ABSTRACT

BOLOOR, KEERTHANA. Management of SOA-based, Data-intensive Applications Deployed in a Distributed Cloud Subject to Response Time Percentile Service Level Agreements. (Under the direction of Yannis Viniotis and Rada Chirkova.)

We consider geographically distributed datacenters forming a collectively managed cloud computing system. Multiple SOA-based applications are hosted in the cloud. Service Level Agreements which dictate Quality of Service (QoS) and pricing models, arenegotiated between the cloud provider and the SOA-based enterprise application vendor. A QoS metric that has been explored in large distributed applications is the percentile of response times; this metric provides a form of guarantee on the shape of the response time distribution for the customer. Typical percentile SLAs require the response time of a certain percentile of the input requests from particular classes of customers to be less than a specified value; if this value is exceeded, a penalty is charged to the cloud provider. In addition, the applications we consider are data-intensive with strict temporal order constraints that have to be enforced on requests within the same session of a customer. We propose Data-aware Session-grained Allocation with gi-FIFO Scheduling (DSAgS), a novel decentralized request management scheme deployed in each of the geographically distributed datacenters, to globally reduce the penalty charged to the cloud computing system. Our simulation and prototype-based evaluation shows that our dynamic scheme far outperforms commonly deployed management policies (typically employing static or random allocation with First In First Out, Weighted Round Robin or dynamic priority-based scheduling). We further optimize our solution by proposing a “context-level” cache replacement algorithm.
Management of SOA-based, Data-intensive Applications Deployed in a Distributed Cloud
Subject to Response Time Percentile Service Level Agreements

by
Keerthana Boloor

A dissertation submitted to the Graduate Faculty of
North Carolina State University
in partial fulfillment of the
requirements for the Degree of
Doctor of Philosophy

Computer Engineering

Raleigh, North Carolina

2012

APPROVED BY:

Rada Chirkova
Co-chair of Advisory Committee

George Rouskas

Michael Devetsikiotis

Robert David Callaway

Yannis Viniotis
Co-chair of Advisory Committee
BIOGRAPHY

Keerthana Boloor is a PhD candidate in the Department of Electrical and Computer Engineering at North Carolina State University (NCSU), Raleigh, NC, USA. Keerthana has received a B.E. degree in Electronics and Communication from The National Institute Of Engineering, Mysore, Karnataka, India (Affiliated to Visvesvaraya Technological University, Belgaum, Karnataka, India) and a M.S. degree in Computer networking from North Carolina State University, Raleigh, NC, USA. Keerthana’s professional experience includes a software engineering position with Philips Healthcare, Bangalore, India spanning two years and an Websphere Technology Institute (WSTI) internship with IBM Software Lab, RTP, NC, USA spanning eighteen months.

Keerthana is a two-time winner of the IBM PhD Fellowship award winner for the years 2010-2011 and 2011-2012. Her research interests are primarily in the fields of cloud computing and software oriented computing.
ACKNOWLEDGEMENTS

I have been very fortunate to have two fabulous advisors Dr. Yannis Viniotis and Dr. Rada Chirkova. Dr. Viniotis, who has also been my Master’s thesis advisor, has been providing expert advice throughout my graduate research; had it not been for him, I would not have transitioned from the M.S to the PhD program. Dr. Rada Chirkova has also been a great source of support and insightful advice over the course of my research. I am also very grateful to both my wonderful advisors for nominating me for the IBM PhD Fellowship for two years in a row.

I would also like to thank Tiia Salo and Dr. Andy Rindos from IBM Software, RTP, NC, USA for mentoring me during my internship and fellowship years at IBM. I am also grateful to all the members of my committee, Dr. Michael Devetsikiotis, Dr. Robert Callaway and Dr. George Rouskas for their valuable advice and feedback over the course of my doctoral research.

Finally, special thanks to my parents and my husband for their encouragement, guidance and love all through my graduate studies.
# TABLE OF CONTENTS

List of Figures ......................................................... vi

Chapter 1 Introduction .................................................. 1
  1.1 Enterprise applications ........................................... 1
  1.2 Service Oriented Architecture ................................... 1
  1.3 Cloud Computing System ........................................ 2
  1.4 Management of SOA-based enterprise applications in a cloud computing system . 4
  1.5 Service Level Agreements ....................................... 6

Chapter 2 Percentile Service Level Agreements ......................... 9
  2.1 Service Level Agreements ........................................ 9
  2.2 Response time percentile Service Level Agreements .......... 11
    2.2.1 SLA Parameters ........................................... 11
    2.2.2 Percentile SLA definition ................................ 13
    2.2.3 Enforcement of the SLA .................................. 13
    2.2.4 Variants of penalty functions ............................ 13
    2.2.5 Selection of penalty functions ............................ 14
  2.3 Evaluation methodology ......................................... 15
  2.4 Evaluation results ............................................. 17
    2.4.1 Comparison of conformance in the 3 variants ......... 18
    2.4.2 Difference in utilizing per-request and per-fraction penalties .... 19
    2.4.3 Relation between step-size and observation interval .... 19
  2.5 Related work .................................................... 20
  2.6 Summary ........................................................ 21

Chapter 3 Management of SOA-based applications ....................... 22
  3.1 Problem formulation ............................................ 22
    3.1.1 System architecture ...................................... 22
    3.1.2 SOA-based data-intensive enterprise applications .... 23
    3.1.3 Deployment pattern of SOA-based applications in a data-center .... 24
    3.1.4 Percentile SLA ........................................... 27
    3.1.5 Problem statement ........................................ 29
    3.1.6 Objectives ................................................ 29
    3.1.7 Solution approach ........................................ 30
  3.2 Reactive management policy description ........................ 30
    3.2.1 Periodic exchange of metrics ............................. 31
    3.2.2 Component 1: Request allocation algorithm ............. 32
    3.2.3 Component 2: Request scheduling algorithm ............ 35
    3.2.4 Component 3: Session reallocation algorithm .......... 36
    3.2.5 Component 4: Context cache replacement algorithm .... 40
    3.2.6 Assumptions .............................................. 41
3.2.7 Centralized variant of DSAgS ........................................... 41
3.3 Implementation of DSAgS ................................................... 43
  3.3.1 Software architecture ................................................. 43
3.4 Evaluation ................................................................. 45
  3.4.1 Simulation-based evaluation ....................................... 46
  3.4.2 Comparison of DSAgS against commonly deployed policies .... 46
  3.4.3 Evaluation of session reallocation ................................. 49
  3.4.4 Evaluation of context replacement algorithm ................... 53
  3.4.5 Robustness of DSAgS ................................................. 56
  3.4.6 Results of implementation in a prototype ....................... 60
3.5 Related Work ............................................................ 76
  3.5.1 QoS in SOA-based enterprise data-intensive applications .... 76
  3.5.2 Percentile Service Level Agreements ................................ 76
3.6 Summary ................................................................. 77

Chapter 4 Future Work ......................................................... 78
  4.1 Future work towards the reactive management policy .............. 78
  4.2 Proactive management of SOA-based data-intensive applications subject to percentile Service Level Agreements .................. 79

References ................................................................. 80
Appendix ................................................................. 84
  Appendix A Evaluation of heuristics for request scheduling subject to response time percentile Service Level Agreements ................................. 85
    A.1 Step-wise Service Level Agreement ................................ 85
    A.2 System model ......................................................... 86
    A.3 Problem statement ................................................... 87
    A.4 Heuristic-based request schedules .................................. 88
      A.4.1 Algorithm description .......................................... 89
      A.4.2 Evaluation ....................................................... 94
    A.5 Conclusion ........................................................ 99
## LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Figure 1.1</td>
<td>Cloud computing system. (Image source [42])</td>
<td>3</td>
</tr>
<tr>
<td>Figure 1.2</td>
<td>SaaS cloud delivery model. (Image Source [10])</td>
<td>4</td>
</tr>
<tr>
<td>Figure 2.1</td>
<td>Linear penalty function.</td>
<td>14</td>
</tr>
<tr>
<td>Figure 2.2</td>
<td>Exponential penalty function.</td>
<td>15</td>
</tr>
<tr>
<td>Figure 2.3</td>
<td>Stepwise penalty function.</td>
<td>16</td>
</tr>
<tr>
<td>Figure 2.4</td>
<td>Comparison of percentile SLA variations (linear, exponential and stepwise)</td>
<td>19</td>
</tr>
<tr>
<td>Figure 2.5</td>
<td>Comparison of per-request and per-fraction penalties</td>
<td>20</td>
</tr>
<tr>
<td>Figure 2.6</td>
<td>Relation between step-size and observation interval</td>
<td>21</td>
</tr>
<tr>
<td>Figure 3.1</td>
<td>System architecture.</td>
<td>23</td>
</tr>
<tr>
<td>Figure 3.2</td>
<td>Deployment pattern of a SOA-based enterprise application in a datacenter.</td>
<td>25</td>
</tr>
<tr>
<td>Figure 3.3</td>
<td>Linear penalty function.</td>
<td>27</td>
</tr>
<tr>
<td>Figure 3.4</td>
<td>Multi-step percentile Service Level Agreement</td>
<td>28</td>
</tr>
<tr>
<td>Figure 3.5</td>
<td>Solution approach</td>
<td>30</td>
</tr>
<tr>
<td>Figure 3.6</td>
<td>Observation interval divided into multiple subintervals in each datacenter.</td>
<td>31</td>
</tr>
<tr>
<td>Figure 3.7</td>
<td>Highlights of the request allocation algorithm</td>
<td>34</td>
</tr>
<tr>
<td>Figure 3.8</td>
<td>End-server model at a datacenter</td>
<td>37</td>
</tr>
<tr>
<td>Figure 3.9</td>
<td>Depiction of varying penalty scores with current conformance levels (Linear Percentile SLAs)</td>
<td>42</td>
</tr>
<tr>
<td>Figure 3.10</td>
<td>High level software architecture of the DSAGS implementation</td>
<td>44</td>
</tr>
<tr>
<td>Figure 3.11</td>
<td>Typical linear percentile SLA examples</td>
<td>47</td>
</tr>
<tr>
<td>Figure 3.12</td>
<td>Comparison of DSAGS with commonly deployed solutions. (Linear percentile SLA)</td>
<td>48</td>
</tr>
<tr>
<td>Figure 3.13</td>
<td>Comparison of DSAGS with commonly deployed solutions. (Stepwise percentile SLA)</td>
<td>49</td>
</tr>
<tr>
<td>Figure 3.14</td>
<td>Demonstration of allocation dependency in all algorithms.(Linear percentile SLA)</td>
<td>50</td>
</tr>
<tr>
<td>Figure 3.15</td>
<td>Demonstration of allocation dependency in all algorithms.(Stepwise percentile SLA)</td>
<td>51</td>
</tr>
<tr>
<td>Figure 3.16</td>
<td>Comparison of DSAGS with and without session reallocation.(Linear percentile SLA)</td>
<td>52</td>
</tr>
<tr>
<td>Figure 3.17</td>
<td>Comparison of DSAGS with and without session reallocation.(Stepwise percentile SLA)</td>
<td>53</td>
</tr>
<tr>
<td>Figure 3.18</td>
<td>Effect of length of the session reallocation interval.(Linear percentile SLA)</td>
<td>54</td>
</tr>
<tr>
<td>Figure 3.19</td>
<td>Effect of length of the session reallocation interval.(Stepwise percentile SLA)</td>
<td>55</td>
</tr>
<tr>
<td>Figure 3.20</td>
<td>Effect of number of simultaneous sessions and number of requests in each session. (Linear percentile SLA)</td>
<td>56</td>
</tr>
<tr>
<td>Figure 3.21</td>
<td>Effect of number of simultaneous sessions and number of requests in each session. (Stepwise percentile SLA)</td>
<td>57</td>
</tr>
<tr>
<td>Figure 3.22</td>
<td>Penalties charged for different context cache coefficient combinations.(Linear percentile SLA)</td>
<td>58</td>
</tr>
</tbody>
</table>
Figure A.12 Comparison of simulated annealing based pairwise interchange, last insertion and gi-FIFO. .......................... 101
Figure A.13 Comparison of total penalty in simulated annealing based pairwise interchange and tabu search based pairwise interchange. .......................... 101
Figure A.14 Total penalty obtained in tabu search pairwise interchange with varying iterations for two input request rates. .......................... 102
Chapter 1

Introduction

1.1 Enterprise applications

Applications that cater to business operations, management of IT needs in large corporations are often termed as enterprise applications. Financial solution applications for banks with global presence, CRM applications for corporations with geographically distributed offices are some examples of enterprise applications [20]. Enterprise applications typically need to be flexible (allow seamless modifications to existing features and functions), extensible (allow seamless addition of new features and functions), scalable (allow seamless increase in performance and capacity) and generally run in geographically located sites on heterogeneous platforms. Enterprise applications are mostly designed on Service Oriented Architecture (SOA) principles that facilitate the incorporation of above mentioned requirements in the application [30].

1.2 Service Oriented Architecture

Service Oriented Architecture is the mainstream technology adopted in designing most enterprise applications. The purpose of this architecture is to address the requirements of loosely coupled (flexibility and extensibility), standards-based (ability to run on heterogenous platforms), and protocol-independent distributed computing [30].

In a SOA-based application, functionality is packaged as “interoperable services”, which are well defined, self-contained modules that provide standard business functionality [38]. These services can be “composed” dynamically to create new functionality at runtime promoting flexibility and extensibility. Since the “services” communicate in a standardized, protocol-independent, mostly stateless fashion across network boundaries, the application can be deployed across geographically distributed sites on heterogenous platforms, migrated and instantiated dynamically to multiple machines making them inherently scalable.
Older software architectures with a large number tightly coupled static libraries interacting over often manually configured protocols and platforms could not support the growing requirements of enterprise applications, fueling the development of a number of distributed application architectures by independent software vendors. Service Oriented Architecture has emerged by adopting principles from several of these previous attempts aimed at increasing flexibility, extensibility and scalability of applications like CORBA [23], RMI [34], COM [33].

Since the emergence of SOA, several research efforts have focused on the adoption of SOA design principles in the enterprise setting for varied industrial applications. For example, in [24], Gu et.al, propose a SOA-based framework for supporting context-aware applications, in [52] Wenbin et.al, propose a SOA-based monitoring platform for refining and chemical enterprises, in [48] Taylor et.al show how SOA can power real-time enterprise applications and so on. In this work, we consider SOA-based enterprise applications and from hereon in this report any reference to enterprise applications assume them being built on SOA principles.

Increasingly, industries are catering to customers globally, typically with dynamic delivery models. To support the global scale and dynamic needs of the application, underlying IT resources need to be provisioned on-demand in geographically distributed locations.

Cloud computing systems provide IT resources over the internet with datacenters located in geographically distributed sites. These resources are typically provided on-demand, increased or decreased based on pricing and availability agreements negotiated between the application vendor and the cloud provider [28] resulting in an increasing number of enterprise applications deployed in cloud.

1.3 Cloud Computing System

Buyya et.al, in [14] define a cloud as

“A Cloud is a type of parallel and distributed system consisting of a collection of interconnected and virtualized computers that are dynamically provisioned and presented as one or more unified computing resources based on service-level agreements established through negotiation between the service provider and consumers.”

Fig. 1.1 is a widely accepted pictorial representation of a cloud.

Usage models in a cloud

The following are the three main usage models (with widely accepted definitions) differentiated by the kind of service offered by the cloud computing system [21]:

- **Infrastructure as a Service (IaaS)**. In this model, the cloud provides low-level services like virtual machines which can be booted with a user-defined hard-disk image, i.e.
Amazon EC2 [1]. Virtual hard-disks that can be accessed from different virtual machines are another example of infrastructure as a service.

- **Platform as a Service (PaaS).** In this model the cloud provider offers an API which can be used by an application developer from the consumer industry to develop “number-crunching” applications or web applications with friendly user-interfaces. An example is Google’s App Engine [2].

