

August 01, 2011

ADuS: Adaptive resource allocation in cluster systems under heavy-tailed and bursty workloads

Zhen Li

Recommended Citation

Li, Zhen, "ADuS: Adaptive resource allocation in cluster systems under heavy-tailed and bursty workloads" (2011). *Electrical and Computer Engineering Master's Theses*. Paper 73. <http://hdl.handle.net/2047/d20002422>

This work is available open access, hosted by Northeastern University.

ADuS: Adaptive Resource Allocation in Cluster
Systems Under Heavy-Tailed and Bursty Workloads

A Thesis Presented

by

Zhen Li

to

The Department of Electrical and Computer Engineering

in partial fulfillment of the requirements
for the degree of Master in Science

in

Electrical Engineering

in the field of

Computer Engineering

Northeastern University
Boston, Massachusetts
August 2011

Abstract

A large-scale cluster computing system has been employed in the multiple areas by offering the pools of fundamental resources. How to effectively allocate the shared resources in a cluster system is a critical but challenging issue, which has been extensively studied in the past few years. Despite the fact that classic load balancing policies, such as Random, Join Shortest Queue and size-based policies, are widely implemented in actual systems due to their simplicity and efficiency, the performance benefits of these policies diminish when workloads are highly variable and heavily dependent. We propose a new load balancing policy ADUS, which attempts to partition jobs according to their sizes and to further rank the servers based on their loads. By dispatching jobs of similar size to the servers with the same ranking, ADUS can adaptively balance user traffic and system load in the system and thus achieve significant performance benefits. Extensive simulations show the effectiveness and the robustness of ADUS under many different environments.

Contents

1	Introduction	4
2	Background	6
2.1	Cluster System	7
2.2	Existing Policies	9
2.3	Workload Distribution	10
3	Motivation	13
3.1	Limitations of JSQ	15
3.2	Limitations of ADAPTLLOAD	16
4	New Load Balancing Policy: ADuS	17
4.1	ROLE ROTATION Policy	17
4.2	ADuS Policy	18
5	Performance Evaluation	21
5.1	Model	22
5.2	Performance of ROLE ROTATION	24
5.3	Performance Improvement of ADuS	25
5.4	Sensitivity to System Loads	28
5.5	Sensitivity to Coefficient of Variation and Temporal Dependence	30
5.6	Sensitivity to Network Size	34
5.7	Case Study: Real TCP Traces	35
6	Conclusion	41
7	Future Works	43

1 Introduction

Present day, large-scale cluster computing environments are being employed in an increasing number of application areas. Examples of these systems include High Performance Computing (HPC), enterprise information systems, data centers, cloud computing and cluster servers, which provide the pools of fundamental resources. In particular, a cloud platform, as a new and hot infrastructure, provides a shared "cloud" of resources as an unified hosting pool, which requires a central mechanism for resource provisioning and resource allocation based on multiple remote clients' demands [1, 2]. Typically, a large-scale cluster system contains a hierarchy of server nodes, which behaves as a single resource pool and is capable of running a separate emulation of an application, and a front-end dispatcher that is responsible for distributing the incoming jobs among the back-end server nodes. In our paper, the front-end dispatcher is featured as a redirect buffer without a central waiting queue while each server node has its own queue for waiting jobs and serves these jobs under the first-come first-serve (FCFS) queuing discipline.

A lot of previous studies have been focusing on developing load balancing policies for a large-scale cluster computing system over the past decades [3, 4, 5, 6, 7]. Examples of these policies include Join Shortest Queue (JSQ) and the size-based ADAPTLLOAD. When there is no a priori knowledge of job sizes and the job sizes are exponentially distributed, JSQ has been proven to be optimal [8]. However, prior research has shown that the job service time distribution is critical for the performance of load balancing policies. [9] evaluated how JSQ performs under various workloads by measuring the mean response times and demonstrated that the performance of JSQ clearly varies with the characteristics of different service time distributions. For

example, the optimality of JSQ quickly disappears when job service times are highly variable and heavy-tailed [10, 11].

Recently, size-based policies were proposed to balance the load in the system, only using the knowledge of the incoming job sizes. The literatures in [6, 11, 12] have shown that such size-based policies are optimal if one aims to achieve the minimum job response times and job slowdowns. The ADAPTLLOAD policy being a representative example of size-based policies, has been developed to improve average job response time and average job slowdown by on-the-fly building the histogram of job sizes and distributing jobs of similar sizes to the same server[12]. Nonetheless, [4] demonstrated that such size-based solutions are not adequate if the job service times are temporally dependent.

Based on different assumptions, other classic load balancing policies are developed under cluster and cloud computing environments. The Min-Min and the Max-Min algorithms that focus on the problem of scheduling a bag of tasks, assume that the execution times of all jobs are known [13]. Meta-schedulers, like Condor-G [14], rely on accurate queuing time prediction in scheduling jobs on computing grids. However, accurate predictions become more challenging in the current visualized and multi-tiered environments. Recently, techniques of advance-reservation and job preemption were presented to allocate jobs in cluster or grid systems [15]. Yet, extra system overheads and job queue disruptions could decrease the overall system performance.

In this paper, we propose a new load balancing policy ADUS, which adaptively distributes work among all servers by taking account of both *user traffic* and *system load*. In detail, ADUS ranks all servers based on their present system loads and updates job size boundaries on-the-fly, allowing the

dispatcher to direct jobs of similar sizes to the servers which have the same ranking. We expect that our new policy can improve the overall system performance by inheriting the effectiveness of both JSQ and ADAPTLOAD and meanwhile overcoming the limitations of these two policies.

Trace-driven simulations are used to evaluate the performance of ADUS. Extensive experimental results indicate that ADUS significantly improves the system performance - job response times and job slowdowns - up to 58% and 66%, respectively, under heavy-tailed and temporal dependent workloads. Sensitivity analysis with respect to system loads, variation and temporal dependence in job sizes, and the number of servers in system demonstrate that ADUS is effective and robust in various environments. We also use the case studies of three real TCP traces to further validate the benefits of ADUS in actual systems, where ADUS again achieves a clear improvement of up to one order of magnitude.

The remainder of this paper organized as follows. In Section 2 we give the background knowledge of cluster system and list a set of existing load balancing policies. By analysing the limitations of such classic policies under highly variable and heavily dependent workloads in Section 3, we are motivated to propose our new algorithms and improve it in Section 4. Section 5 shows experimental results including extensive sensitivity analysis to various system conditions and three real TCP traces. Section 6 summarizes our contributions and finally future work are outlined in Section 7.

2 Background

In this section, we first give an overview of cluster systems, including the concept, the history and the development. We then summarize several classic load balancing policies which are widely accepted across both research

areas and industry implementations. We further discuss workload distributions and show the existence of highly variable and heavily dependent workloads. We also have simulation result to verify their impact on system performance.

2.1 Cluster System

A cluster system contains a group of homogeneous or heterogeneous service nodes, which are linked together and behave as a single virtual server in many respects. Either a centralized front-end dispatcher or a decentralized decision making algorithm is responsible for distributing incoming requests among server nodes. The performance of a cluster system is typically much more cost-effective than that of a single computers with comparable speed or availability. Figure 1 illustrates the model of a simple clustered system where a set of homogeneous servers behave as a single server. each server has its own waiting queue and handles incoming works with the same processing rate μ . Jobs and requests first arrive at a dispatcher with different arrival rates and then are forwarded to the appropriate server based on a certain algorithm.

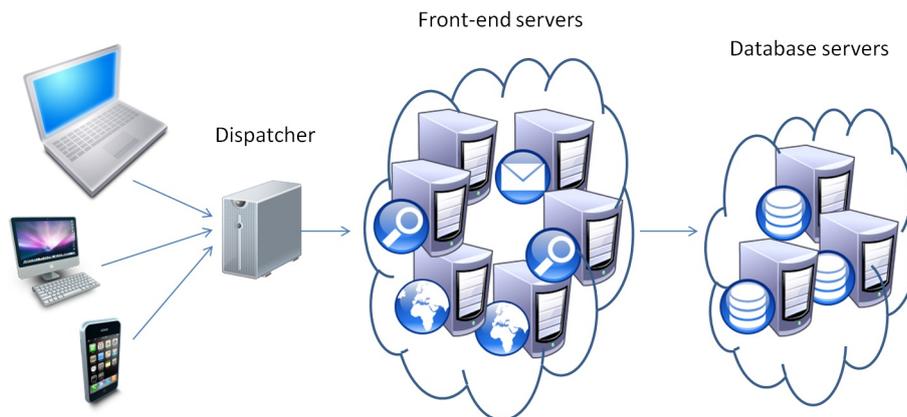


Figure 1: Model of a cluster system.

The exact date of the first time that people mentioned the concept of cluster system is unknown but should not be later than 1960s. Amdahl's Law [16] which was published in 1967 defined the engineering basis for cluster computing. It described mathematically the speedup one can expect from parallelizing for any given tasks running on a parallel architecture. Then the study of customer-built clusters began from the early 1970s as TCP/IP protocols are created and formalized. The first commercial clustering product is Attached Resource Computer Network (ARCnet) [17], developed in 1977, however, ARCnet was not a commercial success until Digital Equipment Corporation (DEC) released their VAXcluster product in 1984 [18]. In order to take the advantages of parallel processing while maintaining data reliability, VAXcluster supports both parallel computing and shared file systems and peripheral devices. This product is still available on HP's Alpha and Itanium systems [19]. In today's climate, cluster systems are established as an industry standard for developing client-server applications. As the development of cluster related techniques and the huge increase of market demands, companies and research labs have to provide more powerful supercomputers - a high performance compute cluster - with more service nodes, faster speed and larger throughput. According to the report of TOP500¹ organization who ranks the 500 fastest supercomputers all over the world semiannually, to the date of January 2011, the top supercomputer is the Tianhe-1A, located in Tianjin, China, with performance of 2566T Flops (floating point operations per second). Figure 2 gives an example of a large scale clustering computer system of Amazon [20].

¹TOP500 is a collaboration between the University of Mannheim, the University of Tennessee, and the National Energy Research Scientific Computing Center at Lawrence Berkeley National Laboratory



Figure 2: Example of a cluster system of Amazon Elastic Compute Cloud [21].

