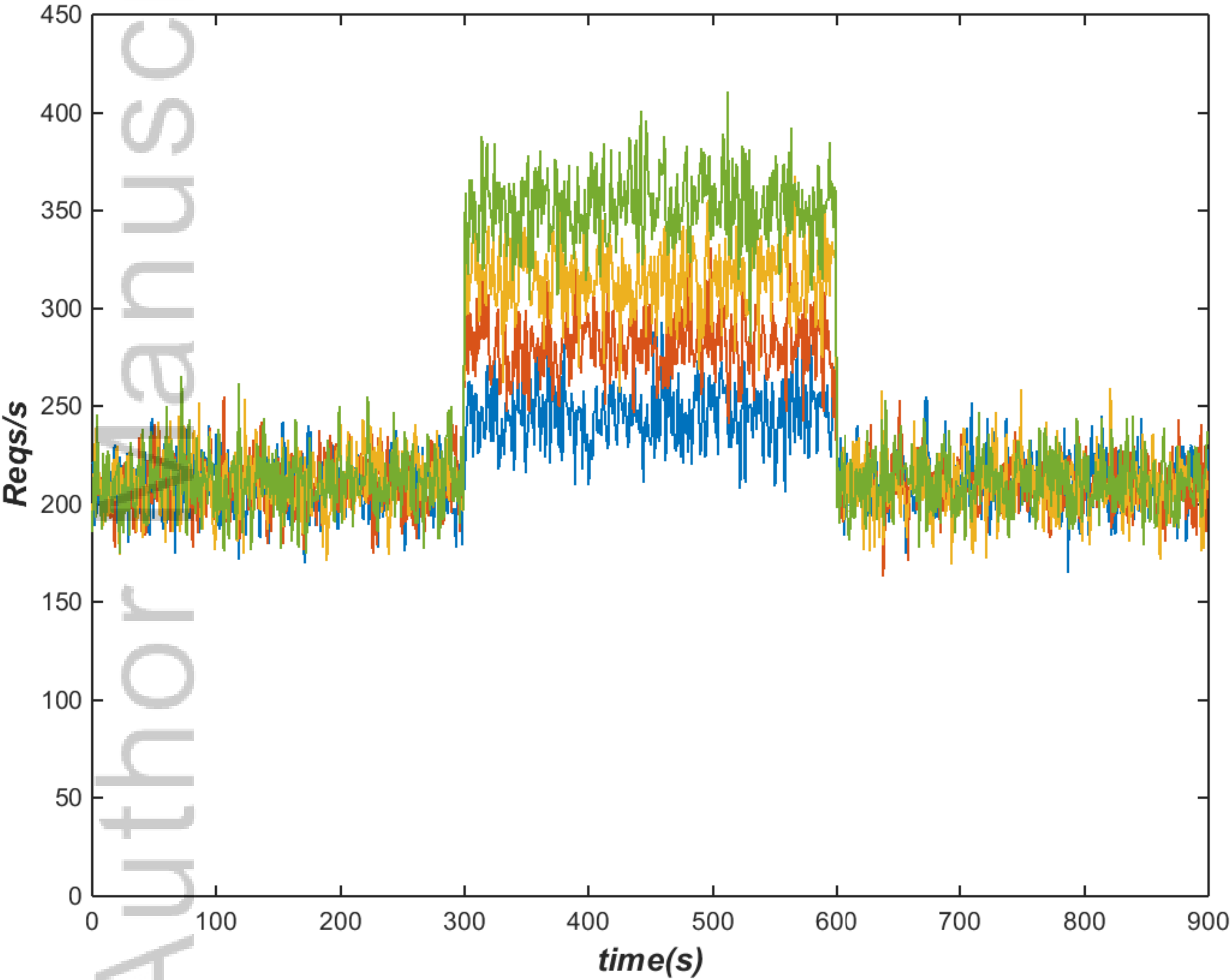


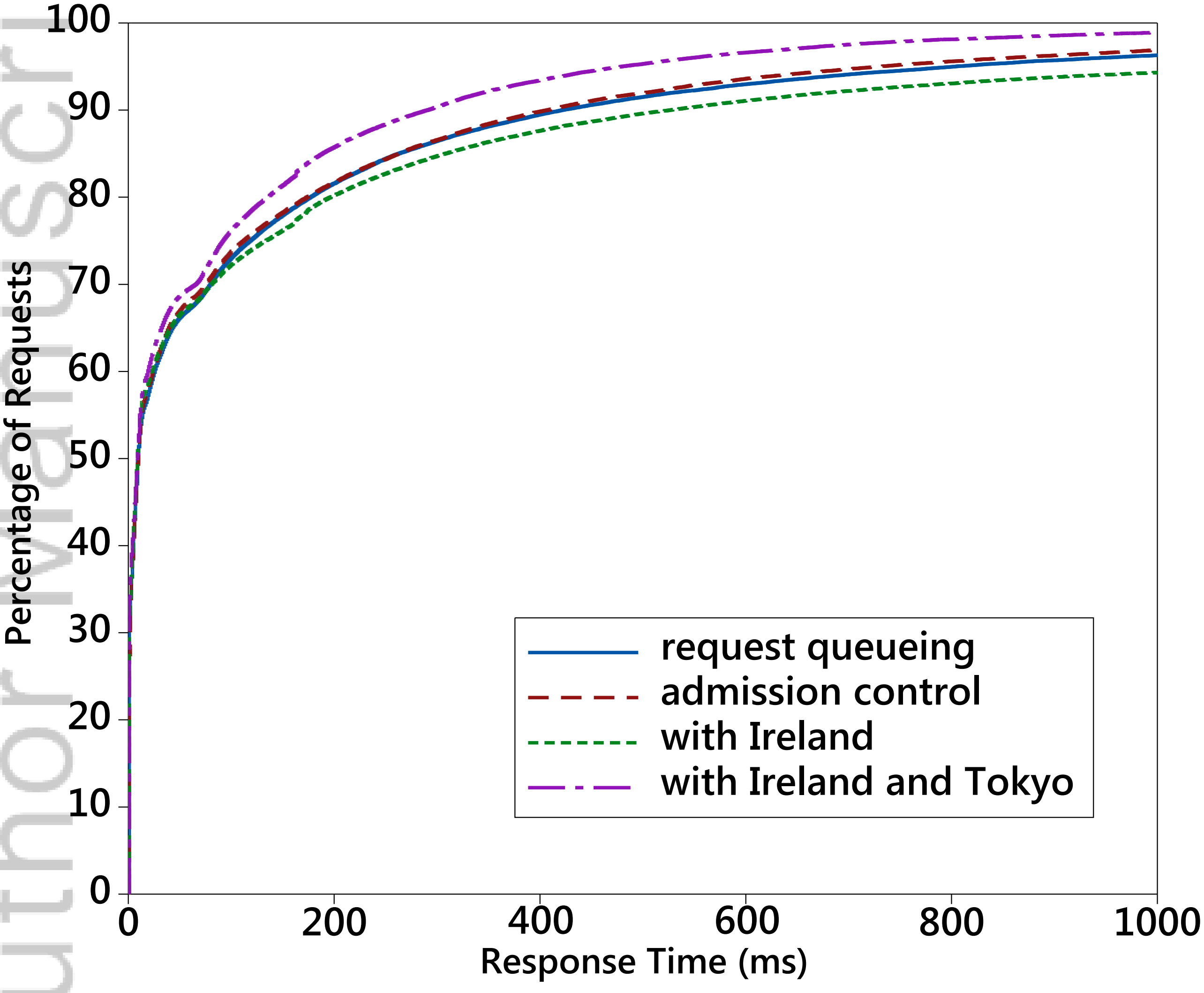
```
listen webcluster
    bind    *:80
    mode    http
    option  httplog
    stats   enable

    balance roundrobin
    option httpchk HEAD / HTTP/1.0
    option forwardfor

    acl monitoring src localhost
    acl test rand(250) lt 33
    tcp-request content reject if test !monitoring

    server local-1 172.31.61.40:80 check inter 2000ms weight 35
    server local-2 172.31.54.227:80 check inter 2000ms weight 35
    server local-3 172.31.55.239:80 check inter 2000ms weight 35
    server local-4 172.31.61.130:80 check inter 2000ms weight 35
    server local-5 172.31.58.35:80 check inter 2000ms weight 35

    server ireland 52.208.134.152:80 weight 13
    server tokyo 54.132.154.120:80 weight 29
```

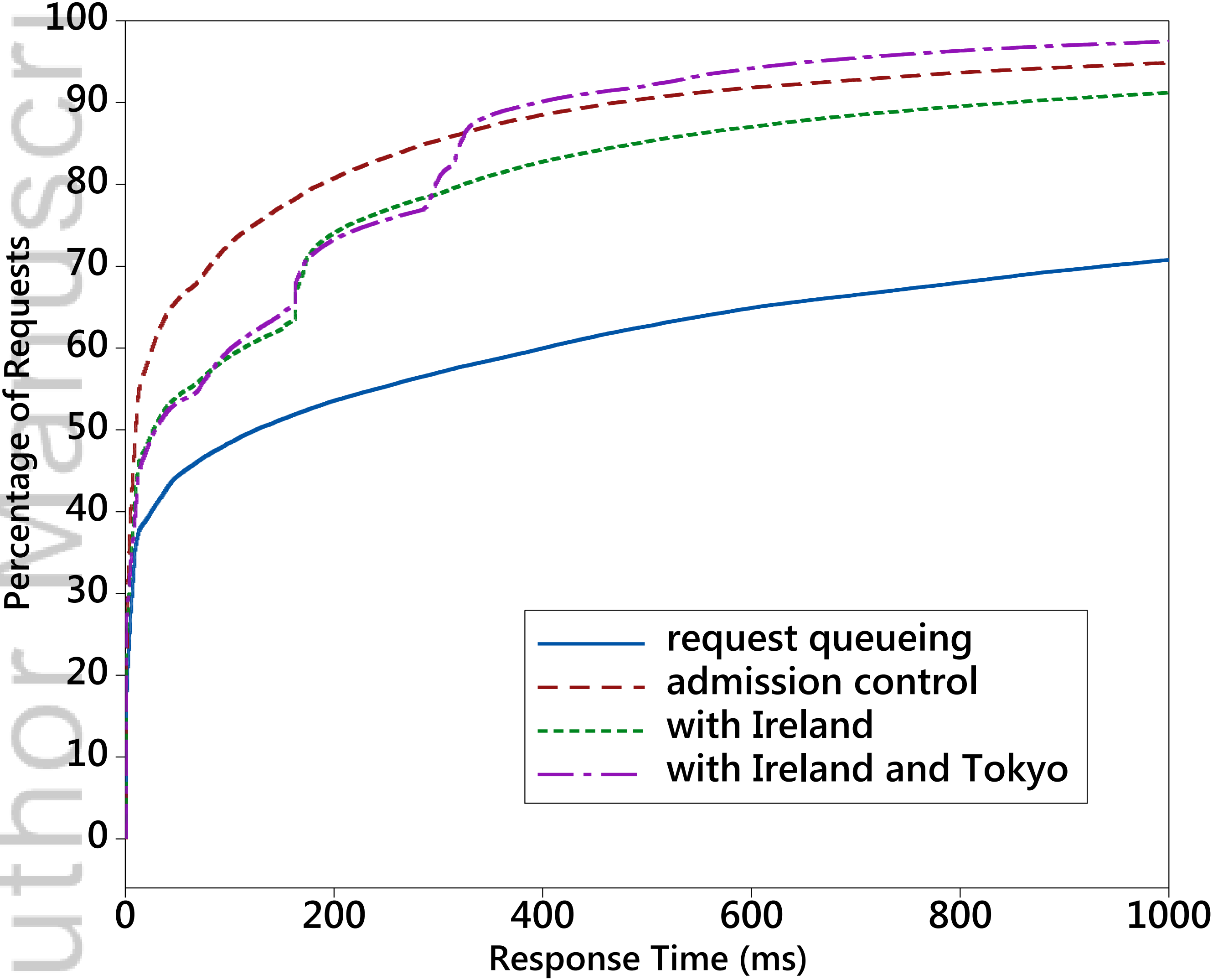

The Client Driver
HAProxy Load Balancer and Our System
MediaWiki Servers
English Wikipedia Database

