Towards Multivariable Architecture for SaaS Multi-tenant Applications

M. D. Samrajesh\textsuperscript{1}\textsuperscript{*} and N.P. Gopalan\textsuperscript{2}

\textsuperscript{1}Research and Development Centre, Bharathiar University, Coimbatore, Tamil Nadu, 641046, India
\textsuperscript{2}National Institute of Technology, Tiruchirappalli, Tamil Nadu, 620015, India
\textsuperscript{1}samrajesh.md@gmail.com, \textsuperscript{2}npgopalan@nitt.edu

Abstract

A multi-tenant Software as a Service (SaaS) application delivers customized run-time application to each tenant using a single application. Multi-tenancy offers several advantages including quick application deployment, reduced application maintenance, effective version control, and much more. However, handling large datacenter resources and providing better Quality of Service (QoS) is a major challenge for SaaS providers. SaaS application have unstable load based on tenant user’s demand. In a typical multitenant application security, reliability and energy efficiency play an important role. However, these aspects are largely ignored while focusing on the performance of the application. In this paper we propose a novel application’s component based multivariable architecture for multi-tenant Software as a Service (SaaS) application. Instances of application’s components are created at component level and dynamically scaled based on the component’s load considering security, reliability and energy based runtime factors applying multi-criteria Zions–Wallenius optimization method. Our evaluation and discussions show that the proposed multi-instance Towards Multivariable Architecture (TMA) offers enhanced application load distribution and at the same time offers improved application response time and conserves energy in datacenter.

Keywords: Auto-scaling, Cloud Computing, Security, Green Computing, Reliability, SaaS, Multi-tenancy, Software Model

1. Introduction

Today, system resources including software applications can be acquired on demand using cloud computing [1, 2]. Multi-tenancy is a key feature of cloud computing where a single application is shared among multiple tenants. Software as a Service (SaaS) using multi-tendency delivers customized run-time application to individual tenants using a single application [3]. Multi-tenancy offers several advantages including reduction in maintenance cost, effective application version control, rapid application customization, quick deployment and much more. Multi-tenant applications (MTA) have significant competitive advantages over classical single-tenant applications [4]. However, handling large datacenter resources and providing better Quality of Service (QoS) is a major challenge for SaaS applications [5].

SaaS applications have unstable load based on tenant users count, user access frequency. Moreover in a typical application security, reliability and energy efficiency play an important role which are largely ignored while focusing on the performance aspect of the application. Multi-tenant application can be designed using a single instance, however scalability of the application is restricted [6] and it is complex [7]. Having

\textsuperscript{*}Corresponding Author
separate instances for each tenant is not economically feasible [6, 8]. Multi-tenant architectures presented by [9, 10] are specific to a platform. In this paper we propose a novel application’s component based Towards Multivariable Architecture (TMA) for multi-tenant Software as a Service (SaaS) applications. Instances of application are created at component level and dynamically scaled based on the component’s load considering security, reliability and energy based runtime factors and applying multi-criteria Zionts–Wallenius optimization method. Generally, the method is based on user defined preferences and input at each iteration the method finds feasible solutions and finally attempts to get a near optimal solution. A component in MTA is defined as a major module or service in the application that is created while designing the application. A component instance can run as a virtual machine [11]. The proposed architecture can be incorporated in OpenNebula by updating its scheduler.

Our evaluation and discussions show that the proposed multi-instance TMA offers enhanced application load distribution, offers improved application response time and reduces energy consumption at the datacenter. The rest of the paper is organized as follows; Section 2 presents the related work in multitenant SaaS application. Section 3 presents the proposed multivariable architecture. Section 4 has the evaluation and discussions and, finally, our conclusion and future work is presented in Section 5.

2. Related Work

A framework for implementing multi-tenant applications’ is proposed in [12]. The framework defined configurations for security, performance, and administration for application. Moreover, it labeled the performance blocks and optimization methods. Resource distribution is centered on Service Level Agreement (SLA). Monitoring of a multi-tenant single-instance application is presented in [13]. The model has a performance monitor, an element to identify unfamiliar conditions in the system and a scheduling module to schedule tenants’ user access on pooled resources. The tenants are ordered based on tenants’ SLA. Higher priority tenant users have early access to shared resources.