2.2 Existing Policies

The performance of cluster systems described above highly depends on the load balancing algorithms, i.e., how to effectively allocate the shared resources in that cluster system. Now we summarize the following classic load balancing policies: *Random*, *Round Robbin*, *Join Shortest Queue (JSQ)* [22], *Join Shortest Weighted Queue (JSWQ)*, and a size-based policy *ADAPT-LOAD* [12]

- ***Random***: this policy randomly determines the server that will be allocated for each incoming job, treating all jobs with the same priority. Since this policy does not require keeping any history of jobs, it is widely implemented in real systems because of its simplicity.
- ***Round Robbin***: this policy assigns each request to different servers in a circular order, handling all request without any priority. Round Robbin scheduling algorithm is simple and easy to implement. Since this policy does not consider any characteristic of requests, it would cause unbalanced load in each server when job size varies a lot.
- ***Join Shortest Queue(JSQ)***: when a new job arrives, it believes that

a server with the least number of jobs in the queue (i.e., the shortest queue) would complete that job in the shortest time. Hence, the newly arrived job should be assigned to that particular server.

- ***Join Shortest Weighted Queue (JSWQ)***, the difference between JSQ and JSWQ is that JSWQ calculates queue length not by the total number of jobs but instead by the total weight of all jobs in the queue. This algorithm provides a more accurate way for balancing workloads than that of JSQ since each job is assigned to the least loaded server. But JSWQ assumes to know the exact size of each job before it is completed.
- **AdaptLoad**: it builds the histogram of job sizes and partitions the work into equal areas. Each server will then dedicate to processing jobs with similar sizes. The boundaries of each size interval are adjusted dynamically according to the past history. ADAPTLLOAD has been proven to achieve high performance by reducing the number of small jobs from waiting behind large ones.

2.3 Workload Distribution

Each of those classic policies is widely used because of its simplicity and efficiency, however, there is no universal policy which claims the best algorithm in all circumstances since the workload distribution is also a key factor. Former studies believed that the requests follow exponential distribution and built policies based on this assumption. As research goes deeper, people realized that it is not always the case especially in network traffic where jobs exhibit long tail characteristics with temporal dependence.

The temporal dependent structure can be captured by autocorrelation function (ACF). Autocorrelation is a mathematical measurement of correla-

tion coefficient in statistics and probability theory, it described the similarity of one process over different points of time [23]. Consider a set of random variables X_n , the following equation defines the autocorrelation $\rho_{X(k)}$ of X_n for different time lags k :

$$\rho_{X(k)} = \rho_{X_t, X_{t+k}} = \frac{E[(X_t - \mu)(X_{t+k} - \mu)]}{\sigma^2}, \quad (1)$$

where “ E ” is the expected value, μ is the mean and σ^2 is the common variance of X_n . The parameter k is called the lag and represents the distance between the observed value at different time and itself, i.e., $X_{(t+k)}$ and X_t . The values of $\rho_{X(k)}$ fall in the range $[-1, 1]$. A positive value of $\rho_{X(k)}$ indicates there exists autocorrelation of X_i while a negative value implies anti-autocorrelation. A high absolute value of X_i determines the process has strong temporal locality, i.e., the following variables have a high probability of having similar values with previous ones. We consider the process as high autocorrelation case if the value of X_i is greater than 0.4. An independent process is determined only if $\rho_{X(k)} = 0$ at lag k .

In [23], the authors observed the existence of correlated (dependent) flows over a wide range of internet traffic including WEB servers, E-mail servers, User Accounts servers and Software Development servers in both inter-arrival times (arrival process) and service times (service process). Their work also examined the different impacts of autocorrelation under open and close system. The autocorrelation induced from any node of an open system will only affect the performance of downward nodes. On the contrary, if autocorrelation exists in any tier of a close system, it will propagate across the entire system. Although it tries to balance the load among all queues, their experiment results show that the system still suffers severe performance degradation.

Consider a simple open system with only one service node, we use a

2-state Markovian-Modulated Poisson Process (MMPP)² to model an autocorrelated process [25]. We evaluate the following two scenarios:

- **Scenario 1:** the arrival rate is exponentially distributed with mean $\lambda^{-1} = 2$, while the service rate is drawn from a 2-state MMPP with mean $\mu^{-1} = 1.5$, squared coefficient of variation $SCV = 6$, and autocorrelation function $ACF = 0.4$ at lag 1.
- **Scenario 2:** the arrival rate is drawn from a 2-state MMPP with mean $\lambda^{-1} = 2$, $SCV = 6$ and $ACF = 0.4$ at lag 1, while the service rate is exponentially distributed with mean $\mu^{-1} = 1.5$.

We consider the simplest queuing algorithm first-come first-serve (FCFS). All simulations are done in a one million sample space. Figure 3 illustrates the autocorrelation for both scenario 1 and scenario 2.

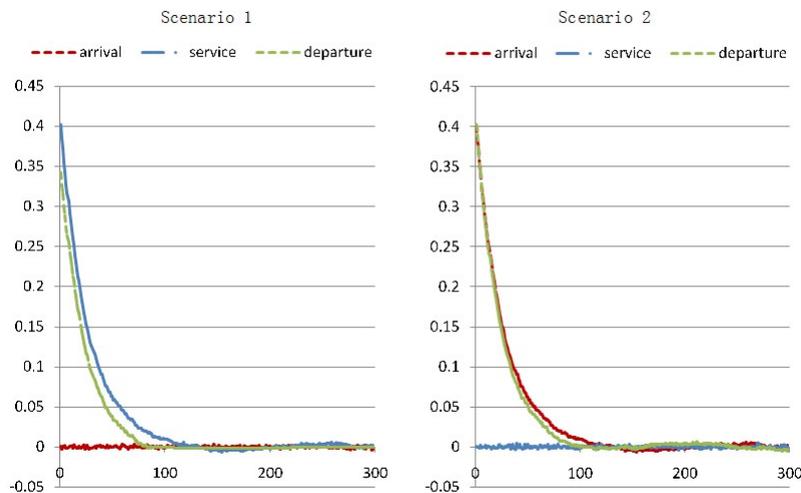


Figure 3: ACFs in arrival, service and departure process for both scenarios

²Markov-modulated Poisson Process (MMPP) is a special case of the Markovian Arrival Process (MAP) [24], which is used here to generate temporally dependent flows because it is analytically tractable.

We maintain the same utilization level (75%) which can be considered as a heavy system load condition for each scenario. In scenario 1, ACF comes from the service process, so the departure time has the similar ACF feature as the service process does. In scenario 2, ACF generates from the beginning, i.e., arrival requests. As the figure suggests, ACF has been propagated to departure jobs. We also record mean response time and mean queue length for both scenarios, see Table 1.

	response time (s)	queue length
scenario 1	241	398
scenario 2	120	198
base case	3.28	1.64

Table 1: Mean response time and queue length for both scenarios

In order to make the comparison, we introduce our base case which has exponentially distributed arrival and service processes with mean $\lambda^{-1} = 2$, $\mu^{-1} = 1.5$, respectively for keeping utilization level the same. Both response times and queue lengths with ACF (no matter in arrival process or service process) are much higher than the basecase. When ACF is in the service process, (i.e., Scenario 1) its performance becomes even worse. All these results above give us concrete evidence that ACF will strongly affect system performance. Therefore, new policies which can prevent performance from degradation under autocorrelation are critical.

3 Motivation

In this section, we first give an overview of four existing load balancing policies which are widely used in cluster systems. By analyzing these policies

under highly variable and/or heavily dependent workloads, we then illustrate their limitations in terms of performance matrices, which motivates the design of a new efficient policy proposed in this paper.

Here, We consider the following four load balancing policies in the performance evaluations: (1) *Random*, randomly determines the server that will be allocated for each incoming job; (2) *Join Shortest Queue* (JSQ)[22], when a new job arrives, it believes that a server with the least number of jobs in the queue (i.e., the shortest queue) would complete that job in the shortest time. Hence, the newly arrived job should be assigned to that particular server; (3) *Join Shortest Weighted Queue* (JSWQ), the difference between JSQ and JSWQ is that JSWQ calculates queue length not by the total number of jobs but instead by the total weight of all jobs in the queue; and a size-based policy (4) *ADAPTLLOAD* [12], it builds the histogram of job sizes and partitions the work into equal areas. Each server will then dedicate to processing jobs with similar sizes. The boundaries of each size interval are adjusted dynamically according to the past history. *ADAPTLLOAD* has been proven to achieve high performance by reducing the number of small jobs from waiting behind large ones.

Despite the fact that such classic load balancing policies are widely used because of their simplicity (e.g., *Random*) and their efficiency (e.g., *JSQ* and *ADAPTLLOAD*), we found that these policies exhibit performance limitations under the workloads with high variability and strong temporal dependence. Here, we exemplify such performance limitations by evaluating the *JSQ* and *ADAPTLLOAD* policies.

3.1 Limitations of JSQ

Since JSQ always sends a job to the shortest (least loaded) queue, there is a high probability such that small jobs are stuck behind large ones, which may consequently cause performance degradation when the distribution of job sizes is highly variable. To verify this limitation, we measure the fraction of small jobs that are stuck in the queue due to large ones, i.e., the ratio of the number of small jobs which wait behind large jobs to the total number of small jobs. Here, we use $T = \mu^{-1}(1 + 2CV)$ to classify small and large jobs, where μ^{-1} is the mean of job sizes and CV is the coefficient of variation of job sizes.

In this experiment, the job interarrival times are exponentially distributed with mean rate $\lambda = 2$ while the job sizes are drawn from a 2-state Markovian-Modulated Poisson Process (MMPP) with mean rate $\mu = 1$ and squared coefficient of variation $SCV = 20$. We also consider 4 homogeneous server nodes in the system. By scaling the mean processing speed of each server node (i.e., $\mu_p = 1$), we obtain 50% utilization per server node. The sample space in this experiment is 1 million.

Table 2 shows the measured results under the three load balancing policies. We first observe that among the four policies, Random as expected obtains the largest fraction of small jobs which have to wait behind large ones in the queue. Observe also that although JSQ is well known for its optimal performance, there are about 23% of small jobs being stuck in the queue due to large ones, which thus degrades the performance of small jobs and further deteriorates the overall system performance. By dispatching jobs based on their sizes, the size-based policy ADAPTLLOAD successfully avoids the majority of small jobs (i.e., about 95%) waiting behind large ones in the queue.

Policy	Random	JSQ	ADAPTLLOAD
Ratio	42.53%	22.93%	4.78%

Table 2: Fraction of small jobs that are stuck behind large ones when the job sizes have high variability, i.e., $SCV = 20$.

3.2 Limitations of AdaptLoad

We investigate the performance of ADAPTLLOAD under the workload with high variance and strong temporal dependence in job sizes by measuring the queuing times of small and large jobs separately. Table 3 shows the measured results, where we use the same experimental setting as shown in the above but introduce the temporal dependence into job sizes. We observe that the effectiveness of ADAPTLLOAD dramatically deteriorates such that its performance becomes comparable to the worst one (i.e., Random). This essentially motivates that simply directing jobs with similar size to the same server is not sufficient under temporally dependent workloads. We interpret that because of the temporal dependence in job sizes, it becomes highly possible that the jobs with similar sizes are clustered together and arrive the system in bursts. If the policy simply assigns such jobs to the same server, then that particular server will experience a sudden load spike during a short period, causing long waiting times for those jobs and thus degrading the overall system performance.