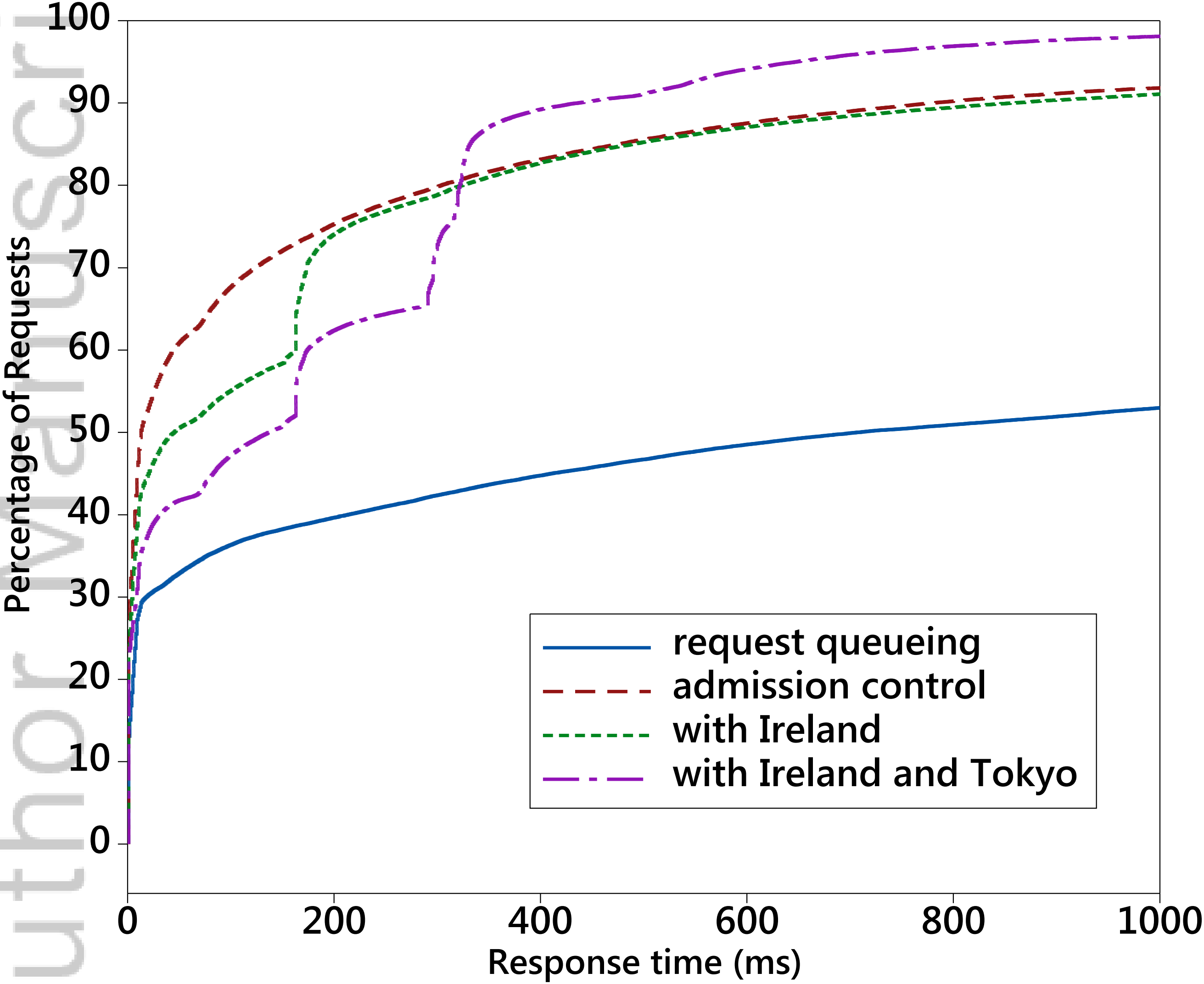

Tokyo
Client Driver
m4.xlarge
HAProxy LB Agent
m4.large
MediaWiki
m3.medium x 8
Database
m4.4xlarge x2
N.Virginia
Client Driver
m4.xlarge
HAProxy LB Agent
m4.large
MediaWiki
m3.medium x 7
Database
m4.4xlarge x2
Ireland
Client Driver
m4.xlarge
HAProxy LB Agent
m4.large
MediaWiki
m3.medium x 4
Database
m4.4xlarge

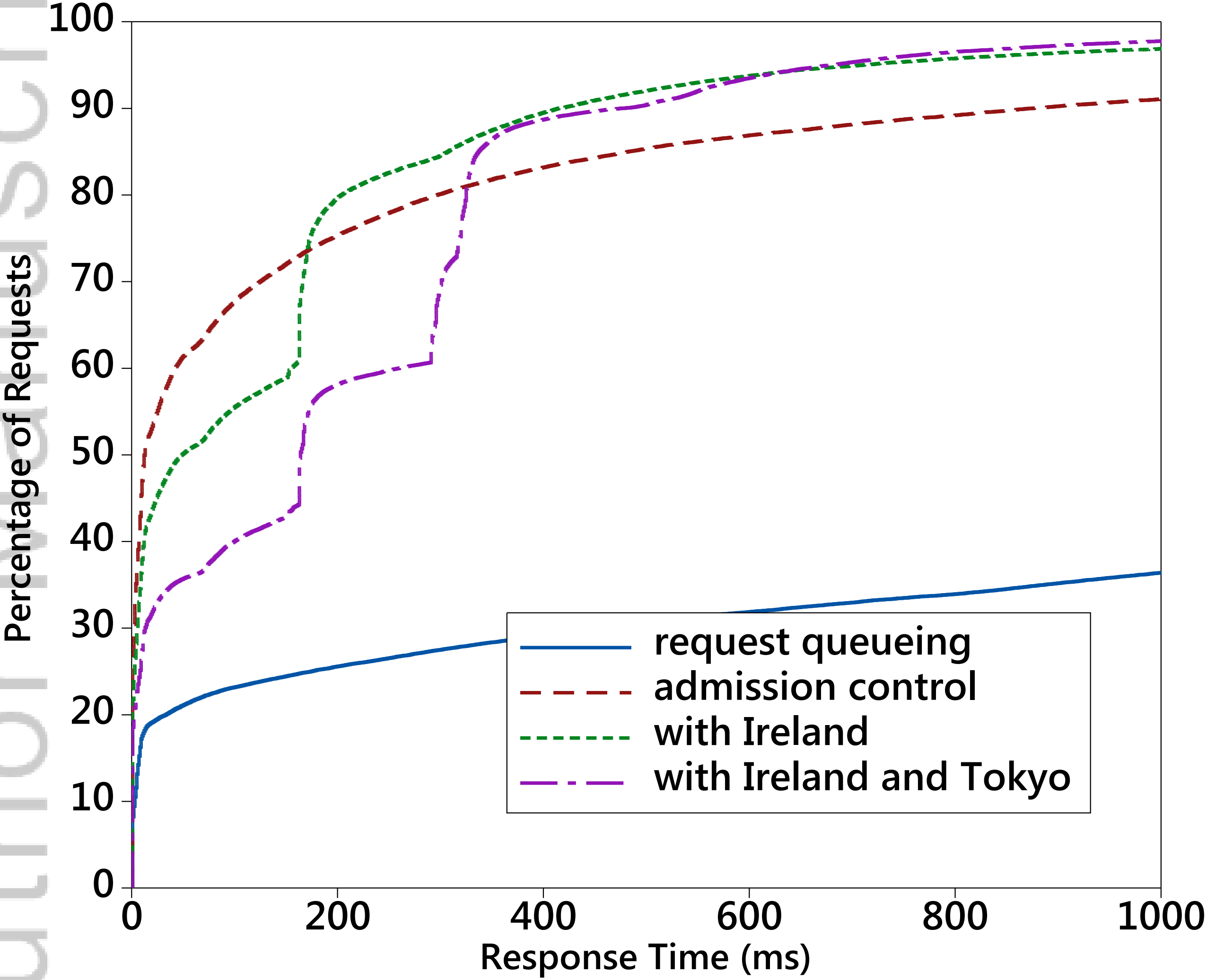

Percentage of Requests
100
80
60
40
20
0
0
200
400
600
800
1000
Response Time (ms)
30 reqs/s
35 reqs/s
40 reqs/s

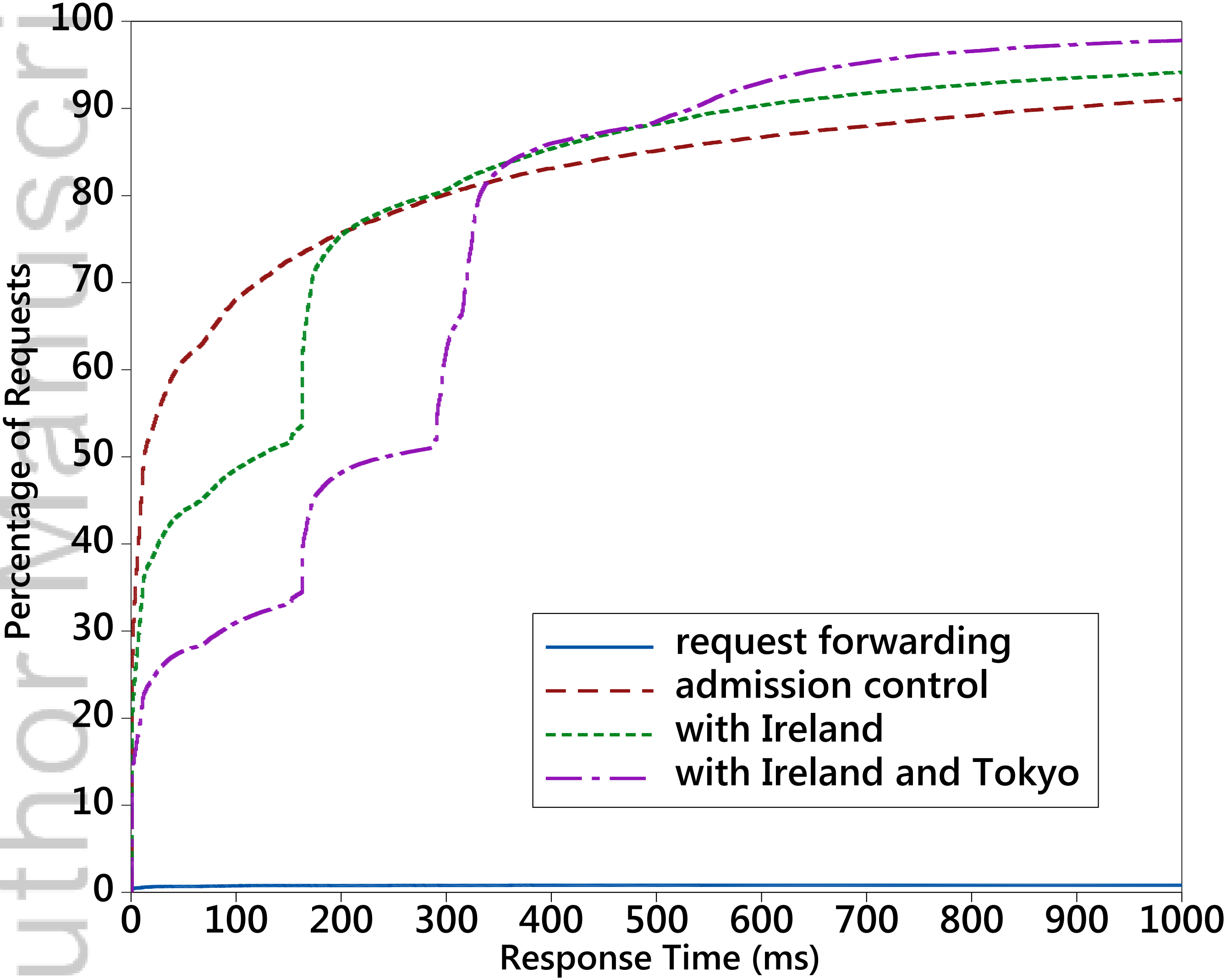

450
400
350
300
250
200
150
100
50
0
Reqs/s
0
100
200
300
400
500
600
700
800
900
time(s)