A secure and load disseminated architecture for multi-tenant applications is proposed in [14]. The architecture dispenses authentication, authorization and validation processes to different layers. The distribution of load in term is said to enhance the performance and improve the user experience by meeting tenant specific SLAs. Responder services sensed the kind of service and transfers it to successive service levels. A flexible database schema based view on multi-tenant architecture is presented in [15]. The database tables are vertically divided into chunks which are folded together into diverse physical multi-tenant tables and combined. Application performance is achieved by mapping the heavily used portions of the logical schemas into the conventional tables and the balance as chunk tables. The evaluation results showed the effectiveness of chunk folding.

A framework for intuitive multi-tenant application development and administration is presented in [16]. The framework is based on service oriented architecture; it exploits the multi-tenancy enablement services such as isolation and tenant customization. Moreover, it enables application developers to focus on business logic layer of their application rather than basic multitenant aspects. The evaluation showed that the proposed framework offers easier maintenance of application. A monitoring architecture based on scalable paradigm for cloud is presented in [17]. The proposed architecture incorporates an effective cloud monitoring of tenant data as per explicit service and tenant necessities and uses standard communication protocols for communication in the multi-tenant cloud. The architecture is based on open-source tool OpenStack. The architecture is compared with other methods for performance and scalability; the results showed it incurs lesser overhead.
A novel monitoring of cloud resources and services is presented in [18]. In the proposed architecture the resource provider can see a full overview of the infrastructure deployed. The cloud consumer can view their resources using adaptive capabilities, configure and customize the monitoring mechanism of their resources. The data examined by the cloud consumer is private. Moreover, it supports disaster recovery by synchronizing information for better retrieval in case of a failure. The evaluation results showed that the proposed method offers better scalability and with lower overhead.

An efficient customization for multi-tenant SaaS applications using service lines is presented in [19]. The proposed method includes design, implementation, functioning and maintenance of a SaaS application. Moreover, the service line includes SLAs and versioning, mapping specifications and instantiation of service line. The proposed method is evaluated using a document processing application and also compared for cost assessment with existing methods. A framework for QoS-aware and ontology-based service deployment across clouds is presented in [20]. However, the energy consumption of the servers is not considered while it’s deployed.

The surge in cloud computing services, have led to an exponentially growth of datacenters [21]. Today, the datacenter itself is called as a computer. Information and Communications Technology (ICT) devices energy consumption is on rise and by 2020 ICT devices are projected to consume about 14.5% of worldwide energy requirement [22]. There is a demand from governments across the globe to reduce carbon emission, due to its significant impact on climate change [23, 24]. Moreover, growing complexity of cloud based interconnected system’s has become too complex for managing. A holistic method that considers performance, reliability, security and Quality of Service (QoS) without manual intervention is required [25]. Application scaling in cloud should take into account the different runtime aspects for better QoS [26]. Scaling application efficiently in cloud is a challenge [27]. There is a lack of approaches for optimizing cloud service provisioning and application deployment, and also satisfy consumers’ QoS expectations [28].

Our proposed architecture is an application’s component based approach for multi-tenant Software as a Service (SaaS) applications. Instances of application are created at component level based on various runtime factors such as to minimize the application response time, minimize tenants security risk, minimize application’s reliability risk and minimize datacenter’s energy requirement.

3. Towards Multivariable Architecture (TMA)

We present our proposed architecture that consists of system model, architecture and algorithms. In the following sub-sections we describe the system model, the variables used, the generalized notation of the multiple objective optimization problem, the proposed architecture and the algorithms used in the implementation of the architecture.