- **Software as a Service (SaaS),** is the model where the cloud consumers deploy their SOA-based enterprise applications to the cloud provider. Here the cloud provider hosts, manages the consumer application and serves the users of the consumer applications. Examples are web-based office applications like Google Docs or Calendar, IBM’s Websphere suite of products offered on EC2.

In this work we focus on the SaaS cloud management.

**Business players in a SaaS model of a cloud**

The major players in a SaaS cloud computing environment are as follows:

- **Cloud provider** These are the infrastructure providers in a cloud. They own the multi-core networked resources where the software and the required platforms operate and are offered as services. to the cloud consumers.

- **Cloud consumers** These are the SOA-based enterprise application vendors who deploy their products on the cloud which are offered as services over the internet. These software
providers “rent” the computing power of the cloud on-demand and negotiate a level of service with the cloud provider.

- **Software user** These are the end consumers of the software hosted on the cloud.

Fig. 1.2 shows a pictorial representation of a SaaS delivery model with the major players.

![SaaS cloud delivery model](Image Source [10])

Typically, in a SaaS usage model of a cloud computing system, the cloud provider offers deployment and management services to the enterprise application vendor. Deployment services include providing suitable compatible platforms, servers with sufficient compute and storage resources for the running of one or more instances of the application. Management services include QoS guarantees on availability, response time via resource provisioning, request allocation and scheduling; security and so on. In this work, we are concerned with the management services offered by the cloud.

### 1.4 Management of SOA-based enterprise applications in a cloud computing system

Typical management goals for a cloud computing system providing hosting services to SOA-based enterprise application vendors include and are not limited to:

- **Availability:** This goal requires the cloud provider to make sure that the hosted applications are always available for access. Typical examples of availability goals are that the application should be available for access 99.95% over 5-minute intervals time over a period of 365 service days [1]. Availability could also include the availability of storage at the cloud.
• **Performance:** This goal requires the cloud provider to make sure that the hosted applications have a pre-specified response time for customers. The customers can be divided into different levels based on the guaranteed response time value.

• **Security:** Guarantees on security is offered by the cloud provider to the enterprise application vendor in the form of specialized firewalls, protection against known malware attacks and so on. But this guarantee is not well-defined in the industry and the formulation is still a subject a active research [40].

• **Speed and ease of deployment:** Guarantees on the ease of deployment of the enterprise applications on the cloud is an area of potential research and no known industry specifications exist for the same.

In this work we deal primarily with response time guarantees offered by the cloud provider to the enterprise application vendor.

Owing to virtualization of resources and the ability to instantiate SOA-based application services on-demand, simultaneously on multiple servers, performance management can be offloaded to the cloud provider. This offloading serves both the provider and the consumer of the cloud services, as the provider does not have to invest in the technical “know-how” of managing the performance of the underlying platform and can focus on business aspects of application development; the cloud provider, with “insider” knowledge of the platform, can better manage resources between multiple consumers and can guarantee increased performance while reducing costs and energy consumption.

Typically, the cloud provider can monitor and optimize the following control parameters to provide performance guarantees to the enterprise application vendors (consumers):

• Allocation of resources (servers, network) to applications.

• Instantiation of service-endpoints, placement of data (database instances, in-memory caches) at servers.

• Allocation of end-user requests to servers.

• Scheduling end-user requests at servers.

Cloud computing, owing to the ability to provide on-demand management and hosting services to application vendors, has resulted in the adoption of the utility computing business model. In utility computing, the application vendors “pay per amount and quality of use” of cloud resources. Vendors that host applications in the cloud are charged based on the performance, availability levels and the duration for which they are hosted. Some examples include Amazon’s Elastic Compute Cloud (EC2) [6], in which customers pay for compute resources
by the hour, and Simple Storage Service (S3) [7], for which customers pay based on storage capacity. Other utility services include EMC’s storage cloud service [19], and those offered by startups such as Joyent [26] and Rackspace [41].

In a utility computing business model, cloud providers and application vendors (cloud consumers) need to agree on the pricing models and terms of service. A Service Level Agreement (SLA) is negotiated between the cloud provider and the cloud consumer that includes the pricing and terms of service offered by the cloud provider to the consumer.

1.5 Service Level Agreements

A Service Level Agreement is not a new concept. It has been employed by the telecommunication industry for several years. As specified in the SLA handbook [4]:

“It is the Service Level Agreement that defines the availability, reliability and performance quality of delivered telecommunication services and networks.”

Due to the loose coupling between cloud providers, cloud consumers and end-users in a SaaS model of compute cloud, similar definition as above can be applied to SLA negotiated between the cloud provider and cloud consumer.

Some example SLAs negotiated in the cloud computing space are:

- **Amazon Web Services**: WS will use commercially reasonable efforts to make Amazon EC2 available with an Annual Uptime Percentage of at least 99.95% during the Service Year. In the event Amazon EC2 does not meet the Annual Uptime Percentage commitment, you will be eligible to receive a Service Credit [1].

- **Windows Azure**: Windows Azure has separate SLAs for compute and storage. For compute, we guarantee that when you deploy two or more role instances in different fault and upgrade domains your Internet facing roles will have external connectivity at least 99.95% of the time. Additionally, we will monitor all of your individual role instances and guarantee that 99.9% of the time we will detect when a role instances process is not running and initiate corrective action.

  For storage, we guarantee that at least 99.9% of the time we will successfully process correctly formatted requests that we receive to add, update, read and delete data. We also guarantee that your storage accounts will have connectivity to our Internet gateway.

- **NetQoS SuperAgent**: This example is of a response time percentile SLA set by a network monitoring tool for analysis purpose and not enforcement. 90% of the observations should have a response time of less than 25 milliseconds; 98% of the observations should have a response time of less than 81 milliseconds.
In this work we deal with Service Level Agreements (SLA) that deal with performance. Typically, SLAs for business applications specify (among other constraints) certain guarantees in terms of the fraction of requests serviced, as opposed to average-performance criterion. Thus, many Service Level Agreements are designed to provide specific percentile-based performance goals. It has been shown in [15] that enforcing percentile performance criterion as a management objective, results in better conformance and improved response times in comparison to average performance criterion guarantees. Recent business trends in cloud computing systems have shown increasing adoption of fixed-step percentile SLAs, where a certain fraction of service requests for a hosted application is required to have a specific response time, if not a penalty is charged to the cloud [22].

The exact relation between the metric of percentile conformance and the monetary penalties charged on the provider due to non-conformance (hereon referred to as the “penalty function”) is an important parameter of the SLA. There are multiple ways in relating the two and thus a variety of penalty functions can be chosen. In this work, we investigate three variants of the penalty function: linear, exponential and step-wise.

Percentile response time has been monitored in web servers, application servers [3], cloud computing systems [39], however, apart from [53] and [15] that make assumptions on knowing the input distribution and $\alpha$-quantile of input distribution respectively, there has been no effort in adaptive management of the cloud resources to obtain a desired response time percentile.

In our work, we consider a cloud computing system consisting of multiple geographically distributed datacenters, each with a large number of end-servers. The centers collectively host, multiple SOA-based enterprise applications. The SOA-based enterprise applications are each negotiated with fixed-step percentile SLAs. We address the problem where the request management techniques employed by the cloud should aim at globally conforming to the percentile SLAs negotiated for each application, thus minimizing the penalty charged.

We consider the SOA-based enterprise applications hosted in the cloud to be data-intensive. Also, the application data is dynamic in nature i.e., there are regular updates with the requests needing the most recent. A typical example is a banking solution, where a money transfer from one bank to another results in a multitude of checks or changes to occur for data from one bank account or changes to the stock prices of a company could result in a large number of requests in mutual fund applications to data from the company.

The main contribution of this work are:

- We investigate three variants of the penalty function: linear, exponential and step-wise. We utilize a heuristic request management policy (based on simulated annealing) which is shown in [12] to maximize response time percentiles in comparison to variants. This scheme is independent of the penalty function utilized and is thus suitable for evaluating the variants.
• We propose DSA$_g$S, a dynamic, distributed, adaptive, measurement based, data-aware, session-grained request management policy for geographically distributed datacenters hosting multiple SOA-based enterprise applications with percentile SLAs, aiming to decrease the penalty charged to the cloud computing system.

• We perform extensive evaluations to demonstrate that our scheme provides improved allocations and schedules of requests at end-servers that aim to decrease the total penalty charged to the cloud globally, in comparison to the commonly deployed allocation and scheduling schemes.

• We propose a cache replacement policy for SOA-based enterprise applications and evaluate the improvement of the percentiles achieved, owing to the replacement policy.

This report is organized as follows. In Chapter 2, we analyze percentile Service Level Agreements to find the penalty function that results in the highest number of requests to meet the response time. In Chapter 3, we present and evaluate DSA$_g$S, our data-aware, session-grained request management policy for minimizing penalties charged to the cloud as per the percentile SLA negotiated. In Chapter 4, we present future work.
Chapter 2

Percentile Service Level Agreements

In this chapter we analyze percentile Service Level Agreements. From the cloud consumer’s point of view, percentiles of response times, as opposed to averages, provide a better control of variability in delays experienced by user requests. Controlling the percentiles enforces a “shape” on the entire distribution of response times, and thus goes beyond simply bounding the average delay; this capability is deemed more suitable for (near real-time) SOA applications.

As mentioned in section 1.5, we consider SLAs that involve percentiles of response times as part of the performance metrics; the SLAs stipulate that a penalty be charged to the cloud provider if the SLA targets are not met. The main motivation for considering such SLAs is their potential for price differentiation.

We focus our analysis on the effects the penalty function has on the achieved response time percentiles. In particular, we analyze the effect of three commonly deployed choices (linear, exponential or step-wise functions) to relate the penalty charged and the achieved percentile. This analysis is NP-hard, so we employ a heuristic algorithm that is based on simulated annealing.

2.1 Service Level Agreements

As mentioned in section 1.5 a Service Level Agreement (SLA) formally specifies terms and conditions negotiated between a service provider and a service consumer. Any SLA, typically lists the services offered by the cloud provider, the terms and conditions under which the service will be offered, quality of the service offered under those terms and the prices of offering the service of a certain quality under stated conditions to the provider. SLAs may also include penalties charged to the provider, if the service of negotiated quality is not offered under the stated terms to the consumer.

The SLA may involve a variety of metrics to evaluate the quality of service offered, called Key Performance Indicators (KPI), that describe the service; typical examples of KPIs include
availability, security, reliability, and delays.

Service Level Agreements have been employed in the IT industry (e.g., Internet Service Providers, telecommunication providers, technical support providers). In the telecommunication industry, in particular, they have been used extensively for more than 30 years. More recently, they are being used in the cloud computing industry, for offering and pricing services such as Software as a Service (SaaS), Platform as a Service (PaaS) and Infrastructure as a Service (IaaS).

SLAs are also heavily used when federated cloud services are offered by multiple providers to consumers. SLAs in cloud federation are complex as they would involve terms of service and pricing between multiple providers that would depend on the service offered collectively by all providers to the consumer; they also include KPIs negotiated between each provider and each consumer [35].

There challenges in management of Service Level Agreements in SaaS environments include and are not limited to:

1. **Automated service level agreements**: A critical issue in the area of cloud computing is for cloud consumers and providers to effectively and automatically negotiate service level agreements. This automation is extremely beneficial for reducing total costs of ownership of the applications, when there is dynamism involved in the workload rate, resource availability and costs charged for service hosting. There is a lack of supporting frameworks and standards to automatically and dynamically negotiate agreements aiming at fulfilling end-to-end quality of service requirements in a cloud [54].

2. **Metric or KPI selection**: SOA-based applications are inherently dynamic and are most commonly adopted by industries whose businesses are in a state of flux. In such cases, selection of KPIs and acceptable metric values is crucial and change frequently. The choice of KPIs to be included in the SLA is important for the consumer as it impacts both business and technical performance and for the provider who can better provision resources on the basis of the metric values.

3. **Monitoring in the cloud**: SLAs in the SaaS delivery model include application and resource usage metrics at a very low granularity, in terms of disk usage, end-to-end transaction time, remote repository data access times. Cloud providers also need to monitor these application artifacts and current resource usage for detection of SLA violation [32]. This monitoring is straightforward if the application were deployed in a dedicated data-center. However, deployment of applications in a cloud is based on time-sharing principles where a resource can be shared by multiple applications and data access can be grouped across consumers for efficiency, accurate monitoring of resources and application artifacts at low granularity is a challenging area of research [13].
As mentioned previously in section 1.5, in the SLAs analyzed in this work, the provider is the cloud provider who offers resources provisioned on-demand and manages hosted applications as a service; the consumer is the SaaS vendor whose applications are hosted in the datacenters. Typical KPIs of interest to the SaaS vendors include availability of the software to their end-customers, response times, data capacity (if storage is required) and so on. In this work, we focus on percentile response times as the KPI metric of interest in the SLA negotiated between the provider and the consumer.

2.2 Response time percentile Service Level Agreements

In this section, we define the response time percentile SLAs more precisely and detail the three penalty function variants one can use in such SLAs.

2.2.1 SLA Parameters

We identify the following parameters of interest in the percentile SLA:

Response time, $R$

The response time we consider in this work is defined as the duration of the time it takes from when a request first arrives at the datacenter and to the time the response for the request leaves the datacenter. The request is said to “complete” once the response leaves the datacenter. The request could involve invoking a composition of multiple service end-points across various applications and/or request functionality across multiple tiers of the same application. The response time includes the queuing and execution delays at each component of the application. $R$ can be monitored by both the consumer and the cloud provider.

Response time threshold, $R_T$

The response time threshold, $R_T$, is the “deadline” by which a request must leave the datacenter. The value of $R_T$ is negotiated between the consumer and the cloud provider. We say that a request is “conforming” if $R \leq R_T$, i.e., if the request completes within the negotiated response time.

Desired conforming percentile, $X_T$

The desired conforming percentile, $X_T$, is the fraction of requests for which the consumer wishes execution to be completed by the “deadline”, i.e., for which the cloud provider must ensure that $R \leq R_T$. The value of $X_T$ is negotiated between the consumer and the cloud provider.
Actual conforming percentile, $X$

The actual (or simply the) conforming percentile, $X$, is the fraction of requests for which the cloud provider has ensured that $R \leq R_T$. The value of $X$ is controlled by the request management policy chosen by the cloud provider.

Observation interval, $T$

The percentile SLA includes a period of time called the observation interval over which the fraction of requests conforming to the negotiated response times is computed and penalties (if any) are charged for the observation interval. For example, a typical percentile SLA could require 70% of requests of a particular application to have a response time of less than 10 milliseconds, computed over an observation interval of 10 days.

Penalty, $P$

The percentile SLAs considered here call for economic penalties if the fraction of conforming requests is less than a certain value. The SLAs also define the method of calculating the penalties. In this work we consider two ways of penalty calculation:

- Per-request penalties: If the fraction of conforming requests is less than a given value, the penalty is calculated in proportion to the number of requests contributing to the reduction in conformance. For example, suppose that the percentile SLA requires $X_T = 70\%$ of the requests of a particular class to have a response time of less than $R_T = 10$ milliseconds. Out of the 1000 requests that arrive at the datacenter only 600 requests conform i.e. have a response time of less than 10 milliseconds resulting in a conformance percentile of $X = 60\%$. Penalty of $P$ is charged on each of the 100 requests resulting in reduction in the conformance percentile. In this case, penalties depend on the number of requests sent to the datacenter.