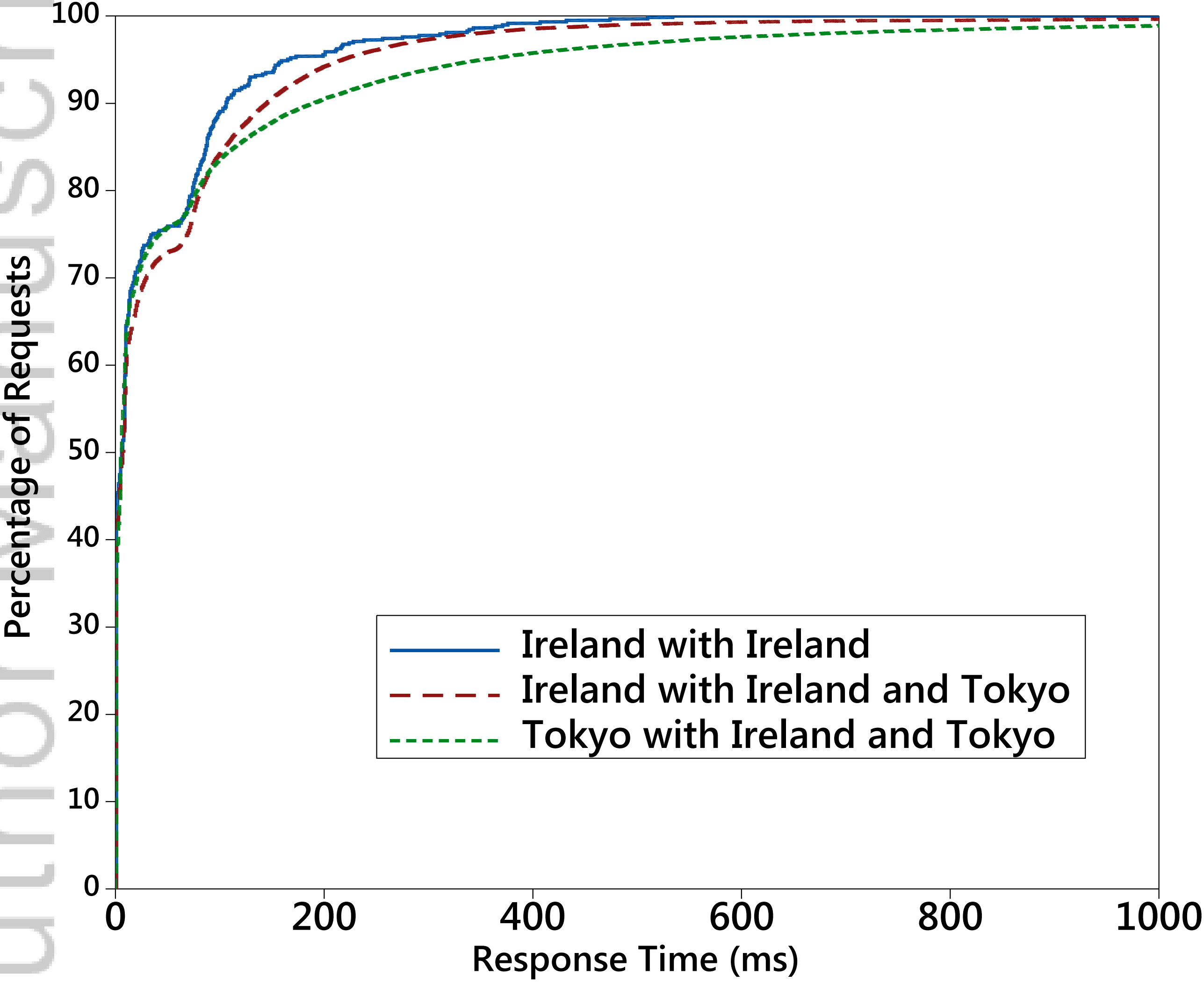

100
90
80
70
60
50
40
30
20
10
0
Percentage of Requests
0
200
400
600
800
1000
Response Time (ms)
request queueing
admission control
with Ireland
with Ireland and Tokyo

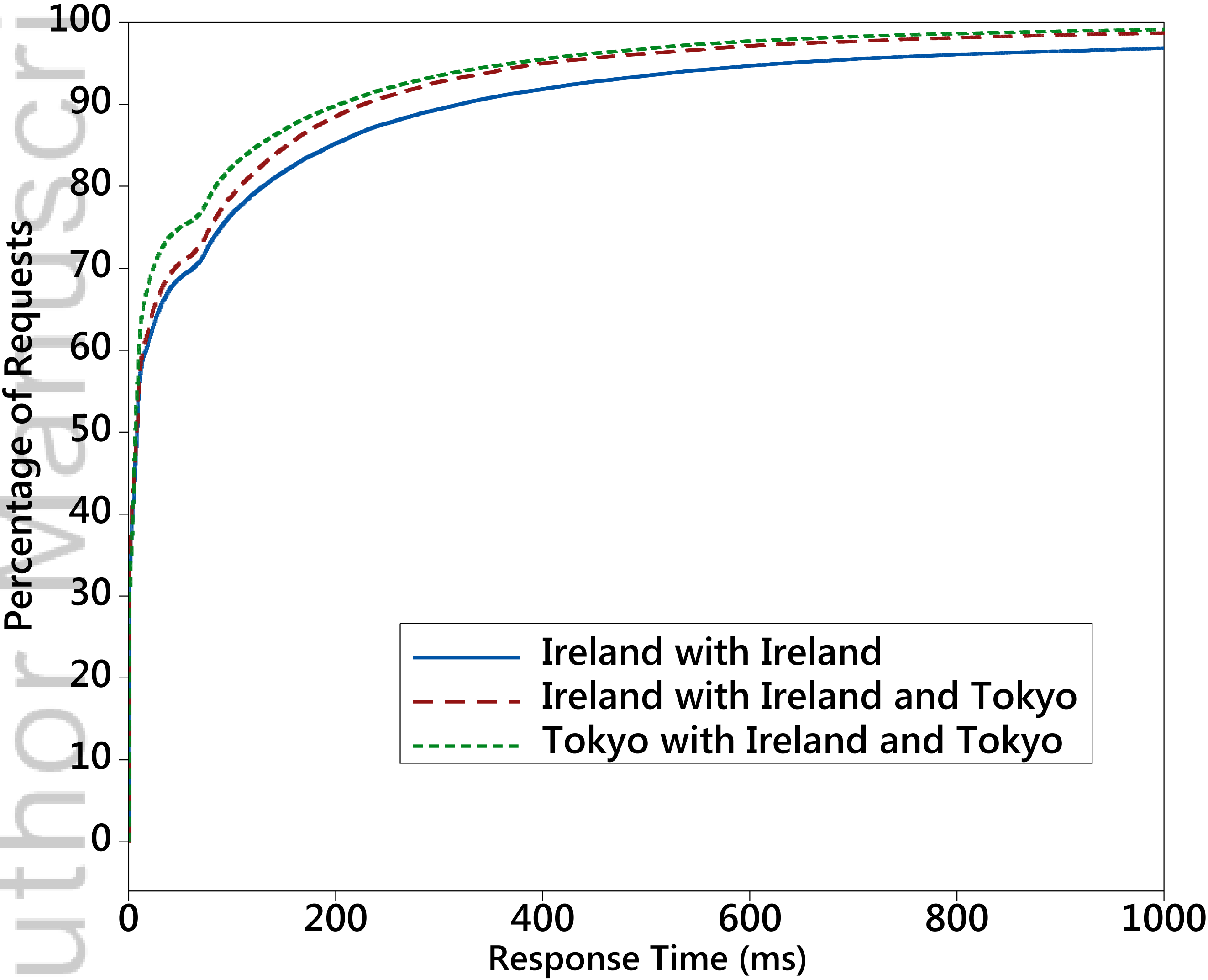

100
90
80
70
60
50
40
30
20
10
0
Percentage of Requests
0
200
400
600
800
1000
Response Time (ms)
request queueing
admission control
with Ireland
with Ireland and Tokyo

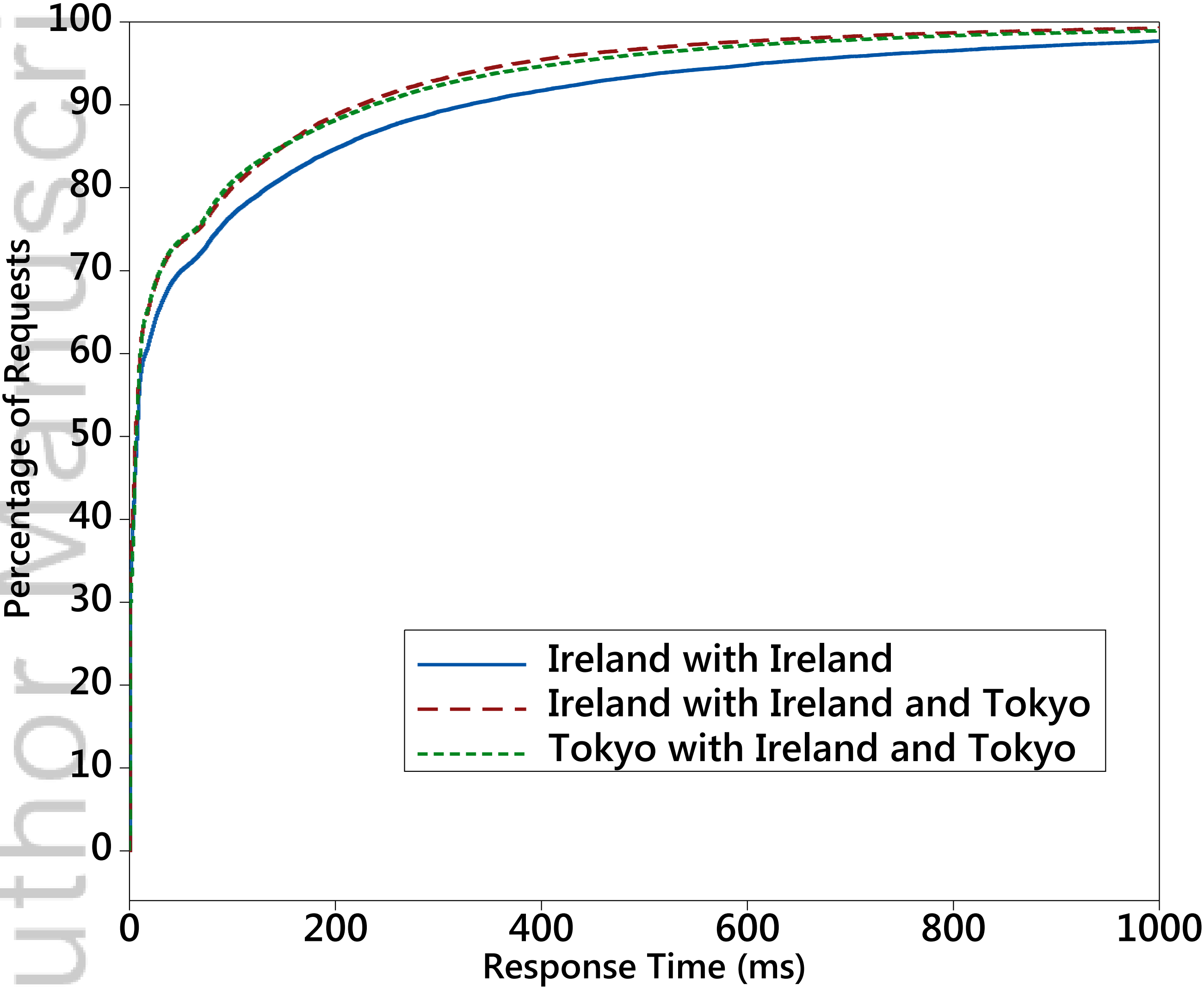

100
90
80
70
60
50
40
30
20
10
0
Percentage of Requests
0
200
400
600
800
1000
Response time (ms)
request queueing
admission control
with Ireland
with Ireland and Tokyo