In summary, both JSQ and ADAPTLLOAD have the limitations under highly variable and/or strongly dependent workloads, which thus cause negative impacts on system performance. Motivated by this problem, we present a new load balancing policy which attempts to address the limitations of both JSQ and ADAPTLLOAD and thus achieve visible performance improvement.

Policy	Overall	Small	Large
Random	22.19	21.72	34.46
JSQ	11.96	11.45	25.62
ADAPTLLOAD	14.73	13.98	34.59

Table 3: Overall queuing times and individual queuing times for small and large jobs under three load balancing policies when the job sizes are highly variable and strongly dependent.

4 New Load Balancing Policy: ADuS

In this section, we first present our original algorithm *role rotation*, then we come up with an improvement policy called ADuS which successfully overcomes the limitations of existing policies (JSQ and ADAPTLLOAD) under high variable and strong dependent workloads.

4.1 Role rotation Policy

Role rotation is also a size based policy which is similar to ADAPTLLOAD. The main idea behind this policy is that under the circumstance of ADAPTLLOAD if a burst of jobs comes, then there must be one server being very busy and some other servers being relatively idle. Our goal is instead of keeping sending the incoming requests to the busy server; we send them to other less busy or idle servers. Unlike ADAPTLLOAD that each server serves relatively the same type (size) of jobs, in our role rotation policy, each server will serve different kinds of jobs in terms of size at different time. People can easily come up with these two questions: when to rotate? Where to rotate?

For the first question, it actually depends on how we detect burstiness. In this paper, we use average response time (avg_{rt}) and average slowdown

(avg_{slid}) as the measurement metrics. More specifically, for every C requests, we compute the current average response time and current average slowdown. Then we compare them to the overall response time and slowdown of requests arrived before. If any of these two parameters becomes worse, then we consider that the system performance is degraded due to autocorrelation. We further predict that there will still exist autocorrelation in inter-arrival times of the next C requests. Hence, we need to invoke our new policy. It brings us to the second question: where to rotate?

Remember one advantage of size based policies is that they prevent small jobs from waiting behind large jobs. We also want to take this advantage. So the algorithm always assigns the requests to the next server. For example, if a job should be assigned to server i before rotation, then it will go to server $i + 1$ in the current round. Figure 4 gives the high level idea of ROLE ROTATION algorithm.

4.2 ADuS Policy

During the experiments we find that the improvement of ROLE ROTATION is not that clear, see Section 5 for the detailed experimental results and the analysis. So, we now turn to present our new improved load balancing policy, called ADuS, which adaptively distributes work among all servers by taking account of both user traffic and system load, aiming at inheriting the effectiveness of JSQ and ADAPTLOAD policies and meanwhile overcoming the limitations of these two policies as shown in the previous section. The main idea of ADuS is to on-the-fly rank all the servers according to their system loads and dynamically tune the job size boundaries based on the current user traffic loads. ADuS then directs the jobs whose sizes locate in the same size boundary to the corresponding servers which are in the same

```

Algorithm: ROLE ROTATION
begin
1. initialization
   a. initialize shift step:  $shift_{stp} \leftarrow 0$ ;
   b. initialize size boundaries:  $[0, b_1), [b_1, b_2), \dots, [b_{n-1}, \infty)$ ;
2. for each incoming job
   Select server:  $i \leftarrow (\text{ADAPTLLOAD} \rightarrow \text{SelectServer} + shift_{stp})$ 
3. upon the completion of every  $C$  jobs
   a. compute current response time and slowdown
   b. if performance is worse than the overall performance
       then shift load to the next server;
4. update the overall performance
end

```

Figure 4: The high level description of ROLE ROTATION.

ranking. Based on the observation that the majority of jobs in a heavy-tailed workload is small, ADUS always gives small jobs high priority by sending them to the highly ranked (i.e., less loaded) servers.

Recall that ADAPTLLOAD evenly balances the load across the entire cluster by determining boundaries of jobs sizes for each server. ADUS adopts such boundaries where the histogram of job sizes is built and partitioned into N equal areas for N servers in the system. Then, the i^{th} server is responsible for the work locating in the i^{th} area, which allows the decreasing variation in job sizes (or job service times) on each server. Consequently, the proportion of small jobs that wait behind long ones is reduced as well and the user traffic is thus well balanced among all servers.

However, as we discussed in Section 3, simply separating requests according to their sizes is not sufficient under temporally dependent workloads.

Therefore, ADUS periodically ranks all servers based on their present system loads and keeps sending the incoming jobs of similar sizes to a server with the same ranking instead of the same server which might be overloaded by the previous arrived jobs. Given N servers $\{S_1, \dots, S_N\}$ and N boundaries of jobs sizes $[0, b_1), [b_1, b_2), \dots, [b_{n-1}, \infty)$. For every window of C jobs, ADUS sorts all the servers in an increasing order of their queue length and then gets a priority list S' . Then, the i^{th} server S'_i in the priority list will be dispatched to the incoming jobs in the next window which have the sizes within the i^{th} boundary $[b_{i-1}, b_i)$. It follows that the first server S'_1 with the least load in the priority list will then serve small yet a large number of jobs while the last server S'_N that became heavy loaded due to serving small ones during the last window then starts to serve large but few jobs. As a result, ADUS further successfully balances the system load (i.e., queue length) among all N servers and thus significantly diminishes the proportion of similar sized jobs (especially small ones) being queued on the same server during a short period.

Figure 5 gives the high level description of our new policy ADUS. We remark that the window size C indicates how often ADUS re-ranks all servers. A large window size updates the rankings of servers in a lower frequency, which introduces less computational overhead but might provide slower reactions to frequently changed workloads. In contrast, a small window size can quickly adapt to workload changes, improving the system performance, but needs a higher computational cost in the meanwhile. Thus, we conclude that selecting an appropriate window size is critical to ADUS' performance as well as its computational overheads. In our experiments, we set C to different values under various workloads and find that the best performance is obtained when the window size ranges from 50 to 200. Thus, we consider

```

Algorithm: ADuS
begin
1. initialization
   a. priority list:  $S' = \{S'_1, \dots, S'_N\}$ ;
   b. size boundaries:  $[0, b_1), [b_1, b_2), \dots, [b_{n-1}, \infty)$ ;
2. upon the completion of every  $C$  jobs
   a. sort all  $N$  servers in an increasing order of queue length
      and update the priority list  $S'$ ;
   b. update the size boundaries such that the work is
      equally divided into  $N$  areas;
3. for each arriving job
   a. if its job size  $\in [b_{i-1}, b_i)$ 
      then direct this job to server  $S'_i$ ;
end

```

Figure 5: The high level description of ADuS.

the window size $C = 100$ in all the experiments throughout this paper. How to dynamically adjust C will be studied in our future work.

5 Performance Evaluation

In this section, representative case studies are presented to illustrate the effectiveness and the robustness of ADuS. We use trace-driven simulations to evaluate the performance improvement of ADuS in a homogeneous clustered system with N (FCFS) servers. Various load balancing algorithms are then executed by a load dispatcher which is responsible for distributing the arrivals to one of N servers.

For all simulations, the specifications of a job include job arrival times

and job sizes, which are generated based on the specified distributions or real traces. Throughout all experiments, the N servers have the same processing rates $\mu_p = 1$. In the following subsections, we first use the synthetic traces to show the performance of ADUS under difference environments and later validate its effectiveness by the case study using a real TCP trace.

5.1 Model

Consider a simple open system model in Figure 6, a set of homogeneous service nodes constitute a cluster system, each of which has its own waiting queue. All the incoming jobs first arrive at the dispatcher and then are assigned to appropriate server based on certain load balancing policy.

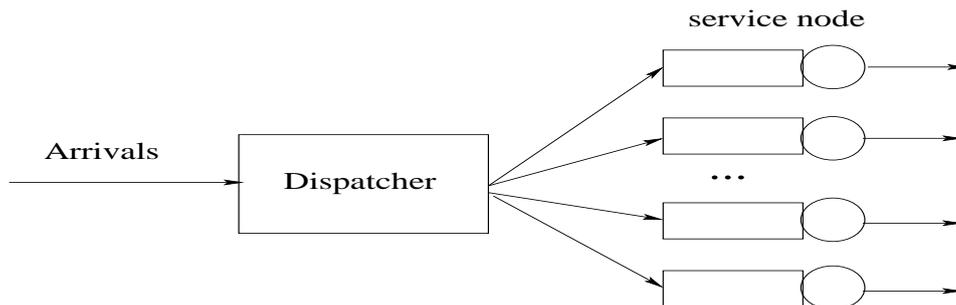


Figure 6: Model of an open system.

In our experiments, we generate two kinds of traces to simulate this procedure: *arrival process* and *service process*. Arrival process simulates the behavior of incoming jobs, representing inter-arrival time for each job, i.e., when the next job enters to the system. Since we consider no autocorrelated nor highly variable workload for arrival process, an exponential distribution function is used to create our arrival process. We adjust only the mean value in order to obtain different utilization levels (low, medium and high load). The service process simulates file size of each job, i.e., the time consumption for a server to complete this request. However, service process

has autocorrelation and is high variance, as mentioned in previous section, we use a 2-state MMPP to model autocorrelated service process, in particular, an existing MATLAB tool is used. For example, a command of `map_mmpp2(1, 20, -1, 0.40)` will output two matrices of equation (2) where the mean service rate $\mu = 1$, squared coefficient of variation ($SCV = 20$), and autocorrelation at lag 1 ($ACF = 0.40$) with the automatic minimization of skewness (-1). Another command `map_mmpp2(1, 10, -1, 0.25)` produces the distribution with the same mean service rate $\mu = 1$ but different variance and autocorrelation ($SCV = 10$ and $ACF = 0.25$), the matrices are shown in equation (3). We further conduct the service process based on those matrices.

$$\begin{aligned}
 D_0 &= \begin{bmatrix} -0.09538513947 & 0.01401612151 \\ 20.39414714380 & -1358.04747200364 \end{bmatrix} \\
 D_1 &= \begin{bmatrix} 0.08136901796 & 0 \\ 0 & 1337.65332485984 \end{bmatrix}
 \end{aligned} \tag{2}$$

$$\begin{aligned}
 D_0 &= \begin{bmatrix} -0.18800839854 & 0.07820493587 \\ 99.09534878315 & -1228.08471887438 \end{bmatrix} \\
 D_1 &= \begin{bmatrix} 0.10980346267 & 0 \\ 0 & 1128.98937009123 \end{bmatrix}
 \end{aligned} \tag{3}$$

Once we have both traces of arrival process and service process, the last step is to merge them together. A segment of the final single input file is shown in Figure 7 where the first and the last column represents inter-arrival time (arrival process) and job size (service process), respectively. The second column denotes job class, since we consider only one job type, this value does not affect anything during our experiments.