3.1. System Model

An application consists of ‘n’ components core (components that are fundamental for the application) and add-on (based on tenant nature of business the components vary) and each component may have \( m_i \{ 0 \leq i \leq n \} \) number of instances based on the factors defined. At any given point of time ‘t’ the Component Instance (CI) of a component ‘i’ is described as ‘\( m_i \)’ \{ 0 \leq i \leq n \}.

\[
\begin{align*}
    n & \quad \text{The number of components in the application} \\
    C_i & \quad \text{Component ‘i’ of the application} \{ 0 \leq i \leq n \} \\
    m_i & \quad \text{No of instances of component ‘i’} \{ 0 \leq i \leq n \}
\end{align*}
\]
CI

Component Instance of component ‘i’, instance ‘j’ {0 ≤ i ≤ n, 0 ≤ j ≤ m}

CS

Security flag Level – I for component ‘i’ of the application {0 ≤ i ≤ n}

CIS

Security flag Level – II for component ‘i’ and instance ‘j’ {0 ≤ i ≤ n, 0 ≤ j ≤ m}

CRR

Component instance’s resource requirement

k

The number of tenants attached to the application

Ti

Tenant ‘i’ of the multi-tenant application {0 ≤ i ≤ k}

ui

Number of active users attached to the tenant Ti {0 ≤ i ≤ k}

TC

Tenant ‘i’ user ‘j’ current number of components accessed

CAF

Component ‘i’ instance ‘j’ user Access Frequency. {0 ≤ i ≤ n, 0 ≤ j ≤ m}

CRL

Component ‘i’ instance ‘j’ Runtime load {1 ≤ i ≤ n, 0 ≤ j ≤ m}

ART

Application Response Time ‘i’ Tenant ‘j’ user {1 ≤ i ≤ k, 0 ≤ j ≤ ui}

p

Number of servers at the datacenter

SC

Server Resource capacity ‘i’ {0 ≤ i ≤ p}

SSR

Server Security Risk ‘i’ {0 ≤ i ≤ p}

SRR

Server Reliability Risk ‘i’ {0 ≤ i ≤ p}

SER

Maximum Energy requirement for Server ‘i’ {0 ≤ i ≤ p}

SL

Server Load ‘i’ {0 ≤ i ≤ p}

CuT

Component up Threshold

CdT

Component down Threshold

CHT

Component hit Threshold

CH

Component Hit count {0 ≤ i ≤ n, 0 ≤ j ≤ mi}

The SaaS multi-tenant application consists of ‘n’ number of components in the application. Each component ‘Ci’ has ‘mi’ component instances represented as CI. Figure 1 shows the illustrative diagram of the system model. There are ‘k’ number of tenants attached to the application and ‘ui’ number of active users attached to each tenant Ti. Runtime factors of the user access is recorded in the following the user count in ‘TCij’, the user access frequency ‘CAFij’. The number of servers at the datacenter is represented as ‘p’, each server’s capacity is represented by ‘SCi’ server resource capacity. The security, reliability, energy and load of the server are represented by SSRi, SRRi, SERi and SLi respectively. The proposed architecture has 2 layers of security one at component level and another at component instance level termed as CS, CISij respectively. The various threshold used in the architecture are CuT Component up Threshold is used start additional CI, CdT Component down Threshold is used to scale down CI(s) and power off underutilized servers. The proposed architecture implementation algorithm observes the CI for a period of time and decides to invoke the ActiveScale() algorithm. This is done based on CHT (Component hit Threshold) and counter CHij (Component Hit count).
Figure 1. The Illustrative Diagram of the System Model

Generalized notation of multiple objective minimization problem

\[
\text{Minimize } z = f(x) = (f_1(x), \ldots, f_{n-1}(x), f_n(x))
\]

subject to

\[
r \in \mathbb{R} \text{ constraints}
= \{ f(x) : x \in X, X \subseteq \mathbb{R}^n \}
\]

where \( X \) is the feasible set and ‘\( x \)’ is the decision variable vector of size ‘\( n \)’

- **Minimize Response Time**
  \[
  f_1(x) = \text{Minimize } \sum_{i} \sum_{j} ART \left( u_{ij} \right)
  \]
  \( \{ 1 \leq i \leq k, 0 \leq j \leq u_i \} \)