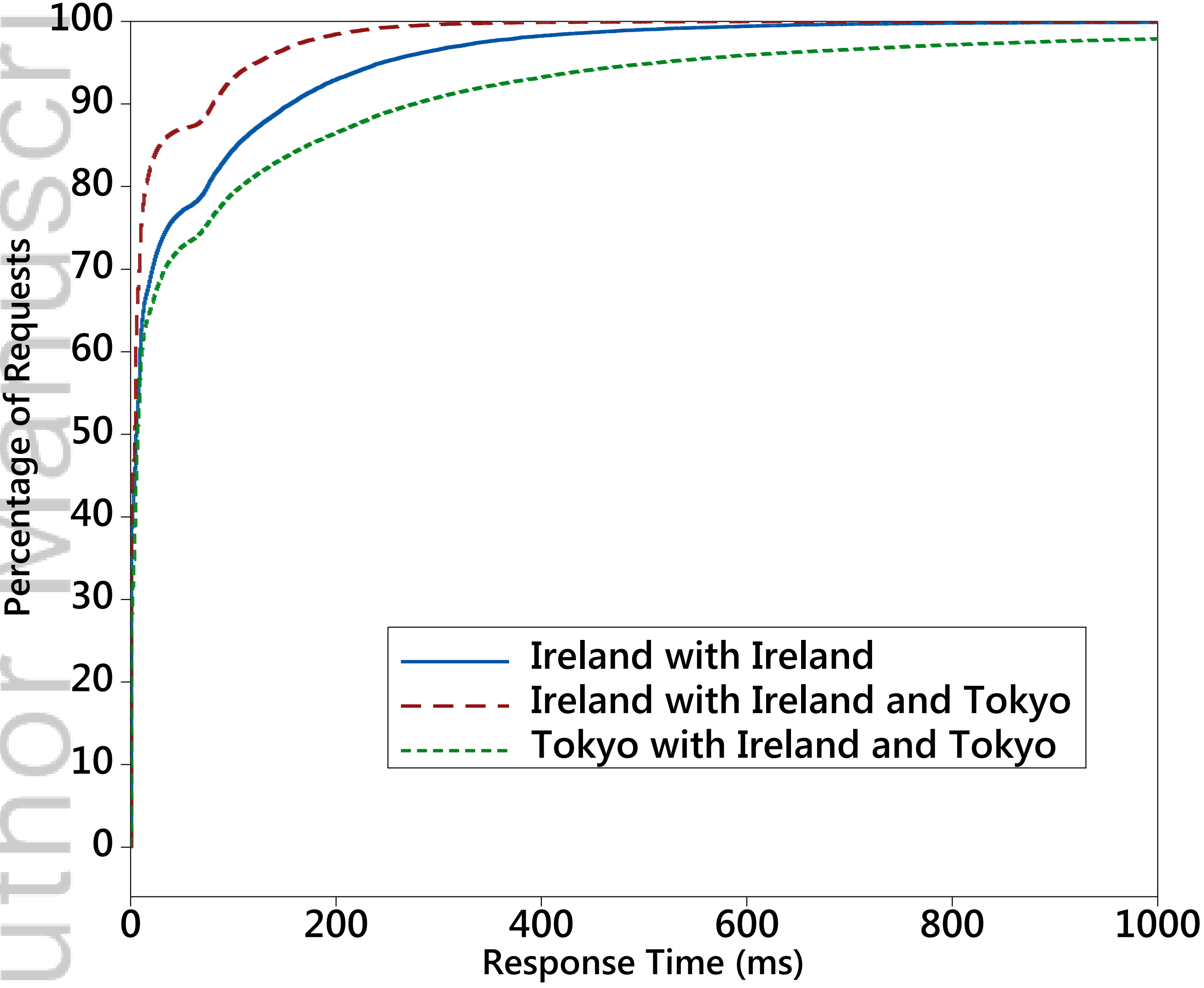

Percentage of Requests
100
90
80
70
60
50
40
30
20
10
0
0
200
400
600
800
1000
Response Time (ms)
request queueing
admission control
with Ireland
with Ireland and Tokyo

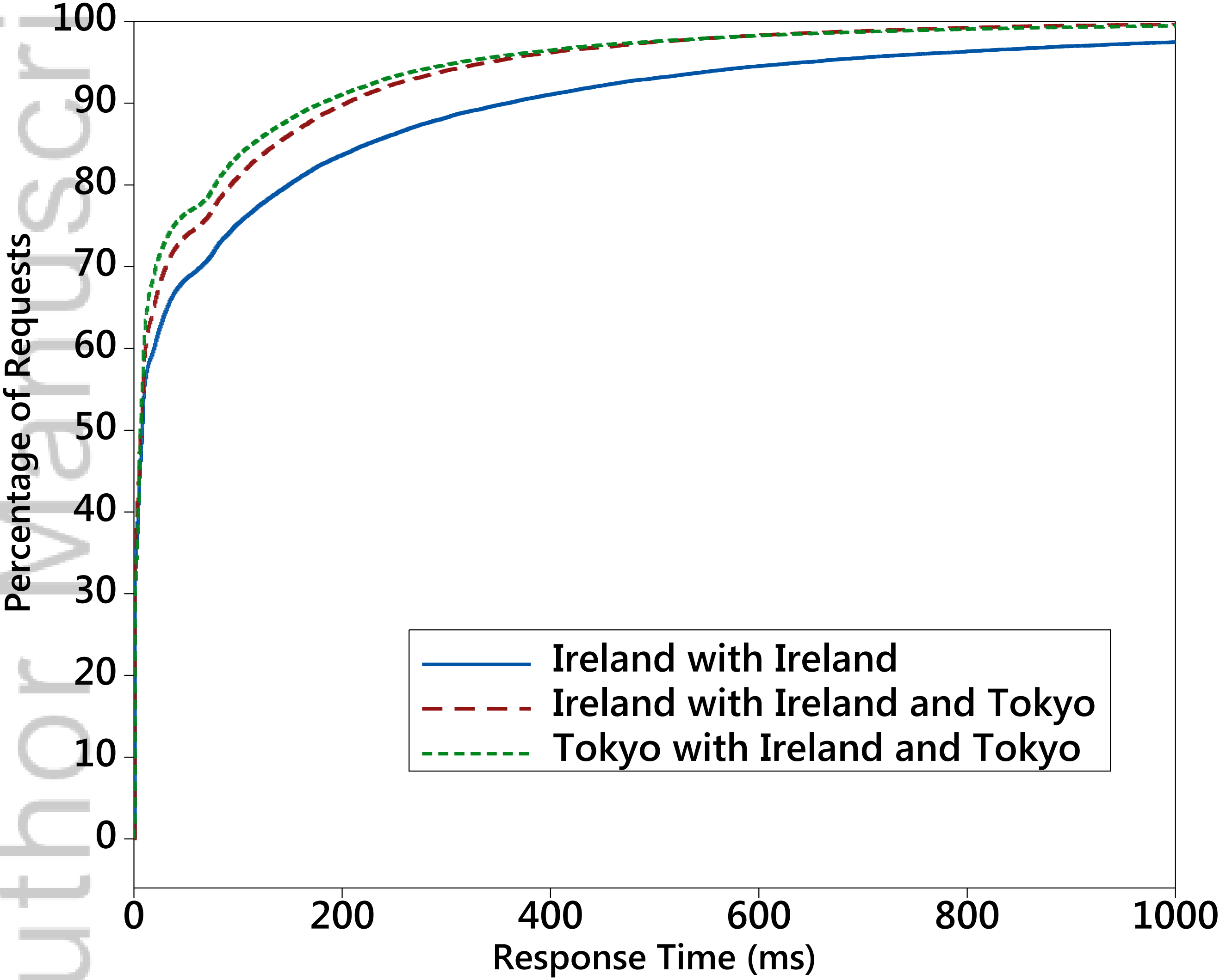

Percentage of Requests
100
90
80
70
60
50
40
30
20
10
0
0
100
200
300
400
500
600
700
800
900
1000
Response Time (ms)
request forwarding
admission control
with Ireland
with Ireland and Tokyo

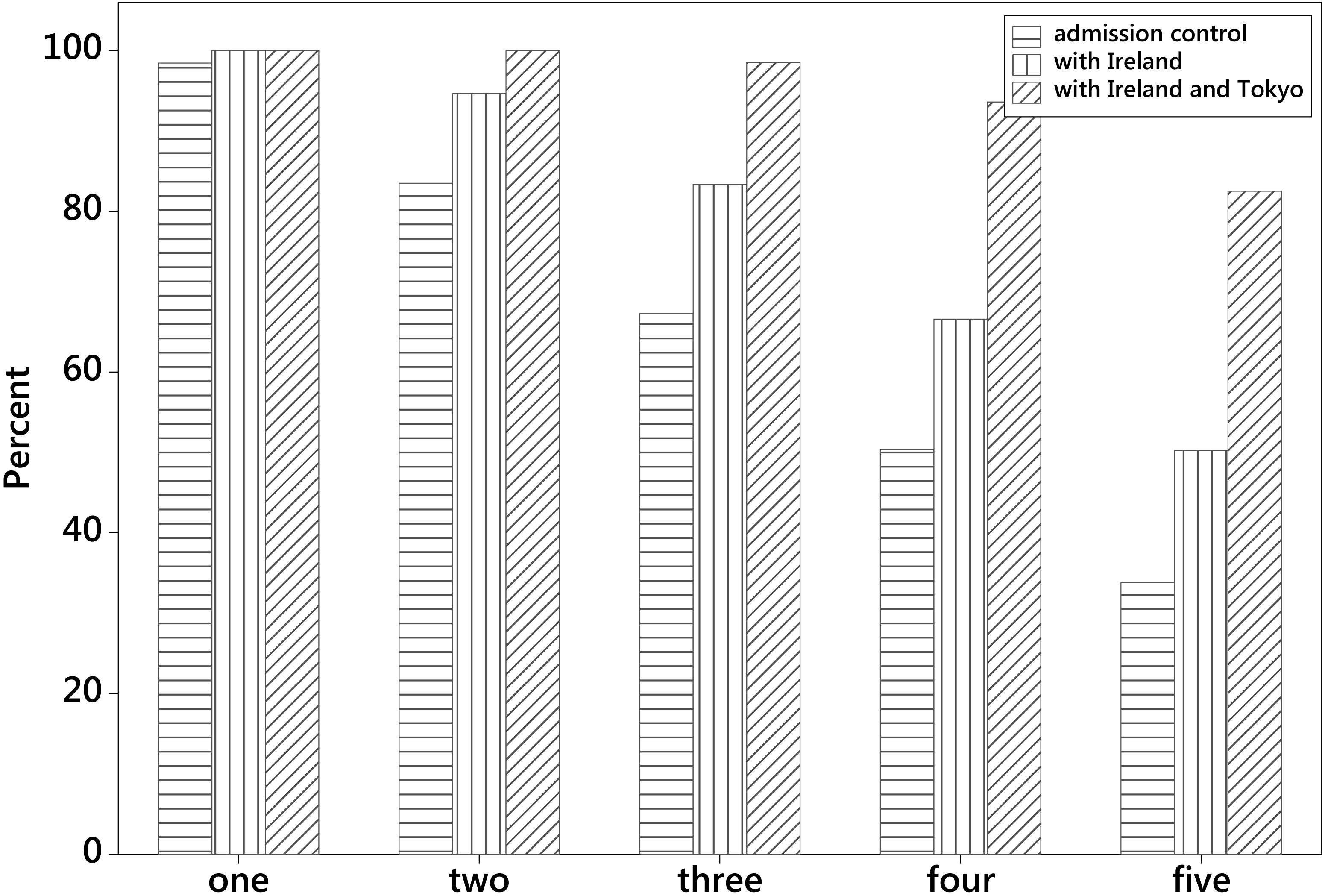

100
90
80
70
60
50
40
30
20
10
0
Percentage of Requests
0
200
400
600
800
1000
Response Time (ms)
Ireland with Ireland
Ireland with Ireland and Tokyo
Tokyo with Ireland and Tokyo

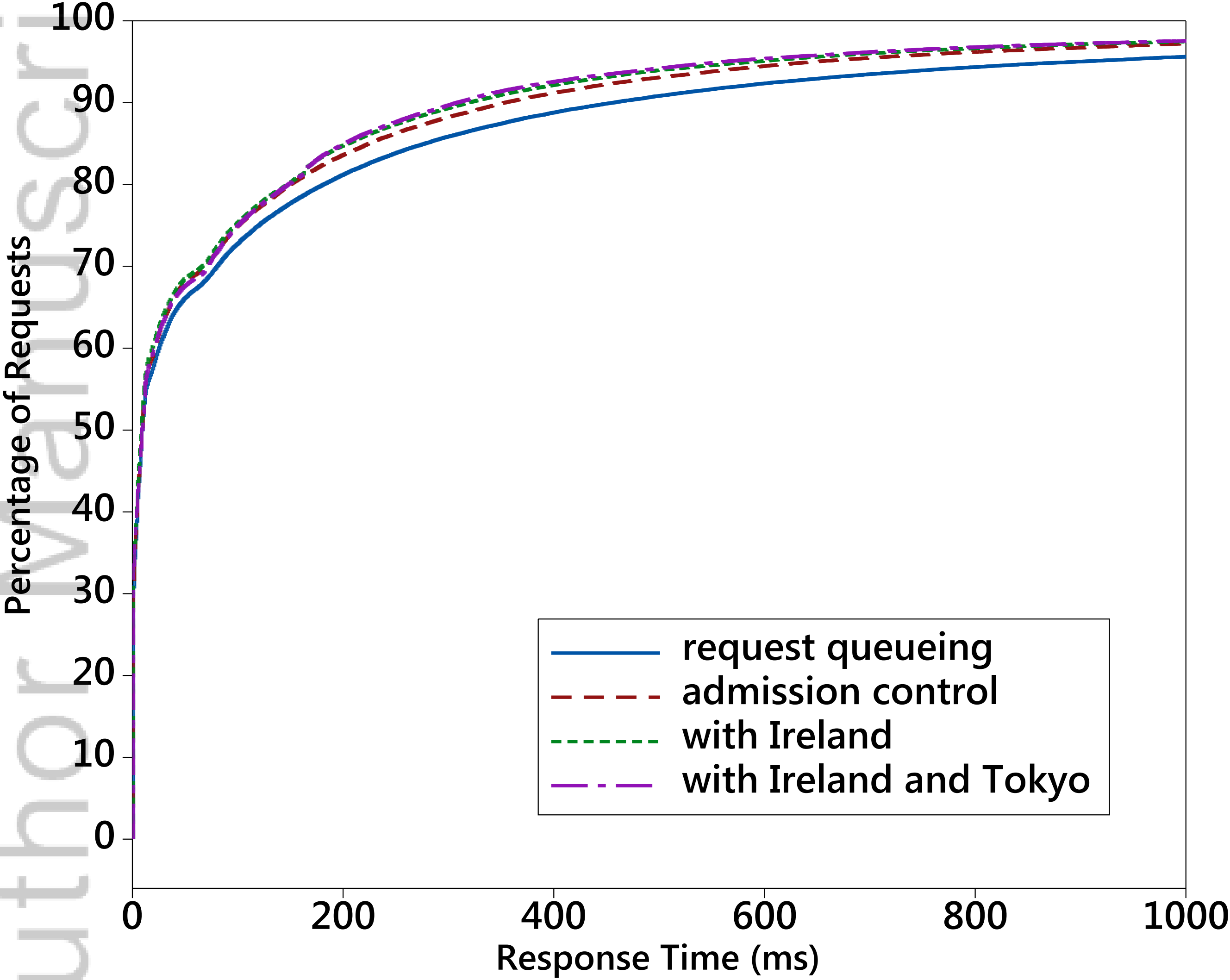

100
90
80
70
60
50
40
30
20
10
0
Percentage of Requests
0
200
400
600
800
1000
Response Time (ms)
Ireland with Ireland
Ireland with Ireland and Tokyo
Tokyo with Ireland and Tokyo