0.541194	1	0.000376
0.314820	1	0.001109
0.006880	1	0.001890
0.491181	1	0.000266
0.052866	1	0.000110
0.135518	1	0.000239
0.047960	1	0.000635
0.023295	1	0.000918
0.191908	1	0.000583
0.043791	1	12.550140
0.849889	1	29.709837
0.651217	1	5.587562
0.454988	1	13.354270
0.177612	1	4.953452
0.156863	1	5.594913
0.290592	1	9.010496
0.067471	1	11.347544
0.239952	1	6.601032

Figure 7: A segment of input file.

5.2 Performance of Role Rotation

Our preliminary results in Figure 8 show that ROLE ROTATION has little improvement compared to ADAPTLLOAD under both low (utilization = 30%) and high (utilization = 60%) system loads. We speculate that is due to the tradeoff between moving requests to less busy server and having small jobs waiting behind large ones. Shifting jobs will avoid the impact of autocorrelation but will increase the chance that small jobs get stuck after large ones at the same time. In some cases, the system has lower ACF, if we continue to rotate under this circumstance, we do not reduce ACF as much as the increase of negative impact of job size. We then propose two ways to improve it. The first one is that the policy somehow can make a clever decision on whether rotate or not. We may set some threshold parameters,

rotation occurs only when the experimental data exceed our threshold. In this case we can make sure that the advantage of removing ACF is greater than the disadvantage of having small jobs waiting. As a result the overall performance won't degrade. The second method is we give some restriction to the large jobs so that small jobs can get service quickly. This is exactly what the improved algorithm ADUS does. The rest of this section shows ADUS has markedly improvement under variety of system loads.

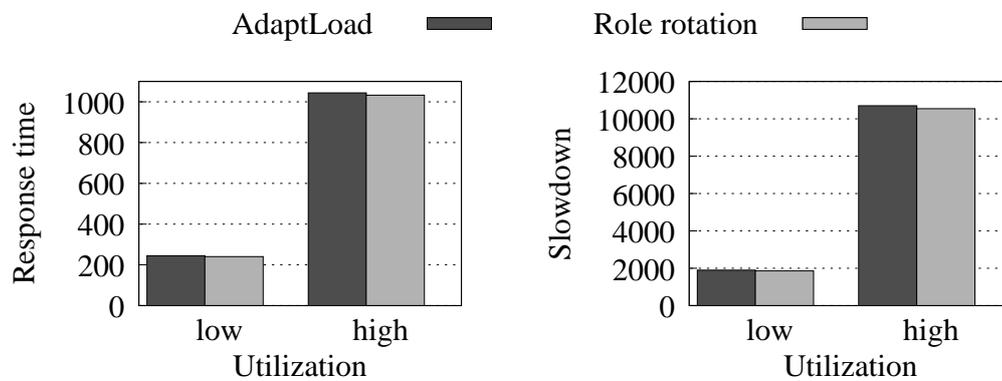


Figure 8: Performance of ROLE ROTATION.

5.3 Performance Improvement of ADUS

We first produce an experiment where the job inter-arrival times are exponentially distributed with mean $\lambda^{-1} = 0.5$ while the job sizes (i.e., the service process) are drawn from a MMPP(2) with the mean equal to $\mu^{-1} = 1$, squared coefficient of variation equal to $SCV = 20$, and autocorrelation function (ACF) at lag 1 equal to 0.40. Therefore, high variability and temporal dependence are injected into the workload, i.e., the service process. Also, we consider $N = 4$ homogenous server nodes in the cluster such that the average utilization levels at each server node are $\rho = 50\%$. Sensitivity to the most important experiment parameters is explored in the next subsections. The sample space in all simulations is 1 million jobs. Simulations stop only

Policy	Random	JSQ	ADAPTLLOAD	ADUS
$Resp_{avg}$	23.17	12.95 (44%)	15.72 (32%)	9.63(58%)
$SLW_{avg}(*10^3)$	175.54	89.40 (49%)	111.35 (36%)	59.24 (64%)
$Resp_{small}$	21.97	11.70 (46%)	14.23 (35%)	7.99 (64%)
$SLW_{small}(*10^3)$	182.18	92.79 (48%)	115.56 (36%)	61.67 (66%)
$Resp_{large}$	54.91	46.07 (16%)	55.03 (-1%)	53.07 (3%)
SLW_{large}	3.03	2.51 (17%)	2.94 (3%)	2.70 (11%)

Table 4: Overall performance and individual performance of four policies when $\lambda = 2, \mu = 1, \mu_p = 1, SCV = 4, ACF = 0.40$, and $\rho = 50\%$.

after all the jobs have been used.

Table 4 shows the system performance under four policies including Random, JSQ, ADAPTLLOAD and ADUS. Here, we measure the mean job response times (i.e., the summation of waiting times in the queue and service times) and the mean job slowdown (i.e., job response times normalized by job service times). In order to evaluate the individual performance, we further show the corresponding mean response times and mean slowdowns of small and large jobs respectively, where $T = \mu^{-1}(1 + 2CV)$ is used for the size classification. The relative improvement with respect to Random is also given in parenthesis, see Table 4.

We observe that compared to Random, both ADAPTLLOAD and JSQ improve the overall system performance, while our new policy ADUS outperforms among all the policies, with 58% and 66% relative improvement on average response time and average slowdown, respectively. Observer also that ADUS achieves the best performance for small jobs with respect to both response time and slowdown. We interpret the significant performance

improvement as an outcome of *always* assigning small jobs to the least loaded servers (i.e., the 1st rank). Furthermore, the performance of large jobs is improved as well, although not as significant as small ones.

We further investigate the tail distribution of response time and slowdown under the four policies. Figures 9 and 10 plot the complementary cumulative distribution function (CCDF) of overall and individual response times and slowdowns, respectively. We observe that under ADUS the majority (about 99.4%) of jobs experience faster response times and almost all jobs have smallest slowdowns. Figure 10 further shows that ADUS benefits small jobs by avoiding them waiting after large ones, see plots (a) and (b). On the other hand, due to its unfairness to large ones by sending them to high loaded servers, ADUS degrades the performance of medium and large jobs, see plot (c) and (d). Fortunately, the proportion of large jobs is quite small. Such performance penalty does not significantly affect the overall system performance.

In addition, we measure the performance metrics shown in Tables 2 and 3 under ADUS and find that only 3.99% of small jobs being stuck behind large ones, and the queueing times of small and large jobs are 7.74 and 32.62, respectively. This further gives an explanation of the performance improvement by using the ADUS policy.

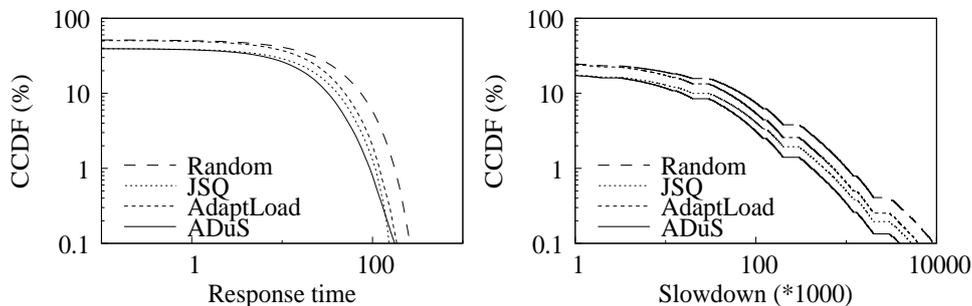


Figure 9: CCDFs of overall response times and slowdowns.

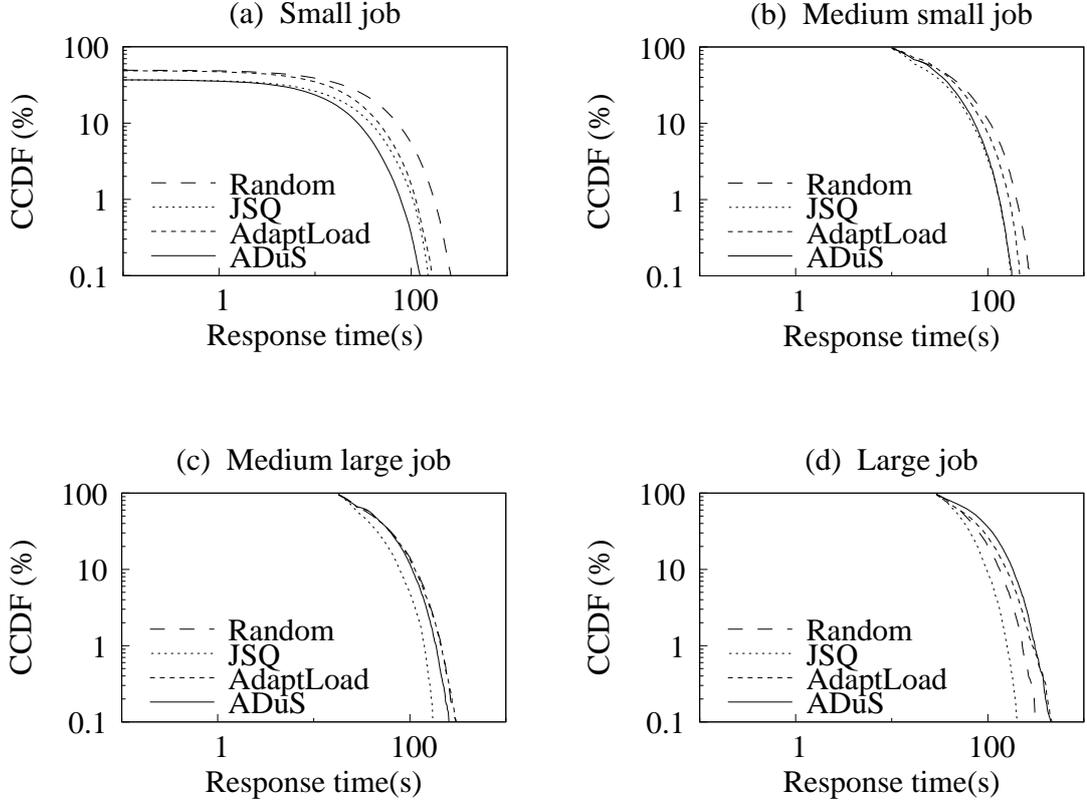


Figure 10: CCDFs of response times for small, medium small, medium large, and large jobs.