- **Minimize Security Risk**
  \[
  f_2(x) = \text{Minimize } \sum_{i} SSR(i).y_i
  \]
  \( \{ 0 \leq i \leq p \} \)

- **Minimize Reliability Risk**
  \[
  f_3(x) = \text{Minimize } \sum_{i} SRR(i).y_i
  \]
  \( \{ 0 \leq i \leq p \} \)
Minimize Energy Requirement

\[ f_d(x) = \text{Minimize} \sum_{i}^p \text{SER}(i) \cdot y_i \]
\[ p \{ 0 \leq i \leq p \} \]

\[ y_i = \begin{cases} 1 & \text{if server 'i' is powered on} \\ 0 & \text{otherwise} \end{cases} \]

\[ \text{Minimize } z = f(x) = (f_1(x), f_2(x), f_3(x), f_4(x)) \]

In the above multi-objective optimization, a feasible solution that minimizes the entire stated objective functions simultaneously typically does not exist. Hence, considering multiple variables, we incorporate Zionts–Wallenius method, it is an interactive method used to find a best available solution in a multi-criteria optimization problem.

\[ \text{Min } \sum_{i} v_i \cdot f_i(x) \]
\[ \sum_{i} v_i = 1 \{ v_i > 0, i=1,2...,n \} \]

\[ x \in X, \text{ specifically, if the vector } V_i > 0, i=1,2...,n \text{ then the minimization is a strict Pareto optimum, whereas if vector } V_i = 0, i=1,2...,n \text{ then it is a weak Pareto optimum.} \]

**Constraints**

1. Server load lower constraint at server ‘i’

\[ \sum_{i}^{n} CRL_i \cdot X_i > CuT \quad \forall j \]

\[ X_i = \begin{cases} 1 & \text{if CI 'ij' is connected to atleast one user and resides in server 'i'} \\ 0 & \text{otherwise} \end{cases} \]

The load in the server is directly related to the number of CI(s) placed in it. The server load is above CuT. In case the load is less the underutilized servers CI’s are migrated and the server is powered off to conserve energy.

2. Component instance resource requirement constraint

\[ \sum_{i}^{n} CRR_i \cdot X_i \leq SCI \quad \forall j \]

\[ X_i = \begin{cases} 1 & \text{if CI 'j' is placed in Server 'i'} \\ 0 & \text{otherwise} \end{cases} \]

The component instance resource requirement is less than the physical capacity of the residing server.

3. Server load upper constraint at server ‘i’

\[ \sum_{i}^{n} CRL_i \cdot X_i < CIT \quad \forall j \]

\[ X_i = \begin{cases} 1 & \text{if CI 'ij' is connected to atleast one user and resides in server 'i'} \\ 0 & \text{otherwise} \end{cases} \]
The load of any server is less than Component lower Threshold ($C_{\text{LT}}$); else present a suitable server for relocating the CI based on TsT, selected CI(s) of the server are migrated to a target server. Moreover, in case all the servers are above WoT, new servers are powered on to balance the datacenter servers’ load.

4. Server energy consumption constraint

\[ SER_i \geq 0 \]
\[ 0 \leq i \leq n, \quad 0 \leq i \leq n, \]

The server energy consumption of servers is above 0

5. CI’s placement Constraint

\[ \sum X_{ij} = 1 \quad \forall i \]

A CI is placed and resides in only one server at a time. When a CI is relocated and placed in a target server $X_{ij} = 1$ the victim server’s value is set to 0, hence the summation is always 1.

6. Security Risk Constraint

\[ SRS_i \geq 1 \]
\[ 0 \leq i \leq n, \quad 0 \leq i \leq n, \]

At least at level 1

### 3.2. Multivariable Architecture

The proposed novel architecture is comprehensive, it considers the resources availability at datacenter (processing power, network bandwidth, memory, energy, reliability), tenant user(s) demand (static, dynamic) and application’s requirement (security, performance requirement). The multi objective criterion considers wide range of factors. We incorporate Zionts–Wallenius method; based on the user defined preferences at each iteration it finds the best available solution for the multi-criteria optimization problem by selecting the best available server for placing application component instance.