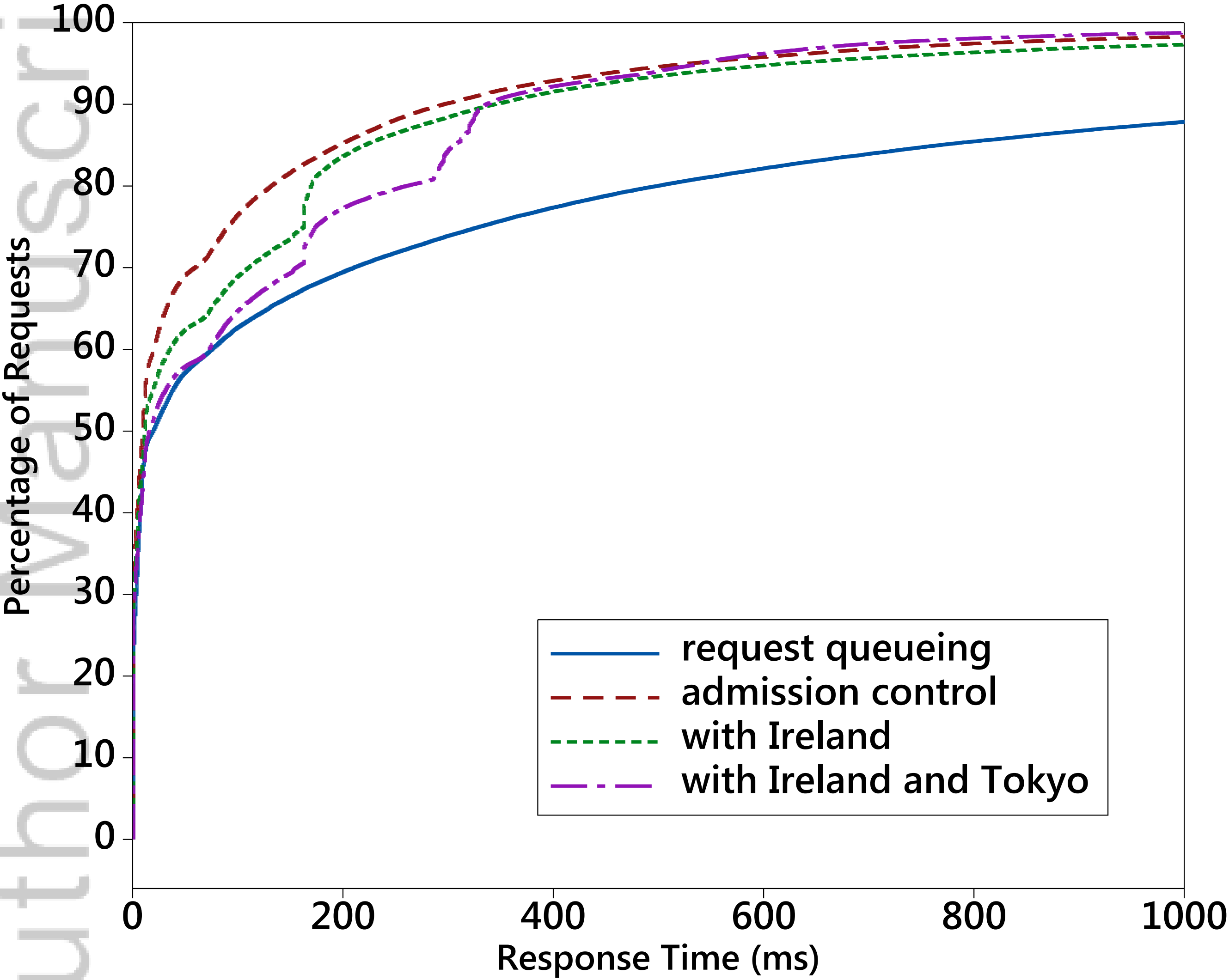

Percentage of Requests
100
90
80
70
60
50
40
30
20
10
0
0
200
400
600
800
1000
Response Time (ms)
Ireland with Ireland
Ireland with Ireland and Tokyo
Tokyo with Ireland and Tokyo

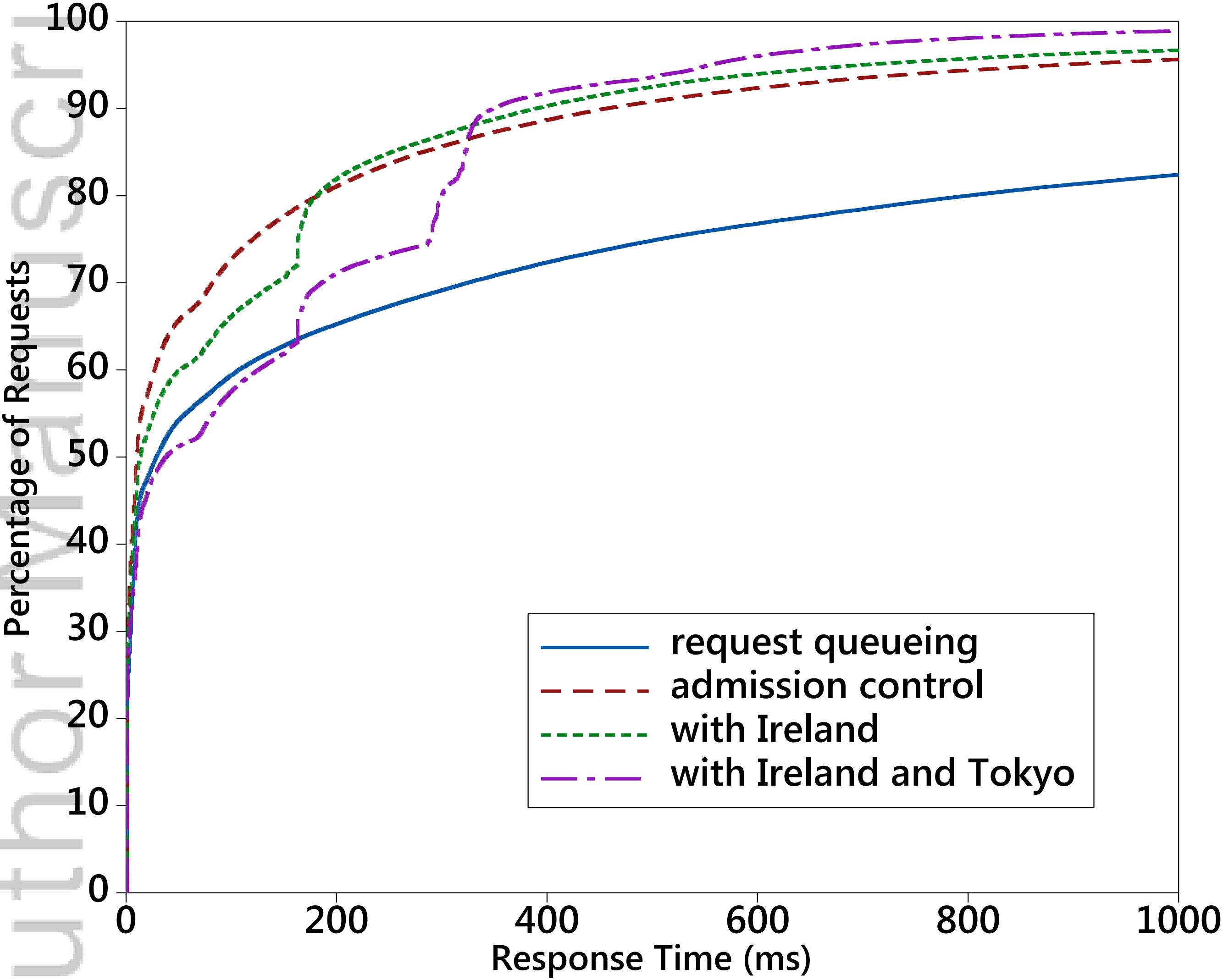

Percentage of Requests
100
90
80
70
60
50
40
30
20
10
0
0
200
400
600
800
1000
Response Time (ms)
Ireland with Ireland
Ireland with Ireland and Tokyo
Tokyo with Ireland and Tokyo

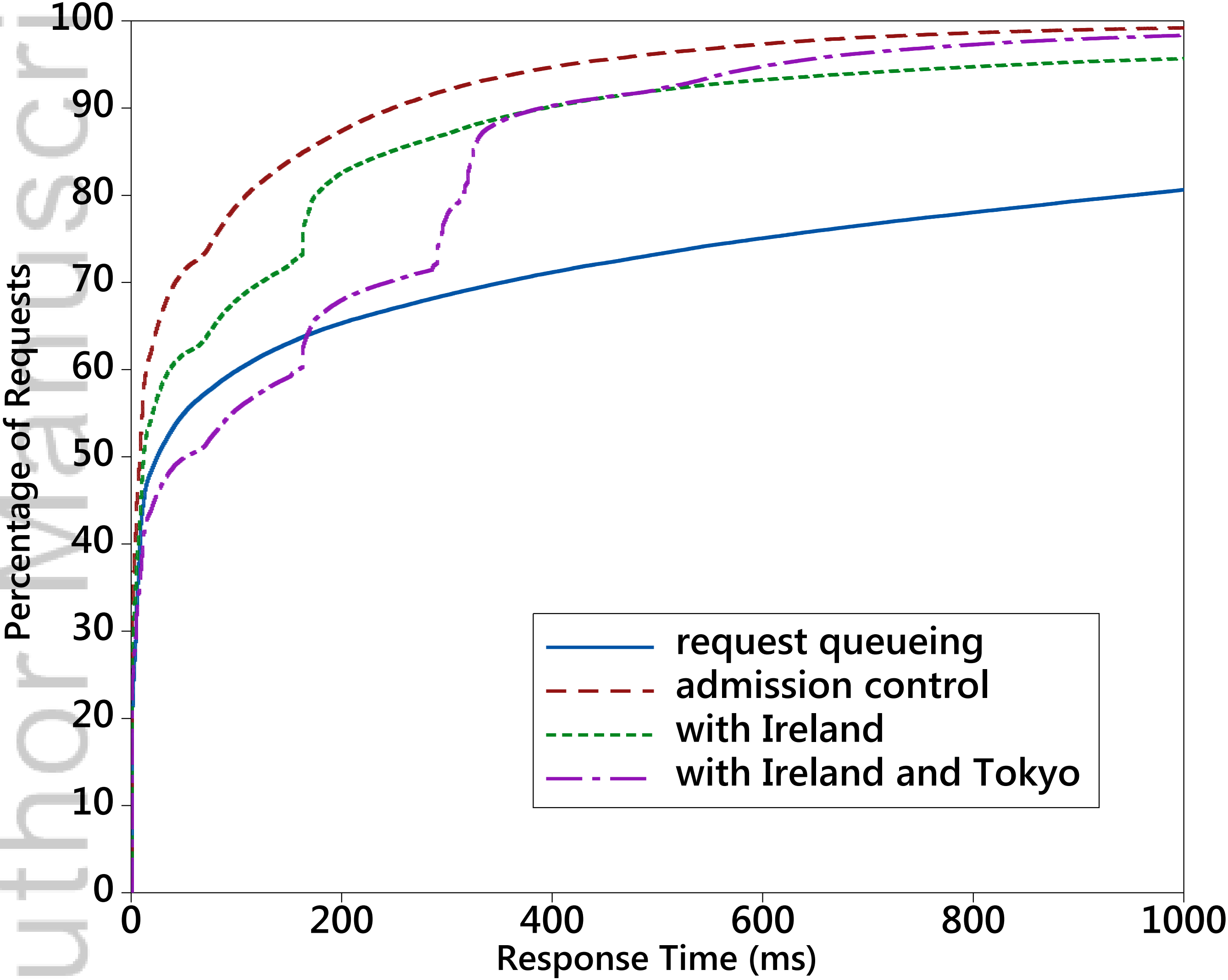

Percentage of Requests
100
90
80
70
60
50
40
30
20
10
0
0
200
400
600
800
1000
Response Time (ms)
Ireland with Ireland
Ireland with Ireland and Tokyo
Tokyo with Ireland and Tokyo