- Per-fraction penalties: If the fraction of conforming requests is less than a given value, the penalty is calculated in proportion to the fraction of requests contributing to the reduction in conformance. For example, consider the percentile SLA requires $X_T = 70\%$ of the requests of a particular class to have a response time of less than $R_T = 10$ milliseconds. Out of the 1000 requests that arrive at the datacenter only 600 requests conform i.e. have a response time of less than 10 milliseconds resulting in a conformance percentile of $X = 60\%$. Penalty of $P$ is charged on the fraction of 10% resulting in reduction in the conformance percentile. In this case, penalties do not depend on the number of requests sent to the datacenter.
2.2.2 Percentile SLA definition

Based on the above parameters, an SLA using percentile of response time KPIs can be precisely stated as follows: “Over the observation interval of $T$ time units, at least $X_T\%$ of requests of the application should have a response time less than $R_T$ time units, else a penalty $\$P$ is charged to the datacenter provider.”

2.2.3 Enforcement of the SLA

The cloud provider has multiple control mechanisms at his/her disposal to monitor and control such an SLA. For example, among the most common ones would be allocation of requests to an available server (e.g., request allocation) and allocation of CPU time to requests inside a server (e.g., scheduling). These controls are collectively called the request management policy. The policy is chosen to satisfy management goals. A typical goal in datacenters would be to determine a request allocation and scheduling algorithm that provides the minimum in Equation 2.1 below:

$$\min \sum_{1 \leq k \leq K} \text{penalty}_k$$

where $\text{penalty}_k$ is the penalty charged for non-conformance of the requests from users of application $k$; $K$ is the total number of SOA-based applications the cloud is hosting.

2.2.4 Variants of penalty functions

In typical response time percentile SLAs, the penalties charged ($\$P$) are obtained as a function $P(X)$ of the fraction of conformance ($X\%$). In this section we present three forms of penalty functions that have been used as parts of SLAs.

Linear penalty function

The Linear penalty function shown in Figure 2.1 is formally defined as:

$$P(X) = \begin{cases} P_{\text{max}} - X \cdot m, & \text{if } X \leq X_{\text{zero}}, \\ 0, & \text{if } X_{\text{zero}} < X \leq 100. \end{cases}$$

where $P_{\text{max}}$ is the penalty charged when the conformance is 0%, $m = P_{\text{max}}/X_{\text{zero}}$, $X_{\text{zero}}$ is the percentile (%) conformance at which the penalty charged is zero.

Exponential penalty function

The Exponential penalty function shown in Figure 2.2 is formally defined as:
Figure 2.1: Linear penalty function.

$$P(X) = P_{\text{max}} \cdot e^{-\lambda \cdot X},$$

where $P_{\text{max}}$ is the penalty charged when the conformance is 0%, $\lambda = \ln(10)/X_\alpha$, and $X_\alpha$ is the percentile at which the penalty should be $(1/10) \cdot P_{\text{max}}$ ($X_\alpha$ is similar to the half-life value in exponential decaying of unstable atoms, except we set the value to $(1/10)\text{th}$ the maximum instead of $(1/2)$).

Step-wise penalty function

The Stepwise penalty function is shown in Figure 2.3 (for the special case of $n = 3$ steps). The function is defined via $n$ pairs $(P_i, X_i)$, where $P_i$ is the penalty charged when the conforming percentile is $X_i$.

2.2.5 Selection of penalty functions

The linear function is the most intuitive and commonly used in pricing models in SLAs across industries [9]. Step-wise functions are heavily used in pricing models as they enable categorization of a range of service parameter values into groups or classes having the same utility value, making it essential to analyze the effect of a utilizing a piecewise linear function in percentile
SLAs [55]. We have chosen the exponential function as a model “in-between”.

The main focus of this work is to analyze the effect of the penalty function on the achieved percentile conformance, under a given request management policy. In the next section, we present the evaluation methodology used to compare the three variants of the percentile SLA described.

### 2.3 Evaluation methodology

The main criterion for comparison is the total number of requests that have a response time less than that specified in the SLA negotiated, when the “best known” allocation and scheduling policy is used for minimizing penalty in each of the three percentile SLAs. The problem we consider belongs to the class of problems encompassing utility-maximizing schemes for allocation and scheduling of multi-class jobs to multiple resources. This class of problems has been proven to be NP-hard [11]. The evaluation methodology is thus based on heuristics. Also, due to the NP-hardness nature of the problem, it is not possible to analytically prove optimality of the proposed heuristics-based solution.

In order to evaluate the three variants, we adopted a methodology based on simulated annealing. Simulated annealing is a probabilistic metaheuristic method for finding the global minimum of a cost function with many local minima. It works by emulating the physical
process whereby a solid is cooled slowly and eventually the object “freezes” at the lowest energy configuration [18]. Following this analogy, at each step of simulated annealing, the current solution is replaced by a “nearby” solution which is chosen based on a probability that depends on the difference between the current function value and the new function value and a global parameter called the temperature that is gradually decreased in each step. The dependency on the temperature is such that the “nearby” solution is random when temperature is large, but randomness decreasingly “descends” as the temperature tends to zero. The randomness allows for uphill movements which prevents searching only at a particular local minimum, thus resulting in a better solution than greedy methods. This characteristic of simulated annealing based solution search technique makes it a good fit for evaluation of the three percentile SLAs as the solution will not be dependent on the shape of the cost function.

Algorithm 1 describes the heuristic simulated annealing based allocation and scheduling scheme used for evaluation of the three percentile SLA variants. We assume a work conserving system. If all end-servers are busy, requests are queued at the datacenter as they arrive. A request from the queue is allocated to an end-server as soon as it is free. The request to be serviced is chosen by a simulated annealing based heuristic described in Algorithm 12. In [12], we have shown that simulated annealing based scheduling schemes result in maximizing utility of percentile response times. In the iterative scheme adopted, the first seed schedule we begin with is First Come First Serve (FCFS), a global start temperature, end temperature and step
Algorithm 1 Allocation and scheduling of requests.

Input: Number of servers in the datacenter: $N$

Incoming requests are inserted at the end of the request queue at the datacenter

while Queue of requests is not empty do
  if any server $i$ in $N$ is free then
    $\text{best\_schedule} =$ Schedule found in Algorithm 12
    Allocate request at the head of $\text{best\_schedule}$ to server $i$
  end if
end while

values are chosen ($t, \epsilon, \alpha$ respectively in Algorithm 12 that determine the number of iterations). In each iteration, two requests from the queue are chosen at random and their positions are interchanged in the schedule and the penalty ensuing from the resulting schedule is calculated. If this schedule is the lowest obtained, the schedule is saved as the final schedule. This schedule now becomes the seed schedule in the next iteration. If the penalty obtained is not the lowest obtained among all the iterations, then the probability that the schedule is used as the seed schedule in the next iteration, depends on the difference of the penalty obtained from that obtained in the previous iteration and the current value of the global temperature variable. At the end of the iterations, the final schedule is the one which has resulted in the least penalty. The request at the head of the final schedule is allocated to the end-server.

This scheduling scheme is compute-intensive and is not practical to implement in real-world deployments. However, since this work focuses on analysis of the three variants of the penalty functions, we chose the scheduling policy which results in the “best” schedules (Appendix A) for percentiles.

In the next section, we evaluate the effect of the three penalty functions based on the methodology described above.

2.4 Evaluation results

We have built a discrete event simulator for the analysis of the choice of the penalty function when the best-known request allocation and scheduling scheme based on simulated annealing technique is employed for minimizing penalties charged. This section details simulation results to answer the following top-level questions:

1. Do we have a clear winner among the 3 variants of the response time percentile SLA considered?

2. Is there a difference between conformance in the per-request penalties and per-fraction penalties in each of the 3 variants considered?
Algorithm 2 Simulated annealing based heuristic algorithm for best schedule resulting in least penalty.

Input:
- Number of requests queued: \( R \)
- Observation interval: \( T \)
- Starting temperature: \( t \)
- Ending temperature: \( \epsilon \)
- Step-value: \( \alpha \)
- \( \text{final}_\text{penalty} \): Infinity
- Seed schedule of requests: \( \text{current}_\text{schedule} \)

Output: Schedule resulting in lowest penalty: \( \text{final}_\text{schedule} \)

while \( (t > \epsilon) \) do
    \( \text{old}_\text{penalty} \) = penalty resulting from \( \text{current}_\text{schedule} \)
    \( \text{new}_\text{schedule} \) = Interchange positions of 2 randomly chosen requests in \( \text{current}_\text{schedule} \)
    \( \text{new}_\text{penalty} \) = penalty resulting from the \( \text{new}_\text{schedule} \)
    if \( (\text{new}_\text{penalty} < \text{final}_\text{penalty}) \) then
        \( \text{final}_\text{penalty} = \text{new}_\text{penalty} \)
        \( \text{final}_\text{schedule} = \text{new}_\text{schedule} \)
        \( \text{current}_\text{schedule} = \text{new}_\text{schedule} \)
    else
        if \( (\text{random}_\text{number} < e^{(\text{new}_\text{penalty} - \text{old}_\text{penalty})/t}) \) then
            \( \text{current}_\text{schedule} = \text{new}_\text{schedule} \)
        end if
    end if
    \( t = t \times \alpha \)
end while

3. What is the relation between step-size and observation interval in step-wise SLA?

A representative scenario involves a datacenter with (a) 10 end-servers in each datacenter, (b) 5 distinct applications with distinct values of the different SLA parameters for each of the 3 variants, (c) the input arrival process is Poisson, (d) the service processes are exponential.

2.4.1 Comparison of conformance in the 3 variants

Typical results in Figure 2.4 (with 95% confidence intervals) show that linear percentile Service Level Agreement consistently results in higher number of conforming requests even as input request rates are increased. Step-wise SLAs with lower number of steps results in the lowest number of conforming requests. However, if the number of steps are increased (essentially more accurately approximating linear SLA), the number of conforming requests increase.
### 2.4.2 Difference in utilizing per-request and per-fraction penalties

We now compare the effect of utilizing per-request and per-fraction penalties. As shown in Figure 2.5 (with 95% confidence intervals), for linear and exponential response time percentile SLAs, per-request penalties result in higher conformance. This is as expected as the penalties are charged at a lower granularity resulting in improved adaptability of the management scheme to the dynamic nature of the system. However, for step-wise percentile SLAs the improvement in utilizing per-request percentile SLA is lower than that in the former penalty functions.

### 2.4.3 Relation between step-size and observation interval

Figure 2.4 shows that as the number of steps in the step-wise interval increase, thereby approximating the linear SLA, the number of conforming requests increase. In Figure 2.6 (with 95% confidence intervals) we show that, as the length of the observation interval increases, the effect of a smaller number of steps in the step-wise SLA reduces resulting in larger percentile of conforming requests. The observation interval is expressed as the number of times the average service time of an end-server in the datacenter. This representation of the observation interval is chosen as the average service time the total number of requests served over a time period and the observation interval of the SLA should be large enough to incorporate a high number
of completed requests to accommodate variations in the input rate.

2.5 Related work

As mentioned in previous sections, to the best of our knowledge, the present work is the first effort towards analyzing the effect of the choice of the penalty function on the total number of requests that have conformed to their negotiated response time. However, there have been several past research efforts that have focussed on management schemes to meet percentile Service Level Agreements. Agarwal et al. [5] propose a scheduling scheme called gi-FIFO and analytically prove that it is the best schedule for multi-class jobs in a single server to maximize response time percentiles; Cardellini et al. [15] show that enforcing percentile performance criterion as a management objective, results in better conformance and improved response times in comparison to average performance criterion guarantees and present a brokering service for management of composite services under percentile-based SLAs; Xiong et al. [53] provide an analytical solution of resource optimization subject to percentile response time and price by modelling the system as an overtake free open tandem queuing network with feedback and provide closed form expressions of the probability distribution function of the response time; Gmach et al. [22] consider step-wise percentile SLAs and propose scheduling algorithms for a single database server. Schroeder et al. [44] propose a framework for providing class-based QoS
targets in transactional workloads, where the QoS targets are response time percentiles.

2.6 Summary

We have considered response time percentile Service Level Agreements that call for economic penalties if percentile response time targets are not met. We analyzed three variants of the penalty functions: linear, exponential and stepwise. We identified that the choice of the penalty function results in variations in the total number of conforming requests when the “best known” allocation and scheduling request management policy is employed. We proposed a penalty minimizing, simulated annealing based request allocation and scheduling heuristic scheme to evaluate the three variants of the percentile response time SLA. Our simulation based evaluation shows that the linear variant results in the highest number of requests that have conformed to the negotiated response time threshold.

Our future work involves analyzing the three variants of the response time percentile SLA when a clairvoyant scheduling policy is employed. Recently, research efforts have been directed towards pro-active request management schemes that maximize utility [49], essentially developing clairvoyant systems. The analysis of the improvement (or lack of) in conformance when clairvoyant solution is used in each of the three variants of the percentile SLA aims to determine the variant of the penalty function that benefits the most from adoption of clairvoyant solutions.
Chapter 3

Management of SOA-based applications

We consider a cloud computing system consisting of multiple geographically distributed datacenters, each with a large number of end-servers. The centers collectively host, multiple SOA-based enterprise applications. The applications are each negotiated with percentile SLAs. We address the problem where the request management techniques employed by the cloud should aim at globally conforming to the percentile SLAs negotiated for each application, thus minimizing the penalty charged.

3.1 Problem formulation

3.1.1 System architecture

The following are the key elements of the system:

- **Clients.** These are nodes that generate the service requests forwarded to the datacenters through the internet.

- **Datacenters.** A datacenter is a cluster of a large number of networked computing resources. In the topology considered, multiple geographically distributed datacenters form the cloud computing system with each datacenter hosting the same set of SOA-based applications. The requests for an application is served by the datacenter assigned to the end user’s primary geographical location. This is a common assumption as sensitive data of a customer would need to be contained within a certain geographical boundary. The system architecture is as shown in Figure 3.1.
3.1.2 SOA-based data-intensive enterprise applications

We consider SOA-based enterprise applications. We identify two characteristics common to a large set of enterprise applications: 1) Temporal order constraints and data dependencies exists among requests issued from the same end-point (user/client application). 2) Data-aware allocation enables reduced response times.

As an example, consider an algorithmic trading or High-Frequency Trading (HFT) application that enables real-time investments. A typical request for such an application could require latest information about stocks of multiple companies from different stock exchanges and related information from risk management agencies [45]. Based on the most recent data, the application continually provides suggestions for investments.

The application described above needs to be “context-aware”, with responses tailored to reflect the current state of multiple aspects of the environment (the latest stock information from different exchanges, latest risk information from management agencies). This translates to the requirement of loading large amounts of context data from multiple sources to an end-server for servicing requests [46], which results in increased response times. Typically, the interval between consecutive updates to a context in e-business enterprise applications is much larger than inter-arrival times of requests for the same context. Data-aware allocation for such applications result in reduced response times, by allocating requests to the end-server that has the required context data cached, resulting in zero context data load times(example of the characteristic described earlier in this section).
In the HFT application considered, for accurate execution of the trading algorithms, bids/offers placed/enquired by a particular user has to be executed in the same order as the requests were received (temporal order constraint). In this work we refer to requests from the same customer/subject as belonging to the same session.

3.1.3 Deployment pattern of SOA-based applications in a data-center

As shown in Figure 3.1, we consider multiple geographically distributed datacenters to form a collectively managed cloud computing system. Each datacenter is capable of serving requests from one or more of the SOA-based applications. In this section we describe the deployment pattern of SOA-based context-aware applications in a datacenter (Figure 3.2). Below are the important features of the application model:

- Built on Service Oriented Architecture (SOA) principles: Multiple portable, duplicable functional service-endpoints form the application. These service-endpoints can be instantiated in any end-server in a datacenter. Requests from clients are XML serialized.

- Multiple context data store access: Each request needs most updated status/information from multiple objects. Thus multiple context datastores with data from different objects are automatically updated with the latest information as shown in (*) in Figure 3.2. For simplicity, we depict all required context data stores to be co-located within a data center.