5.4 Sensitivity to System Loads

Now we turn to evaluate the robustness of ADuS under different experimental parameter settings. We first focus on the sensitivity analysis of various system loads. This is extremely important to understand the performance benefit of ADuS as the system reaches critical congestion. Here, we scale the mean arrival rates to obtain utilization levels $\rho = 20\%$, 50% and 80% , while keeping the other parameters the same as the experiment in Section 5.3.

Figure 11 illustrates the average performance (e.g., response time and slowdown) as a function of system utilizations under the four load balancer. The relative improvement with respect to Random is also shown on each

bar. We further plot the CCDF of response times under three different utilization levels in Figure 12. Consistent to the results shown in Section 5.3, ADUS achieves the best performance (i.e., fastest response times and smallest slowdown) under all the three utilization levels. For example, under the low system load (e.g., $\rho = 20\%$), ADUS performs similar as JSQ, improving the performance by around 50%. As the load increases to $\rho = 80\%$, the performance improvement under ADUS becomes more visible and significant.

The CCDF results in Figure 12 further validate the effectiveness of ADUS under different system loads. Random achieves the worst performance for all the jobs in the three experiments. Although compared to JSQ, our ADUS policy has a longer tail due to its unfairness to large jobs, the number of penalized jobs amounts to less than 1% of the total. That is, the majority of jobs have less response time under ADUS. Furthermore, such a long tail becomes shorter and finally disappears as the system becomes heavy loaded.

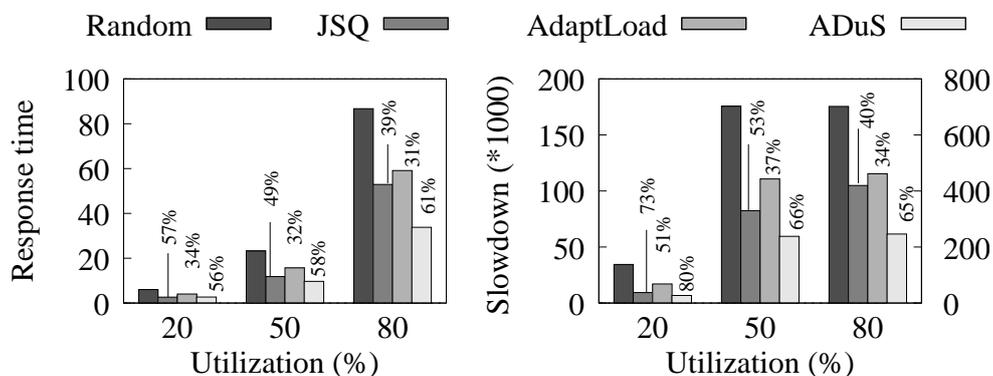


Figure 11: Response times and slowdowns under different utilizations when $\lambda = 2, \mu = 1, \mu_p = 1, SCV = 20$ and $ACF = 0.40$. In order to present all the results under those three utilizations in one single figure, we use a dual-Y-axis method to plot slowdown, where the $\rho = 80\%$ case uses the right side y axis labels.

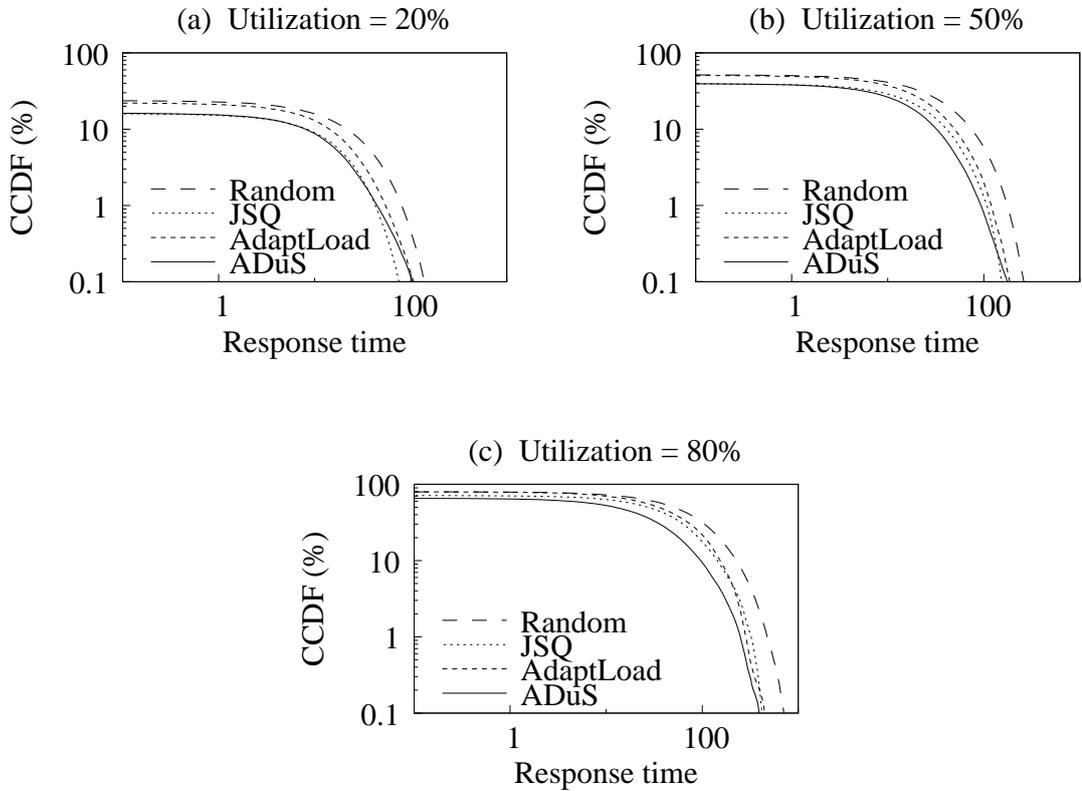


Figure 12: CCDFs under different utilizations when $\lambda = 2$, $\mu = 1$, $\mu_p = 1$, $SCV = 20$ and $ACF = 0.40$.

5.5 Sensitivity to Coefficient of Variation and Temporal Dependence

Now, we analyze the effect of both coefficient of variation and temporal dependence on policy performance as they are two important characteristics in the job size distribution. Figure 13 and Figure 15 show the corresponding experimental results under the workloads with different coefficient of variations and different temporal dependence profiles, respectively.

We first conduct experiments with various SCV s (e.g., 10, 20, and 40) of the job sizes, but always keeping the same mean. As shown in Figure 13, both ADuS and JSQ obtain 30%–40% performance improvement compared

to Random when the job sizes are not highly variable. However, as the variability in job sizes increases (e.g., $SCV = 40$), the workload contains few but extremely large jobs, which unfortunately diminishes the effectiveness of JSQ. It becomes highly likely that under JSQ small jobs may have to wait behind several extremely large jobs, resulting in long response time as well as long slowdown. While our new policy is able to avoid this situation by assigning jobs with similar size to the same ranked server and thus achieves better performance improvement. Additionally, the tail distributions of response times in this set of experiments are qualitatively the same as we show in Figure 12, which are then omitted here in order to save the space.

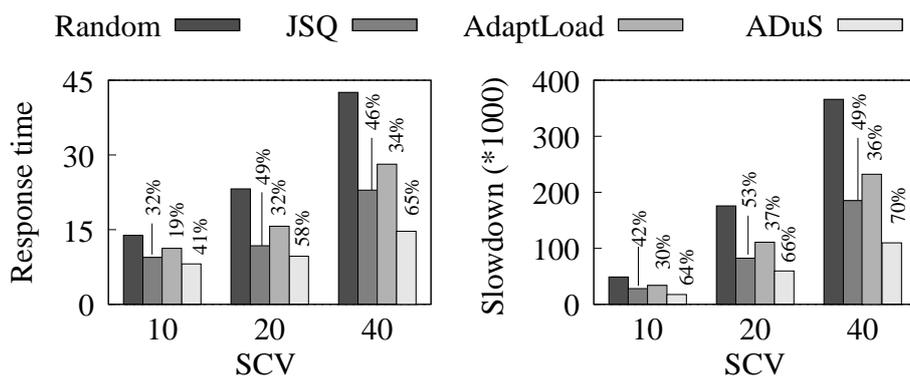


Figure 13: Response times and slowdowns under different SCVs when $\lambda = 2, \mu = 1, \mu_p = 1, \rho = 50\%$, and $ACF = 0.40$.

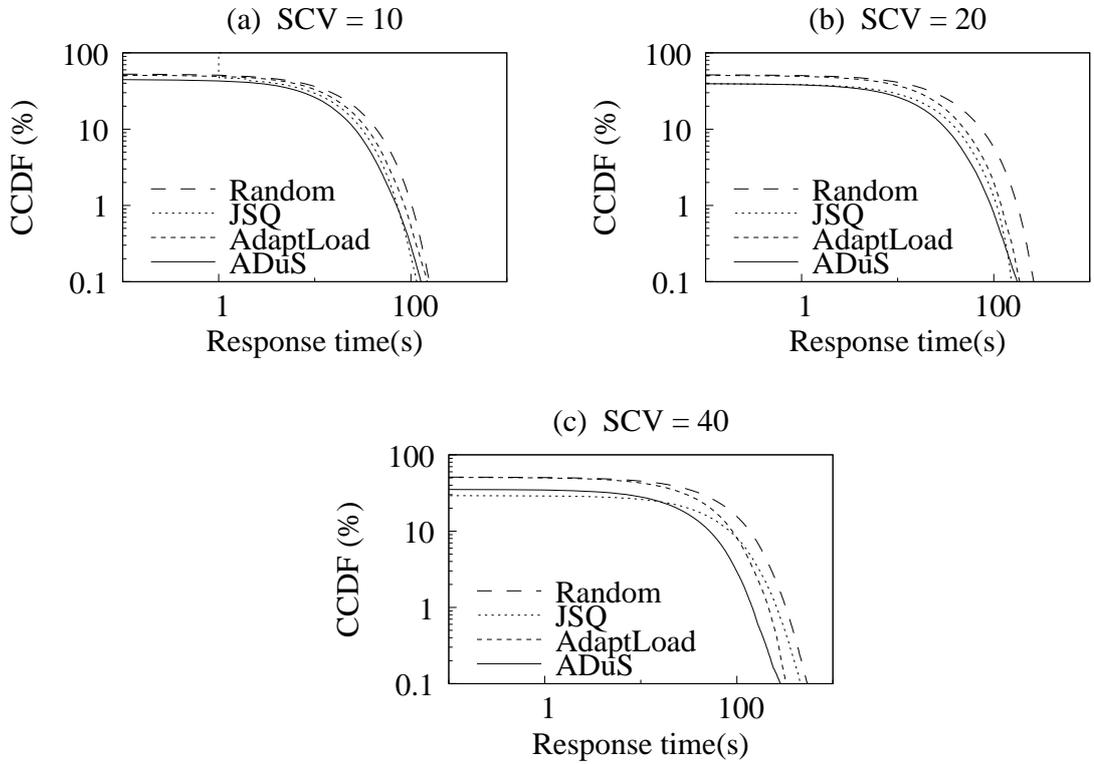


Figure 14: CCDFs under different SCVs when $\lambda = 2, \mu = 1, \mu_p = 1, \rho = 50\%$, and $ACF = 0.40$.