The proposed architecture offers fine grained component level scaling based on component’s load. It provides two level of security for the application access, when the load at a component instance (CI) increases based on the tenant user request and various other runtime factors such as access frequency. As shown in algorithm 1, ActiveScale() is invoked to create additional instances of the component instance such as to ensure that the QoS is not degraded. The algorithm removes instances when the load decreases by consolidating CIs. Moreover, the proposed algorithm tries to minimize the number of servers, by migrating CI(s) from under-loaded servers. Migration is done such as the number of powered-on servers is kept at the minimum. Additionally, migration of CI is done based on the reliability and security of the server and the tenant requirement. This helps application providers reduce energy cost at the same time enhance QoS. The proposed architecture while switching on and off target servers considers the HEF (Highest Energy First) and LEF (Lowest Energy First) respectively hence the energy saved under heterogeneous...
server capacity is significant. The conservation in energy also leads to reduction in carbon emission and substantially contributes to green computing.

The proposed architecture considers various runtime factors including the user access frequency and user access pattern and when users of similar components access during runtime, grouping together and consolidating instances enhances the QoS and minimizes the number of servers. Most available system ignores the relationship in user access patterns. The security and reliability of the application is enhanced by running CI(s) only in servers that has the lower security and reliability risk. The Risk factor of the server is periodically evaluated and the CI(s) are place only on servers that have risk factor SSRi and SRRi minimum. We use a multi-variable defensive criterion \((X_i) \{ 0 \leq i \leq 8 \}\) to determine the security and reliability of the server.

The following criterion are used \((X_i)\) X1-Unsafe Identify Management, X2-Absence of Tenant application monitoring X3-Absence of finer Granularity of application and data access, X4-Absence of Tenant data isolation, X5-Absence of Secure Network access, X6-Insecure Backups storage, X7-Absence of High availability, X8-Absence of Data recovery mechanism. The risk classification is done based on [8] in 3 categories of risks labeled a) Low b) Medium, and c) High.

Reliability of the server is computed using three core factors of the datacenter servers 1) Energy Reliability \((\lambda_1)\) Reliability characteristics include redundancy of power source in the server’s datacenter, availability of highly-efficient UPS (Uninterrupted Power Supply), and efficient datacenter cooling management system. 2) Server Hardware Reliability \((\lambda_2)\) reliable hardware is crucial for the application’s availability. Health of critical devices such as hard disks, capacity and failure of CPU(s), and primary memory are considered. 3) Network Reliability \((\lambda_3)\) includes redundant path, communication reliability, and bandwidth [28].

---

**Algorithm 1. Active Scale Implementation**

```
Algorithm 1. ActiveScale()

Input : Component load, Server Load, Hit Count
Output : Scale application Server Selection and CI’s to Migrates
Trigger : Change in Resources Availability, Tenant
User Load changes

1. Begin
   // Observe Application Components Instances (CI)
2. For (for all Application Component Instances) do
3. Find CI \exists CRLij > CuT \& CHj < CHT // Scale Up
   // Search for components load greater than threshold and
   // # of times it crosses the threshold
4. Create new instance of component ‘ i’
5. Call PickUpServer (Scale up)
6. Start-up new Component instance at ‘k’
7. Map future user access to new Component
8. Find CI \exists CRLij < CdT \& CHj < CHT // Scale Down
   // Search for instance > CdT \& < CHT , Choose target CI(s)
9. Call PickUpServer (Scale Down)
10. If ( Designated Server count(CI) = \phi)
11. Switch off the Designated Server
12. End
```
Algorithm 2. PickUpServer (var)