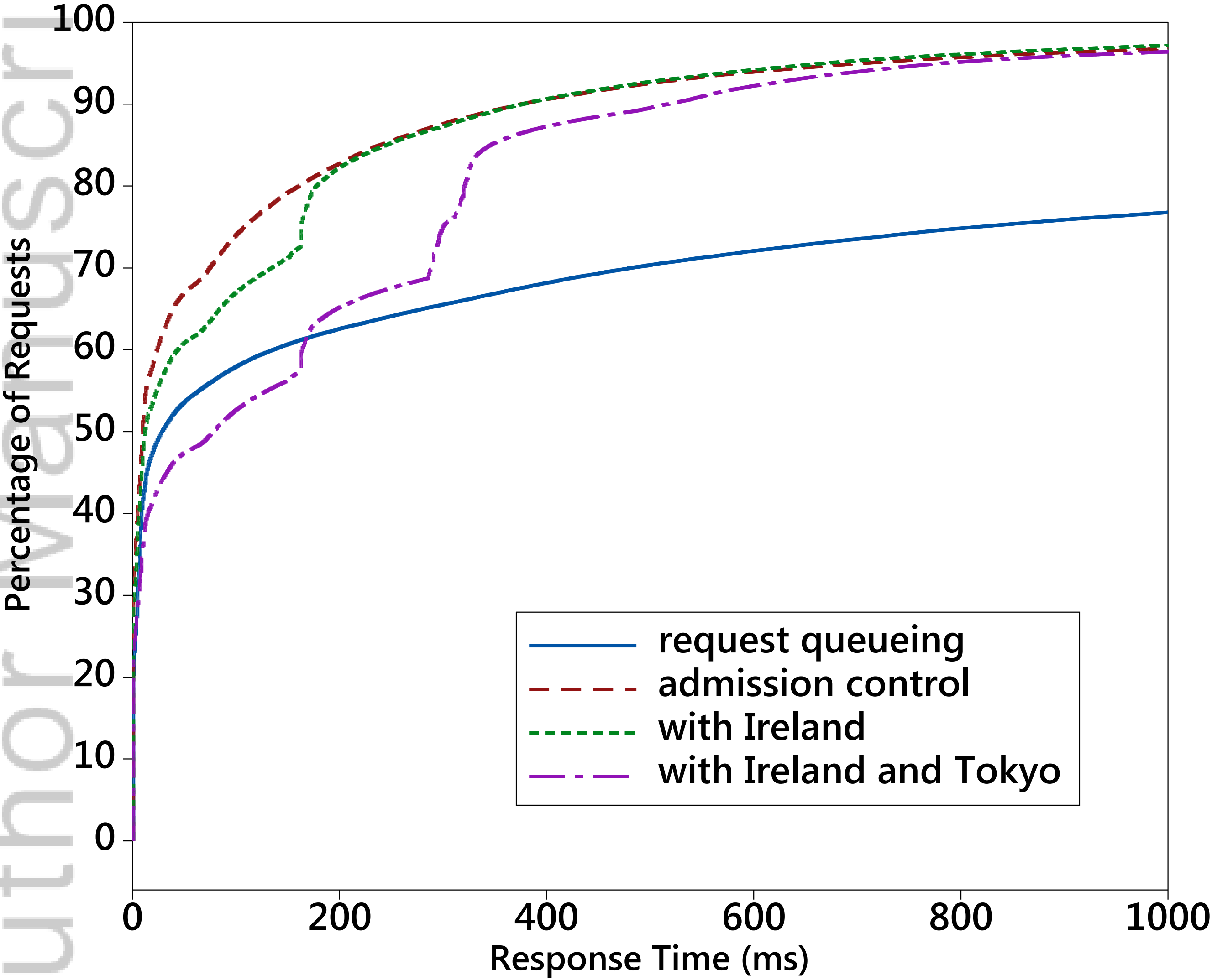

admission control
with Ireland
with Ireland and Tokyo
Percent
100
80
60
40
20
0
one
two
three
four
five

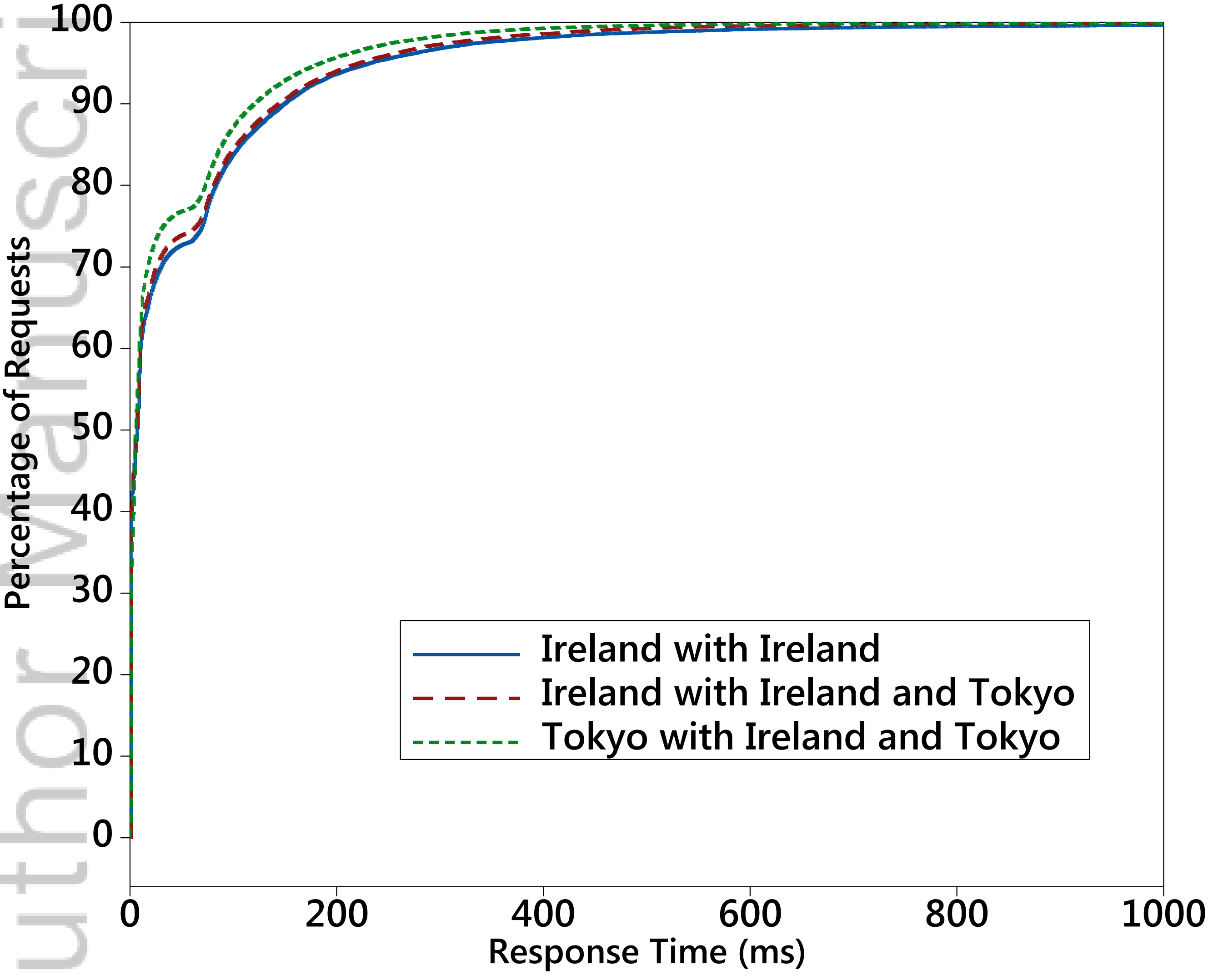

Percentage of Requests
100
90
80
70
60
50
40
30
20
10
0
0
200
400
600
800
1000
Response Time (ms)
request queueing
admission control
with Ireland
with Ireland and Tokyo


Percentage of Requests
100
90
80
70
60
50
40
30
20
10
0
0
200
400
600
800
1000
Response Time (ms)
request queueing
admission control
with Ireland
with Ireland and Tokyo

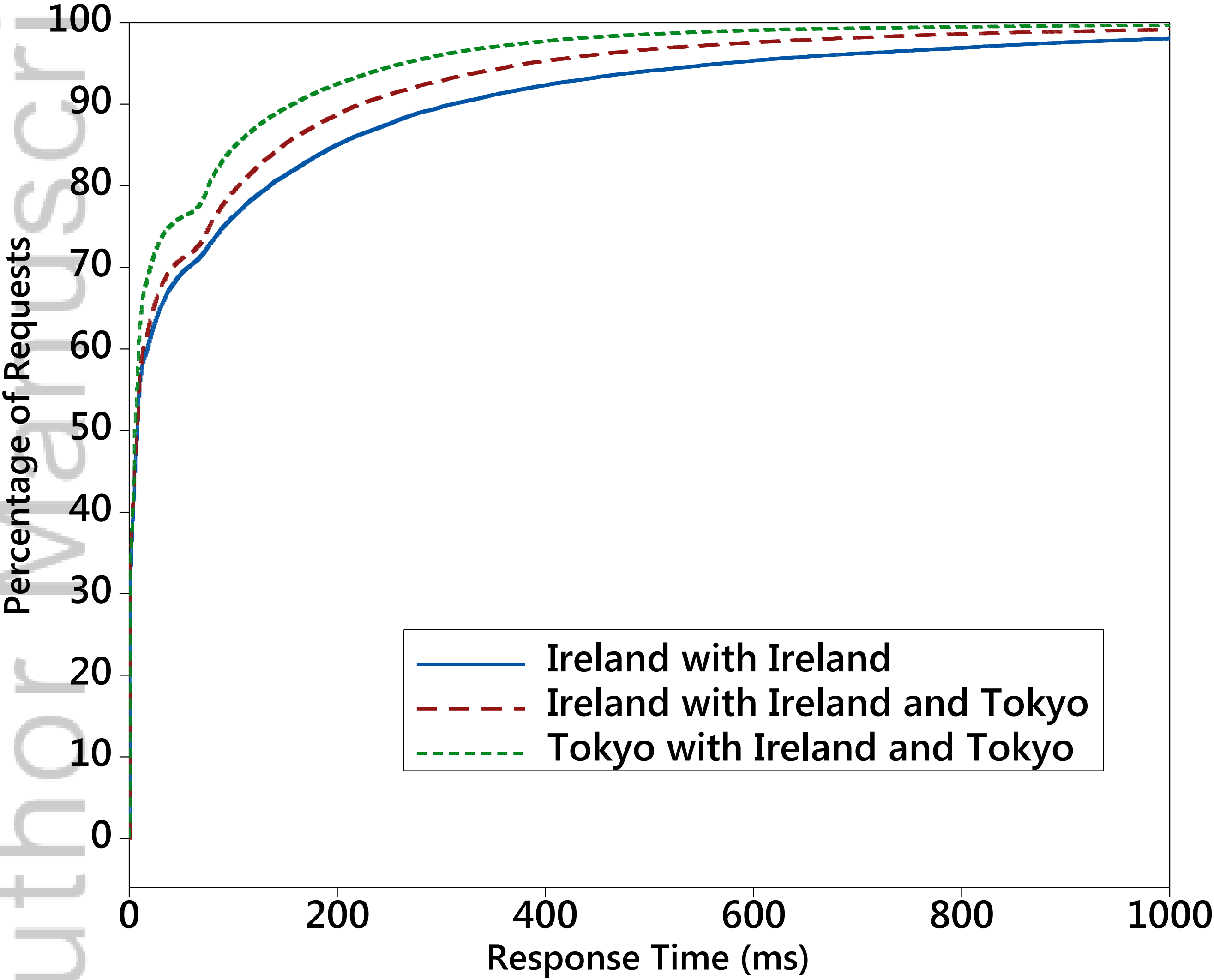

100
90
80
70
60
50
40
30
20
10
0
Percentage of Requests
0
200
400
600
800
1000
Response Time (ms)
request queueing
admission control
with Ireland
with Ireland and Tokyo