- Session data store: Each request is typically made by a registered user in the application whose historical data is stored in the session datastores. Note as depicted in Figure 3.2, the number of session datastores is usually much smaller than the number of context datastores and are typically co-located within a single datacenter due to security/privacy concerns.

We now explain the typical operation that needs to occur when a request for the enterprise application considered, is received (Steps 1 - 5 in Figure 3.2)

1. A request for the enterprise application is sent over the internet by the end-user.

2. The request is allocated to an end-server by the middleware.

3. The required service end-point for the functionality requested by the user is loaded on the chosen end-server.

4. The required session information is loaded on the chosen end-server from the session datastore. Subsequent requests within the same session are dependent on data from the output of processing previous requests from the same session (temporal order constraint).
SOA-based data-intensive applications with multiple portable functional service end-points (represented by interconnected circles)

1. Request from end-user arrived
2. Request allocated to end-server
3. Required services loaded at end-server
4. End-user session loaded at end-server
5. “Context-data” loaded at end-server

Middleware

DATA CENTER

End servers

Internet

User session datastores

Context datastores

* Updates to “context-data”

Figure 3.2: Deployment pattern of a SOA-based enterprise application in a datacenter.

5. The required context information from multiple context datastores is loaded on the chosen end-server. Multiple context datastores with data from different sources, are automatically updated with the latest information as shown in (*) in Figure 3.2.

Let us now correlate the above given operation to a concrete example. Consider a SOA-based CRM application deployed in the geographically distributed cloud computing system. A typical use case scenario of the CRM application would be to log complaints/requests from
users. Let us assume that a user (say user A) logs a complaint on the non-availability of a component (say component B). When the request from this user arrives at the datacenter (step 1), the middleware first has to allocate the request to an end-server (step 2). The required service-endpoint from the CRM application catering to the “complaint” functionality has to be instantiated/loaded on the chosen end-server (step 3). The user’s session information (For eg., users login information, history of complaints filed etc.) has to be loaded onto the chosen end-server (step 4). Data about the component that the user is filing a complaint about or the “context-data” under which the request is operating has to be loaded from the context-data store (step 5) onto the chosen end-server. This context-data could typically be information about the current status of the component, history of previous breakdowns, time since previous checkpoint etc. This data is dynamic in nature and is loaded independently to the context-data stores (step 6). (In this work we do not deal with the updates to the context-data). Once the required data is loaded, the request is executed on the chosen end-server.

Assume, that the user A, sends a second request updating the status of the complaint from “severe” to “critical” and edits information about the delivery of updates from the CRM application. If the second request is allocated to the same end-server as the first, the user’s session information need not be loaded from the session-datastore. As a consequence, the response time for the second request will be lower. In this work, requests from the same user/identifier is considered to belong to a “session”. Management policies catering to most SOA-based applications need to be “session-aware”.

Assume, that another user (say user B), sends a request to the SOA-based CRM application, filing a complaint on the non-availability of component A. Also assume that the status of the component has not changed prior to the arrival of the complaint from user B. If this request is allocated to the same end-server as the first request from user A, then, the context-data need not be loaded onto the end-server, resulting in lower response times. This scenario is to illustrate that under the assumption that the interval between updates to the context-data is much lower than the interval between requests for the context-data, data-aware allocation reduces response times.

A SOA-based application may serve multiple classes of users typically classified using economic considerations. User classes that are designated more important than others need to be guaranteed a higher quality of service. So, the enterprise applications hosted by the cloud will need differentiated quality of service whose terms are negotiated in the Service Level Agreement.

A typical cloud computing system will provide hosting services to multiple software providers. In accordance, the geographically distributed cloud computing system we consider, hosts multiple enterprise applications, each of them serving multiple classes of end-users.
3.1.4 Percentile SLA

We consider both linear percentile SLA and stepwise percentile SLA in this work.

**Linear percentile SLA**

The formal definition of the linear percentile SLA is as below:

\[
P(X) = \begin{cases} 
  P_{\text{max}} - X \cdot m, & \text{if } X \leq X_{\text{zero}}, \\
  0, & \text{if } X_{\text{zero}} < X \leq 100.
\end{cases}
\]

where \( P_{\text{max}} \) is the penalty charged when the conformance is 0%, \( m = P_{\text{max}} / X_{\text{zero}} \), \( X_{\text{zero}} \) is the percentile (%) conformance at which the penalty charged is zero.

As shown in the Chapter 2, adoption of linear percentile goals results in the management policies performing the “best”.

**Stepwise percentile SLA**

We also consider a step-wise SLA where a penalty is charged to the cloud if a certain percentile of requests are not executed within a certain response time as shown in Figure 3.4. The SLA consists of multiple steps with each step associated with a percentile and a penalty value.
As the fraction of requests meeting the response times increase beyond the percentile at the first step, the penalty reduces to that of the second step and so on. The penalty is zero, if the fraction of requests meeting the response time increases beyond the percentile at the last step. The conformance levels, i.e., the percentile of requests which have met the required response times is measured over a certain previously negotiated observation interval. The SLA is global in definition, i.e., all the datacenters in the cloud have to collectively respect the SLA.

![Figure 3.4: Multi-step percentile Service Level Agreement](image)

The formal description of the SLA we consider is as follows: Let $n$ be the number of steps in the SLA ($n = 3$ in Figure 3.4). If the percentile of requests that have response time less than $r$ seconds is less than or equal to $X_s\%$, then the cloud is charged a penalty of $SP_s$, for $s = 1$ to $n$. If the percentile of requests that have the response time less than $r$ time units is greater than or equal to $X_n\%$, then no penalty is charged to the cloud. The percentile of requests are measured over a certain fixed observation interval ($T$ seconds) and the penalty (if any) is charged for the duration of the observation interval.

The reason for adoption of the stepwise percentile SLA is to offer different classes of customers, distinct percentiles response times.
3.1.5 Problem statement

We want to determine a request allocation and scheduling algorithm that provides the minimum in Equation 3.1 below:

$$\min \sum_{1 \leq z \leq Z} \sum_{1 \leq j \leq C_z} \text{penalty}_{zj}$$  \hspace{1cm} (3.1)$$

where $\text{penalty}_{zj}$ is the penalty charged for non-conformance of the requests from users of class $j$ of application $z$; with the cloud hosting $Z$ a SOA-based enterprise applications each serving $C_z$ classes of users.

3.1.6 Objectives

In summary, the objectives of the request management policy are:

Objectives of the request allocation algorithm

A service request arriving at a datacenter has to be allocated to a particular end-server. The objectives of the request allocation algorithm employed are:

1. performs session-grained allocation to satisfy data dependency between successive requests in the same session,
2. performs differentiated allocation based on the current conformance levels of the requests from different user classes of different applications in order to minimize penalty charged to the cloud,
3. performs data-aware allocation,
4. balances load among the end-servers.

Once allocated to an end-server, the request has to be scheduled at the end-server.

Objectives of the request scheduling algorithm employed at the end-server

The objectives of the request scheduling algorithm at the end-server include:

1. maintain strict temporal order constraints between requests of the same session,
2. performs differentiated scheduling based on the current conformance levels of the requests from different user classes of different applications in order to minimize penalty charged to the cloud.

We make no assumptions about any prior knowledge about the number of requests arriving at the cloud computing systems for different applications or at different data centers.
3.1.7 Solution approach

We consider a reactive approach in solving the problem. Reactive approach is feedback based, with periodic sensing of the current status of system parameters and adapting to the changes as shown in the figure 3.5.

![Figure 3.5: Solution approach](image)

3.2 Reactive management policy description

The problem we consider belongs to the class of problems encompassing utility-maximizing policies for allocation and scheduling of multi-class jobs to multiple resources. This class of problems has been proven to be NP-hard [47]. We propose DSA\textsubscript{gS} (Data-aware Session-grained Allocation with $gi$-FIFO Scheduling), a heuristic based management policy with an aim to reduce the penalty charged to the cloud. Since the problem is NP-hard, analytical proof that
our policy achieves the least possible penalty cannot be derived.

The management policy primarily constitutes of two main component algorithms and two sub-component algorithms. The two main component algorithms includes the request allocation algorithm and the request scheduling algorithm. The former is referred to the algorithm in which an arriving request is allocated to a suitable end-server and the latter refers to the algorithm employed for scheduling requests at the end-servers on all the requests queued. The main goal of the two actions is to meet the objectives in Section 3.1.6 and minimize the penalty charged to the cloud. The two subcomponents include session reallocation and context cache replacement algorithm which aim at further reducing the penalty charged to the cloud.

3.2.1 Periodic exchange of metrics

Since the penalty charged to the cloud is global in nature, each datacenter must be aware of the conformance levels of different classes achieved by all the other datacenters forming the cloud, in order to take suitable actions during allocation and scheduling requests. Recall that the penalty is charged for different classes, based on the conformance levels measured over an observation interval as given in Section 3.1.4. We propose to divide the observation interval into multiple subintervals at each datacenter as shown in Figure 3.6. At the beginning of each subinterval, each datacenter exchanges the conformance levels of different classes with the peer datacenters that form the cloud. This exchange enables each datacenter to calculate the “current” global conformance levels of different classes and the corresponding “current” penalty for each class. With “current” penalty, we mean the penalty charged to the cloud if the conformance level of the class is considered at the current instant.

This periodic exchange of conformance levels at every subinterval at every datacenter is detailed in Algorithm 3.

![Figure 3.6: Observation interval divided into multiple subintervals in each datacenter](image-url)
Algorithm 3 Conformance exchange phase at datacenter $d$

**Input:**

- Number of geographically distributed datacenters: $N$
- Number of user classes of requests: $K$
- Number of requests serviced of user class $k$ in datacenter $l$ in current subinterval: $X_{s_{lk}} \forall k \in \{1, \ldots, K\} \forall l \in \{1, \ldots, N\}$
- Number of requests of class $k$ in datacenter $l$ which met the required response time $r$ in current subinterval: $X_{r_{lk}}$
- Total number of requests serviced of class $k$ from the start of the observation interval as measured by $d$: $X_k$
- Current conformance of class $k$ in cloud as calculated by datacenter $d$: $cc_k$  /*At the first subinterval of any observation interval, $cc_k = 0 */$

**Output:** Updated current global conformance calculated by $d$: $cc_k \forall k \in \{1, \ldots, K\}$

```plaintext
for $k = 1$ to $K$ do
    temp = 0
    temp2 = $X_k$
    for $i = 1$ to $(d - 1)$ and $i = (d + 1)$ to $N$ do
        temp = temp + $X_{r_{ik}}$
        $X_k = X_k + X_{s_{ik}}$  /*Get updates from all other datacenters*/
    end for
    $cc_k = (cc_k \ast temp2 + temp)/X_k$  /*Updated conformance level*/
end for
```

3.2.2 Component 1: Request allocation algorithm

The allocation algorithm formalized in Algorithm 4 meets the objectives in Section 3.1.6 in the following fashion: Successive requests of the same session are always allocated to the same end-server. As mentioned in Section 3.1.2, each request from a customer belongs to a session. Typically, each request of the context aware application we consider, will hold a unique id identifying the end-user [24]. This unique id is termed as session-id in this work. To allocate requests of the same session to the same end-server, we propose a hash-based lookup table containing tuples of the form $(session_{id}, server_{id})$. Each tuple in the lookup table represents a particular request(s) with session-id $(session_{id})$ currently allocated on a particular end-server $(server_{id})$. On each arriving request, the lookup table is queried to check if the $session_{id}$ of the arriving request is present. If the $session_{id}$ is present in the lookup table along with the corresponding $server_{id}$, the request is allocated to the same end-server as in the tuple. The tuples are removed once requests with the same session-id are not received at the datacenter within a predetermined duration called the “session-validity period” which is a configurable parameter.

There are a large number of efficient implementations of lookup-tables, especially targeting
IP routing tables in literature [31]. The requirements of our lookup table are parallel to those of the IP routing tables. We consider a hash-based implementation of a dynamic lookup table with constant lookup $O(1)$.

Allocating all requests of the same session to the same end-server meets the objectives of data dependency (objective 1 in Section 3.1.6).

If the request’s session id is not found in the session lookup table (first request of the session), we perform differentiated data-aware allocation (meeting objectives 2 and 3 in 3.1.6), as described in the following:

1. Data awareness is ensured by querying a hash table to find the set of end-servers currently loaded (cached) with some or all of the context data required by the incoming request. This set is termed as the “context-set”. (Algorithm incurs an overhead of $\sum_{1 \leq c \leq C} m_c$, where $C$ number of contexts required by the incoming request and $m_c$, number of end-servers loaded with the required context $c$)

2. For all end-servers in the context-set, a “compatibility” test is performed to estimate if the incoming request can meet its response time in the end-server. If yes, the end-server is said to be compatible with the request. The compatibility test is based on the current penalty of the class of request and its deadline. For example, a request of class with the highest current penalty is compatible in all end-servers or a request of any class with very high deadlines, might also be compatible in some of the end-servers that are lightly loaded. The compatibility test (complexity $O(\sum_{1 \leq i \leq M_s \log n_i})$, $M_s$ is the number of servers in the context-set $s$, $n_i$ is the number of requests of the particular class of request queued at the end-server $i$) derives heavily from the scheduling algorithm described in Section 3.2.4. Compatibility test is an estimate of whether the request can meet its deadline. We assume a priority queue-based data structure used to store the metadata of the requests of each class, the request is essentially inserted into this tree and the time at which the request completes processing is estimated. If the deadline of the request can be met, then the end-server is said to be compatible.

3. Among all the end-servers found compatible, the request is allocated to the one with the most number of required contexts cached, thus resulting in data-aware allocation.

4. The estimation of compatibility of a request with an end-server is greedy in nature. If a request of class with higher penalty is allocated to an end-server, then it may no longer be compatible for a previously allocated request with lower penalty.

5. If no end-server is found compatible, either due to the request belonging to the class with high current conformance or has very small deadlines, the request is allocated to the
least loaded end-server at the datacenter achieving load balancing. This means that the non-conformance of this particular request is allowed as it is not important enough for the cloud to allocate this request to the most suitable end-server.

Requests belonging to classes with zero current penalty charged to the cloud are always allocated to the least loaded end-server in the datacenter. This enables balancing of load at the various servers (meeting objective 4 in Section 3.1.6) but not at the expense of reducing response times of requests belonging to classes charging higher penalty to the cloud.

Thus, the running time of allocation of a request to an end-server could be either \( O(1) \) when the request belongs to a “valid” session else it is \( O(\sum_{1 \leq i \leq M_s} \log n_i) \) where \( M_s \) is the number of servers in the context-set \( s \), \( n_i \) is the number of requests of the particular class of request queued at the end-server \( i \).

The algorithm is depicted pictorially in Figure 3.7

Figure 3.7: Highlights of the request allocation algorithm
Algorithm 4 Request allocation algorithm.