In order to investigate the sensitivity of ADuS to temporal dependence, we conduct experiments with three different temporal dependence profiles, but the same mean and the same SCVs of the job sizes. Figure 15 shows the experimental results as a function of temporal dependence profiles (e.g., weak, medium and strong). When the temporal dependence is weak, all the three policies JSQ, ADAPTLOAD and ADuS perform better than Random. However, the strong temporal dependence in job sizes dramatically degrades the overall system performance and deteriorates the effectiveness of JSQ and ADAPTLOAD, under which the system experiences similar performance as under Random. In contrast, ADuS still achieves a clear performance improvement, which indicates that ADuS is more robust to temporally de-

pendent workloads than the other policies.

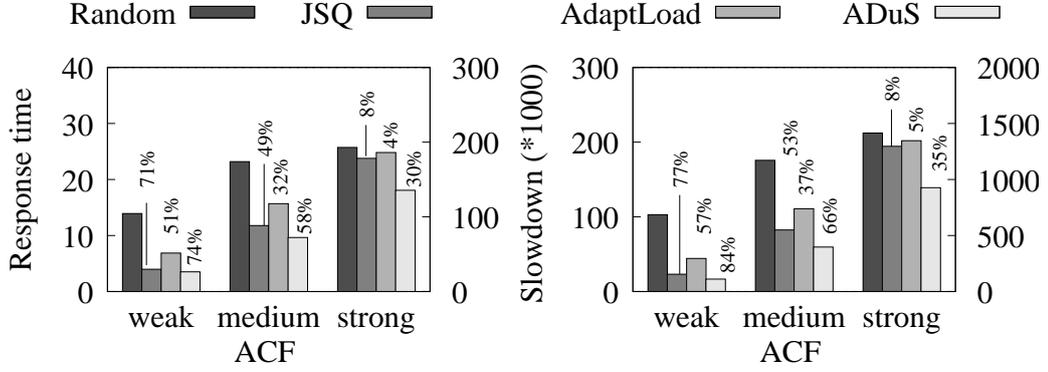


Figure 15: Response times and slowdowns under different ACFs with $\lambda = 2, \mu = 1, \mu_p = 1, SCV = 20$ and $\rho = 50\%$. Here, the strong ACF case uses the right side y-axis labels in both two plots.

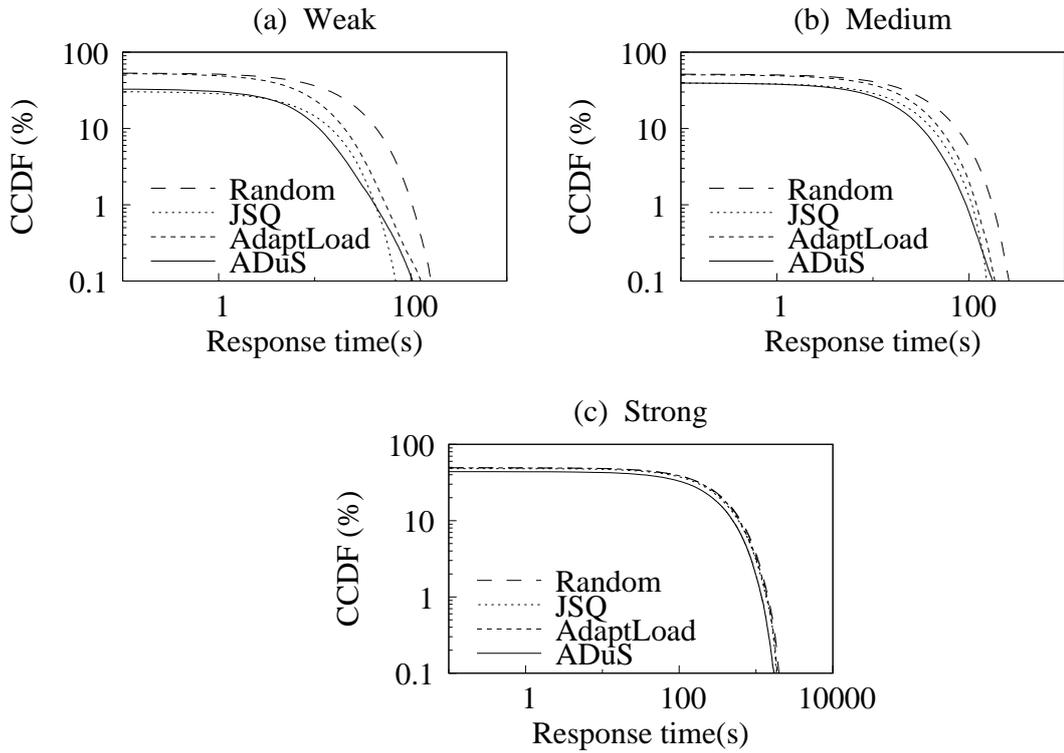


Figure 16: CCDFs under different ACFs with $\mu=0.5, \lambda=1, SCV=20$ and UTIL=50%

5.6 Sensitivity to Network Size

In order to evaluate how ADuS improves system performance under various network sizes, we here set the number of servers in the cluster to $N = 4, 16$ and 32 . We also keep the other parameters the same as the experiment in Section 5.3, except scaling the job arrival rates in order to maintain the 50% utilization level on each server. Figure 17 first shows that the system performance under each policy gets improved as the number of servers increases. We interpret that as the number of server increases, one can have more choices to determine which server will be dispatched to. As a result, it reduces the probability of having one single server been overloaded, and consequently improves the system performance. More importantly, ADuS again becomes the best policy with a clear improvement under three different network sizes.

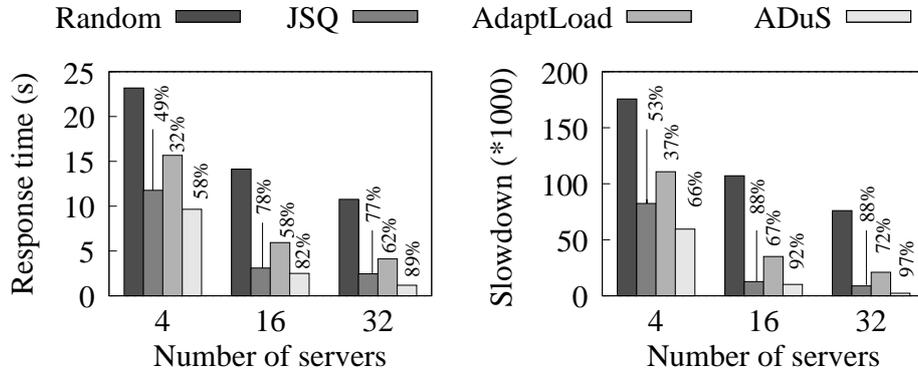


Figure 17: Response times and slowdowns under different network sizes when $\mu = 1, \mu_p = 1, SCV = 20, ACF = 0.40$, and $\rho = 50\%$.

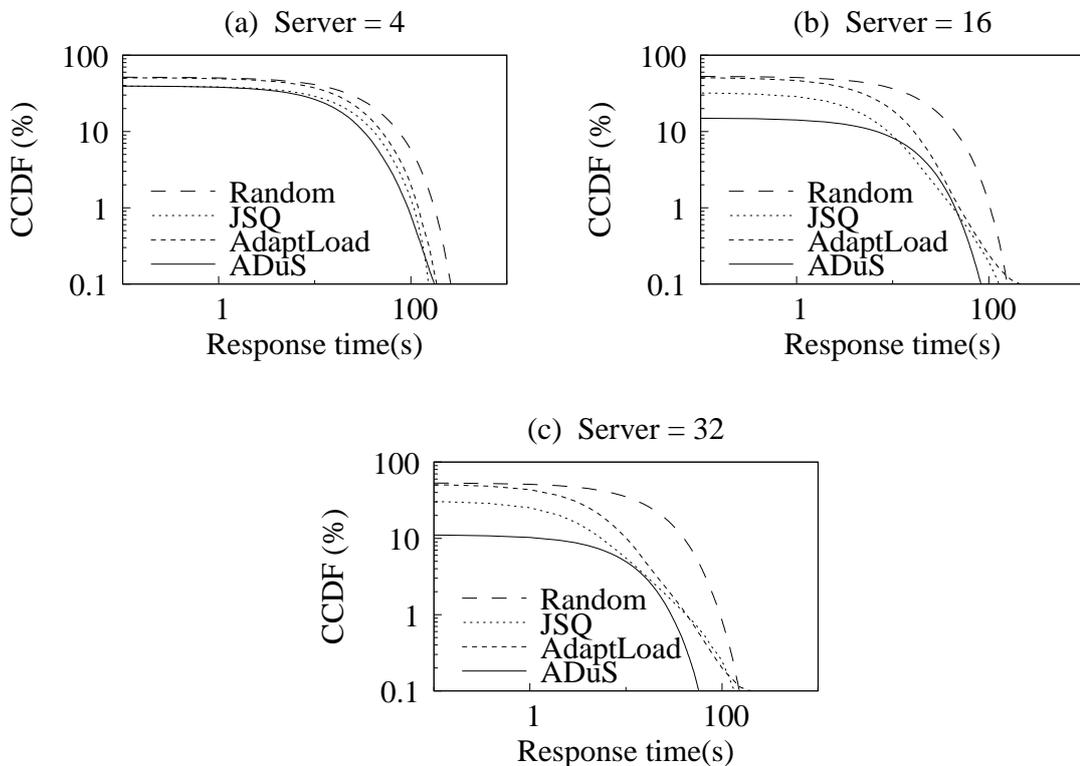


Figure 18: CCDFs under different network sizes with $\lambda=1, SCV=20$ UTIL=50% and ACF=0.40

5.7 Case Study: Real TCP Traces

Now, we validate the effectiveness and robustness of ADuS using three actual traces. The first measured data, named **LBL-TCP-3**, were collected by Lawrence Berkeley Laboratory over a two-hour period in January 1994, which consists of 1.8 million wide-area TCP packets. Each packet has a timestamp and the number of data bytes. We here use the number of data bytes as the input for job sizes, which are highly variable with $SCV = 2$ and strongly temporal dependent with ACF at lag 1 equal to 0.44. Figure 19 shows size distribution on a segment of 4000 requests of LBL-TCP-3. Remember that we set the server processing rate μ_p is equal to 1 throughout

all of our simulations, but different servers of these real traces have different processing rates, therefore, we need to scale inter-arrival times and/or job sizes in order to get a reasonable utilization. Here, we scale the packet inter-arrival times to emulate a heavy loaded system with 75% utilization. Our experimental setup assumes a cluster of remote servers in the communication network.