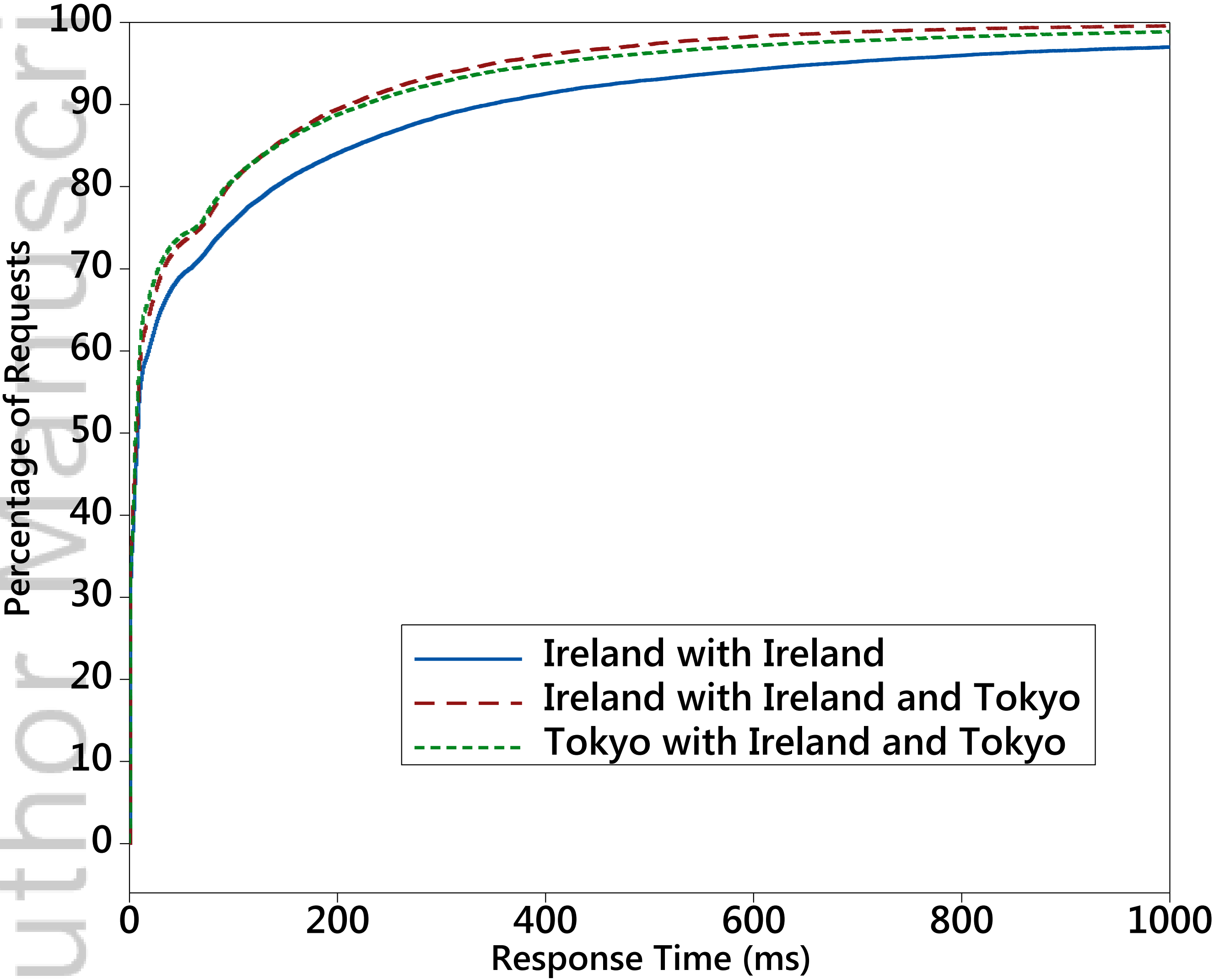

100
90
80
70
60
50
40
30
20
10
0
Percentage of Requests
0
200
400
600
800
1000
Response Time (ms)
request queueing
admission control
with Ireland
with Ireland and Tokyo

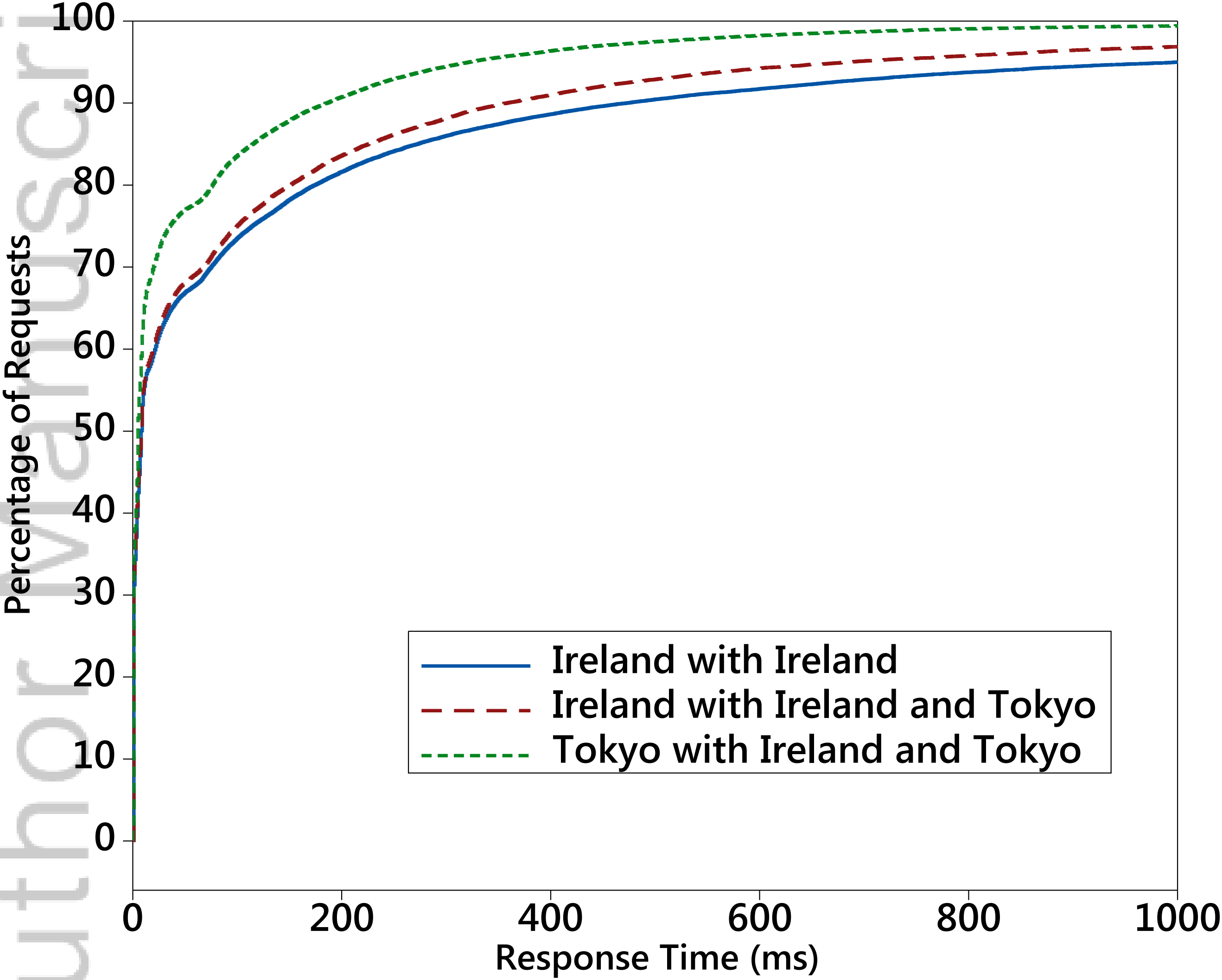

100
90
80
70
60
50
40
30
20
10
0
Percentage of Requests
0
200
400
600
800
1000
Response Time (ms)
request queueing
admission control
with Ireland
with Ireland and Tokyo