**Input:** Request $i$ of class $k$ arrives at datacenter $d$
- Session id contained in request $i$: $s_{id}$
- Tuples in session lookup table for datacenter $d$: $(session_{id},server_{id})$
  /* Presence of tuple with session$id$ contained in the arriving request in the session lookup table, indicates that previous requests with the same session$id$ have been allocated to server$id$ */
- Number of servers in datacenter $d$: $M_d$

**Output:** End-server chosen for request $i$

Query session lookup table for tuple with session$id$: $s_{id}$
if $(s_{id}, n)$ found in lookup table then
- Server chosen for request $i$: $n$
  /*All requests of the same session have to be allocated to the same server*/
  exit
end if

if Current penalty due to class of request $i = 0$ then
- Server chosen for request $i$: Least loaded server in datacenter $d$
  exit
end if

$Numberofcontextsloded = 0$
for $m = 1$ to $M_d$ do
  if Compatibility test$(i, m)$ is TRUE then /*See compatibility test details in Algorithm 5*/
    if $C_{im} > Numberofcontextsloded$ then
      $Numberofcontextsloded = C_{im}$
      Server chosen for request $i = m$
    end if
  end if
end for
if No compatible server for request $i$ found then
- Server chosen for request $i$: Least loaded server in datacenter $d$
end if

3.2.3 Component 2: Request scheduling algorithm

We propose to model each end-server in a datacenter as having multiple queues, one for each user class as shown in Fig 3.8. A request when allocated is inserted into the queue for its class at the chosen end-server. The aim of the scheduling algorithm at the end-server detailed in Algorithm 6, is to maximize the percentiles of the class of requests whose current conformance levels would result in the highest penalty charged to the cloud. Since we adopt a greedy approach, the class with the highest “current” penalty is the “chosen” class. Since strict temporal order constraints need to be maintained among requests of the same session(objective 1 in Section 3.1.6), among the sessions queued of the “chosen” class, we choose the session whose first request has waited the longest but whose deadline can still be met. If no such session exists, we choose the session whose first request has waited the longest. The first request of
Algorithm 5 Compatibility test.

\textbf{Compatibility test}(Request \(i\), server \(m\))

\textbf{Input:} Number of classes: \(K\)

Class of request \(i\): \(k\)

Deadline of request \(i\): \(d_i\)

Current penalty levels of class \(x\): \(P_x \forall x \in \{1, \ldots, K\}\)

Number of requests in queue of class \(x\) in server \(m\): \(R_x \forall x \in \{1, \ldots, K\}\)

Estimated completion time of request \(j\) of class \(x\) queued in server \(m\): \(C_{j_x} \forall j \in \{1, \ldots, R_k\} \forall x \in \{1, \ldots, K\}\)

Number of requests in the queue of class \(k\) of request \(i\) with smaller satisfiable deadlines: \(R_{d_k}\)

\textbf{Output:}

TRUE if request \(i\) can be met in server \(m\), FALSE otherwise

\(Scheduledtime = 0\)

\textbf{for} \(x = 1\) to \(K\) \textbf{do}

\textbf{if} \(P_x > P_k\) \textbf{then}

\textbf{for} \(j = 1\) to \(R_x\) \textbf{do}

\(Scheduledtime = Scheduledtime + C_{j_x}\)

\textbf{end for}

\textbf{end if}

\textbf{end for}

\textbf{for} \(y = 1\) to \(R_{d_k}\) \textbf{do}

\(Scheduledtime = Scheduledtime + C_{y_k}\)

\textbf{end for}

\textbf{if} \(Scheduledtime < d_i\) \textbf{then} /* Request \(i\) can meet its deadline in the server \(m\) */

\textbf{return} TRUE

\textbf{else}

\textbf{return} FALSE

\textbf{end if}

the chosen session is scheduled. This method of choosing a class and then one session of that class, is based on the gi-FIFO scheduling algorithm which has been analytically proven in [5] to maximize percentiles for a single server serving multiple classes of jobs (meeting objective 2 in Section 3.1.6). In Appendix A, we describe gi-FIFO scheduling algorithm in detail and show the comparison of gi-FIFO to an exploratory search technique. Classes with low penalties never get “chosen” and are “starved”; these classes eventually incur higher penalties forcing the adaptive scheduling algorithm to choose the previously starved class.

Since we assume a priority-queue based data structure to store the metadata of requests, the worst case complexity of the gi-FIFO scheduling algorithm is \(O(\log s)\) where \(s\) is the number of sessions of the chosen class.

3.2.4 Component 3: Session reallocation algorithm

The objectives of the allocation algorithm include meeting the temporal constraints and data dependency between requests within the same session are maintained as given in Section 3.1.6.
In order to meet this objective, we allocate requests with the same session-id to the same end-server at which, the first request with that session-id was allocated, without checking for compatibility with the end-server (see Section 3.2.2). Thus to meet temporal constraints and data dependency between requests of the same session, data-awareness is ignored. This dependency is comparable to “allocation dependency” described in the context of HTTP sessions in web clusters in [47]; the authors show a marked reduction in response times due to this dependency. Similar to [47], in our simulations, we found that “session allocation dependency” results in reduced percentiles, details in Section 3.4.3. However, unlike in [47] where the authors propose a policy that considers the request count in a session prior to allocation, in our problem there is no prior knowledge about number of requests in a session and request allocation decisions are taken immediately on their arrival.

If it were found in an end-server, some or all of the requests of a session will not be able to meet their deadlines, we propose reallocation of the session to a more compatible end-server. This reallocation occurs periodically, the interval between successive reallocations is configurable. The reallocation period typically depends on the nature of the application. Sufficient workload characterization is required to estimate this session reallocation period. The periodic reallocation algorithm will migrate the session to the end-server in the datacenter capable of meeting the deadlines of the maximum number of requests of the session. Complexity of the session reallocation algorithm is $O(\sum_{1 \leq r \leq S} \sum_{1 \leq i \leq M_r} \log n_i)$ where $S$ is the number of requests in the session, $M_r$ is number of servers of the context-set for request $r$, $n_i$ is the number of requests queued in end-server $i$ belonging to the class of application of the session.)

This periodic migration is required, since the number of requests in a session is unknown at
Algorithm 6 gi-FIFO scheduling algorithm.

Input:
- Number of servers in datacenter: \( M_d \)
- gi-FIFO scheduling algorithm in server \( m \) of datacenter \( d \), \( \forall m \in \{1, \ldots, M_d\} \forall d \in \{1, \ldots, N\} \)
- Class with highest current penalty in server \( m \) : \( k \)
- Number of requests in queue of class \( k \) in server \( m \): \( R_k \) \( \forall k \in \{1, \ldots, K\} \)
- Deadline of request \( j \) in queue of class \( k \) in server \( m \): \( d_{kj} \) \( \forall k \in \{1, \ldots, K\} \forall j \in \{1, \ldots, R_x\} \)
- Current waiting time of request \( j \) in queue of class \( k \) in server \( m \): \( w_{kj} \) \( \forall k \in \{1, \ldots, K\} \forall j \in \{1, \ldots, R_x\} \)

Current datacenter time: \( t \)

Output:
- Request to be scheduled

\[
\begin{align*}
\text{waitingtime}_{gi} &= 0 \\
\text{waitingtime}_{wi} &= 0 \\
\text{request}_{gi} &= \text{NULL} \quad \text{/* Maximum waiting time among requests which can meet the deadline */} \\
\text{request}_{wi} &= \text{NULL} \quad \text{/* Maximum waiting time among requests which cannot meet the deadline */} \\
\text{for } j = 1 \text{ to } R_k \text{ do} \\
\quad \text{if } d_{kj} < t \text{ then} \\
\qquad \text{if } w_{kj} > \text{waitingtime}_{gi} \text{ then} \\
\quad\quad \text{waitingtime}_{gi} = w_{xj} \\
\quad\quad \text{request}_{gi} = j \\
\qquad \text{end if} \\
\quad \text{else} \\
\qquad \text{if } w_{kj} > \text{waitingtime}_{wi} \text{ then} \\
\quad\quad \text{waitingtime}_{wi} = w_{xj} \\
\quad\quad \text{request}_{wi} = j \\
\qquad \text{end if} \\
\quad \text{end if} \\
\text{end for} \\
\text{if } \text{request}_{gi} = \text{NULL} \text{ then} \\
\quad \text{Request to be scheduled} = \text{request}_{wi} \\
\text{else} \\
\quad \text{Request to be scheduled} = \text{request}_{gi} \\
\text{end if}
\end{align*}
\]

the time of the allocation of the first request of the session (Refer to Algorithm 4). The periodic reallocation algorithm described in Algorithm 7 will verify if the satisfaction of migration of a session holds. In Algorithm 7 we describe one such criterion we investigated where if any one request of the session does not meet its deadline the session qualifies for migration. Several other criteria are investigated and simulation results are presented in the Section 3.4. The session is then migrated to the server which is compatible with the most number of requests in the session.
Algorithm 7 Session reallocation algorithm

Input:
- Number of machines in datacenter $d$: $N_d$
- Set of distinct sessions in machine $m$: $S_m$
- Set of requests in session $s$ in machine $m$: $R_{sm}$
- Number of requests in set $R_{sm}$ incompatible in machine $m$: $Incomp_s$
- Penalty of class of requests in session $s$: $P_s$

1. $Incomp_s = 0$
2. For each session $s$ in $S_m$ do
   - If $P_s > 0$ then
     - $Incomp_s = \text{Number of requests not compatible}(R_{sm})$
     - If $Incomp_s > 0$ then
       - Migrate($R_{sm}, Incomp_s$)
     - End if
   - End if
- End for

- Number of requests not compatible($R_{sm}$)

  - $none = 0$
  - For each request $j$ in $R_{sm}$ do
    - If Compatibility test($j, m$) is FALSE then /* See Algorithm 5
      - $none = none + 1$
    - End if
  - End for

  - Return $none$ /* Multiple criteria can be used here*/

- Migrate($R_{sm}, Incomp_s$)

  - Number of machines in datacenter $d$: $N_d$
  - Number of requests of session $s$ in the set $R_{sm}$ that is compatible in machine $n$ of datacenter $d$: $RC_{s_mn}$
    /* This parameter is determined in the algorithm. Initial value is 0 */

  - $Norc = 0$
  - $Chosenmachine = NULL$
  - For $n = 1$ in $N_d$ do
    - For each request $j$ in $R_{sm}$ do
      - If Compatibility test($j, n$) is TRUE then
        - $RC_{s_mn} = RC_{s_mn} + 1$
      - End if
    - End for
    - If $RC_{s_mn} > Incomp_s$ and $RC_{s_mn} > Norc$ then
      - $Norc = RC_{s_mn}$
      - $Chosenmachine = n$
    - End if
  - End for

  - If $Chosenmachine$ is not NULL then
    - Move the set $R_{sm}$ to $Chosenmachine$
  - End if
3.2.5 Component 4: Context cache replacement algorithm

We consider data-intensive SOA-based enterprise applications; the replacement algorithm for context caches affects the response times achieved. We propose a replacement algorithm for contexts cached in each of the end-servers, detailed in Algorithm 8. For all the contexts cached on an end-server \( m \), we consider 3 parameters of interest, a) the normalized number of requests queued in the end-server \( m \) referencing the context \( c \) (\( ref_m \)), b) the normalized highest penalty value among the classes of requests in the end-server \( m \) referencing the context \( c \) (\( penref_m \)) and c) the normalized latest access time of the context \( c \) in end-server \( m \) (\( acc_m \)). Each of these parameters is associated with a “context cache co-efficient” (\( ref, pen, lacc \) (\( \leq 1 \)), respectively).

We call the sum product of the three parameters with their respective context cache coefficients as the cache replacement index \( (ref \cdot ref_m + pen \cdot penref_m + lacc \cdot acc_m) \). A coefficient of 1 of each of the three parameters assumes equal weight of each towards cache replacement index. A coefficient of 1 for the latest access time \( (lacc) \) and 0 for the others results in the LRU replacement algorithm. The context in the end-server with the lowest cache replacement index is replaced. For simplicity, we assume that the context-data cached at the end-server are always fresh. This is a valid assumption as several research and industry efforts have been directed towards serving dynamic data. For eg., dynamic server-side caching implemented in Websphere Application Server [43] and a scheme for efficiently serving dynamic data at end-servers in [17]. Evaluation of the replacement algorithm and optimum values of the coefficients under different workloads is described in section 3.4.

---

**Algorithm 8** Context cache replacement algorithm

**Input:**
- Number of contexts loaded in machine \( m \): \( maxcon_m \)
- Normalized number of requests in queue referencing context \( c \) in machine \( m \): \( ref_m \)
- Normalized highest penalty value among requests in queue referencing context \( c \) in machine \( m \): \( penref_m \)
- Normalized latest access time of context \( c \) in machine \( m \): \( acc_m \)

\(^*/\) All the above parameters are normalized for fair comparison. Maximum value is 1.*/\(^*

Reference co-efficient: \( ref \)
Penalty co-efficient: \( pen \)
Latest access time co-efficient: \( lacc \)

**Output:** Context to replace

\( temp = 3 \) \(^*/\) Since we are considering 3 parameters, the maximum value of the cache replacement index \( (ref \cdot ref_m + pen \cdot penref_m + lacc \cdot acc_m) \) is 3*/\(^*

for \( c = 1 \) to \( maxcon_m \) do
  if \( (ref \cdot ref_m + pen \cdot penref_m + lacc \cdot acc_m) < temp \) then
    \( temp = ref \cdot ref_m + pen \cdot penref_m + lacc \cdot acc_m \)
    Context to replace: \( c \)
  end if
end for
3.2.6 Assumptions

- The latency in exchanging messages between the geographically distributed clusters is negligible when compared to the request inter-arrival and service times. In the Section 3.4 we demonstrate the robustness of the management policy where we show that the effect due to dropped updates are tolerated to a certain extent.

- The interval between updates to the context-data is significantly higher than the inter-arrival times of requests operating under the same context-data. In the Section 3.4 we evaluate the effect of varying context-data update interval on the management policy.

- The observation interval is of same length for all applications and classes of users. This assumption simplifies the description of DSA_gS, varying observation lengths can be incorporated seamlessly into the proposed policy.

- We assume accurate estimations of processing times of different classes of requests are available.

- In this paper, for simplicity, we assume a non-preemptive scheduling policy.

3.2.7 Centralized variant of DSA_gS

As described in Section 3.2.3, for gi-FIFO scheduling algorithm employed at the end-servers at the data-center, the end-servers should be aware of the current conformance levels of the different classes catered to by different applications. Typically, administrators would require their management policies to be centralized and would require the end-servers to have little/no knowledge of the conformance levels. This is primarily done for better maintainability and loose coupling. For example, if the algorithm would need to undergo a change, then there would be minimum changes required, if the end-servers need not have the required intelligence. In view of this requirement, we propose the adoption of a marker, we term as the “penalty score” which is attached to each request. Now, instead of choosing a request of a particular class of a particular application, the end-servers would choose request(s) with the highest penalty score. The gi-FIFO scheduling algorithm would be, first choose a request(s) with the highest penalty score. Since strict temporal order constraints need to be maintained among requests of the same session, among the sessions queued of the “highest penalty-score requests” , we choose the session whose first request has waited the longest but whose deadline can still be met. If no such session exists, we choose the session whose first request has waited the longest. The first request of the chosen session is scheduled.

We have implemented this policy for only linear percentile SLAs.
Penalty score calculation

The formal description of the penalty score \( PS_k \) is below:

\[
PS_k = \begin{cases} 
  m \cdot (X_{\text{zero}} - X), & \text{if } X < X_{\text{zero}}, \\
  0, & \text{if } X_{\text{zero}} \leq X \leq 100.
\end{cases}
\]

where \( m = P_{\text{max}} / X_{\text{zero}} \), \( P_{\text{max}} \) is the penalty charged when conformance is 0\%, \( X_{\text{zero}} \) is the conformance level at which the penalty charged is 0, \( X \) is the current conformance achieved. Figure 3.9 shows the varying penalty score attached to each arriving request with current conformance levels. In the figure, the penalty charged when conformance is 0\% is 20000 and the conformance after which penalty is 0 is 85\%. Hence after the current conformance level of 85\% is achieved, then the penalty score attached to the request is 0.

The results of evaluation of the centralized policy is described in Section 3.4.6.
3.3 Implementation of DSAgS

In this work we implemented DSAgS in a prototype enterprise system. The enterprise system is built on Apache Software Foundation Virtual Computing Lab (VCL) [50].

VCL is an award-winning open-source implementation of a secure production-level on-demand utility computing and services-oriented technology for wide-area access to solutions based on real and virtualized resources, including computational, storage, network, and software resources. Currently, VCL delivers more than 80,000 image (seat) reservations per semester and more than 7 million HPC CPU-hours per year [50].