Output : Server Selection for Component creation/releases
Input : Server Load, Energy Requirement, Scale Up/Down
Trigger : Change in Resources Availability, Server Load
1. Begin
2. If (var is Scale up)
3. For all datacenters Server do
4. Find server $\exists$ LRE I, { Sort List of Servers based server }
   // Apply Zionts–Wallenius method
   // (considering Load, Reliability & Security, LEF)
5. Choose the first Server ‘k’ from the list
6. In case of tie choose based on CRR match
   //Create Component Instance
7. Else if (var is Scale Down)
8. Consolidate underutilized CI such as to switch off a server
   with HEF (Highest Energy First) // Apply Zionts–Wallenius method
9. Live migrate all users of CI(s) to existing target CI(s)
10. End

Algorithm 2. Pickup Server Implementation

4. Evaluation and Discussion

The proposed model is evaluated using custom built simulator as in [29] [30] and using automation tool Arena [31]. The workload configuration consists of steady, increasing, spike and on-off as in [32].

4.1 Experimental Setup

The initial total number of tenants is 1000. The number of tenant users attached to each tenant follows a Poisson distribution. The add-on components accessed by tenant users are randomly distributed. We generate risk factors according to probability distribution model. The risk levels are classified based on the security and reliability strength within a specific timeframe. Risks at each server are quantitatively measured in scale between 1 and 3.

We model the system using application level traces as in [33] which contain traces of real world HTTP request. Weibull distribution widely used in reliability engineering and failure analysis [34] is used to generate numerical simulation values. Moreover, in order to perform a realistic evaluation, we use existing TPCW benchmark with multi-tenancy support (MTTPC-W) [35]. MTTPC-W is an online book store application, the workload is generated when tenant users access web, to support multi-tenancy, and it differentiates between requests originating from different tenants by isolating their data. The tenant user request follows a Markov chain. As given in Table 1, in our experiments we use 100 servers, 2000 tenant and each tenant having varied number of users who simultaneously access the application. The MTA consists of 5 core components and 10 add-on components, initially single instance of all core components are created. Later based on demand further instances of the core and add-on components are created.

Table 1. Evaluation Setup Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of Servers</td>
<td>100</td>
</tr>
<tr>
<td>Tenants</td>
<td>2000</td>
</tr>
<tr>
<td>Number of Core Components</td>
<td>5</td>
</tr>
<tr>
<td>Add on Components</td>
<td>10</td>
</tr>
<tr>
<td>Tenant Users</td>
<td>250-1000</td>
</tr>
</tbody>
</table>
We evaluate the proposed architecture using 3 types of servers 1) Small Scale Deployment (SSD), 2) Medium Scale Deployment (MSD), and 3) Large Scale Deployment (LSD) as shown in Table 2. The datacenter servers load at time ’t’ is shown in Figure 2. The thresholds are set based on [36, 37].

<table>
<thead>
<tr>
<th></th>
<th>SSD</th>
<th>MSD</th>
<th>LSD</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bandwidth mbps</td>
<td>2 mbps</td>
<td>10 mbps</td>
<td>Unlimited</td>
</tr>
<tr>
<td>Number of Servers</td>
<td>50</td>
<td>30</td>
<td>20</td>
</tr>
<tr>
<td>Energy (watts)</td>
<td>218</td>
<td>638</td>
<td>12682</td>
</tr>
</tbody>
</table>

When the application CI(s) load increases TMA creates additional CI(s) to handle the additional workload and guarantees the user QoS. Additionally, when the load decrease it removes specific CI(s) or migrates CI(s) and switches off servers that are underutilized this reduces energy consumption and increases the reliability of the server.

![Figure 2. Typical Server Load at Datacenter](image)

### 4.2. Comparative Evaluation

We evaluate the average Tenant user Response Time (TRT) under two different scenarios namely the static component access and dynamic component access. We compare our proposed architecture with two existing application instance scaling models 1) Time based Scaling (TBS) as in [38] here application components are scaled based on a planned schedule set by the providers. The highest application load time is selected based on history of earlier logs and traces and based on the type of application 2) Hierarchical Based Scaling (HBS) as in [39], here components are scaled based on the applications core components and related components based on the application’s hierarchy. Each application can also have one or more application component instances considering its level in the hierarchy.