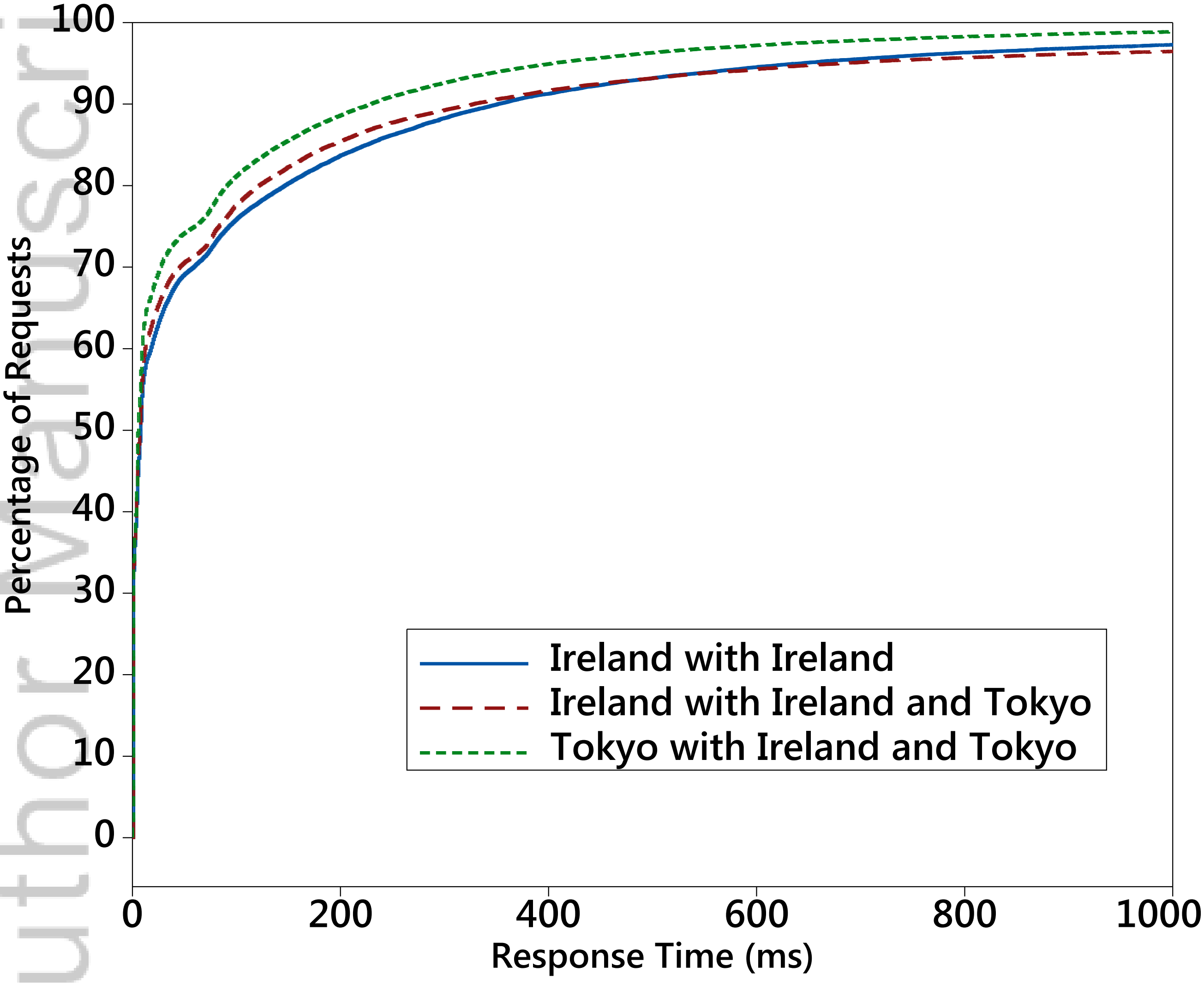


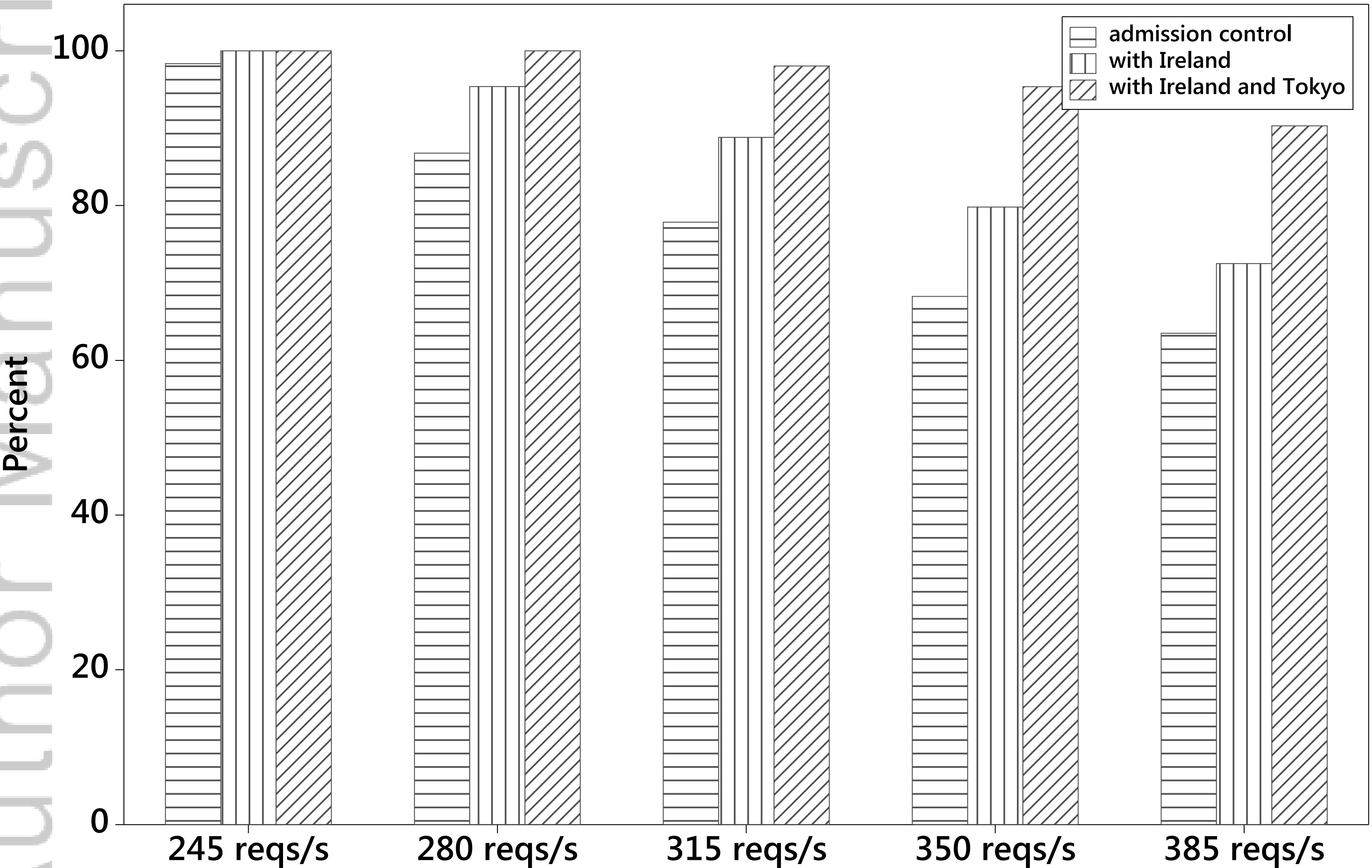
Percentage of Requests
0
10
20
30
40
50
60
70
80
90
100
Response Time (ms)
0
200
400
600
800
1000
Ireland with Ireland
Ireland with Ireland and Tokyo
Tokyo with Ireland and Tokyo

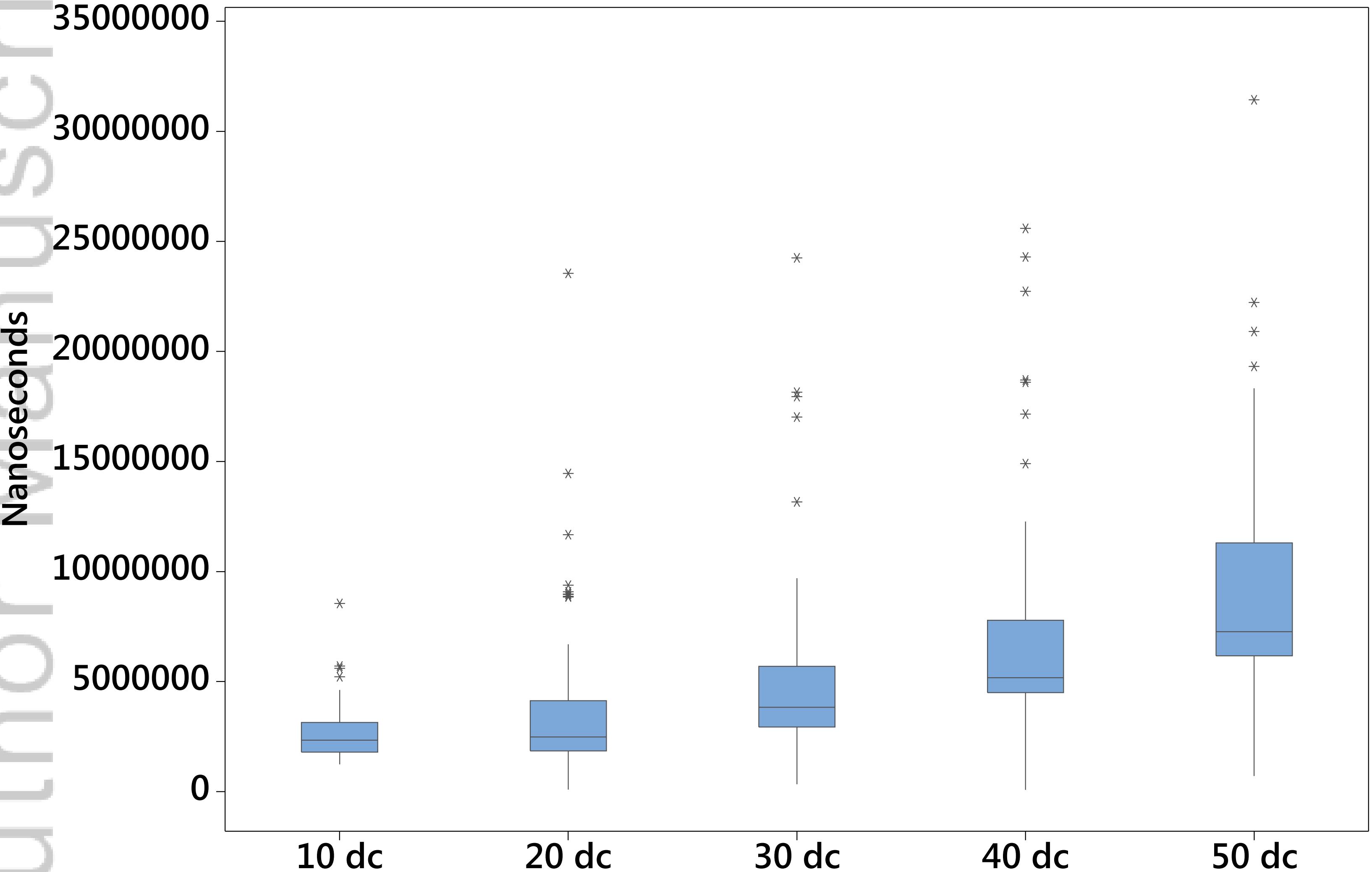

100
90
80
70
60
50
40
30
20
10
0
Percentage of Requests
0
200
400
600
800
1000
Response Time (ms)
Ireland with Ireland
Ireland with Ireland and Tokyo
Tokyo with Ireland and Tokyo

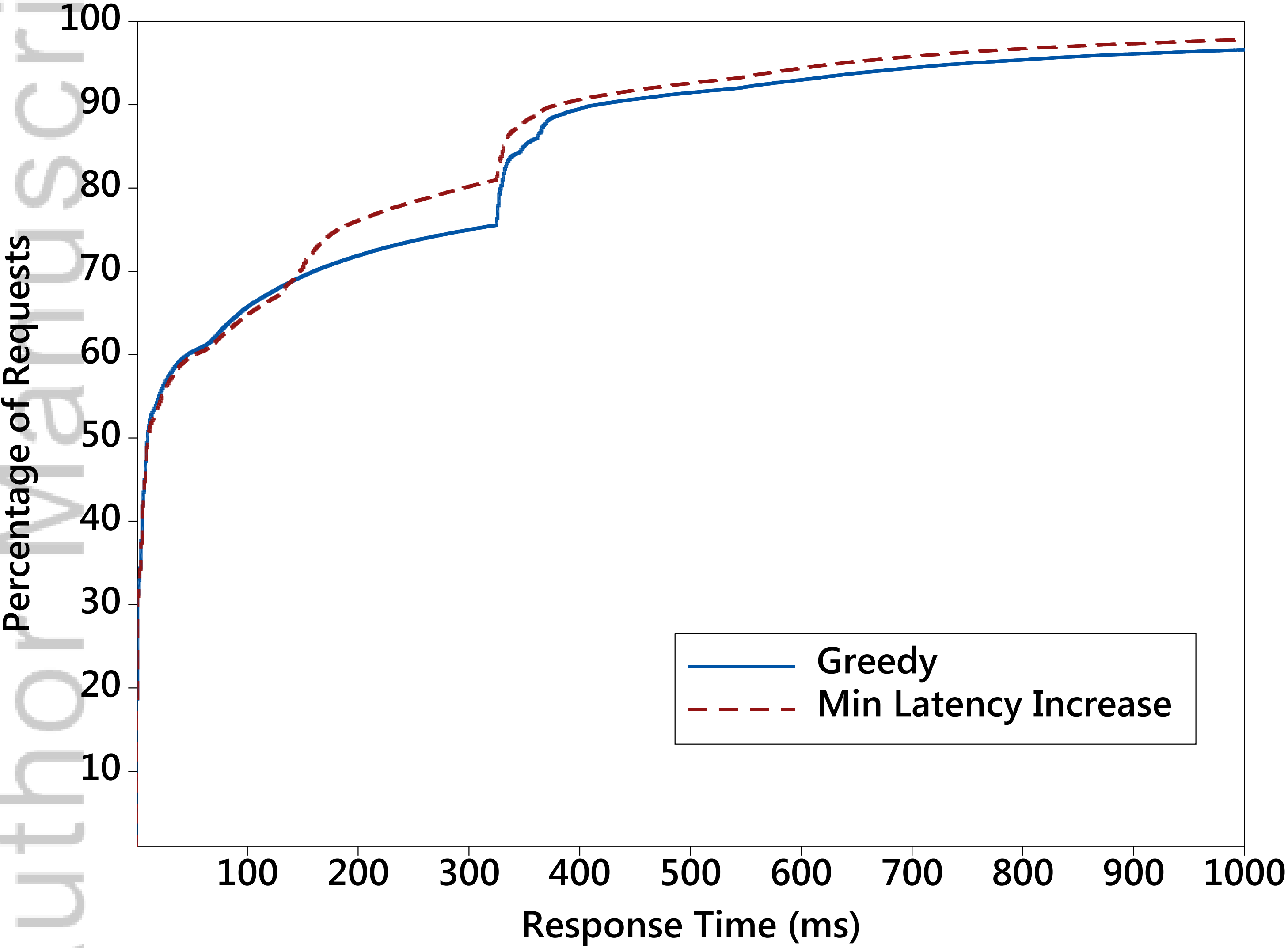

Percentage of Requests
100
90
80
70
60
50
40
30
20
10
0
0
200
400
600
800
1000
Response Time (ms)
Ireland with Ireland
Ireland with Ireland and Tokyo
Tokyo with Ireland and Tokyo


Percentage of Requests
0
10
20
30
40
50
60
70
80
90
100
Response Time (ms)
0
200
400
600
800
1000
Ireland with Ireland
Ireland with Ireland and Tokyo
Tokyo with Ireland and Tokyo


100
90
80
70
60
50
40
30
20
10
0
Percentage of Requests
0
200
400
600
800
1000
Response Time (ms)
Ireland with Ireland
Ireland with Ireland and Tokyo
Tokyo with Ireland and Tokyo


admission control
with Ireland
with Ireland and Tokyo
Percent
100
80
60
40
20
0
245 reqs/s
280 reqs/s
315 reqs/s
350 reqs/s
385 reqs/s


35000000
30000000
25000000
20000000
15000000
10000000
5000000
0
Nanoseconds
10 dc
20 dc
30 dc
40 dc
50 dc


100
90
80
70
60
50
40
30
20
10
Percentage of Requests
100
200
300
400
500
600
700
800
900
1000
Response Time (ms)
Greedy
Min Latency Increase

Author/s:
Qu, C; Calheiros, RN; Buyya, R

Title:
Mitigating impact of short-term overload on multi-cloud web applications through geographical load balancing

Date:
2017-06-25

Citation:
Qu, C., Calheiros, R. N. & Buyya, R. (2017). Mitigating impact of short-term overload on multi-cloud web applications through geographical load balancing. CONCURRENCY AND COMPUTATION-PRACTICE & EXPERIENCE, 29 (12), https://doi.org/10.1002/cpe.4126.

Persistent Link:
http://hdl.handle.net/11343/292613

File Description:
Accepted version