A user accesses VCL through a Web interface, which, after appropriate authentication and authorization steps, the user can select an environment like MATLAB on a Windows OS and so on. The image or image environment is loaded, for the duration requested on automatically assigned resources [51].

In this work, we built images for different components of the prototype. The details of the components are explained in the following section. For each run of experiments, the images were requested via VCL’s web interface for a maximum period of 24 hours.

3.3.1 Software architecture

The software architecture of the implementation of the prototype is shown in Figure 3.10. The architecture can be divided into two main components.

- **Controller**: This is the centralized component that allocates requests to the different business servers or end-servers. Management policy actions like session-based, data-aware allocation and compatibility test has been implemented in the controller.

- **Business server**: As the name implies, this component emulates the behavior of the end-server or business server in the prototype. The component communicates with the controller periodically with status updates. This component also implements the scheduling algorithm gi-FIFO for scheduling requests to maximize percentiles achieved.

The above given components are run on distinct images. A typical run would consist of one image reservation of the Controller and multiple (> 3) image reservation of the Business server.

**Design of the controller component**

The software architecture is as shown in Figure 3.10. In the prototype, the controller component plays the role of the middleware described in Figure 3.2. The operation of the controller is as follows:
1. **Input parser module**: When a new item arrives at the controller, it is parsed by the input parser module which updates internal logs. The newly arrived item could be:

   - A request for the applications hosted: The request is queued in an appropriate queue in the decision module.
   - Updates from the business server component on number of requests of different applications that have met their response time goals: The internal data-structures in the metric collection module are updated.
   - Context-data updates from the business server component - Context-data hashtable is updated
   - Request resent to the controller from the business server during session reallocation: The request is queued in an appropriate queue in the decision module.

2. **Decision module**: In the decision module, all queued requests are sent to appropriate end-servers using information provided by the logs in the metric collection module. A
separate thread continually runs the session-based, data-aware allocation algorithm and compatibility tests and once the decision is made, updates the session hashtable and moves the requests to the output queue in the request allocation module.

3. **Request allocation module**: The request allocation module, moves the requests to the appropriate business servers or end-servers.

4. In the prototype the communication between the controller and the end-server component is proprietary (based on Java Sockets [37]) and not based on a standardized technology like Webservices [36]. This choice of proprietary implementation is due to higher control on the communication protocol and the data structures obtained when a proprietary communication fabric is employed than when using off-the-shelf alternatives.

**Design of the business server component**

In the prototype the business server component emulates the end-server which hosts the service-endpoints and executes the requests arriving for different applications. The operations of the business service component is as follows:

1. **Scheduling service**: When a request arrives at the business server, the scheduling service moves the request into an appropriate queue. When the current request completes execution, the scheduling service runs gi-FIFO algorithm to find the next request to schedule among requests queued in the end-server. The scheduling service also performs periodic session reallocation, where it checks for number of requests of a session that do not meet response time goals, if the number exceeds 90%, then it moves the requests to the reporting service’s queue. The scheduling service also updates the reporting service’s queue with metrics.

2. **Business service**: The business service is the least complicated service at the business server component. All it does is accept a request from the scheduling service and execute it in a separate thread. Once the request has completed execution, it reports the completion back to the scheduling service which then schedules the next request. This is done to emulate a non-preemptive system.

3. **Reporting service**: The reporting service periodically reports all the metrics collected, requests from the session reallocation to the controller component.

**3.4 Evaluation**

In this section we present simulation-based results and results from a prototype implementation.
3.4.1 Simulation-based evaluation

We have developed our own discrete event based simulator for evaluation of DSAgS. This section details results to answer the following top-level questions:

1. How does DSAgS perform when compared to commonly deployed solutions?
2. Does session-aware allocation result in “allocation dependency”?
3. Does session reallocation alleviate the “allocation dependency” problem?
4. What are the parameters on which session reallocation depends on?
5. Does the context replacement algorithm contribute to reduced penalties?
6. What are the parameters on which performance improvement due to context replacement algorithm depends on?
7. Can we derive “rules of thumb” from parameters of session reallocation and context-cache replacement policies?
8. How robust is the algorithm to dropped updates and frequency of context-data updates?

We evaluated our management policy to answer the above questions for both linear and stepwise percentile SLAs.

3.4.2 Comparison of DSAgS against commonly deployed policies

A representative scenario involves a cloud computing system with (a) 3 SOA-based applications hosted, serving 5 classes of users with uniformly distributed requests across datacenters, each negotiated with a linear/stepwise percentile SLA as in Figure 3.3, (b) 5 geographically distributed datacenters, (c) 10 end-servers in each datacenter, (d) 100 distinct sessions (e) 500 distinct contexts accessed during each observation interval, (f) each service request accesses data from 1-10 context-data sources with equal probability, (g) observation intervals of 1000 minutes, (h) context load times are uniformly distributed with a mean more than 3 times the average service rate (to model the data-intensive nature of the context-aware applications), (i) the input arrival process is Poisson, (j) the service processes are exponential.

Typical examples of SLAs for the SOA-based applications hosted in the cloud are as shown in Figure 3.11.

We compare DSAgS (without session reallocation) with commonly deployed policies. The allocation algorithms for comparison include:
Figure 3.11: Typical linear percentile SLA examples

- Static allocation - requests for certain context-data are always allocated to the same end-server. A static hash function based on the context-data requested is used to map the requests to the end-server.

- Random allocation - requests are allocated to any end-server in the datacenter with equal probability.

The scheduling algorithms at end-servers considered:

- FIFO (First In First Out)

- Dynamic priority with FIFO (Classes with highest “current” penalty always have the highest priority; priority is thus assigned dynamically. Once a high priority class is chosen, the request of the class that has waited the longest is scheduled (FIFO).)

- WRR (Weighted Round Robin with weights in proportion to the penalty associated with the highest penalty in the SLA of each class $P_{max}$ in Figure 3.3.)

Typical results in Figure 3.12 show that DSAgS substantially outperforms the commonly deployed solutions (confidence intervals (ci) 95%). As expected, random allocation when employed with any of the scheduling policies, results in the lowest conformance and thus highest
penalty (Random allocation with FIFO or WRR resulted in highest penalties possible, Figure 3.12 and Figure 3.13). As can be seen, a dynamic scheduling algorithm (priority) when used with a static allocation algorithm results in lower penalties and so our dynamic allocation algorithm with a dynamic scheduling algorithm designed to maximize percentiles improves the penalties further.

DSAgS is session-aware by allocating requests with the same session-id arriving within the “session validity” period to the same end-server. So, for consecutive requests of the same session, compatibility tests or data-awareness are not performed. This results in “allocation dependency” as shown in Figure 3.14 and Figure 3.15. In Figure 3.14, we show that the increase in penalties due to allocation dependency can be noticed when other management policies are used. As shown, the highest increase in penalties due to session-awareness is seen in DSAgS and in static allocation with dynamic priority. When static allocation with WRR scheduling is used,
the increase in penalties is very low when compared to that in DSAgS and static allocation with dynamic priority. There is no significant effect of allocation dependency when used in static allocation with FIFO.

As a solution to allocation dependency, we propose periodic session reallocation, the effect of which we notice in the next section.

### 3.4.3 Evaluation of session reallocation

As described in section 3.2.4, we propose periodic reallocation of all requests in a session to a peer end-server in the datacenter, capable of meeting the deadline of the maximum number of requests in the session. Figure 3.16 and 3.17 compares the penalties obtained for DSAgS with and without session reallocation (ci 95%). In this experiment, a) the number of sessions is 10, b) context load times are uniformly distributed with a mean more than 10 times the average service time (The effect of performing periodic session reallocation is profound, when
Figure 3.14: Demonstration of allocation dependency in all algorithms. (Linear percentile SLA)

The context load times are much higher than the service rate. (Refer to section 3.2.4 for details); the remaining attributes are as described in section 3.4.2. As shown in Figure 3.16, the penalties obtained in DSAgS with session reallocation is lower when compared to DSAgS without session reallocation thus alleviating the effects of “allocation dependency”. However as shown in the figure, the improvement with session reallocation does not always equate to employing DSAgS where “sessions” are not used (essentially only one request belongs to each session).

The overhead incurred when DSAgS employs session reallocation is offset by the ensuing large reduction in penalties, when a small number of sessions generating large number of requests request service from highly data-intensive context-aware applications. However, when a large number of sessions with small number of requests access context-aware applications with lower context-data load times, the overhead exceeds the benefits. Three parameters of interest in the evaluation of session reallocation are 1) the length of session reallocation period interval which influences the overhead caused, 2) the number of simultaneous sessions, and, 3) the number of
requests in each session.

**Effect of length of the session reallocation period interval**

The length of the session reallocation period determines the frequency of session reallocation occurring in each end-server. As expected, shown in Figure 3.18 and 3.19 the reduction in penalties due to reduction in the session reallocation period length is seen up to a certain value after which there is no significant reduction till the session reallocation period reduces further during which the penalty charged increases. This increase in penalty is partly due to higher overheads in transferring requests from one machine to another and partly due to errors in estimations of the completion times of requests. The actual completion times of requests might be different and estimations made at due to lower session reallocation period lengths might result in frequent errors in estimations.
Figure 3.16: Comparison of DSAgS with and without session reallocation.(Linear percentile SLA)

**Effect of number of simultaneous sessions and number of requests in each session**

The number of simultaneous sessions addressed by an end-server along with the number of requests in each session influences the reduction in penalties caused by session reallocation. For this evaluation we define and vary the request-session ratio i.e., the ratio of number of requests queued at the end server to the number of sessions they belonged to. So a value of 1 would essentially mean, each request belongs to its own session, a value higher than one means more than one of the requests queued at the end-server belonged to the same session. As we can see from Figure 3.20 and 3.21, as the number of simultaneous sessions increase, i.e the request-session ratio decreases and approaches a value of 1, the reduction in penalties charged to the cloud due to session reallocation is lower and the reduction is highest when there are a sufficiently large number of requests in the end-server belonging to the same session. In this experiment, the distribution of sessions addressed by each end-server is non-uniform. The
request-ratio value is averaged over all runs.

### 3.4.4 Evaluation of context replacement algorithm

We propose a context-cache replacement algorithm described in section 3.2.5. We varied the context cache coefficients \((\text{ref, pen, lacc})\) and observed the effect on the penalties charged. In this experiment, we maintained context load times to be uniformly distributed with a mean more than 10 times the average service time in order to determine the effect of caching on the penalties. The remaining system parameters are same as in section 3.4.2. Figure 3.22 and 3.23 show that an equal value for all context cache coefficients results in marginally lower penalties (ci 95%). For this result, we varied both the distribution of context references, distribution of penalty of context references to be non-uniform.
Effect of distribution of references to contexts in the context replacement algorithm

The effect of context replacement algorithm on the percentile response times is observed when the mean of the context load times are more than 10 times the mean service time. Another parameter of importance is choosing the right coefficients for the calculation of cache replacement index. In Figure 3.24, 3.26 and 3.28 we vary the cache coefficients for 3 different distributions of references of contexts. The first variation is uniform distribution of references for all contexts (Shown in Figure 3.24 and Figure 3.25). The percentiles to be achieved and penalties charged for non-compliance was also same for all classes. In this case, the choice of coefficients were not important and all policies performed about the same. In this case, the effect of applying the context cache replacement algorithm in comparison to LRU was minimal. The second variation is the distribution of references to contexts was non-uniform, essentially some context-data...
Figure 3.19: Effect of length of the session reallocation interval. (Stepwise percentile SLA)

were more important than rest (Shown in Figures 3.26 and 3.27). In this case if the co-efficient related to $ref$ was given a higher weightage over the other two, a slight improvement was noted. The third variation is the distribution of penalties of requests referencing contexts was non-uniform (Shown in Figures 3.28 and 3.29). Some contexts were referenced by application requests that charged more penalty than the rest. In this case if the co-efficient related to $pen$ was given a higher weightage over the other two, then a marginal improvement was noted. In both the above cases, LRU resulted in almost similar penalty values.

From this result, we show that for different applications, workload characterization has to be performed to determine the best mix of co-efficients to be used in context cache replacement algorithm.

Subsections of the Sections 3.4.3 and 3.4.4 describe the “rules of thumb” derived from parameters of session reallocation and context-cache replacement policies. “Rules of thumb” describe
the effect of various system parameters and algorithm artifacts can have on the penalty charged to the cloud. These rules can be then mapped to the hosted SOA-based application constructs to provide improved response times, increased percentiles and thus lower penalties charged to the cloud.

### 3.4.5 Robustness of DSAgS

In this section we evaluate the robustness of the management policy. The two parameters under consideration are a) Frequency of erroneous/dropped updates. b) Frequency of context-data updates.
Effect of erroneous/dropped updates

We evaluate the robustness of the management policy as the distributed nature of the system architecture is prone to erroneous/dropped updates. There are two points of distribution of messages in the system architecture considered. a) Conformance levels exchanged between data-centers and b) Conformance levels updated at each end-server in the data-center by the central dispatcher. The frequency of the former is much higher than the later as the messages are typically updates across geographically distributed locations whereas in the later, the messages are exchanged between co-located end-points. In this work, we thus consider evaluation of the robustness of the algorithm when the messages between data-centers are dropped. As shown in Figure 3.30, as we increase the frequency of the dropped messages between the data-centers, the penalties charged to the cloud increase as each datacenter wrongly calculates the “best” allocation and scheduling choices. But the increase in penalty begins after the frequency of
dropped messages increases beyond a certain threshold and the increase in penalties charged to the cloud appears to increase almost exponentially at each subsequent raise in the error-rate. For this experiment, we maintain the distribution of requests, penalties of different applications such that a wrong choice results has a drastic consequence, to observe the robustness of the algorithm.

**Effect of frequency of context-data updates**

In DSA<sub>g</sub>S, we assume that the interval between updates to the context-data is much higher than the interval between service requests operating under the same context-data. Hence we propose data-aware allocation as part of our management policy. Under the assumption, data-awareness reduces response times by moving requests closer to the required context-data (allocated re-
requests to end-servers cached with the maximum number of context-data, see section 3.2.2). In the following experiment we challenge this assumption, by incorporating varying context-data update intervals. As expected, as the context-update intervals reduce (expressed in times the request intervals), the policy tends to be less data-aware and the response times achieved increases resulting in higher penalties charged to the cloud. From this result as in Figures 3.32 and 3.33, we show that to employ the request allocation algorithm mentioned in DSA_{gS}, it is important for the interval between updates to context-data be much higher than intervals between requests for the same data.
3.4.6 Results of implementation in a prototype

In this section we list the results of the implementation of DSAgS in a prototype enterprise system. The prototype was implemented on NCSU VCL [50]. Two images on VCL were created to emulate the controller and the end-server. Reservations of 5 machines (VM or baremetal is decided by VCL) loaded with the end-server images and one machine loaded with the controller is used for the following experiments. We have no control on the hardware of the machines as they are allocated dynamically to us. This works to our advantage, as the management policy is evaluated on a truly heterogenous system.

A representative scenario involves a cloud computing system with (a) 3 SOA-based context-aware applications hosted, serving 5 classes of users with uniformly distributed requests across datacenters, each negotiated with a linear percentile SLA as in Figure 3.3, (b) 500 distinct sessions on average (c) 1000 distinct contexts accessed during each observation interval on average,
Figure 3.25: Penalties charged for different context cache co-efficients with uniform distribution of reference penalties and number of references. (Stepwise percentile SLA)

(d) each service request accesses data from 1-10 context-data sources with equal probability, (e) observation intervals of 100 minutes, (f) context load times are uniformly distributed with a mean more than 5 times the average service rate (to model the data-intensive nature of the context-aware applications), (g) the input arrival process is Poisson, (h) the service processes are exponential.