Our evaluation results in Figure 3 shows the average tenant response time under static component access (i.e., the number of attached users to a tenant is constant) and dynamic component access (i.e., the number of attached users to a tenant continuously vary). The average tenant response time under static component access at 500 tenant users TMA when compared with TBS, HCS has 10.52%, 21.05% less average tenant response time respectively. TMA was performed well as TMA creates additional CI based on the load and the load at the server is balanced. However at dynamic component access as shown in Figure 3 (b) at 1000 tenant user we see a convergence in response time and the
comparative advantage in response time is less since the data center resources are utilized to the maximum.

A typical datacenter has different energy consuming servers that are heterogeneous. We compare our proposed architecture with respect to energy saving against 1) Non Power-Aware (NPA) as stated in [40] and 2) Dynamic Voltage and Frequency Scaling (DVFS) as in [41]. The number of powered off and the normalized energy saved under homogenous datacenter server capacity and heterogeneous server capacity datacenter is evaluated.

Our experimental results in figure 4 shows that proposed TMA outperforms DVFS by conserving more energy under homogenous datacenter server capacity and heterogeneous datacenter server capacity by 14.28%, 37.12% respectively. The proposed architecture while switching on and off target servers considers the HEF (Highest Energy First) and LEF (Lowest Energy First) respectively hence the energy saved under heterogeneous server capacity is significant. The conservation in energy also leads to reduction in carbon emission and substantially contributes to green computing.

![Figure 3](image1.png)

**Figure 3. Average Tenant Response Time Under (a) Static Component Access (b) Dynamic Component Access for TBS-Time based Scaling, HBS-Hierarchical Based Scaling and TMA- Towards Multivariable Architecture**

![Figure 4](image2.png)

**Figure 4. (a) Homogenous Datacenter Server Capacity (b) Heterogeneous Datacenter Server Capacity. NPA: Non Power-Aware, DVFS : Dynamic Voltage and Frequency Scaling**
5. Conclusion and Future Work

We presented a novel Towards Multivariable Architecture (TMA) along with implementation algorithms for multi-tenant SaaS application. The proposed architecture is an inclusive architecture that considers most aspects for the SaaS application. Instances of application is created at component level and dynamically scaled based on runtime application factors and datacenter resources factors including server security, reliability and energy. We incorporate Zions–Wallenius method in our proposed architecture which attempts to find the best available server for application component instance.

We compared our proposed architecture for energy saving against Non Power-Aware (NPA) and Dynamic Voltage and Frequency Scaling (DVFS). The tenant users’ response time is evaluated in two different scenarios the static component access and dynamic component access. Our evaluation under different scenarios and methods demonstrates that the proposed architecture is effective in enhancing the QoS of multitenant application. The average tenant response time under static component access at 500 tenant users TMA when compared with TBS, HCS has 10.52%, 21.05% less average tenant user response time respectively. TMA outperforms DVFS in conserving energy under homogenous datacenter server capacity and heterogeneous datacenter server capacity by 14.28%, 37.12% respectively. As a part of future work we plan to include more factors in the evaluation of the architecture and use diverse scenarios for testing.

References


Authors

M. D. Samrajesh, received MCA from Bharathidasan University. He got his M.Phil in the field of Computer Science from MS University. He is currently a Research scholar at Research and Development Centre, Bharathiar University, Tamil Nadu. He is a member of IACSIT, IAENG, CSI. His research interest includes Cloud Computing, Software Engineering, Distributed Systems, Green Computing, and VOD.

N. P. Gopalan, received his M.Sc degree Mathematics from Madras University in 1978, PhD in Applied Mathematics from Indian Institute of Science, Bangalore in 1983 and M.E degree in Computer Science and Engineering from NIT, Tiruchirappalli in 1993. He is currently serving as the Professor of Computer Applications in it. He has nearly 60 publications in various national and international Journals. He has authored several text books and a patent in the design of wireless robot. His areas of interest include Algorithms, Combinatorics, Data Mining, Distributed and Mobile Computing.