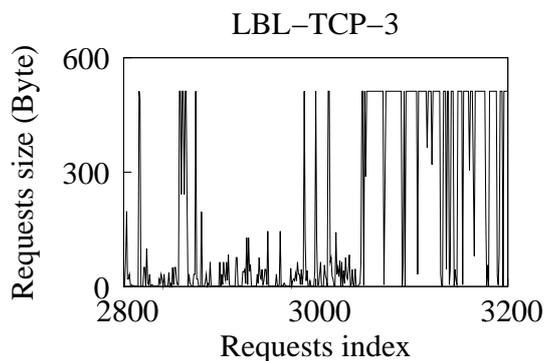


Figure 19: size distribution of LBL-TCP-3.

Figure 20 and Figure 21 summarizes the performance measurement (e.g., TCP packet round trip times and TCP packet slowdown) under the four load balancing policies. Here, the server processing rate μ_p is equal to 1, mean interarrival time is scaled to $\lambda^{-1} = 67$, and the mean package size is $\mu^{-1} = 200.87$. Consistent to the previous experiments, ADUS achieves a clear improvement in terms of TCP packet round trip times and TCP packet slowdown. Furthermore, the slowdown under ADUS is significantly improved by one order of magnitude compared to the Random policy.

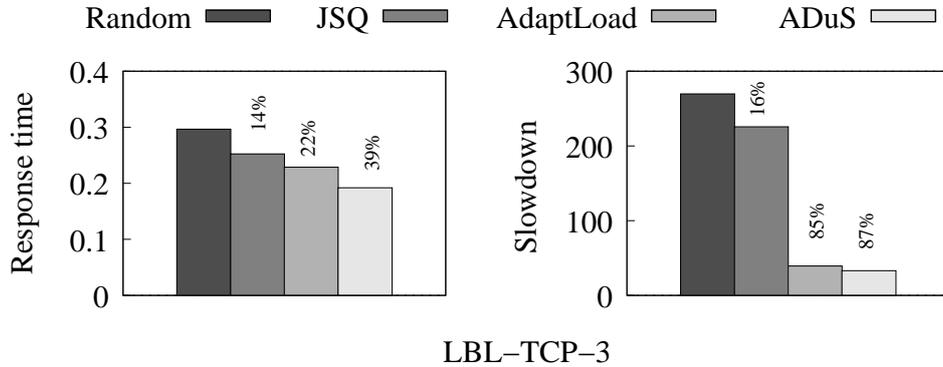


Figure 20: Response times and slowdowns of LBL-TCP-3.

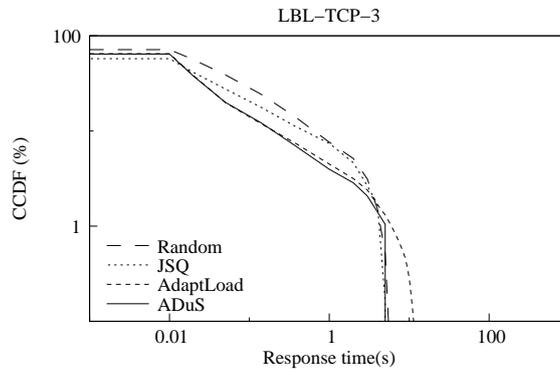


Figure 21: CCDF of LBL-TCP-3

The second trace is **BC-pOct89**, which captures a million packets on an Ethernet at the Bellcore Morristown Research and Engineering facility. The trace began at 11:00 on October 5, 1989, and ran for about 1800 seconds. It is in 2-column format, the first column gives the time in seconds since the start of the trace, i.e., the arrival time, and the second column gives the data length in bytes ranged from 64 to 1518 bytes. The arrival process has an average inter-arrival time of 0.00176s, with $SCV = 3.3$ but has no autocorrelation. Figure 22 gives an distribution of the data length, i.e., the service process, the mean job size of BC-pOct89 is $\mu^{-1} = 638.29$ bytes with $SCV = 0.8$ and $ACF = 0.36$ at lag 1. similar to the first trace, we also

scale the inter-arrival time to obtain a reasonable 40% utilization.

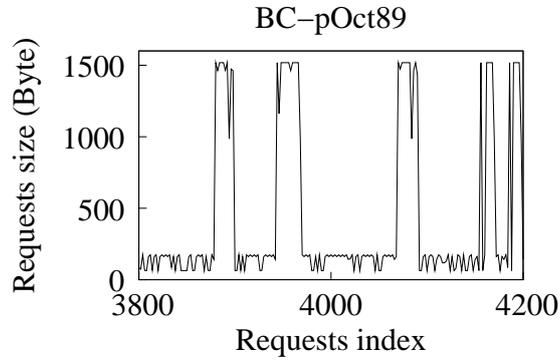


Figure 22: size distribution of BC-pOct89.

Results in Figure 23 and Figure 24 clearly show that ADuS is the best policy. With respect to random policy, ADuS has a 46% and 59% improvement in response time and slowdown, respectively.

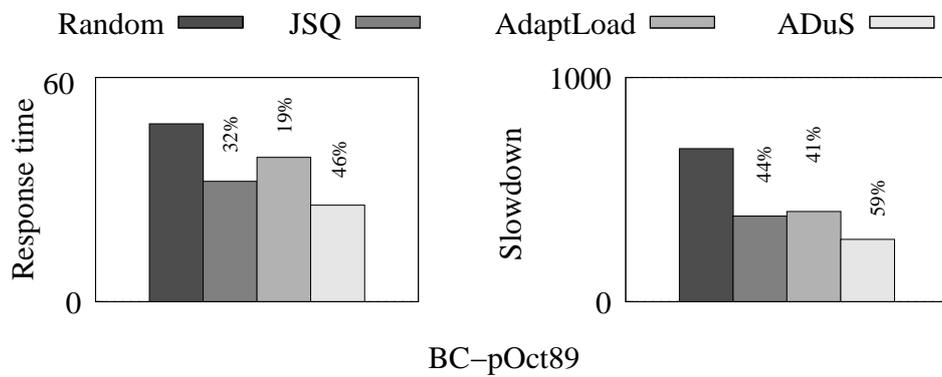


Figure 23: Response times and slowdowns of BC-pOct89.

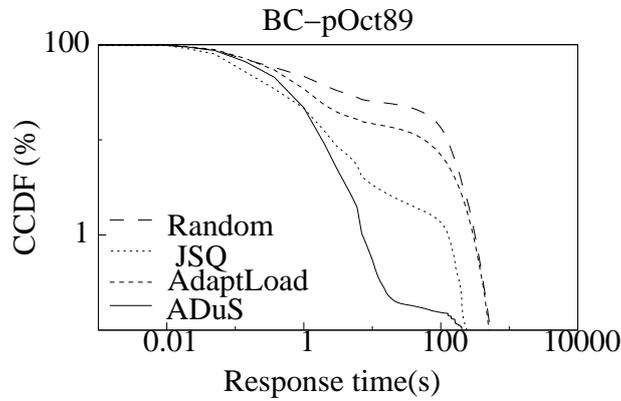


Figure 24: CCDF of BC-pOct89

WorldCup98 dataset contains all the requests made to the 1998 World Cup Web site between April 30, 1998 and July 26, 1998. Our third trace, **WC98-day36**, includes the data on the 36th day, i.e., May 31, 1998. All the data are stored in binary log format, in order to work with the binary log files, a set of tools which can parse those files are provided by the contributors of this dataset. The chosen file consists of 3.3 million of requests with an average 7507 bytes of each request. Unlike the other two traces, *wc98-day36* has little autocorrelation with only $ACF = 0.1$ at lag 1 but has a huge $SCV = 49$. Similar to previous two traces, we abstract the inter-arrival times along with file sizes, then scale them to 50% system load.

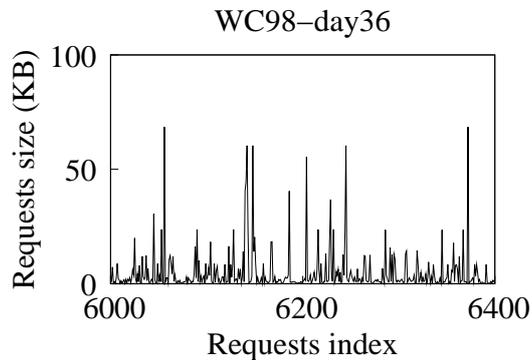


Figure 25: size distribution of WC98-day36.

Although the performance improvement of ADuS is not as clear as previous traces, ADuS still claims the best choice among all the algorithms. See Figure 26 and Figure 27.

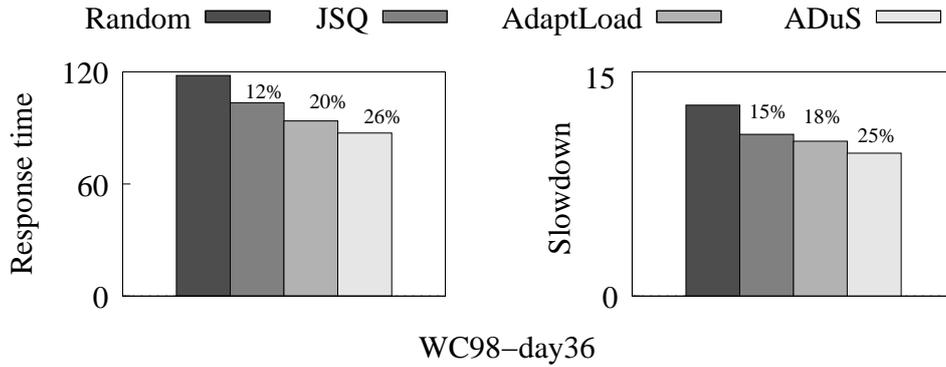


Figure 26: Response times and slowdowns of WC98-day36.