Comparison of gi-FIFO with variants

The first experiment is the comparison of the scheduling policies in the end-servers of the prototype enterprise system. In this result, we compare FIFO (First In First Out), WRR (Weighted Round Robin), dynamic priority and gi-FIFO when implemented at the end-servers.
Figure 3.26: Penalties charged for different context cache co-effecients with non-uniform distribution of reference penalties. (Linear percentile SLA)

The application hosted is assumed to be non data-intensive and the overhead associated with loading the service-end point at each end-server is assumed to be negligible in comparison to the processing time of each request. In the Figure 3.34, we see that in the prototype, gi-FIFO results in the lowest response times and thus the lowest penalties charged.

Comparison of DSA_gS with variants

The second experiment is the comparison of DSA_gS other the management policies in the prototype. The allocation algorithms for comparison include:

- Static allocation - requests for certain context-data are always allocated to the same end-server. A static hash function based on the context-data requested is used to map the requests to the end-server.
Figure 3.27: Penalties charged for different context cache coefficients with non-uniform distribution of reference penalties. (Stepwise percentile SLA)

- Random allocation - requests are allocated to any end-server in the datacenter with equal probability.

The scheduling algorithms at end-servers considered:

- FIFO (First In First Out)
- Dynamic priority with FIFO (Classes with highest “current” penalty always have the highest priority; priority is thus assigned dynamically. Once a high priority class is chosen, the request of the class that has waited the longest is scheduled (FIFO).)
- WRR (Weighted Round Robin with weights in proportion to the penalty associated with the first step in the SLA of each class.)
Figure 3.28: Penalties charged for different context cache co-efficients with non-uniform distribution of number of references (Linear percentile SLA)

Typical results in Figure 3.35, show that DSA_g result in the lowest response times and thus the lowest penalties charged. As the input request rate to the applications hosted increase, the penalties charged also increase. As in the case of simulation results, the management policy with static allocation and dynamic priority for scheduling is the next “best” policy resulting in penalties only higher than DSA_g. Static allocation with WRR and FIFO result in higher response times, resulting in higher penalties charged. Random allocation results in the highest penalties charged irrespective of the scheduling algorithm utilized.

**Evaluation of session reallocation**

Session reallocation was evaluated in the prototype. The session reallocation algorithm was run as a separate thread at the end-server. The end-server would “periodically” evaluate the requests in the queue and notify the controller about non-conforming requests, the controller
would then “add” them back to its queue and find the most compatible end-server among the 5 end-servers. As described above, the session reallocation algorithm was sensitive to the a) length of the session reallocation interval b) distribution of number of sessions among the end-servers. In the prototype, to achieve the balance between overhead of the algorithm and the improvement seen due to session reallocation was quite a challenge. In Figure 3.36, we show a case where improvement due to session reallocation is seen relative to when no session reallocation is performed.

- Effect of session reallocation period length: As shown in Figure 3.37, as the session reallocation period length increases, the reduction in penalty decreases (from 5 mins to 10 mins). However, if the session reallocation period length is too low, the overhead trumps the benefits and the penalty attained is higher.
Figure 3.30: Robustness of DSAgS to dropped/erroneous peer updates. (Linear percentile SLA)

- Average request-session ratio: As described in section 3.4.3, session reallocation is dependent on the distribution of number of requests in each session and the distribution of sessions among the end-servers. As shown in Figure 3.38, as the request to session ratio decreases, the penalty charged increases when session reallocation is employed.

Evaluation of the centralized variant

As mentioned in Section 3.2.7, we implemented the centralized version of our management policy in the prototype enterprise system. As shown in Figure 3.39, the centralized penalty score marker method is employed, the penalties charged are higher than when employing the decentralized policy. The main reason for this increase in penalty charged, is because the markers are assigned to requests at the time they arrive at the middleware/controller. When they are eventually executed in the scheduling service, the current penalties and conformance levels of the class of the request could have changed resulting in the marker penalty score values
Figure 3.31: Robustness of DSAgS to dropped/erroneous peer updates (Stepwise percentile SLA)

to be stale.
Figure 3.32: Robustness of DSAgS to varying context-update intervals. (Linear percentile SLA)
Figure 3.33: Robustness of DSAgS to varying context-update intervals. (Stepwise percentile SLA)
Figure 3.34: Comparison of scheduling policies in the prototype enterprise system
Figure 3.35: Comparison of allocation and scheduling policies in the prototype.
Figure 3.36: Effect of session reallocation in the prototype
Figure 3.37: Effect of varying session reallocation interval length.
Figure 3.38: Effect of varying request-session ratio.
Figure 3.39: Comparison of the penalty score, centralized method with variants.
3.5 Related Work

To the best of our knowledge, we are the first to propose a solution for management of SOA-based enterprise applications subject to percentile constraints. Our work brings together hitherto separate areas of interest, management of SOA-based enterprise applications and enforcement of percentile constraints SLA for web-based applications. In this section, we consider objectives we aim to achieve in each area separately and compare them with previous research.

3.5.1 QoS in SOA-based enterprise data-intensive applications

In recent years, resource and/or request management in SOA-based enterprise applications has garnered considerable interest. Huebscher and McCann [25] proposed a fault-tolerant adaptive middleware framework for context-aware SOA-based applications which selects context-providers with an aim to maximize utility based on the accuracy of context-data from the providers. Lakshmanan et al. [27] address the problem of resource management in semantic event processing applications and propose a horizontal partition that is automatically created by analyzing the semantic dependencies among agents (service-endpoints) using a stratification principle. They implement a profiling-based technique for assigning agents to nodes in each stratum with the goal of maximizing throughput and distributing the load for increased scalability. Lorincz et al. [29] propose a new operating system for sensor nodes that enables resource aware programming while permitting high-level reusable resource management policies for applications. We identify a need for dynamic data-dependency [46] and temporal order constraints based allocation in a set of context-aware applications and propose an adaptive, data-aware, session-grained policy for allocation of requests and aim to conform a performance metric (response times) to specified values. Our policy can be implemented in a framework described by [29].

3.5.2 Percentile Service Level Agreements

Research on management policies for meeting SLAs citing performance percentile criterion has not been as prolific as on policies for meeting average (mean) performance criterion. Gmach et al. [22] consider step-wise percentile SLAs and propose scheduling algorithms for a single database server unlike a distributed solution as proposed in this work. Xiong et al. [53] provide an analytical solution of resource optimization subject to percentile response time and price by modeling the system as an overtake free open tandem queuing network with feedback and provide closed form expressions of the probability distribution function of the response time. Cardellini et al. [15] present a brokering service for management of composite services under percentile-based SLAs; they propose a QoS model for composite services in which they provide
an expression for the percentile of response time, assuming to know the $\alpha$-quantile of the normalized response time in advance. To the best of our knowledge, these are the only 3 efforts for enforcing percentile SLAs in web-based applications. However, in contrast to the method described in the latter two policies, we do not make any assumption about the distribution of input arrivals (or quantiles of the normalized response time) and service times, our request management policy is measurement-based and adaptive in nature.

Adoption of learning techniques for utility maximizing adaptive resource management has been an active area of research in the recent years. Tesauro et al. [49] propose a reinforcement learning based management system for dynamic allocation of servers to web applications aiming to maximize the profit charged to the host datacenter. Reinforcement learning techniques seems most promising and investigation of applicability to this work is part of our future research.

3.6 Summary

We studied the problem of management of SOA based, data-intensive SOA-based enterprise applications hosted in a distributed cloud where the system operates under a global, percentile response time SLA. The SLA calls for economic penalties if percentile targets are not met. We proposed Data-aware Session-grained Allocation with $gi$-FIFO Scheduling (DSA$gS$), a novel decentralized request management policy. Our simulation evaluation shows that our dynamic policy far outperforms commonly deployed management policies in achieving lower penalties. We also proposed and evaluated a “context level” cache replacement algorithm that contributes to reduced penalties charged to the cloud provider hosting the applications.
Chapter 4

Future Work

In our work we proposed dynamic, adaptive, feedback-based management policy for data-intensive SOA-based applications hosted on a cloud computing system. The future work in this context can be primarily divided into two sections.

- Extension of the reactive management policy
- Investigations into proactive management policies.

4.1 Future work towards the reactive management policy

The following are identified as the main areas where this work could be extended:

- As shown in Section 3.4.6, the session reallocation and context-cache replacement policies depend on several system parameters. We have shown that when employing session reallocation and context-cache replacement policies, workload characterization has to be performed to evaluate the appropriate system parameters for improvements to be observed in response times. Hence, when employing these schemes in real-world applications, the general “rules of thumb” developed in this work needs to be applied to the real-world applications parameters. This workload characterization could result in concrete “rules of thumb” for certain classes of applications which would be extremely useful in improving response times and maximizing percentiles.

- An area that has been left unexplored in this work is investigations into relation between the context-data stores and the end-servers context cache. The context-caches at the end-servers could be tuned to improve performance if context-data storage type and update pattern is known.
• A final area of interest in this work would be an ability to improve operational costs by deciding whether certain end-servers can be turned “off” under low loads while continuing to meet percentile SLA goals.

### 4.2 Proactive management of SOA-based data-intensive applications subject to percentile Service Level Agreements

A parallel solution approach would be proactive in nature, where the policy would anticipate the changes in the system parameters and adapt to them. A main contender to this approach would be to apply reinforcement learning techniques with an aim to maximize SLA goals in the management of SOA-based data-intensive applications [49].
REFERENCES


[37] Oracle. Lesson: All about sockets.


82


Appendix A

Evaluation of heuristics for request scheduling subject to response time percentile Service Level Agreements

In this chapter we explain the heuristic-based scheduling algorithm we developed, subject to a percentile response time SLA for a geographically distributed cloud.

A.1 Step-wise Service Level Agreement

The term service level agreement has been explained in detail in chapter 1. In this work,
we consider the single-step percentile SLA, where the fraction of service requests to be executed within a certain response time is specified, along with the penalty charged on the cloud on the non-conformance of the percentile requirement.

The formal description of the SLA we consider is as follows:

Let $X\%$ be the fraction of requests of a particular application which need to have a response time less than $r$ seconds. If the percentile of requests that have response time less than $r$ seconds is less than $X\%$, then each of the non-conforming requests contributing to the drop in the percentile is charged a penalty of $P\$, as shown in Fig. A.1.

### A.2 System model

In this work, we consider multiple geographically distributed data-centers, each with a large number of servers, forming a cloud. The centers host collectively multiple classes of applications.

The general architecture of the system is shown in Fig. A.2. The following are the key elements of the system:

- **Clients.** These are nodes that generate the service requests forwarded to the servers at the different data centers of the cloud. The clients are represented as the internet cloud in Fig. A.2.

- **Data centers and hosted applications.** A data center is a cluster of a large number of networked computing resources. In the topology considered, multiple geographically distributed data centers form the cloud computing system with each data center hosting

![Figure A.2: The general architecture of the system.](Image)
the same set of applications as shown in Fig. A.2. Each data center receives web service requests for applications from clients. An application’s web service end points are replicated in all the data centers, i.e., any data center is capable of serving a request for any application. Each application is identified by a class. So if the cloud hosts K applications, there are K classes of requests to be served by the data centers.

- **Model of resources in a data center.** Each data center has a number of servers (resources) for processing the web service requests. A server processes a single request to completion each time. A request being processed cannot be preemption. Requests arriving when the server is busy are queued. Each server in a data center can process a request of any class (application). So each server in the data center is modeled as serving multiple single-class queues, each queue holding requests of a particular class as shown in Fig A.3.

- **Percentile Service Level Agreements.** In this problem we consider percentile SLAs where the percentile of service requests to be executed within a certain response time is given, along with the penalty charged on the cloud for the non-conformance of any service request beyond the stated percentile as shown in Fig. A.1.

With the applications deployed across multiple data centers, there is a need for resource allocation and request scheduling techniques for the satisfaction of the SLA of each application, globally, across the geographically distributed data centers. Thus the SLA is global in definition, i.e., all the data centers in the cloud have to collectively respect the SLA. So, the response time and percentile constraints of the application should be met at the global cloud system level.

The service requests for different applications from the clients can be routed to any end server at any data center in the cloud. The routing of service requests to the different servers is based on cloud management policies depending on load of individual servers, proximity to databases etc. (We do not consider the problem of routing the service requests to the servers.) The new requests are queued for scheduling at the servers.

### A.3 Problem statement

We want to schedule the incoming service requests of different classes locally at the end-servers in the geographically distributed data centers so as to minimize the total global penalty incurred. More specifically, we want to determine a scheduling algorithm that provides the minimum in equation A.1 below:

$$
\min_{1 \leq j \leq K} \sum \text{pen}_j
$$

(A.1)
where $\text{pen}_j$ is the penalty charged for non-conformance of the requests of class $j$ as described by Fig. A.1 for the entire cloud.

### A.4 Heuristic-based request schedules

The most common scheduling principle in a non-preemptive system are the First-In-First-Out (FIFO) and Weighted Round Robin (WRR) policies.

In FIFO, some requests can be delayed beyond the constraints specified in the SLA and incur penalties. Requests of different classes when delayed, incur different penalties, based on the current number of requests that conform to the response-time constraint mentioned in the SLA.

In WRR, assume that the requests of different classes are assigned different weights based on their penalty or their required conformance. This static schedule will not aim to minimize the penalty charged to the cloud as the schedules have to depend on the current conformance of the requests of each class.

The local scheduling policy at each end server should be such that the cloud globally minimizes the penalty. There is a need for a dynamic scheduling policy which schedules new requests at end servers, to adhere to the SLAs specified for each class of service request and thus minimizing the penalty incurred globally.

In this work, we propose a distributed, measurement based policy to schedule requests queued at individual servers located in each data center in the cloud. There are two basic ideas behind the proposed scheduling policy. The first is that, for the scheduling at each server to be based on the current global SLA conformance, we propose periodic updates of conformance levels of each application between the geographically distributed data centers so that each data center is aware of the current global conformance at periodic intervals. The second is the calculation of penalty incurred by each arriving request, which is charged, if it does not meet
the response time constraint; this calculation (see Algorithm 11) is done adaptively, based on the current global non-conformance of the class of the request, which is the fraction of requests of the class which have not met the response time specified in the SLA ($1 - cc_k$). The aim is to ensure that incoming requests of classes with higher current conformance with respect to their SLA are assigned a lower penalty and vice versa.

A.4.1 Algorithm description

The observation interval ($T$ secs) during which the SLA has to be met is divided into several subintervals, the number of which is configurable. The observation interval is applicable to the entire distributed cloud i.e the observation interval is the duration for which the equation A.1 has to be minimal. This can repeat indefinitely or a set number of times configurable by the cloud administrator. The subintervals start and end at the same instant in all the data centers (synchronization). Each subinterval is partitioned into a “scheduling phase” and an “adaptation phase”, as shown in Fig. A.4 and formalized in Algorithm 9. The two phases are explained in detail in Algorithms 10 and 11.

![Figure A.4: Periodic scheduling and adaptation at each data center.](image)

**Adaptation phase**

In the adaptation phase, described in Algorithm 10, each data center exchanges its current conformance levels of all classes with other data centers in the cloud, and each data center calculates the updated conformance levels for all the classes. This updated current conformance level ($cc_k$ in Algorithm 10 and 11) is used in individual request-penalty calculations during the scheduling phase.
Algorithm 9 Heuristic-based scheduling algorithm for global penalty minimization.

Input:
- Length of observation interval: T
- Number of subintervals: Z

Output: Minimization of equation A.1.

for $z = 1$ to $Z$
    Scheduling phase as in Algorithm 11
    Adaptation phase as in Algorithm 10
end for

Algorithm 10 Adaptation phase at datacenter $d$, calculating updated global conformance levels.