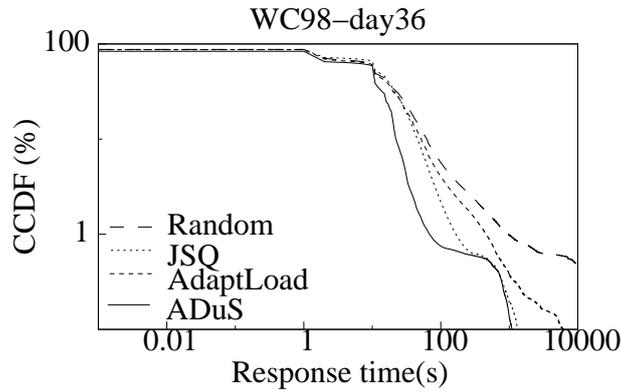


Figure 27: CCDF of WC98-day36

In summary, the extensive experimentation carried out in this section shows that ADuS effectively improves the overall system performance and thus becomes the best choice for load balancing in a cluster system. Sensitivity analysis to system loads, job size variation, job size temporal dependence, and number of servers demonstrates that the gains of ADuS are visible in a variety of different conditions. Finally, the case studies of real TCP traces

further validates the effectiveness and the robustness of ADUS.

6 Conclusion

A large-scale cluster computing system has been employed in the multiple areas by offering the pools of fundamental resources. This paper first gives an overall of cluster system, including the concept, the history and the development. How to effectively allocate the shared resources in such cluster system is a critical but challenging issue. A lot of previous studies have been focusing on developing resource management schemes and load balancing policies for large-scale cluster computing systems over the past decades. Despite the fact that classic load balancing policies, such as Random, Join Shortest Queue and size-based policies, are widely implemented in real systems due to their simplicity and efficiency, since prior research has shown that the job service times distribution is critical for the performance, the performance benefits of these policies diminish when workloads are highly variable and heavily dependent.

Our experimental results in Section 3 reveal both JSQ and ADAPTLLOAD are ineffective under highly variable and/or strongly dependent workloads. In particular, since JSQ always sends a job to the shortest (least loaded) queue, there is a high probability such that small jobs are stuck behind large ones, which consequently cause performance degradation; while some servers under ADAPTLLOAD policy may suffer a sudden load spike during a short time period due to strong autocorrelation. Motivated by these problems of existing policies, we try to propose a new load balancing policy which attempts to address the limitations of both JSQ and ADAPTLLOAD and thus achieves visible performance improvement.

We first present ROLE ROTATION policy which assigns requests to less

busy or idle server under strong dependent workload. From servers perspective, each of them plays different ‘roles’ by serving different types (sizes) of jobs at different time. Unfortunately, during the preliminary experiments we find that there is little improvement of `ROLE ROTATION`. So we turn to present another improved algorithm, called `ADUS`, which adaptively distributes work among all servers by taking account of both *user traffic* and *system load*, aiming at inheriting the effectiveness of `JSQ` and `ADAPTL``OAD` policies and meanwhile overcoming the limitations of these two policies as shown in the previous section. The main idea of `ADUS` is to on-the-fly rank all the servers according to their system loads and dynamically tune the job size boundaries based on the current user traffic loads. `ADUS` then directs the jobs whose sizes locate in the same size boundary to the corresponding servers which are in the same ranking. Based on the observation that the majority of jobs in a heavy-tailed workload is small, `ADUS` always gives small jobs high priority by sending them to the highly ranked (i.e., less loaded) servers.

Trace-driven simulations are used to evaluate the performance of `ADUS`, we induce high variance and strong autocorrelation to service process (job size) while keep the arrival process (inter-arrival time) exponential distribution. Our results indicate that `ADUS` significantly improves the system performance - job response times and job slowdowns - up to 58% and 66%, respectively. Extensive sensitivity analysis with respect to system loads, variation and temporal dependence in job sizes, and the network size demonstrate that `ADUS` is effective in various environments. We also show that `ADUS` can quickly adapt to the workload changes by monitoring user traffic and system loads, repeatedly ranking the servers and partitioning the work in an on-line fashion. Several case studies of real traces further validate the

robustness of ADUS in actual systems.

7 Future Works

In the future, we try to improve the performance of our ADUS policy toward the following directions:

- **Ranking criteria:** In our experiment, we use queue length as the ranking criteria, i.e., we re-rank all the servers in an increasing order of queue length, on the assumption that the server with the shortest queue is the least loaded server, we believe small jobs should be always assigned to the that server. In future study, we may consider other parameter as rank criteria to see if the performance is better, for example we may use server transient utilization, CPU idle time, memory usage,etc. In this paper, we actually try *index of dispersion* as another rank criteria. It is defined as follow:

$$I = SCV(1 + 2 \sum_{k=0}^{\infty} \rho k) \quad (4)$$

where SCV is the squared coefficient of variation and ρk is the auto-correlation on lag- k . the value of I can be used as a good indicator of autocorrelation []. We anticipate index of dispersion can help us in our policy, however, the results indicate that it is not the case. Further studies need to be done to explore the reasons.

- **Window size:** Our current simulations use a fixed window size C of 100 jobs. The value implies how often ADUS re-ranks all servers. This number (100) is chosen after running sensitivity experiments on different window sizes and the best performance is obtained when C

ranges from 50 to 200. The next step is to dynamically adjust the window size to transient workload conditions. We either can set up several threshold parameters and monitor carefully of the system, we will re-rank servers as long as one or multiple parameters violate those thresholds; or combine the work of burst prediction technique [], if we correctly predict a load spike coming soon, we definitely can adjust all server in advance so that the system can work well under such bursty load.

- **Multiple job classes:** We consider only one single class of requests, treating all of the incoming jobs without any priority. In most of the real systems, however, there are multiple classes - general users, VIP members, group customers and administrators, each of them should have different priorities. How to successfully accommodate different users while maintain high system performance is a very challenging issue.
- **Heterogeneous servers:** Similar to job classes, we can also take heterogeneous server into account instead of homogeneous ones.
- **Real system implementation:** Finally we should implement our algorithm to real systems. We may either implement our policy to a TPC-W benchmark which simulates the behaviors of a online bookstore or we apply for a set of Amazon Elastic Compute servers and run our policy on such real cloud computing systems.

References

- [1] P. Chaganti, “Cloud computing with amazon web services,” pp. 1–10, 2008.
- [2] J. Varia, “Building grepheweb in the cloud,” in *Amazon Elastic Compute Cloud Articles and Tutorials*, 2008.
- [3] Y. M. Teo and R. Ayani, “Comparison of load balancing strategies on cluster-based web servers,” *Trans. Soc. for Modeling and Simulation*, vol. 77, no. 5-6, pp. 185–195, Nov. 2001.
- [4] Q. Zhang, N. Mi, A. Riska, and E. Smirni, “Performance-guided load (un)balancing under autocorrelated flows,” *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, vol. 19, no. 2, pp. 652–665, 2008.
- [5] N. Mi, Q. Zhang, A. Riska, and E. Smirni, “Load balancing for performance differentiation in dual-priority clustered servers,” pp. 385–395, 2006.
- [6] H. Feng, M. Visra, and D. Rubenstein, “Optimal state-free, size-aware dispatching for heterogeneous m/g/-type systems,” *Performance Evaluation J.*, vol. 62, no. 1-4, pp. 475–492, Nov. 2005.
- [7] M. Harchol-Balter and A. Downey, “Exploiting process lifetime distributions for dynamic load balancing,” *ACM Trans. Computer Systems*, vol. 15, no. 3, pp. 253–285, Aug. 1997.
- [8] W. Winston, “Optimality of the shortest line discipline,” *Journal of Applied Probability*, vol. 14, pp. 181–189, 1977.

- [9] R. Nelson and T. Philips, “An approximation for the mean response time for shortest queue routing with general interarrival and service times,” *Performance Evaluation*, vol. 17, pp. 123–139, 1998.
- [10] W. Whitt, “Deciding which queue to join: Some counterexamples,” *Operations Research*, vol. 34, no. 1, pp. 226–244, Jan. 1986.
- [11] M. Harchol-Balter, M. Crovella, and C. Murta, “On choosing a task assignment policy for a distributed server system,” *J. Parallel and Distributed Computing*, vol. 59, no. 2, pp. 204–228, Nov. 1999.
- [12] Q. Zhang, A. Riska, W. Sun, E. Smirni, and G. Ciardo, “Workload-aware load balancing for clustered web servers,” *IEEE Trans. Parallel and Distributed Systems*, vol. 16, no. 3, pp. 219–233, Mar. 2005.
- [13] O. H. Ibarra and C. E. Kim, “Heuristic algorithms for scheduling independent tasks on nonidentical processors,” vol. 24(2), 1977.
- [14] J. Frey, T. Tannenbaum, M. Livny, I. Foster, and S. Tuecke, “Condor-G: A computation management agent for multi-institutional grids,” *Cluster Computing*, vol. 5, pp. 237–246, 2002.
- [15] W. Smith, I. Foster, and V. Taylor, “Scheduling with advanced reservations,” pp. 127 – 132, 2000.
- [16] G. Amdahl, “Validity of the single processor approach to achieving large-scale computing capabilities,” *AFIPS spring joint computer conference*, vol. 30, pp. 225–231, Apr 1967.
- [17] M. Stott, G. Beason, and G. Thomas, “Arcnet works,” 1998.

- [18] N. Kronenberg, H. Levy, and W. Strecher, “Vaxcluster: a closely-coupled distributed system,” *ACM Transactions on Computer Systems*, vol. 4, pp. 130–146, May 1986.
- [19] “Compatibility mode migration resources guide.” [Online]. Available: <http://www.hp.com/products1/evolution/e3000/download/59689417.pdf>
- [20] “Amazon offers new cloud-based high performance computing.” [Online]. Available: <http://hothardware.com/News/Amazon-Offers-New-CloudBased-High-Performance-Computing-/>
- [21] “Amazon elastic compute cloud user guide.” [Online]. Available: <http://awsdocs.s3.amazonaws.com/EC2/latest/ec2-ug.pdf>
- [22] L. Kleinrock, *Queueing Systems Volume 1: Theory*, Wiley, 1975.
- [23] N. Mi, Q. Zhang, A. Riska, E. Smirni, and E. Riedel, “Performance impacts of autocorrelated flows in multi-tiered systems,” *Performance Evaluation*, 2007.
- [24] M. F. Neuts, *Structured Stochastic Matrices of M/G/1 Type and Their Applications*. New York: Marcel Dekker, 1989.
- [25] G. Latouche and V. Ramaswami, “Introduction to matrix analytic methods in stochastic modeling,” in *ASA-SIAM Series on Statistics and Applied Probability*, Philadelphia PA, 1999.