Input:
- Number of geographically distributed data centers: $N$
- Number of classes of service requests: $K$
- Number of requests serviced of class $k$ in datacenter $l$ in current subinterval: $X_{slk}$
  $\forall k \in \{1, \ldots, K\} \forall l \in \{1, \ldots, N\}$
- Number of requests class $k$ in datacenter $l$ which met the required response time in current subinterval: $X_{r_{lk}}$
- Total number of requests serviced of class $k$ from the start of the observation interval as measured by $d$: $X_k$
- Current conformance of class $k$ in cloud as calculated by data center $d$: $cc_k$

Output: Updated current global conformance calculated by $d$: $cc_k \forall k \in \{1, \ldots, K\}$

for $k = 1$ to $K$
    temp = 0
    temp2 = $X_k$
    for $i = 1$ to $(d - 1)$ and $i = (d + 1)$ to $N$
        temp = temp + $X_{r_{ik}}$
        $X_k = X_k + X_{slk}$ /*Get updates from all other data centers*/
    end for
    $cc_k = (cc_k * temp2 + temp)/X_k$ /*Updated conformance level*/
end for

Scheduling phase

In the scheduling phase, run at each end-server, shown in Algorithm 11, each arriving request at end-server is assigned a penalty and scheduled. We calculate the effect of delaying the recently arrived request on the current non-conformance, so the numerator and denominator in equation (A) of Algorithm 11 are both incremented by one request. If delaying this request causes the non-conformance to increase beyond that given in the SLA, then the request is assigned a
Algorithm 11 During scheduling phase: A request $j$ arrives at end-server $s$ for service.

Input:
- Number of servers in each data center: $m_l \forall l \in \{1, \ldots, N\}
- Request $j$ of class $k$ arrived at queue of server $s$ at data center $l$ $\forall k \in \{1, \ldots, K\}$, $\forall s \in \{1, \ldots, m_l\}, \forall l \in \{1, \ldots, N\}$
- Penalty per request of class $k$ on non-conformance as per SLA: $p_k \forall k \in \{1, \ldots, K\}$
- Required global conformance of class $k$ as per SLA: $c_k \forall k \in \{1, \ldots, K\}$
- Required response time conformance of class $k$ as per SLA: $r_k \forall k \in \{1, \ldots, K\}$
- Number of requests queued at server $s$ at data center $l$ at time $t$: $q_{ls}$

Output: Penalty applied for newly arrived request $j$ in the schedule if it does not meet $r_k$: $pa_j$

if $q_{ls} = 0$ and server $s$ is free then
    Dispatch request $j$ for processing at end-server $s$
    Depending if request $j$ met the response time, update the conformance for class $k$.
else
    $nonconf = ((1 - cc_k) \times X_k + 1)/(X_k + 1)$ \hspace{1cm} (A)
    if $nonconf > (1 - c_k)$ then
        $pa_j = p_k$ /*non-conformance high, $p_k$ penalty assigned to $j$*/
    else
        $pa_j = 0$ /*non-conformance low, 0 penalty assigned to $j$*/
    end if
    Insert request $j$ charged with penalty $pa_j$ in the queue of end server $s$
    Apply heuristics for scheduling the request as in Algorithm 12.
end if

penalty $p_k$, else it is assigned a penalty of 0. This penalty assignment ($pa_j$) is performed at each end server, upon arrival of every request and so each queued request has a penalty assigned, which is charged if it does not meet the response time.

So our multi-class, multi-server, percentile penalty-based scheduling problem is now converted to the well-investigated, multi-class, single machine scheduling problem, in which the penalty charged is dependent on the request completion time [8] shown in Equation (B) of Algorithm 12. Determining the schedule that minimizes the total penalty for the single server in Equation (C) in Algorithm 12 is known to be NP-hard [11]; typically, such a problem is solved with heuristics for neighborhood searches [11]. The neighborhood search techniques we have chosen are as follows.

- **Simulated Annealing.** In this iterative method, summarized in Algorithm 12, we begin with a seed schedule in the first iteration, typically ordered on the arrival instants (in
Algorithm 12 Simulated Annealing/gi-FIFO based optimum schedule for requests queued at server.

**Input:**
- Number of requests queued at server \( s \) at datacenter \( l \) at time \( t \): \( n \forall s \in \{1, \ldots, m_l\}, \forall l \in \{1, \ldots, N\} \)
- Process time of request \( j \): \( P_j \forall j \in \{1, \ldots, n\} \)
- Starting time of request \( j \): \( x_j \forall j \in \{1, \ldots, n\} \)
- Penalty charged if request \( j \) is scheduled at time \( x_j \): \( pn_j(x_j) \)
- If scheduling the request \( j \) of class \( k \) at time \( x_j \) causes the response time of \( j \) to be less than or equal to \( r_k \), then \( pn_j(x_j) = 0 \), otherwise \( pn_j(x_j) = pa_j / \) can be either 0 or \( p_k \) depending on the current conformance of class \( k \) as assigned in Algorithm 11*/
- \( X: \{ x \mid x_{j_1} = t \text{ and } x_{j_{z+1}} = x_{j_z} + P_{j_z}, z = 1, \ldots, n \text{ for some permutation } j_1, \ldots, j_n \text{ of } 1, \ldots, n \} \) (B)

We have to obtain optimum schedule of the \( n \) requests queued where
\[
\min_{x \in X} \sum_{1 \leq j \leq n} pn_j(x_j) \quad \text{(C)}
\]

**Output:** Most optimum schedule of requests queued at server \( s \)

SA: Use current schedule as starting seed schedule

```plaintext
iterations = 0
repeat
    penalty = Compute penalty with current schedule
    new schedule = pairwise interchange or last insertion
    delta = Penalty of new schedule - penalty
    if delta < 0 then
        final penalty = penalty + delta
        Final schedule = new schedule
    end if
    current schedule = new schedule
    iterations +=
until iterations < MAXITERATION
OR
Apply gi-FIFO policy for the requests queued
```

our implementation); in each subsequent iteration, we re-order the requests queued and calculate the penalty for each schedule obtained. The next schedule to move to is chosen in random and this is continued for a set number of iterations. At the end, we choose the schedule with the lowest penalty [16]. We investigate two methods of neighborhood search in simulated annealing, namely last insertion and pairwise interchange [11]; we have selected them for the computational overhead they introduce. They differ in the way of obtaining the next (neighbor) schedule. In pairwise interchange, we interchange the order of two randomly selected service requests in each neighbor schedule and so we
have a maximum of \( n! \) number of schedules with \( n \) being the number of requests queued. However, depending on the number of iterations, all the schedules may not occur. In last insertion, a new neighbor is generated by inserting the recently arrived request in different positions of the schedule leading to a total of \( (n - 1) \) schedules.

- **Tabu Search.** A heuristic similar to simulated annealing is Tabu Search [11]. Tabu search prevents the occurrence of local optima in any neighborhood search. In tabu search, a search for the optimum solution is carried out in a similar manner as in simulated annealing, in addition, a tabu list is maintained which holds a configurable number of past traversed schedules. The currently found schedule is compared to the list and if it is found, it is discarded and a new schedule is obtained in its place. The two variants of neighborhood search proposed for simulated annealing can be utilized for Tabu search as well.

The neighborhood search techniques as can be inferred above have high execution times. We propose an alternative scheduling technique, less precise than the neighborhood search but with comparatively low execution times. Our second method of scheduling the arriving request at each end sever is the gi-FIFO policy [5], which is described as follows:

*First, choose the request class with the highest penalty; then, amongst all the queued requests of the chosen class, choose one with maximum waiting time but which results in a response time less than or equal to \( r \). If no such request exists, choose the request with higher waiting time resulting in a response time greater than \( r \).*

The gi-FIFO policy was shown to maximize delay percentiles in single-server systems [5].

In Section A.4.2, we compare the two variants of simulated annealing, Tabu search with pairwise interchange and gi-FIFO and FIFO policies with respect to minimizing penalty in our system.

**Assumptions**

In formulating the algorithm we have made the following assumptions:

- The latency in exchanging messages between the geographically distributed clusters is negligible when compared to the request inter-arrival and service times.

- The time required for updates of the status of an arriving request, to propagate to all servers in the data center is negligible compared to the service request inter-arrival and service times.

- The processing time of a service request at a server is known on its arrival.
The data-centers are synchronized and so the subintervals start and end at the same instant at all data-centers.

A.4.2 Evaluation

We evaluated our proposed solution via simulation. A representative scenario involves a cloud computing system with (a) $K = 10$ classes of services each with SLAs as described in Fig. A.1, (b) $N = 5$ geographically distributed data centers, (c) 10 servers in each data center, (d) the input arrival process is Poisson, (e) the service processes are exponential, uniform across all classes. An example representation of the SLA for each class is as shown: $[90,0.09,2]$, $[90,0.1,3]$, $[80,0.08,2]$, $[60,0.09,2]$, $[50,0.05,2]$, $[80,0.12,2]$, $[70,0.09,3]$, $[85,0.03,2]$, $[75,0.2,3]$, $[95,0.04,2]$ where each tuple is the SLA for each class, the first parameter of each tuple is the required conformance, the second parameter is the penalty ($) on each request in the non-conforming region and the third parameter is the cut-off response time.

We center our evaluation on the following questions:

**How does the solution algorithm suggested for solving the minimization problem in equation A.1 perform?**

To answer this question, we compared the total penalty incurred on the cloud with the heuristic-based request schedule (with simulated annealing technique) against that obtained with FIFO and WRR techniques. Fig. A.6 shows (with 95% confidence intervals) the penalty incurred in FIFO is much higher (many times, almost $10^3$ times) than that incurred by our algorithm for varying input rates.

Moreover as Fig. A.5 shows (with 95% confidence intervals) Heuristic-based scheduling typically outperforms FIFO even on a per class basis, with conformance levels for each class matching that required by the SLA whereas conformance levels in FIFO is much lower than that in the SLA.

Fig. A.7 shows with 95% confidence intervals the penalty incurred in WRR is much higher (many times, almost $10^3$ times) than that incurred by the heuristic-based request scheduling algorithm for varying input rates.

**Does the heuristic-based scheduling algorithm adapt the penalty assigned to an incoming request according to the current conformance of its class?**

The estimation of non-conformance in equation (A) in Algorithm 11 (and the corresponding penalty prediction for each individual request) is a key element of our scheduling policy.

In the same configuration as in the first experiment, consider, for example, two classes with differing conformance requirements. Fig. A.8 is a typical depiction of the penalty charged to
Does the heuristic-based scheduling algorithm favor requests with higher penalty?

Intuitively, since our algorithm attempts to minimize the total penalty as expressed in Equation A.1, it must ensure that classes with higher \( p_k \) penalty values are scheduled “sooner” than classes with lower such values.

To answer this we simulated the algorithm with the same cloud computing configuration as mentioned in the first experiment, but with just two classes of service requests, class one with penalty of 0.9$ for each non-conforming request and class two with penalty of 0.1$ for each non-conforming request, with a cut-off conformance of 90% and the same response time requirement for both. The input request rates for the two classes are the same. We ran the simulation multiple times varying the input request rate each time and the results in Fig. A.9 show that the requests for class one are favored over requests of class two by our scheduling algorithm, thus scheduling the requests with higher penalty ahead of requests with lower penalty but FIFO scheduling favors requests of both classes equally.
Is the algorithm distributed in nature?

The scheduling algorithm at each geographically distributed data center should aim to minimize the global penalty. This is achieved by the periodic adaptation phase as shown in Fig. A.4.

In the same configuration as in the first experiment, consider, for example, two clusters, one with large number of resources and another with small number of resources. Results in Fig. A.10 show that the locally calculated current conformance $cc_k$ values, at data center with low resources increase after considering the global value.

Which variant of our algorithm obtains schedules with lower penalty?

We now compare which variant of our algorithm among last insertion, pairwise interchange simulated annealing, tabu search and gi-FIFO provides the best schedules and investigate conditions for obtaining

Our simulations did not reveal a clear winner; in general, pairwise interchange performed better than the other two (and so we simulate only pairwise interchange variant of Tabu search). A typical result from our run is shown in Fig. A.11. In this figure, we compute the total penalty as a function of request rates for both pairwise interchange and gi-FIFO with very stringent SLA criterion, and when the system is stressed where even at lower request rates, the penalty incurred is high. As we can see, at lower request rates (lesser stress), both gi-FIFO and pairwise interchange perform equally well; at higher request rates (more stress), pairwise interchange
performs much better due to its almost exhaustive search for the optimum schedule. However, pairwise interchange takes significantly longer than gi-FIFO to execute and so when the system is less stressed (with comparatively lower rate of requests and more relaxed SLA constraints), gi-FIFO is as effective as pairwise interchange.

Fig. A.12 (obtained with the same topology configuration as in experiment one) depicts a typical per class behavior for all three algorithms: no algorithm meets all per class requirements and no algorithm is a consistent winner, on a per class basis. In last insertion, the penalty only depends on the position of the newly arrived request in the schedule and not on finding the best overall schedule as in the case of pairwise interchange.

Also shown in Fig. A.12 is that gi-FIFO exceeds pairwise interchange for some classes; however, these classes have a low penalty and as can be seen from Fig. A.12, the gi-FIFO policy does not adapt to the required percentile SLA as well as pairwise interchange causing the total penalty incurred in gi-FIFO to be higher as shown in Fig. A.11. Also shown in Fig. A.12 is that Tabu search with pairwise interchange performs as well as simulated annealing with pairwise interchange gaining on the latter for some classes. This is expected as in Tabu search we can expect to obtain a number of schedules more than that in simulated annealing owing to the tabu list where a configurable past number of schedules are stored and each new schedule is checked against that list and if there is any match, the schedule is discarded and another schedule is obtained in its place.
We also compare the penalties obtained with simulated annealing and tabu search both employing pairwise interchange with varying input rates in Fig. A.13. As shown, Tabu search and simulated annealing perform well for almost all of the input rates, with Tabu performing a shade better in most cases. However in a few cases the penalty incurred in Tabu search is slightly less than that in simulated annealing and we attribute this to our implementation of dynamic list sizes, where, if requests queued are very less in number, we do not perform comparison with the tabu list in tabu search and so, the variation is due to the randomness in the input rates in the two simulation runs.

How do the neighborhood search variants of the algorithm perform with varying iterations?

With simulations we have found that pairwise interchange with Tabu search or simulated annealing performs the best among all other variants. The pairwise interchange algorithm finds the optimum schedules over a set number of iterations.

We evaluated pairwise interchange based Tabu search with a set number of iterations for two different request rates, results in Fig. A.14. As expected, as the number of iterations increase,
the total penalty decreases, also the minimum iterations required for the total penalty to be zero is higher for higher input rate.

A.5 Conclusion

In this work, we studied the problem of request scheduling in a cloud computing system with geographically distributed data centers hosting multiple applications; the system operates under a global, percentile response time SLA. The SLA calls for economic penalties if percentile targets are not met. We proposed a distributed request scheduling scheme that aims to minimize the total penalty charged on the cloud. We implemented and evaluated two variants of a heuristic algorithm, one based on simulated annealing and another on gi-FIFO scheduling. Our evaluation has shown that our methods far outperform the commonly deployed FIFO scheduling. We also showed that simulated annealing/tabu search variant outperforms the gi-FIFO variant of the heuristic. However, simulated annealing is compute and time intensive and can be used as an “upper bound” in comparing scheduling techniques.
Figure A.10: Conformance calculated by datacenter with low resources before and after adaptation phase.

Figure A.11: Comparison of gi-FIFO and pairwise interchange under high stress and stringent conformance values.
Figure A.12: Comparison of simulated annealing based pairwise interchange, last insertion and gi-FIFO.

Figure A.13: Comparison of total penalty in simulated annealing based pairwise interchange and tabu search based pairwise interchange.
Figure A.14: Total penalty obtained in tabu search pairwise interchange with varying iterations for two input request rates.